COMPUTATIONAL COMPLEXITY

GRADO EN INGENIERÍA INFORMÁTICA

PRACTICE 3

AUTHOR: TEODORO MARTÍNEZ MÁRQUEZ

Cádiz, 10 de Mayo de 2019

1. INTRODUCTION

In this practice I'm going to study the possible solutions of a graph coloring instance: the Petersen graph. I'm going to test which is the smalles possible number of colors needed to solve it through graph-coloring and through a SAT reduction (using the PICOSAT application to verify its correctness), and I'm going to showcase its relationship with the clique number.

2. METHODOLOGY

- **Representation of graphs**
  For this project I have, again, used a class taken from EDNL's sources to represent a graph, as seen in the file grafoLA.h. However, this time the graph class is significantly modified. It uses a vector of lists (both classes from the c++ std library) to store each vertex' adyacency list. It also stores a vector of ints to simulate the coloring of the graph, with the number 0 being no color. It has a single boolean variable, which will be mentioned later. It provides access to read, add and modify all of these elements.

- **Coloring a graph**
  In order to solve the problem via graph coloring, I have created a function that, given a graph and a maximum amount of colors, will try to color all of the nodes, and set the graph's boolean variable to true indicating that it is solvable with the given amount of colors. This can all be seen in the graphColor.h file.

  This function follows the 'backtracking' method for coloring graphs:

  -It does recursive calls, where each call is responsible of coloring one node.

  -Each call will check which colors are satisfiable for their respective node. When it finds a suitable color, it colors the node and does a recursive call.

  -When a call is able to color the last node, it sets the graph's boolean to true, signifying that a solution has been found, and then the recursive calls finish.

  -When a parent call doesn't receive a solution from its son call, it will try to color its node with a different color and do another recursive call.

  -If a call can't find any suitable colors for its node (or it has done recursive calls with all posible colors without success), it will leave the node blank and return to a previous call.

- **Reduction to SAT**
  To reduce the problem to a SAT problem (specifically in a DIMACS format), I have created another function in the file graph2SAT.h. Like previously, this function recieves a graph and an amount of colors, and will write through the standard output all the information that a .cnf file requires.

  First of all, this function outputs a comment with the graph's amount of vertices and the amount of colors available. Then, it needs to output the initial line of the format which contains the number of variables and the number of clauses for the SAT problem, for which we need to do some calculations. I will regard the number of colors as 'k' and the number of vertices as 'n' for the rest of the explanation.

The number of variables will be k * n, since each variable denotes one node having one color.

The number of clauses is a bit more complicated. Before we get that number, let's first explain how we are going to build the clauses:

- There will be one clause per node, forcing it to take at least one color (ex: node_1_blue or node_1_red or node_1_green, for k = 3)

- Then, for each node there will be a number of clauses to prevent it from having more than one color. This means we need all the 2-permutations from the k colors (ex: not node_1_blue or not node_1_green ; not node_1_blue or not node_1_red ; not node_1_green or not node_1_red).

The amount of 2-permutations for k colors is an arithmetic progression going from 1 to k-1, with a difference of 1. It can be calculated by the formula:

$$\frac{k * (k-1)}{2}$$

So, for now we have one plus the formula above clauses, for every node.

- Lastly, we have to set the constraints that will prevent two adjacent nodes from having the same color. To do that, we need to run through every node's adjacency list and add k clauses per every adjacent node (ex: not node_1_blue or not node_2_blue ; not node_1_red or not node_2_red, for k = 2).

Since adjacency lists can have any number of vertices up to n, there is no formula to calculate the number of clauses needed. So we have to add up every vertex's adjacency list's size times k.

However, the graphs we are working with are undirected graphs, which means that every adjacency is listed twice, so we have to divide the previous amount of clauses by two.

In order to avoid constructing duplicated clauses, I simply add a restriction when reading a vertex' adjacency list: the clause is only written if the "origin" vertex (the one whose list we are reading) is lesser than the "destiny" vertex (the one inside the list). This way, we would write a clause if we find the vertex 7 in 2's list, but not if we find the vertex 2 in 7's list, for example.

With that said, the total amount of clauses would follow the formula:

$$n + n * \frac{k * (k-1)}{2} + edges * k$$

Where, of course, we don't have the number of edges, and instead have to check the adjacency lists.

- **Running everything**

The functionality of every function used has been explained. To test the, I use the testReduction.cpp and testGraphColor.cpp files. Both of these files manually create the petersen graph and set the number of colors to 3, and then call their respective function. The graph and the amount of colors can be altered freely.

To get the executables, there's a small makefile which defines the instructions *make Reduction* and *make GraphColor*, which produce the corresponding executables.

The GraphColor executable will simply display the result of the graphColor function, whereas the Reduction executable will put through the standard output the converted

.cnf file, as mentioned earlier. In order to obtain the file, one must run the executable as such: *./testReduction >> file.cnf*

Make sure that *file.cnf* doesn't exist or it is empty, otherwise the program's output will be concatenated and produce a useless result. In any case, the file *dimacs.cnf* provided is the result of the Petersen graph with 3 colors.

3. RESULTS

After testing the programs, one can easily see that the Petersen graph is unsolvable with two colors, as both the graphColor function and the PICOSAT application fail to find a solution. The graph is however colorable with three colors, with 120 different possible solutions, and 12960 solutions with 4 colors. With this, we can say that the chromatic number of the Petersen graph is three.

Two of the possible 3-color solutions would be the following (assuming that 1, 2 and 3 are different colors, and the vertices are called in order from 1 to 10):

1 2 1 2 3 2 1 3 3 2

2 3 2 3 1 3 2 1 1 3

4. CONCLUSIONS

First of all, we have successfully made a Karp-reduction from graph coloring to SAT problems, as the reduction has no instructions that take longer than polynomial time, and the SAT-solving algorithm only has to be called once to obtain an answer.

We have also proved that the Petersen Graph's chromatic number is three, as it is not solvable with two colors.

If the graph is three-colorable, that means that if we pick any four vertices of the graph, at least one color will be repeated. If that is the case, then it is not possible for said four vertices to form a clique, as they would not satisfy the graph coloring problem. As such, it is impossible for the Petersen graph to have a clique of size four.

In fact, the maximum possible clique that any graph can have is its chromatic number, if we apply that same logic to any other case. This does not mean the clique will be of that size, only that it can't be bigger. In Petersen's case, its maximum clique is of size two, as it doesn't form any triangles.