Algorithms & Computability Laboratories Report

Aleksandra Wieczorek
Witold Trzeciakowski
Szymon Kupisz

November 2024

Contents

1 Introduction **3**

2 Size of the graph **3**
2.1 Methodology for Calculating Size . . . 3
2.2 Example . . . 4
2.3 Implementation . . . 4
2.4 Conclusion . . . 4

3 Reasonable metrics **4**
3.1 Operations on the matrix . . . 4
3.1.1 Algorithm Description . . . 4
3.1.2 Computational Complexity . . . 5
3.1.3 Justification of Approach . . . 5
3.1.4 Proof . . . 6
3.2 Spectral Metric for Graph Isomorphism . . . 7
3.2.1 Algorithm for Graph Comparison Using Spectrum . . . 8
3.2.2 Computational Complexity . . . 8
3.2.3 Justification of Approach . . . 8
3.3 Time complexity test results . . . 8

4 Length of Maximum Cycle and the number of such cycles in a graph **9**
4.1 Methods of finding the maximum cycle in a graph . . . 9
4.2 The exact solution . . . 9
4.2.1 Algorithm . . . 9
4.2.2 Computational complexity . . . 10
4.3 Heuristic solution . . . 10
4.3.1 Algorithm . . . 10
4.3.2 Computational complexity . . . 11
4.3.3 Heuristic solution limitations . . . 11
4.4 Time complexity tests results . . . 12

5 Extension to Hamiltonian Cycle **13**
5.1 Types of algorithms for checking the existence of Hamilton cycle . . . 13
5.1.1 Exact algorithm . . . 13
5.1.2 Dirac Thorem . . . 14
5.1.3 Spectral Theorem Verification . . . 14
5.2 Solutions to Minimal Hamilton Extension problem . . . 15
5.2.1 DFS approach - non-directed graphs only . . . 15
5.2.2 Dirac theorem approach . . . 16
5.2.3 Greedy approach . . . 17
5.2.4 Spectral Extension for Hamiltonicity - non-directed graphs only . . . 18
5.3 Methods for obtaining number of Hamilton cycles in a graph . . . 19
5.3.1 Exact Method . . . 19
5.4 Heuristic approaches . . . 19
5.4.1 Monte-Carlo Approach . . . 19
5.5 Time complexity test results . . . 19
5.6 Summary of tests . . . 23

References **24**

1 Introduction

Graphs and multigraphs serve as crucial tools in representing and solving complex problems in computer science, mathematics, and related fields. Their applicability spans numerous domains, including network design, data structure optimization, and combinatorial analysis. This report explores the computational aspects of graph analysis through the implementation of algorithms for solving problems related to graph properties, comparisons, and cycles.

The laboratory task required the implementation of algorithms for both basic graphs and advanced multigraphs. The primary focus was to evaluate graph properties such as size, metrics for comparison, maximum cycles, and the minimal extension necessary to achieve Hamiltonian cycles. These tasks involve substantial theoretical challenges, as problems like finding Hamiltonian cycles or extending graphs to Hamiltonian graphs are known to be NP-complete. Consequently, exact solutions are computationally prohibitive for large graphs, necessitating heuristic approaches for practical applications.

Key contributions of this report include:

1. Definitions and justifications for fundamental concepts like graph size, metrics, and cycles.
2. Exact and heuristic algorithms for determining graph properties and solving cycle-related problems.
3. Comprehensive computational complexity analysis to evaluate the feasibility of each approach.
4. Experimental results highlighting the performance of the implemented algorithms under various conditions, including dense and sparse graphs of different sizes.

The report also emphasizes the computational trade-offs between precision and efficiency, particularly when comparing exact solutions with heuristics. The inclusion of advanced methods, such as spectral analysis and Monte Carlo approximations, demonstrates the use of mathematical insights to address computational challenges.

This work bridges theoretical and practical aspects of graph analysis, offering insights into algorithm design, complexity management, and real-world applicability.

2 Size of the graph

In this report, the **size of a graph** is defined as the total number of vertices (V) and edges (E) present in the graph. Specifically, we calculate:

- **Vertices (V)**: The total number of nodes in the graph.
- **Edges (E)**: The total number of connections between pairs of vertices, as represented in the adjacency matrix.

For the adjacency matrix representation of a graph:

- Each cell in the matrix, denoted as $A[i][j]$, indicates whether there is an edge from vertex i to vertex j.
- For undirected graphs, the adjacency matrix is symmetric ($A[i][j] = A[j][i]$), and each edge is represented twice in the matrix (once for each direction).

2.1 Methodology for Calculating Size

The number of edges is computed as the sum of all entries in the adjacency matrix. Normally, for undirected graphs, this sum is divided by two, as each edge is counted twice. However, in this task, we intentionally treat each entry as representing a directed edge. Thus, we count all non-zero entries in the adjacency matrix directly, without dividing by two. The formula for calculating the number of edges (E) is:

$$E = \sum_{i=1}^{V} \sum_{j=1}^{V} A[i][j]$$

where V is the number of vertices.

This approach considers each directed connection as an independent edge, effectively treating the undirected graph as directed. While this choice diverges from traditional definitions for undirected graphs, it simplifies the representation and calculation in context of metric definition in section 3.1

2.2 Example

Consider the adjacency matrix:

$$\begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}$$

- **Number of vertices (V)**: The graph has 4 vertices since the matrix has 4 rows and columns.
- **Number of edges (E)**: Summing all the entries in the adjacency matrix:

$$\text{Total sum} = 0+1+1+0+1+0+0+1+1+0+0+1+0+1+1+0 = 8$$

  Thus, the graph is considered to have **8 edges** under our directed-edge convention.

2.3 Implementation

The computation of graph size is implemented using the following Python function:

```
def size_of_the_graph(matrix):
        num_vertices = len(matrix)
        num_edges = int(sum(sum(row) for row in matrix))
        return num_edges, num_vertices
```

This function takes a 2D list representing the graph's adjacency matrix. It calculate the total number of edges by summing all entries in the adjacency matrix. The number of vertices is calculated as length of the input matrix.

2.4 Conclusion

The chosen definition of graph size as the total number of vertices and directed edges provides a precise and computationally efficient representation. This approach accommodates task-specific requirements and simplifies implementation while maintaining logical consistency.

3 Reasonable metrics

3.1 Operations on the matrix

A *reasonable metric* for comparing two graphs G_1 and G_2 should encapsulate structural differences while adhering to the requirements of metric spaces: non-negativity, symmetry, identity of indiscernibles, and the triangle inequality.

In this task, the metric is defined as a combination of:

1. **Added edges (E)**: The total absolute difference in edge weights between two graphs, after extending the smaller graph's adjacency matrix to match the size of the larger graph.
2. **Added vertices (V)**: The difference in vertex counts between the two graphs.

The metric, represented as $d(G_1, G_2) = (E, V)$, provides an intuitive measure of both structural (edges) and size (vertices) differences. This formulation aligns with practical graph comparison scenarios where adjacency matrices are used, and it supports directed and undirected graphs.

3.1.1 Algorithm Description

The algorithm calculates the metric using the adjacency matrices M_1 and M_2 of G_1 and G_2. The steps include:

1. Identify the smaller matrix (M_{smaller}) and larger matrix (M_{bigger}).
2. Expand M_{smaller} to match the dimensions of M_{bigger}, padding with zeros where necessary.

3. Compute the element-wise difference between the expanded M_{smaller} and M_{bigger} to produce a difference matrix (M_3).

4. Calculate E, the total absolute sum of the elements in M_3, which represents the added edges.

5. Compute V, the difference in vertex counts between the two graphs.

The algorithm is implemented as follows:

```
def calculate_matrix_to_match(matrix1, matrix2):
    if len(matrix1) < len(matrix2):
        M_smaller = matrix1
        M_bigger = matrix2
    else:
        M_smaller = matrix2
        M_bigger = matrix1

    rows_smaller = len(M_smaller)
    cols_smaller = len(M_smaller[0]) if rows_smaller > 0 else 0

    rows_bigger = len(M_bigger)
    cols_bigger = len(M_bigger[0]) if rows_bigger > 0 else 0

    expanded_M_smaller = [[0 for _ in range(cols_bigger)] for _ in range(rows_bigger)]

    for i in range(rows_smaller):
        for j in range(cols_smaller):
            expanded_M_smaller[i][j] = M_smaller[i][j]

    M3 = [[M_bigger[i][j] - expanded_M_smaller[i][j]
        for j in range(cols_bigger)]
        for i in range(rows_bigger)]

    M3_abs = [[abs(M_bigger[i][j] - expanded_M_smaller[i][j]) for j in range(cols_bigger)] for i in
    added_E = sum(sum(row) for row in M3_abs)
    added_V = len(M_bigger) - len(M_smaller)

    return M3, (added_E, added_V)
```

3.1.2 Computational Complexity

- **Matrix Padding and Expansion**: $O(n^2)$, where n is the size of the larger graph's adjacency matrix.
- **Matrix Difference Calculation**: $O(n^2)$ for computing M_3.
- **Summation of Absolute Differences**: $O(n^2)$.

Thus, the overall complexity is $O(n^2)$, which is efficient for typical graph sizes encountered in practice.

3.1.3 Justification of Approach

This metric is practical because:

- **Scalability**: It is computable for large graphs within reasonable time limits.
- **Interpretability**: The results (E, V) clearly indicate the structural and size disparities between graphs.
- **Flexibility**: The method accommodates both directed and undirected graphs and handles missing data effectively through zero-padding.

3.1.4 Proof

To prove that $d(A, B)$ is a metric on the space of matrices, we need to verify the four axioms of a metric:

1. **Non-negativity and identity of indiscernibles:** $d(A, B) \geq 0$ and $d(A, B) = 0 \iff A = B$,
2. **Symmetry:** $d(A, B) = d(B, A)$,
3. **Triangle inequality:** $d(A, C) \leq d(A, B) + d(B, C)$.

1. Non-negativity

For any matrices A and B:

$$\sum |a_{ij} - b_{ij}| \geq 0 \quad \text{and} \quad |\dim(A - B)| \geq 0,$$

since the absolute value is always non-negative. Therefore:

$$d(A, B) = \sum |a_{ij} - b_{ij}| + |\dim(A - B)| \geq 0.$$

Identity of indiscernibles

If $A = B$, then:

$$\sum |a_{ij} - b_{ij}| = 0 \quad (\text{because } a_{ij} = b_{ij} \text{ for all } i, j),$$

and

$$|\dim(A - B)| = 0 \quad (\text{because the dimensions of } A \text{ and } B \text{ are identical}).$$

Thus:

$$d(A, B) = 0.$$

Conversely, if $d(A, B) = 0$, then:

$$\sum |a_{ij} - b_{ij}| = 0 \quad \Longrightarrow \quad a_{ij} = b_{ij} \ \forall i, j,$$

and

$$|\dim(A - B)| = 0 \quad \Longrightarrow \quad \dim(A) = \dim(B).$$

Therefore, $A = B$.

2. Symmetry

$$d(A, B) = \sum |a_{ij} - b_{ij}| + |\dim(A - B)| = \sum |b_{ij} - a_{ij}| + |\dim(B - A)| = d(B, A).$$

3. Triangle inequality

For any matrices A, B, and C:

$$d(A, C) = \sum |a_{ij} - c_{ij}| + |\dim(A - C)|.$$

Consider the following:

1. From the triangle inequality for the absolute value :

$$|a_{ij} - b_{ij}| + |b_{ij} - c_{ij}| \geq |a_{ij} - c_{ij}|$$

Summing over all elements of the matrices:

$$\sum |a_{ij} - b_{ij}| + \sum |b_{ij} - c_{ij}| \geq \sum |a_{ij} - c_{ij}|$$

2. For the dimensions:

$$|\dim(A - C)| \leq |\dim(A - B)| + |\dim(B - C)|,$$

Adding these two inequalities, we get:

$$d(A, C) \leq \big(\sum |a_{ij} - b_{ij}| + |\dim(A - B)| \big) + \big(\sum |b_{ij} - c_{ij}| + |\dim(B - C)| \big).$$

Thus:

$$d(A, C) \leq d(A, B) + d(B, C).$$

Conclusion

The function $d(A, B) = \sum |a_{ij} - b_{ij}| + |\dim(A - B)|$ satisfies all four axioms of a metric, and therefore it is a metric on the space of matrices.

3.2 Spectral Metric for Graph Isomorphism

Spectral graph theory is a well-established mathematical discipline that studies the properties of graphs using the eigenvalues of their adjacency matrices or Laplacian matrices. The spectrum of a graph is the set of eigenvalues of its adjacency matrix. Two isomorphic graphs must have the same spectrum; however, the converse is not always true. Spectral properties provide valuable insights into graph structure, making the spectrum a useful tool for comparing graphs and detecting potential isomorphism.

The QR algorithm is a numerical method for finding the eigenvalues of a square matrix. It iteratively decomposes the matrix into a product of an orthogonal matrix (Q) and an upper triangular matrix (R), then recombines them to form a new matrix. Over successive iterations, this process converges to a diagonal matrix whose diagonal entries are the eigenvalues of the original matrix.

Steps of the QR Algorithm

1. Start with the given matrix A.
2. Perform the QR decomposition to obtain Q and R such that $A = QR$.
3. Compute $A' = RQ$.
4. Repeat the decomposition and recombination steps until A' converges to a diagonal matrix.
5. The eigenvalues of A are the diagonal elements of the converged matrix.

Algorithm Implementation

```
def gram_schmidt(A, tolerance=1e-10):
    n = len(A)
    Q = [[0] * n for _ in range(n)]
    R = [[0] * n for _ in range(n)]

    for j in range(n):
        v = [A[i][j] for i in range(n)]
        for i in range(j):
            R[i][j] = sum(Q[k][i] * A[k][j] for k in range(n))
            v = [v[k] - R[i][j] * Q[k][i] for k in range(n)]
        R[j][j] = sum(v[k] ** 2 for k in range(n)) ** 0.5
        if abs(R[j][j]) > tolerance:
            for k in range(n):
                Q[k][j] = v[k] / R[j][j]
        else:
            R[j][j] = 0
            for k in range(n):
                Q[k][j] = 0
    return Q, R

def qr_algorithm(matrix, max_iterations=1000, tolerance=1e-10):
    n = len(matrix)
    A = [row[:] for row in matrix]
    for _ in range(max_iterations):
        Q, R = gram_schmidt(A, tolerance)
        A_next = matrix_multiply(R, Q)
        if all(abs(A_next[i][j] - A[i][j]) < tolerance for i in range(n) for j in range(n)):
            break
        A = A_next
    return [round(A[i][i], 2) for i in range(n)]
```

3.2.1 Algorithm for Graph Comparison Using Spectrum

The method for comparing two graphs G_1 and G_2 is as follows:

1. Compute the degree sequence of both graphs. If the sequences differ, the graphs cannot be isomorphic.
2. If the degree sequences match, calculate the eigenvalues of their adjacency matrices using the QR algorithm.
3. Compare the sorted eigenvalue sets. If they match, the graphs may be isomorphic; otherwise, they are not.

```
def can_be_isomorphic(M1, M2):
    M1_degs = sorted([sum(row) for row in M1])
    M2_degs = sorted([sum(row) for row in M2])
    if M1_degs != M2_degs:
        return False
    return True

def are_spectra_equal(adj_matrix1, adj_matrix2):
    if can_be_isomorphic(adj_matrix1, adj_matrix2):
        eigenvalues1 = qr_algorithm(adj_matrix1)
        eigenvalues2 = qr_algorithm(adj_matrix2)
        return sorted(eigenvalues1) == sorted(eigenvalues2)
    else:
        return False
```

3.2.2 Computational Complexity

- **Degree Sequence Comparison**: $O(n^2)$ for adjacency matrix row summation.
- **QR Algorithm**: $O(n^3)$ per iteration, with typical convergence in a fixed number of iterations.

Overall, the complexity is $O(n^3)$, dominated by the eigenvalue calculation step.

3.2.3 Justification of Approach

This spectral method is advantageous because:

- It leverages structural invariants of graphs, making it robust for comparison.
- It provides a computationally efficient means to rule out non-isomorphic graphs early.

3.3 Time complexity test results

Figure 10 demonstrates the steep time increase for the spectral metric, while Figure 2 shows the more gradual growth for matrix operations. This highlights the significant difference in computational performance between the two methods.

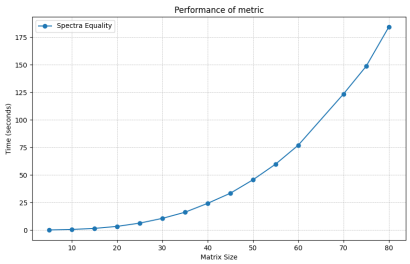

Figure 1: Time complexity for spectral equality

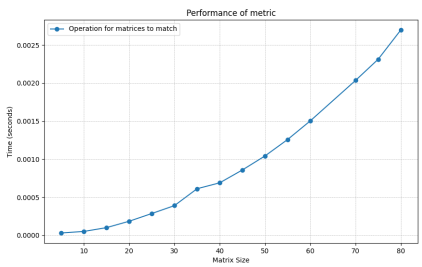

Figure 2: Time complexity for matrix operations.

4 Length of Maximum Cycle and the number of such cycles in a graph

4.1 Methods of finding the maximum cycle in a graph

The problem of finding the maximum cycle in a graph is the NP-complete one, as in the case of graph G that has the maximum cycle in a graph of length $|V| - 1$, where V is the number of vertices, then the problem of finding the maximum cycle in a graph narrows down to the problem of finding a hamiltonian cycle, which is the NP-complete problem. Therefore, the exact solution for this problem cannot provide the results in polynomial time. Hence, to obtain satisfactory computational complexity, a heuristic solution is necessary.

4.2 The exact solution

4.2.1 Algorithm

For the exact solution, we decided to use the DFS-based approach, as this algorithm is a well-known one and it has already been proven that it is valid.

At first, let us discuss the DFS algorithm implemented in the exact solution:

Input data: current node, start node, visited nodes, current path
Moreover, before the declaration of this function, the array for storing all of the maximum cycles has to be declared. Let us name that array as *paths*.

1. Initialize the variable for storing the maximum length of a cycle
2. Mark the current node as visited
3. Add the current node to the current path
4. For all the neighbours of the current node:
   (a) If there is an edge from the current node to the analyzed neighbour node, perform the following steps:
      i. If the neighbour node is the start node and the length of the path is greater than 2:
         A. If the length of the current path is greater than the current cycle length:
            - Set the maximum cycle's length to the current cycle length
            - Clear the *paths* array
            - Add the current path to the *paths* array
         B. Else, if the maximum length of the cycle is equal to the current cycle length:
            - Add the current cycle to the *paths* array
      ii. Else, if the current neighbour node has not been visited yet,
         A. Call this function with the following input data: the neighbour node, the starting node, visited nodes and the path
5. Mark the current node as not visited
6. Delete the current node from the path

Let us name the above function as dfs and move to the discussion of the general algorithm for the exact solution, which is denoted as follows:

1. Initialize the *paths* array.
2. For all of the vertices of the adjacency matrix:
   (a) Create the array of n false boolean values, where n is the number of vertices and name this array *visited*

(b) Perform the dfs function with the following input data: currently checked vertex, currently checked vertex, *visited* array, and an empty array

3. Transform the *paths* array to such a state that no cycle is repeated

4. Return *paths*

4.2.2 Computational complexity

Now, let us take a look upon the computational complexity of the DFS-based approach.

To start with, let us consider the dfs method. If the algorithm starts from a certain vertex, say v, it has to check all of the connections with the other vertices, which will take $O(n)$ time, where n is the number of vertices. For each of the vertices that has not been visited yet, we can perform the recursive call of the dfs function. In the worst case, which is the complete graph, for the first vertex analyzed we have $n-1$ possibilities of calling the dfs function. For the second one, $n-2$, and so on. Therefore, the computational complexity of this part of the dfs function is $O(n!)$, hence the computational complexity of the dfs function is $O(n \cdot n!)$.

Now, let us consider the entire exact solution algorithm. First, we initialize the empty *paths* array and set the variable for storing the maximum length of a cycle, which are both the $O(1)$ operations. After that, we call the dfs function for all of the vertices, which has the computational complexity of $O(n^2 \cdot n!)$, where n is the number of vertices. Next, in case of our implementation, where we implemented the *paths* array as the set in order to manage this array, we have to map it to the actual array, which takes additional $O(n)$ time. After that, it is time to take a look at the process of making sure that no cycle repeats in the array. First, we initialize the *result* table and the *seen* set, for storing the cycles sorted by the number of the vertex from 1 to n, which will take $O(1)$ time. After that, we go through all of the cycles.

Now, we have to distinguish whether the graph considered is a simple graph or a multigraph.
For a simple graph, the worst possible length of the *paths* array is $O(n!)$ for the complete graph. For each of the combination, the algorithm has to check if the given cycle is not already in the final array. To check it, it firstly sorts the array with the in-built python function *sorted*, which sorts the array with the use of divide-and-conquer algorithm and therefore takes $O(n \cdot log(n))$ time, where n is the number of vertices. After the sorting, the algorithm checks if this sorted version of the cycle is stored in the *seen* set. This will take $O(m \cdot n)$ in the worst case, where m is the number of stored cycles in the *seen* set and n is the number of vertices in a cycle. If the cycle has not been placed in the *seen* set yet, we can put the sorted version to it and the original one to the *paths* array. Both of those operations will take $O(1)$ time.

Hence, for a simple, the computational complexity is $O(n^2 \cdot n! + n! \cdot n \cdot log(n) \cdot m \cdot n) = O(n^2 \cdot n! \cdot log(n) \cdot m)$. However, as we have the $n!$ possible paths generated by the dfs function, we can assume that m will be strictly smaller than $n!$ and therefore omit it. Hence, the computational complexity for the simple graph is $O(n^2 \cdot n! \cdot log(n))$.

For a multigraph, if there are two or more edges connecting vertices v_i and v_j, we will consider any choice of the edge equivalent to each other, since the algorithm is focused on finding the order of the vertices in a cycle, and as the edges are unweighted, we may assume that choosing any of them is practically equivalent. Therefore, the computational complexity for the multigraph is also $O(n^2 \cdot n! log(n))$.

4.3 Heuristic solution

4.3.1 Algorithm

As the graph analyzed, say G, is provided in form of the adjacency matrix, one can use the matrix exponentiation technique in order to create the heuristic algorithm solving this problem. The algorithm is given as follows:

1. For all $k = 1, 2, ..., k$, calculate G^k matrix

2. For each G^k, sum all of the occurrences of any value greater than zero for $G[i][i], i = 1, 2, ..., n$, where n is the width/height of the matrix G^k. Let us name that sum as S, hence

$$S = \sum_{i=1}^{n} sgn(G^k[i][i])$$

3. If $S > 0$, then save the S and k values. In case the graph G is the undirected one, divide S by 2 before the saving in order to get rid of the doubling cycles.

4.3.2 Computational complexity

To start with, let us discuss the most crucial part of the heuristic method presented, which is matrix exponentiation. Our solution involves two techniques in order to obtain the $k - th$ power of the G adjacency matrix, we are using the following methods:

1. **Exponentiation by squaring**
   The Exponentiation by squaring is a method for fast computations of the matrix powers. To do so, it is assumed that

   (a) $x^n = x \cdot (x^2)^{((n-1)/2)}$, if n is odd
   (b) $x^n = (x^2)^{(n/2)}$ if n is even

   Basing on those assumptions of the recursion, we are capable of obtaining the G^n in $O(log(n))$ complexity.

2. **Naive method of matrix multiplication**
   The method of the matrix multiplication that was implemented in the algorithm is the so-called "Naive" method. According to it, we perform the usual matrix multiplication and therefore obtain the result. At first, we also create the table for storing the results, which takes $O(n^2)$ time. However, the computational complexity of the multiplication will be higher than that and will reach the $O(n^3)$ computational complexity, as to iterate through all of the possible combinations of the first matrix's rows and the second matrix's columns, the $O(n^2)$ complexity is required, and for calculating the result of the multiplication the additional $O(n)$ operation has to be performed, nested in the loops generating the combinations mentioned above. Hence, the matrix multiplication algorithm computational complexity is $O(n^3)$

Therefore, the computational complexity of the algorithm for obtaining G^n is $O(n^3 log(n))$.

Now, let us consider the final complexity of this algorithm. First, we check whether the graph G is directed or undirected, which takes $O(n^2)$ time. Then, we iterate through all possible lengths of the cycle, which takes $O(n)$ time. In this loop, we generate the G^k matrix, which is $O(n^3 log(n))$. Afterwards, we have to sum up the diagonals of the G^k matrix, which takes $O(n)$ time. After that, we perform the division by 2 in case the graph is undirected, and if the number of cycles is greater than 0, we save the number of cycles and the number k to a special variable. All of those operations have the complexity of $O(1)$.

Hence, in the end, the computational complexity of the heuristic algorithm implemented is $O(n^2 + n \cdot (n^3 log(n) + n) = O(n^4 log(n))$.

4.3.3 Heuristic solution limitations

Even though the solution can approximate the solution, it has indeed some limitations. During the research conducted by our team, it was revealed that the matrix exponentiation method is working unpredictably for the acyclic graphs.
To prove this, let us consider a family of graphs being a 1-tree i.e. a trees such that each node has at most one leaf.
Then, let us consider the graph given by the following adjacency matrix, say A:

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

As it can be deducted from the fact that it is a 1-tree, it has no cycles. Now, let us consider the consecutive powers of the A matrix:

1. A^2:
$$\begin{bmatrix} 2 & 0 & 3 & 0 \\ 0 & 5 & 0 & 3 \\ 3 & 0 & 5 & 0 \\ 0 & 3 & 0 & 2 \end{bmatrix}$$
2. A^3:
$$\begin{bmatrix} 0 & 5 & 0 & 3 \\ 5 & 0 & 8 & 0 \\ 0 & 8 & 0 & 5 \\ 3 & 0 & 5 & 0 \end{bmatrix}$$
3. A^4:
$$\begin{bmatrix} 5 & 0 & 8 & 0 \\ 0 & 13 & 0 & 8 \\ 8 & 0 & 13 & 0 \\ 0 & 8 & 0 & 5 \end{bmatrix}$$

As it can be seen, in this case the algorithm will return the positive length of the cycle, which is obviously the wrong solution. This is due to the fact that the matrix exponentiation technique can provide only the number of so-called walks, not the exact cycles. This means that the vertices can repeat, that's why instead of summing the diagonal of the G^k matrix, we decided to sum only the occurences of values greater than zero on this diagonal for better accuracy of the result. Therefore, in order to avoid such situations, we assume that the graph given by the adjacency matrix A is not acyclic, because otherwise the results will be based on the number of vertices and is henceforth unpredictable.

4.4 Time complexity tests results

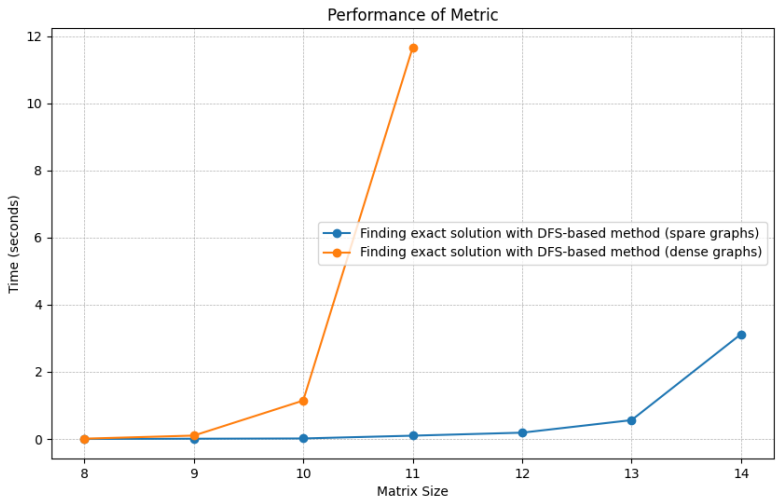

Figure 3: Comparison of the average execution time of the DFS-based approach for spare and dense graphs

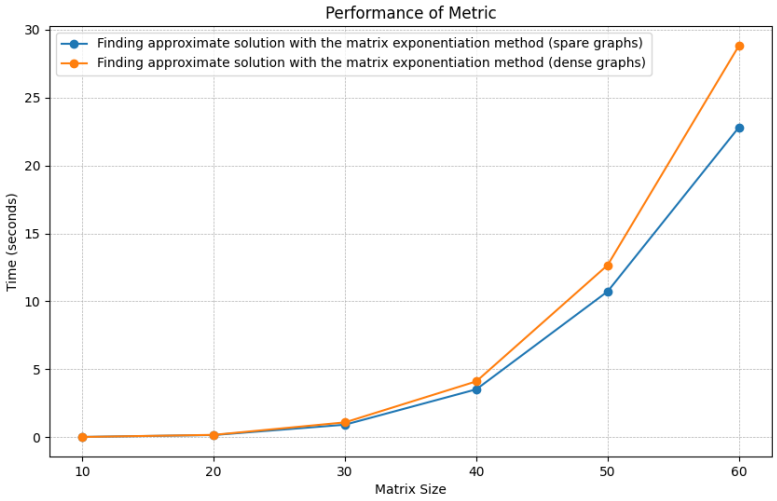

Figure 4: Matrix exponentiation average execution time comparison between the exact solution and the heuristic one

Note: In case of the DFS-based solution, for the number of vertices being 14 and bigger, due to the time complexity rising drastically the solution was terminated each time by the operating system.

Tests summary

As we can see, it is obvious that the matrix exponentiation method is much faster when it comes to finding the approximation of the maximum cycle length and the number of such cycles than finding the exact solution with the DFS-based approach.

5 Extension to Hamiltonian Cycle

The problem of extending a graph G to a Hamiltonian graph is NP-complete. The task of extending G involves adding the minimum number of edges such that the resulting graph contains a Hamiltonian cycle, which inherently encapsulates the Hamiltonian cycle problem. Consequently, finding an exact solution to the Hamiltonian graph extension problem cannot be achieved in polynomial time unless $P = NP$. Due to this, heuristic or approximation methods are essential to achieve practical results within reasonable computational limits. For this task we will be creating a single graph from a multigraph since it doesn't change the Hamiltoniacity. The algorithm replaces elements on diagonal with zeros, and reduces multiple edges between a pair of vertices to 1 edge. This takes $O(n^2)$ operations, where n denotes number of vertices.

5.1 Types of algorithms for checking the existence of Hamilton cycle

First, let's focus on algorithms that will determine whether the graph is Hamiltonian

5.1.1 Exact algorithm

The algorithm employs a backtracking approach to determine if a graph contains a Hamiltonian cycle. The algorithm Is presented below

1. **Initialization:** Start with a path initialized to vertex 0.
2. **Recursive Exploration:** Add vertices to the path one by one
3. **Feasibility Check:** A vertex v is added if:
    - There is an edge between the last vertex in the path and v.

- v is not already in the path.

4. **Cycle Completion:** if $len(path) = n$ Verify if the last vertex connects back to the first to form a cycle.

5. **Backtracking:** If no valid vertex can be added, backtrack and try other options.

Time Complexity: The algorithm explores $O(n!)$ paths, each requiring $O(n)$ checks, leading to a worst-case complexity of $O(n \times n!)$.

5.1.2 Dirac Thorem

Dirac's theorem applies primarily to undirected graphs but can be extended to directed graphs with modifications. Those were shown during Discrete mathematics course. The key differences in implementation are as follows:

1. **Degree Consideration:**

   - For **undirected graphs**, the degree of a vertex is simply the number of edges incident to it. The condition requires that the degree of every vertex satisfies:

     $$\deg(v) \geq \frac{n}{2}, \quad \forall v \in V$$

     where n is the number of vertices in the graph.
   - For **directed graphs**, each vertex has an *in-degree* and *out-degree*. The condition is modified to ensure both satisfy:

     $$\min(\text{in-deg}(v), \text{out-deg}(v)) \geq \frac{n}{2}, \quad \forall v \in V$$

2. **Steps:**

   (a) Check if $n < 3$. If true, return **False**, as Dirac's theorem does not apply to graphs with fewer than 3 vertices.

   (b) For each vertex v in the graph:
   - If the graph is undirected:
     - Compute the degree $\deg(v)$ as the sum of the v-th row in the adjacency matrix.
     - Check if $\deg(v) \geq n/2$. If any vertex fails this condition, return **False**.
   - If the graph is directed:
     - Compute the *in-degree* and *out-degree* of v by summing all edges coming into and out of the vertex.
     - Check if $\min(\text{in-deg}(v), \text{out-deg}(v)) \geq n/2$. If any vertex fails this condition, return **False**.

   (c) If all vertices satisfy the conditions, return **True**.

3. **Complexity Analysis**

4. **Time Complexity:** $O(n^2)$

   - Computing the degree of a vertex requires summing up n elements in the adjacency matrix, which takes $O(n)$ time.
   - This computation is repeated for all n vertices, resulting in $O(n^2)$.

5.1.3 Spectral Theorem Verification

This function is designed to verify whether a graph satisfies certain spectral conditions derived from graph spectral theory. It is based on the following theorem from the work by Ning and Ge (2014)[1]: Let G be a graph with minimum degree δ. The spectral radius of G, denoted by $\rho(G)$, is the largest eigenvalue of the adjacency matrix of G. The following results hold:

- Let G be a graph on $n \geq 4$ vertices with $\delta \geq 1$. If $\rho(G) > n-3$, then G contains a Hamilton path unless $G \in \{K_1 \vee (K_{n-3} + 2K_1), K_2 \vee 4K_1, K_1 \vee (K_{1,3} + K_1)\}$.
- Let G be a graph on $n \geq 14$ vertices with $\delta \geq 2$. If $\rho(G) \geq \rho(K_2 \vee (K_{n-4} + 2K_1))$, then G contains a Hamilton cycle unless $G = K_2 \vee (K_{n-4} + 2K_1)$.

1. **Steps:**
   (a) Compute the largest eigenvalue, denoted $\lambda_{\max}$, of the graph's adjacency matrix.
   (b) **Condition 1:** Check if $\lambda_{\max} > n-3$, where n is the number of vertices in the graph. If true:
      - Identify the graph structure using the function `identify_graph`.
      - Ensure the graph is not one of the exceptional cases:

$$\text{Exceptional cases} = \{K_1 \vee (K_{n-3} + 2K_1), K_2 \vee 4K_1, K_1 \vee (K_{1,3} + K_1)\}$$

      - If the graph is not an exceptional case, return **True** (indicating that the graph contains a Hamilton path).
   (c) **Condition 2:** Check for graphs with $\lambda_{\max} \geq \lambda_{\max}(K_2 \vee (K_{n-4} + 2K_1))$, where $K_2 \vee (K_{n-4} + 2K_1)$ is generated using the function `generate_k2_join_kn4_plus_2k1`.
      - Compute $\lambda_{\max}(K_2 \vee (K_{n-4} + 2K_1))$, the largest eigenvalue of the exceptional graph.
      - Check if $\lambda_{\max} \geq \lambda_{\max}(K_2 \vee (K_{n-4} + 2K_1))$ and the graph is not isomorphic to $K_2 \vee (K_{n-4} + 2K_1)$.
      - If true, return **True** (indicating that the graph contains a Hamilton cycle).
   (d) If neither condition is satisfied, return **False** (indicating that the graph does not meet the spectral conditions for Hamiltonian properties).

2. **Complexity Analysis:**
   - Computing the largest eigenvalue $\lambda_{\max}$ using Power iteration takes $O(num\_of\_iterations \cdot n^2)$, where n is the number of vertices. We set the num_of_iterations for Power iteration method as 10.
   - The graph identification and exceptional graph generation involve structural checks, which are $O(n^2)$ for adjacency matrix-based representations.
   - Overall, the complexity is dominated by the eigenvalue calculation, resulting in:

$$O(num\_of\_iterations \cdot n^2)$$

3. **Drawbacks:** Method can sometimes produce false positives (during testing we found that in most cases the condition one is enough for Hamiltonian cycle assistance, yet the theorem states only existence of Hamilton path). Also, it's hard and computationally expensive to check isomorphism with an algorithm.

5.2 Solutions to Minimal Hamilton Extension problem

The algorithms described below, are our proposed solutions to minimal extension problem.

5.2.1 DFS approach - non-directed graphs only

This function aims to modify a given undirected graph by adding the minimum number of edges required to form a Hamiltonian cycle. The approach leverages the Union-Find (disjoint-set) data structure to ensure connectivity and degree conditions to complete the cycle.

1. **Steps:**
   (a) **Initialize the Union-Find structure:**
      - For each vertex v, create a set in the Union-Find structure using `make_set`.
   (b) **Connect existing edges:**

- For each pair (i, j) where graph$[i][j] = 1$, perform a `union` operation to merge the sets containing i and j.

(c) **Ensure connectivity:**

- Identify connected components using the `find` operation.
- For every pair of components, add an edge between their representative vertices to connect all components into a single connected component.

(d) **Adjust degrees to satisfy Hamiltonian properties:**

- Compute the degree of each vertex.
- Maintain a list of vertices with degree < 2. Iteratively pair such vertices and add edges between them until no vertex has a degree < 2.
- If a single vertex with degree < 2 remains, attempt to connect it to any suitable vertex to satisfy degree requirements.

(e) **Construct the Hamiltonian path:**

- Perform a Depth-First Search (DFS) starting from vertex 0 to construct a potential Hamiltonian path.

(f) **Close the cycle:**

- Check if the last vertex in the path is adjacent to the first. If not, add an edge to close the cycle.

2. **Complexity Analysis:**

- **Union-Find Operations:** Adding and finding edges takes $O(\alpha(n))$ per operation, where $\alpha(n)$ is the inverse Ackermann function.
- **Degree Adjustments:** Calculating vertex degrees and adjusting them requires $O(n^2)$, as it involves processing the adjacency matrix.
- **DFS for Hamiltonian Path:** The DFS traversal has a time complexity of $O(n^2)$ for adjacency matrix representations.
- **Overall Complexity:** Dominated by the adjacency matrix processing, resulting in:

$$O(n^2)$$

- **Drawbacks:** Unfortunately, due to the fact that the Union-find operation is unsuitable for directed graphs this approach works for undirected ones.

5.2.2 Dirac theorem approach

This function modifies a given graph by adding the minimal number of edges necessary to satisfy Dirac's theorem. The theorem states that a simple graph with $n \geq 3$ vertices is Hamiltonian if every vertex has a degree $\geq \frac{n}{2}$.

1. **Input:**

- adj_matrix$[i][j]$: An $n \times n$ adjacency matrix representing the graph.
- directed : A boolean indicating whether the graph is directed (**True**) or undirected (**False**).

2. **Steps:**

(a) **Initialize Degree Information:**

- For a directed graph:
  - Maintain a dictionary `degrees` to store a tuple (out-degree, in-degree) for each vertex.
  - Compute the degrees by iterating over the adjacency matrix.
- For an undirected graph:
  - Maintain a dictionary `degrees` to store the degree of each vertex.
  - Compute the degrees by iterating over the adjacency matrix.

(b) **Define the Edge Addition Subroutine:**

- Implement `add_edge(adj_matrix, u, v, directed)`:
  - For a directed graph:
    - Add a directed edge from u to v in the adjacency matrix.
    - Update the `degrees` dictionary for u (increment out-degree) and v (increment in-degree).
  - For an undirected graph:
    - Add an edge between u and v in the adjacency matrix.
    - Update the `degrees` dictionary for both u and v.

(c) **Add Edges to Satisfy Dirac's Theorem:**

- Iterate over all pairs of vertices (u, v) where $u \neq v$ and there is no edge between them (adj_matrix$[u][v] = 0$).
- For directed graphs:
  - Check if out-degree$(u) < \frac{n}{2}$ or in-degree$(v) < \frac{n}{2}$.
  - If either condition holds, add a directed edge from u to v.
- For undirected graphs:
  - Check if degree$(u) < \frac{n}{2}$ or degree$(v) < \frac{n}{2}$.
  - If either condition holds, add an edge between u and v.

3. **Output:**

- An updated adjacency matrix where the minimal number of edges have been added to satisfy Dirac's theorem.

4. **Complexity Analysis:**

- Computing degrees requires $O(n^2)$, as it involves summing rows or columns of the adjacency matrix.
- Adding edges requires iterating over all $O(n^2)$ pairs of vertices.
- Updating degrees and the adjacency matrix is $O(1)$ for each edge addition.
- Overall complexity is:

$$O(n^2)$$

5.2.3 Greedy approach

This function modifies a graph represented by an adjacency matrix to ensure that it contains a Hamiltonian cycle, by adding the minimal number of edges as necessary.

5. **Algorithm:**

(a) **Initialization:**

- Let n be the number of vertices in the graph.
- Initialize an empty set visited to keep track of visited vertices.
- Initialize an empty list path to store the Hamiltonian path being constructed.
- Compute the degree of each vertex as degrees$[i] = \sum_{j=0}^{n-1}$ adj_matrix$[i][j]$.
- Select the vertex with the maximum degree as the starting vertex current_node.
- Add current_node to path and mark it as visited.

(b) **Iteratively Construct the Path:**

- While the length of path is less than n:
  i. Identify all neighbors of current_node from the adjacency matrix.
  ii. Filter the neighbors to find those not yet visited (unvisited_neighbors).
  iii. If unvisited neighbors exist:
    - Select the neighbor with the highest degree as the next vertex next_node.
  iv. Otherwise:
    - Identify all vertices not yet visited (unvisited_nodes).

- Select the vertex with the highest degree as next_node.
- Add an edge between current_node and next_node in the adjacency matrix.

v. Update current_node, mark next_node as visited, and append it to path.

(c) **Close the Cycle:**

- If the last vertex in path is not connected to the first vertex:
  - Add an edge between these two vertices to close the Hamiltonian cycle.

6. **Complexity Analysis:**

- Calculating the degrees of all vertices requires $O(n^2)$.
- Constructing the Hamiltonian path involves:
  - Finding neighbors, which requires $O(n)$ per vertex.
  - Iterating over all n vertices, leading to $O(n^2)$.
- Adding edges is $O(1)$ for each operation.
- Overall complexity:

$$O(n^2)$$

5.2.4 Spectral Extension for Hamiltonicity - non-directed graphs only

This section discusses the spectral extension approach for modifying a graph to satisfy Hamiltonian conditions based on its spectral properties. It also outlines a method to verify Hamiltonicity using spectral theorems.

- **Algorithm for Spectral Extension:**

1. **Initialization:**
   - Compute the **largest eigenvalue** (spectral radius) ρ of the adjacency matrix.
   - Identify the number of vertices n.
2. **Adding edges:**
   - While $\rho < n - 3$:
     (a) add edges iteratively, prioritizing pairs of vertices that maximize the increase in ρ.
     (b) Recalculate ρ after each edge addition.
     (c) Break the loop if the number of iterations exceeds n^2 to prevent infinite execution or when spectral conditions described previously are satisfied.

- **Complexity Analysis:**
  - Eigenvalue Computation:
    - Calculating the spectral radius of an adjacency matrix requires $O(num\_of\_iterattions \cdot n^2)$.
  - Greedy Edge Addition:
    - Adding an edge requires $O(1)$, but recalculating the spectral radius after each addition incurs $O(n^3)$.
    - For $O(n^2)$ iterations, the total cost is $O(n^5)$.
  - Overall Complexity:

$$O(n^4 \cdot num\_of\_iterattions)$$

- **Diagnostic Cases:**
  - Graph matches an exceptional class (e.g., $K2 \vee 4K1$): Spectral conditions cannot guarantee Hamiltonian properties.
  - Spectral radius condition satisfied ($\rho \geq n - 3$): Hamiltonicity is likely, and further checks confirm the presence of cycles or paths.

5.3 Methods for obtaining number of Hamilton cycles in a graph

5.3.1 Exact Method

Since the Hamilton cycle is a cycle we decided to use the same method as the one described in the previous section.

5.4 Heuristic approaches

Since the Hamilton cycle is a cycle we decided to use the same methods as the one described in the previous section. Though here we will present one statistical method

5.4.1 Monte-Carlo Approach

The function estimates the number of Hamiltonian cycles in a graph using a Monte Carlo simulation based on its adjacency matrix.

- **Algorithm:**
  1. **Initialization:**
     - Let n be the number of vertices in the graph.
     - Initialize hamiltonian_count = 0, to track the number of valid Hamiltonian cycles found.
  2. **Helper Function:** Define `is_hamiltonian_cycle(path)`:
     - For a given path $[v_1, v_2, \ldots, v_n, v_1]$, check if each consecutive pair of vertices is connected:
     $$\forall i \in \{1, \ldots, n\}, \quad \text{adj\_matrix}[v_i][v_{i+1}] = 1,$$
     where $v_{n+1} = v_1$.
  3. **Monte Carlo Simulation:** Perform the following for iterations samples:
     (a) Generate a random permutation of vertices, path = $[v_1, v_2, \ldots, v_n, v_1]$.
     (b) Check if path forms a Hamiltonian cycle using `is_hamiltonian_cycle(path)`.
     (c) If it is a Hamiltonian cycle, increment hamiltonian_count.
  4. **Estimate Total Hamiltonian Cycles:** Compute the estimate:
     $$\text{estimated\_cycles} = \frac{\text{hamiltonian\_count}}{\text{iterations * n * 2}} \times n!,$$
     where $n!$ is the total number of permutations of n vertices.
- **Complexity Analysis:**
  - Generating a random permutation and appending the starting vertex takes $O(n)$ per iteration.
  - Verifying a Hamiltonian cycle takes $O(n)$ per path.
  - With $O(n^2)$ iterations by default, the overall complexity is:
  $$O(n^3).$$
- **Advantages:**
  - Efficient for small graphs where exhaustive enumeration is computationally expensive.
  - Adjustable accuracy by increasing the number of iterations.
- **Limitations:**
  - May require many iterations to provide accurate results for sparse graphs or graphs with few Hamiltonian cycles.
  - The result is an estimate, not an exact count.

5.5 Time complexity test results

first of all, let us look at analysis of time complexity of exact solutions for densly connected graphs. (density was a random number between 0.7 and 0.9)

Time complexity for exact solutions - dense graphs

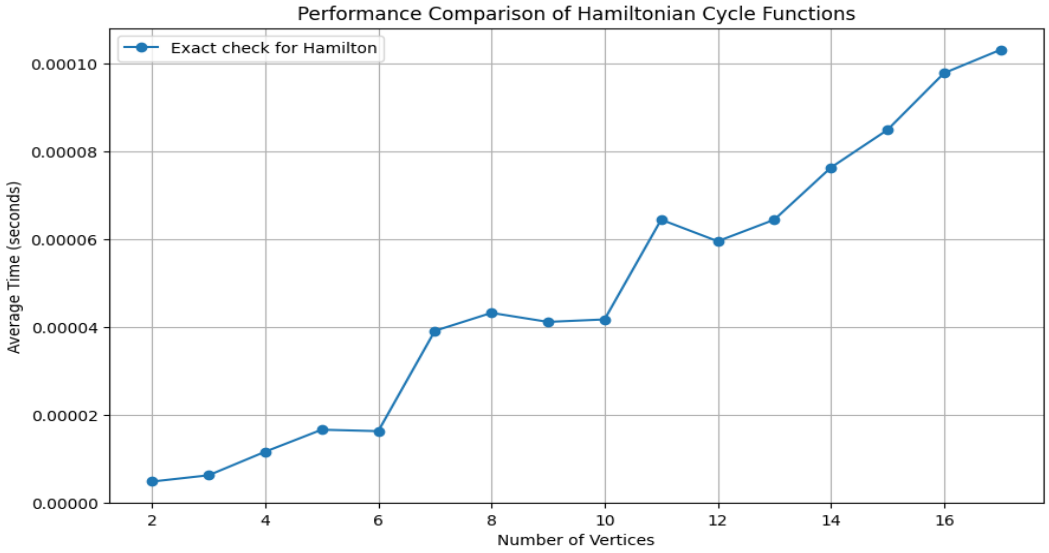

Figure 5: Time complexity for finding the Hamilton cycle on a dense graph.

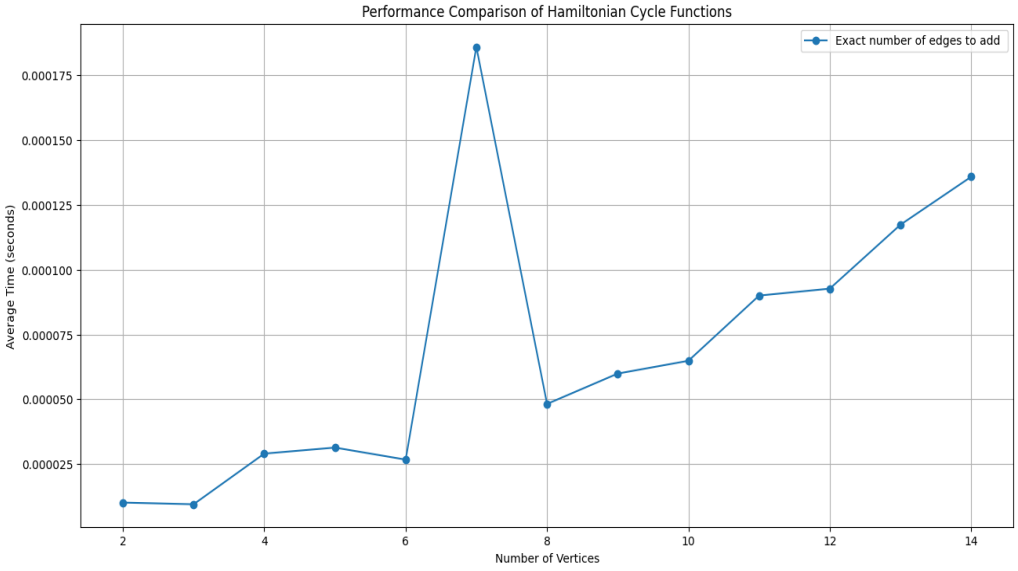

Figure 6: Time complexity for finding the minimal Hamilton extension on dense graph.

Time complexity for exact solutions - sparse graphs

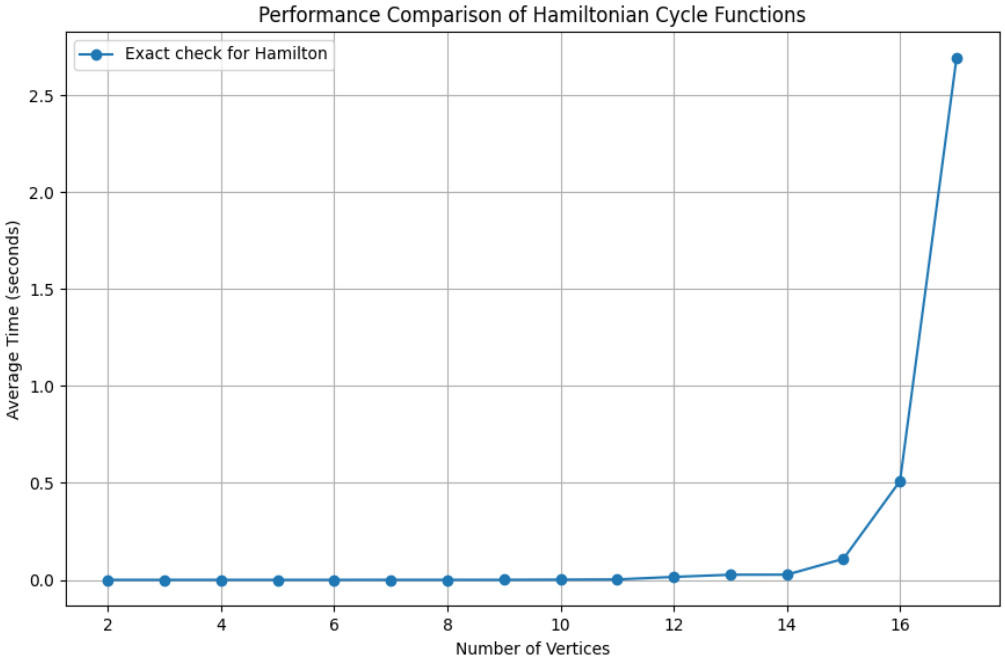

Figure 7: Time complexity for finding the Hamilton cycle on a sparse graph.

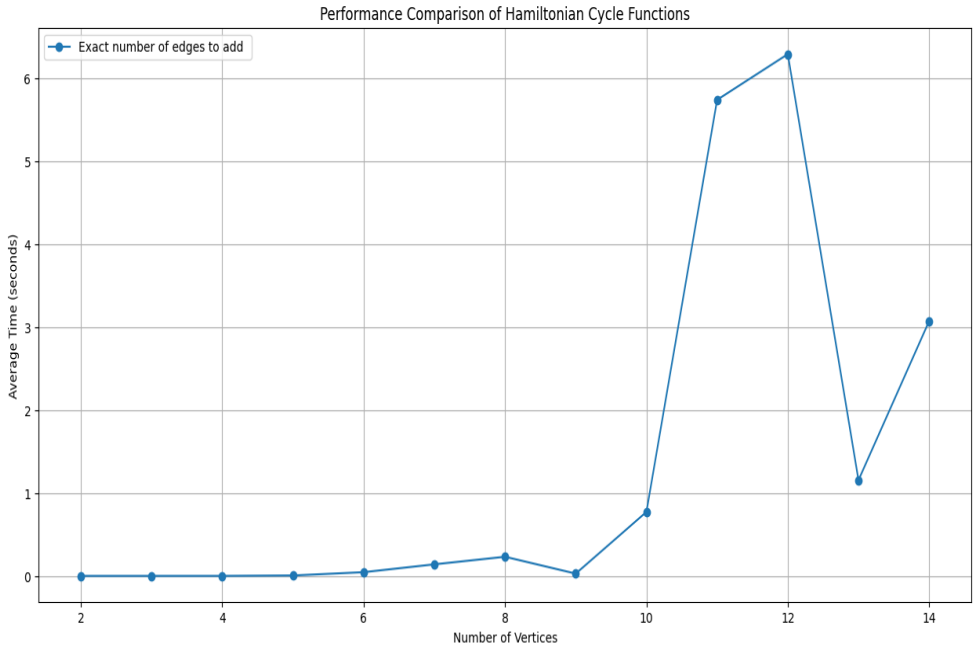

Figure 8: Time complexity for finding the minimal Hamilton extension on a sparse graph.

It is visible that since densely populated graphs have a Hamiltonian cycle the extension algorithms were quick (there were very few edges to add, thus the execution ended in reasonable time). The problems start appearing with sparsely populated graphs, since the problem is not trivial. This justifies the heuristic solutions. Lets look at their execution times. We will omit Spectral extension, since it was more of an experiment (its complexity is still rather high).

Time complexity for sparsely connected graphs - heuristic approach

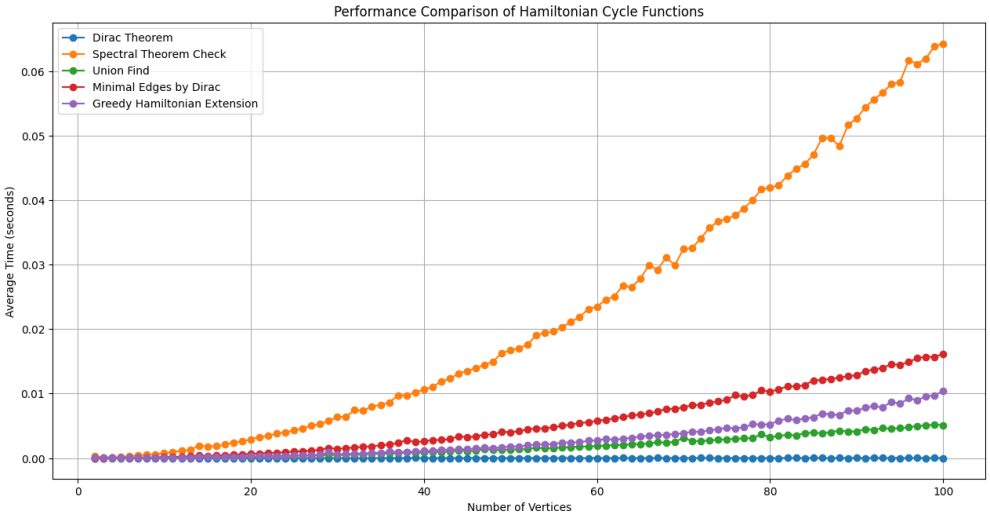

Figure 9: Time complexity for finding the minimal Hamilton extension on sparse graph.

Time complexity for densly connected graphs - heuristic approach

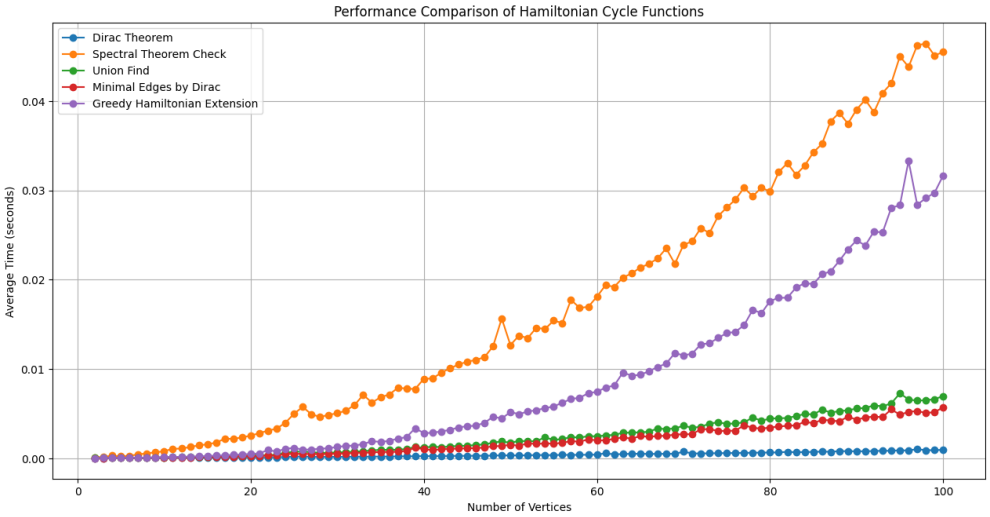

Figure 10: Time complexity for finding the minimal Hamilton extension on sparse graph.

5.6 Summary of tests

As expected, the heuristic approaches were necessary, since the complexity of exact approaches is large and grows rather quickly. The results above also show the slow growth of heuristic algorithms which was the goal of this work. It is also worth noting that since the test were done on personal computers the complexity growth might not be visible for small complexities (things such as background apps/IDE affect it).

References

[1] Bo Ning and Jun Ge. Spectral radius and hamiltonian properties of graphs. *Linear and Multilinear Algebra*, 63(8):1520–1530, August 2014.