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. Polynomial-time algorithms

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.

Definition 1.1.
$\mathbf{P}$:= The set of problems that have a polynomial algorithm.

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:

Questions 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 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 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 will use the assumption that there is a polynomial-time algorithm for k-clique and show a polynomial time algorith for SEARCH-k-CLIQUE. We propuse the following algorithm:

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
```

If G does not have a valid clique of size k, then Algorithm 1 will return `null` on line 3 as required. Otherwise, we know that there is a clique of size k in G(maybe more then one), so if any point by removing some vertex $v \in V(G)$ we get that $A(G - v) = 0$ we know that v is essential to the clique and leave it in $V(G)$. After going over all the vertices we are left only with the essential vertices, leaving us with a clique of size k.

Now, we need to show that the algorithm runs in polynimal time. As we assume that A run in polynomial time, there is some polynom p such that the running time of $A(G)$ is bounded by $f(G)$. The first check is done in $f(G)$ time, then the algorithm will go through all vertices, remove them from the graph and call A on the modified graph, all of this will take $n \cdot f(G)$ time in the worst case making the runnig time of out algorith $O((n + 1)f(n))$ which is polynomial.□

The claim above demonstrates that decision and search problem are equivalent[1], thus we can focus only on decision problems.

3. NP-completeness

While the class **P** contains a large portion of the problems students have faced so far, as it turs out the majoriy of the problems are not easy at all. Lets suppose that you are proffesional safe cracker who are in a competition with your friend who can hack a safe faster. The safe has state of the art defence mechanisims, making it very hard to crack, none of the known method works for you. You are trying entring every combination, but this takes a lot of time. After some time your friend tell you that he was able to find a the code! and gives you a the book "One Million Digits Of Pi", and says the the password is in there. You tell him that this does not count, as there is no way to know if he is right or not, just trying to read the book will take a lot of time. But if he gives you the password, you have a way to know if he won or not,

[1]In our setting only

just enter the password and see if the safe opens and if it does, then it means that your friend is won. But if it is not the right password, is there a way for you to know the real password? or even know that such password exists? This demonstarins captures the essence of out next complexity class **NP**, the set of problems where a proposed solution can be verified quickly. Formally,

Definition 3.1 (NP class)**.**
A language L is said to be in **NP** if we have a polynomial-time algorithm M such that

$$x \in L \Leftrightarrow \exists y \; s.t \; |y| < p(|x|) \text{ and } M(x,y) = 1$$

where p is some polynomial

Remark 3.1.
In most literture y is called a *witness* and V is called *veryfing algorithm*, where y plays the role of the answer, and M should just verify if the answer is correct.

We are ready to meet out first **NP** language

Claim 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 $M(G, V') = 0$

TODO:**time complexity** □

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 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 4.2.
A language $L \subseteq \{0,1\}^*$ is said to be NP-hard if $L' \leq_p L$ for every $L' \in \mathbf{NP}$

Definition 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

Or between literals Literal

$$\varphi = \underbrace{(x_1 \lor x_{17} \lor \overline{x_{25}} \lor x_{80})}_{\text{clause}} \land \underbrace{x_9}_{\text{also clause}} \land \ldots$$

And between clauses

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 4.4.
CNF-SAT := $\{\varphi$: φ is a satisfiable CNF formula$\}$

The follwoing theorem was proved by Cook and Levin:

Theorem 4.1 (Cook-Levin)**.**
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.

Following the discovery of the first **NPC** language, many other problems have been shown to be in **NPC** as well. The first in which we are interested is a variation of the classical CNF. For an integer $k \in \mathbb{N}$, define

$$k\text{-CNF-SAT} := \{\varphi \mid \varphi \text{ is a CNF formula in which each clause has exactly } k \text{ literals}\}.$$

Example.
$(x_1 \vee \overline{x_1}) \wedge (x_2 \vee x_3)$ is in 2-CNF-SAT,
$(x_1 \vee x_4 \vee x_5) \wedge (x_1 \vee \overline{x_2} \vee x_3)$ is in 3-CNF-SAT.

Despite their similar definitions, there is a fundamental gap between these two problems as can be seen in the following claim:

Claim 4.1 (proof is delegated to the practice session)**.**
2-CNF-SAT is in **P**.

Claim 4.2.
3-CNF-SAT is in **NPC**.

The proof that 3-CNF-SAT $\in$ **NP** is omitted and left for the reader. Next, we need to show that for every language $L \in \mathbf{NP}$, $L \leq_p$ 3-CNF-SAT, which can be quite hard for us to do. Instead we will use the fact that reductions are transitive:

Lemma 4.1.
If $L_1 \leq_p L_2$ and $L_2 \leq_p L_3$, then $L_1 \leq_p L_3$.

By using this property, we can skip the long proof, and instead we show a reduction from a known **NPC** language. We are now ready to prove Claim 4.2:
Proof. We will show that CNF-SAT $\leq_p$ 3-CNF-SAT. In order to show this, we need to define a function f such that:
1. **Running Time.**:f runs in polynomial time.
2. **Correctness**: for every formula φ, $\varphi \in$ CNF-SAT $\Leftrightarrow f(\varphi) \in 3 -$ CNF-SAT

We will define f as follows: For each clause $l_1 \vee l_2 ... \vee l_m$ of φ, we will replace it by a *gadget* of clauses according to the following rules:

1. If $m = 3$, then copy the clause as is.
2. If $m < 3$, then repeat one of the literals until the clause has exactly 3 literals. For example the literal $l_1 \vee l_2$ will become $l_1 \vee l_1 \vee l_2$.
3. If $m > 3$ then create $m - 3$ **new** variables named $y_1, ... y_{m-3}$ and replace $l_1 \vee l_2 ... \vee l_m$ with the following:

$$(l_1 \vee l_2 \vee y_1) \wedge (\overline{y_1} \vee l_3 \vee y_2) \wedge (\overline{y_2} \vee l_4 \vee y_3) \wedge ... \wedge (\overline{y_{m-3}} \vee l_{m-1} \vee l_m).$$

It is easy to see that the first two steps take a constant amount of time, in the last condition the creation of $m - 3$ takes $O(m)$ time per clause, and we create $m - 3$ new clauses, each of which takes constant time, put everything together the running time of step 3 is $O(m)$ making our entire algorithm polynomial. Now, to prove the correctness we need to prove both directions:

⇒(Completeness): Assume that $f(\varphi) \in$ CNF-SAT, This implies there exists a satisfying assignment a for φ, We must show that there exists a satisfying assignmenta' for $f(\varphi)$. For each variable that was in φ we copy its value from a to a' unchanged, this ensures that all clauses with 3 or fewer literals to stay satisfied by a'. Let $l_1 \vee l_2 ... \vee l_m$ be the literals of a clause of size at least 4, as the clause is satisfied under a, there is some $i \in [m]$ such that $a(l_i) = 1$. We extend a to the auxiliary variables y_j as follows: for all $j < i - 1$, set $y_j = 1$, otherwise set $y_j = 0$. The clause containing l_i is satisfied because of l_i, all the clauses before them are satisfied due to the positive literals of the new variables being 1. All the clauses that appear after the clause containing l_i are satisfied due to the negative literals of the new variables being 0, meaning $f(\varphi) \in 3-$CNF-SAT.

⇐(Soundness): Assume that $f(\varphi) \in$ 3-CNF-SAT, This implies there exists a satisfying assignment a' for $f(\varphi)$, We must show that there exists a satisfying assignment$a$ for φ. We argue the copying the assignment of the orginal variables form a' to a will produce a satisfying assignment for φ. To see this, assume torward a contradiction that a is not a satsfying assigment for φ, that means that there is a clause $c = l_1 \vee l_2 \vee ... \vee l_m$ that is unsatisfied. If $m \leq 3$ Since f copies these clauses (or simply repeats literals), and a uses the same values as a', the corresponding clause in $f(\varphi)$ would also be unsatisfied. This contradicts our assumption that a' is a satisfying assignment. For the case $m > 3$. If c is not satisfied, then all its literals must be false: $l_1 = l_2 = ... = l_m = 0$. In order to satisfy the clause gadget we need to satisfy all of the clauses it contains. In the first clause we have $l_1 = l_2 = 0$, which requires $y_1 = 1$ in oreder for the clause to be satisfied. For the second clause we have $\overline{y_1} = l_3 = 0$ which requires $y_2 = 1$. Following this chain of logic, each y_j is forced to be 1 to satisfy the j-th clause. Finnaly, we reach the last clause: $\overline{y_{m-3}} \vee l_{m-1} \vee l_m$. Here $\overline{y_{m-3}} = l_{m-1} = l_m = 0$. This last clause cannot be satisfied, which contradicts the assumption that a' satisfies $f(\varphi)$. Therefore, a must be a satisfying assignment for φ, and thus $\varphi \in$ CNF-SAT. □

4.1. Independent set

For a graph G, let $\alpha(G)$ denote its maximum independent set. Define:

Definition 4.1.1.
IS := $\{< G, k >: \alpha(G) \geq k\}$.

Theorem 4.1.1.
IS is in **NPC**.

Proof. We show that 3-CNF-SAT $\leq_p$ IS, proving that IS $\in$ **NP** is left as homework. Here the reduction might seem a little confusing at first, we are translating a formula into a graph and a number. Given a 3-CNF formula φ, we construct a graph G_φ as follows:
1. **Triangles**: For each clause l_1, l_2, l_3 we create a triangle with 3 vertices named $v_{l_1}, v_{l_2}, v_{l_3}$.
2. **Consistency Edges**: For any pair of complementary literals $x_j, \overline{x_j}$ that are in different clauses, put an edge between the vertices that correspond to the literals.

We return the pair $< G_\varphi, m >$ where m is the number of clauses. While not intuitive at first, the number m is chosen because an independent set can contain at most one vertex from each triangle.

This algorithm indeed runs in polynomial time, as looping through all the clauses is linear in the number of clauses, and the maximum number of edges one can add is n^2. Now we need to prove

$$\varphi \in \text{3-CNF-SAT} \Leftrightarrow < G_\varphi, m > \in \text{IS}$$

$\Rightarrow$: Let φ be satisfiable and let a_φ be a satisfying assignment for φ. As a_φ satisfies φ, at least one literal of each clause is satisfied, pick any one such literal from each clause. The set of vertices corresponding to the set of literals chosen is independent in G_φ and has size of m.
$\Leftarrow$: Suppose G_φ has an independent set size m. Define an assignment a_φ for φ by assigning values to the variables of φ so that all the literals captured by the independent set are set to `true`, all others can be set to arbitrary values in a consistent manner. Because the graph is composed of triangles, each triangle can have only one vertex in the independent set, so each clause will have one variable satisfying the clause. Moreover, because complementary literals are connected, only the positive or negative literals of each variable can be in the independent set ensuring our assignment is consistent(there cannot be a variable assigned both `true` and `false`). $\square$

4.2. Graph coloring

For a graph G denote by $\chi(G)$ the least $k \in \mathbb{N}$ such that G is k-colorable.

Definition 4.2.1.
k-COL $:= \{G : \chi(G) \leq k\}$

It is well known that 2-COL$\in$ **P**.

Theorem 4.2.1.
3-COL $\in$ **NPC**.

In order to prove the theorem, we intreduce a new **NPC** language. Let φ be a formula, φ is said to be *not all equal satisfiable*(NAE-SAT) if it has a satisfying assigmnet such that in each caluse it has at least one satisfied literal and at least one that is not satisfied.

Definition 4.2.2 (NAE-k-CNF-SAT).
NAE-k-CNF-SAT:= $\{\varphi : \varphi$ is NAE-SAT, with exactly k literals in each clause$\}$.

The proof that NAE-k-CNF-SAT is NP-Complete is left to the practice sessions. We are ready to start our proof.
Proof. We skip again the proof that 3-COL $\in$ **NP**, which is left as homework, and show that k-NAE-SAT $\leq_p$ 3-COL. Given a 3-CNF formula φ, define G_φ as follows:
1. Start with a single vertex D. This is our *Don't care vertex.*
2. For each variable x_i of φ, add two new vertices $x_i, \overline{x_i}$, add an edge between them, and connect both to D. This are our *variable gadgets.*
3. For each clause $l_1 \vee l_2 \vee l_3$ we create a triangle with 3 vertices named l_1, l_2, l_3, this is our *clause gadget.*
4. For each literal in the clause gadgets, connect it to the complementary variable from the variable gadget.

We skip the proof that the algorithm runs in polynomial time. It remains to prove that

$$\varphi \in \text{NAE-}k\text{-CNF-SAT} \Leftrightarrow G_\varphi \in \text{3-COL}$$

$\Rightarrow$: Given a satisfying NAE assignment a_φ for φ, define the follwing 3-coloting of G_φ:
- D will be colored as D
- For each variable x_i, if x_i is assigned `true` under a_φ color x_i as T and $\overline{x_i}$ in F, otherwise color x_i as F and $\overline{x_i}$ in T
- For each clause gadget, scan the corresponding clause c, color first literal that is assigned `true` with T, the first that assigned `false` with F, and color the vertex that was left with D.

It is clear that edges inside vertex/clause gadgets have both ends in different colors, it remains to show that edges between vertex and clause gadgets has its ends colored in different colors. without loss of generality, let x be a variable assigned `true` by a_φ. As a_φ is proper, all of the vertices baring $\overline{x}$ found in a clause gadgets are colored either F or D, and the claim follows.

$\Leftarrow$: Given a 3-coloring ψ of G_φ, we define a NAE-satisfying assignment for φ. As all of the variable gadgets form a triangles with a common vertex D, it leaves them with two colors to be chosen. Take the variable gadget for an arbitrary variable x and set the color under x to be `true` and the color under $\overline{x}$ to be `false`. This defines a valid assignmet to the variables of φ. The assignment is NAE as each clause gadget has one variable colored `true` and one `false`. □

4.3. Max-cut

Given a graph G, a *cut* is defined as the set of edges between $S \subseteq V(G)$ and $\overline{S} = V(G) \setminus S$. We denote the set of edges by $E_G\left(S, \overline{S}\right) := \left\{(u, v) : (u, v) \in E(G), u \in S, v \in \overline{S}\right\}$, and the number of edges by $e_G\left(S, \overline{S}\right) := \left|E_G\left(S, \overline{S}\right)\right|$ Denote by $\sigma(G) := \max_{S \subseteq V(G)} e_G\left(S, \overline{S}\right)$.

Definition 4.3.1 (MAX-CUT).
MAX-CUT $:= \{< G, k >: \sigma(G) \geq k\}$

Theorem 4.3.1.
MAX-CUT$\in$ **NPC**

Proof. We again skip the proof that MAX-CUT$\in$ **NP**, and show that MAX-CUT $\leq_p$ 3-COL. Given a 3-CNF formula φ, define G_φ as follows:

1. For each variable x_i of φ, add two new vertices $x_i, \overline{x_i}$ and an edge between them. These are our *variable gadgets*.
2. For each clause $l_1 \vee l_2 \vee l_3$, we create a triangle with vertices l_1, l_2, l_3; this is our *clause gadget*.
3. For each literal vertex in a clause gadget, connect it to the complementary literal vertex in the variable gadget.

(This is the same as the NAE-k-CNF-SAT reduction to 3-COL but without the vertex D). We return the pair $< G_\varphi, n+5m >$. We skip the proof that the algorithm runs in polynomial time. It remains to prove that

$$\varphi \in \text{NAE-}k\text{-CNF-SAT} \Leftrightarrow < G_\varphi, n+5m > \in \text{MAX-CUT}$$

$\Rightarrow$: Given a satisfying NAE assignment a_φ for φ, define $S \subseteq V\left(G_\varphi\right)$ to consist of all vertices whose label is a literal assigned `true` under a_φ. As a_φ is consistent, all variable gadgets must cross $\left(S, \overline{S}\right)$ adding n edges to the cut. As a_φ is a valid NAE assignment, at least two edges cross $\left(S, \overline{S}\right)$ in every clause gadget, adding $2m$ edges to the cut. Each edge between a variable and clause gadget has the form $(l, \bar{l})$, which means it is also in the cut, adding $3m$ edges to the cut. Overall, we count at least $n+5m$ edges.

$\Leftarrow$: Suppose that $< G_\varphi, n+5m > \in$ MAX-CUT. Let $\left(S, \overline{S}\right)$ be a cut of G_φ such that $e_{G_\varphi}\left(S, \overline{S}\right) = n+5m$. Define the assignment a_φ for φ in which all variable gadget literals found in S are assigned `true` and all remaining are assigned `false`. This defines a consistent assignment. It remains to prove that a_φ is NAE-satisfying. Every edge between a variable and clause gadget is also in the cut and has the form $(l, \bar{l})$. Because $e_{G_\varphi}\left(S, \overline{S}\right) = n+5m$, each clause gadget has 2 edges crossing $\left(S, \overline{S}\right)$, meaning each gadget has at least one vertex in S and one in $\overline{S}$. Take a vertex l in a clause gadget that is in S. Then the edge $(l, \bar{l})$ implies that $\bar{l} \in \overline{S}$, meaning the literal corresponding to l is assigned `true`. In a similar manner, if l is in $\overline{S}$ the then the edge $(l, \bar{l})$ means the $\bar{l} \in S$ implying that l is assigned `false`. $\square$