Note for the P Versus NP Problem (II)

Frank Vega

June 3, 2024

Article

Note for the P versus NP Problem (II)

Frank Vega [1]

[1] GROUPS PLUS TOURS INC., 9611 Fontainebleau Blvd, Miami, FL, 33172, USA; vega.frank@gmail.com

Abstract: P versus NP is considered as one of the most fundamental open problems in computer science. This consists in knowing the answer of the following question: Is P equal to NP? It was essentially mentioned in 1955 from a letter written by John Nash to the United States National Security Agency. However, a precise statement of the P versus NP problem was introduced independently by Stephen Cook and Leonid Levin. Since that date, all efforts to find a proof for this problem have failed. Another major complexity class is NP-complete. It is well-known that P is equal to NP under the assumption of the existence of a polynomial time algorithm for some NP-complete. We show that the Monotone Weighted-2-satisfiability problem (MW2SAT) is NP-complete and P at the same time. Certainly, we make a polynomial time reduction from every undirected graph and positive integer k in the Vertex Cover problem to an instance of MW2SAT. In this way, we show that MW2SAT is also an NP-complete problem. Moreover, we create and implement a polynomial time algorithm which decides the instances of MW2SAT. Consequently, we prove that P = NP.

Keywords: Complexity classes; boolean formula; graph; completeness; polynomial time

MSC: 68Q15; 68Q17; 68Q25

1. Introduction

The field of computer science grapples with one of its most significant and challenging unsolved problems: the P versus NP question [1]. At its core, this question asks whether efficient verification of a solution translates to efficient solving of the problem itself. Here, "efficient" refers to the existence of an algorithm that tackles the task in polynomial time, meaning the time it takes scales proportionally to the size of the input data.

The class of problems solvable by such efficient algorithms is denoted by P, or "class P." Another class, NP (standing for "nondeterministic polynomial time"), encompasses problems where solutions themselves can be verified efficiently. This verification relies on a "certificate," a piece of succinct information that quickly confirms the solution's validity [1].

The P versus NP problem essentially asks if P and NP are equivalent. If, as many believe, P is strictly contained within NP (meaning P $\neq$ NP), then some problems in NP are inherently harder to solve than to verify. This distinction would have significant ramifications for various fields like cryptography and artificial intelligence [2].

Cracking the P versus NP problem is considered a pinnacle achievement in computer science. A solution would revolutionize our understanding of computation, potentially leading to groundbreaking algorithms that address some of humanity's most pressing challenges. The difficulty of this problem is reflected in its inclusion among the Millennium Prize Problems, a prestigious set of unsolved questions offering a million-dollar reward for a correct solution [1].

2. Materials and methods

NP-complete problems are a class of computational problems that are at the heart of many important and challenging problems in computer science. They are defined by the property that they can be quickly verified, but there is no known efficient algorithm to solve them. This means that finding a solution to an NP-complete problem can be extremely time-consuming, even for relatively small inputs. In computational complexity theory, a problem is considered NP-complete if it meets the following two criteria:

1. **Membership in NP**: A solution to an NP-complete problem can be verified in polynomial time. This means that there is an algorithm that can quickly check whether a proposed solution is correct [3].
2. **Reduction to NP-complete problems**: Any problem in NP can be reduced to an NP-complete problem in polynomial time. This means that any NP-problem can be transformed into an NP-complete problem by making a small number of changes [3].

If it were possible to find an efficient algorithm for solving any one NP-complete problem, then this algorithm could be used to solve all NP problems in polynomial time. This would have a profound impact on many fields, including cryptography, artificial intelligence, and operations research [2]. Here are some examples of NP-complete problems:

- **Boolean satisfiability problem (SAT)**: Given a Boolean formula, determine whether there is an assignment of truth values to the variables that makes the formula true [4].
- **Vertex Cover problem**: Given an undirected graph $G = (V, E)$ (V is the set of vertices and E is the set of edges) and positive integer k, determine whether there is a set $V' \subseteq V$ of at most k vertices such that for each edge $(u, v) \in E$ at least one of u and v belongs to V' [4].

These are just a few examples of the many NP-complete problems that have been studied and have a close relation with our current result. In addition, an edge cover of a graph G is a subset of its edges, denoted by E', such that every vertex in G belongs to at least one edge in E'. If G has edges with weight attributes the edge data are used as weight values else the weights are assumed to be 1.

Definition 1. ***Minimum Weighted Edge Cover***

INSTANCE: An undirected graph $G = (V, E)$ with weight attributes on the edges.

ANSWER: Find the smallest subset $E' \subseteq E$ such that it is an edge cover and the weighted sum of its edges is maximal?

REMARKS: This problem can be easily solved in polynomial time by finding the maximal matching of the graph [5]. This method is based on the "blossom" method for finding augmenting paths and the "primal-dual" method for finding a matching of maximum weight, both methods invented by Jack Edmonds [5].

In this work, we show there is an NP-complete problem that can be solved in polynomial time using the previous problem. Consequently, we prove that P is equal to NP.

3. Results

Formally, an instance of **Boolean satisfiability problem (SAT)** is a Boolean formula ϕ which is composed of:

1. Boolean variables: $x_1, x_2, \ldots, x_n$;
2. Boolean connectives: Any Boolean function with one or two inputs and one output, such as $\wedge$(AND), $\vee$(OR), $\rightharpoondown$(NOT), $\Rightarrow$(implication), $\Leftrightarrow$(if and only if);
3. and parentheses.

A truth assignment for a Boolean formula ϕ is a set of values for the variables in ϕ. A satisfying truth assignment is a truth assignment that causes ϕ to be evaluated as true. A Boolean formula with a satisfying truth assignment is satisfiable. The problem SAT asks whether a given Boolean formula is satisfiable [4].

We define a CNF Boolean formula using the following terms: A literal in a Boolean formula is an occurrence of a variable or its negation [3]. A Boolean formula is in conjunctive normal form, or CNF, if it is expressed as an AND of clauses, each of which is the OR of one or more literals [3]. A Boolean formula is in 2-conjunctive normal form or $2CNF$, if each clause has exactly two distinct literals [3].

For example, the Boolean formula:

$$(x_1 \vee \rightharpoondown x_1) \wedge (x_3 \vee x_2) \wedge (\rightharpoondown x_1 \vee \rightharpoondown x_3)$$

is in $2CNF$. The first of its three clauses is $(x_1 \vee \neg x_1)$, which contains the two literals x_1 and $\neg x_1$.

We define the following problem:

Definition 2. ***Monotone Weighted-2-satisfiability problem (MW2SAT)***

INSTANCE: An n-variable $2CNF$ formula with monotone clauses (meaning the variables are never negated) and a positive integer k.

QUESTION: Is there exists a satisfying truth assignment in which at most k of the variables are true?

REMARKS: The general case (i.e. whenever it is not monotone) is well-known that belongs to NP-complete [6].

The following is key Lemma.

Lemma 1. *$MW2SAT \in$ NP–complete.*

Proof. We can prove that the Vertex Cover problem can be entirely expressed within the MW2SAT framework. Here's a breakdown:

1. **Vertex Cover as MW2SAT**: We can transform any Vertex Cover instance into a MW2SAT problem. Imagine a graph with an associated Vertex Cover problem.
2. **Variable Assignment**: Create a Boolean variable for each vertex in the graph. A variable being "true" signifies the vertex is included in the potential solution (vertex cover).
3. **Clause Construction**: For every edge connecting vertices u and v, construct a MW2SAT clause $(u \vee v)$. This clause ensures at least one of the connected vertices (u or v) must be included (true) in the solution for the clause to be satisfied.
4. **Solution Equivalence**: A satisfying assignment for the generated MW2SAT formula directly corresponds to a solution for the original Vertex Cover problem. Each satisfied clause guarantees an edge is covered by at least one vertex in the assigned true variables (chosen vertex cover).
5. **Optimality**: The MW2SAT solution with at most k true variables translates to a Vertex Cover with at most k vertices, fulfilling the requirement of minimizing the number of vertices in the cover. Conversely, a k-vertex Vertex Cover solution can be mapped back to a MW2SAT solution with k true variables.
6. **Shared Complexity**: Since Vertex Cover is NP-complete, and we've shown it can be reduced to MW2SAT, it implies MW2SAT is also NP-complete. In simpler terms, if solving Vertex Cover is inherently difficult (NP-complete), then solving MW2SAT, which can express Vertex Cover problems, must be at least as hard.

In essence, this proof establishes MW2SAT as a more general framework that can encompass problems like Vertex Cover. The equivalence between solutions and the preservation of NP-completeness solidify this connection. □

This is the main theorem.

Theorem 1. *$MW2SAT \in P$.*

Proof. Finding a satisfying truth assignment in Monotone Weighted 2-Satisfiability (MW2SAT) with at most k true variables is equivalent to finding a set of at most k shared vertices that fulfills each edge within a minimum weighted edge cover in the line graph constructed from the MW2SAT formula. Here's the proof:

1. **Introducing a New Formula**:
    - We've created a new formula for the Boolean Satisfiability Problem with Exactly Two Variables per Clause (MW2SAT). This formula ensures that every variable appears in at least two clauses:

  - To achieve this, we remove clauses containing only two variables (x and y) if neither x nor y appear in any other clauses. We then choose one of these variables (say x), set it to true, and reduce the number of true variables (k) by 1.
  - For example, consider the clause $(x \vee y)$. Since neither x nor y appear elsewhere, we can set x to true and remove the clause from the formula. We also decrease the value of k by 1.

2. **Graph Construction**:
   - We construct a graph G where:
     - Each vertex represents a variable in the new MW2SAT formula.
     - Two vertices in G are connected by an edge if the corresponding variables appear together in a clause (e.g., for clause $(x \vee y)$, an edge connects x and y).
3. **Line Graph and Weighted Edge Cover**:
   - We create the line graph L of G. In L:
     - The line graph L encodes the shared vertices between edges in G. Each edge in L is a pair of edges from G that have a shared vertex This vertex is included in the edge representation of L, for example, as $((u,v),(u,w))$ where u is the shared vertex between the two edges (u,v) and (u,w) in G [7].
     - We assign for each edge in the line graph L a weight that is the natural logarithm of the amount of adjacent vertices in G of the corresponding shared vertex u.
     - Since every variable appears in at least two clauses in the new MW2SAT formula, then we are always able to find an edge cover in the line graph L.
   - We now establish the connection between MW2SAT truth assignments and edge covers in L:
     - A truth assignment in MW2SAT with at most k true variables corresponds to a set S containing at most k shared vertices which belong to an edge cover in L. These vertices are the shared vertices of edges in L where the corresponding variables are true in the MW2SAT assignment.
4. **Equivalence and Properties**:
   - The properties of MW2SAT clauses ensure the following:
     - **Vertex Cover**: Every edge in G (represented by a vertex in L) has at least one endpoint (shared vertex) included in S. This is because:
       * The key to vertex cover in L lies in the shared vertex. It represents variables that are true together in the MW2SAT solution. Since an edge in L connects edges in G that share this vertex, it ensures at least one endpoint (variable) from each edge in G is included in the set S, satisfying the vertex cover condition.
     - **Minimum Cover**: The set S containing at most k shared vertices satisfies the minimum weighted edge cover condition when it is applied the minimum weighted edge cover algorithm on L. This is because:
       * The minimum weighted edge cover algorithm seeks the smallest set of edges that covers all vertices in L which are all existing edges in G, such that every edge of L share a single shared vertex, aligning with the goal of minimizing true variables in MW2SAT. Since we maximize the weighted sum of the possible minimum edge cover in L, then we guarantee that the vertex cover in G is minimized. Besides, we guarantee the minimum cover just removing the redundant vertices from the solution of the minimum weighted edge cover algorithm:

the redundant vertices are those ones which can be removed from the solution and the remaining solution subset is still a vertex cover in G.

5. **Polynomial Time Solvability**:
   - Since finding the minimum weighted edge cover in an undirected graph is solvable in polynomial time [5], finding a satisfying truth assignment with at most k true variables in MW2SAT is also solvable in polynomial time. This is because the problem reduces to finding a minimum weighted edge cover in the constructed line graph, which is equivalent to finding a minimum vertex cover in G.

Conclusion: Therefore, solving MW2SAT with a maximum of k true variables is equivalent to finding a minimum weighted edge cover in the line graph L with a maximum of k shared vertices. As the latter problem is solvable in polynomial time, so is MW2SAT with this constraint. □

4. Discussion

We create a software programming implementation in Python for the whole algorithm that solves MW2SAT instances just using the NetworkX's Library and its dependencies (this code in Python would be outside of the necessary correctness of the paper and thus, this can only be considered as an appendix that will not compromise the whole result). [8]. This is placed into a GitHub repository under my GitHub username "frankvegadelgado" [8]. The last commit was on June 1st of 2024 with a SHA commit **3b67c4591f9c39e75a8dd7bb3e162981ed9f52cc** [8].

5. Conclusion

A proof of $P = NP$ will have stunning practical consequences, because it possibly leads to efficient methods for solving some of the important problems in computer science [1]. The consequences, both positive and negative, arise since various NP-complete problems are fundamental in many fields [2]. But such changes may pale in significance compared to the revolution an efficient method for solving NP-complete problems will cause in mathematics itself [1]. Research mathematicians spend their careers trying to prove theorems, and some proofs have taken decades or even centuries to be discovered after problems have been stated [1]. A method that guarantees to find proofs for theorems, should one exist of a "reasonable" size, would essentially end this struggle [1].

References

1. Cook, S.A. The P versus NP Problem, Clay Mathematics Institute. https://www.claymath.org/wp-content/uploads/2022/06/pvsnp.pdf, 2022. Accessed 1 June 2024.
2. Fortnow, L. The status of the P versus NP problem. *Communications of the ACM* **2009**, *52*, 78–86. https://doi.org/10.1145/1562164.1562186.
3. Cormen, T.H.; Leiserson, C.E.; Rivest, R.L.; Stein, C. *Introduction to Algorithms*, 3rd ed.; The MIT Press, 2009.
4. Garey, M.R.; Johnson, D.S. *Computers and Intractability: A Guide to the Theory of NP-Completeness*, 1 ed.; San Francisco: W. H. Freeman and Company, 1979.
5. Galil, Z. Efficient algorithms for finding maximum matching in graphs. *ACM Computing Surveys (CSUR)* **1986**, *18*, 23–38. https://doi.org/10.1145/6462.6502.
6. Flum, J.; Grohe, M. *Parameterized Complexity Theory, Springer*; Heidelberg, 2006; pp. 69–70. https://doi.org/10.1007/3-540-29953-X.
7. Hemminger, R.L. Line graphs and line digraphs. *Selected Topics in Graph Theory* **1983**, pp. 271–305.
8. Vega, F. MENTE | MW2SAT Solver. https://github.com/frankvegadelgado/mente, 2024. Accessed 1 June 2024.

Short Biography of Authors

Frank Vega is essentially a Back-End Programmer and Mathematical Hobbyist who graduated in Computer Science in 2007. In May 2022, The Ramanujan Journal accepted his mathematical article about the Riemann hypothesis. The article "Robin's criterion on divisibility" makes several significant contributions to the field of number theory. It provides a proof of the Robin inequality for a large class of integers, and it suggests new directions for research in the area of analytic number theory.

6. Appendixes

The following block is a main fragment of the Python code in the GitHub repository [8]:

```python
import networkx
import numpy

def flatten(xss):
    return [x for xs in xss for x in xs]

def minimum_cover(graph):
    edges = list(graph.edges)
    non_incident = []
    for e in edges:
        found = False
        for ee in edges:
            if len(set([e[0], e[1]]) & set([ee[0], ee[1]])) == 1:
                found = True
                break
        if not found:
            non_incident.append(e)
    remaining = list(set(edges) - set(non_incident))
    non_line_cover = []
    line_cover = []
    if len(non_incident) > 0:
        non_line_cover = [y[0] for y in non_incident]
    if len(remaining) > 0:
        G = networkx.Graph()
        G.add_edges_from(remaining)
        initial_edges = list(G.edges)
        old_edges = list(networkx.line_graph(G).edges)
        new_edges = []
        for x in old_edges:
            z = list(set([x[0][0], x[0][1]]) & set([x[1][0], x[1][1]]))
            weight = len([list(set([y[0], y[1]]) - set(z)) for y in
                                                initial_edges if z[0] == y
                                                [0] or z[0] == y[1]])
            new_edges.append((x[0], x[1], {'weight': numpy.log(weight)}))
        L = networkx.Graph()
        L.add_edges_from(new_edges)
        edge_cover = networkx.min_edge_cover(L)
        line_cover = list(set(flatten([list(set([y[0][0], y[0][1]]) & set([y[
                                            1][0], y[1][1]])) for y in
                                            edge_cover])))
        redundant = []
        for vertex in line_cover:
            covered = True
            redundant.append(vertex)
            new_oover = list(set(line_cover) - set(redundant))
            for y in initial_edges:
                if y[0] not in new_oover and y[1] not in new_oover:
                    covered = False
                    break
            if not covered:
                redundant.remove(vertex)
        line_cover = list(set(line_cover) - set(redundant))
    return line_cover + non_line_cover
```