3005 Project Report

Teyfik Yılmaz - Faruk Berk Öztürk - Necip Melik Bilyay

January 2, 2023

Abstract

Using this line of code, you may estimate the size of the greatest clique that occurs inside an undirected graph. A clique is a subset of vertices that are connected by edges, and the largest clique that is presently known to exist is a clique. The user is required to provide two numerical inputs: the total number of vertices in the graph (n) and the minimal distance between vertices in the clique with the most members (dimension). Following that, the computer program does an analysis on any binary strings that are longer than n characters. This is done in order to calculate the Hamming distance between each string and every other string in the clique. If the Hamming distance between a string and any other string in a clique is smaller than the clique's dimension, the string is not added to the clique. The string has been provided in this case due to the requirement that the distance be greater than the dimension. The size of the clique may be established by calculating the number of persons who are members of the group with the most total members. The goal of this portion of code is to find the size of the clique with the most nodes in a graph. A maximal clique is one that contains no subsets of other cliques in the graph and cannot be increased by adding an adjacent vertex. This kind of clique has reached its maximum size and cannot be expanded.

1 Purpose?

This portion of code is used to determine the size of the largest clique that can be found in a graph. In an undirected graph, a clique is a subset of vertices that exists when all of the various vertices that make up the clique are near to one another, which means that they are linked to one another by an edge of the network. In other words, a clique exists when all of the various vertices that make up the clique are connected to one another. To put it another way, a clique is present when each of the several vertices that comprise the clique is linked to every other vertex in the clique. A maximum clique is a clique that cannot be expanded by adding an adjacent vertex; in other words, a maximal clique is a clique that is not a subset of any other clique in the graph. Another way to think of this is that a maximum clique is a clique that cannot be expanded by adding an adjacent vertex. A maximum clique is a clique that cannot be increased by adding an adjacent vertex. This is an alternative method of thinking about what a maximum clique is. It is possible for each particular graph to have no more than a single maximal clique at any given instant.

2 How Does It Work ?

The user will be prompted to input two integers, namely the numberOfbits value and the dimension, in the specified sequence. This prompt will be displayed to the user. The value that is assigned to the variable numberOfBits indicates the total number of vertices in the graph, while the value that is assigned to the variable dimension specifies the minimum distance that must be present between vertices in the largest clique.

```
class hammingGraph{
private:
    std::unordered_set <std::string> maximalClique;
    std::vector<std::string> binaryVector;
    std::vector<int> maximalVector;
    int numberOfBits;
    int dimension;
public:
    hammingGraph(int numOfBits, int dim);
    hammingGraph():numberOfBits(0),dimension(0){}
    int hammingDistance(const std::string& str1,const std::string& str2);
    void hammingGraphCreator();
    int findMaximalClique();
    void printMaximal();
    static int binaryToDecimal(const std::string& str);
};
```

The software first generates a hammigGraph object, then calculates the maximalClique value based on the information provided by the user, and then outputs the clique's length as well as its node locations on the screen.

3 Hamming Distance

```
int hammingGraph::hammingDistance(const std::string& str1, const std::string& str2) {
    if(str1.length() != str2.length()){
        std::cerr << "ERROR Bits size must be equal !!" << std::endl;
        return -1;
    }
    int distance = 0;
    int size = (int)(str1.length());
    for (int i = 0; i < size; ++i) {
        if(str1[i] != str2[i]){distance++;}
    }
    return distance;
    }
```

This code segment contains a function that estimates the Hamming distance between two strings. Hamming distance refers to the number of distinct character combinations that exist between two strings. For example, the Hamming distance between the words "dog" and "cat" is merely 2, as there are only two different letters used in each word. This procedure begins by determining whether or not the lengths of the two strings are identical. If they differ, an error notice is generated and the function returns -1. The length of the two strings can be stored in a variable if it is possible to detect if they are equal. Then, a for loop is utilized to compare each character in each string to every other character in the other string. If there is a discrepancy between the characters, an additional value is added to the distance variable. This procedure is executed for every character present in each of the strings. The final step is returning the Hamming distance variable.

3.1 Hamming Graph Creator

```
void hammingGraph::hammingGraphCreator() {
    for (int i = 0; i < (1 << numberOfBits); i++) {
        std::string str;
        int num = i;
        while (num > 0) {
            str.insert(str.begin(), (num & 1) + '0');
            num >>= 1;
        }
        while (str.size() < numberOfBits) {
            str.insert(str.begin(), '0');
        }

        binaryVector.push_back(str);

    }
}
```

This section of code represents a function for generating a Hamming graph. Hamming graph refers to a graph consisting of a set of numbers, each of which is represented by a bit string. The bit string is the binary version of the number, which can be conceptualized as follows. "101" is one possible binary representation of the numeral 5. This is but one instance. This method creates a loop from 0 to (1 numberOfBits - 1) inclusive. This phrase generates the binary representation of the specified number of integers and specifies that there are "numberOfBits" bits. Additionally, it signifies that there are this many bits. For example, if "numberOfBits" is set to 4, this loop will output the binary representation of integers between 0 and 15, and it will contain values that fall inside this range. During the loop's execution, a string is generated to generate the binary representation of a number, and the value of this integer is placed in the "num" variable. When "num" is greater than zero, the binary representation is created by dividing the integer by 2 and generating a bit string. This is true provided that "num" is not equal to 0. As an illustration, the binary equivalent of the integer 5 is the number "101." The "binaryVector" variable is then augmented with this bit string's contents. If the length of the produced bit string is shorter than "numberOfBits", the bit string is extended to "numberOfBits" length. This only occurs if the length of the produced bit string is less than "numberOfBits". This ensures that all bit strings have the same length, which is required for the Hamming graph's creation.

3.2 Finding and Creating The Maximal Clique

```
int hammingGraph::findMaximalClique() {
    int maxClique = 0;
    for (int i = 0; i < (1 << numberOfBits); i++) {
        bool isOkay = true;
        for(auto iterator= maximalClique.begin();iterator != maximalClique.end(); iterator++)
            {
            const std::string & vector = *iterator;
            if (hammingDistance(binaryVector.at(i), vector) < dimension) {
                isOkay = false;
                break;
            }
        }
        if (isOkay) {
            maximalClique.insert(binaryVector.at(i));
            maximalVector.push_back(binaryToDecimal(binaryVector.at(i)));
        }
    }
    maxClique = (int)maximalClique.size();
    return maxClique;
}
```

This line of code implements a mechanism for identifying the Hamming graph with the greatest clique. When considering a graph, a clique can be conceptualized as a subgraph in which all of the nodes are interconnected.

This method creates a loop from 0 to (1 numberOfBits - 1) inclusive. This phrase generates the binary representation of the specified number of integers and specifies that there are "numberOfBits" bits. Additionally, it signifies that there are this many bits. For example, if "numberOfBits" is set to 4, this loop will output the binary representation of integers between 0 and 15, and it will contain values that fall inside this range.

It decides whether or not a node (binaryVector.at(i)) can be part of the clique while iterating through the loop. If the node can be included, it is appended to the "maximalClique" variable, and its hexadecimal representation is appended to the "maximalVector" variable. If the node is ineligible for inclusion, it is not added to either of these variables. The "isOkay" variable will be set to false if the node cannot be included, at which point the loop will end.

Finally, the value of the "maxClique" variable is returned after its size is extracted from the "maximalClique" variable and then assigned to "maxClique." This variable reflects the size of the largest feasible clique in the Hamming graph.

4 Complexity

When creating efficient code, time and space complexity are of utmost importance. Originally, this code was intended to establish a balance between these two elements: For this, we used the "unordered set" structure. The "unordered set" structure stores strings similarly to a hash table. This structure stores strings in randomly selected positions, therefore lookups often require O(1) time on average. This structure is quicker than a "set" structure since a "set" structure operates like a binary search tree and searches need O(log n) time. In addition, the "unordered set" structure needs less memory than the "set" structure, while a binary search tree structure takes more memory. For search operations, thus, the "unordered set" structure is economical in terms of memory and speed.

4.1 Time Complexity

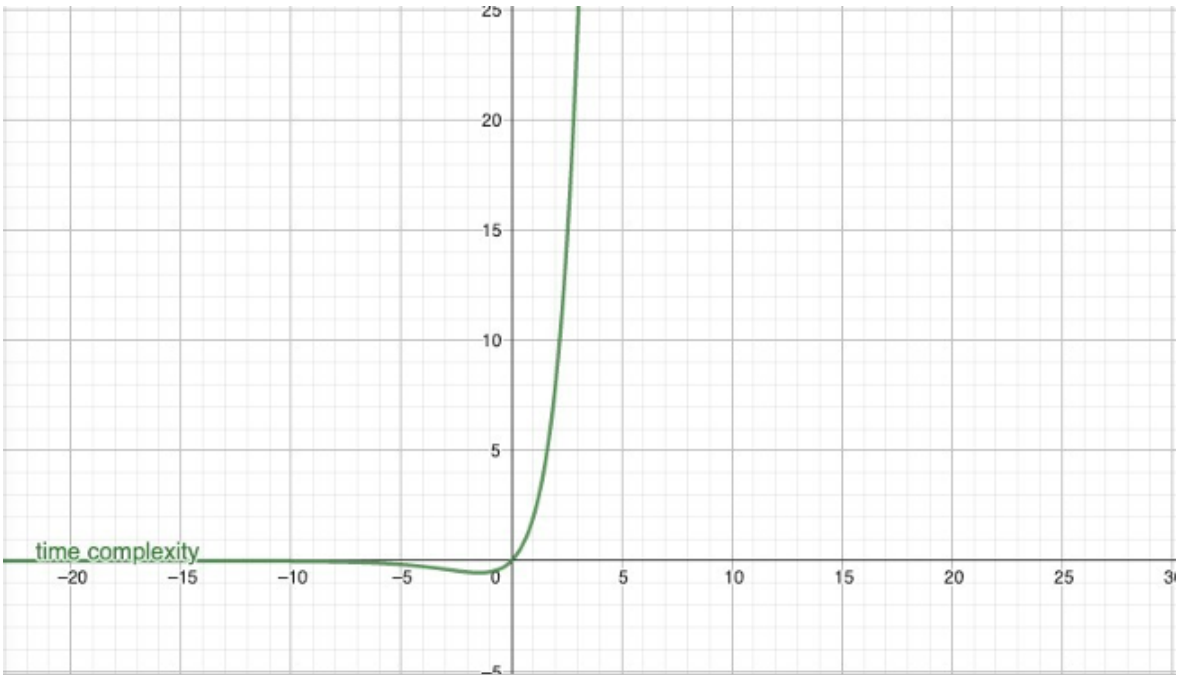

Figure 1: Time Complexity

The time complexity of this code is O(n * 2^n). This is because for each binary string, the algorithm must iterate through all 2^n potential binary strings and for each binary string, it must iterate through all n-length binary strings. This results in a total of n * 2^n steps. This is dependent on the variable n, which is the number of bits in the binary string, and 2^n, which is the number of binary strings. Therefore, the time complexity is O(n * 2^n).

4.2 Space Complexity

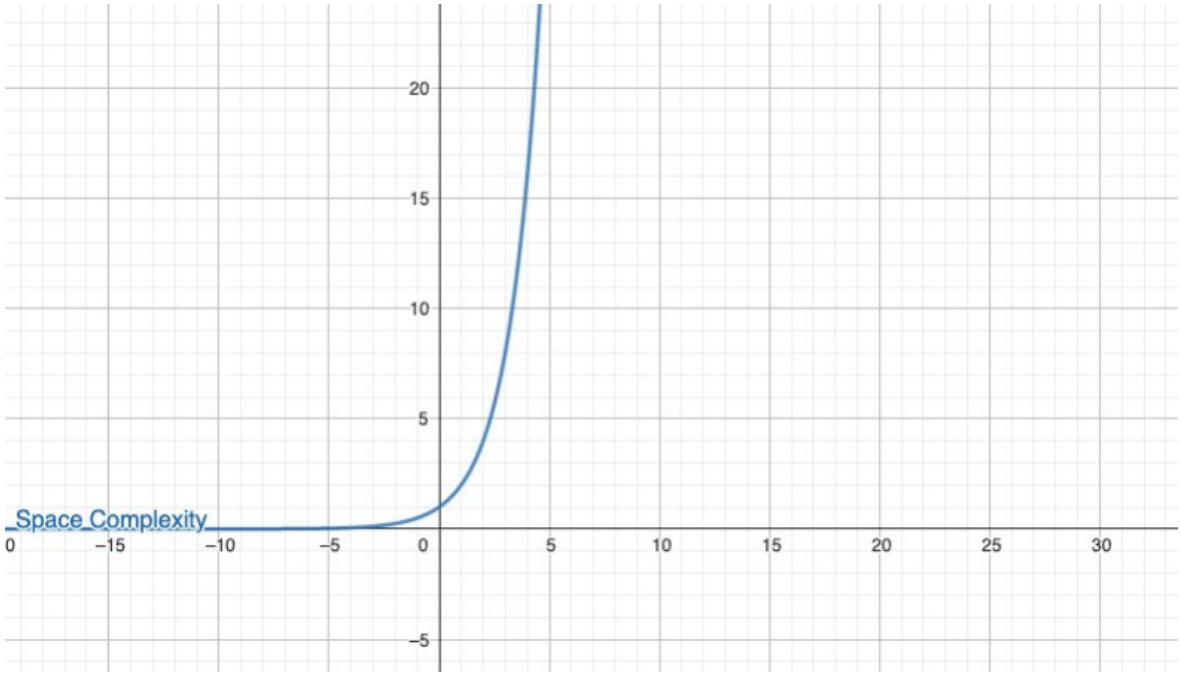

Figure 2: Space Complexity

Space complexity: The space complexity of this code is O(2^n). This is because the "unorderedset" data structure is used to store all 2^n potential binary strings. This data structure uses a hash table-like structure to store the strings and requires O(1) time for insertion, deletion, and search operations. However, the total space of the "unorderedset" structure will be O(2^n), because it needs to store 2^n binary strings. Therefore, the space complexity is O(2^n).