University of Cádiz

School of Engineering

Assignment 3: Polynomial Reductions

Francisco Chanivet Sánchez

14th December 2021

Contents

Tables

Figures

1 Introduction

In this assignment, we are going to work with two concepts: **SAT problem** and **CNF formula**. We are going to see the definition of each one.
We can define the **SAT problem** as "*the problem of determining if there exists an interpretation that satisfies a given Boolean formula*". This means that the main objective of the problem is check that problem is truth or in this context, **satisfiable**.
On the other hand, we can define **CNF formula** as "*a conjunction of one or more clauses, where a clause is a disjunction of literals; otherwise put, it is a product of sums or an AND of ORs*". We can see this definition better with the problem of this assignment.
The definition of the problem of this assignment is the next:

Let G = (V,E) be an undirected graph. S V is a clique of G if all the vertices in S are adjacent to each other. S induces a complete subgraph of G A k-clique is a clique of size k.

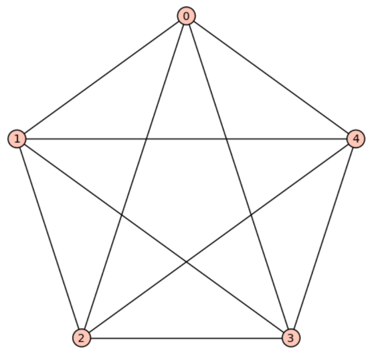


Figure 1: We have a complete graph.

And we have the next properties that we must check if we want to obtain a k-clique:

1. Each vertex in a clique must be a vertex in the graph.
2. A vertex cannot appear twice in the same clique.
3. Two nonadjacent vertices cannot be in the same clique.

Then, we have the next boolean variable:
x_{ir} = "vertex i in G is vertex r in a k-clique of G"
Later, if we translate the clauses of this problem, we have the next formulas:

1. $x_{1r} \vee x_{2r} \vee ... \vee x_{nr}$ for all $1 \leq r \leq k$.
2. $\neg x_{ir} \vee \neg x_{is}$ for all $1 \leq i \leq n$ and $1 \leq r < s \leq k$.
3. $\neg x_{ir} \vee \neg x_{js}$ for all $1 \leq i < j \leq n$ and $1 \leq r$, $s \leq k$ such that $(i, j) \notin E$ and $r \neq s$.

2 How have we implemented the reducer?

At first, we pass to the function the name of the directory where the graph is and the size of the clique called k.
Then, we use the **translator function** that **we implemented in the assignment 2** to translate the graph that we have in DIMACS format to a graph in our program. The changes that we introduce to the function is add new variable called **file_name** where we pass the name of the DIMACS file that we want to translate and we **substract one** to the values of the file because we **add one** to the function generator of DIMACS because we use **Cliquer** to find the cliques of the graph of the file. We insert this change because we need to solve the search problem and the optimisation problem.

```
34  void translator(std::string file_name,std::vector< std::vector<int> >& v){
```

Figure 2: New header for translator function

```
42          while(getline(file,line)){
43                  if(line[0] != 'c'){
44                          transInt(line,values,expression);
45                          if(line[0] == 'p'){
46                                  vector< vector<int> > aux(values[0], vector<int>(values[1],0));
47                                  v = aux;
48                          }
49                          else{
50                                  v[values[0]-1][values[1]-1] = 1;
51                          }
52                  }
53          }
```

Figure 3: As we can see, we change the line 50.

After translating the file, we can reduce to **SAT problem** the graph that we have.
Firtsly, we enumerate each boolean variable that we have because we need it to build the clauses.
Then, we insert in a matrix[1] the enumeration.

```
63          vector< vector<int> > num(g.size(), vector<int>(k,0));
64          int nVar = enumerate(num,g.size(),k);
65          size_t nCla = 0;
```

Figure 4: We call enumerate function here.

[1]we use beacause we use two index in this problem.

```
int enumerate(vector< vector<int> >& x, size_t numVert, size_t nClique){
        // xir = "vertex i in G is vertex r in a k-clique of G"
        int numV = 1;

        for(size_t i = 0; i < numVert; i++){
                for(size_t r = 0; r < nClique; r++){
                        x[i][r] = numV;
                        ++numV;
                }
        }

        return numV; //We obtain the number of variables
}
```

Figure 5: Enumerate function

We have **two variables** that we are going to use in the CNF file, which are the number of boolean variables and the number of clauses that we have.
After enumerating the variables, we open a **temporal file** called **temp.cnf** where first we introduce some information about the file and introduce the word **p**, that we use later to insert the number of boolean variables and clauses.
After adding this information, we can start with the **first clause**. Remembering the first clause, $x_{1r} \vee x_{2r} \vee ... \vee x_{nr}$ for all $1 \leq r \leq k$, it means that each vertex in the graph is vertex in clique graph. Therefore, we can program this clause by this way:

```
        // 1st clause
        for(size_t r = 0; r < k; r++){
                for(size_t i = 0; i < g.size(); i++){
                        file<<num[i][r]<<" ";
                }
                file<<"0\n";
                ++nCla;
        }
```

Figure 6: First clause

As we can see, we go through the **nodes of r** and the **nodes of the graph**. With this, we put the the variable in the file and we pass to the next value. Clarify that DIMACS for **SAT problem** we introduce each variable and we pass to the next variable indicate that we have an **OR** expression. If we want to difference between **OR** and **AND** expressions[2], we put in the end of the line a **0** and we continue in the next line with the next clause.

[2]Remember that we use CNF formula to express the properties of the clique.

After working with the first clause in DIMACS, we continue with the **second clause** which said $\neg x_{ir} \vee \neg x_{is}$ for all $1 \leq i \leq n$ and $1 \leq r < s \leq k$. This means that a vertex cannot appear twice in the same clique. Then, we can program this clause by this way:

```
        // 2nd clause
        for(size_t i = 0; i < g.size(); i++){
                for(size_t s = 1; s < k; s++){
                        for(size_t r = 0; r < s; r++){
                               file<<num[i][r]*(-1)<<" "<<num[i][s]*(-1)<<" 0"<<endl;
                               ++nCla;
                        }
                }
        }
```

Figure 7: Second clause

In DIMACS format, the **negative variable** is represented by the negative element that we have, for example if we have the element 1, it negative form is -1. Then $\neg 1 \Leftrightarrow -1$. Also, we start with 1 in **s** because **s is greater than r**.

After writting the second clause in the file, finally we are going to work with the **third clause**, that is $\neg x_{ir} \vee \neg x_{js}$ for all $1 \leq i < j \leq n$ and $1 \leq r,\ s \leq k$ such that $(i, j) \notin E$ and $r \neq s$. This means that two non adjacents vertex cannot be in the same clique, then in the case that we have a non connected graph, this property would not be satisfiable.

```
// 3rd clause
size_t r = 0, s = 0;
size_t i, j;
while(r < k){
        s = 0;
        while(s < k){
                if(r != s){
                        j = 1;
                        while(j < g.size()){
                                i = 0;
                                while(i < j){
                                   if(g[i][j] == 0){
                                        file<<num[i][r]*(-1)<<" "<<num[j][s]*(-1)<<" 0"<<endl;
                                        ++nCla;
                                   }
                                   i++;
                                }
                                j++;
                        }

                }
                s++;
        }
        r++;
}
```

Figure 8: Third clause

After writting in the file all the clauses, we are going to **edit** the number of variables and clauses that we have and we create a new file called **cnf_file.cnf**. This implies that we erase the temporal file that we created in the reduction function and we save all the clauses in the new file.

```cpp
void editClause(size_t nVar, size_t numC){
        ifstream temporal("file/temp.cnf");
        ofstream cnf("file/cnf_file.cnf");
        string line;
        while(getline(temporal,line)){
                if(line[0] == 'p'){
                        cnf<<"p cnf "<<nVar-1<<" "<<numC<<endl;
                }
                else{
                        cnf<<line<<endl;
                }
        }
        cnf.close();
        temporal.close();
        remove("file/temp.cnf");
}
```

Figure 9: Function where we edit the number of clauses and variables

3 How have we implemented the search and optimisation version of the problem?

At firt, we generate a DIMACS file with the graph that we introduce in the function. In the case that you want to prove with other graphs, you can modify the adjacency matrix in the source code of the main program.
However, **can we generate a DIMACS file for the given graph?** Yes, we use the function **generateDimacs** that we implemented in the assignment 2, but in this case, we hace two functions for each case: when we have a graph and a subgraph. Then, the change that we have in this function is the new header that is: **generateDimacsG** and as we mentioned before, we add 1 to the vertex because we need it for the **Cliquer** program. Then, we have the next:

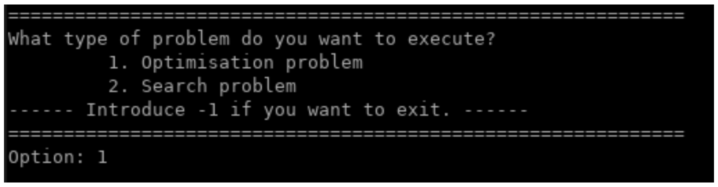
```
void generateDimacsG(string file_name, const vector< vector<int> >& v){
```

Figure 10: DIMACS generator function for a graph

```
        for(size_t i = 0; i < v.size(); i++){
                for(size_t j = 0; j < v.size(); j++){
                        if(v[i][j] == 1){
                                graph<<"e "<<i+1<<" "<<j+1<<endl;
                        }
                }
        }
```

Figure 11: Line that we have changed in the function

After explaning about how we generate a DIMACS file of the graph, we have a menu in the screen where we can choose executing the **optimisation version** of the problem or the search version.



```
============================================================
What type of problem do you want to execute?
        1. Optimisation problem
        2. Search problem
------ Introduce -1 if you want to exit. ------
============================================================
Option: 1
```

Figure 12: Menu.

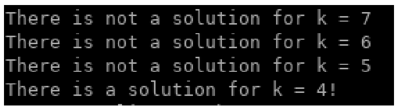
Then, imagine that we choose the optimisation version, we try executing the reductor with different values of k and we take the first value that has a clique in the graph.
Then, we have the next code for the optimisation problem:

```
	vector< vector<int> > mat;
	translator("file/graph.txt",mat);
	size_t nClique = mat.size() + 1;
	int isClique = 0;
	while(nClique > 0 && isClique != 2560){
		--nClique;
		reductorCNF("file/graph.txt",nClique);
		isClique = system("picosat file/cnf_file.cnf > /dev/null");
		if(isClique != 2560){
			cout<<"There is not a solution for k = "<<nClique<<endl;
		}
	}

	if(isClique == 2560){
		cout<<"There is a solution for k = "<<nClique<<"!"<<endl;
	}
```

Figure 13: Optimisation problem

As we can see in the fragment of code, we stop the loop when the value of **isClique** is equal to **2560**, indicate us that we have found a satisfiable solution. For this purpose, we execute with the function **system** the picosat program and as we do not want the output of the program, we take the value that it has.
We decrease the value of nClique from the number of vertex that the graph has to 1, because also, we want to find the maximum-size clique for the graph that we have in the problem.



```
There is not a solution for k = 7
There is not a solution for k = 6
There is not a solution for k = 5
There is a solution for k = 4!
```

Figure 14: Result of the optimisation problem.

In the other hand, if we choose the search problem, we are going to find the combination of nodes that form a k-clique with the value of k that the user has inserted.
Then, we have **lists** in the search problem, a list with the nodes of the graph and the possible combinations of nodes. Therefore, we use the function **next_permutation** that it is implemented in the **algorithm** library. If we put this as condition of the loop, we indicate that we want all the possibles combinations and if this nodes are clique of the graph, we stop the loop.

```
        list<unsigned int> set;
        list<unsigned int> vertex;
        for(unsigned int i = 0; i < mat.size(); i++){
                vertex.push_back(i);
        }
```

Figure 15: Assignment of lists.

```
        while(next_permutation(vertex.begin(),vertex.end()) && isClique != 2560){
                set = vertex;
                set.resize(nClique);
                generateDimacsSG(file,set,mat);

                //We execute the reductor here
                reductorCNF(file,nClique);

                isClique = system("picosat file/cnf_file.cnf > /dev/null");

        }
```

Figure 16: Search problem

As we can see, we assign to the list **set** the list **vertex** and we resize the list **set** with the number of clique that we want. Then, we generate a DIMACS file for the subgraph that we have generated at that time. As we mentioned before, we have two functions for generating DIMACS files, in this case we are generating a DIMACS file of a subgraph, thus we have the next function:

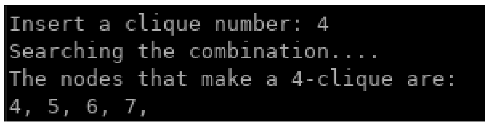
```
void generateDimacsSG(string file_name,list<unsigned int>& vertex, const vector< vector<int> >& v){
        ofstream graph(file_name);
        graph<<"c Subgraph of this problem"<<endl;
        graph<<"c"<<endl;
        graph<<"p edge "<<v.size()<<" "<<v.size()<<endl;
        vertex.sort();
        list<unsigned int>::iterator it = vertex.begin();
        while(it != vertex.end()){
                list<unsigned int>::iterator it2 = vertex.begin();
                while(it2 != vertex.end()){
                        if(v[*it][*it2] == 1){
                                graph<<"e "<<(*it) + 1<<" "<<(*it2) + 1<<endl;
                        }
                        ++it2;
                }
                ++it;
        }

        graph.close();

}
```

Figure 17: DIMACS generator function for a subgraph

As we can see in the DIMACS function, we use the same size for edges, it means, we have the same number of vertex in the subgraph beacuse we only connect the nodes that have been selected.In this function we use iterators because we are going through the list.
After explaining this and having the results, we show at screen the nodes that form a k-clique.



```
Insert a clique number: 4
Searching the combination....
The nodes that make a 4-clique are:
4, 5, 6, 7,
```

Figure 18: Result of search problem at screen

4 How can we execute each version of the problem?

If we want to execute each version of the problem, we need to have this structure in a directory:

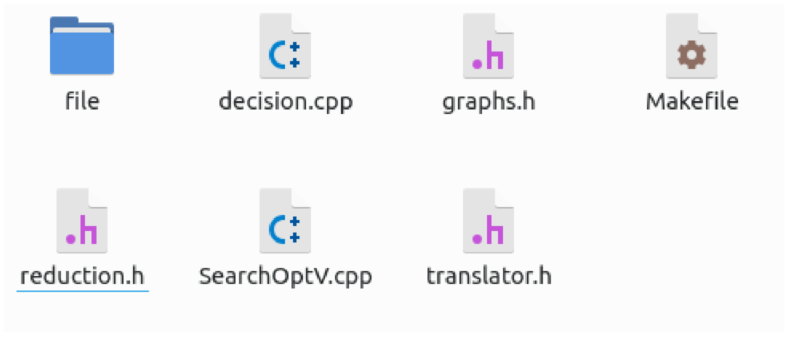


Figure 19: Structure of the directory.

Once that we have the same structure, we go to the terminal and we introduce the command **make** and we compile the program of the decision version and the search-optimisation version. If we want to execute the decision version of the program we introduce in the terminal **./decision** and if we want to execute the search or optimisation version we introduce **./resolution**. As we mentioned before, if you want to modify the adjacency matrix of the graph you can edit it in the source code (the matrix has the name **mat**).
In the case that we want to compile one version, you must introduce the next command in their respective cases:

- **make decision** if we want to execute the **decision version** of the problem.
- **make resolution** if we wanto to execute the **optimisation version** or **search version** of the problem.

5 Results that we obtained

Given this graph,

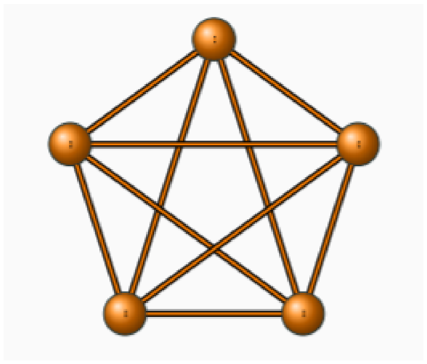
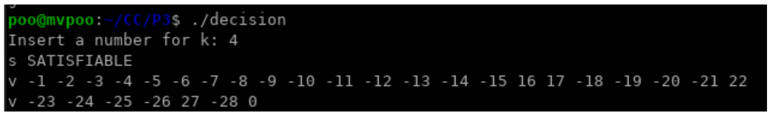


Figure 20: Graph of this exercise

we obtain the next results.

5.1 Results with picosat for 4-clique

For this problem, we use the **decision problem**, then we only introduce in our terminal **make decision** and we can execute the program introducing **./decision**.
Thus, when we execute the program, we only introduce the number 4 and we obtain the result by **picosat** program.

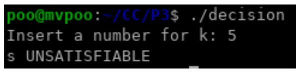


```
poo@mvpoo:~/CC/P3$ ./decision
Insert a number for k: 4
s SATISFIABLE
v -1 -2 -3 -4 -5 -6 -7 -8 -9 -10 -11 -12 -13 -14 -15 16 17 -18 -19 -20 -21 22
v -23 -24 -25 -26 27 -28 0
```

Figure 21: Results with picosat for 4-clique.

5.2 Results with picosat for 5-clique

In this case, we follow the steps that we mentioned before but in this case, we introduce 5.

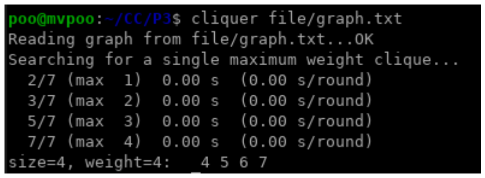


```
poo@mvpoo:~/CC/P3$ ./decision
Insert a number for k: 5
s UNSATISFIABLE
```

Figure 22: Results with picosat for 5-clique.

5.3 Results with cliquer

If we want to confirm our results with **cliquer**, we only introduce in the terminal: **cliquer file/graph.txt** and we obtain the next result:

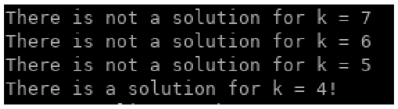


```
poo@mvpoo:~/CC/P3$ cliquer file/graph.txt
Reading graph from file/graph.txt...OK
Searching for a single maximum weight clique...
  2/7 (max  1)  0.00 s  (0.00 s/round)
  3/7 (max  2)  0.00 s  (0.00 s/round)
  5/7 (max  3)  0.00 s  (0.00 s/round)
  7/7 (max  4)  0.00 s  (0.00 s/round)
size=4, weight=4:   4 5 6 7
```

Figure 23: Result with cliquer.

5.4 What is the maximum-size clique for this graph?

If we want to obtain the answer of this question, we are going to use the **optimisation version** of the problem. Then, we obtain the next result:



```
There is not a solution for k = 7
There is not a solution for k = 6
There is not a solution for k = 5
There is a solution for k = 4!
```

Figure 24: Result of the optimisation problem for this graph.

As we can see in the picture, the maximum-size clique for this graph is **4**.

6 Questions of the research task

6.1 Which kind of reduction is obtained?

The type of reduction that we obtained is **Karp-reduction**, because if we can see, we give to the oracle machine[3] the input and this makes a query **f(x)** where f(x) is the **CNF file** that we made it. Then, the oracle, that is the **picosat** program, give us if the answer is yes or no, in other words, if these graph has a k-clique or not.

6.2 How many clauses are generated in terms of n and k?

If we execute the program many times with the graph that we have been working in this assignment, we can see in the file the **number of clauses**. With this results, we put it in the next table:

n	**k**	Number of Clauses
7	2	21
7	3	60
7	4	118
7	5	335

Table 1: Number of clauses for each value of k.

As we know, the **third clause** depends on the number of non adjacent vertex that we have in the graph. Then we have the next formula: $k + n * \sum_{s=1}^{k}\sum_{i=1}^{s}(s-i) + (nAV) * (k * (k-1))$, where **k** is the **size of clique** and **nAV** the **number of non adjacent vertex** of the graph.

[3]In this case is the **reduction function** because this generates the CNF file

7 Conclusion

After realising this assignment, we have achieved the next objectives:

1. Learn about the **DIMACS format** for **SAT CNF problems**.
2. Reduce from **decision problem** to **SAT problem**.
3. Simulate an **oracle**.

The **first objective** is the basis for the resolution of the problems that we have in this assignment, because with this we can check if the graph has a k-clique or not thanks to the file that we have made.

The **second objective** is the first problem that we have faced because with the resolution of this problem, you can use it for the resolution for the **search** and **optimisation version** of the problem.

And the **third objective**, we use the **two previous objective** and we have combined it.

As we can see in this assignment, we can obtain if a **graph has a k-clique or not**, the **maximum-size of clique** and the **combinations of nodes that has a k-clique in the graph**.

8 References

[1] *Boolean satisfiability problem.*
URL: https://en.wikipedia.org/wiki/Boolean_satisfiability_problem.

[2] *Conjuctive normal form.*
URL: https://en.wikipedia.org/wiki/Conjunctive_normal_form.

[3] Oded Goldreich. *P, NP and NP-Completeness.* Cambridge University Press, 2010.

[4] Universidad Autónoma de Madrid. *Grafos.*
URL: https://matematicas.uam.es/~pablo.angulo/doc/laboratorio/b4s2.html.