CS301 Project

Minimum Clique Partition

Project Report

Group 30:

- Enver Atahan Çelik, 28494
- Taylan Karadeniz, 22336
- Mehmet Enes Battal, 26354
- Muhammed Batuhan Odabaşı, 23636

FALL 2020-2021
Instructor: Hüsnü Yenigün

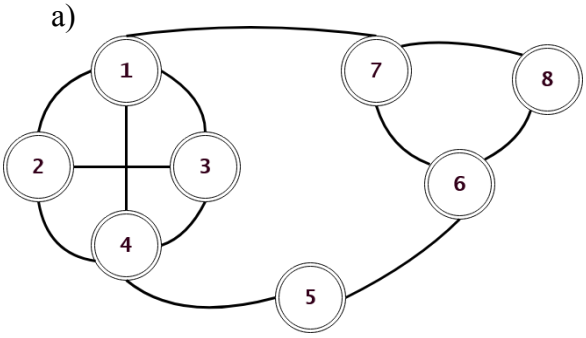
1) Problem Description

A clique is a special subset of vertices in an undirected graph such that any two vertices in the clique are adjacent. Minimum Clique Partition is to find the minimum number for an undirected graph G = (V, E) such that V can be partitioned into a minimum number of cliques. Such minimum number for an undirected graph G is also called the clique cover number of G.

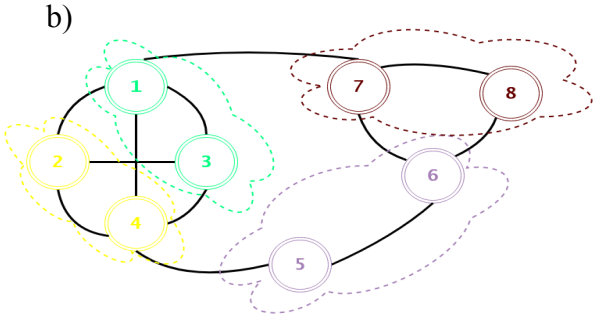
Examples for Minimum Clique Cover:

Consider the graph G = (V,E) in a) ,

a)

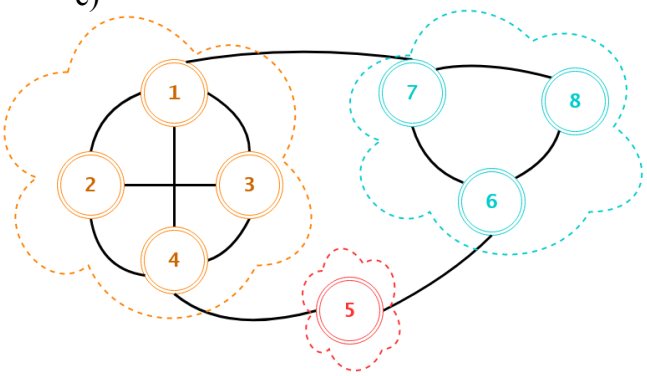


b)

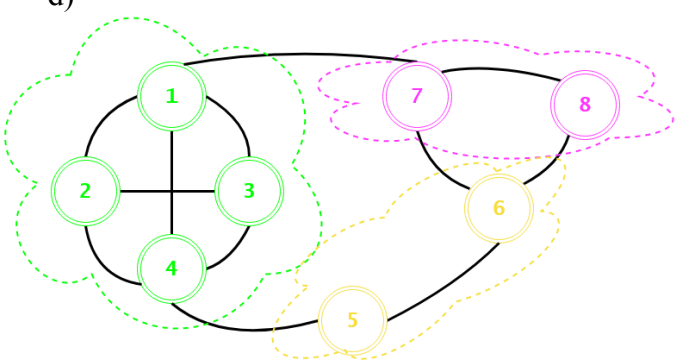


Vertices of G partition into 4 subsets in G. For every subset, any vertex in the set has edges to any other vertex in the set, hence b) is a clique cover of G.

c)



d)



The partitions in c) and d) are clique covers that both partitions V into 3 cliques. Since the graph G can be partitioned into 3 cliques at minimum, both of these partitions are minimum clique partitions with a clique cover number of 3.

Another way to solve the problems is considering the vertex coloring in the complement graph G' by using a minimum number of colors.

A possible real world application is used to cover some desired area with a unit by using minimum resources. This is a general problem for any field such as fast food restaurant chains since it enables the chain to grow larger with the same cost and cancer treatment tests that screens the desired area since it reduces the expenditure of the treatment.

For a party/organization/player, a unit is any extension of the party that is established by consuming the limited resource of the party. Potential sites for locating units are called demand points. A facility covers a demant point if it is positioned in a neighbourhood of the demant point. A Mandatory coverage problem is a problem to find the minimum number of units that cover all demand points. Such a problem instance can be translated into a graph. Hence it is sufficient to solve the minimum clique partition in order to cover demand point with minimum number of units.[1, 2]

Minimum Clique Partition is NP - Complete

Theorem: Minimum Clique Partition is NP Complete.

This can be proved by showing that the following are true:

- Minimum Clique Partition is in NP.
- Show that an NP Complete problem can be reduced to Minimum Clique Partition in polynomial time.

A. Minimum clique partition is in NP

A problem should have polynomial-time verifiability if it belongs to the NP class. According to a given certificate, we should be able to verify in polynomial time if it is a solution to the problem or not.

Proof:

1- Certificate: Suppose certificate is a set S consisting of nodes in the clique and S is a subgraph of G.

2- Verification: It should be checked that if there is a clique of size k in the graph. Therefore, It takes O(1) time verifying if the number of nodes in S equals k. It takes O(k^2) time to verify if each vertex has an out-degree of (k-1)(Since in a complete graph, each vertex is connected to every other vertex through an edge. Therefore the total number of edges in a complete graph = k*(k-1)/2). Hence, it takes O(k^2) = O(n^2) time to check if the graph formed by the k nodes in S is complete or not, (because of k <= n, where n is number of vertices in G).

Because this Clique Decision Problem has polynomial time verifiability it belongs to the NP Class.

B. Minimum clique partition can be reduced to an NP - Complete

If every NP problem is reducible to L in polynomial time then problem L belongs to NP-Hard. Now, let the Clique Decision Problem be referred to as C. For proving C is NP-Hard, we take an already known NP-Hard problem, say that is S, then reduce it to a specific instance. C is also an NP-Hard problem if this reduction can be done in polynomial time. The Boolean Satisfiability Problem (S) is an NP-Complete problem as proved by Cook's theorem. Hence, all problems in NP can be reduced to S in polynomial time. Therefore, if S is reducible to C in polynomial time, then every NP problem can be reduced to C in polynomial time, so proving C to be NP-Hard. [5]

In short, the Satisfiability problem has a reduction to 3-SAT problem. Similarly, 3-SAT can be reduced to Chromatic Number problem, i.e. the famous coloring problem for $k \geq 3$ colors.

Finally the Chromatic Number problem has a reduction to Clique Cover, which one can find Minimum Clique Partition by exhaustively checking the clique covers.

For details, see the following paper:

Karp, Richard. (1972). Reducibility Among Combinatorial Problems.
Complexity of Computer Computations.

2) Algorithm Description

Minimum clique cover is NP-complete and because of it no algorithm has been found that can solve this problem in polynomial time. However other approaches such as heuristic algorithms can be used to approximate an answer in polynomial time. The guarantee of the heuristic algorithm's solution being optimal is not given. Keeping this in mind, A heuristic algorithm for minimum clique cover will be analyzed in this project.[3]

Pseudo code for this algorithm is as follows:

- Create an empty list called clique_list which will hold the cliques that have been found.
- Create a copy of the input called graph.
- While there are nodes in the graph:
    - Create an empty list called clique for a single clique.
    - Create a list consisting of remaining nodes in the graph.
    - Randomly shuffle this list.
    - For every node in this list:
        - Check if node is adjacent to every node in the clique list.
        - If it is, add to the clique list.
    - Remove the nodes that have been added into the clique list from the graph.
    - Append the found clique to clique_list.
- Return the clique_list.

The minimum number of cliques can be found by getting the length of the returned clique_list. Since it actually returns the list of the cliques, the cliques that have been found by the algorithm can be seen as well.

The following is the implementation of the mentioned algorithm using python.

```python
def clique_random_sequential(graph : nx.Graph) -> list:
    graph = graph.copy()
    clique_list = []
    while len(graph.nodes())>0:
        clique = []
        node_list = list(graph.nodes())
        np.random.permutation(node_list)
        for node in node_list:
            flag = True
            for exist_node in clique:
                if node not in graph[exist_node]:
                    flag =False
                    break
            if flag:
                clique.append(node)
        graph.remove_nodes_from(clique)
        clique_list.append(clique)
    return clique_list
```

3) Algorithm Analysis

This heuristic algorithm finds the minimum vertex cover of a given graph. This algorithm runs in $O(V^2)$ and in this section, finding the running time will be explained.

A graph has V vertices and E edges. Algorithm uses the vertices of a given graph to find the minimum clique cover.

The while loop can iterate at most V times because for every iteration at least 1 vertex is removed from the graph decreasing the count of vertices. Similarly the outer for loop can iterate at most V times since a graph has V vertices. The inner for loop also can iterate at most V times since there are V vertices and a clique can consist of all of these vertices. Note that in this case there is only one clique which is all of the graph.

Suppose the following iteration times are valid:

```
def clique_random_sequential(graph : nx.Graph) -> list:
    graph = graph.copy()
    clique_list = []
    while len(graph.nodes())>0:  → iterates V times
        clique = []
        node_list = list(graph.nodes())
        np.random.permutation(node_list)
        for node in node_list:  → iterates V times
            flag = True
            for exist_node in clique:  → iterates V times
                if node not in graph[exist_node]:
                    flag =False
                    break
            if flag:
                clique.append(node)
        graph.remove_nodes_from(clique)
        clique_list.append(clique)
    return clique_list
```

So the running time is O(V^3) by multiplying these. But these iterations are not possible in reality. By assuming that these are possible, an upper bound for this algorithm is found but what if finding a tighter bound is possible?

Let's consider two extreme cases regardless of their possibility:

1. All of the vertices are in the same clique:
   Let's start from the inside for loop. This will iterate V times at maximum which is O(V) because it is assumed that all of the vertices are in the same clique. The outer for loop will also iterate V times at maximum which is O(V) since it looks for all of the nodes in the graph. Since the vertices that are included in a clique are removed from the input graph, in this case all of them, while loop will only iterate once which is O(1). So the overall algorithm will have O(V * V * 1) = O(V^2).

2. None of the vertices are in the same clique:
   Again start from the inside for loop. This will iterate 1 time since it is assumed that none of the vertices are in the same clique. So O(1) time. The outer for loop will iterate maximum V times since it will look for all of the vertices that are not part of a clique.

Hence it is O(V). This time note that the while loop will iterate V times since it starts with V vertices and in every iteration only 1 vertex is removed from the graph. The overall complexity of this case is also O(1 * V * V) = O(V^2).

Note that the difference between the number of elements in a clique is handled by the while loop and the inner for loop. They will balance themselves and because of this, in every case time complexity will be equal to O(V^2).

```
def clique_random_sequential(graph : nx.Graph) -> list:
    graph = graph.copy()
    clique_list = []
    while len(graph.nodes())>0:
        clique = []
        node_list = list(graph.nodes())
        np.random.permutation(node_list)
        for node in node_list:
            flag = True
            for exist_node in clique:
                if node not in graph[exist_node]:
                    flag =False
                    break
            if flag:
                clique.append(node)
        graph.remove_nodes_from(clique)
        clique_list.append(clique)
    return clique_list
```

O(V^2)

4) Experimental Analysis

A. Running time experimental analysis

The algorithm performance analysis is done by comparing the running times for different input sizes (V, E). Below code used in the analysis in order to compute respective running time, mean, standard deviation, standard error and interval for confidence levels.

```
iteration = 100
vertex_sizes = [100, 300, 500, 1000]
densities = [0.1, 0.25, 0.5, 0.75, 1]

i = 1
for vertex_size in vertex_sizes:
  for density in densities:
    time_list = []
    for num in range(iteration):
      G = randomG.fast_gnp_random_graph(vertex_size,density, seed = None, directed = False)
      start = time.perf_counter() #Starts the timer for the run
      clique_random_sequential(G) #The algorithm
      end = time.perf_counter()   #Ends the timer for the run
      time_list.append(end-start)

    std = np.std(time_list)       #Standard Deviation
    mean = np.mean(time_list)     #Mean
    stderr = std / np.sqrt(iteration)#Standard Error

    uppermean90 = mean + (stderr * tval90)
    lowermean90 = mean - (stderr * tval90)
    CL90 = str(lowermean90)[0:8] + "-" + str(uppermean90)[0:8] #%90 Confidence Interval

    uppermean95 = mean + (stderr * tval95)
    lowermean95 = mean - (stderr * tval95)
    CL95 = str(lowermean95)[0:8] + "-" + str(uppermean95)[0:8] #%95 Confidence Interval

    df_v_p.loc[i] = pd.Series(data = (vertex_size, density, mean, std, stderr, #Add the run to the table
                                      CL90, CL95), index = table_v_p, name = i)
    i += 1  #Next configuration
```

The approximation values, 1.645 for %90 confidence level t-value, and 1.96 for %95 confidence level t-value are used in the analysis.

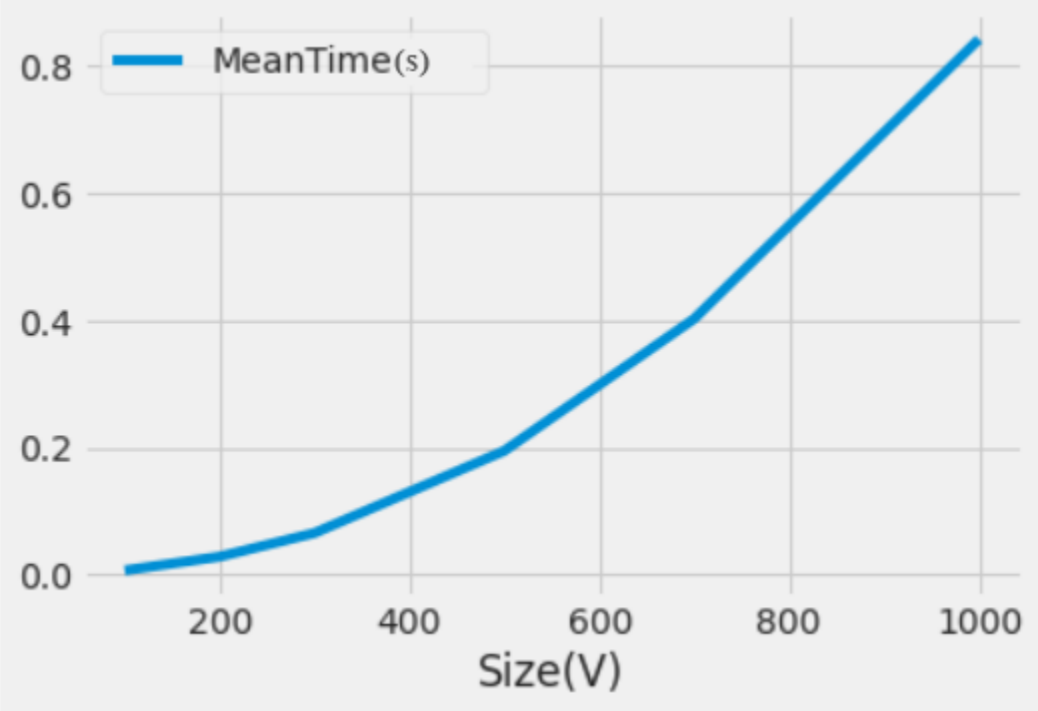
Python networkX library is used for generating random graphs by;

1) Fixed edge size, increasing the vertex size
2) Fixed vertex size, increasing the edge size
3) Increasing the density with the vertex size

On each category and on each configuration of the input size, the algorithm runs for 100 random graphs. Since the algorithm running time is mainly dependent on the vertex size, (1) and (3) are more meaningful than (2). The time values are in seconds.

1. Fixed edge size, increasing the vertex size

	Size(V)	MeanTime(s)	Standard Deviation	Standard Error	%90-CL	%95-CL
1	100.0	0.010571	0.001223	0.000122	0.010370-0.010772	0.010331-0.010811
2	200.0	0.034778	0.002474	0.000247	0.034370-0.035184	0.034292-0.035262
3	300.0	0.066433	0.004803	0.000480	0.065643-0.067223	0.065491-0.067374
4	500.0	0.190096	0.006086	0.000609	0.189094-0.191096	0.188902-0.191288
5	700.0	0.381310	0.007300	0.000730	0.380109-0.382511	0.379879-0.382741
6	1000.0	0.816085	0.014578	0.001458	0.813686-0.818483	0.813227-0.818942



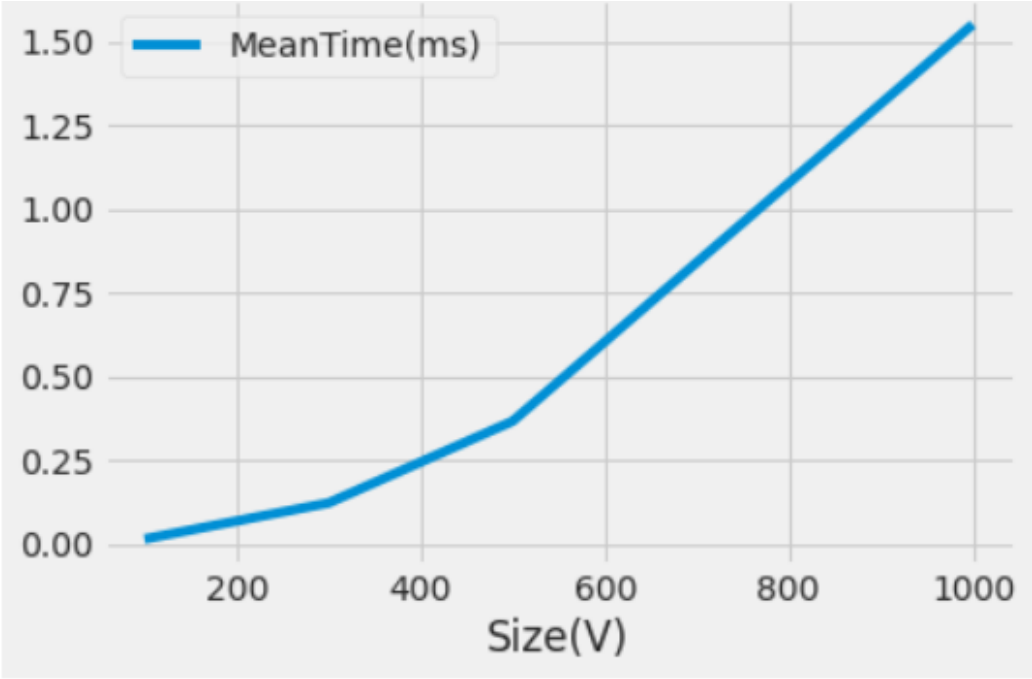
2. Fixed vertex size, increasing the edge number

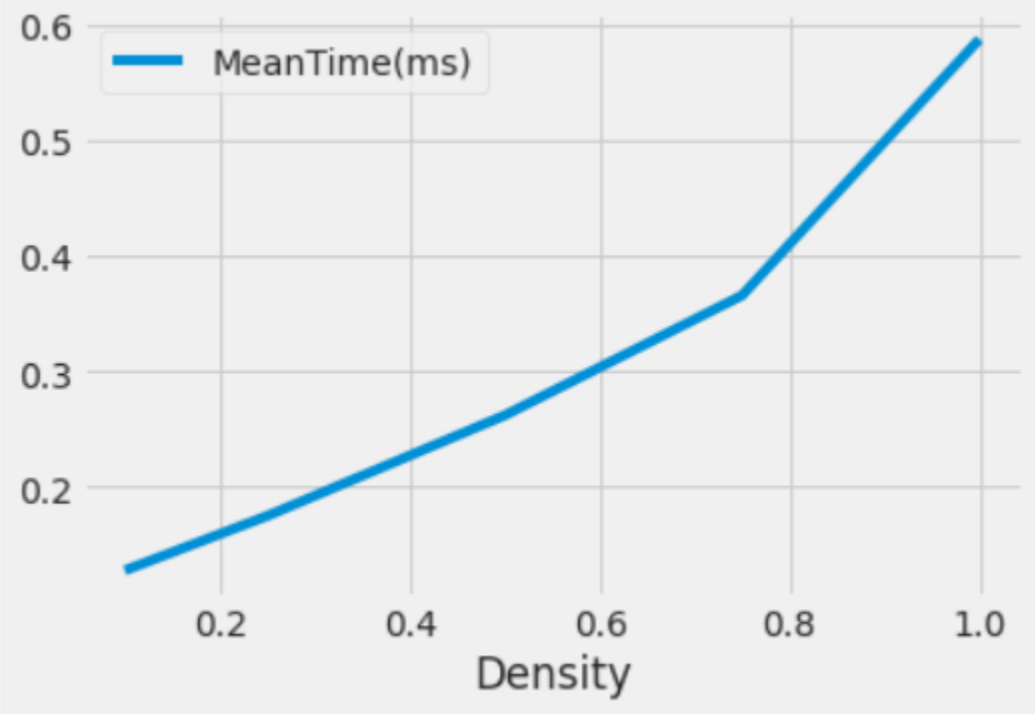
	Size(E)	MeanTime(s)	Standard Deviation	Standard Error	%90-CL	%95-CL
1	100.0	0.009352	0.000646	0.000065	0.009245-0.009458	0.009225-0.009478
2	500.0	0.007972	0.000530	0.000053	0.007885-0.008059	0.007868-0.008076
3	1000.0	0.007461	0.001255	0.000125	0.007254-0.007667	0.007214-0.007706
4	2000.0	0.008903	0.000959	0.000096	0.008744-0.009060	0.008714-0.009090
5	3000.0	0.011357	0.001144	0.000114	0.011168-0.011545	0.011132-0.011581
6	4000.0	0.014115	0.001374	0.000137	0.013888-0.014340	0.013845-0.014384

3. Increasing the density with vertex size

	Size(V)	Density	MeanTime(s)	Standard Deviation	Standard Error	%90-CL	%95-CL
1	100.0	0.10	0.007680	0.000957	0.000096	0.007522-0.007837	0.007492-0.007867
2	100.0	0.25	0.007891	0.001110	0.000111	0.007708-0.008073	0.007673-0.008108
3	100.0	0.50	0.012865	0.026167	0.002617	0.008560-0.017169	0.007736-0.017993
4	100.0	0.75	0.013578	0.001129	0.000113	0.013392-0.013763	0.013356-0.013799
5	100.0	1.00	0.020857	0.002099	0.000210	0.020511-0.021202	0.020445-0.021268
6	300.0	0.10	0.050860	0.026051	0.002605	0.046574-0.055145	0.045754-0.055966
7	300.0	0.25	0.075518	0.043077	0.004308	0.068431-0.082604	0.067074-0.083960
8	300.0	0.50	0.096702	0.029007	0.002901	0.091930-0.101474	0.091016-0.102387
9	300.0	0.75	0.121385	0.002960	0.000296	0.120898-0.121872	0.120804-0.121965
10	300.0	1.00	0.194599	0.005395	0.000540	0.193711-0.195486	0.193541-0.195656
11	500.0	0.10	0.128013	0.005117	0.000512	0.127171-0.128854	0.127010-0.129015
12	500.0	0.25	0.175153	0.038505	0.003851	0.168818-0.181486	0.167605-0.182699
13	500.0	0.50	0.262112	0.030235	0.003024	0.257138-0.267085	0.256185-0.268037
14	500.0	0.75	0.365883	0.040088	0.004009	0.359288-0.372477	0.358025-0.373740
15	500.0	1.00	0.587561	0.031552	0.003155	0.582371-0.592751	0.581377-0.593745
16	1000.0	0.10	0.521458	0.038373	0.003837	0.515145-0.527770	0.513936-0.528979
17	1000.0	0.25	0.712875	0.049819	0.004982	0.704679-0.721070	0.703110-0.722639
18	1000.0	0.50	1.157307	0.112860	0.011286	1.138741-1.175872	1.135186-1.179427
19	1000.0	0.75	1.551674	0.079771	0.007977	1.538552-1.564796	1.536039-1.567309
20	1000.0	1.00	2.513662	0.117575	0.011757	2.494320-2.533003	2.490617-2.536706

In (3), running time increases polynomially as the size of V and the density of the graph increases. As examples, the following first graph shows the running time with respect to the size of V while density is 0.75. The latter graph shows the running time with respect to the density of the graph while the size of V is 500.





B. Correctness

Analized heuristic algorithm for solving Minimum Clique Partition problem doesn't guarantee that the answer is minimum, since it is a greedy alternative to the brute force approach that checks all subsets to find minimum clique partition. On the other hand, the algorithm always returns a valid partition of a given graph. Following code checks if a given partition is a clique in 100 random graphs, for each vertex size and density configuration. NetworkX library is again utilized to calculate the accuracy of the algorithm for finding successful clique partitions.

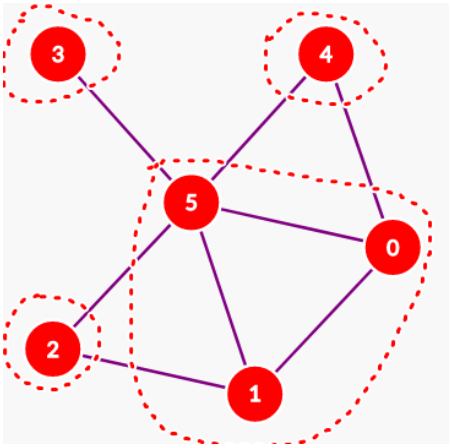
```
import networkx.classes.function as fun
import networkx.generators.random_graphs as randomG

def isCliquePartition(graph: nx.graph, partition) -> bool:
  isClique = True
  for node_list in partition:
    n = len(node_list)
    G = fun.subgraph(graph, node_list)
    isClique = (fun.number_of_edges(G) * 2 == n * (n-1))
    if (isClique ==  False):
      break

    return isClique

iteration = 100
node_sizes = [20, 50, 100, 200, 500]
densities = [0.1, 0.25, 0.5, 0.75, 1]

correct_answers = 0
for node_size in node_sizes:
  for density in densities:
    for i in range(iteration):
      GT = randomG.fast_gnp_random_graph(node_size, density, seed = None, directed = False)
      partition = clique_random_sequential(GT)
      if (isCliquePartition(GT, partition)):
        correct_answers += 1

result = correct_answers / (iteration * len(node_sizes) * len(densities))
print(result)
print("Clique partition accuracy: ", result == 1)
```

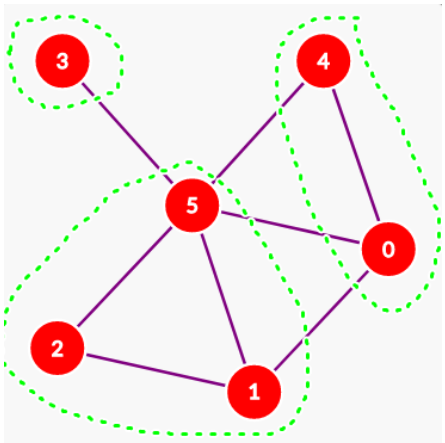
```
1.0
Clique partition accuracy:  True
```

5) Testing

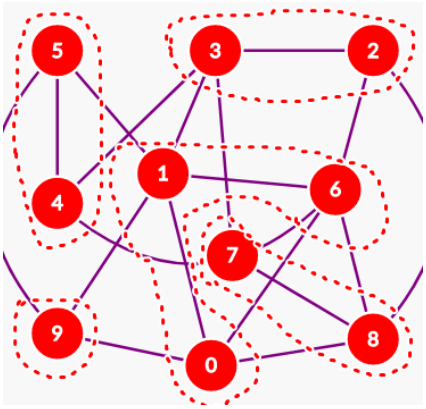
To test this heuristic algorithm, blackbox testing is used to analyze correctness of the algorithm. In order to test the algorithm, random graphs of various sizes are generated and solved with the heuristic algorithm.

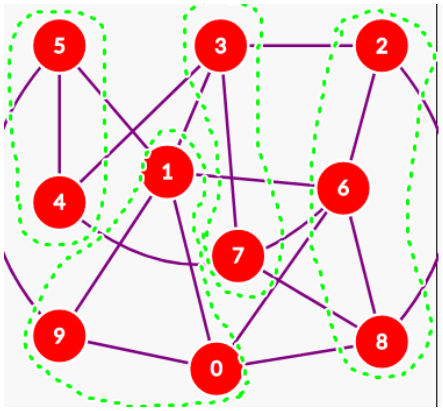
To show that it doesn't always find the correct solution, we decided to include more of the failed cases than the successful ones.



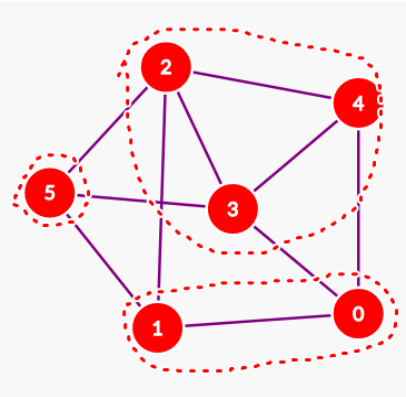


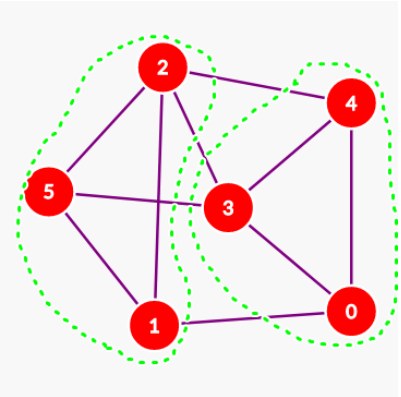
FAILED: First graph is our algorithm's minimum clique partition and it uses 4 cliques, while the second graph is the correct minimum clique partition which has 3 cliques.



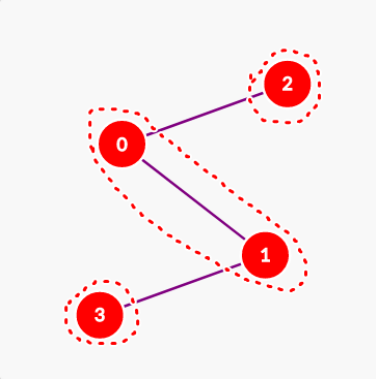


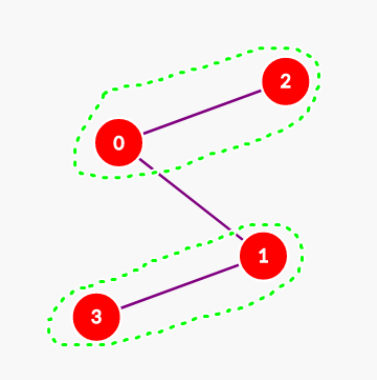
FAILED: First graph is our algorithm's minimum clique partition, that uses 5 cliques, while the second graph is the correct minimum clique partition, that uses 4 cliques.



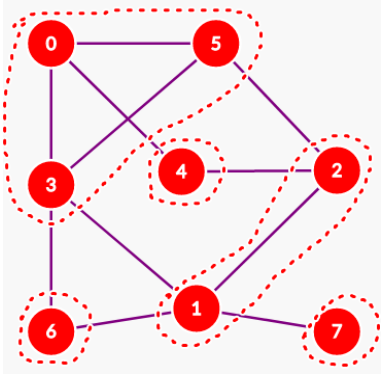


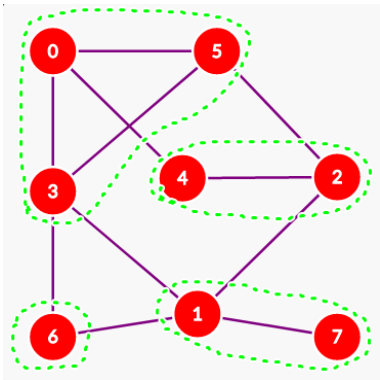
FAILED: First graph is our algorithm's minimum clique partition, that uses 3 cliques, while the second graph is the correct minimum clique partition, that uses 2 cliques.



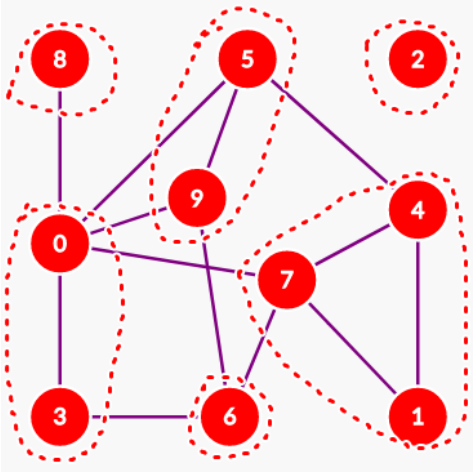


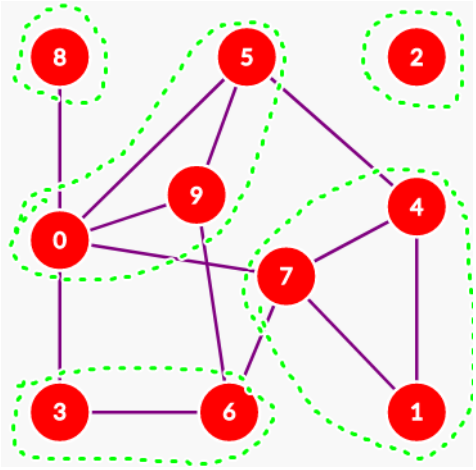
FAILED: First graph is our algorithm's minimum clique partition, that uses 3 cliques, while the second graph is the correct minimum clique partition, that uses 2 cliques.



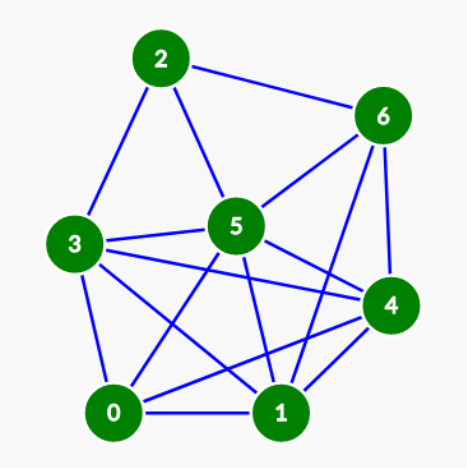


FAILED: First graph is our algorithm's minimum clique partition, that uses 5 cliques, while the second graph is the correct minimum clique partition, that uses 4 cliques.

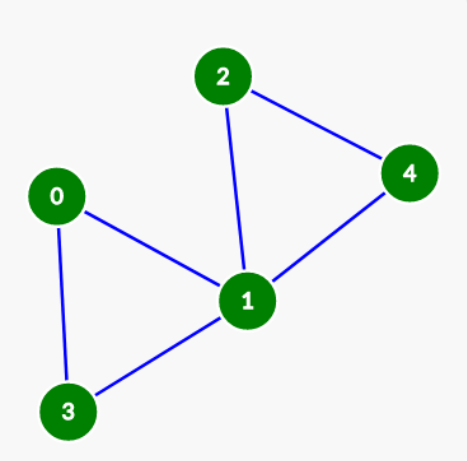




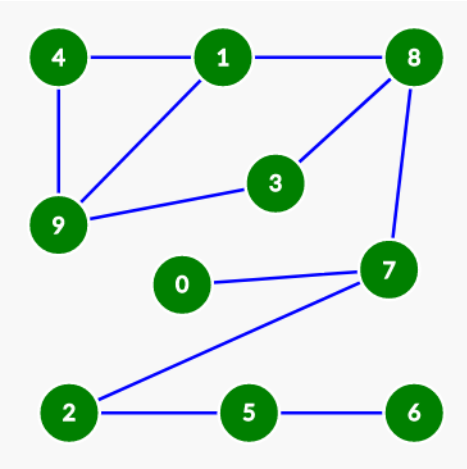
FAILED: First graph is our algorithm's minimum clique partition, that uses 6 cliques, while the second graph is the correct minimum clique partition, that uses 5 cliques.



SUCCEEDED: Our algorithm's minimum clique partition, uses 2 cliques, which is the correct minimum clique partition.



SUCCEEDED: Our algorithm's minimum clique partition, uses 2 cliques, which is the correct minimum clique partition.



SUCCEEDED: Our algorithm's minimum clique partition, uses 5 cliques, which is the correct minimum clique partition.

6) Discussion

To conclude, the Minimum Clique Partition is one of the NP-Complete problems. There is no algorithm found that solves the problem in polynomial time optimally for all instances of the problem. Nevertheless there are some heuristic algorithms that can solve the problem in polynomial time but they are not guaranteeing the 100 percent accuracy with the answers, such as our algorithm.

The algorithm we used is an heuristic algorithm that calculates the minimum number of cliques in polynomial time. In the algorithm analysis the running time of it is explained which is $O(V^2)$ where V is the number of vertices in the input graph. In the experimental analysis, the running time of the algorithm is measured with respect to three different cases. The results of these measures are shown. Finally the algorithm is tested by using blackbox testing.

7) References

1. Pardalos, Panos & Rebennack, Steffen. (2010). Computational Challenges with Cliques, Quasi-cliques and Clique Partitions in Graphs.
2. Computational Challenges with Cliques, Quasi-Cliques and Clique Partitions in Graphs Panos M. Pardalos.
3. https://github.com/heyaroom/QeX/blob/e8b8b7ef32e0bc52369e407609dd8f2053d4c718/util/minimum_clique_cover/clique_cover.py
4. J. Bhasker, The Clique-Partitioning Problem, *Computera Math. Applic*. Vol. 22, No. 6, pp. l-11, 1881
5. https://www.geeksforgeeks.org/proof-that-clique-decision-problem-is-np-complete/