Minimizing CCTV Units Required with Graph Coloring Using the Welsh-Powell Algorithm

Aliya Husna Fayyaza - 13523062[1,2]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]aliyahfayyaza@gmail.com, [2]13523062@std.stei.itb.ac.id

***Abstract*— This paper explores an efficient approach to minimizing the number of CCTV units required to monitor a set of rooms by applying graph coloring principles through the Welsh-Powell Algorithm. Rooms are represented as graph nodes, while connections between rooms, indicating non-visible rooms from each rooms, form the graph edges. The Welsh-Powell Algorithm is used to assign colors (representing CCTV units) to nodes, ensuring that no two connected nodes share the same color, thereby minimizing the total units needed. The algorithm prioritizes nodes with higher degrees, optimizing resource allocation. The adjacency list and matrix representations are utilized for computational efficiency and visual clarity.**

***Keywords*— adjacency list, adjacency matrix, graph coloring, optimization, Welsh-Powell algorithm.**

I. Introduction

CCTV is an essential tool for ensuring the safety and security of buildings, from private homes to high-security facilities like banks. The primary goal of placing CCTV is to eliminate blind spots, ensuring that all areas are monitored. However, achieving complete coverage can be costly due to the high price of CCTV units and their associated operational expenses. Therefore, it is important to find an efficient solution to minimize the number of CCTVs required while maintaining full surveillance coverage.

This problem can be addressed mathematically by modeling the surveillance area as a graph, where nodes represent rooms or areas and edges represent shared visibility. The minimum number of CCTVs required corresponds to the chromatic number of the graph, which can be determined using the Welsh-Powell algorithm. By prioritizing nodes with higher connections and systematically assigning colors (representing CCTVs), this algorithm ensures optimal placement and reduces costs, providing an effective and structured solution for resource allocation.

II. Basic Theory

A. Graph

A graph G is defined by G = (V,E). V is a non-empty set containing the vertices of graph G and E is a non-empty set containing the edges that connect a pair of vertices. For a G to be called graph, it should contains minimum one vertices and one edge. The number of edges that a node is connected to is called the degree. There are some kind of graph, the first one is simple graph which does not contain loops or multiple edges.

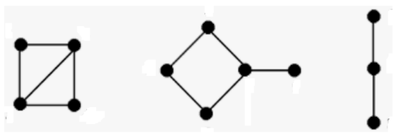


Fig 1. Simple graph examples (source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf)

The second one is un-simple graph, which contain loops.

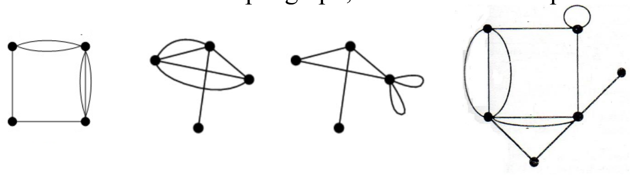


Fig 2. Unsimple graph examples (source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf)

This kind of graph is divided into two category, multi-graph which contains multi edges and pseudo-graph which does not.

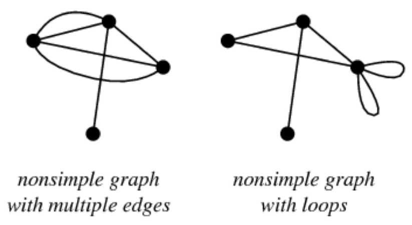


Fig 3. Unsimple graph categories (source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf)

Next, graphs are also divided based on the direction orientation. The first one is undirected graph, meaning that the edges does not have directions. The second one is directed graph or digraph, means that every edges has a direction symbolized by an arrow pointing to the desired direction.

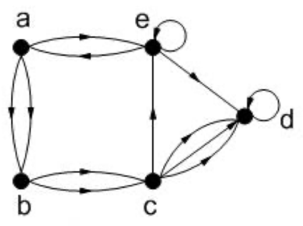


Fig 4. Directed graph example

(source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf)

Graph has a lot application in real life, such as in electrical circuits, isomers of carbon chemical compounds, food chain representation, program testing, vending machine modelling, intercity railway network, social network, computer network, and many more.

B. Graph Adjacency

Two vertices is called adjacent is when there is an edge that connect the two vertices directly connected directly. For example,

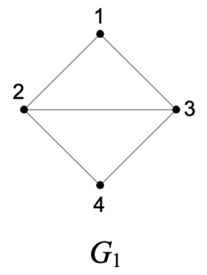


Fig 4. Graph example
(source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf)

node 1 is adjacent with node 2 and 3 and node 1 is not adjacent with node 4.

C. Graph's Adjacency Visualization

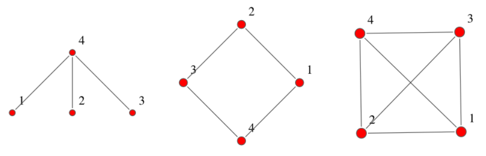
One method to represent the connections in a graph is called an adjacency matrix. The matrix is formed by the rule shown below.

$$A = [a_{ij}],$$

$$a_{ij} = \begin{cases} 1, & \text{if vertex } i \text{ and } j \text{ are adjacent} \\ 0, & \text{if vertex } i \text{ and } j \text{ are not adjacent} \end{cases}$$

Fig 4. Adjacency matrix rule
(source: author's archive)

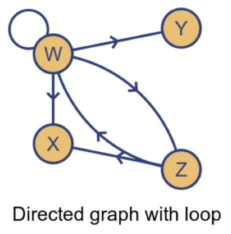
For example, the graph at the top could be represented as the matrix below it.



$$\begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix} \quad \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix} \quad \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

Fig 5. Adjacency matrix examples
(source: https://mathworld.wolfram.com/AdjacencyMatrix.html)

For undirected graphs, the matrix will be symmetrical because the edges are considered goes both way. For directed graphs, the matrix may be not symmetrical depending on the direction of the edges and for multiple edges, corresponding matrix element will be valued based on the number of multiple edges.



	W	X	Y	Z
W	1	1	1	1
X	0	0	0	0
Y	0	0	0	0
Z	1	1	0	0

Fig 6. Adjacency matrix example for directed graph
(source: https://graphicmaths.com/computer-science/graph-theory/adjacency-matrices/)

Another visualization that could be used is adjacency list. Each index of the list represents the vertices of the graph. Then, each index will be connected to another list that contains the vertices that it is adjacent with. For example, the graph at the left could be represented as the list at the right.

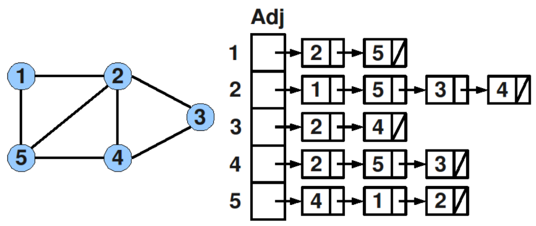


Fig 7. Adjacency list example
(source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/21-Graf-Bagian2-2024.pdf)

D. Graph Vertices Coloring

Graph Vertex Coloring is a concept in graph theory where the vertices (nodes) of a graph are assigned colors such that no two adjacent vertices (vertices connected by an edge) share the same color. The primary goal of vertex coloring is to minimize the number of colors used while satisfying this condition. This minimum number of colors is called the chromatic number.

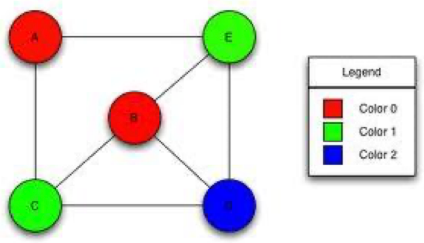


Fig 8. Colored graph example
(source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/22-Graf-Bagian3-2024.pdf)

An empty graph (a graph which no vertices is connected) has the chromatic number one, a complete graph (every vertices is connected with every other vertices) has the chromatic number is same as the number of vertices in the graph. A bipartite graph (the vertices can be divided into two disjoint sets). A circular graph with odd number of vertices will have 3 as the chromatic number and even number of vertices will have 2 as the chromatic number. For other graph, the chromatic number could not be generalized thus needed an algorithm to solve the chromatic number.

Graph coloring has numerous practical applications in various fields. In scheduling problems, it is used to allocate resources such as time slots or classrooms, ensuring no conflicts occur. In frequency assignment for telecommunications, it helps assign frequencies to cell towers while minimizing interference. Graph coloring is also applied in register allocation in compilers, where variables are assigned to registers without conflicts. In networking, it is used to optimize channel assignments and reduce data collisions.

E. Welsh-Powell Algorithm

The Welsh-Powell Algorithm is a method used for graph vertex coloring, aiming to assign colors to vertices such that no two adjacent vertices share the same color. The implement this algorithm begin by calculating the degree of each vertex in the graph. Next, arrange the vertices in descending order based on their degrees, from the highest to the lowest. Start by assigning the first color to the vertex with the highest degree in the sorted order. Then, move to the next vertex in the list and assign the same color if it is not adjacent to any previously colored vertex.

Repeat this process for all remaining vertices, introduce a new color whenever a vertex cannot be colored with an existing one. Continue following this procedure until all vertices are colored, starting with the next highest degree vertex each time.

E. Isomorphic Graphs

A graph can exist in different visual forms but still have the same number of vertices and edges, with the edges maintaining the same connectivity and the vertices maintaining the same adjacency. If there are two graphs that follow this rule, then those two graphs are said to be isomorphic. In other words, isomorphic graphs are structurally identical, even if their layouts or visual representations differ. For example, this is two graph that are isomorphic.

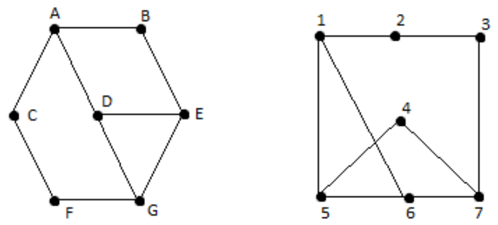


Fig 9. Isomorphic graphs example (source: https://math.stackexchange.com/questions/2041534/are-two-graph-isomorphic)

Let the graph in the left be G1 and the graph in the right be G2. All edges in G1 have their resemblance that has the same connectivity. To prove this, here is the breakdown of every edge in G1. Node A is connected to node B that has the degree of 2, to node C that has the degree of 2, and to node D that has the degree of 3. This is similar to node 7 of G2 that also is connected to 2 vertices with the degree of 2 (node 3 and 4) and 1 node with the degree of 3 (node 6). Node B in G1 is connected to node A that has the degree of 3 and to node D that has the degree of 3. This is similar to node 3 in G2 that is connected to node 7 with the degree of 3 and node 6 with the degree of 3. Node C in G1 is connected to node A with the degree of 3, to node D with the degree of 3, and to node F with the degree of 2. This is similar to node 4 in G2 that is connected to node 7 with the degree of 3, to node 6 with the degree of 3, and to node 5 with the degree of 2. Node D in G1 is connected to node A with the degree of 3, to node B with the degree of 2, to node C with the degree of 2, and to node E with the degree of 2. This is similar to node 6 in G2 that is connected to node 7 with the degree of 3, to node 3 with the degree of 2, to node 4 with the degree of 2, and to node 2 with the degree of 2.

Node E in G1 is connected to node D with the degree of 3 and to node G with the degree of 2. This is similar to node 2 in G2 that is connected to node 6 with the degree of 3 and to node 1 with the degree of 2. Node F in G1 is connected to node C with the degree of 2 and to node G with the degree of 2. This is similar to node 5 in G2 that is connected to node 4 with the degree of 2 and node 1 with the degree of 2. Finally, node G in G1 is connected to node F with the degree of 2 and to node E with the degree of 2. This is similar to node 1 in G2 that is connected to node 5 with the degree of 2 and node 2 with the degree of 2.

Thus, all vertices and edges in G1 have a corresponding match in G2 with the same degree and connectivity, proving that G1 and G2 are isomorphic.

III. IMPLEMENTATION

In this implementation, several libraries are used to facilitate the functionality of the program:

1. numpy: widely used for numerical computations. In the implementation of this paper, it is used to initiate adjacency matrices, which are crucial for representing graph relationships in a mathematical format.
2. pandas: used for data manipulation and analysis. In the implementation of this paper, it is used to handle tabular data and convert adjacency matrices into a more structured format for easier processing or visualization.
3. networkx: specializes in the creation, manipulation, and analysis of graph. It provides tools to represent graphs as nodes and edges, perform graph algorithms, and visualize networks. It is especially useful for working with graph data structures like adjacency matrices and lists. In the implementation of this paper, it is used to visualizing the colored graph.
4. matplotlib.pyplot: used for data visualization in Python. In the implementation of this paper, it is used to visually represent the graph by plotting nodes and edges, making it easier to understand the graph's structure.

These libraries work together to manage data, perform computations, and visualize the results, making the implementation more efficient and easier to understand.

A. Representing Problems As Graph

The program will take an adjacency list that consists of the positions covered by the CCTV. For example, "Room A": ["Room C"] means that Room C is the only room that could not be seen from Room A. This adjacency list will be converted into an adjacency matrix for easier visualization. This conversion is done by the function *create_adjacency_matrix_from_list*. Within this function, the nodes (representing rooms) are first extracted from the keys of the adjacency list, and the total number of nodes is determined. A square matrix of size n × n, filled with zeros using *np.zeros*. A dictionary *node_index* is also created to map each node to its corresponding index in the matrix. The function then iterates through the adjacency list, updating the matrix so that *matrix[i][j]* is set to 1 if there is a connection between node i and node j. The matrix is symmetric since the connections between nodes does not contain directions. Finally, the adjacency matrix and the list of nodes are returned.

```python
def create_adjacency_matrix_from_list(adj_list):
    nodes = list(adj_list.keys())
    n = len(nodes)
    matrix = np.zeros((n, n), dtype=int)

    node_index = {node: i for i, node in enumerate(nodes)}

    for node, neighbors in adj_list.items():
        for neighbor in neighbors:
            matrix[node_index[node]][node_index[neighbor]] = 1
            matrix[node_index[neighbor]][node_index[node]] = 1

    return matrix, nodes
```

Fig 10. create_adjecency_matrix_from_list function part 1 (source: author's source code)

Then, the adjacency matrix is converted back into a graph structure. This process involves creating a dictionary where each node is initialized with an empty list to store its neighbors. The program iterates through each pair of nodes in the adjacency matrix, and if *adj_matrix[i][j]* equals to 1, it appends the second node as a neighbor of the first node. This reconstructed graph allows further operations to be performed using the adjacency relationships, maintaining the connectivity defined in the matrix.

```python
graph = {node: [] for node in nodes}
for i, node1 in enumerate(nodes):
    for j, node2 in enumerate(nodes):
        if adj_matrix[i][j] == 1:
            graph[node1].append(node2)
```

Fig 11. create_adjecency_matrix_from_list function part 2 (source: author's source code)

B. Welsh-Powell Algorithm for Graph Coloring

The main algorithm, Welsh-Powell Algorithm is done to the graph that has already been made in the function *welsh_powell*. This algorithm starts by sorting the nodes of the graph in descending order based on their degree (the number of edges connected to a node). The sorting is performed using *sorted(graph.keys(), key=lambda x: len(graph[x]), reverse=True)*, where the nodes with the highest degrees are processed first. This method ensures that nodes with more neighbors are prioritized, which helps in minimizing the number of colors needed for the entire graph. During this step, an empty dictionary *colors* is initialized to store the assigned colors for each node, and a list *available_colors* is created to keep track of the colors already introduced during the process.

This function's output is a dictionary made of the node and the corresponding color needed. After sorting the nodes, the algorithm iterates through each node to determine the appropriate color to assign. For each node, it identifies the colors already used by its neighboring nodes using the line *used_colors = {colors[neighbor] for neighbor in graph[node] if neighbor in colors}*. Then, it checks if any of the colors in *available_colors* are not in *used_colors*. If such a color exists, it assigns that color to the current node using *colors[node] = color*. If no color is available, the algorithm introduces a new color by incrementing the count of *available_colors* and assigning it to the node, as shown in *new_color = len(available_colors) + 1* and *available_colors.append(new_color)*. This process ensures that no two adjacent nodes share the same color, resulting in a valid graph coloring. This function's output is a dictionary made of the node and the corresponding color needed.

```python
def welsh_powell(graph):
    sorted_nodes = sorted(graph.keys(), key=lambda x: len(graph[x]), reverse=True)
    colors = {}
    available_colors = []
    for node in sorted_nodes:
        used_colors = {colors[neighbor] for neighbor in graph[node] if neighbor in colors}
        for color in available_colors:
            if color not in used_colors:
                colors[node] = color
                break
        else:
            new_color = len(available_colors) + 1
            colors[node] = new_color
            available_colors.append(new_color)
    return colors
```

Fig 12. welsh_powell function source: author's source code)

C. Chromatic Number Determination

The minimum number of CCTV needed is determined by the chromatic number of the graph, which will be discovered by extracting the values in the colors dictionary produced in the *welsh_powell* function and taking the maximum color, or in this case number (to ease the process, colors are represented by integers).

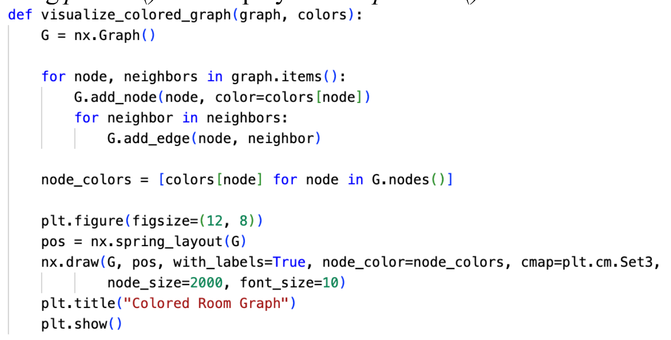
```python
min_cctv = max(colors.values())
```

Fig 13. Chromatic number determining code (source: author's source code)

D. Colored Graph Visualization

To generate a visual representation of a graph with nodes colored according to a given coloring scheme (from the Welsh-Powell method result), the *visualize_colored_graph* function is used. It starts by initializing an empty graph G using *networkx.Graph()*. The function then iterates through the graph, adding each node to G with the assigned color specified in the *colors* dictionary and creating edges between the node and its neighbors.

After building the graph structure, the function extracts the colors of all nodes into a list called *node_colors*, which will later be used for visualization. The visualization begins by defining the figure size using *plt.figure(figsize=(12, 8))* and calculating the layout of the graph using *nx.spring_layout(G)*. The graph is then drawn using *nx.draw()*, with parameters to include node labels, assign colors to nodes based on *node_colors*, A colormap *plt.cm.Set3* is applied to differentiate node colors. Finally, the graph is titled "Colored Room Graph" using *plt.title()* and displayed with *plt.show()*.

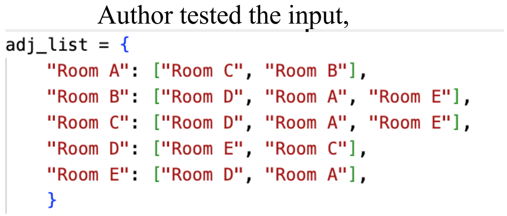


```
def visualize_colored_graph(graph, colors):
    G = nx.Graph()

    for node, neighbors in graph.items():
        G.add_node(node, color=colors[node])
        for neighbor in neighbors:
            G.add_edge(node, neighbor)

    node_colors = [colors[node] for node in G.nodes()]

    plt.figure(figsize=(12, 8))
    pos = nx.spring_layout(G)
    nx.draw(G, pos, with_labels=True, node_color=node_colors, cmap=plt.cm.Set3,
            node_size=2000, font_size=10)
    plt.title("Colored Room Graph")
    plt.show()
```

Fig 14. visualize_colored_graph (source: author's source code)

IV. TESTING

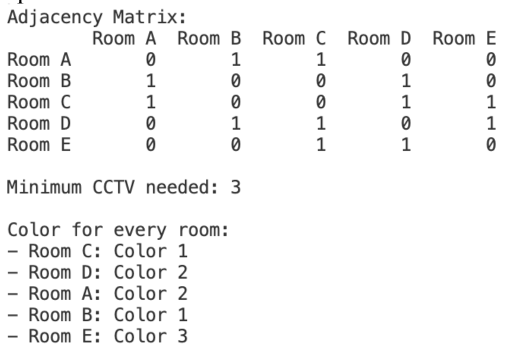
Author tested the input,



```
adj_list = {
    "Room A": ["Room C", "Room B"],
    "Room B": ["Room D", "Room A", "Room E"],
    "Room C": ["Room D", "Room A", "Room E"],
    "Room D": ["Room E", "Room C"],
    "Room E": ["Room D", "Room A"],
    }
```

Fig 15. Input testing 1 (source: author's source code)

which indicates a room as the node and the rooms that could not be seen from the node room as the adjacent nodes. The result contains the adjacency matrix, the color for every room, the minimum number of CCTV needed, and the graph representation.



```
Adjacency Matrix:
        Room A  Room B  Room C  Room D  Room E
Room A       0       1       1       0       0
Room B       1       0       0       1       0
Room C       1       0       0       1       1
Room D       0       1       1       0       1
Room E       0       0       1       1       0

Minimum CCTV needed: 3

Color for every room:
- Room C: Color 1
- Room D: Color 2
- Room A: Color 2
- Room B: Color 1
- Room E: Color 3
```

Fig 16. Result for testing 1 (source: author's source code)

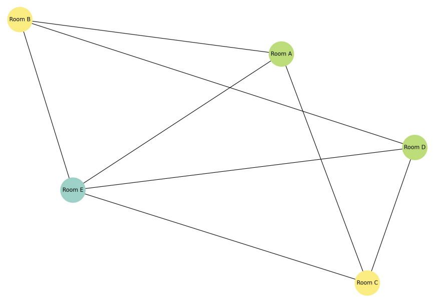


Fig 17. Graph visualization for testing 1 (source: author's source code)

Needed to be highlighted that the color assigned to every rooms and the shape of the graph could be different for every runtime but the chromatic number will always be the same. The graphs got for every time the program run will be isomorphic to each other. The node for the same room will be the resemblance. For example, here is another graph that is visualized in the next runtime,

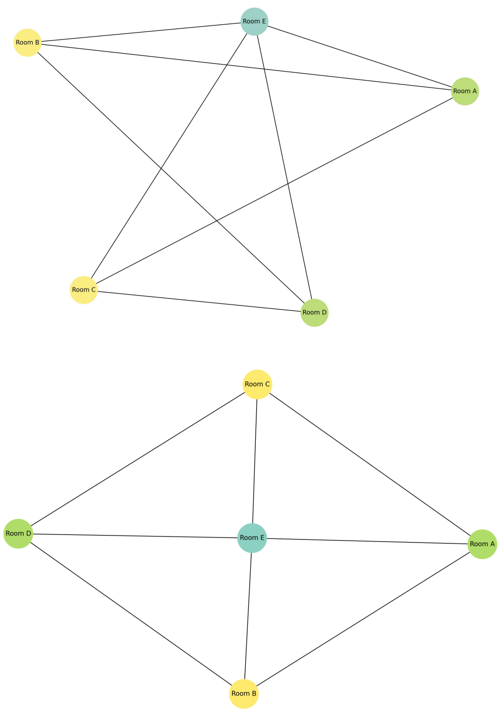
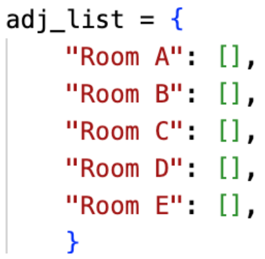


Fig 18. Isomorphic graphs for testing I result (source: author's source code)

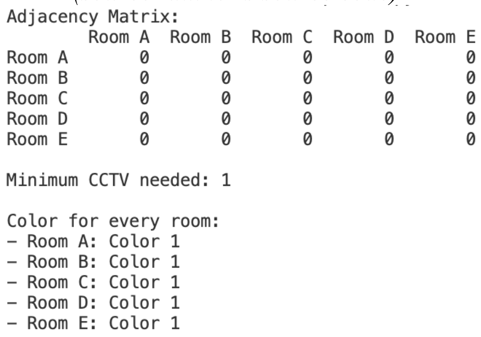
These isomorphic graphs displayed the possibility of the layout of the rooms and the edges shows the rooms that could not be viewed from the room.

If given that we can see every room from any room, the number of CCTV will only be one and could be placed in any room and will still cover all of the other room. Author tested this scenario by using the test case,



```
adj_list = {
    "Room A": [],
    "Room B": [],
    "Room C": [],
    "Room D": [],
    "Room E": [],
    }
```

Fig 19. Input testing 2 (source: author's source code)



```
Adjacency Matrix:
        Room A  Room B  Room C  Room D  Room E
Room A       0       0       0       0       0
Room B       0       0       0       0       0
Room C       0       0       0       0       0
Room D       0       0       0       0       0
Room E       0       0       0       0       0

Minimum CCTV needed: 1

Color for every room:
- Room A: Color 1
- Room B: Color 1
- Room C: Color 1
- Room D: Color 1
- Room E: Color 1
```

Fig 20. Result for testing 2 (source: author's source code)

meaning that there is no room that could not be seen from any available room.

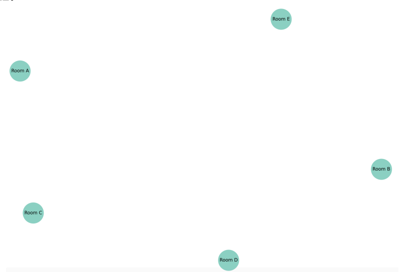


Fig 21. Graph visualization for testing 2
(source: author's source code)

This is reciprocal with the theory stated before, an empty graph (no vertices is connected) has the chromatic number one.

If given that we can not see any other room from any room, or in another word, all rooms are completely separated from each other, the number of CCTV will be same as the number of room available because one CCTV will only be covering one room. Author tested this scenario by using the test case,

```
adj_list = {
    "Room A": ["Room B", "Room C", "Room D", "Room E"],
    "Room B": ["Room A", "Room C", "Room D", "Room E"],
    "Room C": ["Room A", "Room B", "Room D", "Room E"],
    "Room D": ["Room A", "Room B", "Room C", "Room E"],
    "Room E": ["Room A", "Room B", "Room C", "Room D"],
    }
```

Fig 22. Input testing 3
(source: author's source code)

```
Adjacency Matrix:
        Room A  Room B  Room C  Room D  Room E
Room A      0       1       1       1       1
Room B      1       0       1       1       1
Room C      1       1       0       1       1
Room D      1       1       1       0       1
Room E      1       1       1       1       0

Minimum CCTV needed: 5

Color for every room:
- Room A: Color 1
- Room B: Color 2
- Room C: Color 3
- Room D: Color 4
- Room E: Color 5
```

Fig 23. Result for testing 3
(source: author's source code)

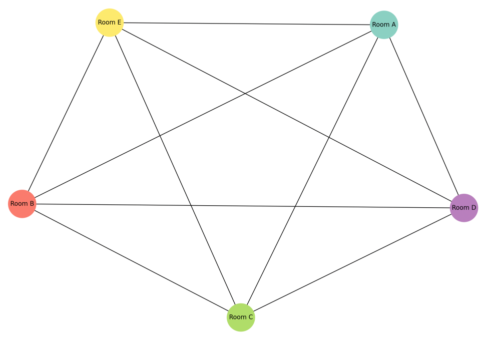


Fig 24. Graph visualization for testing 3
(source: author's source code)

This is reciprocal with the theory stated before, a complete graph (every node is connected to other vertices) has the chromatic number is same as the number of vertices in the graph

By using the Welsh-Powell algorithm, the program calculates the minimum number of CCTV units required, effectively reducing unnecessary expenses on additional equipment. This is particularly beneficial in large-scale applications, such as office buildings, shopping malls, or industrial complexes, where maximizing cost efficiency without compromising security is critical. The program's ability to model surveillance areas as graphs and optimize CCTV placement makes it a valuable tool for planners and security professionals seeking to balance coverage and budget constraints.

V. Conclusion

The application of the Welsh-Powell algorithm for optimizing CCTV placement demonstrates an efficient approach to minimizing surveillance costs while ensuring complete area coverage. By modeling the surveillance area as a graph, with rooms represented as nodes and visibility overlaps as edges, the program effectively calculates the minimum number of CCTVs required. The algorithm guarantees that no two adjacent nodes share the same color, aligning with graph coloring principles to determine the chromatic number.

Through various test cases, the program validated theoretical scenarios, such as an empty graph requiring only one CCTV and a complete graph requiring the same number of CCTVs as rooms. This prove the accuracy of the algorithm in handling diverse scenarios. The system's scalability and ability to provide cost-effective solutions make it particularly valuable for large-scale applications, such as office buildings and industrial facilities. The approach not only optimizes resource allocation but also highlights the practical utility of graph theory in solving real-world problems.

VI. Appendix

Github: https://github.com/aliyahusnaf/MatdisPaper2024.git
Bonus video: https://youtu.be/0PxppeJFyKE

VII. Acknowledgment

Author expresses gratitude to God Almighty for Hi blessings and grace, which have provided the strength to write this paper titled "Minimizing CCTV Units Required with Graph Coloring Using the Welsh-Powell Algorithm" successfully. The author would also like to extend heartfelt thanks to Ir. Rila Mandala, M.Sc., Ph.D, the lecturer of the Discrete Mathematics course for the Odd Semester of 2023/2024, Class 02, for imparting valuable knowledge that served as a foundation for this paper. The author also wishes to thank all the reference sources utilized in the preparation of this paper. Lastly, the author sincerely apologizes for any errors that may be present in this paper.

REFERENCES

[1] Munir, Rinaldi. 2023. "Graf (Bag. 1)". https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf. Accessed 21 December 2024, 8:30 AM.

[2] Munir, Rinaldi. 2023. "Graf (Bag. 2)". https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf. Accessed 21 December 2024, 8:30 AM. Accessed 21 December 2024, 9:30 AM.

[3] Munir, Rinaldi. 2023. "Graf (Bag. 3)". https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/22-Graf-Bagian3-2024.pdf. Accessed 21 December 2024, 8:30 AM. Accessed 21 December 2024, 11:30 AM.

[4] Mayank, Mohit. 2021. "Visualisizing Networks in Python). https://towardsdatascience.com/visualizing-networks-in-python-d70f4cbeb259. Accessed 21 December 2024, 4:30 PM.

[5] McBride, Martin. 2023. "Adjacency matrices". source: https://graphicmaths.com/computer-science/graph-theory/adjacency-matrices/. Accessed 21 December 2024, 7:30 PM.

[6] Grohe, Martin. 2020. "The Graph Isomorphism Problem". source: https://cacm.acm.org/research/the-graph-isomorphism-problem/. Accessed 21 December 2024, 10:30 PM.

[7] Sauras, Altuzarra, Lorenzo and Weisstein, W., Eric. 2025. "Adjacency Matrix". source: https://mathworld.wolfram.com/AdjacencyMatrix.html. Accessed 4 January 2025, 09:30 AM.

PERSONAL STATMENT

I hereby declare that the paper I have written is my own work, not an adaptation or translation of someone else's paper, and not plagiarism.

Bandung, December 27 2024

[illegible]

Aliya Husna Fayyaza/13523062