TWO PROBLEMS PROVING P ≠ NP

Valdir Monteiro dos Santos Godoi
valdir.msgodoi@gmail.com

ABSTRACT. Is proved that P ≠ NP, showing 2 problems that are executed in constant complexity time *O*(1) in a nondeterministic algorithm, but in exponential complexity time related to the length of the input (input size) in a deterministic algorithm. These algorithms are essentially simple, so they can not have a significant reduction in its complexity, what could cause the proofs shown here to become invalid.

Mathematical subject classification: 68Q10, 68Q15, 68Q17.

Keywords: P x NP, P versus NP, P ≠ NP, P Class, NP Class, deterministic algorithm, nondeterministic algorithm.

INTRODUCTION

P ≠ NP, like many computer scientists think.

P x NP is the most important problem of the Computer Science that is not yet solved. It was created in 1971, based on the work of Cook[2], although Karp[14], Edmonds[6,7,8], Hartmanis and Stearns[10], Cobbam[1], von Neumann[15] and others also participated in the elaboration of this question. For a most exact history of this problem, it is suggested to look at [9], [11] and [16].

The P x NP problem intend to determine whether every language L accepted by some nondeterministic algorithm in polynomial time ($L \in NP$) is also accepted by some deterministic algorithm in polynomial time ($L \in P$) [3].

If P = NP, then every problem solved in polynomial time (related to the input size) by a nondeterministic algorithm, it will also be solved in polynomial time (related to the input size) by a deterministic algorithm.

If P ≠ NP, there will be at least one problem belonging to NP which will not have a deterministic algorithm with polynomial complexity related to the input size for its solution.

The NP class owns a great number of important problems, which the solution in polynomial time it's only possible when in approached ways, or in particular cases (even if there are infinites particular cases), like the problems of satisfiability (SAT), subset sum, knapsack, travelling salesman, quadratic diophantine equations, quadratic congruences, etc.

As until today there is not a "perfect and fast" algorithm (in polynomial time related to the input size) to solve these problems in NP, so the opinion of the specialists is that P ≠ NP [4].

In this article, there will be shown 2 simple problems, which the solution is in polynomial time in a nondeterministic algorithm, but in non-polynomial time in a deterministic algorithm, proving that P ≠ NP.

Although it is not fundamental, we will adopt the perspective that a nondeterministic Turing Machine working in polynomial time has the ability of predefine a exponential number of possible solutions and verify each one in polynomial time, "in parallel" [12]. We will adopt this notion of parallelism, according to other authors too, for example, Diverio and Menezes [5]. In particular, we will be despising the idea that the Turing Machines "guesses" a correct solution, or always hit on the "first attempt", like defends Papadimitriou *et al* in [4].

PROBLEM 1

Our first problem (p_1), it's a searching problem, with the special feature that, instead find an element in a list of numbers input by an user (what will increase our input size), will search directly on the computer's memory, in the state it be.

p_1 = "Read 2 memory addresses (pointers) n_1 and n_2, and a character x verifies if, from the memory position n_1 to n_2, the x element exists."

Suppose that each pointer has a maximum of c characters, the input size will have a maximum of LEN = 2c + 3 (pointers in the hexadecimal base) and its complexity in a deterministic algorithm will be of the order $O(n_2)$ = $O(16^{LEN/2})$, i.e., a complexity of exponential order related to the input size. As we can not guarantee that the memory is ordered, that will able us to execute a binary search, or that it contains only spaces, zeros, etc., it is not possible to do any reduction important in the algorithm, meaning that the problem do not belongs to P.

The deterministic algorithm is as following:

```
void p₁D (char *n₁, *n₂, x)
{ char *i;
  for (i = n₁; i ≤ n₂; i++)
      if (*i == x)
         {printf("Yes.\n");
          return;
         }
  printf("No.\n");
  return;
}
```

The corresponding nondeterministic algorithm, following the style of Nivio Ziviani [17], which based himself in E. Horowitz e S. Sahni [13], is as following:

```
void p₁ND (char *n₁, *n₂, x)
{ char *i;
  i = CHOICE(n₁ .. n₂);
  if (*i == x)
      SUCCESS;
  else
      FAILURE;
}
```

This algorithm is of complexity *O*(1), then $p_1 \in NP$, and obviously uses the unique power of a nondeterministic Turing Machine: the capacity of being called simultaneously for various times, until there is success. In the case of x is not found in the memory positions n_1 to n_2, the operation will not enter in infinite loop and will stop.

As $p_1 \in NP$, but $p_1 \notin P$, so $P \neq NP$.

PROBLEM 2

Our second problem (p_2), although at first glance does not seems to have any practice utility, it clearly demonstrates that $P \neq NP$.

It is about the generation of random numbers y until the generated number be equal to a given parameter x. It is read as input two integer numbers n and x, with $1 \leq x \leq n$ and $1 \leq y \leq n$.

In the deterministic version (p_2D), the probability of obtaining the valor of x it is equal to 1/n, assuming uniform distribution in the generation of numbers, and, on average, you should expect n repetitions until the generated number y be equal to x, a number of repetitions that is of exponential order related to the input size. Then, $p_2 \notin P$.

```
void p2D (int n, x)
{int y;
 srand(time(NULL));
 do
    y = rand()%n + 1;
 while (y != x);
 printf("Success.\n");
 return;
}
```

On the nondeterministic version (p_2ND), the time of execution is of order $O(1)$. Then $p_2 \in NP$.

As $p_2 \in NP$, but $p_2 \notin P$, so $P \neq NP$.

```
void p2ND (int n, x)
{int y;
 y = CHOICE(1 .. n);
 if (y == x)
    SUCCESS;
 else
    FAILURE;
}
```

In this second algorithm, the function *CHOICE* takes place of the generator function of random numbers, what is totally compatible with the feature of a nondeterministic algorithm. This same function also could have taken place of the expression that generates these random numbers in the deterministic version, and be defined, for example, this way:

```
int CHOICE(int y1, y2)
{int y;
 y = y1 + rand()%(y2 – y1 + 1);
 return y;
}
```

CONCLUSION

We shown two simple problems and your respective solution algorithms. Both are solved with polynomial complexity related to the input size, more exactly, with constant complexity $O(1)$, by a nondeterministic algorithm, characterizing that both belongs to NP.

The solutions by the respective deterministic algorithm requires light modifications, but the “time” of execution raises in exponential form, characterizing that these problems does not belong to P, and that P ≠ NP.

The nondeterministic algorithms has the advantage that for each call, it chooses a alternative and tests it, and when the solution it is the right one, the processing ends with success. In the other hand, if there is no correct solution, some of the set of possibilities chosen, the processing ends with no success, and, in theory, in finite time, without infinite loop, demanding the same time as if there were a valid solution.

If a problem assumes, for example, 2^n alternatives for a correct solution, like the SAT, a nondeterministic algorithm would have the capacity of testing “simultaneously” the 2^n alternatives, while a deterministic algorithm, following the same strategy, should test each alternative, successively (and not simultaneously), and yet would have to control the moment of the stop. This is a fundamental difference between these two types of algorithms, and for that, it will be extremely improbable that P = NP.

Of course, a algorithm of exponential order complexity could eventually be optimized to a algorithm of polynomial order complexity, for example, the sum of a arithmetic progression. The sum $S = \sum^n_{\kappa=1} a_k = \sum^n_{\kappa=1} a_1 + (k-1)r$ it is of exponential complexity related to the length of the input (a_1, $r_,$ n), while $S = (a_1 + a_n) / 2 = a_1 + (n-1)r/2$ it is of constant complexity $O(1)$, although both brings you to the same solution.

The algorithms p_1D and p_2D presented, however, are already “simple enough” for that they can have a exponential reduction in its complexity, and would have invalidated our proof that P ≠ NP.

p_1D, for example, if not used with the instruction *for*, a similar structure of the type *while*, *repeat* or *until*, or even the lousy algorithm that contains n_2 or more successives conditions *if*, all resulting in exponential complexity, should use a instruction “supposedly” $O(1)$, like *TEST*($*n_1$, $*n_2$, x), but that will hide in itself the real exponential complexity of the problem.

REFERENCES

[1] A. Cobham, *The intrinsic computational difficulty of functions*, in Proceedings of the 1964 International Congress for Logic, Methodology, and Philosophy of Science, Y. Bar-Hille, ed., Elsevier/North-Holland, Amsterdam (1964), 24–30.

[2] S. Cook, *The complexity of theorem-proving procedures*, in Conference Record of Third Annual ACM Symposium on Theory of Computing, ACM, New York (1971), 151–158.

[3] Stephen Cook, *The P versus NP Problem*, available in http://www.claymath.org/millennium/P_vs_NP/pvsnp.pdf, accessed in 02/22/2011.

[4] S. Dasgupta, C. Papadimitriou and U. Vazirani, *Algoritmos*, McGraw-Hill Interamericana do Brasil Ltda., São Paulo (2009), 244.

[5] T.A. Diverio and P.M Menezes, *Teoria da Computação – Máquinas Universais e Computabilidade*, Sagra Luzzatto, Porto Alegre (2004), 122-124.

[6] J. Edmonds, *Maximum matchings and a polyhedron with 0,1-vertices*, Journal of Research at the National Bureau of Standards, Section B, 69 (1965), 125-130.

[7] J. Edmonds, *Minimum partition of a matroid into independent subsets*, J. Res. Nat. Bur. Standards, Section B, 69 (1965), 67–72.

[8] J. Edmonds, *Paths, trees and flowers*, Canadian Journal of Mathematics, 17 (1965), 449-467.

[9] L. Fortnow and S. Homer, *A Short History of Computational Complexity*, available in http://people.cs.uchicago.edu/~fortnow/beatcs/column80.pdf, accessed in 02/22/2011.

[10] J. Hartmanis and R.E. Stearns, *On the computational complexity of algorithms*, Transactions of the AMS 117 (1965), 285-306.

[11] R. Herken, ed., *The Universal Turing Machine – A Half-Century Survey*, Verlag Kammerer & Unverzagt, Hamburg-Berlin (1988).

[12] J.E. Hopcraft, J.D. Ullman and R. Motwani, *Introdução à Teoria dos Autômatos, Linguagens e Computação*, Elsevier e Campus, Rio de Janeiro (2003), 452.

[13] E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms*, Computer Science Press (1984), 501-510.

[14] R.M. Karp, *Reducibility among combinatorial problems*, in Complexity of Computer Computations, R.E. Miller and J.W. Thatcher, eds., Plenum Press, New York (1972), 85–103.

[15] J. von Neumann, *A certain zero-sum two-person game equivalent to the optimal assignment problem*, in Contributions to the Theory of Games II, H.W. Kahn and A.W. Tucker, eds., Princeton Univ. Press, Princeton, NJ (1953), 5–12.

[16] M. Sipser, *The History and Status of the P versus NP question*, Proceedings of the 24th Annual ACM Symposium on the Theory of Computing (1992), 603-619, available in http://www.win.tue.nl/~gwoegi/P-versus-NP/sipser.pdf, accessed in 02/22/2011.

[17] N. Ziviani, *Projeto de Algoritmos com Implementações em Java e C++*, Thomson Learning, São Paulo, Brasil, (2007), 381-383, 388, 551.