

EP2 Redes

Christian Rojas Rojas

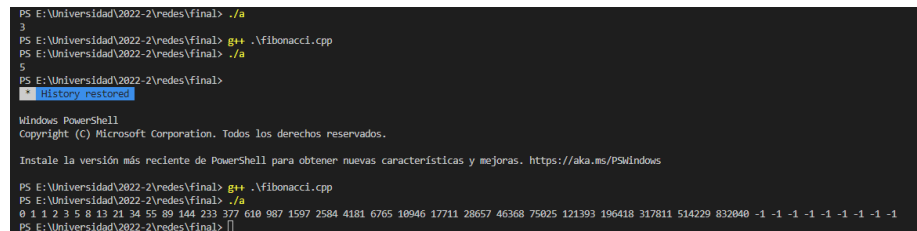
December 2022

1 Introduction

El presente trabajo consta de la implementación de los algoritmos de Dijkstra y Bellman Ford para el análisis de las topologías Torus 1D y Torus 2D; se trabajará con 9 vertices y la construcción de las aristas se harán basadas en la secuencia Fibonacci, la implementación esta construida con el lenguaje C++, posteriormente esto será plasmado usando la herramienta mininet.

2 Preparación de topologías

Para la implementación de ambas topologías lo primero que se realizó fue crear una función de Fibonacci para obtener todos los números que necesitaremos y no realizar esta operación manualmente; en la Figura 1 se observa el resultado de los primeros 30 valores Fibonacci



```
PS E:\Universidad\2022-2\redes\final> ./a
3
PS E:\Universidad\2022-2\redes\final> g++ .\Fibonacci.cpp
PS E:\Universidad\2022-2\redes\final> ./a
5
PS E:\Universidad\2022-2\redes\final>
History restored
Windows PowerShell
Copyright (C) Microsoft Corporation. Todos los derechos reservados.

Instale la versión más reciente de PowerShell para obtener nuevas características y mejoras. https://aka.ms/PSWindows

PS E:\Universidad\2022-2\redes\final> g++ .\Fibonacci.cpp
PS E:\Universidad\2022-2\redes\final> ./a
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946 17711 28657 46368 75025 121393 196418 317811 514229 832040 -1 -1 -1 -1 -1 -1 -1 -1
PS E:\Universidad\2022-2\redes\final> |
```

Figure 1: 30 primeros valores Fibonacci

Usaremos estos valores para setear los valores de nuestras aristas. Cabe mencionar que los valores de -1 no tienen importancia, es a razón de que setee un arreglo mayor a 30 valores

2.1 Torus 1D

La topología Torus de una dimensión es simple y es una buena base para construir Torus de mayores dimensiones.

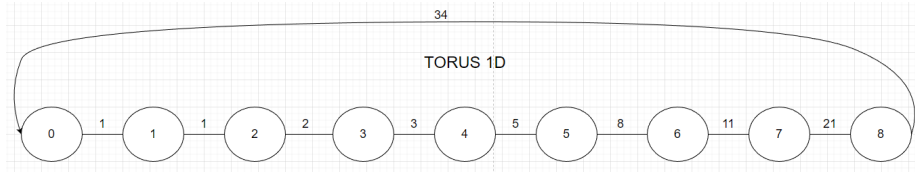


Figure 2: Topología Torus 1D

En la Figura 2 se observa la topología construida; los vertices tienen por nombre números del 0-8 y para el valor de las aristas usamos el resultado de Fibonacci

2.2 Torus 2D

Para la construcción y una mejor explicación de esta topología use 3 pasos

En el primer paso Figura 3 es replicar el Torus 1D con solo 3 vertices

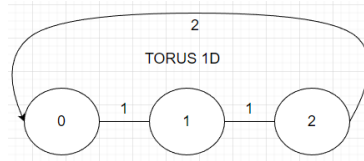


Figure 3: Torus 2D paso 1

El segundo paso Figura 4 es replicar lo del paso uno 3 veces hacia abajo y setear para cada arista valores de Fibonacci consecutivos; este paso sirve para construir valores en sentidos horizontales.

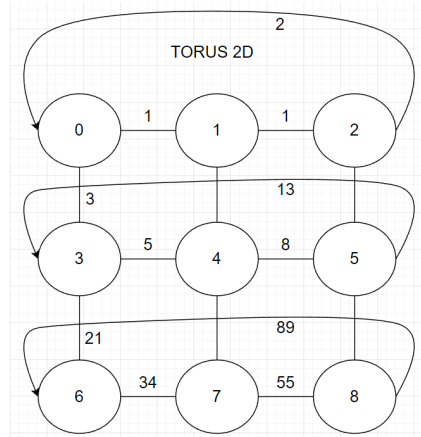


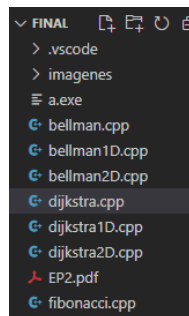
Figure 4: Torus 2D paso 2 - valores horizontales

The diagram illustrates a 2D torus network topology, labeled "TORUS 2D" at the top. It consists of 9 nodes arranged in a 3x3 grid, labeled 0 through 8. The nodes are connected in a grid pattern with wrap-around connections (torus topology). The edges and their weights are as follows:

- Horizontal edges:
 - 0 to 1: 233
 - 1 to 2: 1597
 - 3 to 4: 5
 - 4 to 5: 8
 - 6 to 7: 34
 - 7 to 8: 55
- Vertical edges:
 - 0 to 3: 3
 - 3 to 6: 21
 - 1 to 4: 377
 - 4 to 7: 610
 - 2 to 5: 2584
 - 5 to 8: 4181
- Wrap-around edges:
 - 0 to 2: 2
 - 2 to 0: 144
 - 3 to 5: 13
 - 5 to 3: 6765
 - 6 to 8: 89
 - 8 to 6: 987

Luego de seguir estos pasos finalmente tenemos construido la topología Torus 2D con valores consecutivos de Fibonacci.

Para la construcción de las topologías en los algoritmos, primero se implementaron los algoritmos como se explicará líneas abajo, luego se llamaron a estas funciones creando grafos de a cuerdo a las topologías. El directorio mostrado en la Figura 6



Muestra los archivos `dijkstra.cpp` y `bellman.cpp` los cuales tienen la lógica de los algoritmos; mientras que los otros archivos como `bellman1D.cpp` o di-

jkstra2D.cpp, solo hacen llamados a los algoritmos y construyen su grafo de acuerdo a la dimension de la topología. Use estos últimos archivos si desea ejecutar el código. Ejemplo consola

- g++ dijkstra2D.cpp
- ./a

3.1 Dijkstra

La implementación de Dijkstra consta de 3 funciones:

- **printPath:** Esta es una función recurrente que va imprimiendo cada vertice que se recorrió, usando la lógica de vértice padre
- **printSolution:** Esta función imprime los vertices, las distancias y el path que se usó para obtener esa distancia
- **dijkstra:** Esta es la función principal y se basa en lógica de matriz de adjacencia; en el vector added se va insertando los vértices que van quedando permanentes, en el vector shortestDistances se va guardando las distancias mínimas para cada vértice y en el vector parents se guarda el vértice padre que tiene cada arista. Usando estos principales vectores simplemente el resto del código es la implementación lógica para darle vida al algoritmo.

Luego como vemos en la Figura 7 para ambas topologías solo hacemos la llamada al algoritmo principal y hacemos la construcción correspondiente de la matriz de adjacencia

```

G- dijkstra1D.cpp > ...
1  #include "dijkstra.cpp"
2
3  using namespace std;
4
5  int main(){
6      vector<vector<int>> adjacencyMatrix
7          = { { 0, 1, 0, 0, 0, 0, 0, 0, 34 },
8              { 1, 0, 1, 0, 0, 0, 0, 0, 0 },
9              { 0, 1, 0, 2, 0, 0, 0, 0, 0 },
10             { 0, 0, 2, 0, 3, 0, 0, 0, 0 },
11             { 0, 0, 0, 3, 0, 5, 0, 0, 0 },
12             { 0, 0, 0, 0, 5, 0, 8, 0, 0 },
13             { 0, 0, 0, 0, 0, 8, 0, 11, 0 },
14             { 0, 0, 0, 0, 0, 0, 11, 0, 21 },
15             { 34, 0, 0, 0, 0, 0, 0, 21, 0 } };
16      dijkstra(adjacencyMatrix, 8);
17      return 0;
18  }

```

```

G- dijkstra2D.cpp > main()
1  #include "dijkstra.cpp"
2
3  using namespace std;
4
5  int main(){
6      vector<vector<int>> adjacencyMatrix
7          = { { 0, 233, 2, 3, 0, 0, 144, 0, 0 },
8              { 233, 0, 1597, 0, 377, 0, 0, 987, 0 },
9              { 2, 1597, 0, 0, 0, 2584, 0, 0, 6765 },
10             { 3, 0, 0, 0, 5, 13, 21, 0, 0 },
11             { 0, 377, 0, 5, 0, 8, 0, 610, 0 },
12             { 0, 0, 2584, 13, 8, 0, 0, 0, 4181 },
13             { 144, 0, 0, 21, 0, 0, 0, 34, 89 },
14             { 0, 987, 0, 0, 610, 0, 34, 0, 55 },
15             { 0, 0, 6765, 0, 0, 4181, 89, 55, 0 } };
16      dijkstra(adjacencyMatrix, 8);
17      return 0;
18  }

```

(a) Dijkstra1D

(b) Dijkstra2D

Figure 7: construcción de ambas topologías con Dijkstra

3.2 Bellman Ford

Este algoritmo es bastante usado cuando tenemos aristas dirigidas y valores negativos en algunas de ellas, para nuestro caso al estar usando la secuencia de Fibonacci el cual no tiene valores negativos y usar aristas no dirigidas, probablemente no sacaremos mucho provecho a este algoritmo, esto lo veremos en la parte de resultados. Este algoritmo fue implementado usando solo una función que viene a ser la función principal; esta función recibe como parámetros primero el grafo que es un array doble o matriz, que tiene como segundo parametro [3] de valor estático ya que siempre habrán 3 valores el nodo inicial, el nodo final y el valor de la arista; como siguientes parámetros recibe el número de vértices, número de aristas y el nodo del cual obtendremos sus distancias. Esta implementación consta de 3 bucles for principales; el primero sirve para setear los valores iniciales a infinito de las distancias, el segundo que es un doble bucle for, es el encargado de realizar el análisis de Bellman y el tercero es quien nos dirá si en algún lado se formo un ciclo negativo, el cual haría que nuestro grafo este mal. Como podemos ver en la Figura 8 en ambas topologías solo hacemos la llamadas a la función principal y seteamos los valores de número de vertices, número de aristas y el grafo respectivo



```
bellman1D.cpp > main()
1  #include "bellman.cpp"
2
3  using namespace std;
4
5  int main()
6  {
7      int V = 9;
8      int E = 18;
9
10     int graph[][3]
11     = {{ 0, 1, 1 }, { 0, 8, 34 },
12        { 1, 0, 1 }, { 1, 2, 1 },
13        { 2, 1, 1 }, { 2, 3, 2 },
14        { 3, 2, 2 }, { 3, 4, 3 },
15        { 4, 3, 3 }, { 4, 5, 5 },
16        { 5, 4, 5 }, { 5, 6, 8 },
17        { 6, 5, 8 }, { 6, 7, 11 },
18        { 7, 6, 11 }, { 7, 8, 21 },
19        { 8, 7, 21 }, { 8, 0, 34 } };
20
21     BellmanFord(graph, V, E, 1);
22     return 0;
23 }
```

(a) Bellman1D

```
bellman2D.cpp > main()
1  #include "bellman.cpp"
2
3  using namespace std;
4
5  int main(){
6      int V = 9;
7      int E = 36; //36 = 4*9 = 4 aristas por cada vertice
8
9      int graph[][3]
10     = { { 0, 1, 233 }, { 0, 2, 2 }, { 0, 3, 3 }, { 0, 6, 144 },
11         { 1, 0, 233 }, { 1, 2, 1597 }, { 1, 4, 377 }, { 1, 7, 987 },
12         { 2, 0, 2 }, { 2, 1, 1597 }, { 2, 5, 2584 }, { 2, 8, 6765 },
13         { 3, 0, 3 }, { 3, 4, 5 }, { 3, 5, 13 }, { 3, 6, 21 },
14         { 4, 1, 377 }, { 4, 3, 5 }, { 4, 5, 8 }, { 4, 7, 610 },
15         { 5, 2, 2584 }, { 5, 3, 13 }, { 5, 4, 8 }, { 5, 8, 4181 },
16         { 6, 0, 144 }, { 6, 3, 21 }, { 6, 7, 34 }, { 6, 8, 89 },
17         { 7, 1, 987 }, { 7, 4, 610 }, { 7, 6, 34 }, { 7, 8, 55 },
18         { 8, 2, 6765 }, { 8, 5, 4181 }, { 8, 6, 89 }, { 8, 7, 55 } };
19
20     BellmanFord(graph, V, E, 5);
21     return 0;
22
23     return 0;
24
25 }
```

(b) Bellman2D

Figure 8: construcción de ambas topologías con Bellman Ford

4 Resultados

Para un test inicial y realizar comparaciones en los resultados, haremos un primer análisis tomando como vértice de partida al vértice 0. Posteriormente

a ello se realizará el cálculo para hallar las distancias de: $D_1(9)$, $D_4(6)$, $D_3(7)$, $D_5(6)$. Cabe mencionar que para este caso al ser vértices con nombres del 0 al 8 las distancias a medir serán $D_0(8)$, $D_3(5)$, $D_2(6)$, $D_4(5)$.

4.1 Topología Torus 1D

4.1.1 Análisis y comparación para vértice 0

A continuación en la siguiente imagen se plasman ambos resultados

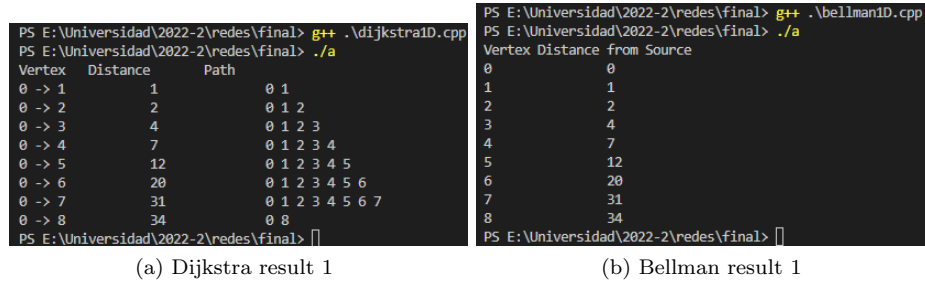


Figure 9: Resultados de algoritmos para Torus 1D

De acuerdo a l Figura 9 podemos ver que ambos algoritmos arrojan los mismos resultados, en el caso de Dijkstra podemos ver el path, pero analizando el resultado de Bellman podemos notar que usa el mismo path que Dijkstra, por lo cual podemos concluir que para la topología Torus 1D ambos algoritmos nos dan el mismo resultado.

4.1.2 resultados de distancias

Vértices	Distancia mínima	Path
$D_0(8)$	34	0, 8
$D_3(5)$	8	3, 4, 5
$D_2(6)$	18	2, 3, 4, 5, 6
$D_4(5)$	5	4, 5

Table 1: Resultado de distancias mínimas - Torus 1D

En la Tabla 1 se muestras las distancias mínimas para cada vértice, en el caso de la topología Torus 1D el path es simple ya que practicamente esta topología solo tiene un camino.

4.2 Topología Torus 2D

4.2.1 Análisis y comparación para vértice 0

A continuación en la siguiente imagen se plasman ambos resultados

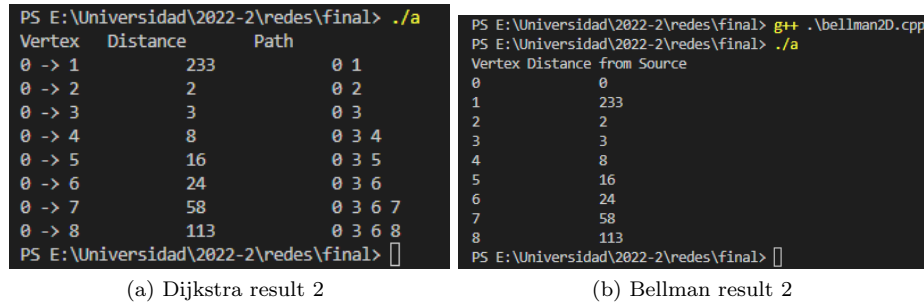


Figure 10: Resultados de algoritmos para Torus 2D

Al igual que en Torus 1D, para el caso del vértice 0 obtuvimos los mismos resultados en ambos algoritmos, podría ser entonces que para esta topología ambos algoritmos también nos den el mismo resultado, esto lo corroboraremos ahora hallando los caminos mínimos de otros vértices

4.2.2 resultados de distancias

Vértices	Distancia mínima	Path
$D_0(8)$	113	0, 3, 6, 8
$D_3(5)$	13	3, 5
$D_2(6)$	26	2, 0, 3, 6
$D_4(5)$	8	4, 5

Table 2: Resultado de distancias mínimas - Torus 2D

De acuerdo a la Tabla 2 se puede corroborar que ciertamente ambos algoritmos calculan las mismas rutas y distancias mínimas para Torus 2D.

En la Figura 11 se muestran los resultados en consola para la consulta de los últimos 2 vectores de la Tabla 2

```

PS E:\Universidad\2022-2\redes\final> g++ .\dijkstra2D.cpp
PS E:\Universidad\2022-2\redes\final> ./a
Vertex Distance Path
2 -> 0 2 2 0
2 -> 1 235 2 0 1
2 -> 3 5 2 0 3
2 -> 4 10 2 0 3 4
2 -> 5 18 2 0 3 5
2 -> 6 26 2 0 3 6
2 -> 7 60 2 0 3 6 7
2 -> 8 115 2 0 3 6 8
PS E:\Universidad\2022-2\redes\final> g++ .\bellman2D.cpp
PS E:\Universidad\2022-2\redes\final> ./a
Vertex Distance from Source
0 2
1 235
2 0
3 5
4 10
5 18
6 26
7 60
8 115
PS E:\Universidad\2022-2\redes\final>

PS E:\Universidad\2022-2\redes\final> g++ .\dijkstra2D.cpp
PS E:\Universidad\2022-2\redes\final> ./a
Vertex Distance Path
4 -> 0 8 4 3 0
4 -> 1 241 4 3 0 1
4 -> 2 10 4 3 0 2
4 -> 3 5 4 3
4 -> 5 8 4 5
4 -> 6 26 4 3 6
4 -> 7 60 4 3 6 7
4 -> 8 115 4 3 6 8
PS E:\Universidad\2022-2\redes\final> g++ .\bellman2D.cpp
PS E:\Universidad\2022-2\redes\final> ./a
Vertex Distance from Source
0 8
1 241
2 10
3 5
4 0
5 8
6 26
7 60
8 115
PS E:\Universidad\2022-2\redes\final>

```

Figure 11: Resultados de ambos algoritmos en consola

5 Topologías con distribución de Gauss

Para esta sección setearémos valores aleatorios primero entre 1 y 10 y luego entre 1 y 100, luego se probarán las topologías con los algoritmos anteriores. Para mantener ambas topologías con 2 aristas por vértice en Torus 1D y 4 aristas por vértice en Torus 2D, setearémos estos valores en la posición respectiva para cada grafo, como se muestra en la Figura 12

```

#define NOD 100
0
1 int main()
2 {
3     srand (time(NULL));
4     vector<int> rand_nums;
5     for (int i=0; i< 9; i++){
6         rand_nums.push_back(rand() %NOD + 1);
7     }
8     int V = 0;
9     int E = 18;
10
11     int graph[E][3]
12     = { { 0, 1, rand_nums[0] }, { 0, 2, rand_nums[1] },
13         { 1, 0, rand_nums[0] }, { 1, 2, rand_nums[2] },
14         { 2, 1, rand_nums[1] }, { 2, 3, rand_nums[3] },
15         { 3, 2, rand_nums[2] }, { 3, 4, rand_nums[4] },
16         { 4, 3, rand_nums[3] }, { 4, 5, rand_nums[5] },
17         { 5, 4, rand_nums[4] }, { 5, 6, rand_nums[6] },
18         { 6, 5, rand_nums[5] }, { 6, 7, rand_nums[7] },
19         { 7, 6, rand_nums[6] }, { 7, 8, rand_nums[8] },
20         { 8, 7, rand_nums[7] }, { 8, 0, rand_nums[1] } };
21
22     Bellmanford(graph, V, E, 0);
23
24     cout << endl << "-----" << endl << endl;
25
26     vector<vector<int>> > adjacencyMatrix;
27     = { { 0, rand_nums[0], 0, 0, 0, 0, 0, 0, 0, rand_nums[1] },
28         { rand_nums[0], 0, rand_nums[2], 0, 0, 0, 0, 0, 0 },
29         { 0, rand_nums[2], 0, rand_nums[3], 0, 0, 0, 0, 0 },
30         { 0, 0, rand_nums[3], 0, rand_nums[4], 0, 0, 0, 0 },
31         { 0, 0, 0, rand_nums[4], 0, rand_nums[5], 0, 0, 0 },
32         { 0, 0, 0, 0, rand_nums[5], 0, rand_nums[6], 0, 0 },
33         { 0, 0, 0, 0, 0, rand_nums[6], 0, rand_nums[7], 0 },
34         { 0, 0, 0, 0, 0, 0, rand_nums[7], 0, rand_nums[8] },
35         { rand_nums[1], 0, 0, 0, 0, 0, 0, 0, rand_nums[1] } };
36     dijkstra(adjacencyMatrix, 0);
37
38     return 0;
39 }
40
41 #define NOD 100
42 0
43 int main()
44 {
45     srand (time(NULL));
46     int V = 9;
47     int E = 36;
48
49     vector<int> rand_nums;
50     for (int i=0; i< 18; i++){
51         rand_nums.push_back(rand() %NOD + 1);
52     }
53
54     int graph[E][3]
55     = { { 0, 1, rand_nums[0] }, { 0, 2, rand_nums[1] }, { 0, 3, rand_nums[2] }, { 0, 6, rand_nums[3] },
56         { 1, 0, rand_nums[0] }, { 1, 2, rand_nums[4] }, { 1, 4, rand_nums[5] }, { 1, 7, rand_nums[6] },
57         { 2, 0, rand_nums[1] }, { 2, 1, rand_nums[4] }, { 2, 5, rand_nums[7] }, { 2, 8, rand_nums[8] },
58         { 3, 0, rand_nums[2] }, { 3, 4, rand_nums[9] }, { 3, 5, rand_nums[10] }, { 3, 6, rand_nums[11] },
59         { 4, 1, rand_nums[5] }, { 4, 3, rand_nums[9] }, { 4, 5, rand_nums[12] }, { 4, 7, rand_nums[13] },
60         { 5, 2, rand_nums[7] }, { 5, 3, rand_nums[10] }, { 5, 4, rand_nums[12] }, { 5, 6, rand_nums[14] },
61         { 6, 0, rand_nums[3] }, { 6, 3, rand_nums[11] }, { 6, 7, rand_nums[15] }, { 6, 8, rand_nums[16] },
62         { 7, 1, rand_nums[6] }, { 7, 4, rand_nums[13] }, { 7, 6, rand_nums[15] }, { 7, 8, rand_nums[17] },
63         { 8, 2, rand_nums[8] }, { 8, 5, rand_nums[14] }, { 8, 6, rand_nums[16] }, { 8, 7, rand_nums[17] } };
64
65     Bellmanford(graph, V, E, 0);
66
67     cout << endl << "-----" << endl << endl;
68
69     vector<vector<int>> > adjacencyMatrix;
70     = { { 0, rand_nums[0], rand_nums[1], rand_nums[2], 0, 0, rand_nums[3], 0, 0 },
71         { rand_nums[0], 0, rand_nums[4], 0, rand_nums[5], 0, 0, rand_nums[6], 0 },
72         { rand_nums[1], rand_nums[4], 0, 0, 0, rand_nums[7], 0, 0, rand_nums[8] },
73         { rand_nums[2], 0, 0, 0, rand_nums[9], rand_nums[10], rand_nums[11], 0, 0 },
74         { 0, rand_nums[5], 0, rand_nums[9], 0, rand_nums[12], 0, rand_nums[13], 0 },
75         { 0, 0, rand_nums[7], rand_nums[10], rand_nums[12], 0, 0, 0, rand_nums[14] },
76         { rand_nums[3], 0, 0, rand_nums[11], 0, 0, 0, rand_nums[15], rand_nums[16] },
77         { 0, rand_nums[6], 0, 0, rand_nums[13], 0, rand_nums[15], 0, rand_nums[17] },
78         { 0, 0, rand_nums[8], 0, 0, rand_nums[16], rand_nums[16], rand_nums[17], 0 } };
79     dijkstra(adjacencyMatrix, 0);
80
81     return 0;
82 }
83

```

Figure 12: Dijkstra y Bellman con valores aleatorios

En la parte inicial del código tenemos un valor definido que viene a ser el módulo, simplemente manipulando el módulo obtendremos valores máximos para nuestros costos; luego tenemos vectores donde guardaremos los valores aleatorios y finalmente estos serán seteados tanto en el grafo de Bellman (parte superior) y en la matriz de Dijkstra (parte inferior)

Se analizaron 2 resultados, donde cada resultado contiene tanto al algoritmo de Bellman como al de Dijkstra, en las imágenes que se mostrarán a continuación se observaran 2 resultados por imagen divididas por una linea vertical; también se observará la impresión de ambos algoritmos por cada resultado, esto dividido por una linea horizontal punteada.

5.1 Distribución de Gauss para Torus 1D

Resultados para topología Torus de una dimensión con algoritmos de Dijkstra y Bellman Ford siguiendo una distribución de Gauss

<pre>PS E:\Universidad\2022-2\redes\final> ./a Vertex Distance from Source 0 0 1 10 2 11 3 15 4 16 5 15 6 8 7 2 8 1 ----- Vertex Distance Path 0 -> 1 10 0 1 0 -> 2 11 0 1 2 0 -> 3 15 0 1 2 3 0 -> 4 16 0 0 7 6 5 4 0 -> 5 15 0 0 7 6 5 0 -> 6 8 0 0 7 6 0 -> 7 2 0 0 7 0 -> 8 1 0 8 PS E:\Universidad\2022-2\redes\final> []</pre>	<pre>PS E:\Universidad\2022-2\redes\final> ./a Vertex Distance from Source 0 0 1 6 2 10 3 12 4 20 5 16 6 9 7 3 8 2 ----- Vertex Distance Path 0 -> 1 6 0 1 0 -> 2 10 0 1 2 0 -> 3 12 0 1 2 3 0 -> 4 20 0 1 2 3 4 0 -> 5 16 0 0 7 6 5 0 -> 6 9 0 0 7 6 0 -> 7 3 0 0 7 0 -> 8 2 0 8 PS E:\Universidad\2022-2\redes\final> []</pre>
--	--

Figure 13: Resultados para costos entre 1 y 10 Torus 1D

<pre>PS E:\Universidad\2022-2\redes\final> ./a Vertex Distance from Source 0 0 1 83 2 175 3 262 4 270 5 227 6 205 7 115 8 85 ----- Vertex Distance Path 0 -> 1 83 0 1 0 -> 2 175 0 1 2 0 -> 3 262 0 1 2 3 0 -> 4 270 0 0 7 6 5 4 0 -> 5 227 0 0 7 6 5 0 -> 6 205 0 0 7 6 0 -> 7 115 0 0 7 0 -> 8 85 0 8 PS E:\Universidad\2022-2\redes\final> []</pre>	<pre>PS E:\Universidad\2022-2\redes\final> ./a Vertex Distance from Source 0 0 1 44 2 46 3 86 4 87 5 89 6 144 7 60 8 22 ----- Vertex Distance Path 0 -> 1 44 0 1 0 -> 2 46 0 1 2 0 -> 3 86 0 1 2 3 0 -> 4 87 0 1 2 3 4 0 -> 5 89 0 1 2 3 4 5 0 -> 6 144 0 0 7 6 0 -> 7 60 0 0 7 0 -> 8 22 0 8 PS E:\Universidad\2022-2\redes\final></pre>
--	---

Figure 14: Resultados para costos entre 1 y 100 Torus 1D

5.2 Distribución de Gauss para Torus 2D

Resultados para topología Torus de dos dimensión con algoritmos de Dijkstra y Bellman Ford siguiendo una distribución de Gauss

<pre>PS E:\Universidad\2022-2\redes\final> ./a Vertex Distance from Source 0 0 1 6 2 4 3 3 4 8 5 6 6 6 7 9 8 7</pre>	<pre>PS E:\Universidad\2022-2\redes\final> ./a Vertex Distance from Source 0 0 1 9 2 8 3 4 4 7 5 10 6 6 7 12 8 13</pre>
<pre>----- Vertex Distance Path 0 -> 1 6 0 2 1 0 -> 2 4 0 2 0 -> 3 3 0 3 0 -> 4 8 0 3 4 0 -> 5 6 0 3 5 0 -> 6 6 0 6 0 -> 7 9 0 6 7 0 -> 8 7 0 6 8</pre>	<pre>----- Vertex Distance Path 0 -> 1 9 0 1 0 -> 2 8 0 2 0 -> 3 4 0 3 0 -> 4 7 0 3 4 0 -> 5 10 0 3 5 0 -> 6 6 0 6 0 -> 7 12 0 1 7 0 -> 8 13 0 2 8</pre>

Figure 15: Resultados para costos entre 1 y 10 Torus 2D

<pre>PS E:\Universidad\2022-2\redes\final> ./a Vertex Distance from Source 0 111 1 80 2 97 3 79 4 0 5 77 6 89 7 52 8 111</pre>	<pre>PS E:\Universidad\2022-2\redes\final> ./a Vertex Distance from Source 0 26 1 56 2 50 3 25 4 0 5 42 6 34 7 51 8 57</pre>
<pre>----- Vertex Distance Path 4 -> 0 111 4 7 1 0 4 -> 1 80 4 7 1 4 -> 2 97 4 7 1 2 4 -> 3 79 4 3 4 -> 5 77 4 5 4 -> 6 89 4 7 6 4 -> 7 52 4 7 4 -> 8 111 4 7 1 2 8</pre>	<pre>----- Vertex Distance Path 4 -> 0 26 4 3 0 4 -> 1 56 4 3 6 7 1 4 -> 2 50 4 3 5 2 4 -> 3 25 4 3 4 -> 5 42 4 3 5 4 -> 6 34 4 3 6 4 -> 7 51 4 3 6 7 4 -> 8 57 4 3 5 2 8</pre>

Figure 16: Resultados para costos entre 1 y 100 Torus 2D

6 Conclusiones

- Tomando como referencia todo el análisis realizado en el presente informe, podemos concluir que en todos nuestros resultados tanto el algoritmo de Dijkstra como el de Bellman Ford nos proporcionaron los mismos resultados, por lo cual podemos decir que para estas topologías y usando las distribuciones de Gauss y Fibonacci, ambos algoritmos calculan la misma distancia mínima y por ende el mismo camino.
- Tomando como referencia las Figuras 13 14 15 y 16 podemos observar que hay una gran diferencia entre las topologías 1D y 2D; en la topología Torus 1D vemos que el resultado de las distancias mínimas excede muchas

veces al valor máximo del costo que puede tener cada arista, mientras que en la topología Torus 2D difícilmente esto sucede.

- Referenciando a lo anterior, finalmente también se puede concluir que la topología Torus 2D es mejor que la topología Torus 1D, con ello podemos decir también que es muy probable que mientras la topología Torus aumenta su dimensionalidad, aumentará su complejidad pero también mejorará su performance.

7 Repositorio

Todo el directorio y códigos de implementación lo podrá clonar desde el siguiente Repositorio <https://github.com/chrisis17/EP2-redes>

8 Mininet

Para esta sección quería mencionar que estuve investigando como poder usar mi implementación y plasmarla en mininet; sin embargo no encontraba solución alguna hasta que encontré el siguiente blog [1] y pude tener una primera observación de lo que encontraría más tarde en la documentación de Mininet [2], el cual sería la gran sorpresa que tanto mininet como el código que se usa con mininet se trabajan en python, por lo cual mi implementación realizada en C++ no podía ser usada; Por lo cual a causa de todo el trabajo realizado ya previamente y más aun por disponer de poco tiempo y sería al parecer imposible rehacer, en el tiempo que dispongo, todo el trabajo en python; ya no pude realizar la implementación en mininet.

References

- [1] <https://mailman.stanford.edu/pipermail/mininet-discuss/2013-November/003319.html>
- [2] <https://github.com/mininet/mininet/wiki/Documentation>
- [3] <https://www.geeksforgeeks.org/bellman-ford-algorithm-dp-23/>
- [4] <https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/>