Theory of Computation - Assignment 6

May 27, 2022

1 INTRODUCTION

The clique problem is the computational problem of finding cliques in a graph. Given an undirected graph with N nodes and E edges and a value K, the task is to print all set of nodes which form a K size clique.

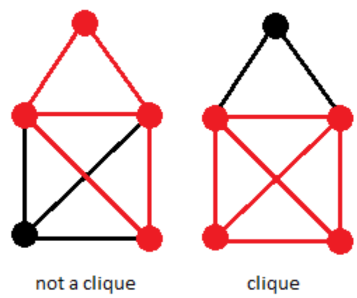


Figure 1: Example of a clique.

A clique C of a graph G is usually defined as a subset of the vertex set of G (V(G)) such that every pair of distinct vertices in C is adjacent in G (That is, two vertices u,v $\in$ C and u$\neq$v implies that u,v $\in$ E(G)). In other words, a subset of the vertex set of G is a clique if and only if its induced subgraph is a complete graph, that is if all distinct vertices are joined by and edge.
The input of the k-clique problem is an undirected graph and a number k. An undirected graph is formed by a finite set of vertices and a set of unordered pairs of vertices, which are called edges. The output is a list of k vertices which form a clique, if one exists, or a `false` otherwise.

2 REDUCTION: 3-SAT TO K-CLIQUE

2.1 3-SAT

SAT, or the Boolean satisfiability problem (sometimes called propositional satisfiability problem and abbreviated SATISFIABILITY, SAT or B-SAT), is the problem of determining if there exists an interpretation that satisfies a given formula in Boolean algebra (with unknown number of variables) whether it is satisfiable, that is, whether there is some combination of the (binary) values of the variables that will give 1.

In other words, it asks whether the variables of a given Boolean formula can be consistently replaced by the values TRUE or FALSE in such a way that the formula evaluates to TRUE. If this is the case, the formula is called satisfiable (SAT). On the other hand, if no such assignment exists, the function expressed by the formula is FALSE for all possible variable assignments and the formula is unsatisfiable (UNSAT).

For the rest of this report, we will be referring to the conjunctive normal form version of the SAT problem, or CNF-SAT, that is one where there are $m \in \mathbb{N}$ clauses of $n \in \mathbb{N}$ literals L that are arranged in the following form:

$$\bigwedge_{i<m} \bigvee_{j<n} L_i^j$$

Where a literal L_i is either the variable x_i of its negation $\neg x_i$.

Cook's theorem, formulated in 1971 by Stephen Cook in "The complexity of theorem-proving procedures"proves that the SAT problem is NP-complete, that is every NP-hard problem can be reduced to it in polynomial time.

3-SAT is simply the problem of solving a SAT instance with at most 3 variables, so where $n \leq 3$. The following is an example of a 3-SAT problem:

$$(\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3) \tag{1}$$

This problem is satisfiable, for example for the values $x = [\text{TRUE}, \text{TRUE}, \text{FALSE}]$.

2.2 3-SAT to k-clique

We can reduce the 3-SAT problem to the k-clique problem to prove that it is NP-hard.

We do this by generating an instance of the k-clique problem from an instance of a 3-SAT problem in polynomial time. If we can do this and solve the clique problem in polynomial time then we can solve any instance of 3-SAT in polynomial time by transforming it to a k-clique instance with this same method.

For a CNF formula with k clauses we can find this reduction by building a graph with the following features:

- It is k-partite, that is for each clause of the 3-SAT problem there exists a group of indipendent vertices where no vertex is connected to another vertex of the same group.
- Each group or part consists of 3 vertices, each representing one literal of the corresponding clause.
- Each vertex is connected to all other vertices in differenct parts that represent compatible literals. Two literals are considered compatible if there could exists a solution where both are true, for example x_1, $\neg x_2$, x_3 are all compatible, while x_1 and $\neg x_1$ are not compatible.

Take the previous example of a 3-SAT problem (1). Build the graph as defined above:

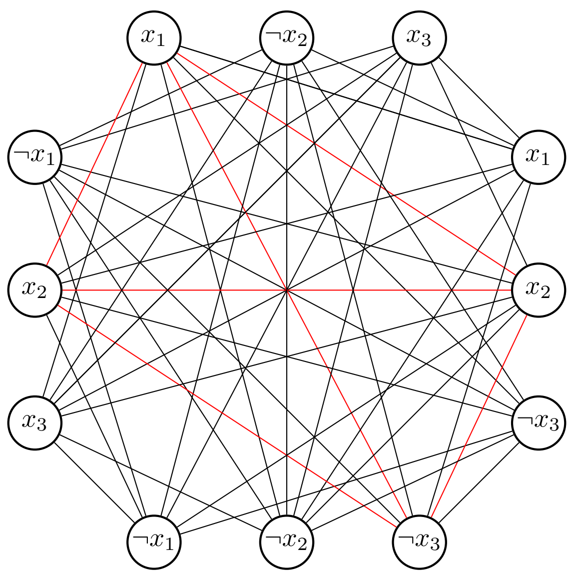


Figure 2: 3-SAT to k-clique reduction of example (1)

Notice that for each clause, there exits a node with neighbours from eavery other part. Therefore, there exists a 4-clique and by taking the literals from the nodes in the clique we have a valid solution to the 3-SAT problem.

3 K-CLIQUE TO SAT

Given a graph G=(V,E) and a number k, we will have variables x_{iv} for every 1≤i≤k and every v∈V. You should think of x_{iv} as stating that v is the ith vertex in the clique. We want to encode the following constraints:

1. For each i, there is an ith vertex in the clique:

$$\forall i \in \{1, \ldots, k\},\ x_{i1} \vee x_{i2} \vee x_{i3} \vee \cdots \vee x_{iv}$$

2. For each i,j the *ith* vertex is different from the *jth* vertex:

$$\forall v \in V, \forall (i,j) \in \{1,\ldots,k\},\ \neg x_{iv} \vee \neg x_{jv}$$

3. For each i,j, the *ith* vertex is connected to the *jth* vertex:

$$\forall v \in V, \forall (i,j) \in \{1,\ldots,k\},\ \neg x_{iv} \vee x_{ju_1} \vee \cdots \vee x_{ju_m}$$

, s.t. $u_{1,\ldots,m}$ are the neighbours of v.

If we take all these clauses together, we get a CNF which states that "the x_{iv} encode a k-clique in G". This CNF is satisfiable if and only if G contains a k-clique. In order to get a 3CNF, we need to convert the constraints of the first kind into 3-clauses. If the vertices are v_1,...,v_n, we replace $V_{v\in V} x_{iv}$ with:

$$\begin{gathered} x_{iv_1} \vee x_{iv_2} \vee y_{iv_2} \\ \neg y_{iv_2} \vee x_{iv_3} \vee y_{iv_3} \\ \neg y_{iv_3} \vee x_{iv_4} \vee y_{iv_4} \\ \ldots \\ \neg y_{iv_{n-2}} \vee x_{iv_{n-1}} \vee x_{iv_n} \end{gathered}$$

No matter what value of y we choose. Each clause must have one true literal. To make this true, we need to decide if we need true or false value in the first clause, but for the rest, it will always be true, because each have y and ¬y

Here the y_{iv} are new variables. This set of clauses is equivalent to the original clause $x_{iv_1} \vee x_{iv_2} \vee ... \vee x_{iv_n}$.

4 ALGORITHM

We have found a solution, in which, for a complete graph with v vertices:

1. We take all unconnected pair of vertices.
2. For each, at least one of them doesn't belong in a clique (If both did, it wouldn't be a clique since the two don't connect).
3. Then, check that at least k vertices are part of the clique.

5 CODE

To solve the problem, we have created the following functions. The *solve()* function will create clauses for pairs of vertices that are not connected to each other. It will do this in the following way:

- First we check that the size (which we have stored in the variable called "size") is equal to 0. If it is, we will return an error message since the network must contain at least one vertex (otherwise it is unsolvable and makes no sense).
- If it is not, we continue.

We will consider the vertices as a triangular matrix, in which in the first row will appear the connections that vertex 0 has, in the second row those of vertex 1, and so on until the last vertex (being 0 not connected with another vertex and 1 connected with another vertex).

For example, if we have 5 vertices and vertex 0 is connected to vertices 1 and 3, the first column of the matrix will be the following: 0 1 0 1 0 0

Therefore, we will go through the matrix row by row and column by column, i.e. element by element.

For each element of the matrix:

- If it is 1, i.e. the vertices are connected to each other, we continue.

- If it is 0, i.e. the vertices are not connected, we create a clause, since that row and that column (i.e. that pair of vertices) cannot be in the clique.

Then we call the function *recursiveClaurses()*, this function computes the rest of the clauses to make sure that at least k variables are true.

Once we have this, we will join the clauses created for the unconnected vertices with the clauses for the connected vertices, which will form our solution.

We will pass these clauses created for a given size to the *satSolve()* function. If it does not find a solution, no solution will be displayed, but if it does, it will be displayed. There may be more than one possible clique within a network for a given size k. This function will display one of the possible solutions at random.

The code of the function is as follows:

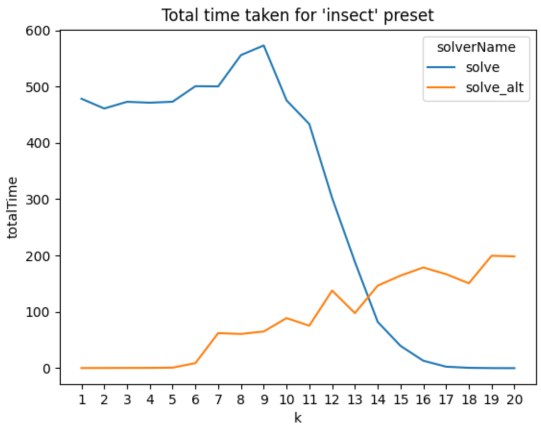
```
Function solve() {
    let size = instanceVertexCount;
    let clauses = []
    if (size == 0) throw("Graph must contain at least 1 vertex")
    for (row in instance) {
        for (col in instance[row]) {
            // For each possible combination of vertices
            if (!instance[row][col]) {
                // If there is no edge create a clause
                // "Either row or col are not in the clique"
                clauses.push([-(parseInt(row)+1), -(parseInt(col)+1)])
            }
        }
    }
    let others = recursiveClauses(size - k + 1)
    clauses = clauses.concat(others)
    let solution = satSolve(size, clauses)
    if (!solution) return solution
    return solution.filter(i => i>0).map(i => i-1)
}
```

As we have already said, the function *recursiveClaurses()* computes the rest of the clauses to make sure that at least k variables are true. We will call this function recursively, creating a vector for each iteration in which we will add clauses for each vertex, until we reach the last vertex where we will stop the recursion.

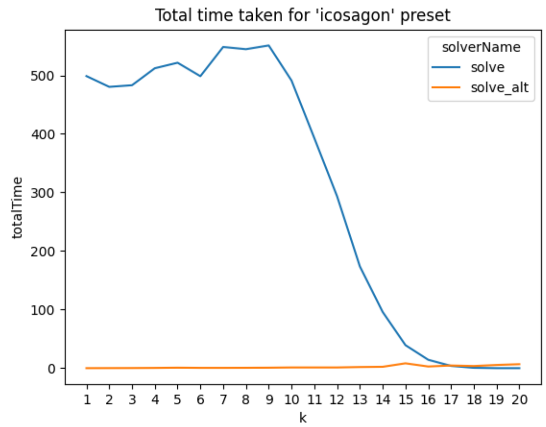
The code of the function is as follows:

```
function recursiveClauses(stop, v = [], j = 0) {
let out = []
    if (v.length == stop) return [v]
    for (let i = j; i < instanceVertexCount; i++) {
        next = [...v]
        next.push(i+1)
        out = out.concat(recursiveClauses(stop, next, i+1))
    }
    return out
}
```

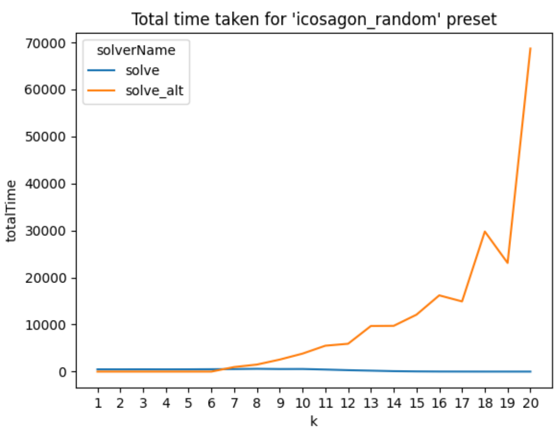
6 TEST CASES



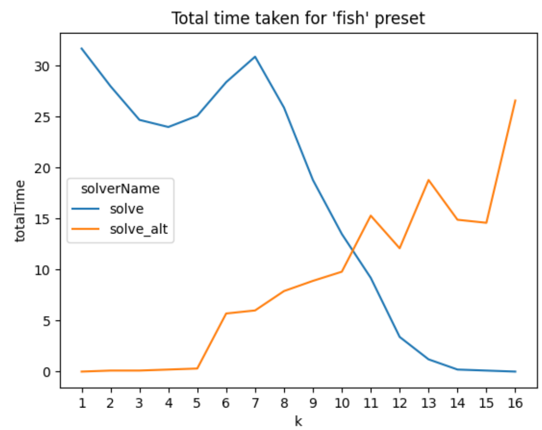
(a) Insect preset



(b) Icosagon preset with full clique



(c) Icosagon preset with random edges

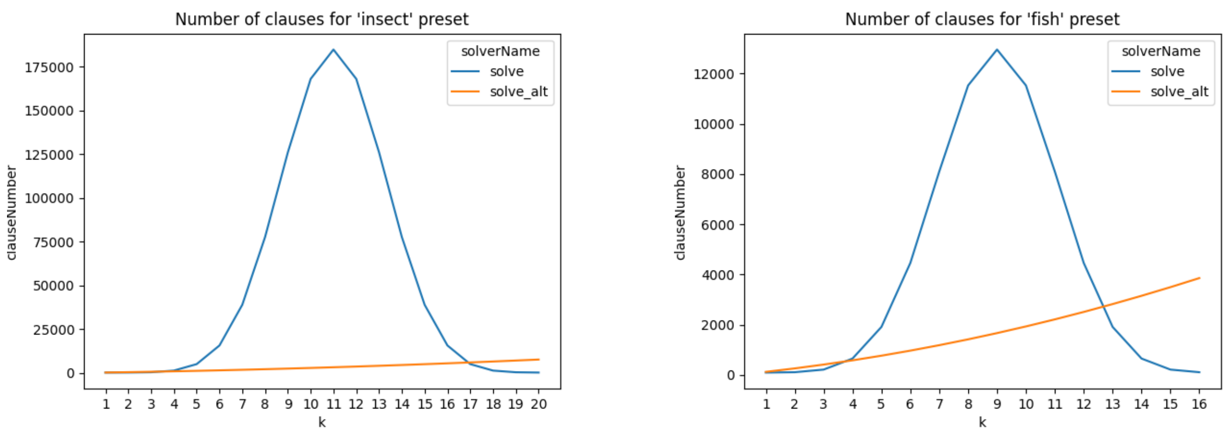


(d) Icosagon preset with random edges

Figure 3: Recorded times for the program to solve each preset at different values of k. (Times are given in milliseconds)

7 BIBLIOGRAPHY

- Clique problem
- 3 SAT
- CLIQUE is NP-complete
- SAT to clique
- reduction to clique
- Boolean satisfiability problem
- SAT
- cnf
- reduction 3 sat to clique
- cook



(a) Insect preset

(b) Icosagon preset with random edges

Figure 4: Number of clauses generated by each preset at different values of k.