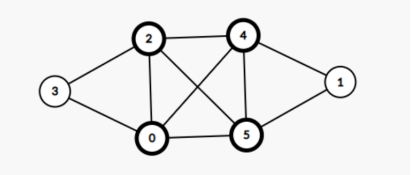
kClique polynomial reduction to SAT

Mihăilescu Eduard-Florin - 322CB
Faculty of Automatic Control and Computer Science

1 Introduction

The purpose of this project is to implement and compare different approaches of solving the **NP-Complete** problem kClique (k is a variable). There will be three categories of tests that we will analyze and discuss how they differ in terms of *computation time* when solving them with these methods.



2 Algorithms

2.1 Prerequisites

2.1.1 Software and Packages

Python version used is 3.9 and no additional packages need to be installed to run the two algorithms. In order to run the script that verifies the correctness of the solutions, the **z3-solver** package has to be installed. This can be done with the following command:

```
pip3 install z3-solver
```

2.1.2 Graph definition and Input

We will define a graph **G** as a tuple **(V,E)**, where $V = \{v_1, v_2, \ldots, v_n\}$ is a set of nodes/vertices and $E = \{e_1, e_2, \ldots, e_m\}$ is a set of edges between these nodes. For our purpose, we will only consider *undirected* graphs, meaning if there is an edge (u,v) there is also and edge (v,u).
To represent this **structure** in our programs, the following class was created which has methods for *creating* and *initializing*, *adding* and *verifying*:

```
class Graph(object):
    """ Graph data structure, undirected by default. """

    def __init__(self, connections):
        self._graph = defaultdict(set)
        self.add_connections(connections)

    def add_connections(self, connections):
        """ Add connections (list of tuple pairs) to graph """

        for node1, node2 in connections:
            self.add(node1, node2)

    def add(self, node1, node2):
        """ Add connection between node1 and node2 """

        self._graph[node1].add(node2)
        self._graph[node2].add(node1)

    def is_connected(self, node1, node2):
        """ Is node1 directly connected to node2 """
        return node1 in self._graph and node2 in self._graph[node1]
```

The input for the algorithms has the following format:

- **k** on the first line
- **n** (the number of nodes) on the second line
- **m** (the number of edges) on the third line
- **m lines** with format "a b" where a and b define an edge

```
3
6
7
1 5
4 5
3 5
1 3
4 6
2 4
3 6
```

2.2 Brute Force Approach (Backtracking) - Exponential Time

2.2.1 Usage

The code for this this approach can be found **src/kCliqueBKT.py** and can be run with following command:

```
python kCLiqueBK.py "inputFile"
```

2.2.2 Implementation

This algorithm generates all possible subsets of nodes in $O(k * C_n^k)$ time complexity (which is exponential when k is not set) using a predefined function from the **itertools** python module:

```
for subset in list(itertools.combinations(graph.get_nodes(), k)):
```

and then verifies the solution in $O(k^2)$ by checking that each pair of nodes in the subset has an edge that connects them.

```
            for i in range(k-1):
                for j in range(i+1, k):
                    # if is not connected
                    if graph.is_connected(subset[i], subset[j]) == False:
                        valid = False
                        break

                if valid == False:
                    break
```

In conclusion, the algorithm has a combined time complexity of $O(k^3 * C_n^k)$ and runs all available tests on my machine (medium specs) in $\approx$ 1s.

- **Category1** $\approx$ 0.2s
- **Category2** $\approx$ 0.4s
- **Category3** $\approx$ 0.4s

2.3 Polynomial reduction to SAT (kClique $\leq_P$ SAT)

2.3.1 Usage

The code for this this approach can be found **src/kCliqueReduction.py** and can be run with following command:

```
python kCLiqueReduction.py "inputFile"
```

2.3.2 Implementation

This **algorithm** receives the same input as the previous one: *i.e.* a graph $G = (V, E)$, an integer k, and solves the problem by transforming this input into a conjunctive normal form boolean formula, which is then piped into an efficient SAT solver.

We will have $k * |V|$ boolean variables x_{iv} in the formula with the meaning that node v occupies index i in the **Clique**. The transformation is done following a set of three rules, each of them running in polynomial time, *hence* the name.

2.3.3 Rule 1 - At each index i there should always be a node

For all i where $1 \leq i \leq k$, we build clauses with this format $\boxed{\bigvee_{v \in V} x_{iv}}$

This can be done in $O(k * |V|)$, but $|V| = n$ and $k \leq n \implies O(n^2) =$ **polynomial** time

```
        for i in range(1,k+1):
            resultSAT = ''.join((resultSAT,'('));
            for v in range(1,n+1):
                resultSAT = ''.join((resultSAT, 'x', str(i), str(v), ' V '));
            resultSAT = resultSAT[:-3];
            resultSAT = ''.join((resultSAT, ') ^ '));
```

2.3.4 Rule 2 - A node can be uniquely placed at a single index i in the clique

For all i, j where $1 \leq i < j \leq k$, and $v \in V$ we build clauses with this format $\boxed{\overline{x}_{iv} \vee \overline{x}_{jv}}$

This can be done in $O(|V| * k^2)$, but $|V| = n$ and $k \leq n \implies O(n^3) =$ **polynomial** time

```
        for v in range(1,n+1):
            for i in range(1,k):
                for j in range(i+1, k+1):
                    resultSAT = ''.join((resultSAT, "(~x", str(i), str(v), " V ", "
    ~x", str(j), str(v), ")", " ^ "));
```

2.3.5 Rule 3 - Any two nodes in the clique should have an edge between them

In order to satisfy this statement we can go through all of the edges that don't exist, *i.e.* the **complement** of G and check that the two nodes that define that edge cannot be at the same time in the clique. To accomplish this, we can do the following:

For all $v, u \in V$ where $(v, u) \notin E$ and all i, j $where 1 \leq i < j \leq k$, we build clauses with this format $\boxed{\overline{x}_{iv} \vee \overline{x}_{ju}}$

This can be done in $O(|V|^2 * k^2)$, but $|V| = n$ and $k \leq n \implies O(n^4) =$ **polynomial** time

```
        for v in range(1, n+1):
            for u in range(1,n+1):
                # if there is not an edge
                if graph.is_connected(v,u) == False:
                    for i in range(1,k):
                        for j in range(i+1,k+1):
                            resultSAT = ''.join((resultSAT, "(~x", str(i), str(v),
    " V ", "~x", str(j), str(u), ")", " ^ "));
```

2.3.6 Conclusion

To recap, the algorithm has a combined complexity of $O(n^4)$ which verifies that our reduction is polynomial, although on my machine, it runs much slower than the previous algorithm $\approx$ 60s.

- **Category1** $\approx$ 2s
- **Category2** $\approx$ 56s
- **Category3** $\approx$ 2s

A slight improvement that I can see is to implement the algorithm in programming language that has string buffers because there is a lot of string concatenation which runs pretty slowly in python.

3 Comparison

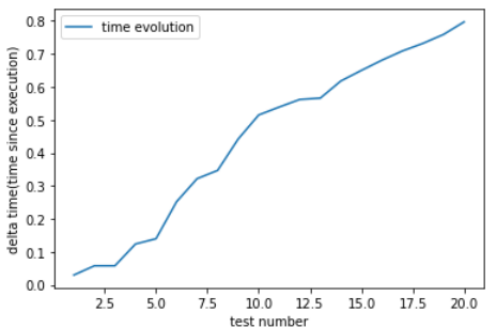
By running the checker we can see the difference in execution time between the two algorithms called speedup, which in mathematical terms translates to $\frac{\Delta t_{red}}{\Delta t_{bkt}}$, where Δt_{red} is the total execution time for the polynomial reduction algorithm and Δt_{bkt} is the total execution time for the exponential algorithm.

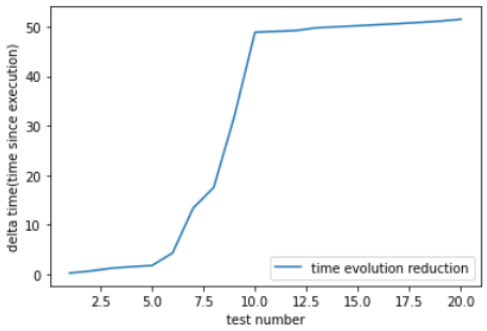
- **Category1** ≈ 18
- **Category2** ≈ 55
- **Category3** ≈ 10

A smaller speedup means that is more favorable to use the reduction algorithm, and a bigger one, the complete opposite.

In order to analyze why there is such a steep difference between the 3 categories we need to state the particularities of each category. that being said, we observe that:

- **Category1** has n and k pretty small and close to each other
- **Category2** has a very small k and and a large number of nodes(n) and edges(m)
- **Category3** has a smaller number of edges compared to the other two and k is pretty close to n and m





It can be observed that the plot on the left which shows the evolution in time of the tests when running the **bkt** algorithm is fairly linear, compared to the second, reduction plot in which we see an exponential growth for the second category. This is due to the huge *discrepancy* between k and $|V| = n$ which in turn gives a smaller boolean expression that is harder to solve by the SAT solver because it is less restrictive then an expression with more variables.

Also, by the same logic expressed earlier, the number of *edges* influences the run-time because fewer edges, add in turn more clauses to our boolean expression because the graphs complement is denser (more edges).