Universidad de Cádiz

University of Cádiz

School of Engineering

Assignment 2: The classes P and NP

Francisco Chanivet Sánchez

16th November 2021

Contents

Tables

Figures

1 Problem that we tackle

The problem that we tackle in this assignment is the **exam scheduling** problem defined as:

Given a list of subjects and students enrolled in each of them, determine a schedule for conducting their exams in the minimum number of days, bearing in mind that a student cannot sit more than one exam on the same day

2 Initial questions

To introduct this assignment, we ask some questions about some elements of the problem that we need:

- **What does the chromatic number mean in this context?**
  The chromatic number in this context mean how many days to do the exams of all subjects.
- **What does the clique number mean in this context?**
  The clique number in this context mean how many subjects coincide the same day.
- **What is the relation between both numbers?**
  The relation between both numbers is we can compute the number of days that we need to do the exams based in the coincidences.
- **What is their relation to the maximum vertex degree?**
  The relation with the maximun vertex degree is the maximum number of subjects coincide with one subject.

3 Implementation of the algorithm to solve the exam scheduling

If we want to solve the problem of the exam scheduling, we use the chromatic number algorithm here. So at first to understand better this algorithm, we classify the elements that we use in this algorithm:

- **Graph G** with the exams that coincide.
- An **array of colours C** where we have the available subjects numbered from 0 to N. The size of this array is N, that is the number of subjects that we have.
- A **variable B** that indicates us the amount of days that we need to do these exams.

After seeing this, we need to generate a graph to simulate the coincidences of this exams. Then, we need programming a function that generates a graph.

3.1 Generator function

To simulate the exam scheduling, we want to to simulate a graph that the edges of the graph represent the coincidence of the exams. In addition, we want to generate a file text with the format **DIMACS**[1]. But, **how can we generate random data in this format?** With this algorithm we can do this:

```
        file<<"c FILE: exams.txt\n";
        file<<"c --- Random Exam Scheduling ---\n";

        srand(time(NULL));
        file<<"p edge "<<k<<" "<<k<<std::endl;
        int nC,nA;
        for(int i = 0; i < k; i++){
                nC = rand() % k;
                for(int j = 0; j < nC; j++){
                        nA = rand() % k;
                        while(nA == i)
                                nA = rand() % k;
                        file<<"e "<<i<<" "<<nA<<std::endl;
                }
        }
```

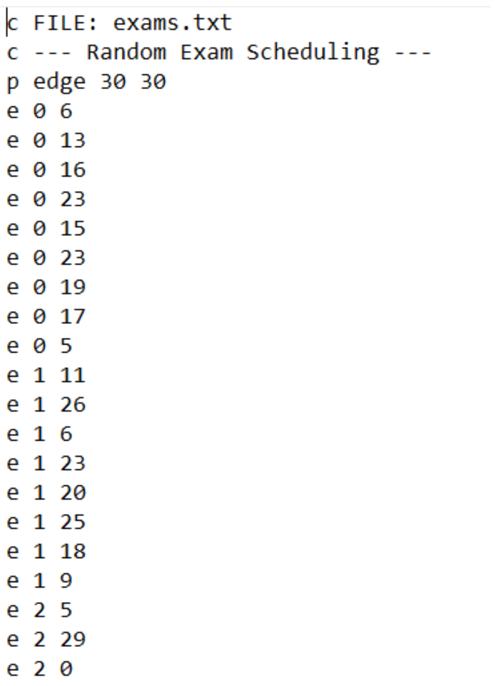
Figure 1: Generator function code

[1]DIMACS is a format to represent instances of a problem that use graphs, in this assignment we represent graph colouring instances

This algorithm do the next steps:

1. Create a file named **exams.txt** inside the directory **file**.
2. We define the number of edges of our graph. In that case, depends of the number of subjects we have.
3. After we create random edges for each subject. For generating that edges, we use two random numbers indicate us the subjects that we connect except itself[2].

After doing these steps we obtain the next file:



```
c FILE: exams.txt
c --- Random Exam Scheduling ---
p edge 30 30
e 0 6
e 0 13
e 0 16
e 0 23
e 0 15
e 0 23
e 0 19
e 0 17
e 0 5
e 1 11
e 1 26
e 1 6
e 1 23
e 1 20
e 1 25
e 1 18
e 1 9
e 2 5
e 2 29
e 2 0
```

Figure 2: A fragment of output file in DIMACS format

After generating the graph, we want to translate this file text into a graph, then we need a **translator**.

[2]There could be some cases that any subjects coincide with other.

3.2 Translator function

We need this function if we want to obtain a graph with the random data that we generated before.

```
void translator(std::vector< std::vector<int> >& v){

        using namespace std;

        string line;
        ifstream file("file/exams.txt");
        regex expression("((?:[0-9])+?)+");
        int values[2];
        while(getline(file,line)){
                if(line[0] != 'c'){
                        transInt(line,values,expression);
                        if(line[0] == 'p'){
                                vector< vector<int> > aux(values[0], vector<int>(values[1],0));
                                v = aux;
                        }
                        else{
                                v[values[0]][values[1]] = 1;
                        }
                }
        }

        file.close();
}
```

Figure 3: Translator function code

What do we do in this function for translating the information of file text into a graph?
We pass to the function an adjacency matrix. Then, we open the file text that we generated before, it is in **file/exams.txt**.
As we can see, we use here **regular expressions** to obtain the number of each line and build the adjacency matrix of our graph. We use the regular expression **((?:[0-9]+?)+**.
With this regular expression we can extract the numbers of each line, for example, if we have the line *p edge 20 20*, the algorithm knows that adjacency matrix have a size of **25x25**. Also, if we have the line *e 1 12*, we put 1 in that position.
However, how can we extract this numbers? Because there is not any sign of extraction in the fragment. For this purpose, we programm the next function where we pass an array with size 2, the regular expression and a line of the file.

```
void transInt(std::string str, int *v,std::regex exp){
        std::sregex_iterator currentMatch(str.begin(), str.end(), exp);
        std::sregex_iterator lastMatch;
        int cont = 0;
        while(currentMatch != lastMatch && cont < 2){
                std::smatch match= *currentMatch;
                std::stringstream s1(match.str());
                s1>>*(v + cont);
                currentMatch++;
                cont++;
        }
}
```

Figure 4: Translator to integer numbers code

At first, we point to the first match, that is the first number that we have, then we convert to integer number and we put it in the corresponding position[3]. The functions that we use for translating the file text are declared inside the **regex**[4] library. Therefore, we can build the adjacency matrix correctly. If we have at first the letter **p**, we define the matrix adjacency with the size that we have in that line and we fill it with 0, indicate that there is not any edge in that graph inityally[5]. After defining the matrix, we continue reading the file text, adding edges to the graph while we have **e** as first letter of the line.

[3]It depends of the variable *cont*

[4]For more information about library regex, see: https://en.cppreference.com/w/cpp/regex

[5]The graph of this problem is non weighted, it means that the graph has 0 or 1 as values.

3.3 Solver function

In order to solve the initial problem, we use this function to obtain the days that we need to do the exams.

As we can see before, we pass to the solver function **backGCP** the adjacency matrix of the graph, two integer numbers: **0** and **number of subjects + 1**, and a variable **vertex** where we explain it use after.
So, at first we check if variable vertex is equal to the size of the adjacency matrix because if we reach this point means that we have gone through the matrix completely. But if it is not the case, we go through **array of subjects** to check if there is any value of the list in each vertex exceed the limit[6].

```
        if(vertex == G.size()){
                return k;
        }

        for(int i = vertex; i < G[vertex].size(); i++){
                        if(no_colour_underbound(C[i],B)){
                                return B;
                        }
        }
```

Figure 5: First septs in solver algorithm

```
bool no_colour_underbound(const list<int>& cv, int b){
        bool valid = true;
        list<int>::const_iterator it = cv.cbegin();

        while(it != cv.cend() && valid){
                if(*it < b){
                        valid = false;
                }
                ++it;
        }
        return valid;
}
```

Figure 6: Function where we check if the elements of the list exceed the value of B

After checking the necessary, we can obtain the number that we are searching since the beginning. Then, at fisrt, we select the first vertex available in the graph, this is given by the variable **vertex** because we can reduce the operation to $\theta(1)$, therefore we have an elemental operation.
After this, we go through the list of available subjects and while the value of the list that we have in that moment is **less than B**, we remove the value if the value in the adjacent vertex. So, in the next step, we call the function pass:

[6]The limit is given by variable B.

- Adjacency matrix.
- The maximum number bewteen k or the number that we have from the list.
- Variable B.
- The array of subjects.
- We add to the vertex the value 1, indicate that the vertex we have now has removed from the graph.

```
        list<int>::iterator it = C[vertex].begin();

        while(it != C[vertex].end()){
                if(*it < B){
                        for(int j = vertex; j < G[vertex].size(); j++){
                                if(G[vertex][j] == 1 && vertex != j){
                                        C[j].remove(*it);
                                }
                        }

                        B = backGCP(G,max(k,*it),B,C,vertex+1);

                        for(int j = vertex; j < G[vertex].size(); j++){
                                if(G[vertex][j] == 1 && vertex != j){
                                        C[j].push_front(*it);
                                        C[j].sort();
                                }
                        }
                }

                ++it;

```

Figure 7: We are obtaining the number of days here.

After the call, we insert the element in the adjacent vertex and we sort[7] the list.

[7]This operation can do the algorithm more slowly.

3.4 How can we run the programm?

At first, we need to extract the zip of the assignment **ChanivetSanchezFrancisco_P2** in a directory or desktop. We need to have the next structure:

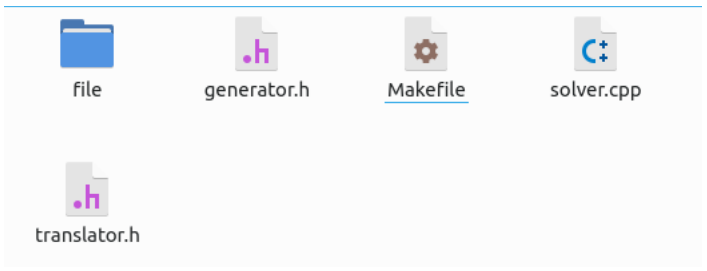
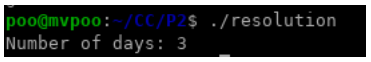


Figure 8: Structure of the directory

Also, this is important, **we need to have installed c++11** in our operative system because we use **regex** and this library is included since c++11. Then, we open the terminal of our operative system and we go to the directory where we put the files and we introduce in terminal the order **make** to compile the program. If we want to remove the files with extension .o, we have only introduced **make clean**.
After compiling the program, we can execute it introduce the order **./resolution**. Then, we obtain the next output in screen:



```
poo@mvpoo:~/CC/P2$ ./resolution
Number of days: 3
```

Figure 9: Output that we obtain for 30 subjects

If we want to modify the number of subjects, we only edit the number of variable **n_asig** that is inside **solver.cpp** in the main program.
Also and this is important, for big values of **n_asig**, the program can execute slowly.

4 Research questions

4.1 Analysis

Now, we are going to analyze the time of execution of this algorithm. To analyze, we are going to use the **ctime** library, where we use two variables: **t0** and **t1**.
We do this:

```
        unsigned t0,t1;
        t0 = clock();
        int B = backGCP(G,0,G.size() + 1,C,0);
        t1 = clock();
```

Figure 10: We are obtaining the time execution

As we can see, we call to **clock()** before and after calling the algorithm because we obtain the time when we called the algorithm and the time when we stop.
Then, we obtain the time in seconds with this formula:

$$(t1 - t0)/CLOCKS\_PER\_SEC$$

Being $CLOCKS\_PER\_SEC$ a constant value from **ctime** library.
We execute the algorithm with different values of n and we obtain the next table and graphic:

n	Execution time (seconds)
5	4e-6
10	1.4e-05
15	0.00035
20	0.014761
25	0.031974
30	0.024921

Table 1: Time values obtained for every number of subjects

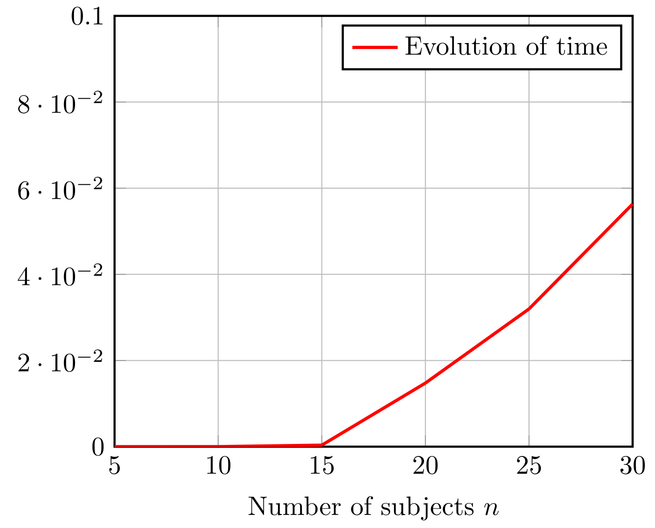


Figure 11: Plot of the execution time versus the number of subjects n

With the graphic, we can see that when we have 15 subjects, the execution time increase in a lineal form.

4.2 What if we reduce the vertex selection to decreasing degree order?

As we said before, we reduce the vertex selection to an elemental operation, that has a constant cost ($\theta(1)$). Then, we have the same plot and we do not have any differences here.