IJETST- Vol.||11||Issue||05||Pages 8064-8076||February||ISSN 2348-9480 2024

Open access Journal International Journal of Emerging Trends in Science and Technology

DOI: http://dx.doi.org/10.18535/ijetst/v2024.04

P ≠ NP: A Formal Proof

Author

A. Mahdoum

Centre de Développement des Technologies Avancées, Cité 20 Août 1956, Baba Hassan, Algiers, Algeria, Department of computer science, University Saad Dahlab of Blida, Algeria

Abstract

According to the conjecture that P ≠ NP, we recall in this paper that class NP includes P, NP-intermediate and NP-complete problems (some of Co-NP and NP-hard problems also belong to NP). It is obvious that if a single problem belonging to NP is formally proved non-polynomial, then P ≠ NP no longer remains a conjecture but rather becomes a formal statement. In this purpose, we formally prove that the Graph-isomorphism problem (belonging to class NP) is non-polynomial time, which leads that P ≠ NP is a formal statement, not a conjecture.

Keywords: *NP complete problems, NP hard problems, NP-intermediate problems, polynomial-time problems, P ≠ NP*

1 Introduction

In this paper we prove that P ≠ NP is not a conjecture. As this research problem is fundamental and not yet solved, we only recall, in section 2, the fundamental notions of NP-completeness[1]. Then, in section 3, we formally prove that the Graph-isomorphism problem is not time-polynomial. As this problem is known as belonging to the NP class [2]), then it is obvious to formally claim that P ≠ NP is not a conjecture. Finally, we conclude the paper.

Areas:

- **68Q15** Complexity classes (hierarchies, relations among complexity classes, etc.)
- **68Q17** Computational difficulty of problems (lower bounds, completeness, difficulty of approximation, etc.)

2 Fundamental Notions on NP-Completeness

2.1 Computational complexity

2.1.1 Introduction

The execution time of a program depends on several factors:

- Program data: for example, sorting out about ten numbers requires less execution time than sorting out millions of numbers;
- Quality of the code generated by the compiler: codes generated by programming languages do not have the same quality (it is the case of C, C++ and Java: if the application is not object-oriented and it would be executed on a given platform, it is better to implement it in C, which generates a "lighter" code);

- Computational complexity: this metrics gives the developer an idea of the execution time of his program, independently of programming language and platform (processor, memory capacity, etc.). Computational complexity is therefore an important concept, all the more so as it offers the developer an indication on whether his program takes a reasonable CPU time to reach results, even when significant resources (processor frequency, memory capacity) are at his disposal. In other terms, it may signal the designer the need to modify his algorithm, if it takes an "infinite" time to reach results.

2.1.2 Big O notation

Definition: Computational complexity T(n) is O(f(n)) if there are constants c and n_0 such that: $T(n) \leq c * f(n) \ \forall\ n \geq n_0$; $c > 0$; $n_0 \geq 0$

Example 2.1: $T(n) = (n+1)^2$

T(n) is $O(n^2)$. Indeed, $T(n) \leq 4 * n^2 \ \forall\ n \geq 1$ ($c=4$; $n_0=1$)

We will subsequently see how the function T(n) associated with a given algorithm can be determined.

Example 2.2: Consider $T(n) = n^2+n$. Is it possible to consider that T(n) is O(n) ?

If yes, we would have: $n^2+n \leq c * n \Leftrightarrow n^2+n(1-c) \leq 0 \Leftrightarrow n(n+1-c) \leq 0$. Since $n \geq 0$, we would have: $c \geq n+1$ $\Rightarrow$ c is not a constant and therefore T(n) is not O(n).

2.1.2.1 Sum

If $T_1(n)$ is O(f(n)) and $T_2(n)$ is O(g(n)), then $T_1(n)+ T_2(n)$ is O(max(f(n),g(n)))

Proof:

$T_1(n)$ is $O(f(n)) \Leftrightarrow \exists\ c_1 > 0, n_1 \geq 0: T_1(n) \leq c_1 f(n) \ \forall\ n \geq n_1 \Rightarrow T_1(n) \leq c_1 \max(f(n), g(n)) \ \forall\ n \geq n_1$

$T_2(n)$ is $O(g(n)) \Leftrightarrow \exists\ c_2 > 0, n_2 \geq 0: T_2(n) \leq c_2 g(n) \ \forall\ n \geq n_2 \Rightarrow T_2(n) \leq c_2 \max(f(n), g(n)) \ \forall\ n \geq n_2$

$\Rightarrow T_1(n) + T_2(n) \leq (c_1 + c_2) \max(f(n),g(n)) \ \forall\ n \geq \max(n_1, n_2)$

Generally speaking, if $T_i(n) \leq c_i f_i(n) \ \forall\ n \geq n_i$; i=1, 2, …, p then:

$$\sum_{i=1}^{p} T_i(n) \leq \max_{1\leq i\leq p}[f_i(n)] \sum_{i=1}^{p} c_i \ \forall\ n \geq \max_{1\leq i\leq p} n_i \Rightarrow \sum_{i=1}^{p} T_i(n)\ is\ O\left(\max_{1\leq i\leq p} f_i(n)\right)$$

2.1.2.2 Product

If $T_1(n)$ is O(f(n)) and $T_2(n)$ is O(g(n)), then $T_1(n)* T_2(n)$ is O(f(n)*g(n))

Proof:

$T_1(n)$ is $O(f(n)) \Leftrightarrow \exists\ c_1 > 0, n_1 \geq 0: T_1(n) \leq c_1 f(n) \ \forall\ n \geq n_1$

$T_2(n)$ is $O(g(n)) \Leftrightarrow \exists\ c_2 > 0, n_2 \geq 0: T_2(n) \leq c_2 g(n) \ \forall\ n \geq n_2$

$\Rightarrow T_1(n)* T_2(n) \leq (c_1 * c_2) f(n) * g(n) \ \forall\ n \geq \max(n_1, n_2) \Rightarrow T_1(n) * T_2(n)$ is $O(f(n) *g(n))$

Generally speaking, if $T_i(n) \leq c_i f_i(n) \ \forall\ n \geq n_i$; i=1, 2, …., p then:

$$\prod_{i=1}^{p} T_i(n) \leq \prod_{i=1}^{p} c_i \prod_{i=1}^{p} f_i(n) \ \forall n \geq \max_{1\leq i\leq p} n_i \ ,$$

which means $\prod_{i=1}^{p} T_i(n)$ is $O\left(\prod_{i=1}^{p} f_i(n)\right)$

2.1.3 Ω Notation

Definition: Computational complexity T(n) is Ω(g(n)) if there are constants c and n_0 such that: $T(n) \geq c * g(n) \ \forall\ n \geq n_0$; $c > 0$; $n_0 \geq 0$

Example: $T(n) = n^2+n$. It can be verified that T(n) is Ω(n) due to the fact that $T(n) \geq n^2 \ \forall\ n \geq 0$

2.1.4 Calculation of T(N)

It is obvious that O and Ω notations cannot be used unless T(n) is determined. The following section shows how the function T(n) associated with a given algorithm is determined.

Example 2.3:

ALGORITHM 1: Bubble_Sort

```
{for i=1 up to n-1
  do for j=n downto  i+1
       do  if A[j-1] > A[j]
              then {temp= A[j-1] ;
                          A[j-1]= A[j] ;
                         A[j]=temp ;
                      }
               end if
        done
  done
}
```

i=1: j varies from n to 2, hence (n-1) processes

i=2: j varies from n to 3, hence (n-2) processes

………………………………

i=n-1: j varies from n to n, hence only one process

Overall, the number of processes is:

(n-1)+(n-2)+……..+1 = S

Let us write S in a different manner:

S = 1+ 2 +………+ (n-1)

We then obtain: 2S= n + n + …………… +n = n(n-1) $\Rightarrow$ S = n(n-1) /2

For each of these n(n-1)/2 processes, one condition and 3 assignments must be executed. This execution is independent of n and therefore constant, requiring a certain CPU time *d* that depends on the platform on which the algorithm is executed.

Therefore for the algorithm Bubble_Sort, we have:

$$T(n)=\frac{n(n-1)}{2}d$$

Example 2.4 :

Factorial of a number

```
Factorial(n)
{if n ≤ 1
  then {Fact=1 ; return;}
  else Fact =N * Factorial(n-1) ;
  end if
}
```

If n ≤ 1, the process is constant: d

Else, there is a constant process (c) that concerns multiplication, in addition to a process corresponding to the recursive call Factorial(n-1): T(n-1), which is:

$$T(n)=\begin{cases} d & if\ n\le 1 \\ c+T(n-1) & else \end{cases}$$

Consider $n > 1$. We then have: $T(n-1)=c+T(n-2) \Rightarrow T(n)=2c+T(n-2)$
By recurrence, we have: $T(n) = i * c + T(n-i)$
Consider $n-i=1$, hence $i=n-1$. We then have: $T(n)=(n-1) * c + T(1) = (n-1)c + d$
It can be easily verified that $T(n)$ is $O(n)$. Indeed, we should then have: $(n-1)c + d \leq k * n \;\; \forall n \geq n_0 ; k > 0 ; n_0 \geq 0$, which means $n(k-c) \geq d-c$
Given that multiplication requires more time than assignment, we have $d < c$. With $k > c$, we get:
$n \geq (d-c) / (k-c) ; k > c$; given $k=c+1 \Rightarrow n \geq d-c \Rightarrow T(n) \leq (c+1) * n \;\; \forall n \geq 0$

Example 2.5: Let us consider the problem of the Tower of Hanoi. There are three rods A, B and C. There are N disks on rod A, arranged in descending order of size from bottom to top. The objective is to move the N disks from A to C using B (size order must be preserved on each rod). Find the computational complexity of this problem.
As a first step, let us find the algorithm. For this purpose, the following reasoning is used:
Let us assume we know how to correctly move (N-1) disks from one rod to another, using the third rod. The solution would then be the following:

- Correctly move (N-1) disks from A to B using C;
- Move the remaining disk (of the current larger size) from A to C;
- Move the (N-1) disks from B to C, using A (we assume we know how to correctly move the N-1 disks from one rod to another);

We then obtain the following recursive algorithm:

```
Hanoi(A, B, C, N)    // The order of arguments is important
{
  if N = 0
  then return;
  end if
  Hanoi(A, C, B, N-1); // The order of parameters is important
  Move the remaining disk from A to C;
  Hanoi(B, A, C, N-1); // The order of parameters is important
}
```

The computational complexity T(N) is then equal to:
- d1 if N=0
- $T(N-1) + d + T(N-1)$ else; d is the CPU time required for moving the remaining disk from one rod to another.

We then have:
$T(N) = 2*T(N-1) + d$ for $N \neq 0$
By recurrence, we get: $T(N-1) = 2*T(N-2)+d \Rightarrow T(N) = 4*T(N-2)+3d$
$T(N-2) = 2*T(N-3)+d \Rightarrow T(N) = 8*T(N-3)+7d$
$T(N-3) = 2*T(N-4)+d \Rightarrow T(N) = 16*T(N-4)+15d$
…………………………………..
$T(N-i) = 2*T(N-i-1)+d \Rightarrow T(N)=2^{i+1}*T(N-i-1)+(2^{i+1}-1)d$
With $i = N-2$, we get: $T(N) = 2^{N-1}*T(1)+(2^{N-1} -1)d = 2^{N-1}d + 2^{N-1}d - d = 2^N d - d$
Since $d > 0$, we have $T(N) \leq d*2^N \;\; \forall N \geq 1 \Rightarrow T(N)$ is $O(2^N)$

Example 2.6: Find the computational complexity of the following algorithm:

```
Begin
        S=0 ;
        for i=1 up to N by step of 2
        do S = S + i !   // well read Factorial of i
        done
End
```

Solution:

Let us recall that the execution time of Factorial of N is equal to (N-1)d+d1

i=1 : d1+d2 (exec time of 1 ! + exec time of the sum)

i=3 : 2d+d1+d2 (exec time of 3 ! + exec time of the sum)

i=5 : 4d+d1+d2 (exec time of 5 ! + exec time of the sum)

……………………………………..

i=N : (N-1)d+d1+d2 ; N is odd

$\Rightarrow$ T(N) = (d1+d2)+(2d+d1+d2)+(4d+d1+d2) + …. (N-1)d+d1+d2 = (d1+d2)(1+(N-1)/2)+d(2+4+ …. + (N-1))

S1= 2 + 4 + … +(N-1)

S1=(N-1)+(N-3)+ … + 2

$\Rightarrow$ 2S1=(N+1)(1+(N-3)/2) $\Rightarrow$ S1 = (N+1)(N-1) /4 $\Rightarrow$ T(N) = (d1+d2)((N+1)/2) + d(N+1)(N-1) /4

$= N^2$ d/4 + N(d1+d2)/2 + (d1+d2)/2 – d/4

Let us show that T(N) is O(N^2), which leads to:

$$T(N) \le C * N^2 \ \forall\ N \ge N_0 \ ;\ C > 0;\ N_0 \ge 0$$

$\Rightarrow N^2(C - d/4) – N(d1+d2)/2 – (d1+d2)/2 + d/4 \ge 0$

$\Delta = (d1+d2)^2/4 - 4(C-d/4)(d/4 – (d1+d2)/2)$

It is worth noting that if C=d/4, it would not be possible to have: d/4 $\ge$ N(d1+d2)/2 + (d1+d2)/2 (the value of N can be large, while C and d are constant).

Let us then consider C=5d/4.

We should note that constants d, d1 and d2 are used for multiplication, assignment and addition. If d=2(d1+d2), then we get: $\Delta = (d1+d2)^2/4 - 4d((d1+d2)/2 – (d1+d2)/2) = (d1+d2)^2/4$

It can then be verified that the roots are: N1= 0; N2 = (d1+d2)/(2*2(d1+d2))

Between these two roots, C*N^2 – T(N) is negative. Since the objective is to have C*N^2 – T(N) $\ge$ 0, the two parts should be considered outside the [N1, N2] interval. But if we consider N_0=N1, we would not have C*N^2 – T(N) $\ge 0\ \forall\ N \ge N_0$ since in [N1, N2], we have C*N^2 – T(N) $\le$ 0. We should therefore have $N_0=N2=\lceil 1/4 \rceil = 1$. Since C=5d/4 and f(N)=$N^2$, we get T(N) $\le$ 5d/4 * N^2 $\forall$ N $\ge$ 1 and therefore T(N) is O(N^2).

2.2 Non-deterministic algorithm and class NP

Assume a function *choice()* is available, and find the answer, within a polynomial time, expressed as TRUE or FALSE to a **decision** problem.

Example 2.7:
Given N numbers, find the largest among them.

ALGORITHM 2 : MAX_NON_DETERMINISTIC (N)

```
{max=choice({$a_1$, $a_2$, …., $a_N$}) ;
 for i=1 up to N
 do    if $a_i$ > max
         then {Answer = ″FALSE″ ; exit ; }
         end if
  done
 Answer = ″TRUE″ ;
}
```

Note that this algorithm is:

- Non-deterministic because we do not know how the function choice() has defined max
- Polynomial time (its computational complexity is O(N))

The deterministic algorithm is the following:

ALGORITHM 3: MAX_DETERMINISTIC (N)

```
{max=$a_1$ ;
  For i=2 up to N
  Do   If $a_i$ > max
         Then max=$a_i$ ;
         End if
  Done
 }
```

This algorithm is also polynomial time, its computational complexity being O(N). Note that in general, deterministic and non-deterministic algorithms associated with the same problem have not necessarily the same computational complexity. It is, for example, the case of the following problem:

Example 2.8 (Satisfiability problem):
Consider $X=\{x_1, x_2, \ldots, x_n\}$ a set of Boolean variables
$E= C_1 . C_2 . \ldots\ldots . C_m$ a logical expression;
The dot represents the logical operator AND; C_i is a clause where $C_i = u_1 + u_2 + \ldots + u_k$; k=1, 2, …., n ; u_i is one of the variables either complemented or not. The + represents the logical operator OR.

Example 2.9:

$E=(x_1 + x_2 + x_3') . (x_1' + x_2) . x_3$; $x_i' = NOT(x_i)$

Let us consider the decision problem: Is there an assignment of variables x_k; k=1, 2, 3 to 0 or 1 such that E=1?

ALGORITHM 4: SAT_NON_DETERMINISTIC (n, E)

```
{for i=1 up to n
  do $x_i$=choice({0, 1}) ;
  done
  if E($x_1$, $x_2$, ….., $x_n$) = 1
  then answer = ″TRUE″;
  else answer = ″FALSE″;
  end if
}
```

This (non-deterministic) algorithm is polynomial time (of complexity O(n)). On the other hand, the deterministic algorithm is not (complexity equal to $O(2^n)$).

Hence, for a given problem, it is possible that deterministic and non-deterministic algorithms have the same computational complexity (case of the determination of the maximal number), or not (case of the satisfiability problem). This leads us to a first classification of problems.

Definition:

A problem Π belongs to the class NP if it can be executed within a polynomial time by a NON-DETERMINISTIC algorithm.

From this definition, the 2 problems described above (maximum of N numbers and satisfiability) both belong to the class NP.

The problem of the maximum of N numbers can be solved in a polynomial time by a **deterministic** algorithm. It therefore belongs to the class P (see further details about the deterministic Turing machine with freely downloading a part of our book from https://media.wiley.com/product_data/excerpt/76/17863059/1786305976-29.pdf). As this problem belongs to class P, unlike the satisfiability problem, class P is included in class NP (note that NP is an acronym of **N**on-deterministic **P**olynomial, and not of Non Polynomial. It would otherwise be a contradiction, since the problem of the maximum of N numbers is polynomial and belongs to class NP).

The class NP includes polynomial, NP-complete, NP-hard and NP-intermediate problems (it also includes some Co-NP problems -please see: https://media.wiley.com/product_data/excerpt/76/17863059/1786305976-29.pdf)

2.2.1 Polynomial problems

Definition:

Class P is defined as a set of languages L each of which has a corresponding deterministic polynomial time program recognizing the considered language: P = {L : there is a deterministic polynomial time program for which $L = L_M$} // L_M is a language accepted by a Turing machine – for further details, please see: https://media.wiley.com/product_data/excerpt/76/17863059/1786305976-29.pdf

2.2.2 NP-complete problems

Definition:

A language L_1 is NP-complete if:

- $L_1 \in NP$
- $L_2 \ \alpha \ L_1 \quad \forall \ L_2 \in NP$ (α is a polynomial transformation – for further details, please see: https://media.wiley.com/product_data/excerpt/76/17863059/1786305976-29.pdf)

Theorem:

If: - *P_1 is NP-complete*

- *$P_1 \ \alpha \ P_2$*
- *$P_2 \in NP$*

Then: P_2 is NP-complete

Proof:

P_1 is NP-complete $\Rightarrow P_1 \in NP \wedge \ P_3 \ \alpha \ P_1 \ \ \forall \ P_3 \in NP$ (by definition)

According to the hypothesis, $P_1 \ \alpha \ P_2 \Rightarrow P_3 \ \alpha \ P_2$ (α is transitive) and $P_2 \in NP \Rightarrow P_2$ is NP-complete

Conjecture:

For NP-complete problems, there is no algorithm giving the exact solution within polynomial time. This is nevertheless just a conjecture. To this day, we conjecture that there is no exact and polynomial time algorithm for NP-complete problems.

2.2.3 NP-Hard problems

Let us recall that polynomial time problems can be executed by a deterministic Turing machine. Those of class NP (and of class P) are executed by a non-deterministic Turing machine. Before defining the NP-hard problems, let us now consider the Oracle Turing Machine involving a module (oracle) that executes a program P_1.

Definition:

A polynomial Turing reduction of problem P_2 to problem P_1 (denoted $P_2 \ \alpha_T \ P_1$) is a transformation that enables the polynomial-time resolution of P_2 through the algorithm of P_1 if the resolution of P_1 is assumed of computational complexity O(1).

In other terms, this involves:

- Stage 1: Transforming the data of P_2 into data of P_1
- Stage 2: Solving P_1 by means of a module (oracle)
- Stage 3: Transforming the results of P_1 into results of P_2

Assuming the complexity of stage 2 is O(1), if stages 1 and 3 have polynomial time complexity, then it can be said that $P_2 \ \alpha_T \ P_1$.

Definition:

A problem P_2 is NP-hard (NP-difficult) if there is an NP-complete problem P_1 such that $P_1 \ \alpha_T \ P_2$.

According to the conjecture $P \neq NP$, therefore an NP-hard problem cannot reach the exact solution within polynomial time (NP-hard problems are the most difficult in the class NP).

NOTE: α and α_T should not be confused (further details about α and α_T are found in: https://media.wiley.com/product_data/excerpt/76/17863059/1786305976-29.pdf).

2.2.4 NP-Intermediate Problems

Certain problems of class NP are neither polynomial, nor NP-complete, nor NP-hard (see Figure 2.1). Informally speaking, they are more complicated than the polynomial problems, and less complicated than the NP-complete problems.

In order to prove that a problem P_1 is NP-intermediate, one must prove that:

- P_1 belongs to the class NP (it can be solved by a Nondeterministic Polynomial algorithm)
- P_1 cannot be solved in a deterministic manner within a polynomial time (ie $P_1 \notin$ class P)
- There is no polynomial transformation of an arbitrary NP-complete problem P_2 into problem P_1 (meaning that P_1 is not NP-complete)

Example 2.10:
Given 2 graphs G=(V, E) and G'=(V', E'), are G and G' isomorphic? Namely, is there a one-to-one function $f : V \rightarrow V'$ such that: $(u, v) \in E$ if and only if $(f(u), f(v)) \in E'$?
This problem is neither proved P nor NP-complete [2]. As it belongs to NP (it can be solved by a Non-deterministic Polynomial algorithm), it is then NP-intermediate one.

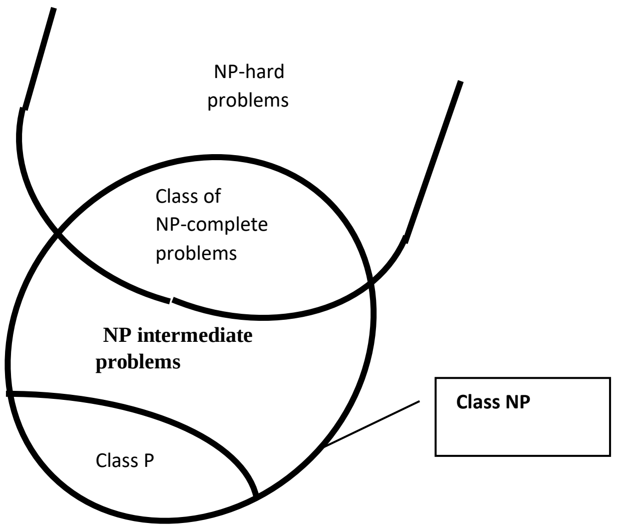


Figure 2.1: The class NP.

It is worth noting that polynomial-time problems are exactly solved while intractable ones need heuristic- or metaheuristic- based methods to reach a solution. Several conventional meta-heuristics are used but note that obtaining a near-optimal solution (or an optimal solution for some problem instances) does not depend on the application of such or such method, but rather on other criteria (initializations, moving one solution to another, exploration of the solution space, etc.) [1,3].

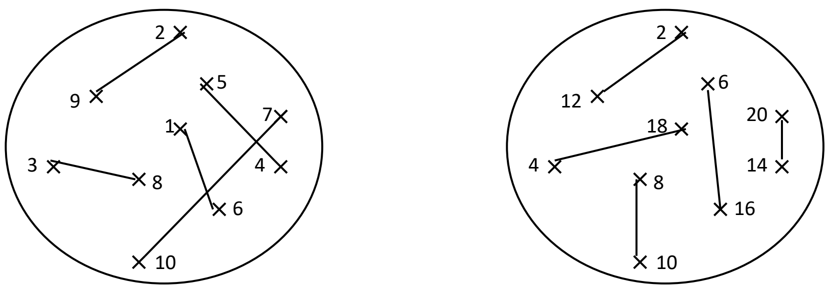
3 Formal Proof That P ≠ NP

As NP-intermediate and NP-complete problems belong to NP (some of Co-NP and NP-hard ones also belong to NP), it is sufficient to prove that one of these problems is not polynomial time in order to formally claim that $P \neq NP$. To do this, we consider the graph-isomorphism problem which belongs to NP. This problem is defined as follows:
Given 2 graphs G=(V, E) and G'=(V', E'), are G and G' isomorphic? Namely, is there a one-to-one function $f : V \rightarrow V'$ such that: $(u, v) \in E$ if and only if $(f(u), f(v)) \in E'$?

Let us consider the 2 following cases:

3.1 Function f is known

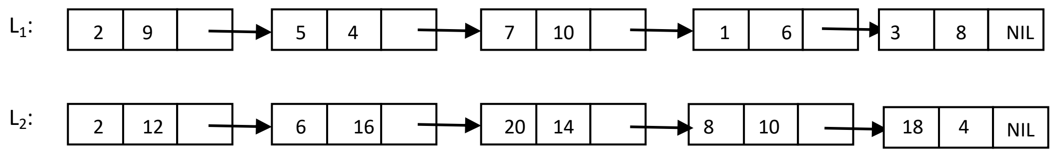
It is easy to assert that the 2 graphs depicted in Fig. 3.1 are isomorphic according to function f: val(v) = 2 * val (u); $u \in V_1$, $v \in V_2$; v=f(u).



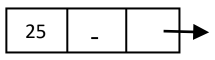
(a) (b)

Figure 3.1 Graph instances (a) $G_1 = (V_1, E_1)$ (b) $G_2 = (V_2, E_2)$. G_1 and G_2 are isomorphic according to function f: val(v) = 2 * val (u); $u \in V_1$, $v \in V_2$; v=f(u).

G_1 and G_2 can be represented by the 2 following lists:



It is obvious that if node 25 $\in V_1$ and is not connected to any other node, L_1 would include this structure member:



A simple algorithm to check if G_1 and G_2 are isomorphic is the following:

ALGORITHM 5: DETERM_POLYN_GRAPH_ISOM_ALG (f, G_1, G_2)

```
{
  if($|E_1| \neq |E_2|$)
  then {print(" $G_1$ and $G_2$ are not isomorphic");
            exit();  // The algorithm terminates
          }
 end if
 W = $\varnothing$;   // W is the set of the explored edges $(v_1, v_2) \in E_2$ such that: $(v_1, v_2) = (f(u_1), f(u_2))$ OR
                                                  $(v_2, v_1) = (f(u_1), f(u_2))$;   $(u_1, u_2) \in E_1$
  for each $(u_1, u_2) \in E_1$
  do {found = 0;
        for each $(v_1, v_2) \in E_2 - W$
        do if $(v_1, v_2) = (f(u_1), f(u_2))$ OR $(v_2, v_1) = (f(u_1), f(u_2))$
             then {found = 1;
                       W = W $\cup$ {$(v_1, v_2)$};
                       break;  // exit the inner loop
                      }
           end if
```

```
      done
      if (found = 0)
      then {print(" $G_1$ and $G_2$ are not isomorphic");
              exit();  // The algorithm terminates
            }
       end if
      }
    Done
    print(" $G_1$ and $G_2$ are isomorphic");
 }
```

- For the 1st edge in L_1, there are, in the worst case, $|E_2|$ searches. The worst case happens when "found" remains 0 with the exploration of $|E_2|$-1 members of L_2;
- For the 2nd edge in L_1, there are, in the worst case, $|E_2|$-1 searches (thanks to W, the number of edges to be explored in L_2 is reduced);
- etc.
- For the last edge in E_1, there is a single search.

The total number of searches is then:

$S = M + (M-1) + \dots + 1; \quad M = |E_2|$

$= 1 + 2 + \dots + M$ (rewriting S in the reverse order)

$\Rightarrow 2S = (M+1) + (M+1) + \dots + (M+1) = M^* (M+1) \Rightarrow S = M * (M+1) / 2$

Thus, the time complexity of the previous algorithm is $O(M^2)$, which is polynomial time. In other words, if the function *f* is known, the "Graph-isomorphism" problem belongs to the class P (not to the intermediate one).

3.2 A formal proof that P ≠ NP (Function f is unknown)

Unfortunately, according to the formal definition of the Graph-isomorphism problem, the function f is unknown ("*Is there a one-to-one function f such that" etc.*).

It is obvious that we only need to prove that the Graph-isomorphism problem belongs to NP, but it is outside class P. According to the definition we previously gave, a problem belongs to NP if it can be solved by a Non-deterministic Polynomial algorithm. We give hereafter such an algorithm:

```
ALGORITHM 6: NON_DETERM_POLYN_GRAPH_ISOM_ALG ($G_1$, $G_2$)
{
  // $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are 2 graph instances
  if ($|E_1| \neq |E_2|$)
  then {Answer = "FALSE";
          exit();  // the algorithm terminates
        }
  endif
  f = choice (F);  // selecting function f from the set F
  // Write here the algorithm corresponding to the case in which function f is known, beginning with the
instruction  W = ∅ till the end of that algorithm. However, replace the statements print(" $G_1$ and $G_2$ are
not isomorphic"); and print(" $G_1$ and $G_2$ are isomorphic"); with Answer="FALSE"; and Answer =
"TRUE", respectively.
// Note that ALGORITHM 5 is deterministic while the present one is non-deterministic.
}
```

- This algorithm is polynomial time since its computational complexity is $O(|E_2|^2)$;
- It is Non-deterministic because function f is selected in a manner that is beyond our grasp

Thus, according to the definition, **the Graph-isomorphism problem belongs to NP**.

Let us now prove that it does not belong to class P. In other words, let us prove that the **deterministic** associated algorithm is **not polynomial time.**

Let us assume that $|E_1| = |E_2|$ (otherwise, G_1 and G_2 could not be isomorphic).

Notation: Let $e_i \, \underline{f_k} \, e_j'$ means ($v_{j1} = f_k(u_{i1})$ AND $v_{j2} = f_k(u_{i2})$) OR ($v_{j2} = f_k(u_{i1})$ AND $v_{j1} = f_k(u_{i2})$);

$e_i = (u_{i1} , u_{i2}) \in E_1$ and $e_j' = (v_{j1} , v_{j2}) \in E_2$

An obvious way to retrieve function f_k that prospectively makes G_1 and G_2 isomorphic is the following:

Let us assume that $e_1 \, \underline{f_k} \, e_1'$ is O(1), thus f_k is determined in polynomial time. In order to make G_1 and G_2 isomorphic according to function f_k, we need:

$e_2 \, \underline{f_k} \, e_2'$ AND $e_3 \, \underline{f_k} \, e_3'$ ... AND $e_M \, \underline{f_k} \, e_M'$ $(M = |E_2|)$ OR

$e_2 \, \underline{f_k} \, e_3'$ AND $e_3 \, \underline{f_k} \, e_2'$ ... AND $e_M \, \underline{f_k} \, e_M'$ OR

…

OR

$e_2 \, \underline{f_k} \, e_M'$ AND $e_3 \, \underline{f_k} \, e_{M-1}'$ ... AND $e_M \, \underline{f_k} \, e_2'$

But G_1 and G_2 could be non isomorphic according to f_k. Thus, we have to determine ANOTHER function f_k, for example, the one that is determined from e_1 and e_2': $e_1 \, \underline{f_k} \, e_2'$. G_1 and G_2 will be prospectively isomorphic according to the new function f_k if:

$e_2 \, \underline{f_k} \, e_1'$ AND $e_3 \, \underline{f_k} \, e_3'$ ... AND $e_M \, \underline{f_k} \, e_M'$ $(M = |E_2|)$ OR

$e_2 \, \underline{f_k} \, e_3'$ AND $e_3 \, \underline{f_k} \, e_1'$ ... AND $e_M \, \underline{f_k} \, e_M'$ OR

…

OR

$e_2 \, \underline{f_k} \, e_M'$ AND $e_3 \, \underline{f_k} \, e_{M-1}'$ ... AND $e_M \, \underline{f_k} \, e_1'$

Again, G_1 and G_2 could be non isomorphic according to this new function f_k. The computational complexity which is determined from the worst case consists to consider ALL the possibilities, namely:

$e_1 \, \underline{f_k} \, e_1'$ AND $e_2 \, \underline{f_k} \, e_2'$ ... AND $e_M \, \underline{f_k} \, e_M'$ OR

$e_1 \, \underline{f_k} \, e_2'$ AND $e_2 \, \underline{f_k} \, e_1'$ ... AND $e_M \, \underline{f_k} \, e_M'$ OR

…

OR

$e_1 \, \underline{f_k} \, e_M'$ AND $e_2 \, \underline{f_k} \, e_{M-1}'$ ... AND $e_M \, \underline{f_k} \, e_1'$

The number of attempts to determine the function f_k that prospectively makes G_1 and G_2 isomorphic is then: M ! (please read Factorial of M).

For each of these M ! attempts, checking whether G_1 and G_2 are isomorphic according to the interested function f_k is O(M) (not O(M^2) because in this case, for each edge in E_1 a SINGLE search is done in E_2).

Thus, the time complexity of this problem is O(M ! * M), which is **not polynomial**.

In our proof, even if the function f_k could be determined in polynomial time, checking whether G_1 and G_2 are isomorphic (according to f_k) or not is NOT polynomial time. In reality, given 2 any graph instances, it is not obvious to define the function f (please see Figure 3.2). Indeed, it should be retrieved from an INFINITE set F:

F= {val(v)=val(u)-1, val(v)=$\log_5$(4+val(u)), val(v)=2*val(u), val(v) = sqrt(97+val(u))*3, ……};

In this case, one should pick, successively, a new function f from the INFINITE set F, then check among the M ! (factorial of M) possibilities whether G_1 and G_2 are isomorphic. In other words, the time complexity of the interested problem could not be polynomial (whether f is determined in polynomial time or not).

Thus, P ≠ NP.

1
2
15984
29017

Figure 3.2: These 2 Graphs are isomorphic, according to the function f I myself defined. Could someone retrieve this function ?

4 Conclusion

In this paper, we recalled fundamental notions of computational complexity (further details that help better understand the notions given in Section 2 can be freely downloaded from: https://media.wiley.com/product_data/excerpt/76/17863059/1786305976-29.pdf) and showed that the Graph-isomorphism problem belongs to class NP. We then demonstrated that this problem does not belong to class P. Thus, we formally claim that P ≠ NP.

References

1. Mahdoum "CAD of Circuits and Integrated Systems" Wiley (1st ed.), October 2020, Hoboken, USA.
2. M.R. Garey, D.S. Johnson "Computers and Intractability: a Guide to the Theory of NP-Completeness" Freeman (1st ed.), 1979, San Fransisco, USA.
3. Mahdoum "Book review: Representations for genetic and evolutionary algorithms, written by F. Rothlauf" J. The Computer 49, 5 (September 2006).