Warsaw University of Technology

algorithms and computability

Multi-graphs Project

Authors:
Świstak Jakub, Hamed Rayan, Kołodziejczyk Filip

January 4, 2024

Contents

1 ABSTRACT

This project aims to find and implement algorithms for finding the maximum clique and the maximum common sub-graph of two multi-graphs and performing computational tests with time characteristics on the algorithms implemented. The implementation of the algorithms is written in C++.

2 DEFINITIONS

Graph Size The definition of graph size is a pair of 2 numbers (vertices and edges) sorted by the number of vertices and then by the number of edges.

Metric in the Set of All Graphs The Distance between two graphs is a pair of two numbers, vertices, and edges sorted by the number of vertices and in the case of equality by the number of edges. When constructing this metric we were based on the Graph Edit Distance (GED), we are taking into account unlabeled graphs where the value is the number of operations (edge/node insertions, deletions) needed to transform one graph into the other. The distance can be derived from LCS since a larger LCS corresponds to a smaller distance.

K-Clique A K-Clique in a graph is a clique that contains a K number of edges between all pairs of vertices.

Maximal Clique The maximal clique is a clique in which the addition of any adjacent vertex does not result in a larger clique, meaning it cannot be contained within a larger clique.

Maximal Common Subgraph The Maximum Common Subgraph (MCS) is the largest graph which is simultaneously isomorphic to two sub-graphs of two given graphs.

L-Connectivity There is a connection between 2 vertices if there are at least L edges between these vertices.

3 MAXIMAL CLIQUE

3.1 Description of Algorithm

Various algorithms can be used and implemented but the Bron–Kerbosch algorithm was chosen here to solve the maximal clique problem. The algorithm will help us to find K-Cliques defined above. The Bron-Kerbosch recursive backtracking algorithm is an enumeration method used to identify every maximal clique within an undirected graph. In other words, it systematically generates a list of vertex subsets with two specific characteristics: first, every pair of vertices within a listed subset is connected by an edge, and second, no additional vertices can be included in any of these subsets without compromising their full connectivity. The most basic form of the algorithm does not require neither pivoting nor vertex sorting but later on variants used those 2 methods which saved time and allowed for quicker backtracking in branches of the search that contain no maximal cliques.

The algorithm maintains three disjoint sets of nodes R, P, and X. Set R stands for the currently growing clique; set P stands for prospective nodes that are connected to all nodes in R and using which R can be expanded; set X contains nodes already processed i.e. nodes which were previously in P and hence all maximal cliques containing them have already been reported. An important invariant is that all nodes that are connected to every node of R are either in P or X. The Purpose of X is to avoid reporting the same maximal clique multiple times, through the update $P = P - u_i$. To avoid reporting cliques that are not maximal, the algorithm checks whether set X is empty, if X is non-empty, the nodes in X may be added to R, but this would yield previously found maximal cliques.

```
BronKerbosch1(R, P, X):
        if P and X are both empty:
            report R as a maximal clique
        for each vertex v in P:
            BronKerbosch1(R ∪ {v}, P ∩ N(v), X ∩ N(v))
            P := P \ {v}
            X := X ∪ {v}
```

Listing 1: Bron-Kerbosch Pseudocode

In the worst case, the original Bron-Kerbosch algorithm has an exponential time complexity, specifically $O(3^{n/3})$, where n is the number of vertices in the graph which is a result of the recursive nature of the algorithm.

Complexity analysis The Bron-Kerbosch algorithm exhibits exponential time complexity, specifically $O(3^{n/3})$, in the worst-case scenario primarily due to its recursive approach in exploring the cliques. This complexity arises as follows: In the worst case, every recursive call of the algorithm potentially divides the problem into smaller sub-problems, each requiring its own recursive call. These calls form a recursion tree.

At each step, the algorithm chooses a vertex v from set P and makes two recursive calls - one with v added to the current clique R, and another with v removed from future consideration. In the worst case, the size of P reduces by only one vertex in each step, leading to a maximum recursion depth of n (the number of vertices).

Each node in the recursion tree has a number of children equal to the size of P in that recursive call. In the worst case, this branching factor can be as large as the number of vertices in the graph, leading to an exponential growth in the number of nodes in the recursion tree.

The $3^{n/3}$ complexity specifically comes from an analysis of the recursion tree's size. It has been shown that for any n-vertex graph, the number of maximal cliques (which corresponds to the number of leaves in the recursion tree) is bounded above by $3^{n/3}$. This is because each maximal clique can be uniquely identified by a sequence of at most $n/3$ choices, where each choice is among three options: adding a vertex to R, moving a vertex from P to X, or doing nothing (which happens when a vertex in P is adjacent to all vertices in R). This gives rise to the upper bound of $3^{n/3}$ on the total number of possible maximal cliques, and consequently, on the number of leaves in the recursion tree of the algorithm, leading to its exponential time complexity.

3.2 Approximation Algorithm

Since the time complexity of the algorithm above is in exponential complexity we ought to implement an approximation algorithm with polynomial complexity. The Monte Carlo approximation can be used to compute the maximal clique in a multi-graph with polynomial complexity. The Monte Carlo approximation is a randomized algorithm that is used to find approximate solutions to the clique problem. It doesn't guarantee the exact solution but provides an approximate solution with a certain probability. It works by iteratively selecting vertices and attempting to form a clique of a specified size.

1. The algorithm initializes an empty set to store the vertices forming the potential clique.

2. It randomly shuffles the order of vertices in the graph to introduce randomness in the selection process.

3. It iterates through the randomized order of vertices and attempts to build a clique. For each vertex, the algorithm decides whether to add it to the clique or not.

4. If the current size of the potential clique is less than the found clique size, it replaces the current biggest clique with the new biggest clique candidate

5. The algorithm returns the set of vertices obtained as the approximate clique at the end of the process.

Complexity Analysis The complexity of the Monte Carlo approximation algorithm is done in polynomial time, because:

- The initialization of the algorithm is independent of the input and is constant.

- The number of iteration is constant.
- In a $O(n^2)$ time we can check whether the set of vertices form a clique or not.
- Updating the largest clique can be done in linear time.

Summing these complexities we can see that the Monte Carlo approximation algorithm for finding a maximal clique runs in polynomial time. This makes it significantly more efficient than an exponential time algorithm, especially for large graphs. However, it is important to note that this efficiency comes at the cost of accuracy (described as a constant k), as the algorithm provides only an approximation of the maximal clique.

3.3 Computational Tests with Time Characteristics

K_CLIQUE	**Graph Size**	**BronKerbosch Time**	**MonteCarlo Time**
1	3	103	1984
	4	181	3168
	5	102	4249
	10	11982	15790
2	3	99	1952
	4	145	3074
	5	101	4531
	10	12033	15855
5	3	99	1966
	4	184	3038
	5	104	4267
	10	12027	15720
10	3	101	1970
	4	153	3049
	5	101	4244
	10	11951	16067
15	3	98	1984
	4	148	3014
	5	102	4295
	10	11902	16073

4 MAXIMAL COMMON SUBGRAPH

In the maximum common subgraph (MCS) problem, we are given a pair of graphs and asked to find the largest induced subgraph common to them both. With its plethora of applications, MCS is a familiar and challenging problem. Many algorithms exist that can deliver optimal MCS solutions, but whose asymptotic worst-case run times fail to do better than mere brute force.

4.1 Description of Algorithm

The maximal common subgraph problem is an NP-hard problem, a brute force algorithm was used to find the maximal common subgraph of 2 graphs. It recursively finds all possible subsets and checks if it is a feasible solution. The algorithm goes through all the vertices until the set is empty and returns. Once the set of vertices is empty it appends each vertex to the current subset and checks if it is a common subgraph between the graphs.

The provided algorithm below is a brute-force approach that employs backtracking. It explores all possible combinations of vertex mappings between the two graphs, making use of recursion and backtracking to efficiently navigate the solution space. The key insight is that it systematically attempts to map vertices, exploring all possibilities, and backtracks when necessary to explore alternative mappings.

The use of backtracking is evident in the recursive function maximalCommonSubgraph-Process. This function explores the neighborhoods of vertices and attempts to map them, but it also backtracks by undoing mappings and removing vertices from the sets mapped-Vertices1 and mappedVertices2 when needed. This backtracking mechanism allows the algorithm to explore different paths in the solution space, ensuring that it exhaustively considers all potential mappings.

In summary, the algorithm uses a brute-force strategy combined with backtracking to find the maximal common subgraph between two graphs. While it may not be the most efficient approach for large graphs, it guarantees a comprehensive exploration of the solution space, providing a correct solution to the problem.

```
MCSProcess(graph1, graph2, edges1, edges2)
    For each element in vertexMap:
        For each neighbor v1 of element in graph1:
            If v1 is already mapped in mappedVertices1:
                continue
            If no edge between element and v1 in graph1:
                continue

            Add v1 to mappedVertices1
            For each neighbor v2 of vertexMap[element] in graph2:
                If v2 is already mapped in mappedVertices2
                    continue
                If no edge between vertexMap[element]:
                    If v2 in graph2:
                        continue

                Add v2 to mappedVertices2
                Set vertexMap[v1] = v2

                Call MSCProcess(graph1, graph2, edges1, edges2)

                Remove vertexMap[v1]
                Remove v2 from mappedVertices2

            Remove v1 from mappedVertices1

    Create empty vector: mapping
    For each pair in vertexMap:
        Add the pair to mapping

    If size(mapping) > size(largestMappings[0]):
        Clear largestMappings
        Add mapping to largestMappings
    Else If size(mapping) == size(largestMappings[0]):
        Add mapping to largestMappings
```

Listing 2: MCSProcess Brute Force Pseudocode

```
MCS(graph1, graph2)
 Initialize empty mapping: vertexMap
 Initialize empty sets: mappedVertices1, mappedVertices2
 Initialize empty vector of mappings: largestMappings

 For each vertex i in graph1:
     For each vertex j in graph2:
         Set vertexMap[i] = j
         Add i to mappedVertices1
         Add j to mappedVertices2

         Create empty vectors: edges1, edges2
         Call MCSProcess(graph1, graph2, edges1, edges2)

         Remove i from mappedVertices1
         Remove j from mappedVertices2
         Remove vertexMap[i]

 Output largestMappings
```

Listing 3: MCS Pseudocode

The algorithm explores all possible subsets of vertices, resulting in an exponential time complexity.

Complexity Analysis The complexity of the brute-force algorithm for the maximal common subgraph (MCS) problem can be proven to be exponential. The core of this algorithm involves exploring all possible subsets of vertices from the given graphs to identify the largest common subgraph. This exhaustive search strategy inherently leads to a time complexity of $O(2^n)$, where n is the number of vertices in the input graphs.

The algorithm operates by attempting to map each vertex of one graph to each vertex of the other graph, exploring all such mappings. For each vertex in the first graph, the algorithm iterates through every vertex in the second graph. During this process, it performs recursive calls to the MCSProcess function, which further explores each possible mapping. The recursive nature of MCSProcess contributes to the exponential growth of the algorithm's runtime, as it generates all combinations of vertex mappings.

Furthermore, for each mapping considered, the algorithm performs additional operations, such as checking for the presence of corresponding edges and updating the sets of mapped vertices.

To sum it all, the algorithm must consider every possible combination of vertices, the number of combinations grows exponentially with the number of vertices in the graphs. This determines exponential complexity of the solution.

4.2 Approximation Algorithm

1. Iterate over all starting mappings

2. Start DFS traversal from each of selected pair of vertices in both input graphs simultaneously.

3. Explore vertices in both graphs using DFS. Using random vertex from the second graph. When a pair of corresponding vertices is encountered (one from each graph), consider them as a potential match. If the vertices match, add them to the current common subgraph.

4. If the current common subgraph cannot be extended to a larger common subgraph, if current subgraph is bigger than the previous max replace it, and backtrack to the previous state and explore other possibilities. Use pruning strategies to eliminate branches in the search tree that cannot lead to a maximal common subgraph.

5. If all candidates are being considered return the biggest result

Complexity Analysis The approximation algorithm for the maximal common subgraph problem demonstrates polynomial complexity, primarily determined by its DFS traversal. The key steps are:

- **Iterating Over Starting Mappings**: With n vertices in each graph, iterating over pairs of vertices leads to $O(n^2)$ complexity.
- **DFS Traversal**: Performing DFS on each graph contributes $O(n^2)$ complexity, as E (number of edges) in dense graphs can approach $O(n^2)$.
- **Vertex Matching and Backtracking**: Matching vertices and the subsequent backtracking within DFS do not exceed the established $O(n^2)$ complexity.
- **Final Output**: The final step of returning the largest common subgraph is a constant-time operation, $O(1)$.

Given that the most computationally intensive step is the DFS traversal with $O(n^2)$, the overall time complexity of the algorithm is polynomial, specifically $O(n^2)$, making it efficient for large graph sizes.

4.3 Computational Tests with Time Characteristics

L_CONN	Graph1 Size	Graph2 Size	MCS Time	MCS (Approximate) Time
1	4	3	1628	775
	5	5	3827	2056
	10	4	22107645	6587
2	4	3	1536	759
	5	5	3822	2182
	10	4	21530118	6306
5	4	3	1568	767
	5	5	3830	2222
	10	4	21467576	6126
10	4	3	1546	766
	5	5	3770	2051
	10	4	21481676	6246
15	4	3	1556	763
	5	5	3801	2177
	10	4	21631596	6227

5 DISTANCE METRIC

The Distance between two graphs is a pair of two numbers, vertices, and edges sorted by the number of vertices and in the case of equality by the number of edges. When constructing this metric we were based on the Graph Edit Distance (GED), we are taking into account unlabeled graphs where the value is the number of operations (edge/node insertions, deletions) needed to transform one graph into the other. The distance can be derived from LCS since a larger LCS corresponds to a smaller distance.

Metric given above satisfies the metric space axioms.

1. Distance between the same graph is 0 because MSC is the same as the following graphs, so we are not adding or removing vertices or edges.

2. Distance between two graphs is always positive since we are only counting the sum of edges/vertices.

3. Distance is symmetric because 2 graphs have the same MCS and we are summing the distinct edges/vertices between two of them from MCS.

4. Triangle inequality holds, it will be the equality only if the third graph with first has the same MCS as first and second, otherwise the MCS of this 3 graphs will be smaller which implies traingle ineqality.

6 INSTRUCTIONS FOR RUNNING PROGRAM

Assumption: All commands have to be run in PowerShell in the directory containing the project (after being extracted from the archive) if not stated otherwise.

This project is written in C++, therefore it requires compilation before it can be executed. To make compilation simple, Makefile was created. Thus, to compile this program run the following snippet:

```
.\source\Makefile.ps1 Build-All
```

It will produce two files in `exe\` directory:

- `main.exe` - core application which looks for maximal cliques and maximal common subgraphs in provided input. It takes input from files provided as arguments (1 or 2),
- `generator.exe` - utility application which generates sample graphs. The output is an adjacency matrix of the generated graph, which is preceded by the number indicating the size of the graph. It can be used for the generation of input for the core application.

To run `exe\main.exe` execute:

```
.\exe\main.exe <path to input file 1> [<path to input file 2]  | Out-Host
```

By default, `exe\main.exe` searches for maximal 3-cliques and maximal common subgraphs with L-Connectivity 2. It can be adjusted by setting the environmental variables. For this purpose run (wheres `k` comes from k-clique and `l` comes from l-connectivity):

```
$env:K_CLIQUE = <k>
$env:L_CONN = <l>
```

Input file for the `exe\main.exe` have to be a text file (with extension `.txt` in the following form:

1. First line contains number of graphs.
2. Starting from second line, graphs shall be written (as many as stated in first line). Each graph is constructed as follows:
   (a) If not a first graph in a file, one empty line.
   (b) First line with number indicating size of the graph (let's denote it n).
   (c) In the following n lines adjacency matrix of the first graph.

With the

To generate graph with `exe\generator.exe` and append the output to the file run:

```
.\exe\generator.exe <size of graph> <max number of edges between two
↪  nodes> >> <path to file>
```

For testing purposes, 3 input files are provided inside `examples` directory. On top of that, PowerShell script was prepared, which will run the application with output from all input files located in `examples` directory (assuming proper structure and `.txt` extension) with different k-clique and l-connectivity setups. To run it, one have to execute:

```
.\exe\test.ps1
```

To adjust k-clique and l-connectivity setups, one have to open `exe\test.ps1` script with some editor and adjust variable `$TestVariants`.

7 REFERENCES

References

Abu-Khzam, Faisal N. (2007), "The maximum common subgraph problem." URL `https://www.academia.edu/14538910/The_Maximum_Common_Subgraph_Problem_Faster_Solutions_via_Vertex_Cover`.

Etsuji Tomita, Haruhisa Takahashi, Akira Tanaka (2006), "The worst-case time complexity for generating all maximal cliques and computational experiments." URL `https://www.sciencedirect.com/science/article/pii/S0304397506003586`.

ProofWiki (2023), "Graph (discrete mathematics)." URL `https://proofwiki.org/wiki/Definition:Graph_(Graph_Theory)/Size`.

StackOverFlow (2021), "Finding a single maximal clique using bron–kerbosch." URL `https://stackoverflow.com/questions/68609547/finding-a-single-maximal-clique-using-bron-kerbosch`.

Vishwas, Sadanand (2020), "Using bron kerbosch algorithm to find maximal cliques." URL `https://iq.opengenus.org/bron-kerbosch-algorithm/`.