Algorithms and Computability laboratory task

Jakub Oganowski, Filip Szmit, Mikołaj Trebski, Karol Wanielista

December 7, 2024

1 Task 1: Graph Size Calculation Algorithm

The graph size calculation algorithm computes the total size of a graph or multigraph by counting the total number of edges in its adjacency matrix. This size represents the cumulative edge count, making it applicable to both simple graphs and multigraphs.

1.1 Algorithm Description

The algorithm accounts for both directed and undirected multigraphs. For directed graphs, it sums only the entries in the upper triangular part of the adjacency matrix to avoid double-counting directed edges. For undirected graphs, it processes the entire adjacency matrix, ensuring that all edge occurrences, including multiple edges between the same pair of vertices, are included.

1.2 Complexity Analysis

The algorithm's complexity depends on the size of the adjacency matrix, which has dimensions $V \times V$, where V is the number of vertices. Consequently, the algorithm has a time complexity of $O(V^2)$. This complexity applies to both directed and undirected multigraphs since the adjacency matrix must be traversed in either case.

2 Task 2: Graph Edit Distance

Graph Edit Distance (GED) is a method used to find the minimal set of transformations required to convert one graph into another. Introduced by Sanfeliu and Fu in 1983, GED involves a series of edit operations such as inserting, deleting, and substituting vertices and their corresponding edges in a graph. The aim is to transform a graph g_1 into another graph g_2 by minimizing the cumulative cost of these operations.

Let $g_1 = (V_1, E_1, \mu_1, v_1)$ and $g_2 = (V_2, E_2, \mu_2, v_2)$ be two graphs. The graph edit distance between g_1 and g_2 is defined as:

$$GED(g_1, g_2) = \min_{e_1,\ldots,e_k \in \gamma(g_1,g_2)} \sum_{i=1}^{k} c(e_i)$$

where c denotes the cost function measuring the strength $c(e_i)$ of an edit operation e_i, and $\gamma(g_1, g_2)$ represents the set of edit paths transforming g_1 into g_2. The edit operations allowed include vertex insertion ($\epsilon \to v$), vertex deletion ($u \to \epsilon$), and vertex substitution ($u \to v$). The same operations can be done on edges.

GED is often addressed through heuristic methods that may yield suboptimal solutions due to their approximate nature. These methods are effective for faster computation but do not guarantee the best transformation. In contrast, only a few exact methods have been developed, such as Riesen et al. (2007), which offer exact solutions but are limited by computational feasibility for large graphs.

2.1 Exact Graph Edit Distance Computation

A widely utilized method for the exact calculation of GED is based on the A* algorithm, known as A*GED (Abu-Aisheh et al. 2015). This algorithm explores the space of possible mappings between two graphs using a dynamically constructed search tree. The A*GED approach iteratively expands nodes in the search tree by creating successor nodes, thereby finding the minimum cost edit path from g_1 to g_2.

Input: Non-empty attributed graphs $g_1 = (V_1, E_1, \mu_1, v_1)$ and $g_2 = (V_2, E_2, \mu_2, v_2)$

where $V_1 = \{u_1, \ldots, u_{|V_1|}\}$ and $V_2 = \{v_1, \ldots, v_{|V_2|}\}$

Output: Minimum cost edit path $p_{\min}$ from g_1 to g_2, such as $\{u_1 \to v_3, u_2 \to \epsilon, \epsilon \to v_2\}$

```
1: OPEN ← $\{\phi\}, p_{\min} \leftarrow \phi$
2: For each node $w \in V_2$, OPEN ← OPEN $\cup \{u_1 \to w\}$
3: OPEN ← OPEN $\cup \{u_1 \to \varepsilon\}$
4: while true do
5:    $p_{\min} \leftarrow \arg\min\{g(p) + \mathrm{lb}(p)\}$ such that $p \in$ OPEN
6:    OPEN ← OPEN $\setminus p_{\min}$
7:    if $p_{\min}$ is a complete edit path then
8:       Return $p_{\min}$ as a solution (i.e., the minimum cost edit distance from $g_1$ to $g_2$)
9:    else
10:      Let $p_{\min} \leftarrow \{u_1 \to v_{i1}, \ldots, u_k \to v_{ik}\}$
11:      if $k < |V_1|$ then
12:         For each $w \in V_2 \setminus \{v_{i1}, \ldots, v_{ik}\}$, OPEN ← OPEN $\cup \{p_{\min} \cup \{u_{k+1} \to w\}\}$
13:         $p_{\text{new}} \leftarrow p_{\min} \cup \{u_{k+1} \to \varepsilon\}$
14:         OPEN ← OPEN $\cup \{p_{\text{new}}\}$
15:      else
16:         $p_{\text{new}} \leftarrow p_{\min} \cup \bigcup_{w \in V_2 \setminus \{v_{i1}, \ldots, v_{ik}\}} \{\varepsilon \to w\}$
17:         OPEN ← OPEN $\cup \{p_{\text{new}}\}$
18:      end if
19:   end if
20: end while
```

2.1.1 Computational Complexity

Heuristic Calculation The function 'lb(p)' for A*GED involves solving an assignment problem with a complexity of $O(\max(n_1, n_2)^3)$, where n_1 and n_2 are the number of vertices in g_1 and g_2, respectively. This cubic complexity makes it impractical for very large graphs.

Search Complexity The total complexity of the A*GED algorithm is primarily driven by the complexity of the heuristic function. In the worst case, it can explore an exponential search space, making it non-polynomial in nature.

2.2 Approximate Graph Edit Distance (GED) Algorithm

Given the computational complexity of exact Graph Edit Distance (GED) algorithms, especially for large graphs, approximate methods provide a feasible alternative to estimate the GED within reasonable computational limits.

2.2.1 Algorithm Description

The algorithm approximates the GED by following these steps:

Greedy Node Mapping The algorithm starts by performing a greedy mapping of nodes from graph g_1 to graph g_2. For each node u_1 in g_1, it attempts to find the best matching node u_2 in g_2 by evaluating the substitution cost. The steps are as follows:

1. Initialize an unordered map to store the mapping of nodes from g_1 to g_2, and a set to track already mapped nodes in g_2.

2. For each node u_1 in g_1, iterate over all nodes u_2 in g_2 to find the node with the minimum substitution cost that has not been mapped yet.

3. If a suitable match is found, record the mapping and update the set. If no match is found, mark the node u_1 as deleted by mapping it to -1.

Compute Node Mapping Cost Once the nodes are mapped, the algorithm computes the total cost associated with the node mappings:

1. Initialize the total cost to zero.
2. Iterate over each mapping. For each node u_1 in g_1:
   - If the node is marked as deleted (mapped to -1), add the deletion cost to the total cost.
   - Otherwise, add the substitution cost for the mapped nodes to the total cost.

Compute Edge Mapping Cost Next, the algorithm calculates the cost of transforming edges based on the previously determined node mappings:

1. For each edge (u_1, v_1) in g_1, determine if the corresponding edge (u_2, v_2) exists in g_2 based on the node mappings.
2. If an edge does not exist (i.e., either u_2 or v_2 is -1), add the edge deletion cost to the total cost.
3. Otherwise, add the edge substitution cost based on the adjacency matrices of g_1 and g_2.

Add Cost for Unmapped Nodes in g_2 Finally, the algorithm accounts for any nodes in g_2 that were not mapped from g_1:

1. Iterate over each node u_2 in g_2. If the node is not present in the set, add the insertion cost to the total cost.

2.2.2 Computational Complexity of the Approximate GED Algorithm

The algorithm primarily consists of three main steps: greedy node mapping, node mapping cost computation, and edge mapping cost computation. Here is the evaluation of the complexity of each step individually.

Greedy Node Mapping The first step involves a greedy approach to map nodes from graph g_1 to graph g_2.

- For each node u_1 in g_1, the algorithm searches for the best matching node u_2 in g_2.
- The search involves iterating over all nodes in g_2 to find the node with the minimum substitution cost.

Let n be the number of vertices in g_1 and m be the number of vertices in g_2. The time complexity of this step is:

$$O(n \cdot m)$$

Compute Node Mapping Cost The second step computes the cost associated with the node mappings.

- The algorithm iterates over each node in g_1 and checks if it has been marked as deleted or substituted.

The time complexity of this step is:

$$O(n)$$

Compute Edge Mapping Cost The third step calculates the cost of transforming edges based on the node mappings.

- For each edge (u_1, v_1) in g_1, the algorithm checks if the corresponding edge (u_2, v_2) exists in g_2 based on the node mappings.
- This involves iterating over all possible pairs of nodes in g_1.

The time complexity of this step is:

$$O(n^2)$$

Add Cost for Unmapped Nodes in g_2 The final step adds the cost for any nodes in g_2 that were not mapped from g_1.

- The algorithm iterates over each node in g_2 to check if it has been mapped.

The time complexity of this step is:

$$O(m)$$

Overall Complexity The overall time complexity of the approximate GED algorithm is the sum of the complexities of the individual steps. Therefore, the total complexity is:

$$O(n \cdot m) + O(n) + O(n^2) + O(m)$$

Since $O(n^2)$ is the dominant term, the overall computational complexity of the algorithm is:

$$O(n^2)$$

This analysis indicates that the algorithm scales quadratically with respect to the number of vertices in graph g_1. While this approximation method provides a quick estimate for the GED, its quadratic complexity may still pose challenges for very large graphs.

2.3 Computational testing

Graph Generation for Verification

To analyze both exact and approximate solutions, random multigraphs with varying sizes were generated. It creates a random graph with a given number of vertices and populates the adjacency matrix with random number of edges, ensuring variability in the graphs for each test.

Analysis of Exact Timing

The exact timing for solving GED problems was analyzed using data from graph sizes 3 to 8. The computation times were recorded for each pair of graphs with the same size:

- Size 3: 0.0030445 seconds
- Size 4: 0.0166687 seconds
- Size 5: 0.134561 seconds
- Size 6: 1.26792 seconds
- Size 7: 13.5313 seconds
- Size 8: 164.751 seconds

An exponential model was fitted to these data points to predict the times required for graph sizes 9 and 10. The results indicated a rapid increase in computation time:

- Predicted time for graph size 9: 1308400.56 seconds
- Predicted time for graph size 10: 4.47×10^{14} seconds

This highlights the limitations of exact solutions for large graph sizes due to exponential growth in computational requirements.

2.3.1 Analysis of Approximate Solution

For the approximate solution, a scatter plot was created to visualize the computation times for graph sizes from 10 to 500. This plot allowed us to compare the real-time data with the predicted values using an exponential model. The scatter plot illustrates how, as graph size increases, the computation times predicted by approximate methods remain reasonable and feasible compared to exact solutions. This provides evidence of the practicality of using approximate methods for larger graphs, where exact solutions become computationally unfeasible.

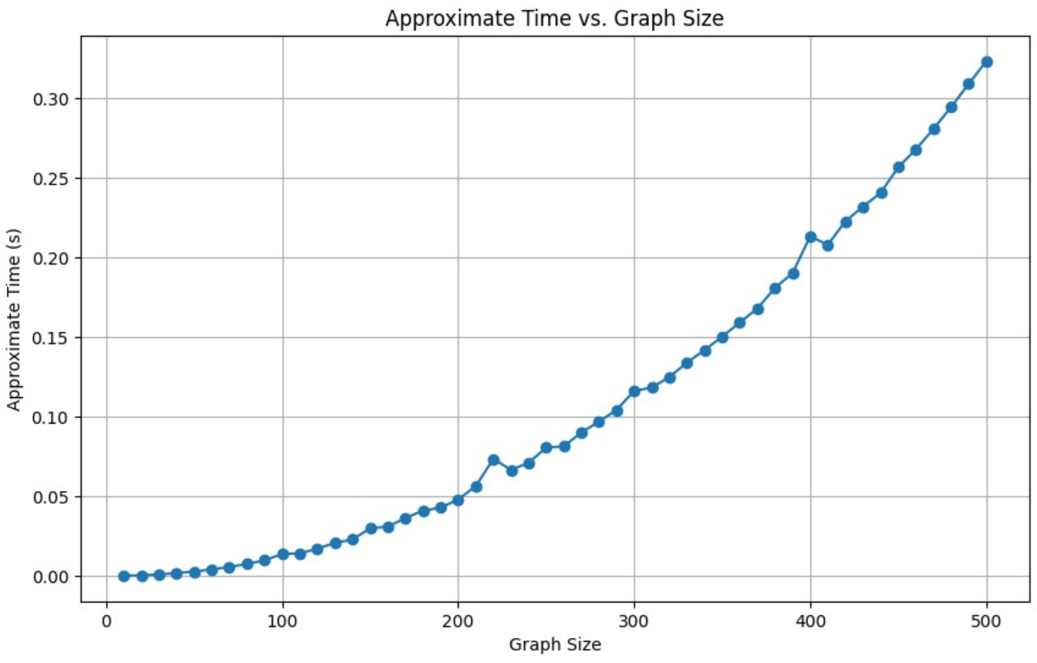


Figure 2.1: Approx algorithm computation time for various sizes of a graph

3 Task 3: Maximal Cycle in a Graph; Number of maximal cycles

For the purposes of this algorithm we define a simple cycle in a directed graph as a non-empty trail with at least 3 distinct vertices, where only the first and last verices are equal.

The problem of finding the maximal cycle in a graph necessarily requires answering the question of whether that graph has a Hamiltonian Cycle. The Hamiltonian cycle problem is NP-complete, so no polynomial time exact solution to the posed problem exists unless $P = NP$.
As such, two algorithms are presented: an exact algorithm with super-polynomial time complexity and an approximation algorithm with polynomial time complexity.
It should be noted that these algorithms treat all passed graphs as directed graphs, and return the number of directed cycles.

3.1 Exact Algorithm

This algorithm is based on the work of (Giscard, Kriege, and Wilson 2019), who found an efficient algorithm for counting simple cycles of a given length in directed graphs. In short they produced the following formula:

$$\gamma(l) = \frac{(-1)^l}{l} \sum_{H \prec_{conn} G} \binom{|N(H)|}{l - |H|} (-1)^{|H|} Tr(A_H^l)$$

Where:

l denotes the length of the cycles we wish to count

$\gamma(l)$ is the number of cycles of length l in the given graph

$\lvert H\rvert$	denotes the number of vertices in the given subgraph of G
$N(H)$	denotes the neighbors of the subgraph H in G
A_H	is the adjacency matrix of H
$Tr(M)$	is the matrix trace

A neighbor of H in G is a vertex $v \in G$ such that $v \notin H$ and such that there exists at least one edge from v to at least one vertex in H. The above sum iterates over all weakly connected subgraphs of G.

This algorithm itself is derived from previous work by (Giscard and Rochet 2018), who found a similar algorithm for simple paths in a graph, based on the fact that subsequent powers of the adjacency matrix enumerate the number of simple paths of increasing length. More specifically:

$$(A_G^k)_{ij} = \sum_{p: i \to j, l(p)=k} p, \; i, j = 0, 1, 2, \ldots, N$$

Our solution calculates $\gamma(l)$ up to $|V(G)|$ times for the passed graph G, returning the highest such $l \leq |V(G)|$ where $\gamma(l) \neq 0$ and the corresponding value of $\gamma(l)$.

The authors of this algorithm have calculated the time complexity of finding $\gamma(l)$ as $O(|V(G)| + |E(G)| + (l^\omega + l\Delta)|S_l|)$, where $|S_l|$ is the number of weakly connected induced subgraphs of G on at most l vertices, Δ is the maximum degree of any vertex in G and ω is the exponent of matrix multiplication.

We analyze our own implementation to check whether we have achieved the same time complexity.

Our graphs are stored as $|V(G) \times V(G)|$ adjacency matrices, with each entry in the matrix being a number in $\mathbb{N}_{0+}$. We first flatten the matrix, reducing all numbers greater than 1 to 1. This requires $O(|V|^2)$ operations.

We then calculate $\gamma(l)$ for each possible l, starting at $|V|$ and descending to $l = 3$, which in the worst case scenario requires $O(|V|)$ iterations.

In each iteration, we generate all possible sub-graphs of H, of which there are $2^{|V|}$. Generating a sub-graph requires $O(|V|^2)$ operations and also produces the number of neighbors of H in G. If a graph is weakly connected, which we can find out in $O(|V|^2)$ operations, we calculate the l-th power of the adjacency matrix of the sub-graph, which requires $O(\log_2(|V|) * |V|^3)$ operations. Likewise, for each weakly connected sub-graph we must take the trace of the resulting matrix power and calculate the binomial, both of which can be calculated in $O(|V|)$ operations.

In the end, we obtain the final complexity of our algorithm:

$$O(|V|^2 + |V| * (2^{|V|} * |V|^2 + S_{|V|} * (2|V| + \log_2(|V|) * |V^3|))$$

Since the dominant element in the complexity analysis is $O(2^{|V|}$, we can conclude that the above algorithm is of exponential complexity with a linear exponent.

We have therefore lost some efficiencies from the presented algorithm, as we scale proportionally to the number of all sub-graphs, as opposed to the number of weakly connected sub-graphs; However, both of these variable scale exponentially with respect to the number of vertices and edges.

3.2 Approximation Algorithm

The approximation algorithm removes vertices that logically cannot be a part of a cycle, uses the Kosaraju-Sharir's algorithm to find strongly connected components of the resultant sub-graph, and finds the longest cycle in each component via a heuristic DFS algorithm by (Kumar and Gupta 2014) and estimates the number of such cycles with the binomial and gamma functions.

A vertex of a digraph can never be a part of a cycle if it lacks incoming or outgoing edges (or both), or it it has only one unique neighbor. These vertices are removed and the algorithm is applied again, until no more vertices are removed.

Each iteration of the algorithm requires $O(|V|^2)$ operations, and in the worst-case scenario (when only one vertex is removed per iteration) the algorithm will require $|V|$ iterations. Notably, when the graph is acyclic, the entire graph will be removed and the algorithm will correctly return max cycle size of 0 and 0 total cycles.

Next, the Kosaraju-Sharir's algorithm is used to separate the graph into strongly-connected components, which requires $O(|V|^2)$ operations.

Finally, the DFS-based algorithm presented in (Kumar and Gupta 2014) is used to find the longest cycle in each component, with time complexity $O(|V|^2)$, due to our graph being represented as an adjacency matrix.

Once that calculation is complete, we use the gamma and binomial functions to estimate the number of these cycles. Since we know the resultant graphs are not acyclic, as otherwise the algorithm would have returned in the pruning step, the number of cycles for a given sub-graph is bounded between 1, $(n-1)!$, with $(n-1)!$ being the number of Hamiltonian cycles in a complete digraph of size $|V| = n$. We likewise note that the number of cycles will be proportional to the average degree of the graph Δ_{Avg}

With this in mind, we estimate the number of cycles of length l in a sub-graph $H = (V, E)$ to be

$$\gamma(l) \approx \lfloor \binom{|V|}{l} * \Gamma(\Delta_{Avg}) \rfloor$$

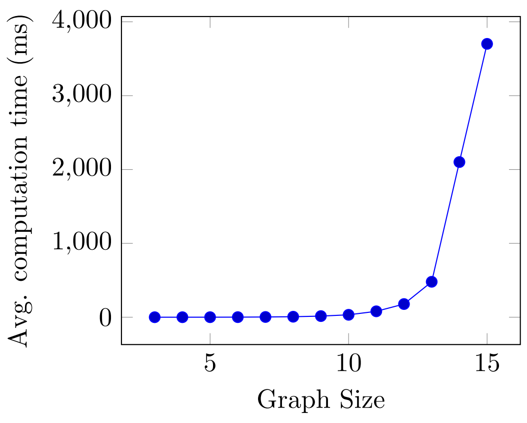
We sum these results for each component for the highest obtained l to obtain the final result.

3.3 Computational testing

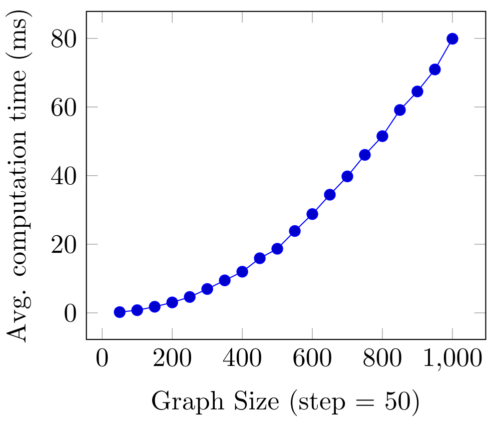
Benchmarks were performed for both the exact and the approximate algorithm by generating 100 random graphs of sizes between 3 and 15. Note that the problem as posed is not affected by edge multiplicities in a graph, only whether a directed edge connecting two vertices exists.

As expected, the benchmarks for the exact algorithm follow an exponential curve with respect to graph size (Figure 4.1)

Benchmarks for the approximate algorithm are much more noisy, but appear as expected for polynomial time complexity. It should be noted that the approximate algorithm is much more vulnerable to random noise that is otherwise irrelevant to the task, such as the ordering of the vertices in the generated graphs.

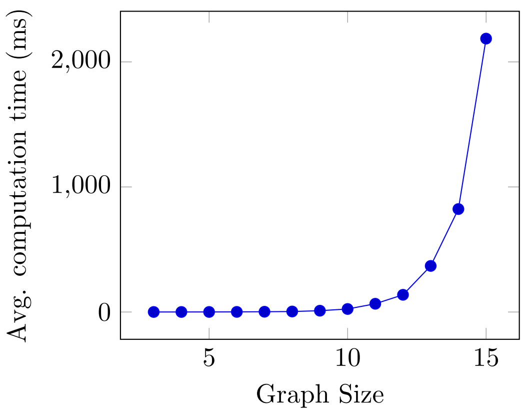


(a) Exact algorithm

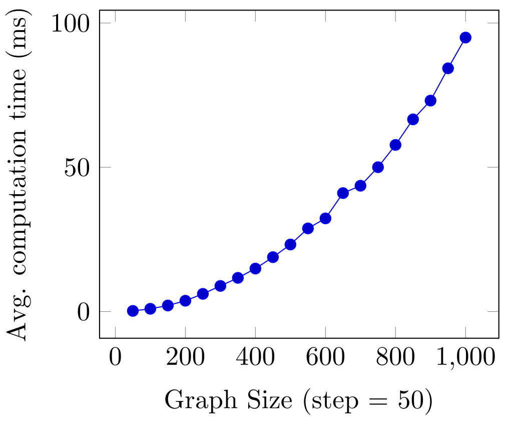


(b) Approximate algorithm

Figure 3.1: Benchmark results for task 3, randomized densities

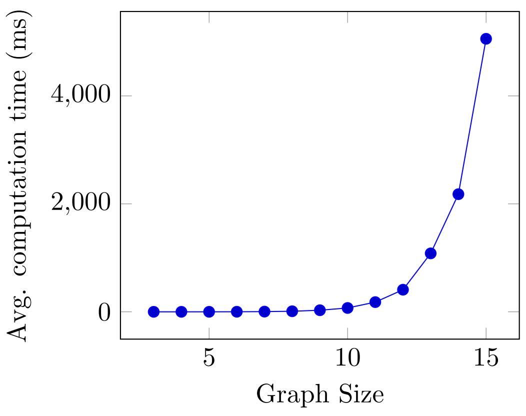


(a) Exact algorithm

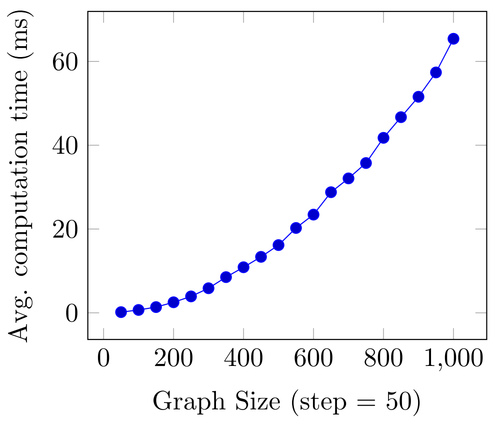


(b) Approximate algorithm

Figure 3.2: Benchmark results for task 3, dense ($p(1) == 0.8$) matricies



(a) Exact algorithm



(b) Approximate algorithm

Figure 3.3: Benchmark results for task 3, dense ($p(1) == 0.8$) matricies

4 Task 4: Minimal extension of a graph to generate Hamiltonian cycle, number of all Hamiltonian cycles

This problem focuses on graph theory, particularly Hamiltonian cycles. A Hamiltonian cycle is a closed loop in a graph that visits each vertex exactly once before returning to the starting point. The main tasks are:

1. **Minimal Extension of a Graph to Generate a Hamiltonian Cycle:** Determine the minimal number of edges that must be added to the graph to create such a cycle. We may also refer to this problem as **Hamiltonian completion problem**
2. **Counting Hamiltonian Cycles:** Enumerate all unique closed loops in the graph that visit every vertex exactly once.

4.1 On the applicability of the problem to Multigraphs

Due to the nature of a Hamiltonian cycle, the problem is not influenced by the properties of multigraphs. A Hamiltonian cycle requires visiting each vertex exactly once, regardless of the presence of multiple edges between the same pair of vertices.

As a result, the problem can be equivalently analyzed on simple graphs, where each edge between any two vertices is unique. Even if the input graph is a multigraph, we can treat it as a simple graph by considering only the existence of a connection between vertices, rather than the number of edges.

This simplification does not affect the correctness or efficiency of the result:

- **Correctness:** Since a Hamiltonian cycle inherently visits each vertex only once, additional edges between vertices in a multigraph do not contribute to the solution.
- **Efficiency:** Treating the graph as a simple graph reduces redundancy and ensures we focus on the structural connectivity necessary to form a Hamiltonian cycle.

4.2 Hamiltonian completion - Exact algorithm

4.2.1 Exact Algorithm Description

The algorithm solves two related problems:

1. Determine if a graph already has a Hamiltonian cycle.
2. Find the minimal number of edges to add to the graph to make it Hamiltonian if it does not already have a Hamiltonian cycle.

The algorithm starts by checking whether the initial graph contains a Hamiltonian cycle using a backtracking approach. The backtracking algorithm verifies:

1. All vertices are visited exactly once.
2. There exists a direct edge connecting the last vertex back to the starting vertex.

If the graph is already Hamiltonian, the algorithm returns 0. Otherwise, it proceeds as follows:

1. Construct a list of all missing edges (pairs of vertices not directly connected).
2. Iteratively test subsets of the missing edges:
   (a) Temporarily add edges in the subset to the graph.
   (b) Check if the modified graph has a Hamiltonian cycle using function, which is backtracking a graph.
   (c) Restore the graph to its original state after testing.
3. Return the vector of pair of edges that should be added to make a graph Hamiltonian

4.2.2 Computational Complexity

The computational complexity of the algorithm arises from two main components:

- **Hamiltonian Cycle Check:** The backtracking algorithm explores all possible permutations of the vertices to determine if a Hamiltonian cycle exists. In the worst case, there are $n!$ permutations, and each path must be checked for validity. Hence, the complexity of this step is $O(n!)$.
- **Subset Testing:** If the graph is not Hamiltonian, the algorithm generates all subsets of m missing edges to determine the minimal number of edges needed to create a Hamiltonian cycle. The total number of subsets is 2^m, and for each subset, a Hamiltonian cycle check is performed. This results in a complexity of $O(2^m \cdot n!)$ for this step.
- **Total Complexity:** The overall complexity is dominated by the combination of subset testing and the Hamiltonian cycle check, leading to a total complexity of $O(2^m \cdot n!)$.

4.2.3 Proof of Correctness

1. Tests all possible edge additions (completeness).
2. Identifies the smallest subset of edges needed (minimality).
3. Accurately detects Hamiltonian cycles via backtracking.

4.2.4 Testing

The algorithm was tested on various types of graphs, including:

- Sparse Graphs - generates a minimally connected graph where each vertex is connected to the next in a chain-like structure. The graph has exactly n-1 edges for n vertices, forming a path.
- Dense Graphs - Creates a graph where most vertices are connected to each other. Every possible edge between vertices is included, except for self-loops (no edges from a vertex to itself).
- Random Graphs (50% edge density) - This function creates a graph with a specified number of vertices and a given edge density. Each possible edge between vertices is randomly included based on the density parameter (a percentage value). For example, if the density is 50, each edge has a 50
- Graph with only one edge - Generates a graph with n vertices and only one edge connecting the first two vertices (0 and 1). All other vertices are disconnected. This is directed graph.
- Directed Star graph - Creates a directed star graph with one central vertex (vertex 0) pointing to all other vertices. Each edge is directed outward from the central vertex.
- Undirected Star Graph - Generates an undirected star graph where one central vertex (vertex 0) is connected to all other vertices, and the edges are bidirectional.

4.2.5 Test Results

```
Testing for n = 3 vertices:
Sparse graph: Minimal extensions = 1, Time taken = 8.4e-06 seconds.
Dense graph: Minimal extensions = 0, Time taken = 1.6e-06 seconds.
Random graph: Minimal extensions = 3, Time taken = 9e-06 seconds.
One Edge graph: Minimal extensions = 2, Time taken = 1.37e-05 seconds.
Star Directed graph: Minimal extensions = 2, Time taken = 7.3e-06 seconds.
Star Undirected graph: Minimal extensions = 1, Time taken = 2.2e-06 seconds.

Testing for n = 4 vertices:
Sparse graph: Minimal extensions = 1, Time taken = 4.5e-06 seconds.
Dense graph: Minimal extensions = 0, Time taken = 1.9e-06 seconds.
Random graph: Minimal extensions = 2, Time taken = 9.7e-06 seconds.
One Edge graph: Minimal extensions = 3, Time taken = 0.0001979 seconds.
Star Directed graph: Minimal extensions = 3, Time taken = 7.71e-05 seconds.
Star Undirected graph: Minimal extensions = 2, Time taken = 6.9e-06 seconds.

Testing for n = 5 vertices:
Sparse graph: Minimal extensions = 1, Time taken = 7.2e-06 seconds.
Dense graph: Minimal extensions = 0, Time taken = 2.2e-06 seconds.
Random graph: Minimal extensions = 2, Time taken = 1.89e-05 seconds.
One Edge graph: Minimal extensions = 4, Time taken = 0.0057831 seconds.
Star Directed graph: Minimal extensions = 4, Time taken = 0.0016498 seconds.
Star Undirected graph: Minimal extensions = 3, Time taken = 4.5e-05 seconds.

Testing for n = 6 vertices:
Sparse graph: Minimal extensions = 1, Time taken = 1.21e-05 seconds.
Dense graph: Minimal extensions = 0, Time taken = 2.7e-06 seconds.
Random graph: Minimal extensions = 3, Time taken = 0.0004232 seconds.
One Edge graph: Minimal extensions = 5, Time taken = 0.20576 seconds.
Star Directed graph: Minimal extensions = 5, Time taken = 0.0499779 seconds.
Star Undirected graph: Minimal extensions = 4, Time taken = 0.0005181 seconds.

Testing for n = 7 vertices:
Sparse graph: Minimal extensions = 1, Time taken = 1.7e-05 seconds.
Dense graph: Minimal extensions = 0, Time taken = 3.1e-06 seconds.
Random graph: Minimal extensions = 0, Time taken = 5.2e-06 seconds.
```

```
One Edge graph: Minimal extensions = 6, Time taken = 9.55669 seconds.
Star Directed graph: Minimal extensions = 6, Time taken = 1.97943 seconds.
Star Undirected graph: Minimal extensions = 5, Time taken = 0.0084967 seconds.
```

4.2.6 Plots

*Sparse Random is One Edge - error in naming

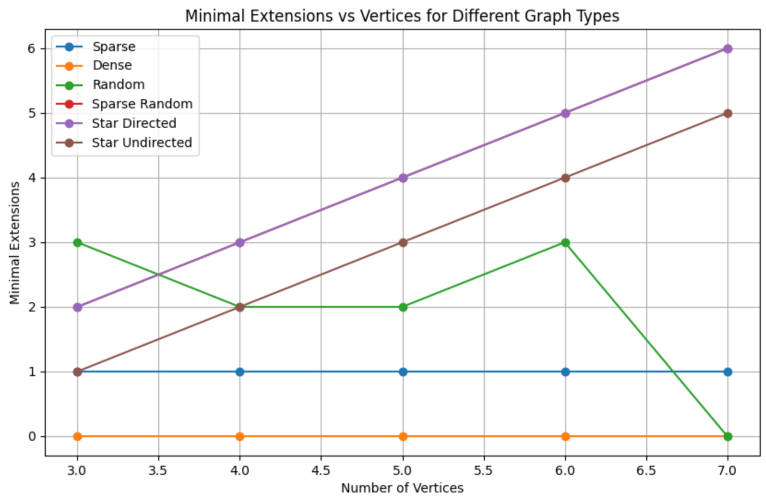


Figure 4.1: Minimal Extensions vs Vertices for Different Graph Types

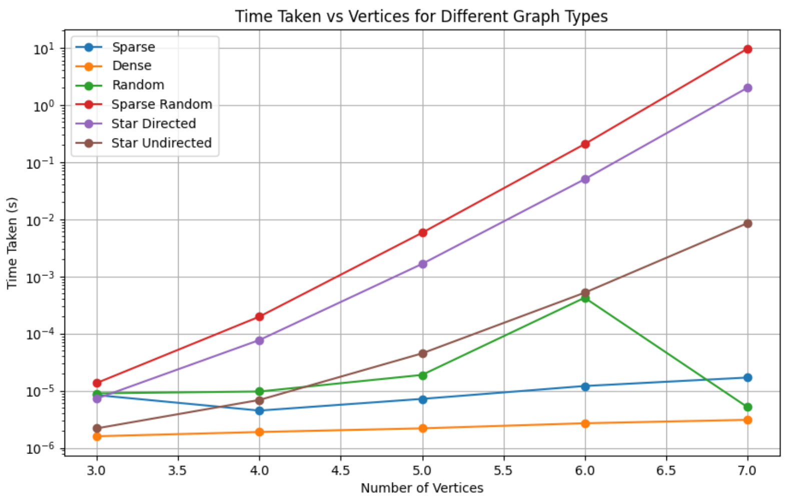


Figure 4.2: Time Taken vs Vertices for Different Graph Types (Log Scale)

4.2.7 Conclusions from the Test Results

The testing was conducted on various graph types, ranging from sparse to dense, random, and star graphs (both directed and undirected). Below are the key conclusions drawn from the results:

1. **Sparse Graphs:**
   - Sparse graphs consistently required exactly 1 minimal extension to become Hamiltonian.
   - This aligns with the expectation that a sparse graph, resembling a linear chain, needs only a single edge to close the cycle.
   - The time taken to compute minimal extensions remained very low, even as the number of vertices increased.

2. **Dense Graphs:**
    - Dense graphs required 0 minimal extensions, as they are already Hamiltonian due to their high connectivity.
    - The computation time for dense graphs was the lowest among all graph types, indicating that the algorithm quickly identifies their Hamiltonian property.
3. **Random Graphs:**
    - Random graphs showed variability in minimal extensions required, depending on their initial structure.
    - The time taken was higher compared to sparse and dense graphs, reflecting the increased complexity of analyzing random connectivity patterns.
4. **Sparse Random Graphs:**
    - Sparse random graphs required an increasing number of minimal extensions as the number of vertices grew.
    - The time taken for these graphs grew significantly, especially for larger n, due to the algorithm's need to test subsets of missing edges extensively.
5. **Star Directed Graphs:**
    - Directed star graphs consistently required $n-1$ minimal extensions, as all leaves need to be connected into a cycle.
    - The time taken for directed star graphs grew exponentially for larger n, reflecting the complexity of finding the necessary edge additions in a directed structure.
6. **Star Undirected Graphs:**
    - Undirected star graphs required fewer minimal extensions compared to their directed counterparts, aligning with the simpler nature of undirected connectivity.
    - The time taken for computation was moderate but increased steadily with larger n.

4.2.8 Overall Observations

- The algorithm performed efficiently for sparse and dense graphs, quickly identifying the minimal extensions required or confirming that none were needed.
- For more complex graph types, such as random, sparse random, and star graphs, the time complexity increased due to the need for iterative edge testing.
- Directed graphs were more computationally intensive compared to their undirected counterparts, as expected due to the additional constraints imposed by directionality.

4.3 Hamiltonian completion - Approximation algorithm

In this subsection, we will discuss an approximation algorithm to the Hamiltonian completion problem, which runs in polynomial time.

4.3.1 Reduction of Hamiltonian completion problem to a Travelling salesman problem

Let us formally define the reduction from the Hamiltonian Completion problem to the Traveling Salesman Problem (TSP), considering both directed and undirected cases.

Undirected Case (Symmetric TSP) Given an undirected graph $G = (V, E)$, we construct a complete weighted graph $G' = (V, E')$ where:

- V is the same vertex set as in G
- E' contains all possible undirected edges between vertices in V
- For each edge $e \in E'$:
    - If $e \in E$, set weight $w(e) = 0$
    - If $e \notin E$, set weight $w(e) = 1$

This results in a symmetric TSP instance where $w(u, v) = w(v, u)$ for all vertices $u, v \in V$.

Directed Case (Asymmetric TSP) Given a directed graph $G = (V, A)$, we construct a complete weighted directed graph $G' = (V, A')$ where:

- V is the same vertex set as in G
- A' contains all possible directed arcs between vertices in V
- For each arc $a \in A'$:
  - If $a \in A$, set weight $w(a) = 0$
  - If $a \notin A$, set weight $w(a) = 1$

This results in an asymmetric TSP instance where $w(u, v)$ may differ from $w(v, u)$ for vertices $u, v \in V$.

Proof for both undirected and directed case:

Theorem: For both directed and undirected cases, the minimum number of edges/arcs needed to make G Hamiltonian equals the minimum weight of a Hamiltonian cycle in G'.

Proof: Let k be the minimum number of edges/arcs needed to make G Hamiltonian, and let w^* be the minimum weight of a Hamiltonian cycle in G'.

Since G' is complete, a Hamiltonian cycle exists in G'.

($k \leq w^*$): Let T be a minimum weight Hamiltonian cycle in G' with weight w^*. The edges/arcs of weight 1 in T correspond to edges/arcs that need to be added to G to make it Hamiltonian.

($w^* \leq k$): Let S be a minimum set of k edges/arcs whose addition makes G Hamiltonian. The resulting Hamiltonian cycle uses these k edges/arcs (weight 1 each) and existing edges/arcs from G (weight 0 each), giving a TSP tour of weight k.

4.3.2 Choice of approximation heuristic for TSP / ATSP

There are a couple of remarks with respect to heuristics we can use to find and approximate minimal weight Hamiltonian cycle in G':

1. Some heuristics are for undirected graphs (and symmetric TSP) only. We want an algorithm that can be adapted for both directed and undirected graphs (symmetric and assymetric TSP).
2. Many well known approximation algorithms, such as Christofides algorithm for TSP (Gutin and Punnen 2007, p. 403), or one developed by Frieze, Galbiati, and Maffioli for ATSP (Gutin and Punnen 2007, p. 458) assume triangle inequality is satisfied. In our complete graph G' we make edge weights equal to 0 or 1. Triangle inequality will not be satisfied when there is no edge a, b in G, making the weight in G' $w(a, b) = 1$, while there are edges a, c and b, c in G, making the weights in G' $w(a, c) = w(b, c) = 0$. Then, $w(a, c) + w(b, c) \ngeq w(a, b)$.

We decided to implement a **Iterated 3-opt** heuristic based on (Gutin and Punnen 2007, pp. 407–408, 460–462). As it can be adapted for both symmetric and asymmetric cases of TSP, and it does not assume any conditions like the triangle inequality.

4.3.3 Iterated 3-opt heuristic description

Initial cycle construction The algorithm begins with a nearest neighbor heuristic to create a cycle:

1. Start from a vertex with the highest degree in the original graph
2. Repeatedly select the closest unvisited vertex until all vertices are visited
3. Close the cycle by returning to the start vertex

Then, **3-Opt local search** is used to optimize this cycle and find a shorter one.

3-Opt local search The 3-opt local search examines all possible ways to improve the cycle by removing three edges and reconnecting the resulting segments in a different configuration.

1. Selecting three edges $(i, i+1)$, $(j, j+1)$, and $(k, k+1)$ from the current tour
2. Removing these edges
3. Reconnecting the resulting segments in a different configuration that yields a valid tour
4. Accepting the move if it reduces the total weight (number of added edges)

For directed graphs (ATSP), segment reversals are not considered as they would change multiple edge weights. For undirected graphs (STSP), segment reversals are allowed. In the implementation, segments spanning through the start/end vertex of the list representation of a cycle are carefully considered and properly handled, as well as for pairs of edges that are not disjoint.

The local search continues until no improving 3-opt move is found or until a maximum number of iterations ($10n$, where n is the number of vertices) is reached.

4.3.4 Time Complexity

- Finding out if graph is directed or not: $O(n^2)$
- Initial tour construction: $O(n^2)$
- Each 3-opt iteration: $O(n^3)$, consisting of:
  - Loop going through all combinations of 3 edges in a cycle: $O(n^3)$
  - Iterating over all combinations of the selected 3 edges: $O(1)$ - 8 combinations maximum
- Maximum number of iterations: $O(n)$

Overall worst-case complexity: $O(n^4)$

4.3.5 Implementation notes

The current implementation does not include several optimizations mentioned in the literature:

- Avoiding search space redundancy
- Bounded neighbor lists
- Don't-look bits
- Tree-based tour representation

These optimizations could potentially improve the running time, but would add considerable implementation complexity and would also trade a potential slight degradation in tour quality. (Gutin and Punnen 2007, p. 408).

4.3.6 Testing

Testing setup is the same as in the section for an exact algorithm. Let's compare the output of approximation algorithm to the exact algorithm results for graphs with n vertices with $3 \leq n \leq 7$.

```
Testing for n = 3 vertices:
Sparse graph: Minimal extensions = 1, Time taken = 0.00013 seconds.
Dense graph: Minimal extensions = 0, Time taken = 8.33e-05 seconds.
Random graph: Minimal extensions = 1, Time taken = 7.31e-05 seconds.
One Edge graph: Minimal extensions = 2, Time taken = 9.77e-05 seconds.
Star Directed graph: Minimal extensions = 2, Time taken = 5.89e-05 seconds.
Star Undirected graph: Minimal extensions = 1, Time taken = 8.08e-05 seconds.

Testing for n = 4 vertices:
Sparse graph: Minimal extensions = 1, Time taken = 0.000411 seconds.
Dense graph: Minimal extensions = 0, Time taken = 0.0001612 seconds.
Random graph: Minimal extensions = 0, Time taken = 0.0001896 seconds.
```

```
One Edge graph: Minimal extensions = 3, Time taken = 0.0001138 seconds.
Star Directed graph: Minimal extensions = 3, Time taken = 0.00012 seconds.
Star Undirected graph: Minimal extensions = 2, Time taken = 0.0001599 seconds.

Testing for n = 5 vertices:
Sparse graph: Minimal extensions = 1, Time taken = 0.0010244 seconds.
Dense graph: Minimal extensions = 0, Time taken = 0.0006519 seconds.
Random graph: Minimal extensions = 2, Time taken = 0.0006778 seconds.
One Edge graph: Minimal extensions = 4, Time taken = 0.0004026 seconds.
Star Directed graph: Minimal extensions = 4, Time taken = 0.0004148 seconds.
Star Undirected graph: Minimal extensions = 3, Time taken = 0.0006866 seconds.

Testing for n = 6 vertices:
Sparse graph: Minimal extensions = 1, Time taken = 0.0019096 seconds.
Dense graph: Minimal extensions = 0, Time taken = 0.001051 seconds.
Random graph: Minimal extensions = 2, Time taken = 0.0018039 seconds.
One Edge graph: Minimal extensions = 5, Time taken = 0.0006899 seconds.
Star Directed graph: Minimal extensions = 5, Time taken = 0.0005295 seconds.
Star Undirected graph: Minimal extensions = 4, Time taken = 0.0007667 seconds.

Testing for n = 7 vertices:
Sparse graph: Minimal extensions = 1, Time taken = 0.0026771 seconds.
Dense graph: Minimal extensions = 0, Time taken = 0.0007313 seconds.
Random graph: Minimal extensions = 1, Time taken = 0.0019357 seconds.
One Edge graph: Minimal extensions = 6, Time taken = 0.0002674 seconds.
Star Directed graph: Minimal extensions = 6, Time taken = 0.0005769 seconds.
Star Undirected graph: Minimal extensions = 5, Time taken = 0.0004056 seconds.
```

And finally, let's plot the calculation time as a function of the input graph size:

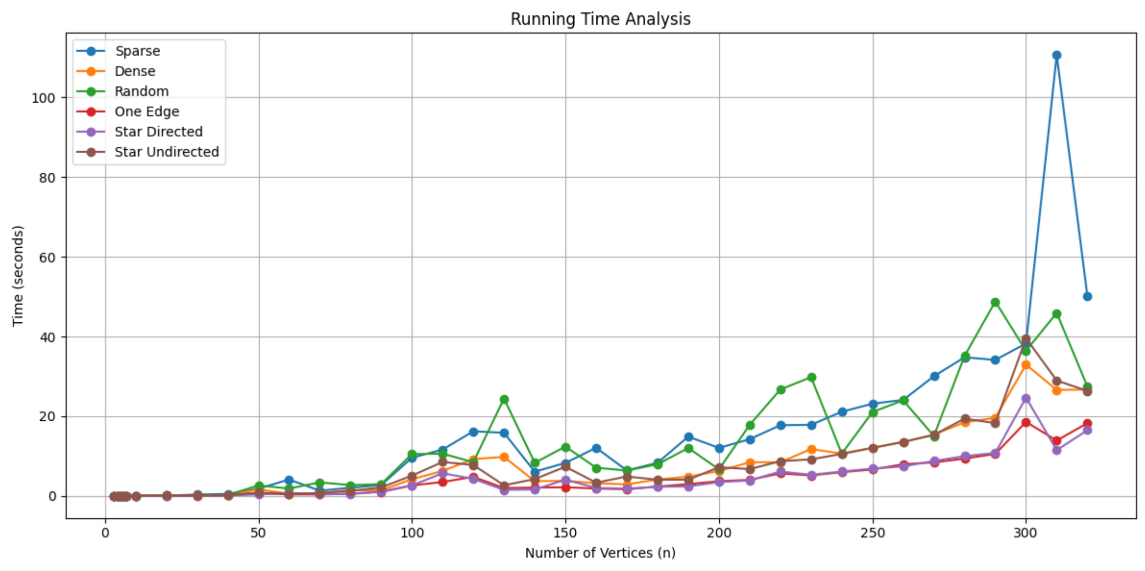


Figure 4.3: Plot of calculation time based on the size of the input

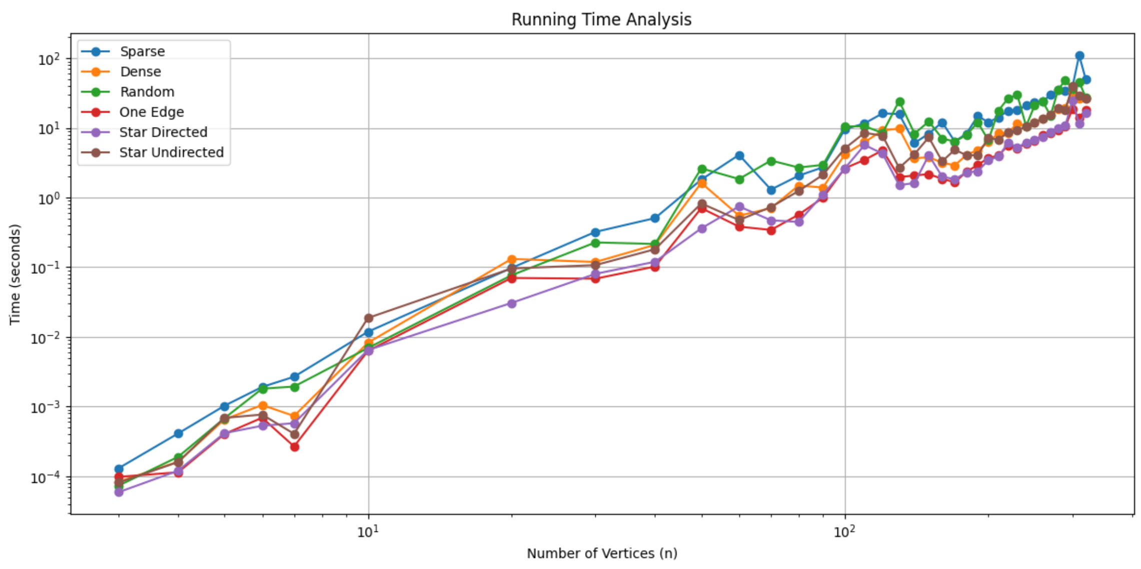


Figure 4.4: Log-log plot of calculation time based on the size of the input

4.3.7 Conclusions

The approximation algorithm for the Hamiltonian completion problem, based on the Iterated 3-opt heuristic, demonstrates several notable characteristics. First, it achieves identical results to the exact algorithm for small graphs ($n \leq 7$), suggesting high accuracy for these cases.

Second, the approximation algorithm demonstrates significantly better running time performance compared to the exact algorithm. For smaller graphs ($n \leq 50$), the execution time remains under 3 seconds across all graph types. As the graph size increases to $n = 300$, the running time grows to around 30-40 seconds for most graph types. The algorithm shows some variability in performance depending on the graph structure - dense graphs and one-edge graphs generally process faster than random graphs, which can take up to 48 seconds for $n = 290$ vertices. Even for the largest tested instances ($n = 320$), the algorithm completes in under 50 seconds for most cases, with sparse graphs occasionally taking longer, as seen in the case of $n = 310$ where it peaked at 110 seconds. This polynomial-time performance makes the approximation algorithm a practical choice for larger instances where the exact algorithm's exponential running time becomes prohibitive

4.4 Number of all Hamiltonian cycles

To determine the number of Hamiltonian cycles in a graph, we use the algorithm from task 3. This algorithm identifies the size of the largest cycle in the graph and counts how many such cycles exist.

- If the size of the largest cycle is smaller than the total number of vertices, then the graph has no Hamiltonian cycles (the number of Hamiltonian cycles is 0).
- If the size of the largest cycle is equal to the number of vertices, then the number of Hamiltonian cycles in the graph is equal to the number of maximal cycles.

References

Gutin, G. and A.P. Punnen, eds. (2007). *The Traveling Salesman Problem and Its Variations.* Springer New York, NY. ISBN: 978-0-387-44459-8. DOI: `10.1007/b101971`.

Kumar, Parveen and Nitin Gupta (July 2014). "A Heuristic Algorithm for Longest Simple Cycle Problem". In: *Int'l Conf. Wireless Networks — ICWN'14*.

Abu-Aisheh, Zeina et al. (Jan. 2015). "An Exact Graph Edit Distance Algorithm for Solving Pattern Recognition Problems". In: DOI: 10.5220/0005209202710278. DOI: `10.5220/0005209202710278`.

Giscard, Pierre-Louis and Paul Rochet (Nov. 2018). "Enumerating Simple Paths from Connected Induced Subgraphs". In: *Graphs and Combinatorics* 34.6, pp. 1197–1202. ISSN: 1435-5914. DOI: `10.1007/s00373-018-1966-9`. URL: `https://doi.org/10.1007/s00373-018-1966-9`.

Giscard, Pierre-Louis, Nils Kriege, and Richard C. Wilson (July 2019). “A General Purpose Algorithm for Counting Simple Cycles and Simple Paths of Any Length”. In: *Algorithmica* 81.7, pp. 2716–2737. ISSN: 1432-0541. DOI: `10.1007/s00453-019-00552-1`. URL: `https://doi.org/10.1007/s00453-019-00552-1`.