**CS 2302 Data Structures**

**Fall 2019**

**Lab Report #7**

Due: December 9th, 2019

Professor: Olac Fuentes

TA: Anindita Nath

**Introduction**

For this lab, we were asked to implement functions that utilize different algorithm design techniques to solve two distinct problems. The first problem involves finding a Hamiltonian cycle in an undirected graph represented as an adjacency list by implementing randomized and backtracking algorithms to solve the problem. The second problem involves finding the edit distance between two words by implementing dynamic programming algorithms to solve the problem. For this lab, it is key to remember the properties and characteristics of randomization, backtracking, and dynamic programming since they are required for all exercises. The main objective of this lab is to implement different functions for each of the two problems and compare their performance with each other (i.e. randomization vs. backtracking for the Hamiltonian cycle problem and two different variations of dynamic programming for the edit distance problem).

**Proposed Solution Design and Implementation**

**Operation #1: Randomized algorithm to solve the Hamiltonian cycle problem**

For this operation, I implemented a function that takes the sets of vertices and edges of an undirected graph represented as an adjacency list and returns a randomly generated subset of the set of edges that forms a Hamiltonian cycle (A Hamiltonian cycle exists if the graph that is built from the subset of the set of edges has one connected component, and the in-degree of every vertex is two). If none of the randomly generated subsets form a Hamiltonian cycle, it returns None. To do this, I set up a loop that repeats the process of finding a Hamiltonian cycle up to 1000 times. Inside the loop, the function generates a random subset of the set of edges of the original graph. The size of every randomly generated subset equals the number of vertices in the original graph. Once the random subset is generated, it builds the graph that will contain all the vertices of the original graph and the edges of random subset. Once that graph is generated, the function determines if it has one connected component. If it does, the function then determines if all the vertices in that graph have an in-degree of two. If they all do, the function then returns the randomly generated subset that forms a Hamiltonian cycle. However, if the graph fails either of the two requirements to have a Hamiltonian cycle, the function moves on to the next randomly generated subset and repeats this process until all 1000 tries have been exhausted.

**Operation #2: Backtracking algorithm to solve the Hamiltonian cycle problem**

For this operation, I implemented a function that takes the sets of vertices and edges of an undirected graph represented as an adjacency list and an empty set representing the subset of the set of edges and returns the subset of the set of edges that forms a Hamiltonian cycle from all possible subsets. If none of the subsets form a Hamiltonian cycle, it returns None. To do this, I set up a recursive function that builds all possible subsets of the set of edges until it finds the subset that forms a Hamiltonian cycle. The recursive function contains two base cases. The first base case returns an empty list ([]) if the size of the subset of the set of edges equals the number of vertices in the original graph and the graph built from that subset has a Hamiltonian cycle (In order to check the second condition, I implemented a helper function that takes the subset of the set of edges and determines if it forms a Hamiltonian cycle). The second base case returns None if the size of the subset of the set of edges is greater than the number of vertices in the original graph or there are no more edges in the set of edges to construct a subset. Past the base cases, the function calls itself by taking the first edge in the set of edges. This creates a subset of the set of edges. If the subset returned is not None, then the function returns the subset of the set of edges that has been determined to form a Hamiltonian cycle. Otherwise, it calls itself by not taking the first edge in the set of edges. The function will continue making recursive calls until it has determined that all possible subsets of the set of edges do not form a Hamiltonian cycle.

**Operation #3: Dynamic programming algorithm to solve the edit distance problem (original variation)**

For this operation, I added and modified the edit distance function provided in class by the instructor so that it returns not only the value of the edit distance between the two strings but also the edit distance table that was completed in order to determine the value of the edit distance. Other than that modification, the algorithm remains the same. The function constructs a 2D array in which the number of rows and columns are based on the lengths of the two strings. Once the 2D array is constructed, the function sets the default values for the elements of the first row and the first column. Afterwards, it begins traversing through the rest of the array to calculate the value of edit distance between the two strings. If both strings share the same letter at the current position, the function sets the value of that element to the value of the upper-left element in the same diagonal. Otherwise, it sets the value of the current element to the lowest value of the surrounding elements plus one. It should be noted that each surrounding element corresponds to a different character operation that can be done to convert the first string into the second string. The element to the left of the current element corresponds to insertion; the element above the current element corresponds to removal; the element on the same diagonal as the current element corresponds to replacement. The function repeats this process until it sets the value of the bottom-right corner of the 2D array as the value of the edit distance between the two strings.

**Operation #4: Dynamic programming algorithm to solve the edit distance problem (modified variation)**

For this operation, I modified the previous operation to allow replacement only in the case where the characters being interchanged are both vowels or both consonants. To do this, I added a list of vowels that contains the five vowels of the alphabet. As the function compares characters of the two strings to determine the value of the current element in the 2D array, it determine whether the two characters being currently compared are both vowels (i.e. both are found in the list of vowels) or both consonants (i.e. both are not in the list of vowels). If either of the two cases are true, the function allows replacements in order to calculate the value of the current element. Otherwise, it does not allow replacements, and it checks only the other two adjacent elements in order to calculate the value of the current element.

In order to test the first two functions, I implemented a function that generates a complete graph Kn, where n is the number of vertices entered by the user, that is used in both functions to compare their efficiency by computing the running time in nanoseconds it takes each function to test a large number of subsets of the set of edges in order to find the subset that forms a Hamiltonian cycle in an undirected, complete graph, if there is one. It should be noted that the number of edges in the complete graph, which is calculated as follows: |E| = , is the main variable that is used to compare the efficiency of these two functions. Additionally, the program displays to the terminal, in no particular order, the subset of the set of edges that forms a Hamiltonian cycle for each function. Furthermore, it draws not only the original graph but also the two graphs that correspond to each function, which contain the Hamiltonian cycle represented as the subset of the set of edges displayed to the terminal.

In order to test the remaining two functions, I prompted the user to enter two words that are used in both functions to compare the efficiency of each function by computing the running time in nanoseconds it takes each function to calculate the edit distance between the two words entered by the user. It should be noted that the length of the two words is the main variable that is used to compare the efficiency of these two functions.

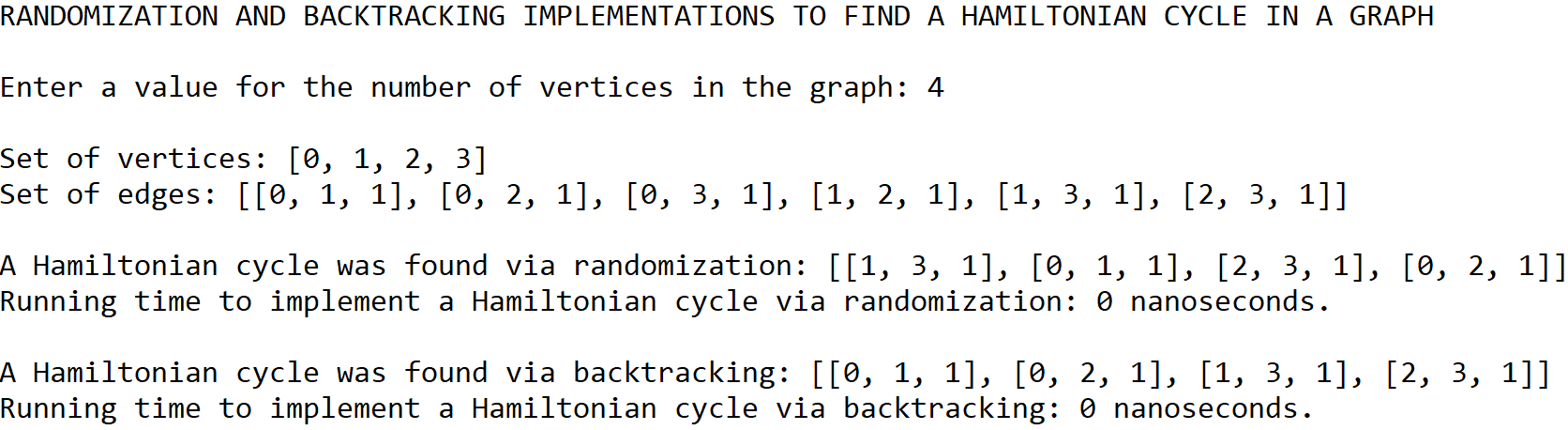
**Experimental Results**

**Randomized and backtracking algorithms implemented to find a Hamiltonian cycle in an undirected graph**

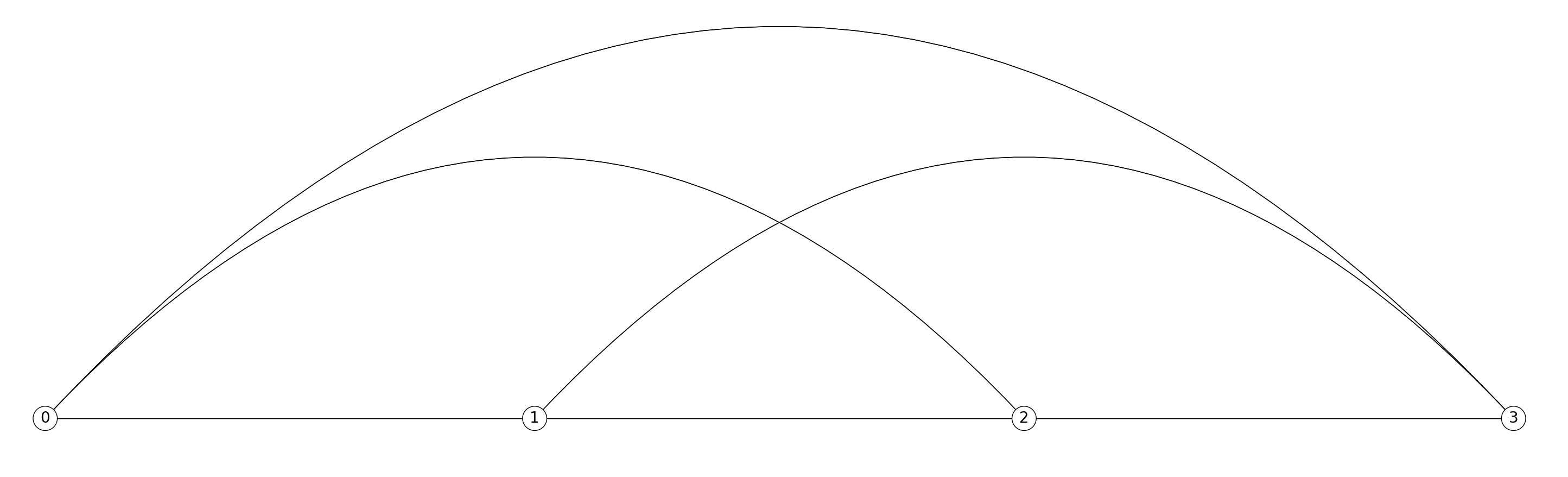
**Operations #1 and #2:**

For the first two operations, I will test undirected, complete graphs that have 4, 5, 6, and 7 vertices, which means that they will have 6, 10, 15, and 21 edges, respectively. Additionally, I will display the graphs that result from finding a Hamiltonian cycle in the original graph.

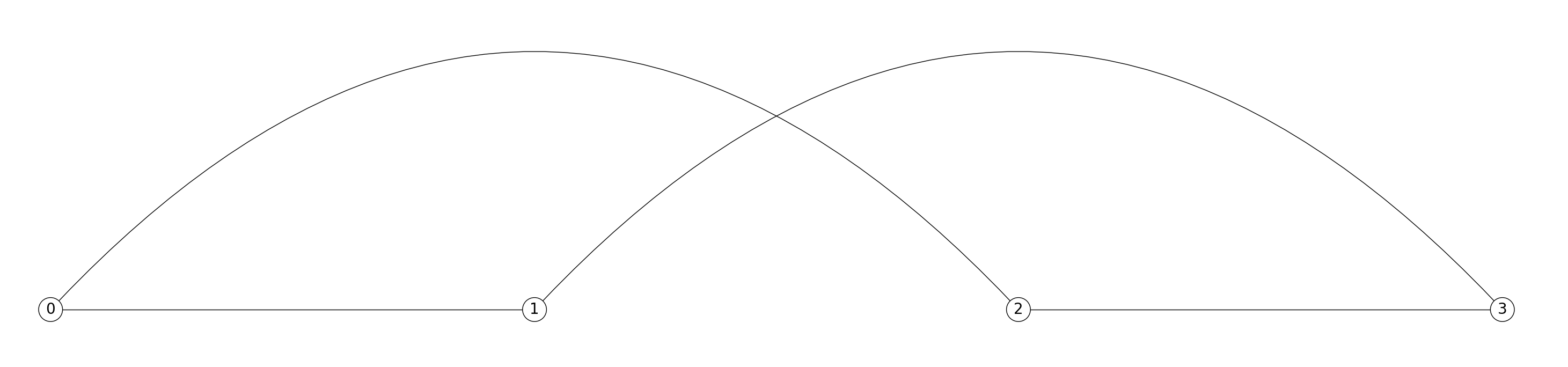
Case 1: Number of edges in the graph = 6



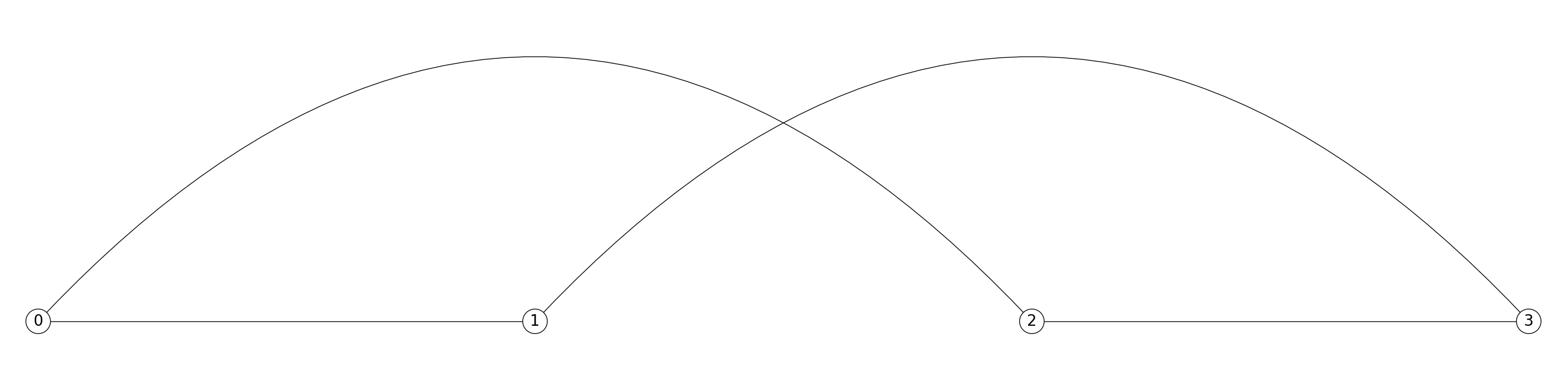
Original graph containing its 4 vertices and 6 edges



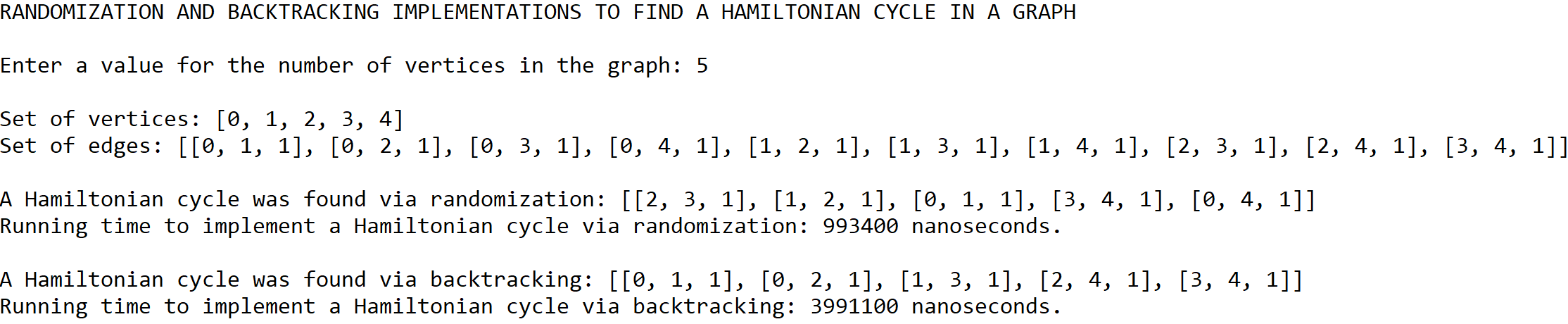
Graph containing the Hamiltonian cycle found via the randomization algorithm



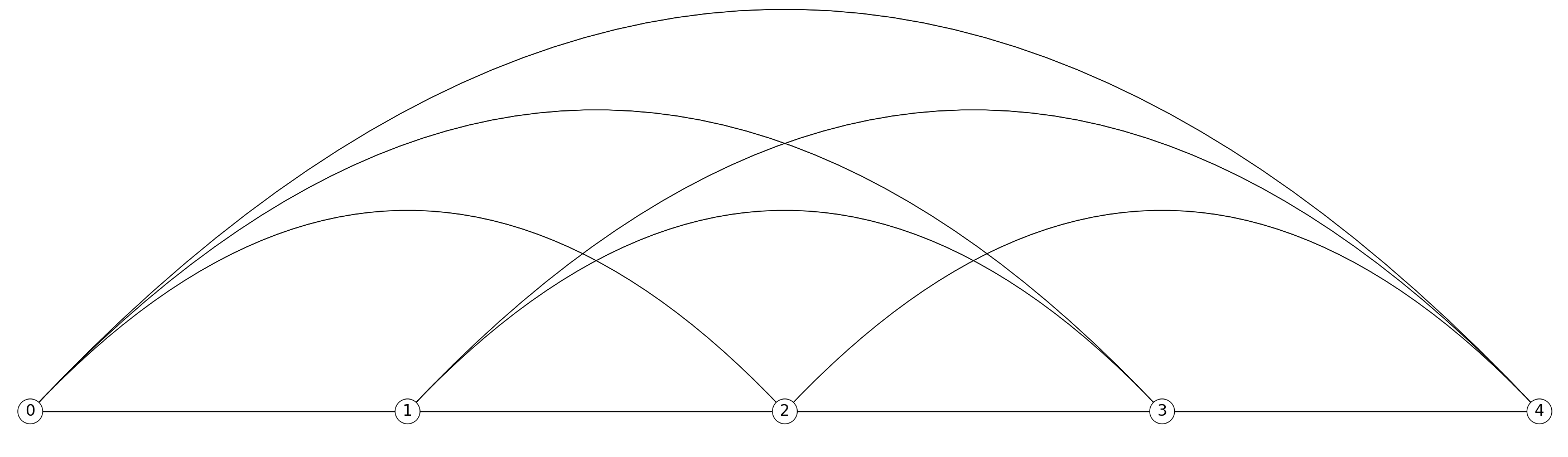
Graph containing the Hamiltonian cycle found via the backtracking algorithm



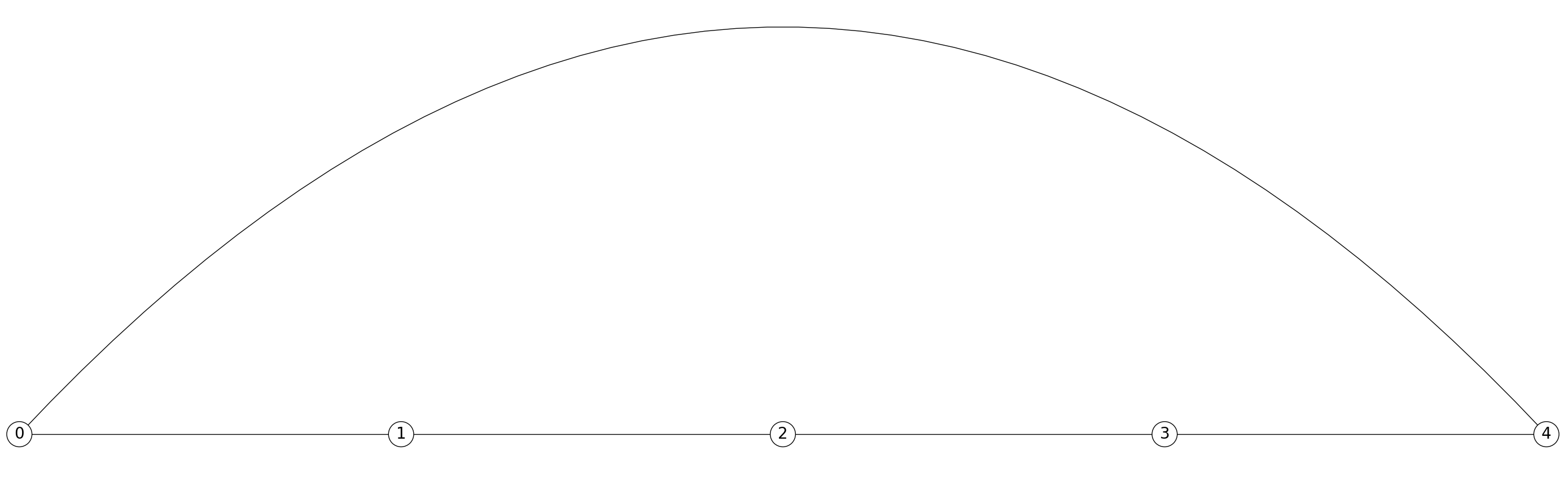
Case 2: Number of edges in the graph = 10



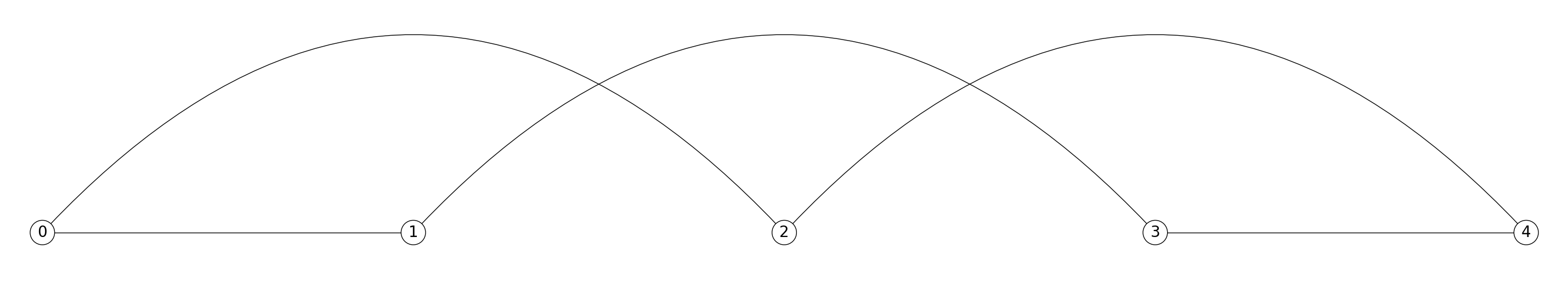
Original graph containing its 5 vertices and 10 edges



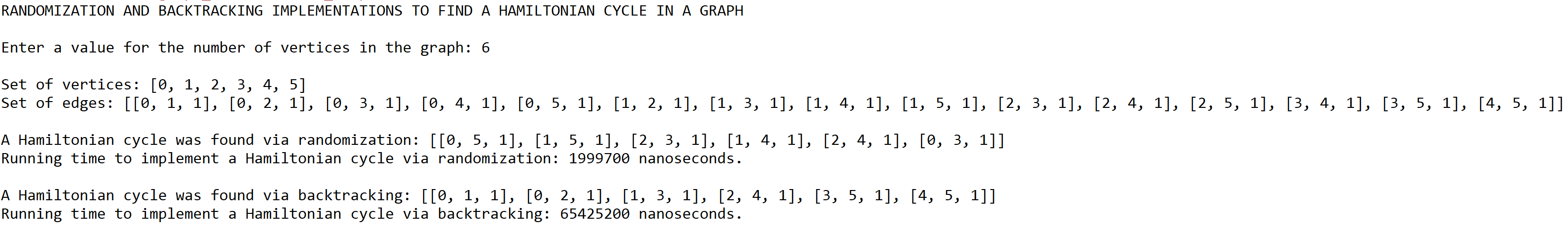
Graph containing the Hamiltonian cycle found via the randomization algorithm



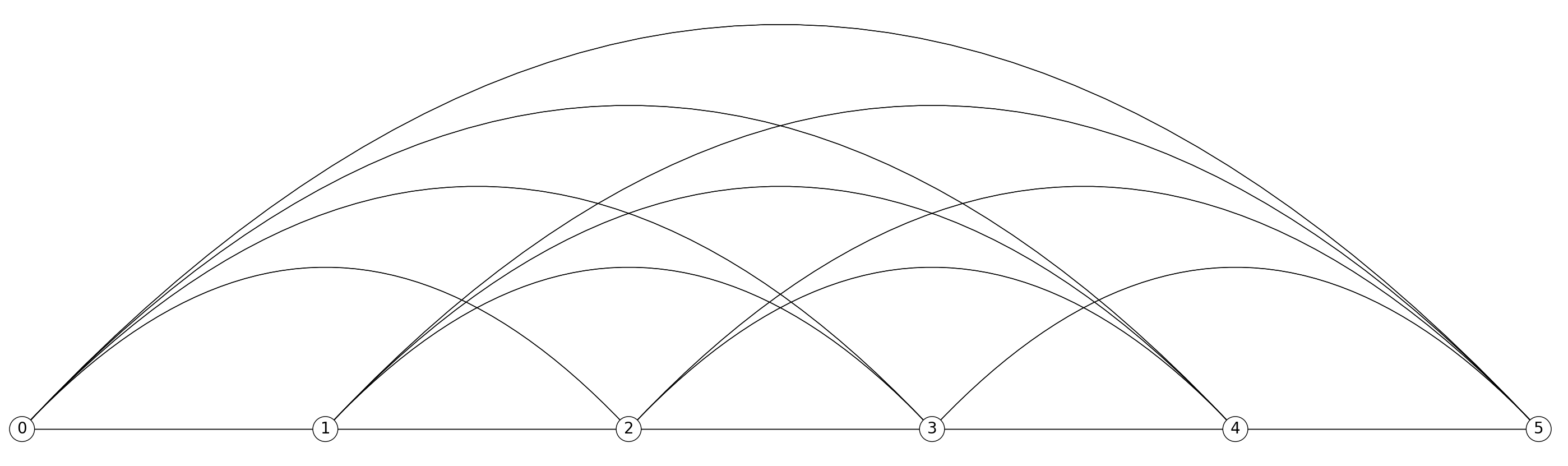
Graph containing the Hamiltonian cycle found via the backtracking algorithm



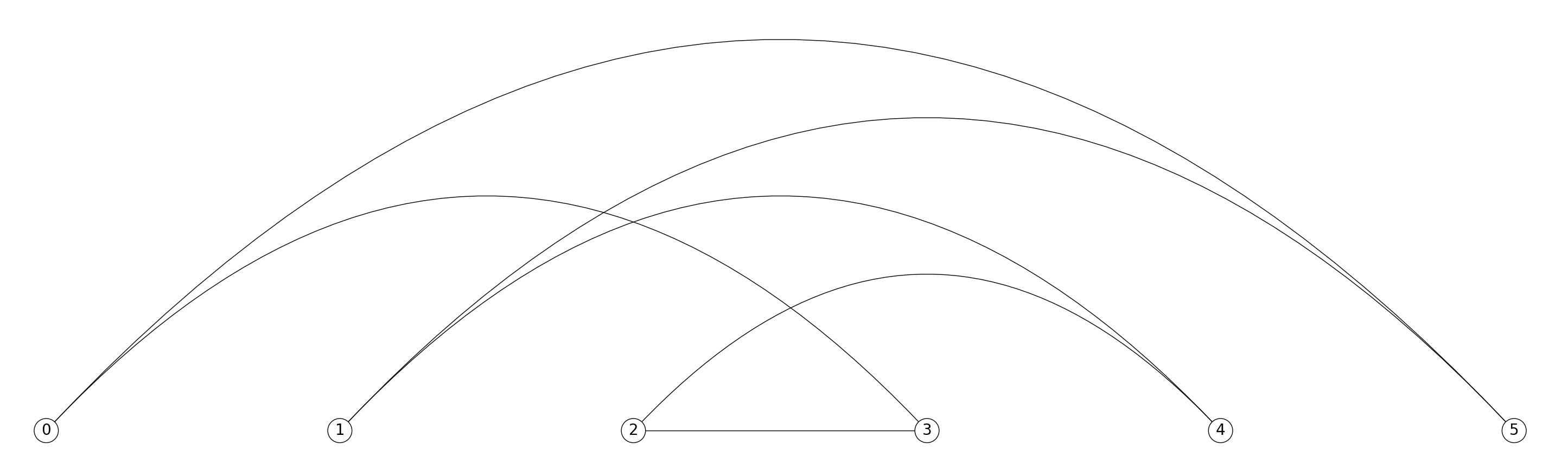
Case 3: Number of edges in the graph = 15



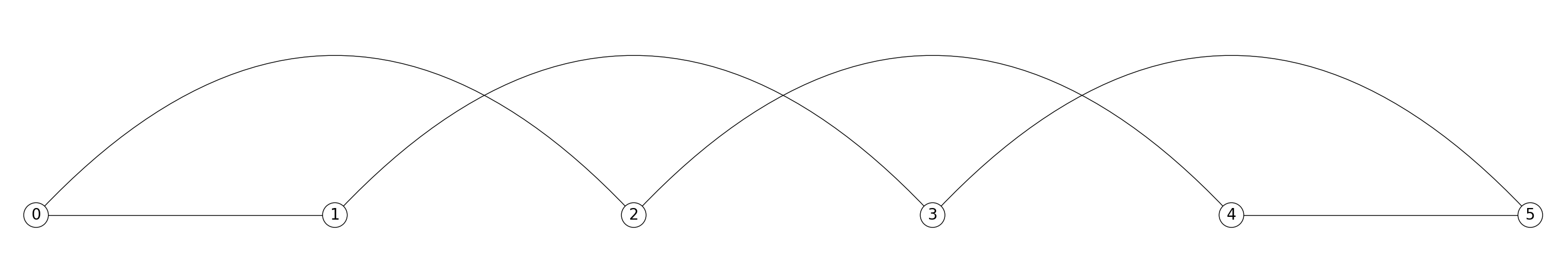
Original graph containing its 6 vertices and 15 edges



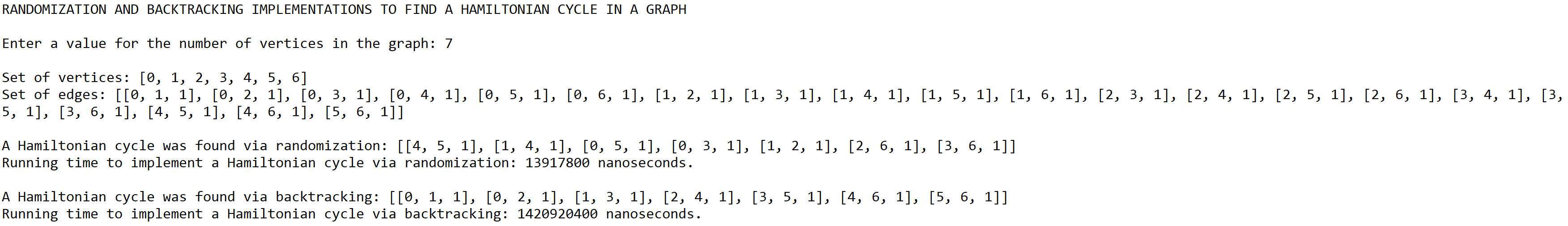
Graph containing the Hamiltonian cycle found via the randomization algorithm



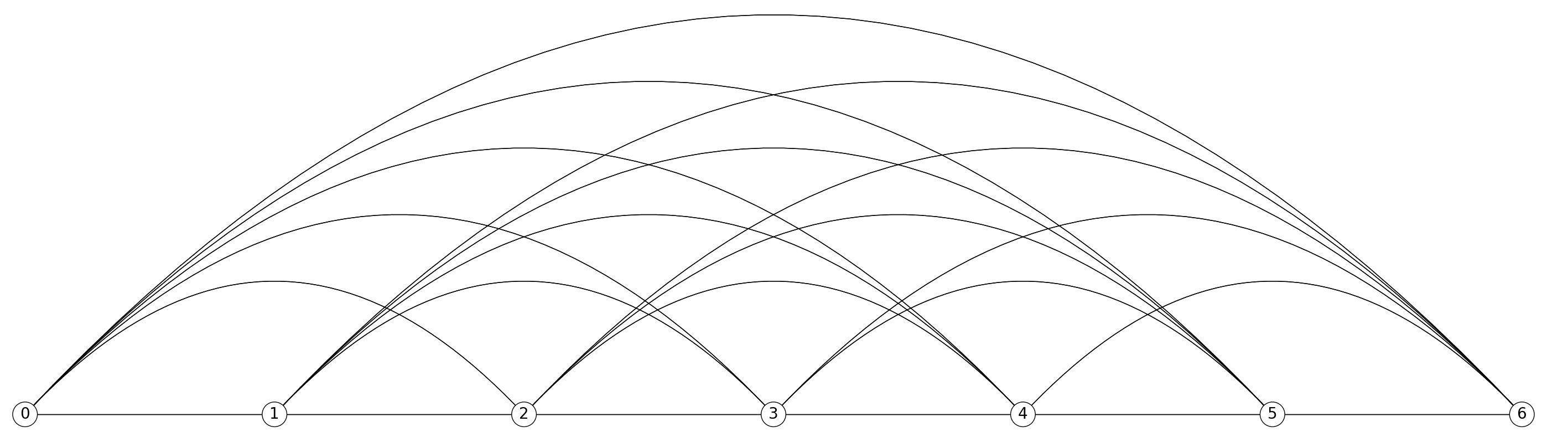
Graph containing the Hamiltonian cycle found via the backtracking algorithm



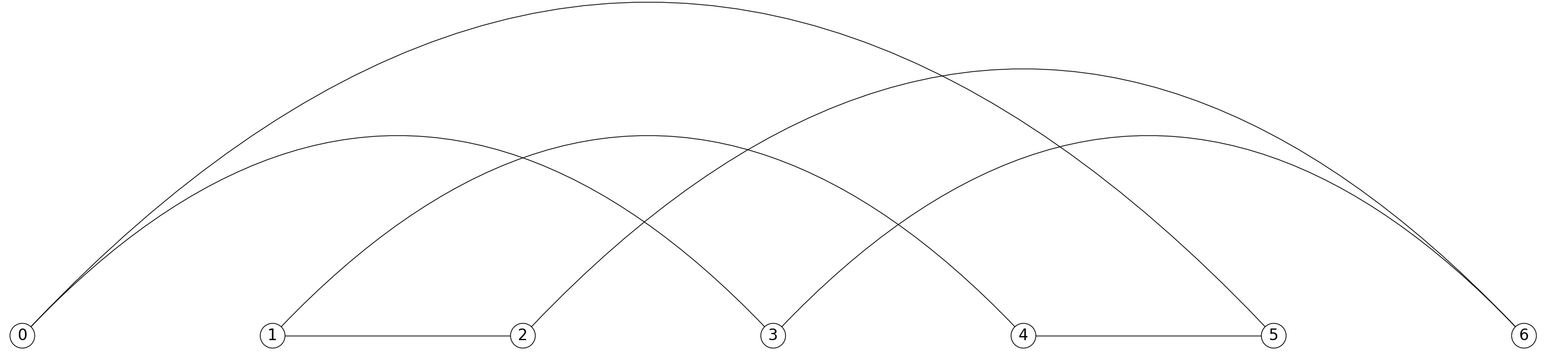
Case 4: Number of edges in the graph = 21



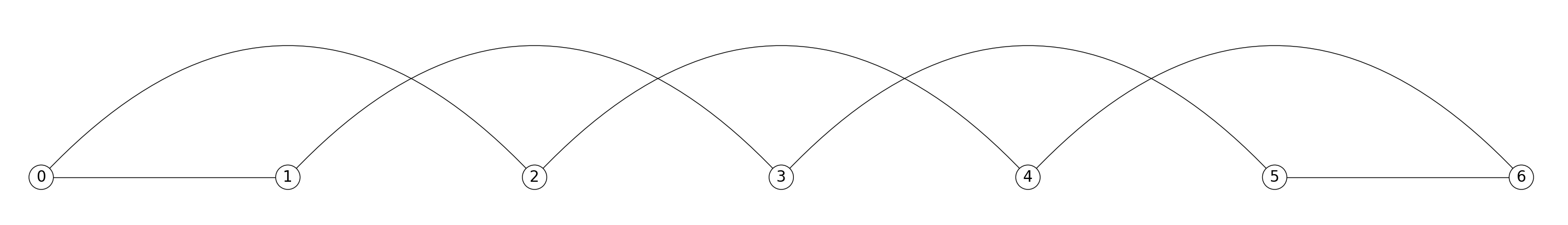
Original graph containing its 7 vertices and 21 edges



Graph containing the Hamiltonian cycle found via the randomization algorithm



Graph containing the Hamiltonian cycle found via the backtracking algorithm

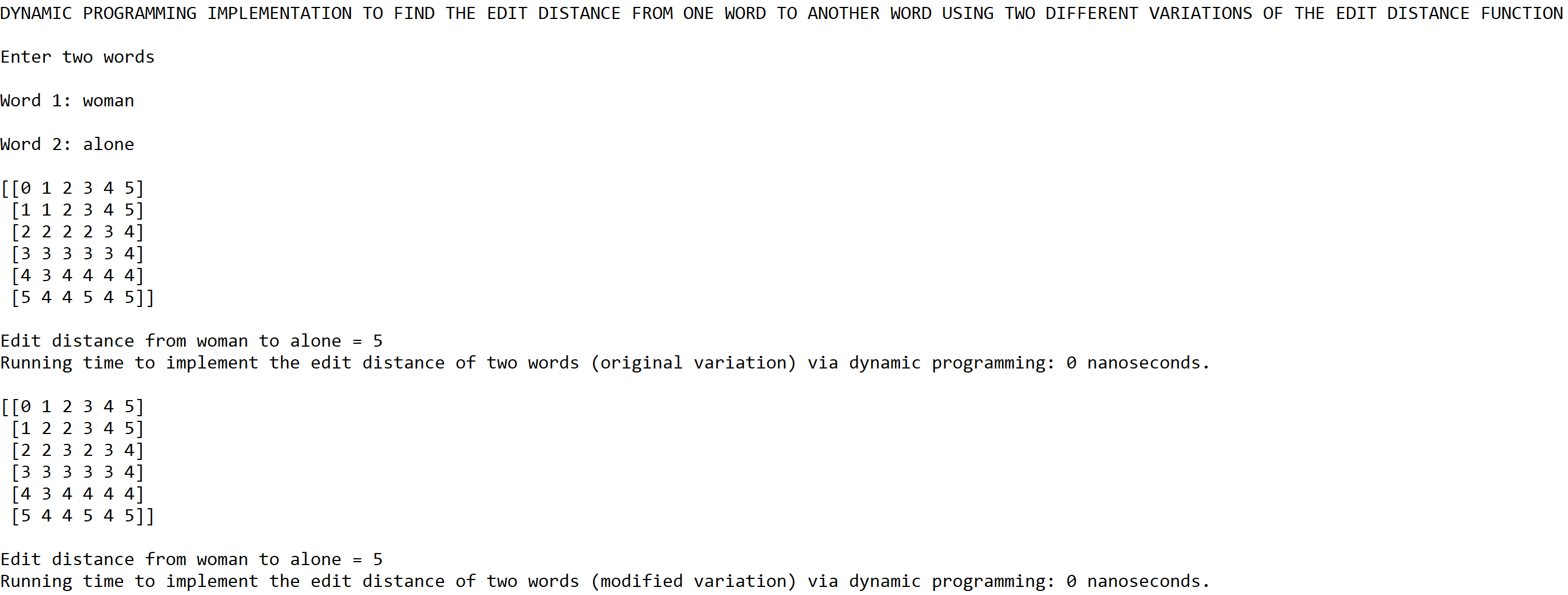


**Dynamic programming algorithms implemented to find the edit distance between two words using two variations of the edit distance function**

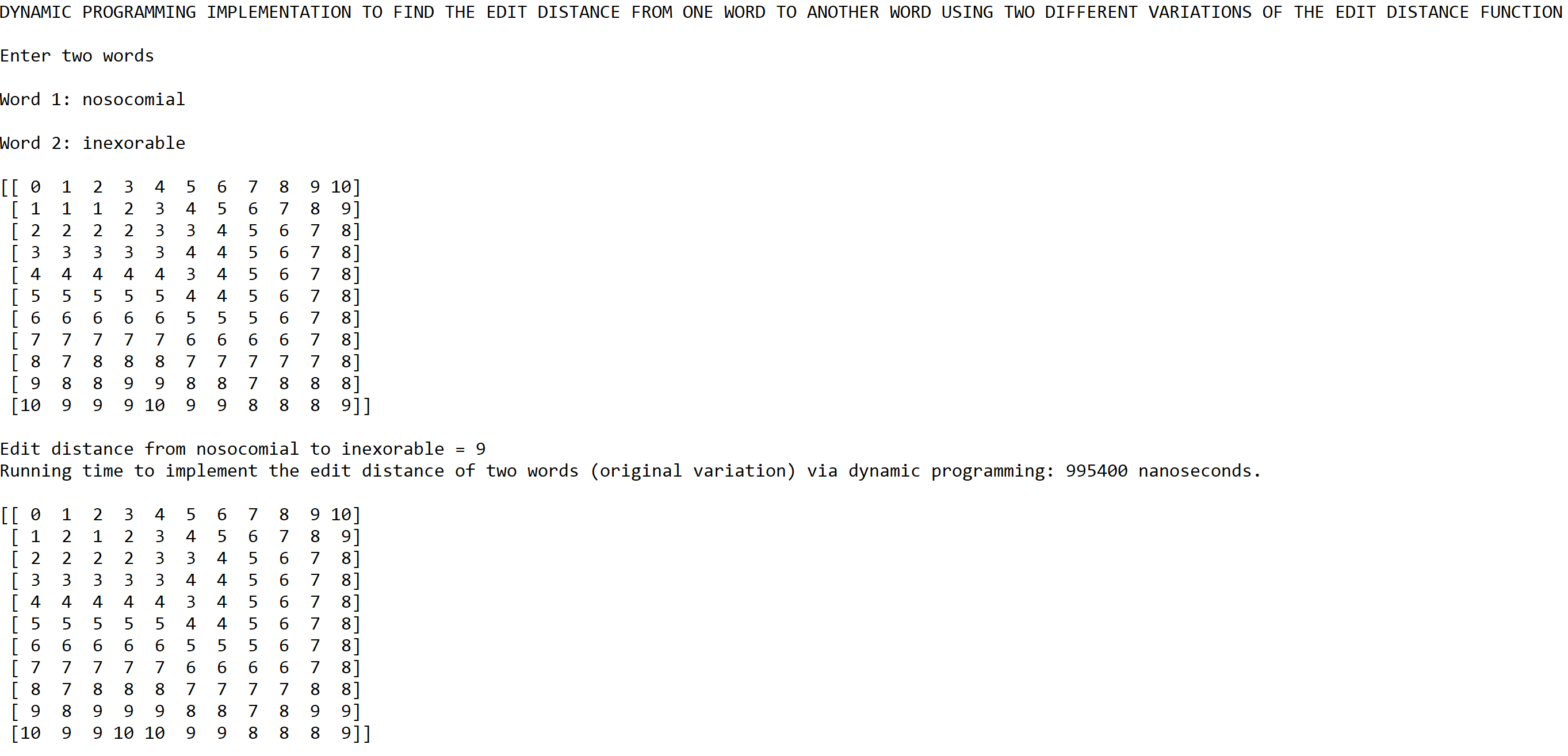
**Operations #3 and #4:**

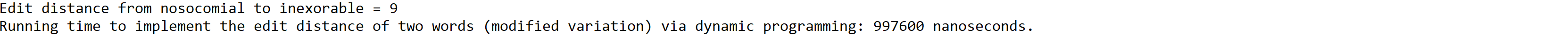
For the remaining two operations, I will test words of lengths 5, 10, 15, and 20. Additionally, I will test words in which the majority of the characters can be easily replaced in the case of Operation #3, but in the case of Operation #4, replacements are much more limited.

Case 1: Length of the words = 5; String 1 = woman; String 2 = alone

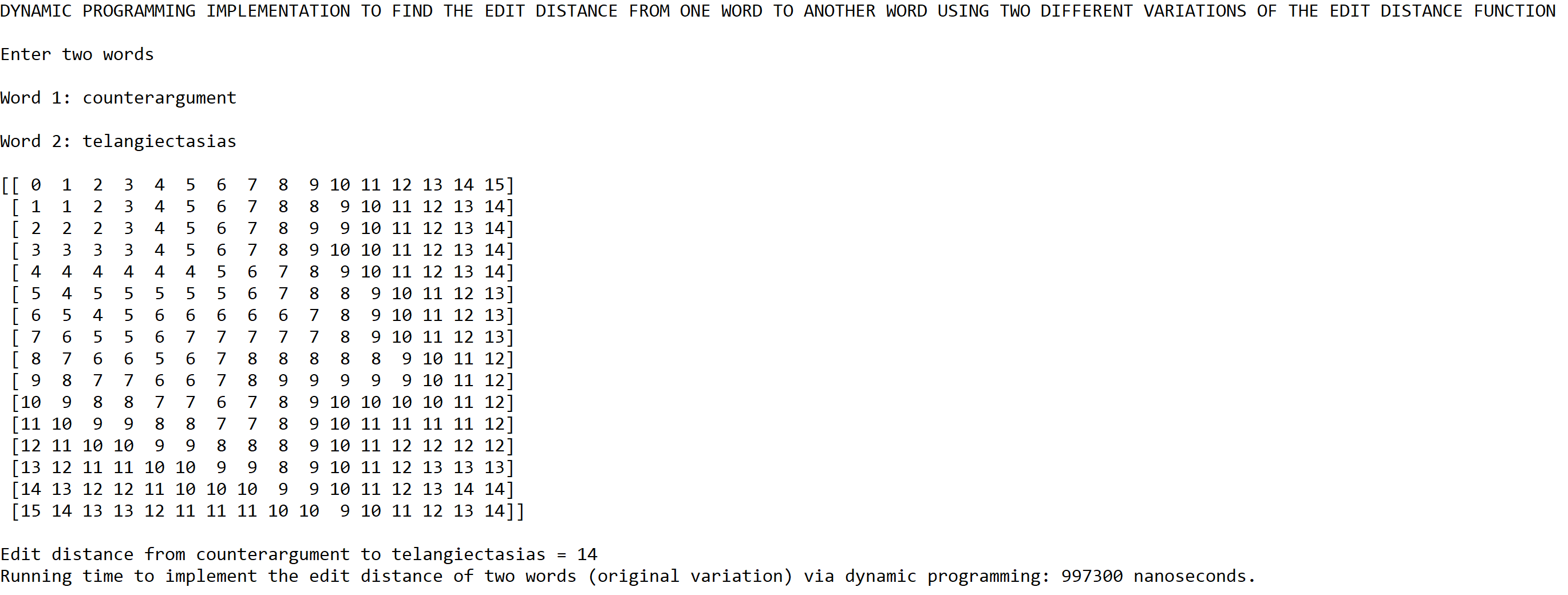


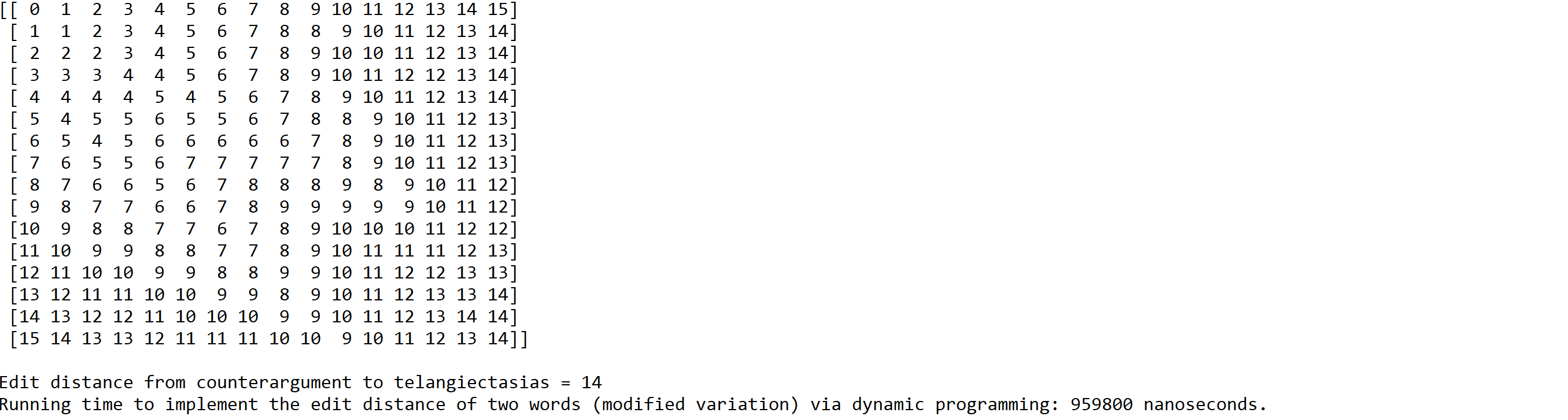
Case 2: Length of the words = 10; String 1 = nosocomial; String 2 = inexorable





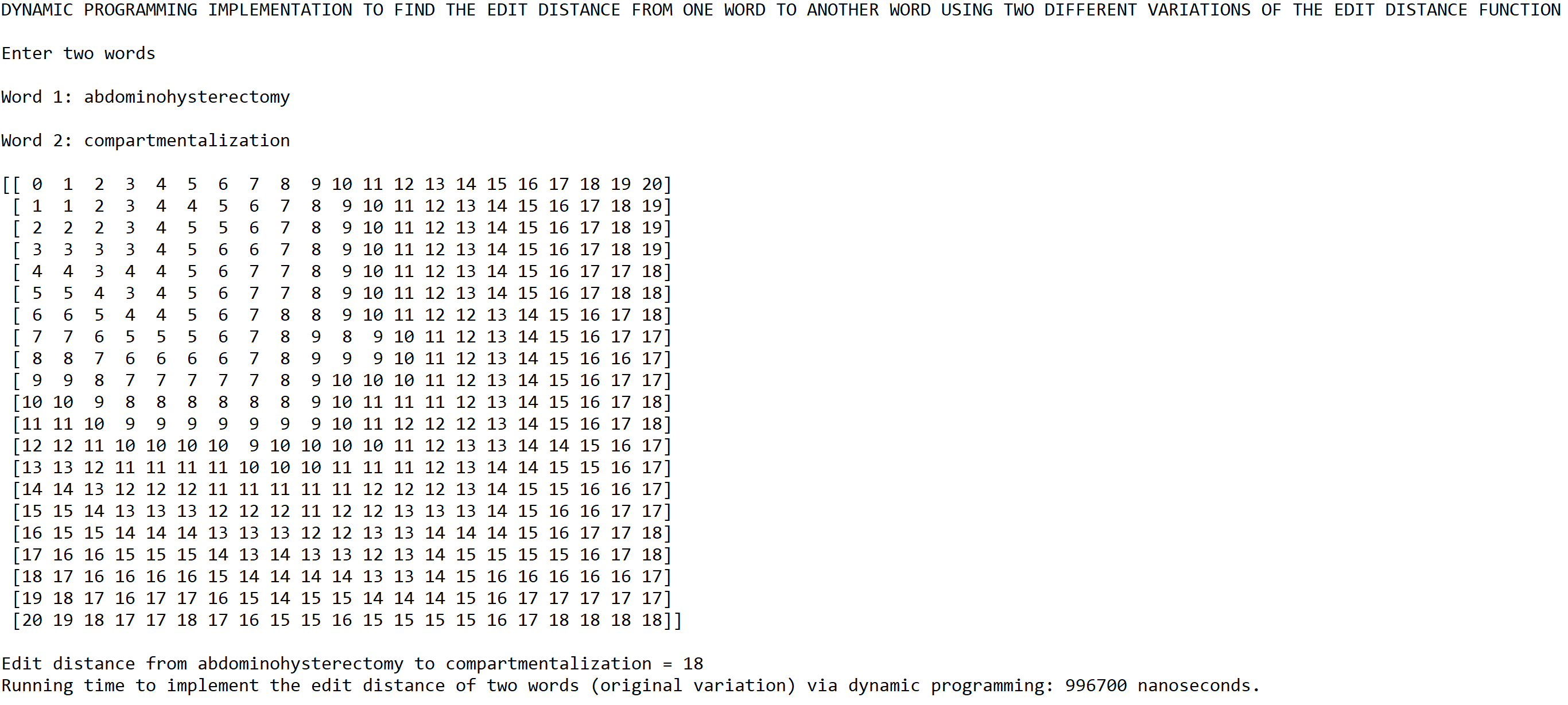
Case 3: Length of the words = 15; String 1 = counterargument; String 2 = telangiectasias

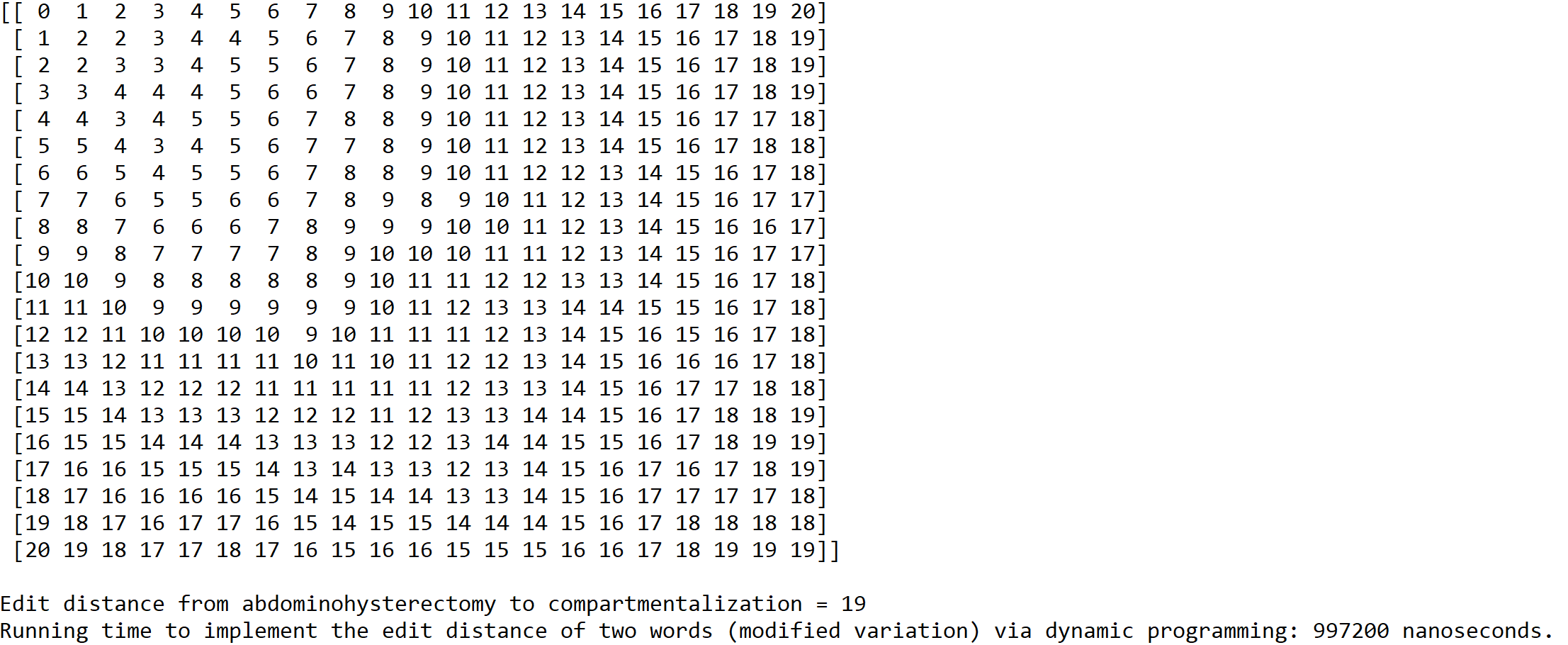




Case 4: Length of the words = 20; String 1 = abdominohysterectomy;

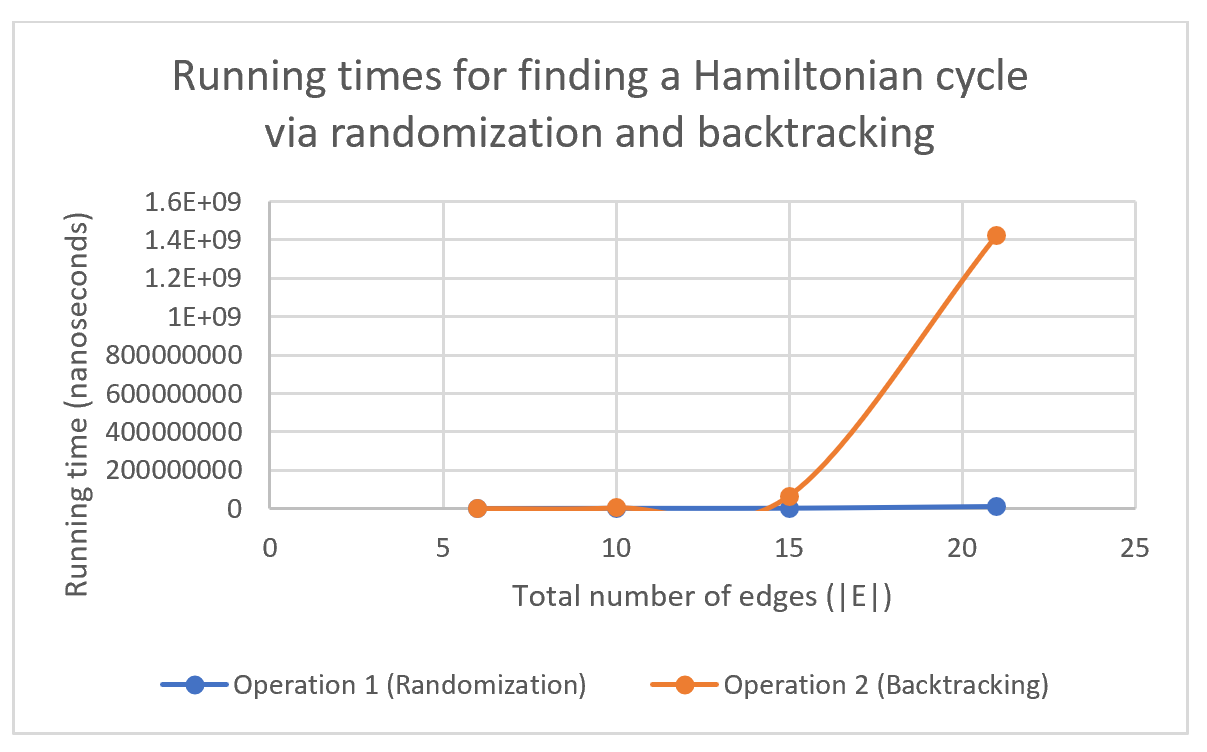
String 2 = compartmentalization





**Graph of running times for performance of randomization and backtracking algorithms in finding a Hamiltonian cycle in an undirected graph**

N = Number of edges(6, 10, 15, 21)



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 6 edges | 10 edges | 15 edges | 21 edges |
| Randomization | 0 | 993400 | 1999700 | 13917800 |
| Backtracking | 0 | 3991100 | 65425200 | 1420920400 |

**Graph of running times for performance of two variations of the same dynamic programming algorithm in finding the edit distance between two words**

N = Size of both words(5, 10, 15, 20)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 5 characters | 10 characters | 15 characters | 20 characters |
| Original variation of edit distance function | 0 | 995400 | 997300 | 996700 |
| Modified variation of edit distance function | 0 | 997600 | 959800 | 997200 |

As the results show, there is a large difference in performance when it comes to finding a Hamiltonian cycle in an undirected graph via randomization and backtracking. In general, the randomized algorithm takes much less time than the backtracking algorithm to find a subset of the set of edges of the graph that forms a Hamiltonian cycle. This is because the randomized algorithm does not need to check every subset possible in order to find one that forms a Hamiltonian cycle. With that being said, it is likely that the randomized algorithm will have a difficult time finding a random subset of the set of edges that forms a Hamiltonian cycle in a graph that contains a large number of edges such as 30, 40, or 50 edges because this algorithm only has about thousands of chances to generate one valid subset out of 2|E| possible subsets, where |E| is the number of edges in the graph. Therefore, the randomized algorithm may be efficient in terms of time, but it is not as efficient when it comes to finding Hamiltonian cycles in larger graphs unless the number of chances is increased to accommodate for the increased number of possible subsets, which is mainly determined by the number of edges in the graph.

On the other hand, the backtracking algorithm takes much more time to find a subset of the set of edges of the graph that forms a Hamiltonian cycle because it checks for every possible subset until it finds one that forms a Hamiltonian cycle. However, the backtracking algorithm has the advantage of always returning a subset that forms a Hamiltonian cycle while the randomized algorithm only has a chance of returning said subset. Additionally, the backtracking algorithm will always return the same subset of the set of edges that forms a Hamiltonian cycle while the randomized algorithm could return a different subset every time the program runs. This is especially noticeable in complete graphs Kn, where it is likely that there will be more than one Hamiltonian cycle in the graph. Regardless, the backtracking algorithm is still considered to be less efficient than the randomized algorithm because it will need to check for every single subset of the set of edges in a specific order before it gets to the one subset out of 2|E| subsets that forms a Hamiltonian cycle. Overall, both algorithms serve their intended purpose of finding a subset of the set of edges that forms a Hamiltonian cycle, but they run at different times in order to find said subset.

In the case of the edit distance algorithm, the running times for both variations are similar to each other even though the modified variation is designed to make more comparisons than the original variation before it assigns a value to each element of the edit distance table, including the value of the edit distance at the bottom-right corner of the table. Additionally, the edit distance between two words across most cases is the same for both variations, except for the last case (i.e. the length of both words is 20) where the edit distance for the modified variation is greater than that of the original variation. With that being said, the running times for both variations would not change drastically if other words were used. However, the edit distance for both variations would be different if other words were used because the vowels and consonants across both words would determine if two characters can be replaced depending on the algorithm implemented. Additionally, the original variation is more straightforward than the modified variation when it comes to converting the first word into the second word because it allows the direct replacement of two characters, regardless of whether they are vowels or consonants. Meanwhile, the modified variation takes a much less obvious approach to exchange characters around in order to determine the edit distance between two words. Overall, both variations of the edit distance algorithm work similarly to find the edit distance between two words, yet words of different sizes should be tested in order to continue comparing the running times and edit distance of both variations.

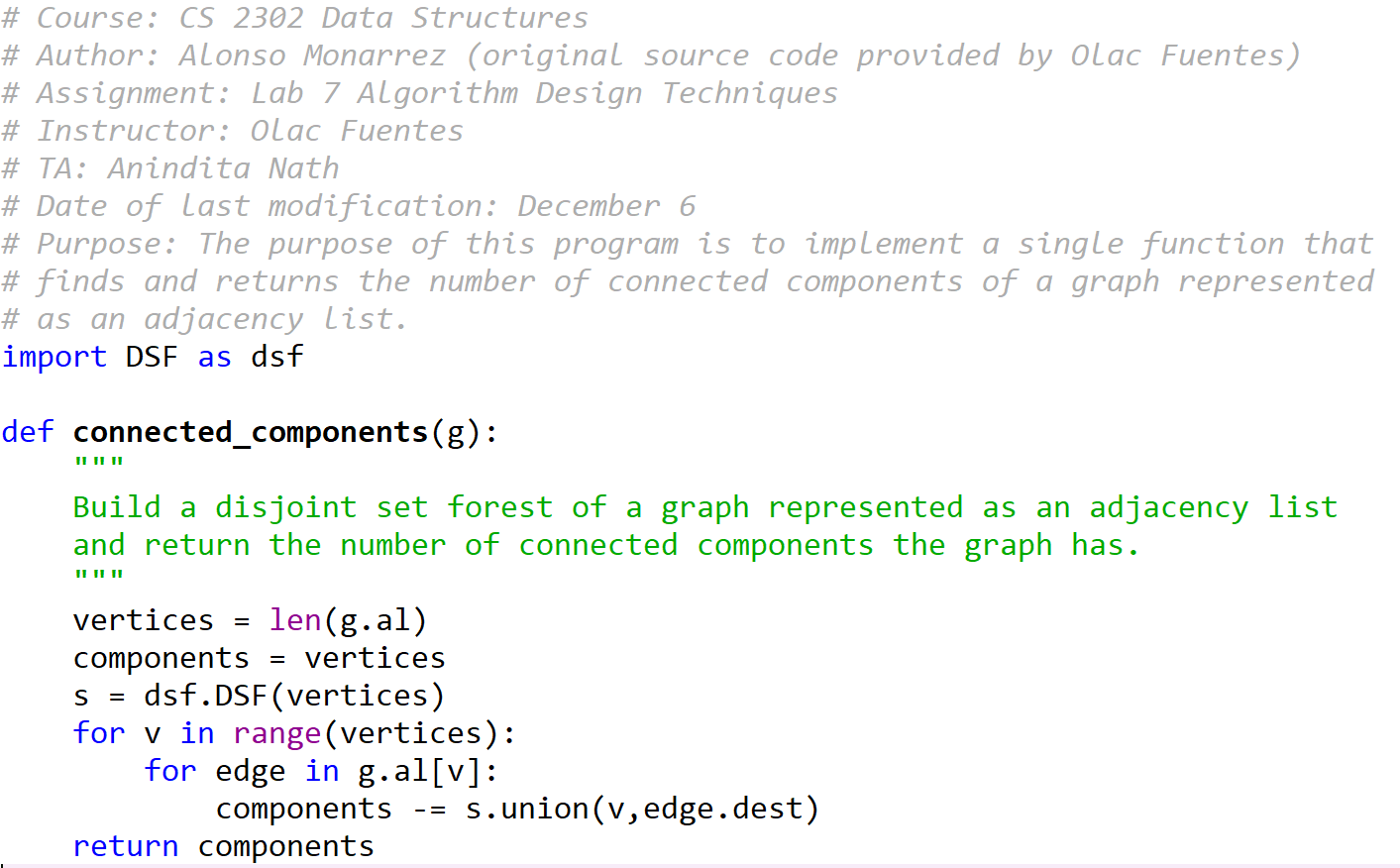
**Conclusion**

This lab helped me understand backtracking better by allowing me to trace the steps this algorithm design techniques takes to solve a problem that involves finding a subset of a set that meets a goal. That way, I can see the different subsets of the original set that are built to find the subset or subsets that meet the goal. For example, I started with the original empty subset of the set of edges of the graph and took the first edge of the set of edges. Afterwards, I followed the scenario in which I did not take the first edge of the set of edges. I continued this process until I found a subset of the set of edges that was both of size |V| and valid subset that forms a Hamiltonian cycle. Once I understood this process, I was able to implement it into a function that generates all possible subsets of the set of edges until it finds one that has both conditions to meet the ultimate goal of a subset that forms a Hamiltonian cycle. Overall, this lab helped me understand backtracking better by allowing me to analyze the process one step at a time.

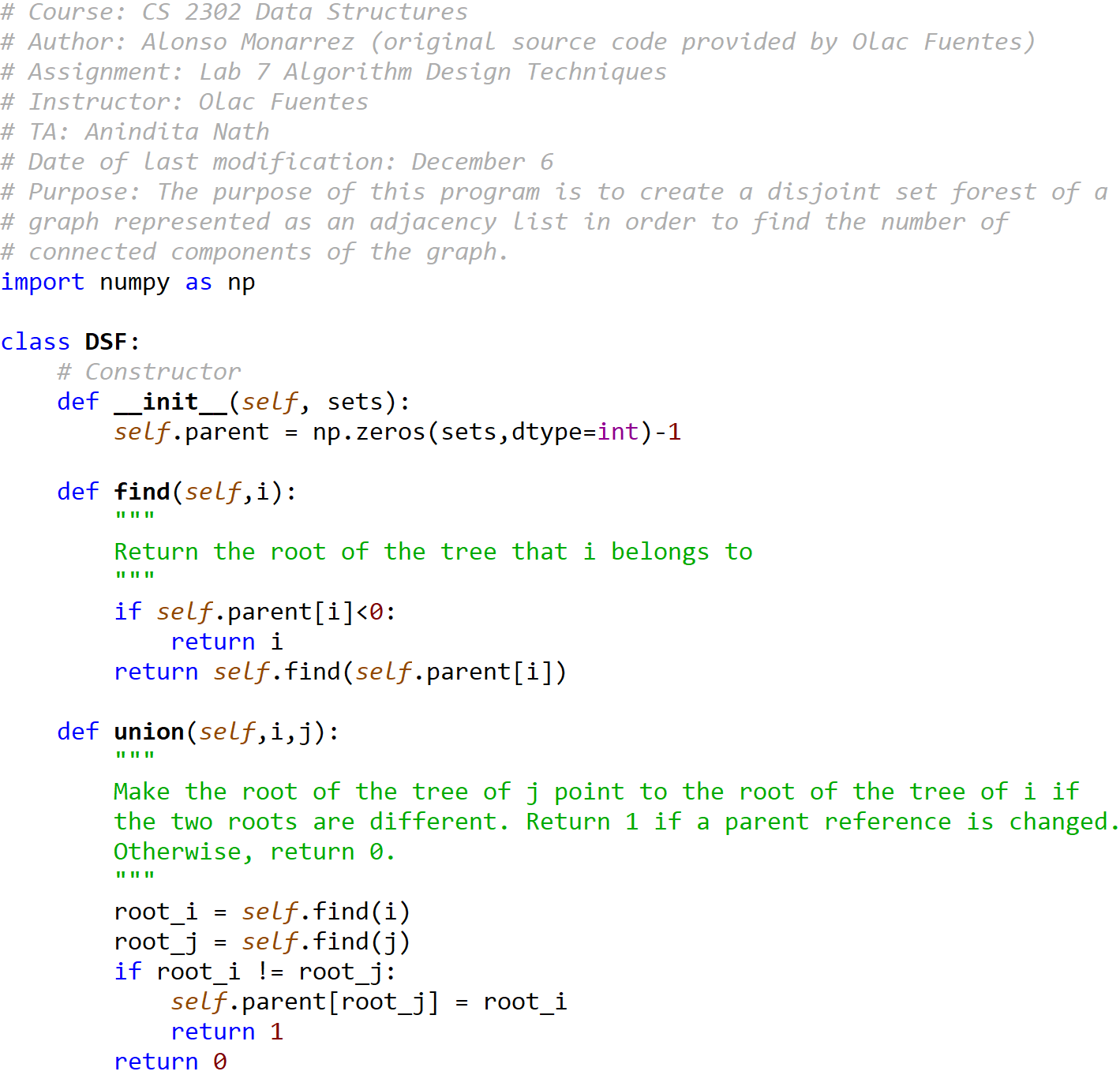
Additionally, this lab helped me understand dynamic programming better by allowing me to analyze the steps the edit distance algorithm takes to assign the value of each element in the edit distance table, including the edit distance itself. For example, I noticed that each of the three elements around the current element corresponds to a different character operation that can be done to convert the first word into the second word. Once I noticed that feature of the edit distance algorithm, I was able to modify the edit distance algorithm in such a way that it compares two characters of both words and determines if they are both vowels and consonants in order to apply the correct character operations. This means that if two characters being compared are not of the same type (i.e. one is a vowel, and the other is a consonant), then the first character will not be replaced by the second character. Instead, the first character will be deleted, and the second character will be inserted in its place. Overall, this lab helped me understand dynamic programming better by allowing me to analyze the purpose of each section of code added to the algorithm implemented via dynamic programming.

**Appendix**

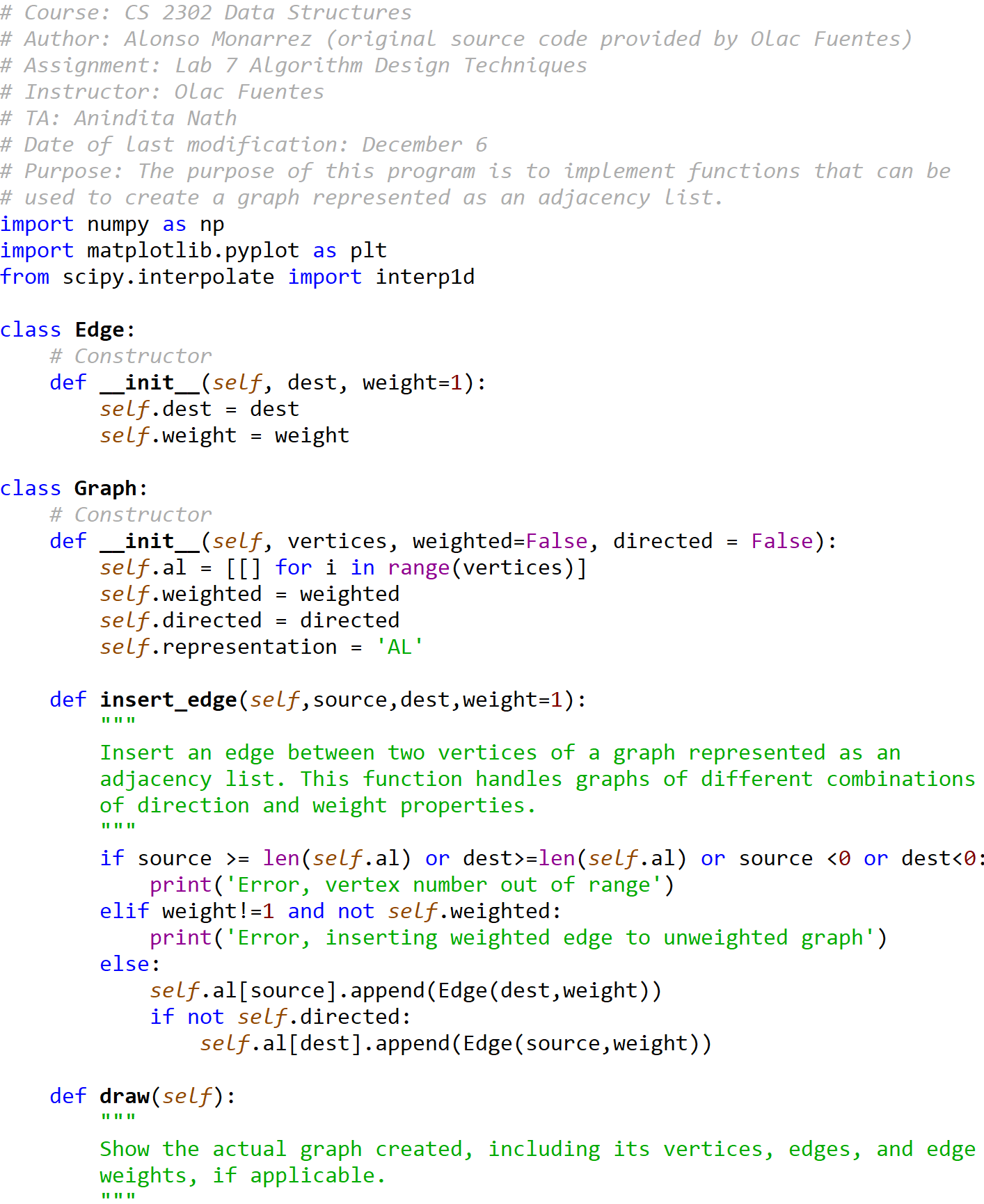
connected\_components.py

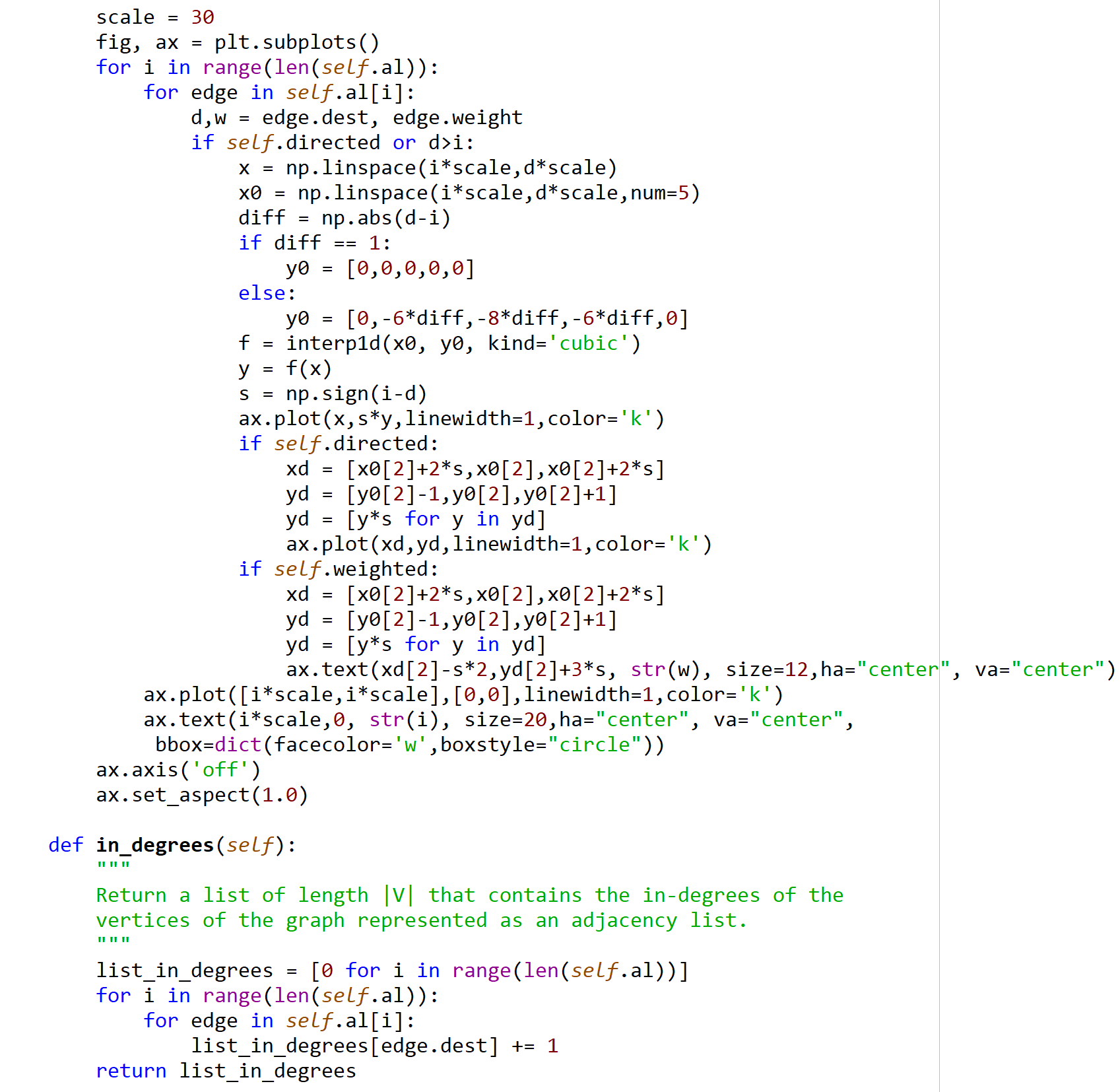


DSF.py

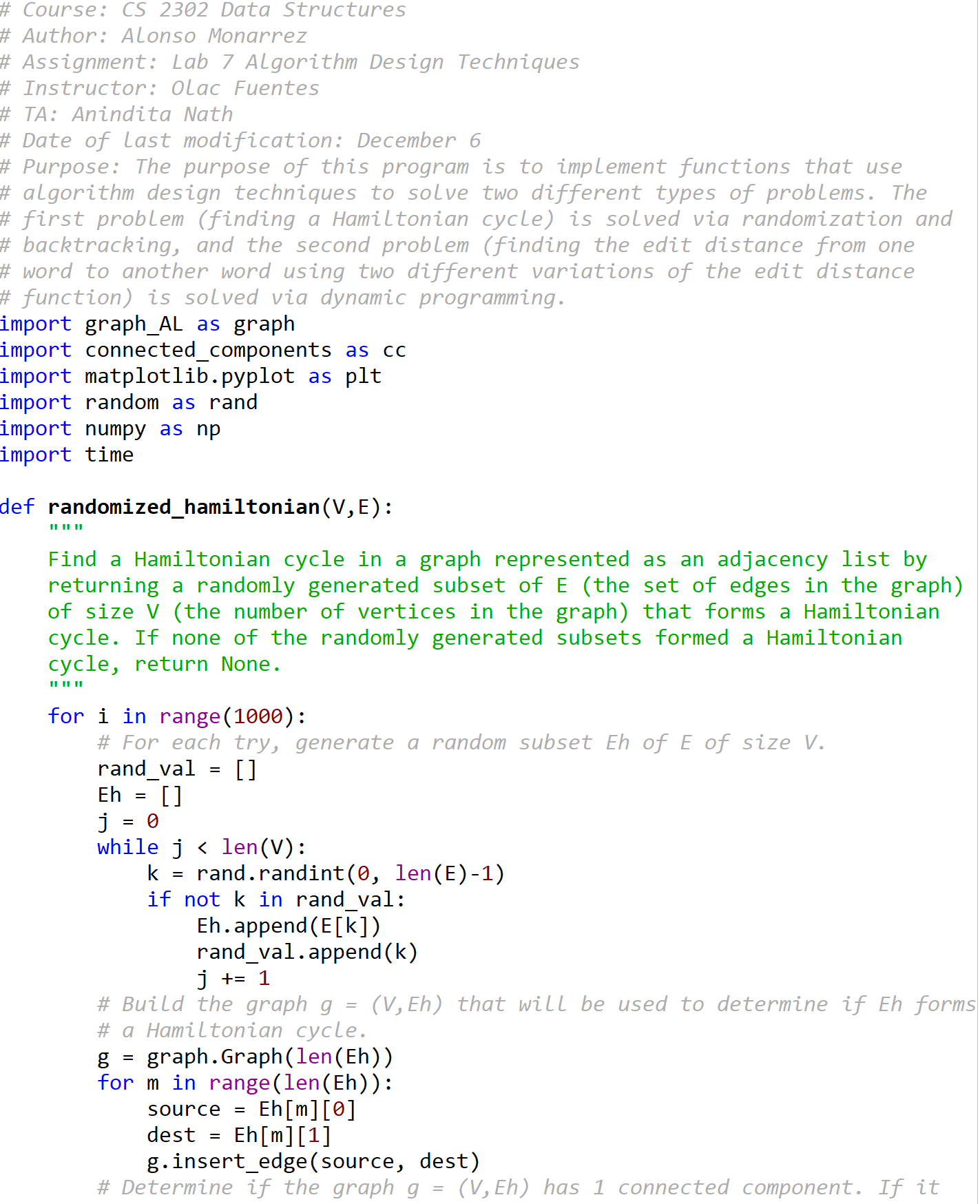


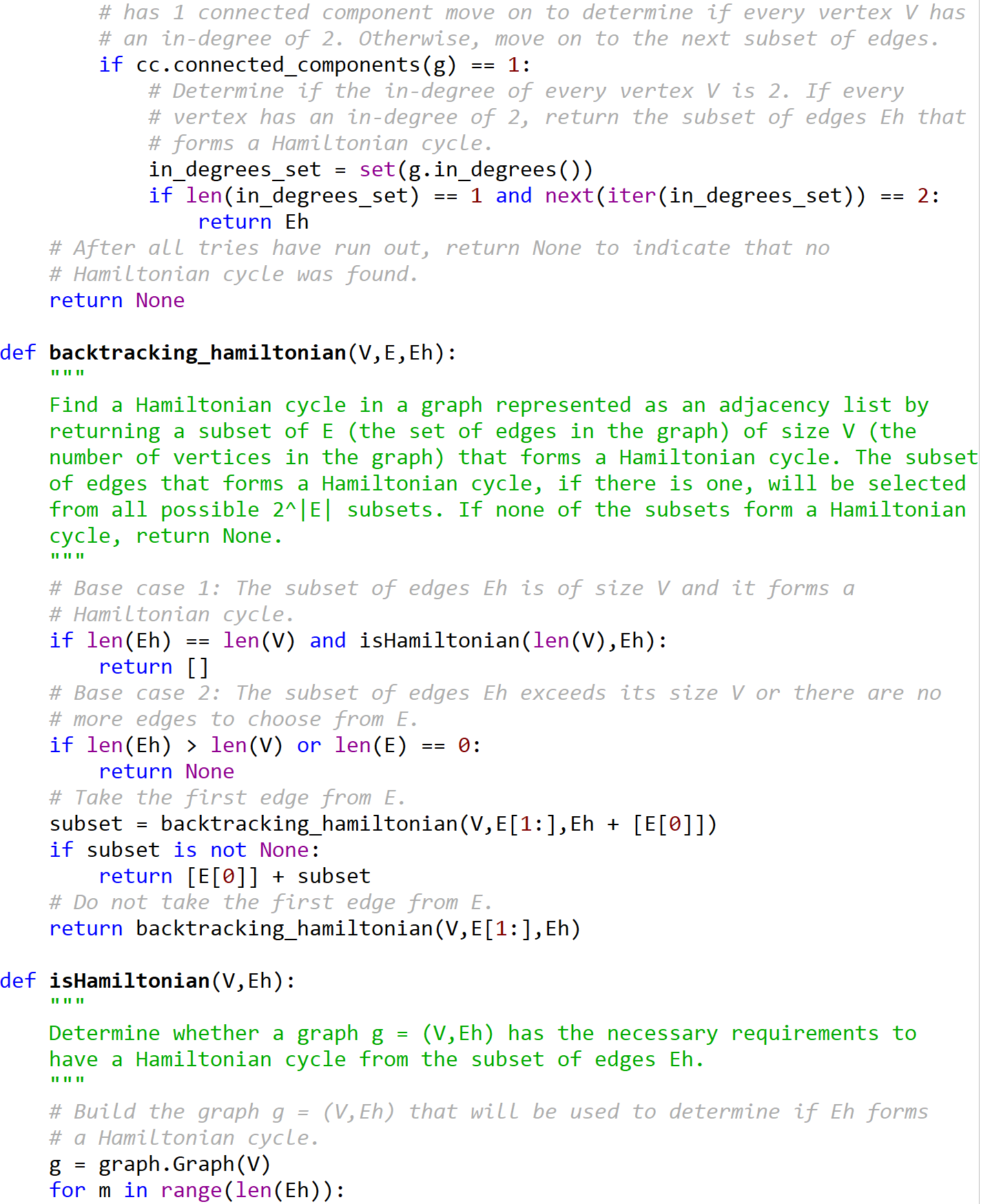
graph\_AL.py

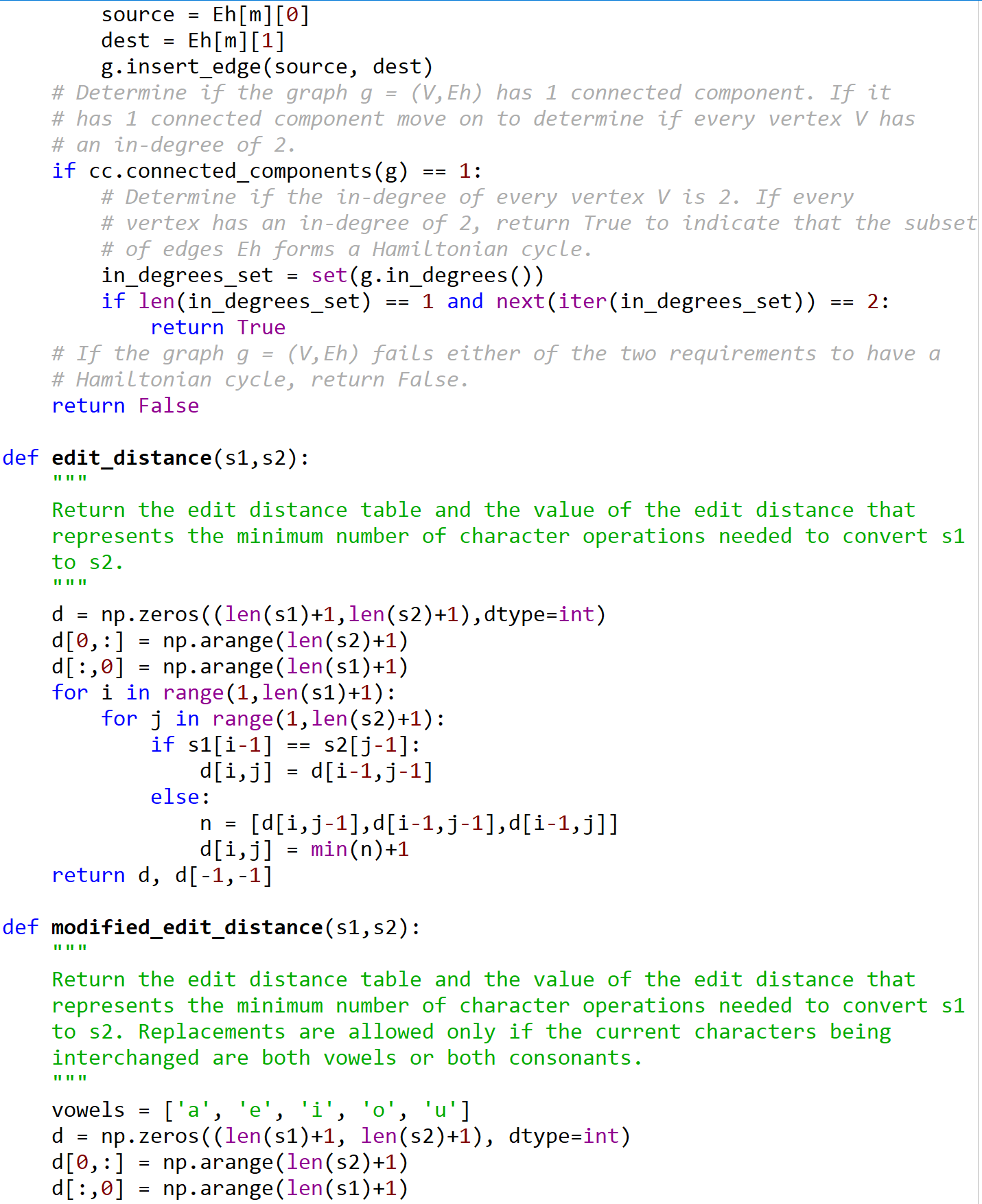


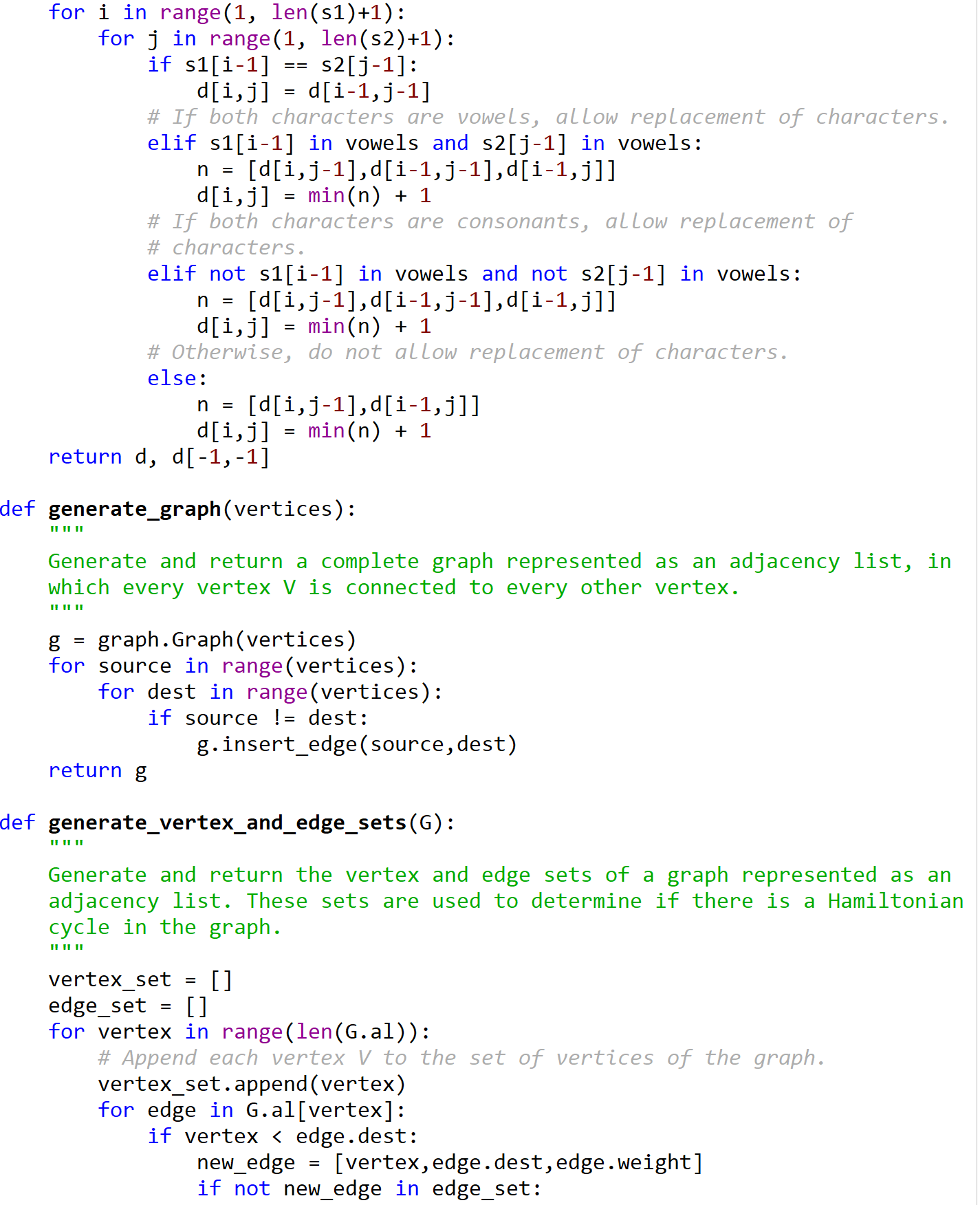


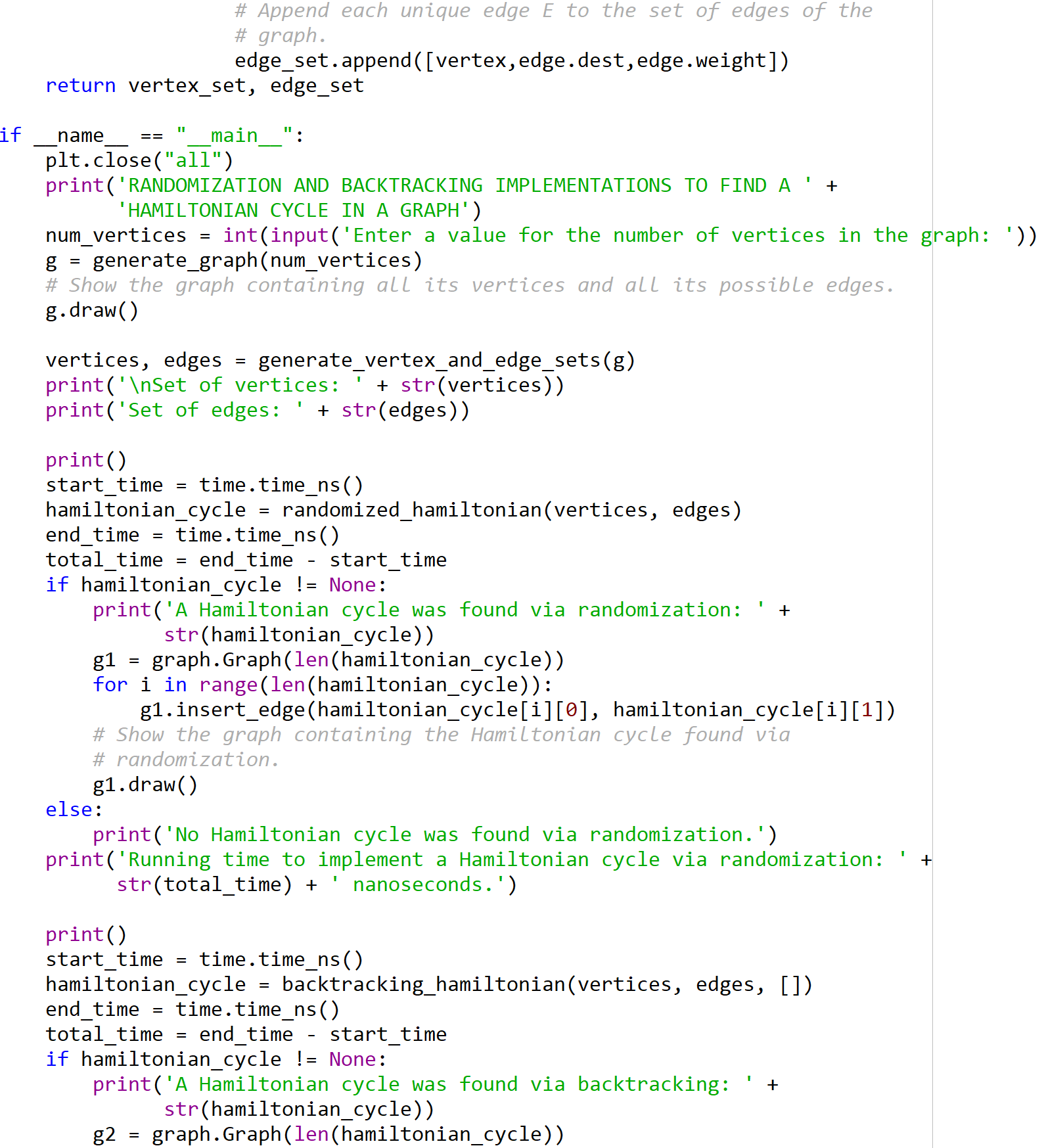
lab7\_algorithm\_design\_techniques.py

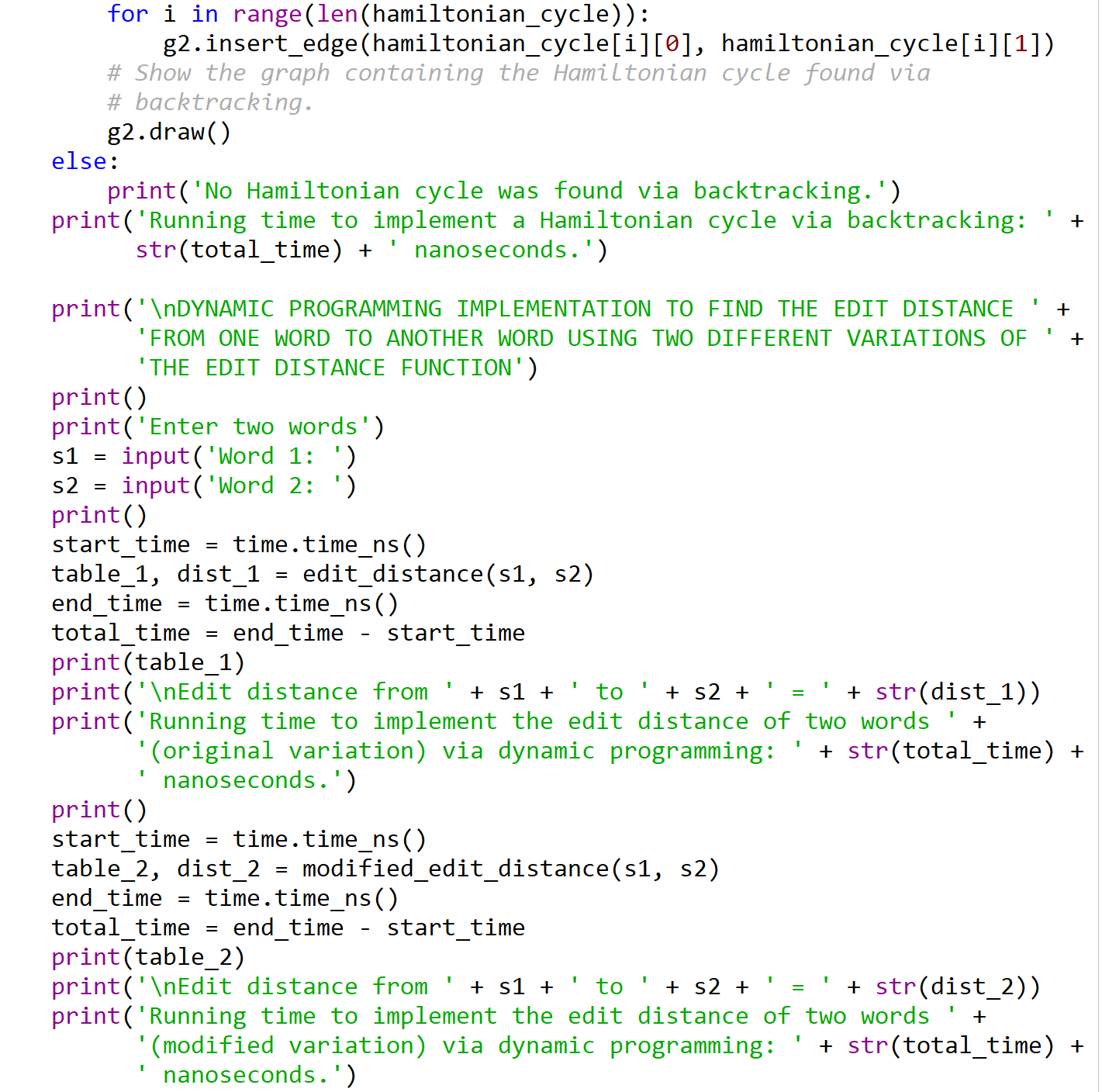












I certify that this project is entirely my own work. I wrote, debugged, and tested the code being presented, performed the experiments, and wrote the report. I also certify that I did not share my code or report or provided inappropriate assistance to any student in the class.