BAU
Bahçeşehir University

CMP 3005 Term Project

MAXIMAL CLIQUE PROBLEM USING HAMMING GRAPH AND HAMMING DISTANCE

Student Name	Student ID
Umniyah Sameer Haitham Abbood	1900812
Sevval Ozlem Carkit	1803772
Mehmet Emre Astarcioglu	1904454

1. Introduction

This project aims to find an efficient algorithm to compute the maximal clique in the

Hamming graph.

The concept of the **Hamming distanc**e was introduced by Richard Hamming introduced, which is a measure of the difference between two equal-length strings of symbols. According to Richard Hamming, the Hamming distance is calculated by counting the number of positions where the corresponding symbols are not the same. The scientist introduced the concept of the Hamming distance in his 1950 paper "Error Detecting and Error Correcting Codes," which was published in the Bell System Technical Journal [1].

After the hamming distance concept, the concept of the **Hamming graph** emerged and is named after Richard Hamming. Hamming graphs are a type of graph used in computer science and mathematics. In a Hamming graph, the vertices represent elements in a set, and two vertices are connected by an edge if the corresponding elements are a certain distance apart, as measured by the Hamming distance. Hamming graphs have a number of interesting properties and have been studied in a variety of contexts, including in the study of error-correcting codes, coding theory, and graph theory [2].

The **clique problem** was offered by Richard Karp in his 1972 paper "Reducibility Among Combinatorial Problems," which was published in the journal Complexity of Computer Computations [3].A **clique** is a subset of vertices of a graph such that every two distinct vertices in the clique are adjacent, meaning they are connected by an edge.

A **maximal clique** is a clique that is not a subset of any other clique in the graph. The maximal clique is an NP-hard type of problem, however, there are several different algorithms that have been developed for solving the maximal clique problem, including brute-force algorithms, heuristics such as the maximal clique enumeration algorithm and Bron-Kerbosch algorithm, and exact algorithms branch-and-bound algorithm.

2. İmplementation

Our project is implemented using Java programming language. We have implemented a brute force algorithm with a time complexity of O(2^n * n).

Our *CMP3005Projec*t Class has 4 methods:

1.generateBinaryNumbers(): the method returns an array of binary numbers of a given length which takes it as a parameter.

The method begins by declaring the variables

- String array called *binaryNumbers* and initializing it with its size using the bit-shifting operator (1 << n). This array will be used to store the generated binary numbers
- Char array called *binaryNum* and initializing it with its size to n which is the length. This array will be used to store the current binary number being generated in each iteration in the for a loop.

Then we have 2 nested loops

- an outer loop that iterates 2^n times, with the loop variable i ranging from 0 to $2^n - 1$ which is 1 << n. each iteration creates a new String object from the *binaryNum* char array and stores it in the ith index of *binaryNumbers* string array.
- an inner loop that iterates n times, with the loop variable j ranging from 0 to n - 1. The inner loop sets the value of the jth index of *binaryNum* to either '0' or '1' based on the value of i & (1 << j). If the result is 0, the jth bit of i is 0. If the result is not 0, the jth bit of i is 1

After the nested for loop finishes, the method returns the binaryNumbers string array which contains all the possible combinations of the binary number represented in n bits

The time complexity of this method is O(2^n).

2.maximalClique(): This method have 2 parameters (d: min Hamming Distance for vertices, vertices[]: all possible binary strings for the specified digit) and returns the size of maximal clique that can be created with all strings of the specified digit for the specified minimum Hamming Distance.

This method works like this:

- ○ First of all, it has an array list to store the items in the Hamming Graph. This array list is defined as *maximalClique*.
- ○ And then the first item is added to *maximalClique* to be able to compare the items in this array.
- ○ Then we created 2 nested for loops to compare each item in *maximalClique* with other items.
- ○ In this loop we create a boolean variable to be able to add items to *maximalClique* and when this is true we add it to *maximalClique*.
- ○ We checked Hamming Distance while adding it to *maximalClique*. For this reason, we used the *hammingDistance* method we created.
- ○ If the distance between the two compared items is less than *d*, we did not add it to the list. If it is greater than or equal to *d*, we add it to the *maximalClique* in the other if block we created.
- ○ And finally, our method returns the size of the maximal clique, which is the output of our project.

The time complexity of this method is $O(2^n * n)$. Because the 2 for loops inside the maximalClique method look at each string first and then the digits inside each string.

3. hammingDistance(): This method returns the hamming distance between 2 strings.

- So how to find the hamming distance between 2 strings?
- We will explain this through an example:
- Let's say we have the strings “0010” and “1011”. Hamming distance is the number of different digits between these 2 strings. So the hamming distance for these strings will be 2. Because only the numbers in the 1st and 4th digits are different.

Now let's go back to the method we created.

Method works like this:

- Our method has 2 string type parameters.
- And the variable *distance* is the variable that will store the distance between these 2 strings.
- The for loop we created compares each element in both strings one by one and does not take any action if they are the same, if they are different, they add them to the distance variable.
- And finally the *distance* variable is returned by the method.
- There is no reason the loop should run based on the length of the first string. In our project, we chose it because both strings will have the same length.

The time complexity of this method is O(length of the string).

Finally, let's look at the main method and the output of the code:

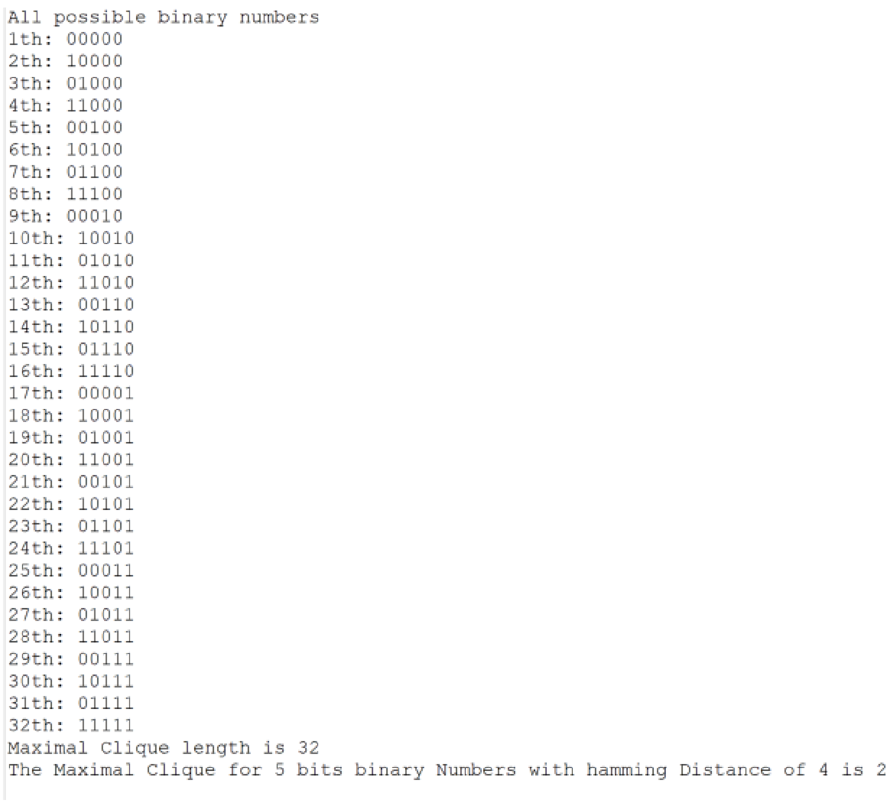
4. main():

Here, we first used our generateBinaryNumbers method to generate all possible strings at the specified digit. It was stated in the project document that 2 strings could be created with a minimum Hamming Distance of 4 for all 5-digit binary numbers. That's why we used it.

Then, with the for loop, we made all possible 5-digit binary numbers appear on the screen.

Finally, using our maximalClique method, we displayed the size of the 5-digit maximal clique with a minimum Hamming Distance of 4 and the strings in the maximal clique on the screen.

Output of the code:



```
All possible binary numbers
1th: 00000
2th: 10000
3th: 01000
4th: 11000
5th: 00100
6th: 10100
7th: 01100
8th: 11100
9th: 00010
10th: 10010
11th: 01010
12th: 11010
13th: 00110
14th: 10110
15th: 01110
16th: 11110
17th: 00001
18th: 10001
19th: 01001
20th: 11001
21th: 00101
22th: 10101
23th: 01101
24th: 11101
25th: 00011
26th: 10011
27th: 01011
28th: 11011
29th: 00111
30th: 10111
31th: 01111
32th: 11111
Maximal Clique length is 32
The Maximal Clique for 5 bits binary Numbers with hamming Distance of 4 is 2
```

3. Better Implementations

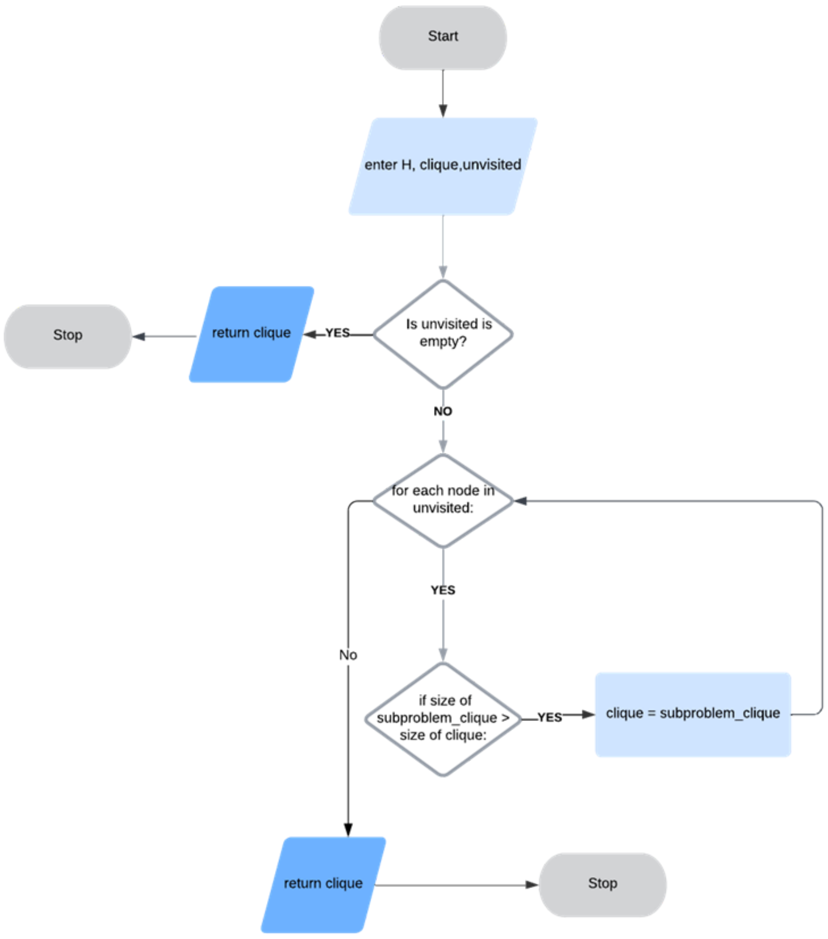
1. Exact solution

○ **Branch and bound** is a technique for finding the optimal solution to a problem by dividing it into smaller subproblems, solving the subproblems, and then combining the solutions to obtain the optimal solution to the original problem. It is a type **of divide-and-conquer algorithm**. Ailsa Land and Alison Doig developed this method while conducting research sponsored by British Petroleum at the London School of Economics in 1960 for discrete programming [4]. The time complexity is O(b^d), where b is the branching factor and d is the depth of the search tree. The space complexity is O(bd). It is not worth implementing and using as it is an exact solution it will not work with large input, there are better solutions such as **the Bron-Kerbosch algorithm,** which has worst-case time complexity $O(3^{n/3})$ or $O(2^{0.528n})$ [5]. The idea of the algorithm of branch and bound algorithm for finding the maximal clique in a graph, we initialize an empty clique and a list of unvisited nodes in the graph. Then, we pick a node from the list of unvisited nodes and add it to the clique. We remove all nodes that are not adjacent to the node added in the previous step from the list of unvisited nodes. If the list of unvisited nodes becomes empty, we return the clique as the maximal clique. If the list of unvisited nodes is not empty, we divide it into two subproblems by selecting a node from the list and creating two new lists: one containing all nodes that are adjacent to the selected node and the other containing all other nodes. We recursively apply the branch and bound algorithm to each of the two subproblems, Finally, we return the larger of the two cliques as the maximal clique for the subproblem.

Pseudocode of the Branch and bound algorithm

```
function maximal_clique(H, clique, unvisited):
  if unvisited is empty:
    return clique
  else:
    for each node in unvisited:
      clique' = clique + [node]
      unvisited' = [n for n in unvisited if n is adjacent to node]
      subproblem_clique = maximal_clique(H, clique', unvisited')
      if size of subproblem_clique > size of clique:
        clique = subproblem_clique
    return clique
```

Flowchart of the Branch and bound algorithm



Start
enter H, clique,unvisited
Is unvisited is empty?
YES
return clique
Stop
NO
for each node in unvisited:
YES
No
if size of subproblem_clique > size of clique:
YES
clique = subproblem_clique
return clique
Stop

2. Heuristic solution

- **Bron-Kerbosch algorithm:** is an algorithm for finding all cliques (i.e., fully connected subgraphs) in an undirected graph. It is named after Coen Bron and Joep Kerbosch, who published it in 1973.

The algorithm has two versions. The technique is implemented simply in the first version, and all cliques are generated in alphabetical (lexicographic) order and based on the first, the second version attempts to reduce the number of branches that must be traveled by generating cliques in a somewhat random sequence [7]. The same reference asserts that the latter version has performed better with graphs that have an abundance of cliques in performance tests.

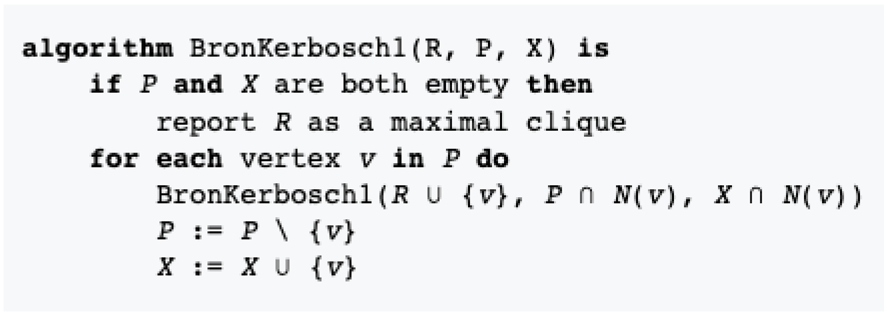
Three disjoint sets of nodes R, P, X are utilized by the algorithm. The first set R is for containing potential cliques. The second set P contains vertices that are connected with all vertices in R and it extends R with each iteration. The set X is for already visited vertices from previous Ps; in other words, vertices in X have already been found in all maximal cliques they could be found. This algorithm works recursively.

In the first version, the algorithm takes nodes from set P one by one and places nodes to set R and if the node has edges with all the vertices in the R then the node is placed in R. Intersection of set P and neighborhoods of current node is taken and it becomes a new state of P. Same applies for X and intersection of X and neighborhoods of the current node is taken and it becomes a new state of X. After all those, the function is called recursively with a new state of the sets. Function halts when set P and X becomes empty, this will report R as a maximal clique. After a recursive call, the current vertex is removed from P and added to set X to indicate that this vertex was processed before.

For the second version, the algorithm chooses pivot vertex u from set P for each recursive call. Non-neighbor nodes of u from set P are looped and like the first version, those vertices picked one by one and added to set R and neighbors of the current vertex and P are taken intersections and the same applies for set X. With pivoting less recursive call is made and as mentioned above this version performs better than the version.

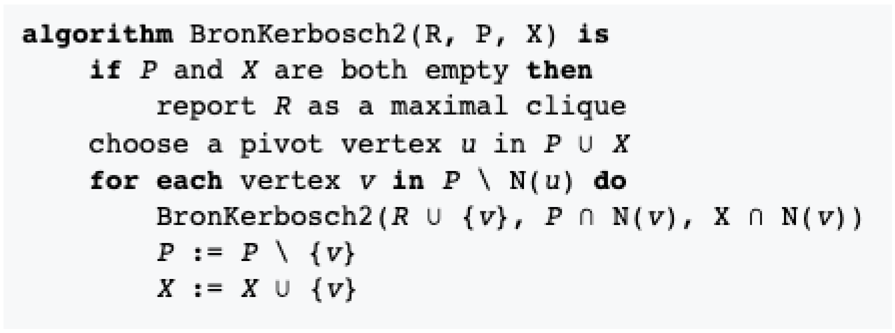
The Bron-Kerbosch algorithm is relatively efficient, with a time complexity of O(3^(n/3)).

Pseudocode of Bron-Kerbosch Algorithm without pivoting



```
algorithm BronKerbosch1(R, P, X) is
    if P and X are both empty then
        report R as a maximal clique
    for each vertex v in P do
        BronKerbosch1(R ∪ {v}, P ∩ N(v), X ∩ N(v))
        P := P \ {v}
        X := X ∪ {v}
```

Pseudocode of Bron-Kerbosch Algorithm with pivoting



```
algorithm BronKerbosch2(R, P, X) is
    if P and X are both empty then
        report R as a maximal clique
    choose a pivot vertex u in P ∪ X
    for each vertex v in P \ N(u) do
        BronKerbosch2(R ∪ {v}, P ∩ N(v), X ∩ N(v))
        P := P \ {v}
        X := X ∪ {v}
```

- **maximal clique enumeration:** this is an algorithmic problem that involves finding all the maximal cliques in an undirected graph, the algorithm can be implemented in several ways few of which are the following**:**

  1. **Basic backtracking:** backtracking is used to systematically explore all the possible combinations of vertices that could form a maximal clique in the graph. The algorithm starts with an empty clique and adds vertices to it one at a time, recursively searching for larger cliques that include the added vertex. If the search reaches a point where it is not possible to add any more vertices to the clique without violating the definition of a clique such as without introducing any non-adjacent vertices, the algorithm backtracks and continues the search with a different vertex. This process continues until all possible cliques have been explored, and the resulting set of cliques is returned as the solution to the maximal clique enumeration problem [6]**.**

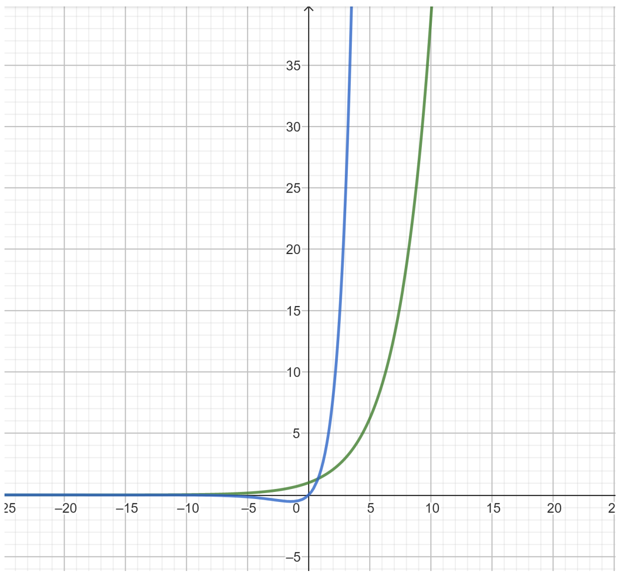
  2.Intelligent backtracking: this algorithm is similar to basic backtracking with 2 improvements. Firstly, by using knowledge of the maximal clique size to improve the efficiency of the search process. Specifically, the algorithm checks the number of vertices in the current search tree and prunes the search tree if there are fewer than the maximal clique size. This can help the algorithm find a maximal clique more efficiently by eliminating paths that cannot lead to a valid solution. Secondly, using Color preprocessing which is a technique that involves partitioning the vertices of the graph into groups based on some standards, and using this information to reduce the size of the search space. For example, if two vertices cannot be part of the same clique because they have different colors, we can eliminate one of them from the search space. This can help the algorithm find a maximal clique more efficiently by reducing the number of vertices that need to be considered [6].

  In general, intelligent backtracking algorithms can be more efficient than basic backtracking algorithms but may require more computational resources and may be more complex to implement. Basic backtracking algorithms are often simpler and easier to implement but may not be as efficient in certain cases.

4. Comparison Between Implementations

n	Our implementation	Bron-Kerbosch algorithm
1	O (2 * 1) = O (2).	O (3^ (1/3)) = O (1).
10	O (2^10 * 10) = O (10240).	O (3^ (10/3)) = O (27).
100	O (2^100 * 100) = O(1.26e+30).	O (3^ (100/3)) = O (3^33) = O(3.67e+10).
1000	O (2^1000 * 1000) = O(infinity).	O (3^ (1000/3)) = O (3^333) = O(infinity).

5. Comparison Between Time Complexities Using Graphs



The time complexity graphs of the Brute Force and Bron Kerbosch algorithms are as shown in the adjacent graph. The green color represents the time complexity of the Bron Kerbosch algorithm and the blue color the Brute Force algorithm.

In other words, the Bron Kerbosch algorithm is more useful than Brute Force.

Brute Force Algorithm Time Complexity: O(2^(x)*x)

Bron Kerbosch Algorithm Time Complexity: O(3^(x/3))

References

[1] R. W. Hamming, "Error detecting and error correcting codes," Bell System Technical Journal, vol. 29, no. 2, pp. 147-160, 1950.

[2] N. J. A. Sloane, "Unsolved problems in graph theory arising from the study of codes," Graph Theory Notes of New York, vol. 18, pp. 11-20, 1989.

[3] R. M. Karp, "Reducibility among combinatorial problems," Complexity of Computer Computations, pp. 85-103, 1972.

[4] A. H. Land and A. G. Doig, "An automatic method of solving discrete programming problems," Econometrica, vol. 28, no. 3, pp. 497-520, 1960. [Online]. Available: http://dx.doi.org/10.2307/1910129.

[5] R. H. C. Yap and M. Y. J. Tan, " Branch and Bound Algorithm for Finding the Maximum Clique Problem" in Proceedings of the International Conference on Industrial Engineering and Operations Management, pp. 2-7, 2018. [Online]. Available: https://ieomsociety.org/ieom2018/papers/648.pdf.

[6] J. D. Eblen, C. A. Phillips, G. L. Rogers, and M. A. Langston, "The maximum clique enumeration problem: algorithms, applications, and implementations," in 7th International Symposium on Bioinformatics Research and Applications (ISBRA'11), Changsha, China, 27-29 May 2011.

[7] C. Bron, J.A.G.M. Kerbosch and H. J. Schell, “Finding cliques in an undirected graph” in TH Eindhoven. ORS, Vakgr. operationele research : rapport, Vol. BKS-1, Technische Hogeschool Eindhoven, 1972