CMP 3005 Project Report

Ege Eke 1904511
Computer Engineering
ege.eke@bahcesehir.edu.tr

Çağatay Akkaş 2004204
Computer Engineering
cagatay.akkas@bahcesehir.edu.tr

Fırat Kutluhan İslim 1803509
Computer Engineering
firatkutluhan.islim@bahcesehir.edu.tr

***Abstract*—In this project, we developed an algorithm for finding the maximal clique in a given bit length and minimal distance using the Hamming distance function. Our approach involves constructing a Hamming graph which includes vertices and edges and using dynamic programming recursion to calculate every possible outcome of the system and return the maximum value. Our final algorithm has similar time complexity and accuracy to existing approaches like Bron-Kerbosch, with time complexity of (2^n * n^2) and an accuracy rate of %100 until we hit the memory limitage.**

***Index Terms*—Hamming Distance , Recursion , Maximal Clique, Hamming Graph , NP-Hard , Algorithm**

I. Introduction

Finding the maximal clique is a well-known, NP-hard problem that involves finding the largest subset of vertices in a Hamming Graph, which includes vertices and edges, such that every two vertices in the subset are connected by an edge. Many algorithms have been developed to solve this problem, including the Bron-Kerbosch algorithm and various graph-theoretic approaches. However, these algorithms have certain limitations.

II. First Approach

In our initial code, we attempted to solve the problem by creating our own recursive algorithm that calculates every possible clique subset and returns the maximum value. In the main function, we get the binary string length as input(”n”), with converting every integer from 0 until 2^n -1 to binary, we calculate the Hamming distance between every pair of nodes in the array. Pairs with a Hamming distance greater than a specified “d” value which we also get it as input in main function, were stored with their values in an adjacency array to define the edges of the graph. To solve the problem of storing the edges in both directions, we implemented a boolean variable. When an edge is iterated in the other direction, the storing process is stopped. Our function for finding the maximal clique tries to evaluate each node by increasing the node value (”cliqueCluster”) after every call and recursively calling itself until node count is reached, then determines if it belonged to a clique, and, if so, continued to check other nodes to see if they formed a maximal clique or any clique subset. However, the code was not yet complete and produced incorrect outputs for certain inputs and we had limitations of inputs due to usage of array which we need to specify the size and due to the implementation of ”getMaximalCliqueSize” and ”isClique” functions. Also boolean ”flag” variable approach was not working as intended because the edges iteration on the other side was occurring on different indexes non-serial way for certain inputs.

```
#include <iostream>
#include <math.h>
using namespace std;
int hammingDistance(std::string n1,
    std::string n2) {
   int counter = 0;
   for (int i=0; i<n1.size(); i++) {
      if (n1[i] != n2[i]) {
         counter++;
      }
   }
   return counter;
}
std::string int_to_binary(int n) {
   std::string token = "";
   while (n > 0) {
      token = std::to_string(n % 2) + token;
      n /= 2;
   }
   return token;
}
const int MAX = 100;
int verticeStore[MAX], nodeCount;
int graph[MAX][MAX];
int d[MAX];
bool isclique(int b){
   for (int i = 1; i < b; i++) {
      for (int j = i + 1; j < b; j++) {
         if
             (graph[verticeStore[i]][verticeStore[j]]
             == 0) {
            return false;
         }
      }
   }
   return true;
}
int getMaximalCliqueSize(int currentNode ,
    int cliqueCluster){
   int maxC = 0;
   for (int j = currentNode + 1; j <=
       nodeCount; j++) {
      verticeStore[cliqueCluster] = j;
      if (isclique(cliqueCluster + 1)) {
         maxC = std::max(maxC, cliqueCluster);
         maxC = std::max(maxC,
             getMaximalCliqueSize(j,
             cliqueCluster + 1));
      }
   }
   return maxC;
```

```
}
int main() {
   int lenghtofstr;
   std::cout << "Please enter the lenght of
       binary: ";
   std::cin >> lenghtofstr;
   int max = pow(2,(lenghtofstr));
   int min = pow(2,(lenghtofstr-1));
   int minDistance;
   std::cout << "Please enter the d: ";
   std::cin >> minDistance;
   nodeCount = pow(2,lenghtofstr);
   int x=0;
   int y=0;
   int arrayOfResults[100][100];
   bool flag = true;
   for(int a=0 ; a<max && flag ; a++){
      for(int b =0 ; b<max ; b++){
         int n1 = a, n2 = b;
 if(hammingDistance(
 int_to_binary(n1),int_to_binary(n2))
     >=minDistance){
 if (n1 == arrayOfResults[x-1][1]){
 flag=false;
 break;
 }
arrayOfResults[x][y]=n1;
y++;
arrayOfResults[x][y]=n2;
x++;
y=0;
std::cout << "n2 is " << n2 << "and n1 is "
    << n1 << std::endl;
         }
      }
   }
   for(int a = 0 ; a < x ; a++){
      std::cout << arrayOfResults[a][0] <<
          "," <<arrayOfResults[a][1] <<
          std::endl;
   }
   std::cout << "Size of array: " << x <<
       std::endl;
   int size = x;
   for (int i = 0; i < size; i++) {
graph[arrayOfResults[i][0]][arrayOfResults[i][1]]
    = 1;
graph[arrayOfResults[i][1]][arrayOfResults[i][0]]
    = 1;
d[arrayOfResults[i][0]]++;
d[arrayOfResults[i][1]]++;
   }
   std::cout << "Max cliques: " <<
       getMaximalCliqueSize(0, 1);
  return 0;
}
```

III. Second Approach

In our second code, we decided to switch from using arrays to vectors to store data, as vectors can expand to fit as much data as the system's memory allows, whereas arrays have a fixed size which causes problems on larger graphs. Also, for our hamming distance and integer to binary functions, we realized that it does not necessarily require using std::string data set for bit strings but there is a library dedicated for that called bitset which allows xor operations directly and it has a count function that fastens the process. Additionally, we corrected an error in our for loops that resulted in saving each edge in both directions, which we fixed by starting the second loop at the element after the one being considered in the first loop instead of using boolean variable ("flag") approach. Furthermore we created NULL , NULL pair at the end of the edge vector prevent the last node not being included in the "isClique" function check since std::find() looks for a value until end value is reached. However, our system gave wrong outputs if we got input 0 and still struggled with large calculations due to memory issues. We also had to use the find function twice from the vector library in our "isClique" function, which took a long time to complete.

```
#include <iostream>
#include <math.h>
#include <vector>
#include <bitset>
using namespace std;
std::vector<std::pair<int, int>> edges;
std::vector<int> clique;
int nodeCount;
int calculateHammingDistance( std::bitset<64>
    &n1, std::bitset<64> &n2){
   return (n1 ^ n2).count();
}
std::bitset<64> intToBinary(int n){
   return std::bitset<64>(n);
}
bool isClique(int x){
   for (int i = 1; i < x; i++){
      for (int j = i + 1; j < x; j++){
         if (std::find(edges.begin(),
             edges.end(),
             std::make_pair(clique[i],
             clique[j])) == edges.end() &&
            std::find(edges.begin(),
                edges.end(),
                std::make_pair(clique[j],
                clique[i])) == edges.end()){
            return false;
         }
      }
   }
   return true;
}
int getMaximalCliqueSize(int currentNode ,
    int cliqueCluster){
   int max_size = 0;
   for (int j = currentNode + 1; j <
       nodeCount; j++){
      clique[cliqueCluster] = j;
      if (isClique(cliqueCluster + 1)){
         max_size = std::max(max_size,
             cliqueCluster);
         max_size = std::max(max_size,
             getMaximalCliqueSize(j,
             cliqueCluster + 1));
      }
   }
   return max_size;
}
```

```
int main()
{
   int binary_string_length;
   std::cout << "Please enter the length of
       binary string: ";
   std::cin >> binary_string_length;
   int max = pow(2, binary_string_length) - 1;
   int minimum_distance;
   std::cout << "Please enter the d: ";
   std::cin >> minimum_distance;
   nodeCount = pow(2, binary_string_length);
   for (int a = 0; a <= max; a++){
      for (int b = a + 1; b <= max; b++){
         bitset<64> aToBinary =
             intToBinary(a);
         bitset<64> bToBinary =
             intToBinary(b);
         int calculatedHammingDistance =
             calculateHammingDistance(aToBinary,
             bToBinary);
         if (calculatedHammingDistance >=
             minimum_distance){
            edges.emplace_back(a, b);
         }
      }
   }
   for (auto &edge : edges){
      std::cout << edge.first << " " <<
          edge.second << std::endl;
   }
   clique.push_back(0);
   edges.emplace_back(NULL, NULL);
   int clique_size = getMaximalCliqueSize(0,
       1);
   cout << "The size of the maximal clique
       is: " << clique_size << endl;
   return 0;
}
```

IV. Third Approach

In our third code implementation, we encountered an issue with the calculation of 0's due to starting the loop variable "j" from "currentNode +1" which is "1". We attempted to address this issue through trial and error, and initially thought we had found a solution by changing the indices of the "getMaximalCliqueSize" function call to (0,2). However, we later discovered that this solution only worked on iOS computers, not on Windows computers(our best guess is that it has something to do with the xcode compiler). After further investigation, we identified the real solution to be starting the for loop in the "getMaximalCliqueSize" function from "currentNode" and changing the first recursion index from "j" to "j+1".Additionally, we found that the use of the "find" function in the "isClique" function was significantly slowing down the algorithm, leading to poor time complexity. To address this issue, we decided to use a hash map, as finding a value in a hashmap takes O(1) time. To use std::pair in a hash map, we needed to create our own constructor (template) called "pair_hash", which allowed us to store edges as pairs in an unordered_map and assign a Boolean value indicating the presence or absence of an edge between two nodes. We also used the "memo" hash table to keep track of the results of the "getMaximalCliqueSize" function, so that the function does not need to be called repeatedly when the same values are encountered.

```
#include <iostream>
#include <math.h>
#include <vector>
#include <bitset>
#include <unordered_map>
using namespace std;
std::vector<int> clique;
int nodeCount;
struct pair_hash{
   template <class T1, class T2>
   std::size_t operator()(const std::pair<T1,
       T2> &p) const{
      auto h1 = std::hash<T1>{}(p.first);
      auto h2 = std::hash<T2>{}(p.second);
      return h1 ^ h2;
   }
};
std::unordered_map<std::pair<int, int>, int,
    pair_hash> memo;
std::unordered_map<std::pair<int, int>, bool,
    pair_hash> edge_hash;
int calculateHammingDistance( std::bitset<64>
    &n1, std::bitset<64> &n2){
   return (n1 ^ n2).count();
}
std::bitset<64> intToBinary(int n){
   return std::bitset<64>(n);
}

bool isClique(int x){
   for (int i = 1; i < x; i++){
      for (int j = i + 1; j < x; j++){
         if (!edge_hash[{clique[i],
             clique[j]}] &&
             !edge_hash[{clique[j],
             clique[i]}]){
            return false;
         }
      }
   }
   return true;
}
int getMaximalCliqueSize(int currentNode ,
    int cliqueCluster){
   int max_size = 0;
   if (memo.count({currentNode ,
       cliqueCluster})){
      int maxSize = memo[{currentNode ,
          cliqueCluster}];
      return maxSize;
   }
   for (int j = currentNode ; j < nodeCount +
       1; j++){
      clique.resize(cliqueCluster + 1);
      clique[cliqueCluster] = j;
      if (isClique(cliqueCluster + 1)) {
         max_size = std::max(max_size,
```

```
            cliqueCluster);
        max_size = std::max(max_size,
            getMaximalCliqueSize(j + 1,
            cliqueCluster + 1));
      }
   }
   memo[{currentNode , cliqueCluster}] =
       max_size;
   return max_size;
}
int main(){
   int binary_string_length;
   std::cout << "Please enter the length of
       binary string:";
   std::cin >> binary_string_length;
   int max = pow(2, binary_string_length) - 1;
   int minimum_distance;
   std::cout << "Please enter the d:";
   std::cin >> minimum_distance;
   nodeCount = pow(2, binary_string_length);
   for (int a = 0; a <= max; a++){
      for (int b = a + 1; b <= max; b++) {
         bitset<64> aToBinary =
             intToBinary(a);
         bitset<64> bToBinary =
             intToBinary(b);
         int calculatedHammingDistance =
             calculateHammingDistance(aToBinary,
             bToBinary);
         if (calculatedHammingDistance >=
             minimum_distance){
            edge_hash[{a, b}] = true;
         }
      }
   }
   clique.push_back(0);
   int clique_size = getMaximalCliqueSize(0,
       1);
   cout << "The size of the maximal clique
       is: " << clique_size << endl;
   return 0;
}
```

V. Fourth Approach

With further deeper review, we realized that because of we are scanning every possible outcome of a binary length the zero always be included in the max clique and every node's clique has the same size which is max clique. Therefore in this particular case we do not necessarily need to go through every single clique of the nodes we just need to check one whole clique which starts with zero. This is not the most reliable way to find maximal clique. However for this case it reduces time complexity to O(n'2).It is important to note that if the nodes was given one by one in the input, this function is useless.

```
int simplerVersion(){
   int count=1;
   int size = 1;
   for (int i=0; i<nodeCount; i++) {
      int a = 0;
      for (int j=0; j<size; j++) {
         if (edge_hash[{i, memory[j]}] ||
             edge_hash[{memory[j], i}]) {
            a ++;
            if (a == size &&
                !std::count(memory.begin(),
                memory.end(), i)) {
               memory.push_back(i);
               size ++;
               count++;
            }
         }
      }
   }
   return count;
}
```

VI. Final Approach

In our final approach, we made some small clean-tidy changes like instead of giving initial values to our "getMaximalCliqueSize" in the main function, we are giving the initial values on the function itself. Additionally we put a while loop in main to check if the input is greater than zero if not we asked the user to insert another valid input. We noticed that the unordered_map memo was not helping as we expected in the "getMaximalCliqueSize" , so we considered putting it inside the for loop and getting (j,cliqueCluster) indexes instead of (currentNode ,cliqueCluster) which allowed us to avoid unnecessary recursions in for loop and it significantly increased the time efficiency of the code(our input trials worked approximately 10 times faster). Also we tried many inputs to our and we saw that it is around %95 accurate but has some calculation errors for the inputs like A(10,5), but since it is way faster, the function is a good approximation strategy.

```
#include <iostream>
#include <math.h>
#include <vector>
#include <bitset>
#include <unordered_map>
using namespace std;
std::vector<int> clique;
int nodeCount;
struct pair_hash {
   template <class T1, class T2>
   std::size_t operator () (const
       std::pair<T1, T2> &p) const {
      auto h1 = std::hash<T1>{}(p.first);
      auto h2 = std::hash<T2>{}(p.second);
      return h1 ^ h2;
   }
};
//std::vector<int> memory; // For
    simplerVersion uncomment this.
std::unordered_map<std::pair<int, int>, bool,
    pair_hash> edgeHash;
int calculateHammingDistance(std::bitset<64>&
    n1, std::bitset<64>& n2) {
   return (n1 ^ n2).count();
}
std::bitset<64> intToBinary(int n) {
   return std::bitset<64>(n);
}
bool isClique(int x) {
   for (int i = 1; i < x; i++) {
```

```
      for (int j = i+1; j < x; j++) {
         if (!edgeHash[{clique[i],
             clique[j]}] &&
             !edgeHash[{clique[j],
             clique[i]}]) {
            return false;
         }
      }
   }
   return true;
}
std::unordered_map<std::pair<int, int>, int,
    pair_hash> memo;
int getMaximalCliqueSize(int currentNode = 0,
    int cliqueCluster = 1) {
   int maxSize = 0;
   for (int j = currentNode ; j < nodeCount +
       1; j++) {
      if (memo.count({j, cliqueCluster})) {
         return memo[{j, cliqueCluster}];
      }
      clique.resize(cliqueCluster+1);
      clique[cliqueCluster] = j;
      if (isClique(cliqueCluster+1)) {
         maxSize = std::max(maxSize,
             cliqueCluster);
         memo[{j, cliqueCluster}] = maxSize;
         maxSize = std::max(maxSize,
             getMaximalCliqueSize(j + 1,
             cliqueCluster + 1));
      }
   }
   return maxSize;
}
/*
int simplerVersion() {
   int count=1;
   int size = 1;
   for (int i=0; i<nodeCount; i++) {
      int a = 0;
      for (int j=0; j<size; j++) {
         if (edgeHash[{i, memory[j]}] ||
             edgeHash[{memory[j], i}]) {
            a ++;
            if (a == size &&
                !std::count(memory.begin(),
                memory.end(), i)) {
               std::cout << "putting "<< i <<
                   " to the memo"<< std::endl;
               memory.push_back(i);
               size ++;
               count++;
            }
         }
      }
   }
   return count;
}
*/
int main() {
   int binaryStringLength;
   std::cout << "Please enter the length of
       binary string:";
   std::cin >> binaryStringLength;
   while (binaryStringLength < 1) {
      std::cout << "Length of binary string
          can't be lower than one. Please
          enter the length of binary string
          again: ";
      std::cin >> binaryStringLength;
   }
   int max = pow(2, binaryStringLength) - 1;
   int minimumDistance;
   std::cout << "Please enter the d (minimum
       distance):";
   std::cin >> minimumDistance;
   nodeCount = pow(2, binaryStringLength);
   for (int a = 0; a <= max; a++) {
      for (int b = a + 1; b <= max; b++) {
         bitset<64> aToBinary =
             intToBinary(a);
         bitset<64> bToBinary =
             intToBinary(b);
         int calculatedHammingDistance =
             calculateHammingDistance(aToBinary,
             bToBinary);
         if (calculatedHammingDistance >=
             minimumDistance) {
            edgeHash[{a, b}] = true;
         }
      }
   }
// memory.push_back(0); // For simplerVersion
    uncomment this.
   int maximalCliqueSize =
       getMaximalCliqueSize();
   cout << "The size of the maximal clique
       is: " << maximalCliqueSize << endl;
   return 0;
}
```

VII. NP-Hard Restrictions

Due to the NP-hard nature of the maximal clique problem, we encountered certain restrictions and challenges while trying to solve it efficiently. We implemented various optimization techniques and algorithms, such as unorderedmap (hashmap) and dynamic programming, to overcome these restrictions. However, we still faced limitations, including memory and time constraints. To address the memory limitation, we implemented an unordered map and only stored the edges that met the minimum distance requirement in one direction to conserve memory. To address the time complexity, for the reliable algorithm, we implemented unorderedmap which has O(1) search time in its find function. But for this case we realized that either all completed cliques are maximal cliques or 0 is always part of a maximal clique. This allowed us to run two for loops to only check after 0's cliques and find one maximal clique without using recursion to compare all of them, reducing the time complexity to O(2n^2). Despite these challenges, we were able to calculate the maximal clique for bit lengths up to high inputs depending to the memory space of hardware with using our approach.