Algorithms 2

by Daniel Rosenberg and Michael Trushkin

These are lecture notes for the course Algorithms and Optimization at Ariel University Students are assumed to have basic knowledge of graph theory, crucial definitions will be reminded. While we strive for accuracy, these notes may contain mistakes. Students are encouraged to report any errors or typos they encounter.

1. Complexity

1.1. Basic definitions

An algorithm is called *polynomial-time* if its running time is bounded by $O(n^c)$ where n is the length of the input and c is some (maybe huge) constant.

Example.
Common examples of polynomial-time algorithms include: DFS, BFS, Dijkstra's algorithm, 2-Coloring, and various sorting algorithms.

Skipping some fundamental knowledge[1] we can now informally define:

Definition 1.1.1.
$\mathbf{P}$:= The set of problems that have a polynomial algorithm.

Definition 1.1.2.
$\mathbf{NP}$:= The set of problems that have a **non-deterministic** polynomial algorithm.

Definition 1.1.3.
$\mathbf{NPC}$:= The set of problems that if we find a polynomial-time algorithm that solves them then $\mathbf{P} = \mathbf{NP}$.

1.2. Self reduction

There are two types of problems: *decision problems* and *search problems*. Decision problems are those that require a 'yes' or 'no' answer, whereas search problems require finding an actual solution if one exists. For example, finding a path between two nodes the decision problem will be "**Is** there a path between node A and node B?" and the search problem will be "**What** is the path between node A and B?" both of these can be solved by the same algorithm. A non trivial problem is finding k-clique, given a graph G, the decision problem is:

[1]For more information about this topic the reader is adviced to loop up Computational Comlexity by Aroara and Barak

Questions 1.2.1 .
Is there a clique of size k in G?

The *language* $k - \text{clique} := \{G \mid \omega(G) \geq k\}$ is the set of all graphs that contain a clique of size at least k, answering Questions 1.2.1 is the same as asking if some graph is in k-clique or not. The search problem denoted SEARCH-k-CLIQUE is given some graph G find a copy of K_k in G, if there is no such copy return `null`.

One might ask whether the problems are equivalent, by that we mean given a way to solve one, can we solve the other? By the nature of the problems if we have a way to find the solutions, it is easy to answer whether there is a solution. The other way seems more complicated, but as it turns out for **NPC** languages it is possible.

Claim 1.2.1 .
If the decision problem for k-clique can be solved in polynomial time, then there is a polynomial-time algorithm for SEARCH-k-CLIQUE.

Proof. In order to prove the claim, we need to provide a polynomial time algorithm. As we dont know any algorithm, we will use the fact that we have a polynomial algorithm for k-clique denoted by A.

Algorithm 1:

```
procedure SELF-REDUCTION(G)
    if $A(G) == 0$ then
        return null
    end

    while $v(G) > k$ do
        pick $v \in V(G)$
        if $A(G - v) == 1$ then
            $G \leftarrow G - v$
        end
    end
    return $G$
end
```

The rest of the proof is left for the reader. □

The claim above demonstrates that decision and search problem are equivalent[2], thus we can focus only on decision problems.

1.3. NP-completeness

We know how to tell if a language is in **P**, how can we know if a language is in **NP**? We say that a language $L \in \mathbf{NP}$ if there is a polynomial algorithm M, such that for every instance $x \in L \Leftrightarrow \exists y \; s.t \; |y| < p(|x|)$ and $M(x, y) = 1$, where p is some polynomial. The string y is called a *witness* and the algorithm is called a *verifying algorithm*, and the witness should play the role of the solution. If such a solution for the problem exists, our algorithm should verify it and accept it, otherwise, the algorithm should reject every option for a solution.

[2]In our setting only

Remark 1.3.1 .
The witness y is the non-deterministic choices that the algorithm can make.

Claim 1.3.1 .
k-clique is in **NP**

Proof. To prove this, we must provide a polynomial-time verifier M that takes a graph G and a subset of vertices Y, and checks whether Y forms a clique of sufficient size.

Algorithm 2: Verifying algorithm for k-clique

```
procedure $M$(G, Y)
    ▷ Check if the size of the group is large enough
    if $|Y| < k$ then
        return false
    end

    ▷ Check all the vertices are real
    for $v \in Y$ do
        if $v \notin V(G)$ then
            return false
        end
    end

    ▷ Check all the edges exist
    for $v, u \in Y$ do
        if $(v, u) \notin E(G)$ then
            return false
        end
    end

    return true
end
```

if $G \in k$-clique, then there is a subset $V' \subseteq V(G)$ such that $G[V'] \cong K_k$, and $M(G, V') = 1$ if $G \notin k$-clique, then no matter which subset $V' \subseteq V(G)$ we take, $G[V']$ will never be a clique, meaning there will be some edge missing, and $M(G, V') = 0$ □

1.4. Reductions

Suppose we have two languages $L_1, L_2 \subseteq \{0,1\}^*$, can we know which one of them is *harder*? The intuition is that if by solving L_2, we can solve L_1, then L_2 is harder. This is done by "translating" our problem from L_1 to L_2, solving our L_2 problem, and then answering accordingly. The translation between languages is called a *reduction*, formally

Definition 1.4.1 (polynomial time reduction).
Given two languages $L_1, L_2 \in \mathbf{NP}$, we write $L_1 \leq_p L_2$ if there exists a function $f : \{0,1\}^* \to \{0,1\}^*$ and a polynomial $p : \mathbb{N} \to \mathbb{N}$, such that:

- $x \in L_1 \Leftrightarrow f(x) \in L_2$
- for every $x \in \{0,1\}^*$, f runs in $p(|x|)$ time.

Now if we have a black box A that solves L_2, and we are given an instance x for which we need to decide whether $x \in L_1$, all we have to do is run $A(f(x))$ and return the same, that means that L_2 is at least as hard as L_1.

Definition 1.4.2.
A language $L \subseteq \{0,1\}^*$ is said to be NP-hard if $L' \leq_p L$ for every $L' \in \mathbf{NP}$

We can now refine our definition for **NPC**:

Definition 1.4.3.
A language $L \subseteq \{0,1\}^*$ is said to be NP-complete if $L \in \mathbf{NP}$ and L is NP-hard

One should be able to see now why having a polynomial algorithm for an **NPC** problem will result in $\mathbf{P} = \mathbf{NP}$, and the first step in solving this is to find such a language.

Let $x_1, \ldots x_n$ be boolean variables (x_i can be assigned either 0 or 1). A boolean formula φ is said to be in conjunctive normal form(CNF) if it has the form

$$\varphi = \underbrace{(x_1 \vee x_{17} \vee \overline{x_{25}} \vee x_{80})}_{\text{clause}} \qquad \overbrace{\wedge}^{\text{and between clauses}} \qquad \underbrace{x_9}_{\text{also clause}} \quad \wedge \ldots$$

The appearances of the variable x_i are called *literals*. Each literal can be positive(x_i) or negative($\overline{x_i}$). A clause is a disjunction(OR) of literals, and the formula φ is a conjunction(AND) of these clauses. An assignment to the variables of φ evaluates to either `true` or `false`, and φ is said to be satisfiable if there is some assignment such that φ evaluates to `true`, such assignment is called *satisfying assignment*.

Definition 1.4.4.
CNF-SAT := $\{\varphi$: φ is a satisfiable CNF formula$\}$

The follwoing theorem was proved by Cook and Levin:

Theorem 1.4.1.
CNF-SAT is npc.

Fortunately for the students, the proof is beyond the scope of this course and will be omitted, although curious students can look at up in Computational Comlexity by Aroara and Barak