National Polytechnic Institute
Superior School of Computer Sciences

Algorithm Analysis.

Practice 10: Hamiltonian Cycle.

Hernandez Martinez Carlos David.
Burciaga Ornelas Rodrigo Andres.

davestring@outlook.com.
andii_burciaga@live.com.

Group: 2cv3.

December 4, 2017

Contents

1 Introduction:

First, although we may reasonably regard a problem that requires time $\mathcal{O}(n^{100})$ to be intractable, very few practical problems require time on the order of such a high-degree polynomial. The polynomial-time computable problems encountered in practice typically require much less time. Experience has shown that once the first polynomial-time algorithm for a problem has been discovered, more efficient algorithms often follow. Even if the current best algorithm for a problem has a running time of $\mathcal{O}(n^{100})$, an algorithm with a much better running time will likely soon be discovered. Second, for many reasonable models of computation, a problem that can be solved in polynomial time in one model can be solved in polynomial time in another. Third, the class of polynomial-time solvable problems has nice closure properties, since polynomials are closed under addition, multiplication, and composition.

1.1 Polynomial-Time Verification:

The problem of finding a **Hamiltonian Cycle** in an undirected graph has been studied for over a hundred years. Formally, a **Hamiltonian Cycle** of an undirected graph $G = (V, E)$ is a simple cycle that contains each vertex in V. A graph that contains a **Hamiltonian Cycle** is said to be **Hamiltonian**; otherwise, it is non-hamiltonian. The name honors W. R. Hamilton, who described a mathematical game on the dodecahedron (Figure 1.1.0 **(a)**) in which one player sticks five pins in any five consecutive vertices and the other player must complete the path to form a cycle containing all the vertices.

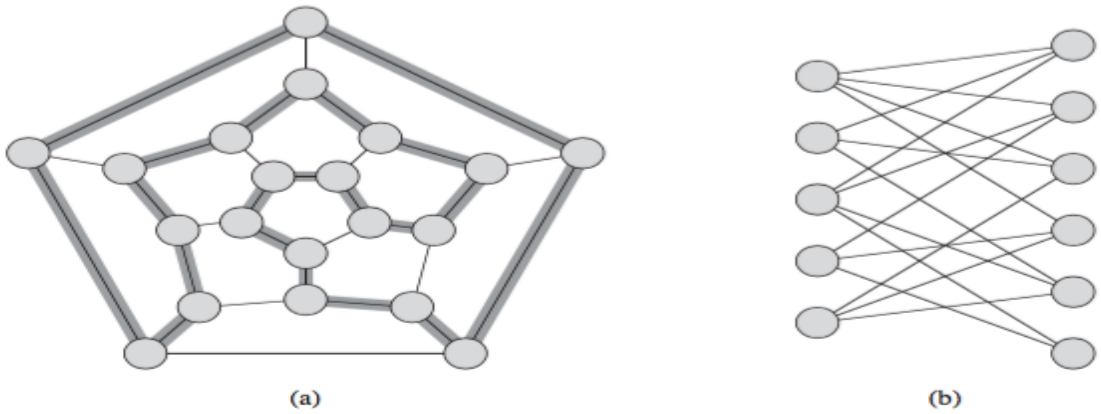

Figure 1.1.0: **(a)** A graph representing the vertices, edges, and faces of a dodecahedron, with a **Hamiltonian Cycle** shown by shaded edges. **(b)** A bipartite graph with an odd number of vertices. Any such graph is non-hamiltonian.

The dodecahedron is **Hamiltonian**, and Figure 1.1.0 **(a)** shows one **Hamiltonian Cycle**. Not all graphs are **Hamiltonian**, however. For example, Figure 1.1.0 **(b)** shows a bipartite graph with an odd number of vertices.

We can define the **hamiltonian-cycle problem**, Does a graph G have a Hamiltonian cycle? as a formal language:

$$\text{HAM-CYCLE} = \{ \langle G \rangle : G \text{ is a } \textbf{Hamiltonian} \text{ Graph} \}$$

How might an algorithm decide the language HAM-CYCLE? Given a problem instance $\langle G \rangle$, one possible decision algorithm lists all permutations of the vertices of G and then checks each permutation to see if it is a Hamiltonian path. What is the running time of this algorithm? If we use the reasonable encoding of a graph as its adjacency matrix, the number **m** of vertices in the graph is $\Omega(\sqrt{n})$, where $n = |\langle G \rangle|$ is the length of the encoding of G There are **m!** possible permutations of the vertices, and therefore the running time is $\Omega(m!) = \Omega(\sqrt{n}) = \Omega(2^{\sqrt{n}})$, which is not $\mathcal{O}(n^k)$ for any constant k. Thus, this naive algorithm does not run in polynomial time. In fact, the **hamiltonian-cycle problem** is NP-complete.

1.2 Verification Algorithms:

Consider a slightly easier problem. Suppose that a friend tells you that a given graph G is **Hamiltonian**, and then offers to prove it by giving you the vertices in order along the hamiltonian cycle. It would certainly be easy enough to verify the proof: simply verify that the provided cycle is hamiltonian by checking whether it is a permutation of the vertices of V and whether each of the consecutive edges along the cycle actually exists in the graph. You could certainly implement this verification algorithm to run in $\mathcal{O}(n^2)$ time, where n is the length of the encoding of G. Thus, a proof that a **Hamiltonian Cycle** exists in a graph can be verified in polynomial time.

We define a **verification algorithm** as being a two-argument algorithm A, where one argument is an ordinary input string x and the other is a binary string y called a certificate. A two-argument algorithm A verifies an input string x if there exists a certificate y such that $\mathbf{A\ (\ x,\ y\ ) = 1}$. The **language verified** by a verification algorithm A is:

$$\mathrm{L} = \{\ x \in \{\ 0,\ 1\ \}^* : \text{there exists } y \in \{\ 0,\ 1\ \}^* \text{ such that } A\ (\ x,\ y\ ) = 1\ \}$$

Intuitively, an algorithm A verifies a language **L** if for any string $x \in L$, there exists a certificate y that A can use to prove that $x \in L$. Moreover, for any string $x \notin L$, there must be no certificate proving that $x \in L$. For example, in the **Hamiltonian-Cycle problem**, the certificate is the list of vertices in some Hamiltonian Cycle. If a graph is Hamiltonian, the Hamiltonian Cycle itself offers enough information to verify this fact. Conversely, if a graph is not-hamiltonian, there can be no list of vertices that fools the verification algorithm into believing that the graph is Hamiltonian, since the verification algorithm carefully checks the proposed cycle to be sure.

2 Basic Concepts:

Hamiltonian Cycle Definition: If $G = (V, E)$ is a graph or multigraph with $|V| \geq 3$, we say that G has a *Hamilton Cycle* if there is a cycle in G that contains every vertex in V.

Theorem: Let $G = (V, E)$ be a loop-free undirected graph with $|V| = n \geq 3$. If $deg(x) + deg(y) \geq n$ for all nonadjacent $x, y \in V$, then G contains a *Hamilton Cycle*.

Corollary: If $G = (V, E)$ be a loop-free undirected graph with $|V| = n \geq 3$ and if $deg(v) \geq \frac{n}{2}$ for all $v \in V$, then G has a *Hamilton Cycle*.

Corollary: If $G = (V, E)$ be a loop-free undirected graph with $|V| = n \geq 3$ and if $|E| \geq \binom{n-1}{2} + 2$, then G has a *Hamilton Cycle*.

3 Development:

In this section we will explain the algorithm that verifies if a *certificate* C it's or not a **Hamiltonian-Cycle** of a graph G.

3.1 Hamiltonian-Cycle Verification:

As we have seen in section 1, the **Hamiltonian-Cycle Problem** is *NP-complete*, but if we suppose that a graph G it's *Hamiltonian* and we have a *Certificate* C (list of vertices in order along the *Hamiltonian-Cycle*) that can prove this supposition, then the problem can be solvable in polynomial time only by checking wheter it is a permutation of the vertices of V and whether each of the consecutive edges along the cycle actually exists in the graph. In theory, the algorithm should run in $\mathcal{O}(n^2)$.

The algorithm that we wrote it's in an only *python module* that has 3 methods including the main.

```
1 def main ( ):
2     graph = { 1: [ 2, 3, 4 ], 2: [ 1 ], 3: [ 1, 2, 4 ], 4: [ 1, 2, 3 ] }
3     certificate = [ 2, 1, 3, 4, 2 ]
4     verify_hamiltonian ( graph, certificate )
```

As we can see, the main method only declares a *graph (G)* and the *certificate (C)*, and in line 4 calls the method *verify_hamiltonian (G, C)*.

```
1 def verify_hamiltonian ( graph, certificate ):
2     for i in range ( len ( graph ) ):
3         neighbors = graph [ certificate [ i ] ]
4         if ( certificate [ i + 1 ] not in neighbors ):
5             printer ( -1, certificate )
6     printer ( 0, certificate )
```

The previous algorithm verify if C it's or not a **Hamiltonian-Cycle** of G. The algorithm works as follows, in the input has the *graph* and the *certificate* the in line 2 there is a **for** loop that will run from *0* to *n - 1* where n it's the number of vertices in G, then in line 3 we will store the *neighbors* or *adjacent* vertices of the node in which the algorithm it's situated. In line 4 the program asks if the next vertex in the given *certificate* it's or not a neighbor of the current node, in case that it is, the program continues running, otherwise calls the method printer with an error flag (-1) and exits the program.

Finally the last method prints in console if C it's a **Hamiltonian-Cycle** of G.

```
1 def printer ( flag, C ):
2     if ( flag == -1 ):
3         print ( "\n\n\tThe certificate {} isn't a Hamiltonian Cycle.\n".format ( C ) )
4         exit ( 0 )
5     print ( "\n\n\tThe certificate {} is a Hamiltonian Cycle.\n".format ( C ) )
```

4 Results:

All the code shown above doesn't have significance if its operation is not shown. This section will show you the output of the **Hamiltonian-Cycle** algorithm, and we will verify if it's $O\ (\ n^2\ )$.

4.1 Test 1:

The first test consists to prove that the **Hamiltonian-Cycle** in Figure 4.1.0 is actually one (*C = [8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 1, 2, 3, 4, 5, 6, 7, 8]*), so first we declare the graph in our program, for this we use something that python call *dictionary* which it's very similar to a *Hash-map*. The left elements of the ':' will be the nodes and right element is a list whit its adjacent vertices. We can corroborate this in Figure 4.1.0. After running the program we will have a console output (Figure 4.1.1) and the plot of the computational time of the algorithm (Figure 4.1.2). Finally the table 1 describe the point of Figure 4.1.2.

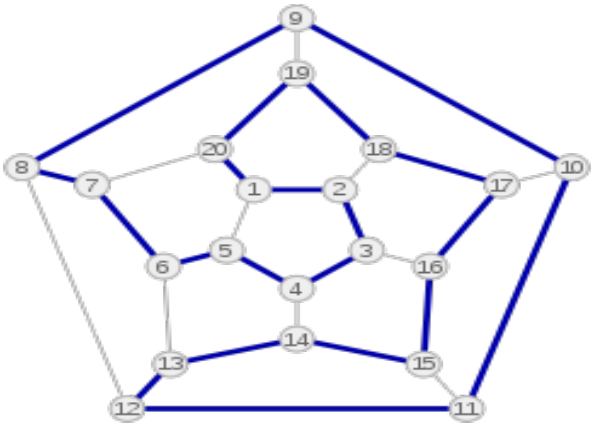

Figure 4.1.0: Graph to analyze.

```
def main ( ):
    graph = {
        1: [ 20, 2, 5 ],  2: [ 1, 18, 3 ], 3: [ 2, 16, 4 ], 4: [ 3, 5, 14 ],
        5: [ 4, 6, 1 ], 6: [ 5, 13, 7 ], 7: [ 8, 6, 20 ], 8: [ 7, 9, 12 ],
        9: [ 8, 19, 10 ], 10: [ 9, 17, 11 ], 11: [ 10, 15, 12 ], 12: [ 8, 13, 11 ],
        13: [ 12, 6, 14 ], 14: [ 13, 4, 15 ], 15: [ 14, 11, 16 ], 16: [ 15, 17, 3 ],
        17: [ 10, 16, 18 ], 18: [ 17, 2, 19 ], 19: [ 9, 18, 20 ], 20: [ 1, 19, 7 ]
    }
    certificate = [ 8, 9, 10, 11, 12, 13, 14, 15, 16, 17,
                    18, 19, 20, 1, 2, 3, 4, 5, 6, 7, 8 ]
    verify_hamiltonian ( graph, certificate )
```

```
(myenv) MacBook-Pro-de-David:Hamiltonian Cycle Davestring$ python3 main.py

        The certificate [8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 1, 2, 3, 4, 5, 6, 7, 8] is a Hamiltonian Cycle.
```

Figure 4.1.1: Console output.

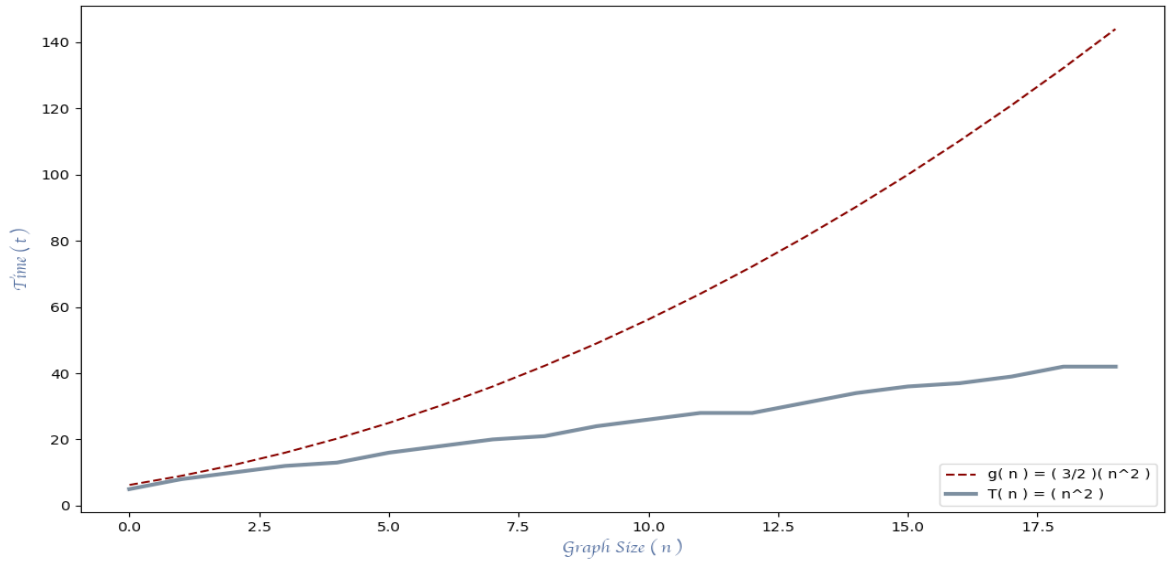

Figure 4.1.2: Plot of the algorithm.

Observation: *Figure 4.1.2 has 2 plots, the red one it's and asymptotic function n^2 for the blue one, which it's the computational time of the algorithm.*

Graph Size	Time
0	5
1	8
2	10
3	12
4	13
5	16
6	18
7	20
8	21
9	24
10	26
11	28
12	28
13	31
14	34
15	36
16	37
17	39
18	42
19	42

Table 1.

4.2 Test 2:

The second test consists to prove that the **Hamiltonian-Cycle** in Figure 4.2.0 is actually one (*C = [1, 4, 8, 7, 5, 6, 3, 2, 1]*). Once we have declared the graph and the certificate, we run the program and see the console output (Figure 4.2.1) and the plot of the computational time of the algorithm (Figure 4.2.2). Finally the table 2 describe the point of Figure 4.2.2.

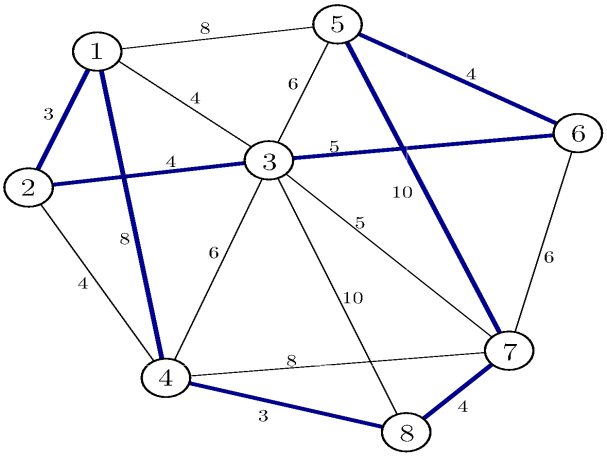

Figure 4.2.0: Graph to analyze.

```
graph = { 1: [ 2, 4, 3, 5 ], 2: [ 1, 3, 4 ], 3: [ 1, 2, 4, 5, 6, 7, 8 ],
          4: [ 1, 2, 3, 7, 8 ], 5: [ 1, 3, 7, 6 ], 6: [ 5, 3, 7 ],
          7: [ 8, 4, 3, 5, 6 ], 8: [ 4, 3, 7 ] }
    certificate = [ 1, 4, 8, 7, 5, 6, 3, 2, 1 ]
    verify_hamiltonian ( graph, certificate )
```

```
(myenv) MacBook-Pro-de-David:Hamiltonian Cycle Davestring$ python3 main.py

[       The certificate [1, 4, 8, 7, 5, 6, 3, 2, 1] is a Hamiltonian Cycle.
```

Figure 4.2.1: Console output.

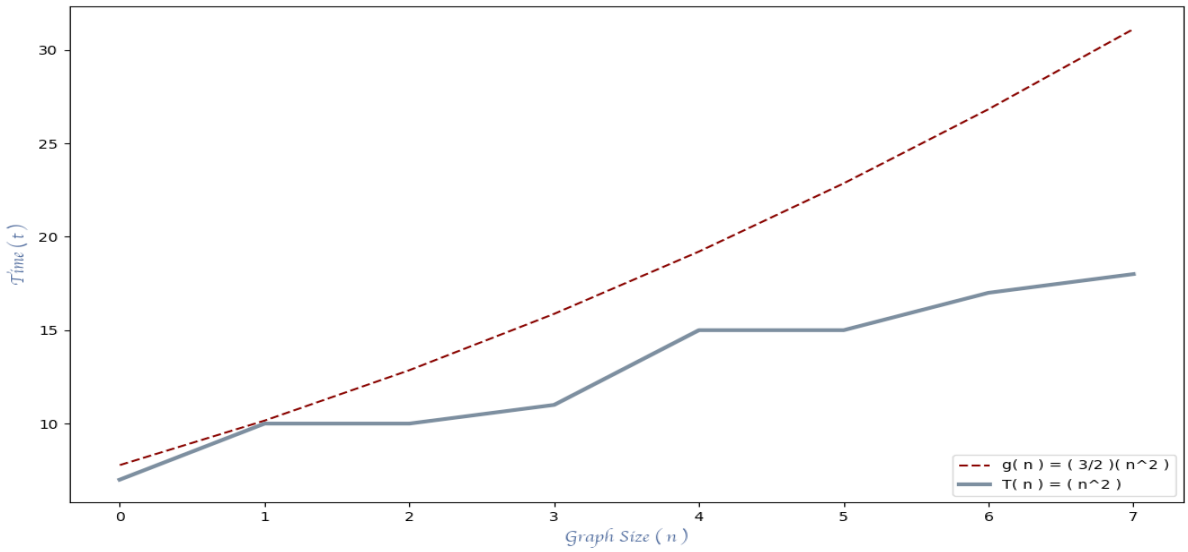

Figure 4.2.2: Plot of the algorithm.

Graph Size	Time
0	7
1	10
2	10
3	11
4	15
5	15
6	17
7	18

Table 2.

4.3 Test 3:

The third test consists to prove that the **Hamiltonian-Cycle** in Figure 4.3.0 is actually one ($C = [\ 1, 2, 3, 4, 5, 6, 7, 8, 1\ ]$). Once we have declared the graph and the certificate, we run the program and see the console output (Figure 4.3.1) and the plot of the computational time of the algorithm (Figure 4.3.2). Finally the table 3 describe the point of Figure 4.3.2.

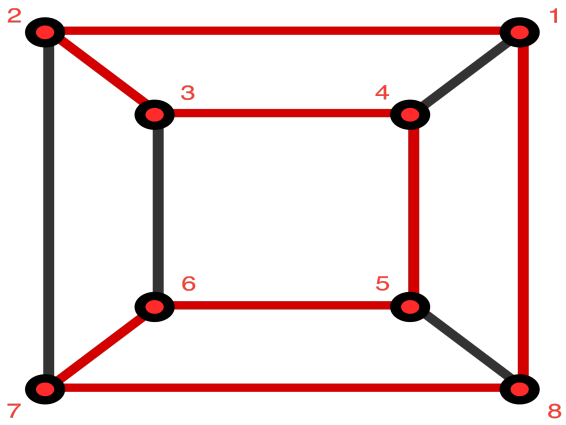

Figure 4.3.0: Graph to analyze.

```
def main ( ):
    graph = { 1: [ 2, 4, 8 ], 2: [ 7, 3, 1 ], 3: [ 2, 6, 4 ],
              4: [ 1, 3, 5 ], 5: [ 4, 8, 6 ], 6: [ 5, 3, 7 ],
              7: [ 6, 2, 8 ], 8: [ 7, 5, 1 ] }
    certificate = [ 1, 2, 3, 4, 5, 6, 7, 8, 1 ]
    verify_hamiltonian ( graph, certificate )
```

```
(myenv) MacBook-Pro-de-David:Hamiltonian Cycle Davestring$ python3 main.py

        The certificate [1, 2, 3, 4, 5, 6, 7, 8, 1] is a Hamiltonian Cycle.
```

Figure 4.3.1: Console output.

Graph Size	Time
0	4
1	7
2	10
3	12
4	14
5	16
6	18
7	20

Table 3.

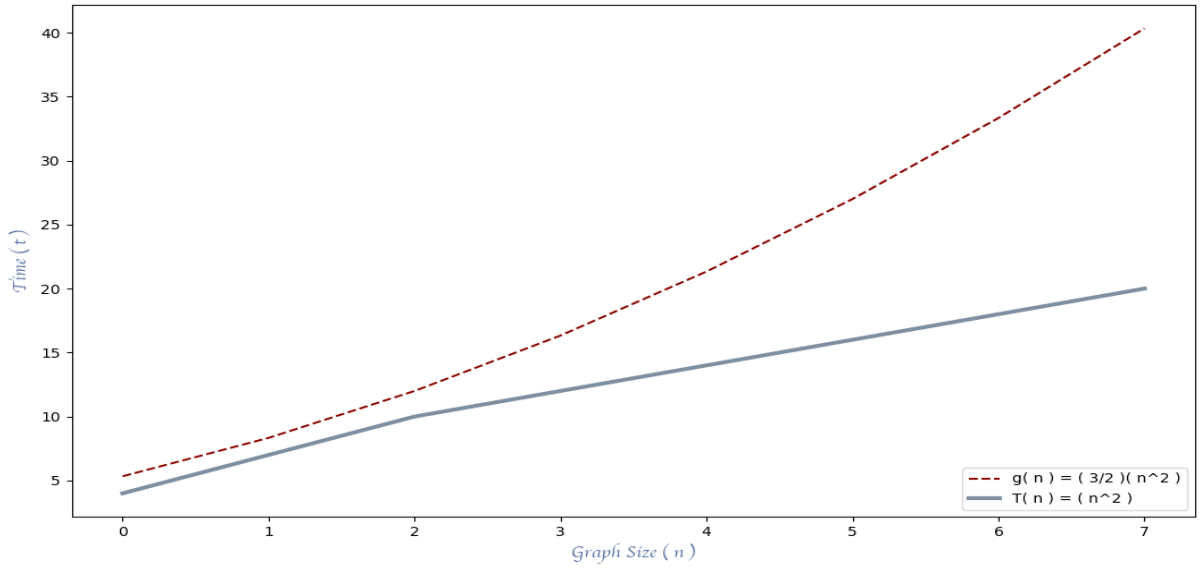

Figure 4.3.2: Plot of the algorithm.

4.4 Test 4:

For the same graph of Figure 4.3.0, we will analyze the certificate *C = [1, 2, 3, 4, 5, 6, 7, 7, 1]*, as we can see clearly, this isn't a **Hamiltonial-Cycle** because has a repeated vertex which it's the node 7, so, the program will stop analyzing and will only print on the console output (Figure 4.4.0) that the certificate isn't a **Hamiltonial-Cycle**.

```
def main ( ):
    graph = { 1: [ 2, 4, 8 ], 2: [ 7, 3, 1 ], 3: [ 2, 6, 4 ],
              4: [ 1, 3, 5 ], 5: [ 4, 8, 6 ], 6: [ 5, 3, 7 ],
              7: [ 6, 2, 8 ], 8: [ 7, 5, 1 ] }
    certificate = [ 1, 2, 3, 4, 5, 6, 7, 7, 1 ]
    verify_hamiltonian ( graph, certificate )
```

```
(myenv) MacBook-Pro-de-David:Hamiltonian Cycle Davestring$ python3 main.py
[
        The certificate [1, 2, 3, 4, 5, 6, 7, 7, 1] isn't a Hamiltonian Cycle.
```

Figure 4.4.0: Console output.

5 Conclusion:

In this occasion we observed that not all problems that can be easily checked are so trivial to solve, a problem can have a verifiable solution in polynomial time but it does not mean that this solution has been reached in polynomial time, how is it possible that can that be done ?, the reason is because they are complete np, they are those that are verified polynomially but their solution is not in the same time, however there are some problems that besides being verifiable in polynomial time, have a solution in time polynomial, which tells us this, that the problems that most concern us are those that are in this range.

A good engineer in computer science should be able to distinguish between this type of problems easily, the problems p, np, np-complete and np-hard should be attacked and differentiated in order to know at what moment the resolution of a problem would be implosible. conflict, how can improve response times and also have a set of viable solutions for a certain occasion, not necessarily the best time is the algorithm that can help us meet our needs.

6 Bibliography References:

[1] Baase and Van Gelder. "Computer Algorithms: Introduction to Design and Analysis". Addison-Wesley.

[2] Thomas H. Cormen. "Introduction to Algorithms". The MIT press.

[3] Ralph P. Grimaldi. "Discrete and Combinatorial Mathematics". Addison-Wesley.