AFRICAN INSTITUTE FOR MATHEMATICAL SCIENCES

(AIMS RWANDA, KIGALI)

Name: Sittana Osman Afifi Mohamedelmubarak ch 3
Zero Knowledge proof Date: May 21, 2020

In the previous chapter, we gave an introduction on Zero-knowledge proofs and Graph isomorphism based on zero-knowledge. In this chapter, we present an implementation of Graph isomorphisms based on zero-knowledge using Python for the original interaction between the prover and the verifier and for the simulator S that we mentioned in chapter 2.
In addition, we show different scenarios for the prover and the verifier and examine the corresponding outputs. According to our code, we suppose that the graphs we use are undirected and so the adjacency matrices are symmetric.

1 Implementation

According to the protocol that we discussed in the previous chapter, the input is two graphs and the output is a list containing: $H = \sigma(G_0)$, ch, φ and 'Accept' or 'Reject'.

1.1 Packages we import

- Math.
- Numpy.
- Graph-tools.
- Sympy.combinatorics.
- Csv.
- Networkx.
- Matplotlib.pyplot.

1.2 Functions we use

- **generate-graph**
  "generate-graph" function enables us to transform the adjacency matrix into a graph using graph-tools package.
- **compose-permutation**
  "compose-permutation" function lets us compose two permutations (p, q) and return $p \circ q$.

- **apply-permute**
  "apply-permute" function enables us to apply a permutation P on adjacency matrix AM and return another adjacency matrix $P(AM)$.

- **inv**
  "inv" function takes a permutation P as an input and returns the inverse of P (P^{-1}).

- **are-equal**
  "are-equal" function checks if two adjacency matrices are equal or not.

- **honest-prover**
  "honest-prover" function applies the protocol honestly for prover's side: It takes two adjacency matrices to represent two corresponding graphs (G_0, G_1), the secret Π, seed s to generate random permutation, and $mess - list$ as an output for the whole protocol to update it during interaction with the verifier.

- **honest-verifier** "honest-verifier" function applies the protocol honestly for the verifier's side: It takes two adjacency matrices to represent two corresponding graphs (G_0, G_1), and $mess - list$ to update it during interaction with the prover.

- **graph-isomorphism**
  "graph-isomorphism" function enables us to run the protocol between the prover and the verifier by controlling the turn of each part.

- **test-isomorphism**
  "test-isomorphism" function enables us to run "graph-isomorpism" by pass "honest-prover" and "honest-verifier" as a parameter to it.

- **cheating-prover**
  "cheating-prover" function applies the protocol for cheating prover who doesn't know the secret Π, it takes two adjacency matrices to represent two corresponding graphs (G_0, G_1), seed s to generate random permutation, and $mess - list$ as an output for the whole protocol to update it during interaction with the verifier.

- **protocol-dishonest-prover**
  "protocol-dishonest-prover" function enables us to run "graph-isomorpism" by passing "cheating-prover" and "honest-verifier" as a parameter to it.

- **simulator**
  "simulator" function enables us to applies the protocol for simulator S that we mentioned in chapter 2.

- **get-graph-from-file**
  "get-graph-from-file" lets us take a graph from csv file.

- **get-pi-from-file**
  "get-pi-from-file" lets us take a secret Π from csv file.

- **equal**
  "equal" function is used after "get-graph-from-file" function to remove all additional data.

- **plot-graph**
  "plot-graph" function uses networkx package to plot graphs using the corresponding adjacency matrix.

1.3 Simulations and result

In this section we show different scenarios and their results. Firstly Figure 1 shows the csv file that contains our data:

```
'0101101001011010'
'0011001111001100'
'0312'
'1001011110100101'
'011111110110111101101101110010111011000011110100011111101011100001010111000010100110010100100011100 0'
'011001011010111101101101011111011011000001010000111111000110001000011111100110101110111101001010101 0'
'8150729436'
```

Figure 1: csv file

We have four cases, three of them using the original protocol and the last case using simulator, each one of them has two possible cases; which are **ch=0** (the fourth component is always Accept) and **ch=1**(the fourth component is different from one case to another):

- **Case 1: Interaction between honest prover and honest verifier**
  First we get the adjacency matrices and Π from a csv file and delete any additional data as follows:

```
In [17]: M=get_graph_from_file(0)
         AM=equal(M)
         M1=get_graph_from_file(1)
         AM1=equal(M1)
         pi=get_pi_from_file(2)
```

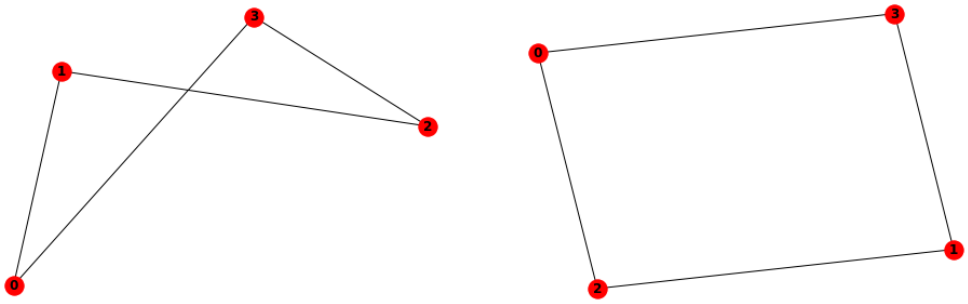
Figure 2: Read the data from csv file

Then we plot G_0 and G_1, after that run the protocol when **ch=0** using code in Figure 3:

```
In [105]: print('G_0 is')
          plot_graph(AM)
          print('G_1 is')
          plot_graph(AM1)
          test_isomorphism(AM,AM1,pi)
```

Figure 3: Runing the protocol

The first output is printing the two graphs shown by Figure 4:



(a) G_0. (b) G_1.

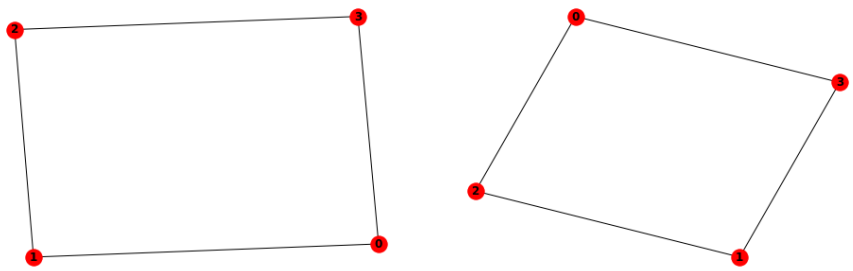
Figure 4: G_0 and G_1 case 1 with $ch = 0$

Let us see the situation when the verifier chooses **ch=0**:

```
Out[105]: [// Generated by graph-tools (version 1.0) at 2020/54/05/09/20 22:54:52
           // undirected, 4 vertices, 4 edges
           graph export_dot {
             node [color=gray90,style=filled];
             "0";
             "1";
             "2";
             "3";
             "0" -- "1";
             "0" -- "3";
             "1" -- "2";
             "2" -- "3";
           }, 0, array([3, 0, 1, 2]), 'Accept']
```

Figure 5: $mess - list$ when $ch = 0$

As we mentioned earlier the fourth component is Accept.
For **ch=1**, the first output is printing the two graphs shown by Figure 6:



(a) G_0. (b) G_1.

Figure 6: case 1, G_0 and G_1 with $ch = 1$

Let us see the situation when the verifier chooses **ch=1**:

```
Out[109]: [// Generated by graph-tools (version 1.0) at 2020/55/05/09/20 22:55:21
           // undirected, 4 vertices, 4 edges
           graph export_dot {
             node [color=gray90,style=filled];
             "0";
             "1";
             "2";
             "3";
             "0" -- "1";
             "0" -- "3";
             "1" -- "2";
             "2" -- "3";
           }, 1, [0, 2, 1, 3], 'Accept']
```

Figure 7: $mess - list$ when $ch = 1$

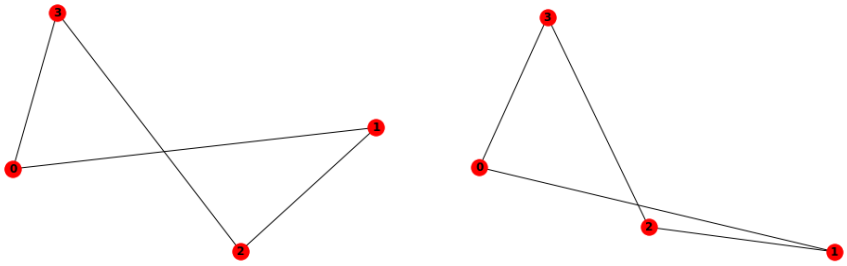
Since **ch=1** and the prover is honest, then the fourth element is Accept:

- **Case 2: Interaction between cheating prover and honest verifier**
  First, we update the second graph by reading it from csv file using code shown in Figure 8:

```
In [110]: M1=get_graph_from_file(3)
          AM1=equal(M1)
```

Figure 8: Update the second graph

when **ch=0** the first output is printing the two graphs shown by Figure 9:



(a) G_0. (b) G_1.

Figure 9: case 2, G_0 and G_1 with $ch = 0$

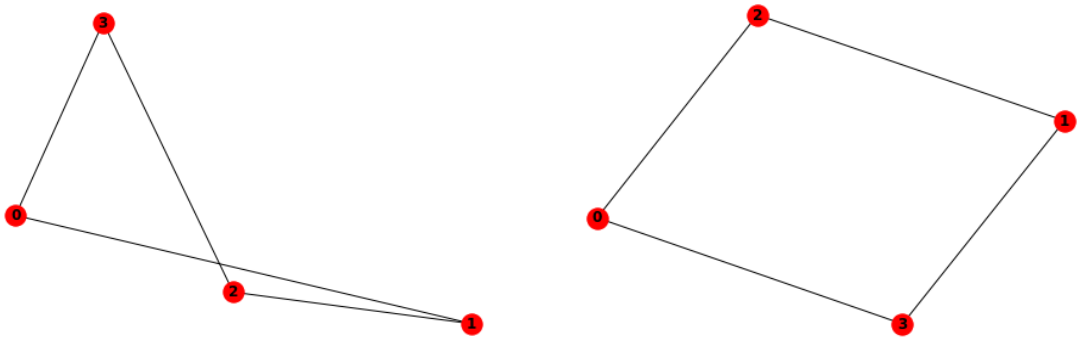
Let us see the situation when the verifier chooses **ch=0**:

```
Out[111]: [// Generated by graph-tools (version 1.0) at 2020/55/05/09/20 22:5
           // undirected, 4 vertices, 4 edges
           graph export_dot {
             node [color=gray90,style=filled];
             "0";
             "1";
             "2";
             "3";
             "0" -- "2";
             "0" -- "3";
             "1" -- "2";
             "1" -- "3";
           }, 0, array([3, 0, 2, 1]), 'Accept']
```

Figure 10: $mess - list$ when $ch = 0$

Since **ch=0** then the fourth component is always Accept even if the prover is dishonest.

When **ch=1**, the first output is the graph of G_0 and G_1 shown by Figure 11:



(a) G_0. (b) G_1.

Figure 11: G_0 and G_1 case 2 with $ch = 1$

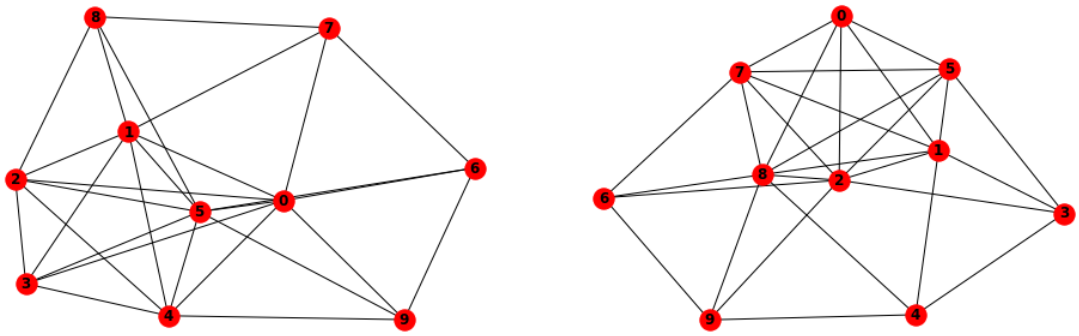
The second output is $mess - list$, the last output is Reject because the prover is dishonest.

- **Case 3: Interaction between honest prover and honest verifier for big graph (v=10,e=28)**
  Now we apply ZKP for big isomorphic graphs that we can not check easily. First we read the data from the file using code shown by Figure 12:

```
In [19]: M=get_graph_from_file(4)
         AM=equal(M)
         M1=get_graph_from_file(5)
         AM1=equal(M1)
         pi=get_pi_from_file(6)
```

Figure 12: updating data and run the protocol

With **ch=0**, the first output is the graph of G_0 and G_1 shown by Figure 13:



(a) G_0. (b) G_1.

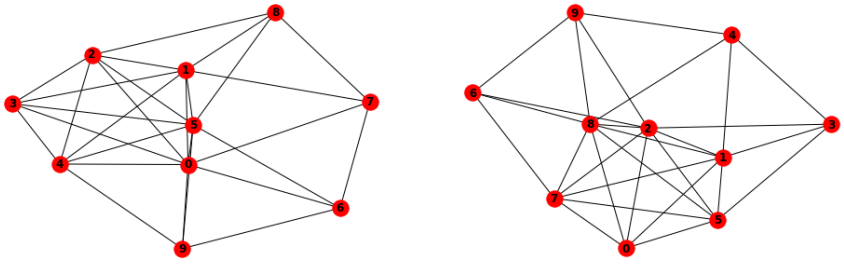
Figure 13: case 3,G_0 and G_1, with $ch = 0$

The second output is $mess-lis$, and the fourth component when $ch = 0$ is always Accept, Figure 14 shows the result:

```
Out[20]: [// Generated by graph-tools (version 1.0) at 2020/47/05/10/20 00:47:17
          // undirected, 10 vertices, 28 edges
          graph export_dot {
            node [color=gray90,style=filled];
            "0";
            "1";
            "2";
            "3";
            "4";
            "5";
            "6";
            "7";
            "8";
            "9";
            "0" -- "1";
            "0" -- "2";
            "0" -- "3";
            "0" -- "4";
            "0" -- "5";
            "0" -- "6";
            "0" -- "8";
            "0" -- "9";
            "1" -- "2";
            "1" -- "4";
            "1" -- "5";
            "1" -- "6";
            "1" -- "7";
            "1" -- "8";
            "1" -- "9";
            "2" -- "3";
            "2" -- "4";
            "3" -- "5";
            "3" -- "7";
            "4" -- "9";
            "5" -- "6";
            "5" -- "7";
            "5" -- "8";
            "5" -- "9";
            "6" -- "8";
            "6" -- "9";
            "7" -- "8";
            "8" -- "9";
          }, 0, array([0, 5, 8, 6, 9, 1, 2, 3, 7, 4]), 'Accept']
```

Figure 14: case 3 with $ch = 0$

When **ch=1**, the first output is the graph of G_0 and G_1 shown by Figure 15:



(a) G_0. (b) G_1.

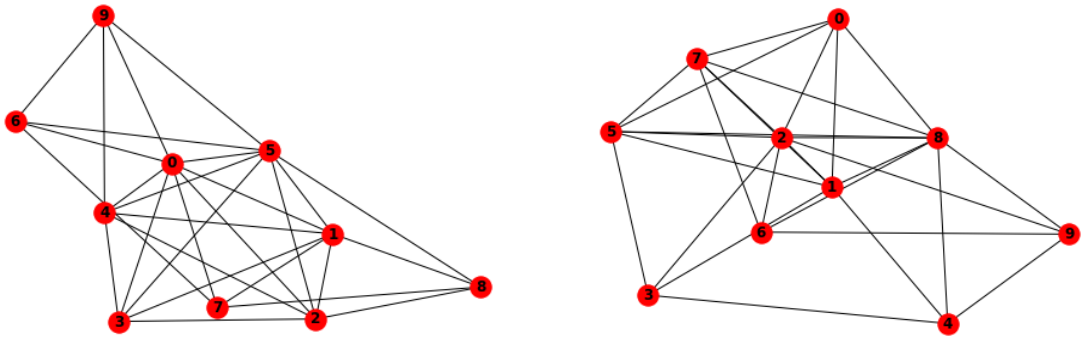
Figure 15: case 3,G_0 and G_1, with $ch = 1$

Let us see the situation when the verifier chooses **ch=1**, Figure 16 shows the result:

```
Out[20]: [// Generated by graph-tools (version 1.0) at 2020/47/05/10/20 00:47:17
          // undirected, 10 vertices, 28 edges
          graph export_dot {
            node [color=gray90,style=filled];
            "0";
            "1";
            "2";
            "3";
            "4";
            "5";
            "6";
            "7";
            "8";
            "9";
            "0" -- "1";
            "0" -- "2";
            "0" -- "3";
            "0" -- "4";
            "0" -- "5";
            "0" -- "6";
            "0" -- "8";
            "0" -- "9";
            "1" -- "2";
            "1" -- "4";
            "1" -- "5";
            "1" -- "6";
            "1" -- "7";
            "1" -- "8";
            "1" -- "9";
            "2" -- "3";
            "2" -- "4";
            "3" -- "5";
            "3" -- "7";
            "4" -- "9";
            "5" -- "6";
            "5" -- "7";
            "5" -- "8";
            "5" -- "9";
            "6" -- "8";
            "6" -- "9";
            "7" -- "8";
            "8" -- "9";
          }, 0, array([0, 5, 8, 6, 9, 1, 2, 3, 7, 4]), 'Accept']
```

Figure 16: G_1 case 3 with $ch = 1$

- **Case 4: Interaction between cheating prover and honest verifier using simulator**
  We use the same data in the previous case,so there is no need to update it,when $ch = 0$ the first output is G_0 and G_1 shown by Figure 17:



(a) G_0. (b) G_1.

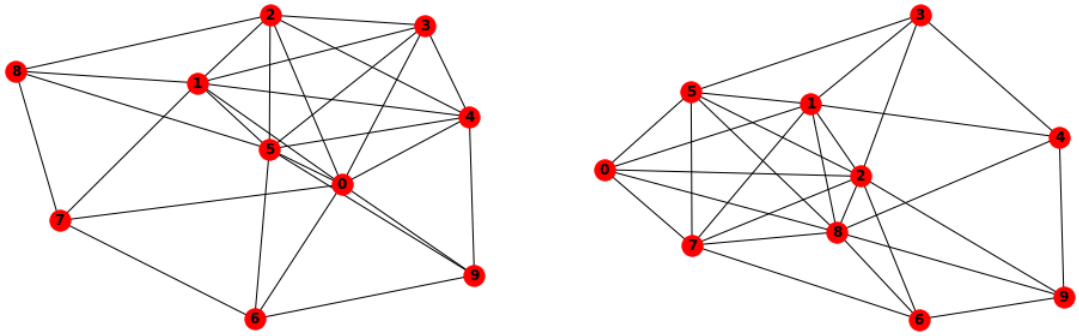
Figure 17: Case 4, G_0 and G_1 case 4 with $ch = 0$

The second output is $mess - list$, and the fourth component when $ch = 0$ is Accept, Figure 18 shows the result:

```
Out[34]: [// Generated by graph-tools (version 1.0) at 2020/56/05/16/20 21:56:44
          // undirected, 10 vertices, 28 edges
          graph export_dot {
            node [color=gray90,style=filled];
            "0";
            "1";
            "2";
            "3";
            "4";
            "5";
            "6";
            "7";
            "8";
            "9";
            "0" -- "1";
            "0" -- "3";
            "0" -- "4";
            "0" -- "7";
            "0" -- "9";
            "1" -- "3";
            "1" -- "4";
            "1" -- "7";
            "1" -- "8";
            "1" -- "9";
            "2" -- "6";
            "2" -- "7";
            "2" -- "8";
            "2" -- "9";
            "3" -- "4";
            "3" -- "5";
            "3" -- "6";
            "3" -- "7";
            "3" -- "8";
            "3" -- "9";
            "4" -- "5";
            "4" -- "7";
            "4" -- "9";
            "5" -- "6";
            "5" -- "9";
            "6" -- "9";
            "7" -- "8";
            "7" -- "9";
          }, 0, array([9, 7, 1, 0, 4, 3, 6, 2, 8, 5]), 'Accept']
```

Figure 18: case 4, $mess - list$ when $ch = 0$

When $ch = 1$ the first output is the graph of G_0 and G_1, shown by Figure 19:



(a) G_0. (b) G_1.

Figure 19: Case 4, G_0 and G_1 with $ch = 1$

The second output is $mess - lis$, and the fourth component when $ch = 1$ is Accept since the prover is honest, Figure 20 shows the result:

```
Out[118]: [// Generated by graph-tools (version 1.0) at 2020/56/05/09/20 22:56:39
           // undirected, 10 vertices, 28 edges
           graph export_dot {
             node [color=gray90,style=filled];
             "0";
             "1";
             "2";
             "3";
             "4";
             "5";
             "6";
             "7";
             "8";
             "9";
             "0" -- "2";
             "0" -- "3";
             "0" -- "5";
             "0" -- "6";
             "0" -- "7";
             "0" -- "9";
             "1" -- "2";
             "1" -- "4";
             "1" -- "7";
             "1" -- "9";
             "2" -- "3";
             "2" -- "5";
             "2" -- "6";
             "2" -- "7";
             "2" -- "8";
             "2" -- "9";
             "3" -- "4";
             "3" -- "5";
             "3" -- "6";
             "3" -- "7";
             "3" -- "8";
             "3" -- "9";
             "4" -- "7";
             "4" -- "8";
             "5" -- "7";
             "5" -- "9";
             "6" -- "8";
             "7" -- "9";
           }, 1, array([5, 7, 2, 1, 4, 9, 6, 0, 3, 8]), 'Accept']
```

Figure 20: $mess - list$ case 4 with $ch = 1$