Partition into Cliques

Group 37

December 27, 2021

Introduction

"Partition into Cliques" problem is an undirected graph problem, which asks for the smallest number k that the graph can be partitioned with disjoint cliques. Clique is another name for the "complete graph" term, which is intensively used in graph theory. One other important aspect of this problem is that, it is NP-Hard; and also highly related with the problem "k-Coloring" in graphs. The problem is researched in theoretical computer science field, and used in areas such as computer networks and industrial engineering.

Showing that 'Partition into Cliques' problem belongs to the NP and NP-Hard classes, we need to prove that the 'decision problem' version of the problem is NP-Complete. The following proof is taken from R. M. Karp's '*On the Computational Complexity of Combinatorial Problems*'.

Let there be a set S consisting of nodes in the clique and S is a subgraph of G, checking if there exists a clique of size k in the graph takes $\mathcal{O}(k^2) = \mathcal{O}(|V|^2)$ time where $|V|$ is the number of vertices in G. Therefore, it has a polynomial time verifiability and hence, belongs to NP class.

Knowing that SAT (Boolean Satisfiability) problem is an NP-Complete problem, (proved by Cook's theorem), every problem in NP can be reduced to SAT. Let's call our 'Partitioning Into Cliques' problem as C, If S can be reduced to C in polynomial time for a particular instance, we would say that every NP problem can also be reduced to C in polynomial time, thus, proving that C is an NP-Hard problem.

Let the Boolean expression be $F = (x_1 \wedge x_2) \vee (\neg x_1 \wedge \neg x_2) \vee (x_1 \wedge x_3)$ Expression within each parantheses be a clause, C_1, C_2, and C_3 respectively. Considering the vertices as $< x_1, 1 >; < x_2, 1 >; < \neg x_1, 2 >; < \neg x_2, 2 >; < x_1, 3 >; < x_3, 3 >$ (second term denoting the clause number) and connecting these vertices such that no variable is connected to its complement and no two vertices which belong to the same clause are connected, we would get a graph G which is presented in Figure 1.

Considering the subgraph of G with the vertices $< x2, 1 >; < \neg x1, 2 >; <$

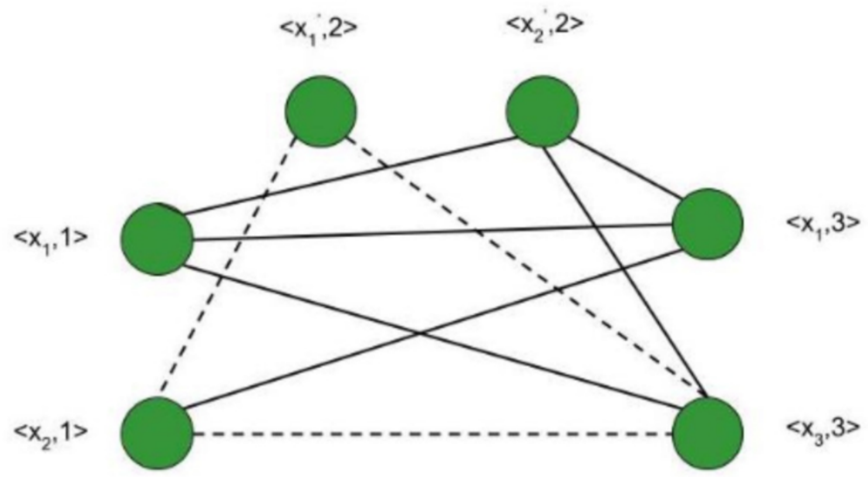

Figure 1: Graph G

$x3, 3 >$ which forms a clique of size 3, for the assignment $x1 = 0, x2 = 1, x3 = 1$, F evaluates to true. In other words, we get a max clique of size k when we have k clauses in our satisfiability expression. Moreover, for the corresponding assignment of values, satisfiability expression evaluates to true.

Hence, we showed that for a particular instance, SAT problem can be reduced into the 'Partition into Cliques' problem, meaning that our problem is NP-Hard.

Algorithm Description

Our group come with an algorithm for this problem. This algorithm is looking for every possible solution-candidate and returns the most optimal one.

First, we are storing every vertex in a list and find the power set of this list. We shall call this list 'PowerVertexSet'. Then, we will iterate over 'PowerVertexSet' and store all the elements that are constructing complete graphs. We will call this list 'CliqueSet'. Next, we will also find the power set of the set 'CliqueSet', namely 'PowerCliqueSet'. (We should denote that an element of the set 'PowerCliqueSet' is containing a set of complete graphs which are all subgraphs of the original graph, our problem instance.) We will, then, iterate over 'PowerCliqueSet' and check the elements whether the sets contained by the elements are disjoint and have all the vertices that the original graph has. If an element is not eligible for these two criterion, it can be ignored. Else, we will look for how many sets that the element contains. If an element which contains the least amount of sets -and also is eligible for the two criterion- contains k amount of sets, k will be returned.

The pseudocode will be given below.

The algorithm described above is correct, however significantly inefficient. For

Algorithm 1: Brute Force Algorithm

```
$Vertices \leftarrow$ Collect all vertices in the graph;
$PowerVertexSet \leftarrow$ Power set of $Vertices$;
$CliqueSet \leftarrow$ An empty set;
while $v \leftarrow$ Each element in $PowerVertexSet$ do
    if $v$ is a complete graph then
        Add v to $CliqueSet$;
    end
end
$PowerCliqueSet \leftarrow$ Power set of the $CliqueSet$;
$min \leftarrow len(Vertices) + 1$;
while $c \leftarrow$ Each element in $PowerCliqueSet$ do
    if $c$ contains all the vertices in the graph then
        if $c$ is made of disjoint sets then
            if $len(c) < min$ then
                $min \leftarrow len(c)$;
            end
        end
    end
end
return $min$;
```

dense graphs with vertex size of 10 or more, this algorithm can not come up with a solution in a reasonable time. Our group designed a sub-optimal algorithm, which works in polynomial time, described below.

Firstly, we look at every vertex and find their degrees. Then, we sort the vertices according to their degrees, from most to least. Then we will inspect the first vertex in our list - which has the highest degree -. We need to find a big clique that this vertex is a part of. Finding this big clique will be explained after this paragraph. After finding this clique, we need to store this clique and take this clique out of our graph. Then we will repeat the process - we will find the new degrees of the vertices that still exist in our graph, sort them and find a clique which the first vertex in our sorted list took part in and store the clique. - We will continue to do this process until the graph has no vertices left. Then the number of cliques we stored will give us the answer of the question.

Finding a Big Clique: Probabilisticly, the vertices with the higher degrees should construct bigger cliques. After we sort the vertices, we add to the next clique we will find the vertex with the highest degree. However, what are the other vertices that the clique is constructed by? We start by traversing the sorted list from the second element - since the first element is already in the clique ($CliqueSuggestion$) we will find -. If the vertex makes an edge with all of the vertices in $CliqueSuggestion$, we will add this vertex to the $CliqueSuggestion$; if not, we will continue traversing. We can find a clique by this method, however it is not a guarantee that this clique is relatively big. So to increase our chances for finding a big clique, we will do this operation $\lfloor(lg|V|)\rfloor$ times; in every loop we start traversing from the element index $1+loopCounter$ in the sorted list (loopCounter starts 1 and counts the loops). Then, we will look at every clique we found with these loops and select the biggest clique we found. We can just traverse the list once and select the clique we found, however, we are doing this operation $\lfloor(lg|V|)\rfloor$ times to increase our chances for finding a bigger clique.

The polynomial time algorithm is constructed by using the techniques of the 'Greedy Algorithm Design'. This algorithm is greedy; since in each step, we are picking out a clique that the vertex with the greatest degree took part in. It seems that the vertex with the greatest degree will construct the biggest clique - although it may not be -. By greedy choice, we always look for the cliques that the highest vertex took part in. Similarly, we are mostly looking for vertices with greater degrees to add in our cliques; which can be also considered 'greedy'. If a sub-optimal algorithm is sufficient, we can use this polynomial time algorithm to come up with an answer.

Algorithm Analysis

Brute Force Algorithm

The brute force algorithm is correct and works in $O(|V|(2^{2^{|V|}}))$ time, where $|V|$ denotes the number of vertices.

This correctness can be explained as: Suppose a graph $G = (V, E)$ has the solution k for this problem. So, this graph can be constructed from the subgraphs $G_{sub} = \{G_1, G_2, .., G_k\}$ where $\forall i, 1 \leq i \leq k, G_i$ is a complete graph. Also, all G_i are disjoint with each other. Suppose the graph G has n cliques, shown in the set $C = \{C_1, C_2, .., C_n\}$. Clearly $n > k$ and $\forall i, 1 \leq i \leq k, G_i \in C$. In our algorithm, we find the set C. Then, we will take the power set of C -namely 'PowerCliqueSet'. The set G_{sub} is an element of 'PowerCliqueSet'. When we are iterating over 'PowerCliqueSet', after ignoring the elements where the sets of it are not disjoint or it can not construct G; when the algorithm iterates over the element G_{sub}, it will store $|G_{sub}|$ as it has the least amount of sets. Then it will return $|G_{sub}|$ which is equal to k.

Algorithm 2: Greedy Algorithm

```
$ConfirmedCliques \leftarrow$ An empty set;
$Vertices \leftarrow$ Contains all vertices in the graph;
while $Vertices$ is not empty do
    $CliquestoConfirm \leftarrow$ An empty set;
    Sort $Vertices$ from greatest degree to lowest;
    $GreatestVertex \leftarrow Vertices[0]$;
    Remove $GreatestVertex$ from $Vertices$;
    $i \leftarrow 0$;
    while $i < lg(len(Vertices))$ do
        $CliqueSuggestion \leftarrow$ An empty set;
        $i \leftarrow i + 1$;
        Add $GreatestVertex$ into $CliqueSuggestion$;
        while $Ver \leftarrow$ Each element in $Vertices$ after $Vertices[i-1]$ do
            if $Ver$ has an edge with all elements in $CliqueSuggestion$ then
                Add $Ver$ to $CliqueSuggestion$;
            end
        end
        Add $CliqueSuggestion$ to $CliquestoConfirm$;
    end
    $BiggestClique \leftarrow$ Biggest clique in $CliquestoConfirm$;
    Add $BiggestClique$ to $ConfirmedCliques$;
    Remove $BiggestClique$ from Graph;
    $Vertices \leftarrow$ Collect all vertices in the graph;
end
return $len(ConfirmedCliques)$;
```

The complexity can be explained as: When we are finding the power set of the vertices, it will take $\mathcal{O}(2^{|V|})$ time as there are $2^{|V|}$ sets. When examining whether the elements of the power set are a complete graph or not, it will take $\mathcal{O}(|V|^2)$ each iteration. In the worst case, all the elements of the power set will be cliques. Then, we will take this power set's power set, namely 'PowerCliqueSet'. It will take $\mathcal{O}(2^{2^{|V|}})$ time. Then, we will iterate over 'PowerCliqueSet' where in every iteration we will check whether the element is constructed of disjoint sets and whether the element is including all the vertices that the graph has. Every iteration step will take $\mathcal{O}(|V|)$ time. In total, the algorithm will take $O(|V|(2^{2^{|V|}}))$ time.

Greedy Algorithm

The greedy algorithm returns a sub-optimal answer to the 'Partition into Cliques' question, working in polynomial time. It executes in $\mathcal{O}(|V|^3 lg|V|)$ time where $|V|$ is the amount of vertices.

The algorithm works by selecting a big clique that the vertex with the greatest degree is a part of, then taking this clique out and doing this procedure repeatedly until it terminates. We can show why the pseudocode/code is working just like the description of the algorithm given above: We are first sorting the vertices and selecting vertex with the greatest degree. Then we are storing this vertex in a set -We call this set $CliqueSuggestion$-. Later, we traverse all the other vertices; if a vertex is making an edge with all the vertices in $CliqueSuggestion$ add this vertex to the clique, else ignore the vertex. We traverse the vertices $\lfloor (lg|V|) \rfloor$ times, and selecting the biggest clique that in each loop we found. Then we are taking out that clique from the graph. We will continue this whole process until the graph has no vertices. This algorithm always terminates: At each step we are taking out at least one vertex, the vertex with the biggest degree. Therefore, the algorithm will terminate after at most $|V|$ loops.

This algorithm works in $\mathcal{O}(|V|^3 lg|V|)$ time. In the most inner loop, we are iterating over the $Vertices$ list to find out whether the vertex we are looking for will take part in the clique we will consider choosing. The list have at most $\mathcal{O}(|V|)$ elements and to decide whether the vertex will be in the $CliqueSuggestion$ or not takes $\mathcal{O}(|V|)$ time in the worst case. So in total, we will spend $\mathcal{O}(|V|^2)$ time. We will execute this loop $\mathcal{O}(lg|V|)$ times. Also, we have one more costly operation, which is sorting the vertices. It will take $\mathcal{O}(|V|^2)$ time. So, considering the most outer loop we will spend $\mathcal{O}(|V|^2 + |V|^2 lg|V|)$ time in every execution. Since the most outer loop executes at most $\mathcal{O}(|V|)$ times, this algorithm will take $\mathcal{O}(|V|^3 lg|V|)$ time in total.

Sample Generation

For generating sample graphs to use in our algorithms, we implement two sample generator algorithms, $randomGraph(vertexSize)$ and $randomGraph(vertexSize, density)$. Both algorithms work similarly, however, the latter algorithm also allows the user to pre-define the density of the graph. The algorithms are explained below. With the function input $vertexSize$, we first produce all the edges that a graph with $vertexSize$ amount of vertices possibly have, which is also equal to $(vertexSize) \times (vertexSize + 1) \div 2$. We store these -possible- edges in an array $possibleEdges$. Next step is implemented differently on both algorithms. In the former algorithm, we choose a random number between 0 and $(vertexSize) \times (vertexSize + 1) \div 2$ - both inclusive -. Suppose this chosen number will be n. Then, we choose n distinct elements from $possibleEdges$. These selected edges will be the edges our graph contains. After some formatting, we output the graph. In the latter algorithm, the amount of edges that the graph will have are not randomized, instead the user determines with the parameter $density$: $\lfloor density \times (vertexSize) \times (vertexSize + 1) \div 2 \rfloor$ will give us the edge amount. Density should be a real number between 0 and 1. Later, we can choose determined amount of edges from $possibleEdges$ and after some formatting, output the graph. Pseudocodes for the both algorithms are given below.

Algorithm 3: randomGraph(vertexSize)

```
$possibleEdges \leftarrow$ Contain all possible edges in a graph;
$graph \leftarrow$ An empty list;
$i \leftarrow 1$;
$N \leftarrow random(0, len(possibleEdges))$;
$edgeList \leftarrow randomSample(possibleEdges, N)$;
while $i \neq vertexSize + 1$ do
    $edges \leftarrow$ An empty list;
    $j \leftarrow 0$;
    while $j \neq N$ do
        if $edgeList[j]$ $contains$ $i$ then
            Add the other vertex in $edgeList[j]$ to $edges$;
        end
        $j \leftarrow \jmath + 1$;
    end
    Add $[i, edges]$ to $graph$;
    $i \leftarrow i + 1$;
end
return $graph$;
```

Algorithm 4: randomGraph(vertexSize, density)

```
$possibleEdges \leftarrow$ Contain all possible edges in a graph;
$graph \leftarrow$ An empty list;
$den \leftarrow int(density \times len(possibleEdges))$;
$i \leftarrow 1$;
$edgeList \leftarrow randomSample(possibleEdges, den)$;
while $i \neq vertexSize + 1$ do
    $edges \leftarrow$ An empty list;
    $j \leftarrow 0$;
    while $j \neq len(edgeList)$ do
        if $edgeList[j]$ $contains$ $i$ then
            Add the other vertex in $edgeList[j]$ to $edges$;
        end
        $j \leftarrow \jmath + 1$;
    end
    Add $[i, edges]$ to $graph$;
    $i \leftarrow i + 1$;
end
return $graph$;
```

Experimental Analysis of The Performance

The experimental results are presented below. The computer's specs in which the experiments conducted are listed: Windows 10 OS, Intel Core i7-9750H CPU @ 2.6GHz, 8GB RAM, x64 based system.

We did running time experiments on the Greedy Algorithm, which is presented in Figure 2. As input sizes, $n = 100, 200, 300, 400, 500, 600$ are used. For each input size, 15 different graphs are used. For each input size, the mean running time is shown with a red dot.

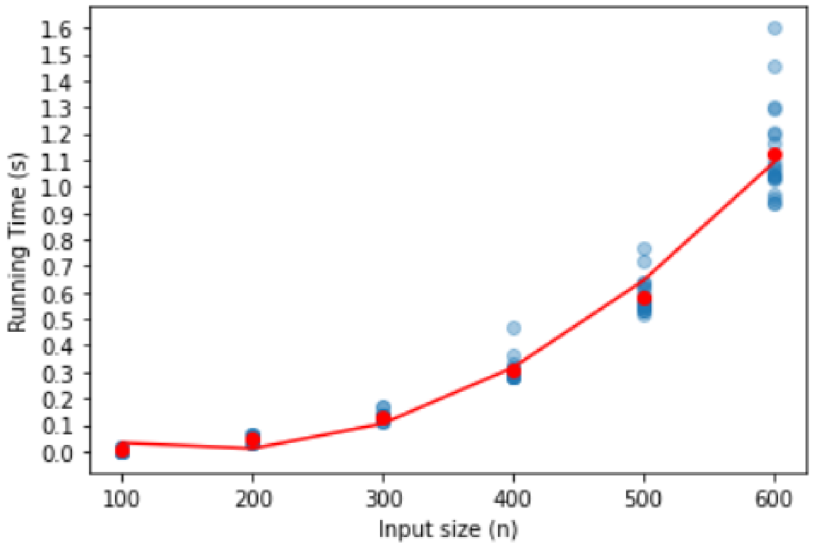

Figure 2: Experimental Results for Greedy Algorithm

If we inspect the running-time graph for Greedy Algorithm, the line is not linear. We found the running time is $\mathcal{O}(|V|^3 lg|V|)$ theoretically and the running-time graph is confirming this result.

Similarly, Brute Force algorithm is examined with various experiments. We have to use much smaller input sizes as the algorithm was significantly slow. For input sizes, $n = 2, 3, 4, 5, 6, 7$ are used. For each input sizes, due to slow execution speed, only 5 random graphs are given as inputs. Additionally, we have to use log-scaling in plotting for the y-axis (Running time). The mean running time is shown with a red dot. Experiment results are presented in Figure 3. As we can see from the scatter plot, the points are linearly increasing. Since we used semi-log scaling, we can confirm that the Brute Force algorithm is exponential. We did not use linear fitting line in the graph since the results for input $n = 2$ made the line unfitting for the rest of the graph. Clearly for small inputs, the computer might not calculate the time precisely or there could be other

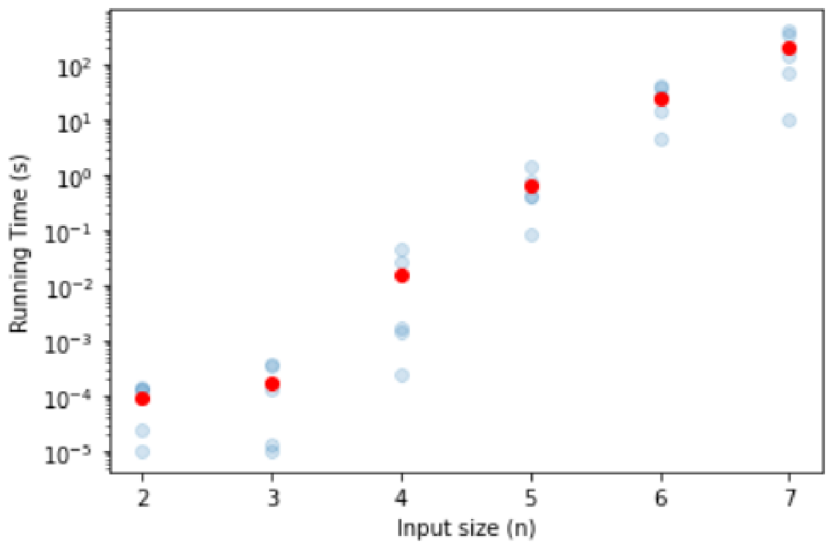

Figure 3: Experimental Results for Brute-Force Algorithm

executing programs in the OS causing the algorithm run slowly. Therefore, the results for input $n = 2$ can be considered as 'not accurate'.

Experimental Analysis of the Correctness

We applied both black box testing methods and white box testing methods to both algorithms, Brute Force algorithm and Greedy Algorithm.

Black Box Testing

We considered some of the edge cases for this part of the functional testing. These tests covered some of the extreme graphs such as:
- A graph with no vertices and edges
- A graph with only one vertex
- A graph with different amount of vertices but no edges
- Cycle graphs
- Bipartite graphs
- Complete graphs
- Graphs made of exactly n amount of disjoint cliques
- Random graphs that the $randomGraph()$ function produced

White Box Testing

With white box testing, we try to cover all the statements in every paths in both of the algorithms. We created some special test cases for covering all of the decisions in the algorithms, however, we also used our two random graph generators to create some inputs.

Discussion

In this report, we showed two algorithms for the ”Partition into Cliques” problem: Brute Force Algorithm and Greedy Algorithm. We analyzed these algorithms first theoretically, then experimentally. We found out that the theoretical and experimental results are overlapping, except for very small input sizes. Both algorithms are working as intended, we inferred this fact from the correctness experiments we conducted, but there could be some improvements. We believe that the Brute Force algorithm can be implemented with $\mathcal{O}(2^{|V|})$ time complexity. Also, these algorithms might work faster if we know some information about the input beforehand. For example, if the algorithms know that a complete graph will be given as an input, algorithms can conclude that the answer is 1 without any computation.

References

Garey, M., & Johnson, David S., joint author. (1979). Computers and intractability : A guide to the theory of NP-completeness (A Series of books in the mathematical sciences). San Francisco: W. H. Freeman.

Karp, R. M. (1975). On the computational complexity of combinatorial problems.