Examen Final

Anthony Guimarey Saavedra

Diciembre, 2022

Este trabajo consiste en la implementación del algoritmo Vector-Distancia en C++. Para ello, se implementaron dos redes de routers con topología Torus que cuentan con 9 nodos. La topología Torus 1D esta interconectada de modo que cada nodo tiene dos interconexiones. Mientras que en la topología Torus 2D cada nodo tiene 4 interconexiones.

1. Implementación

Se pueden visualizar las topologías Torus 1D y 2D en la Figura 1 y 2, respectivamente. La conexión C_i representa el costo asociado en la red. Los valores asignados siguen la serie de Fibonacci, es decir, $C_1 = 1, C_2 = 1, C_3 = 2, ..., C_9 = 34$. Las conexiones en la topología Torus 2D también siguen los valores de la serie fibonacci, donde C_{ij} representa el costo de un nodo i hacia un nodo j. Los costos asociados son las siguientes:

C_{12}	1
C_{23}	1
C_{31}	2
C_{14}	3
C_{45}	5
C_{56}	8
C_{64}	13
C_{47}	21
C_{78}	34
C_{89}	55
C_{97}	89
C_{71}	144
C_{82}	233
C_{25}	377
C_{58}	610
C_{93}	987
C_{36}	1597
C_{69}	2584

Cuadro 1: Costos en cada conexión de Torus 2D

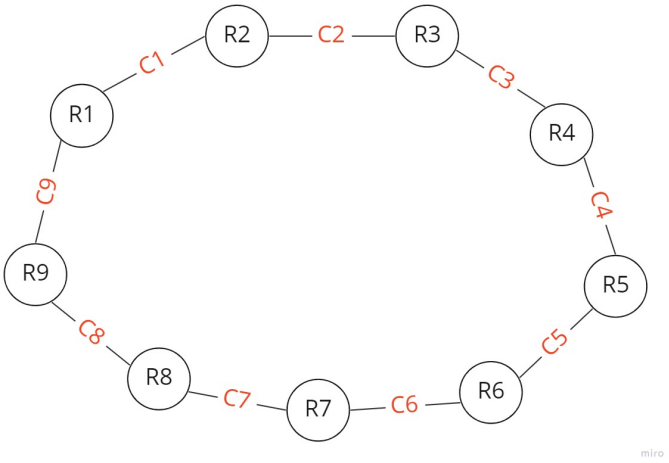

Figura 1: Topología Torus 1D

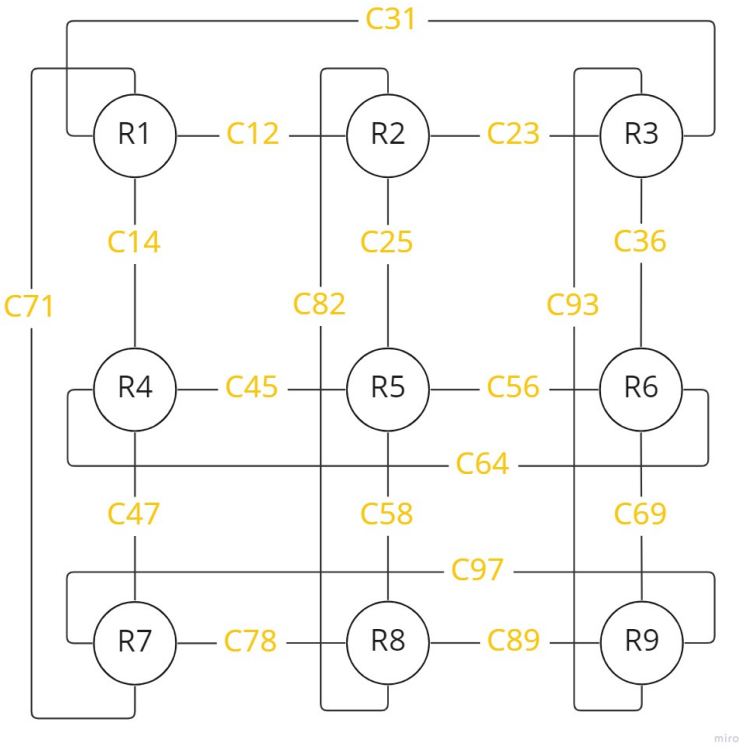

Figura 2: Topología Torus 2D

En cuanto a la implementación como tal, todos los archivos se encuentran en github. Se ha creado dos clases, *Router* y *VectorDistanceRouting*, con el fin de poder obtener el costo mínimo de cualquier router a cualquier router que tenga algún camino.

La clase *Router* está compuesta por tres atributos, que se detallan a continuación:

- `id` - `int`: Indica el índice del router, y va desde 0 hasta $n-1$, donde n es la cantidad total de routers.
- `neighbors` - `vector<pair<Router*, int>>`: Es un vector de pares, en donde el primer elemento es un puntero hacia el router vecino, y el segundo elemento corresponde al peso (costo) de llegar a ese router.

- `routingTable` - `vector<vector<int>>`: Es una matriz que almacena los costos desde cualquier couter x hasta cualquier router y, donde x y y son routers declarados inicialmente.

Por otro lado, la clase *VectorDistanceRouting* solamente cuenta con un atributo, que se detalla a continuación:

- `routers` - `vector<Router*>`: Almacena punteros a todos los routers que se quieren incluir dentro del algoritmo de *VectorDistanceRouting* y permite tener acceso a todos los routers para poder aplicar el algoritmo después, así mismo poder acceder a sus atributos.

Dentro de la clase *Router*, se destaca el método `buildDistanceVector`, que se encarga de llenar la fila correspondiente al índice del router con los costos a todos los vecinos. En caso no tenga conexión con todos los demás routers, se coloca un valor de -1.

```
void buildDistanceVector(int numberOfRouters) {
    routingTable = vector<vector<int>>(numberOfRouters, vector<int>(numberOfRouters,
    -1));
    for (int i = 0; i < numberOfRouters; i++) {
        if (i == id) {
            routingTable[id][i] = 0;
        } else {
            routingTable[id][i] = getCost(i);
        }
    }
}
```

Listing 1: Función buildDistanceVector de la clase Router

Por otra parte, dentro de la clase *VectorDistanceRouting* se encuentra el método `buildRoutingTables`, que se encarga de ejecutar toda la lógica necesaria para poder obtener como resultado las tablas de rutas de todos los routers.

Para ello, primero se ejecuta el método `buildDistanceVector` para cada router, con el objetivo de que cada uno llene el costo hacia cada uno de sus vecinos.

```
for (auto & router : routers) {
    router->buildDistanceVector((int)routers.size());
}
```

Listing 2: Ejecución buildDistanceVector en buildRoutingTables

A continuación, primero se copian los vectores distancia calculados entre todos los routers, para que después se ejecute la ecuación de *Bellman-Ford* para hallar el costo mínimo entre un router de inicio x y un router de llegada y.

```
for (auto & router : routers) {
    for (auto & neighbor : router->getNeighbors()) {
        for (int i = 0; i < (int)routers.size(); i++) {
            router->changeRoutingTableValue(neighbor.first->getId(), i, neighbor.first
    ->getRoutingTableValue(neighbor.first->getId(), i));
        }
        for (int i = 0; i < (int)routers.size(); i++) {
            if (router->getRoutingTableValue(neighbor.first->getId(), i) != -1) {
                if (router->getRoutingTableValue(router->getId(), i) == -1) {
                    router->changeRoutingTableValue(router->getId(), i, router->
    getRoutingTableValue(neighbor.first->getId(), i) + neighbor.second);
                } else {
                    router->changeRoutingTableValue(router->getId(), i, min(router->
    getRoutingTableValue(router->getId(), i), router->getRoutingTableValue(neighbor.
    first->getId(), i) + neighbor.second));
                }
            }
        }
    }
}
```

Listing 3: Ejecución Bellman-Ford en buildRoutingTables

Luego, se verifica que el costo calculado por la ecuación de *Bellman-Ford* sea efectivamente el mismo para todas las tablas de los routers. En caso de encontrar algún costo que no vaya acorde entre todos los routers, se queda como valor el mínimo entre esos dos valores.

```
for (int i = 0; i < (int)routers.size(); i++) {
    for (int j = 0; j < (int)routers.size(); j++) {
        int minValue = INT_MAX;
        for (auto & router : routers) {
            if (router->getRoutingTableValue(i, j) != -1) {
                minValue = min(minValue, router->getRoutingTableValue(i, j));
            }
        }
        for (auto & router : routers) {
            router->changeRoutingTableValue(i, j, minValue);
        }
    }
}
```

Listing 4: Ejecución verificación de costo en buildRoutingTables

Finalmente, se hace una última comprobación, se verifica que el costo de llegar desde un router x a un router y sea exactamente el mismo que el costo de ir del router y al router x. Y, al igual que el caso anterior, en caso no sean iguales, prevalece el valor mínimo entre ambos.

```
for (int i = 0; i < (int)routers.size(); i++) {
    for (int j = 0; j < (int)routers.size(); j++) {
        if (i != j) {
            for (auto & router : routers) {
                if (router->getRoutingTableValue(i, j) != -1) {
                    int minValue = min(router->getRoutingTableValue(i, j), router->
    getRoutingTableValue(j, i));
                    router->changeRoutingTableValue(i, j, minValue);
                    router->changeRoutingTableValue(j, i, minValue);
                }
            }
        }
    }
}
```

Listing 5: Ejecución verificación de rutas inversas en buildRoutingTables

Luego de ejecutar este método, se tendrá como resultado una matriz nxn, donde n es el número total de routers, por cada router. Y todas las matrices obtenidas son idénticas, ya que los costos entre todos los vectores deben ser los mismos, tal como se muestra a continuación:

```
Routing table of router 0:
   0    1    2    3    8   16   24   58  113
   1    0    1    4    9   17   25   59  114
   2    1    0    5   10   18   26   60  115
   3    4    5    0    5   13   21   55  110
   8    9   10    5    0    8   26   60  115
  16   17   18   13    8    0   34   68  123
  24   25   26   21   26   34    0   34   89
  58   59   60   55   60   68   34    0   55
 113  114  115  110  115  123   89   55    0
```

2. Resolución

Los cálculos de $D_1(9), D_4(6), D_3(7), D_5(6)$ para Torus 1D son los siguientes:

- Para $D_1(9)$ el resultado es 34, tal como sale en la Figura 3.
- Para $D_4(6)$ el resultado es 8, tal como sale en la Figura 4.
- Para $D_3(7)$ el resultado es 18, tal como sale en la Figura 5.
- Para $D_5(6)$ el resultado es 5, tal como sale en la Figura 6.

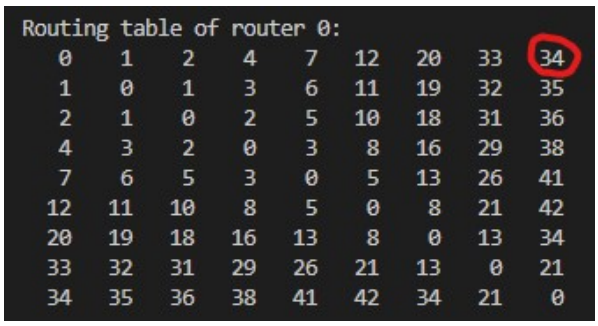

Figura 3: Distancia de R0 a R8 para Torus 1D

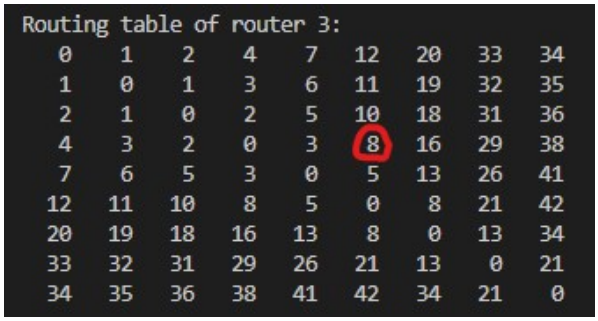

Figura 4: Distancia de R3 a R5 para Torus 1D

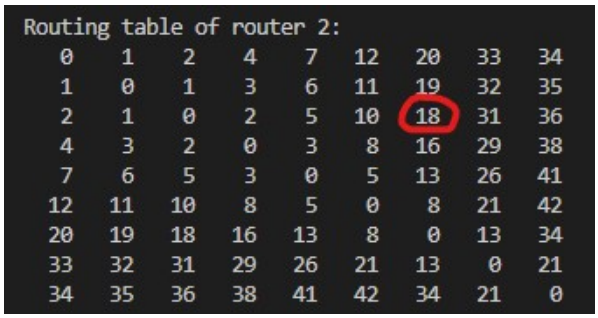

Figura 5: Distancia de R2 a R6 para Torus 1D

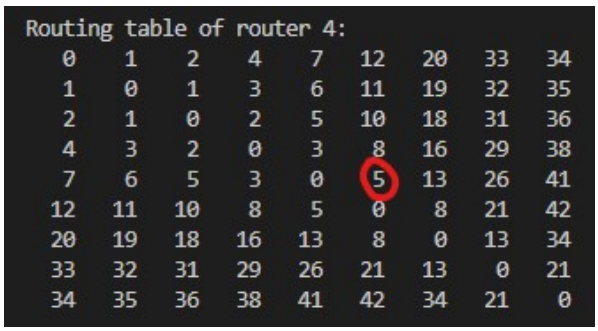

Figura 6: Distancia de R4 a R5 para Torus 1D

Tal como se esperaba, los nodos más lejanos obtuvieron distancias más largas que los nodos más cercanos, como es el caso de $D_4(6)$ y $D_5(6)$. El camino que recorre se muestra a continuación:

Vértice	Distancia	Camino
0 -> 8	34	0 -> 8 ->
3 -> 5	8	3 -> 4 -> 5 ->
2 -> 6	18	2 -> 3 -> 4 -> 5 -> 6 ->
4 -> 5	5	4 -> 5 ->

Los cálculos de $D_1(9), D_4(6), D_3(7), D_5(6)$ para Torus 2D son los siguientes:

- Para $D_1(9)$ el resultado es 113, tal como sale en la Figura 3.
- Para $D_4(6)$ el resultado es 13, tal como sale en la Figura 4.
- Para $D_3(7)$ el resultado es 26, tal como sale en la Figura 5.
- Para $D_5(6)$ el resultado es 8, tal como sale en la Figura 6.

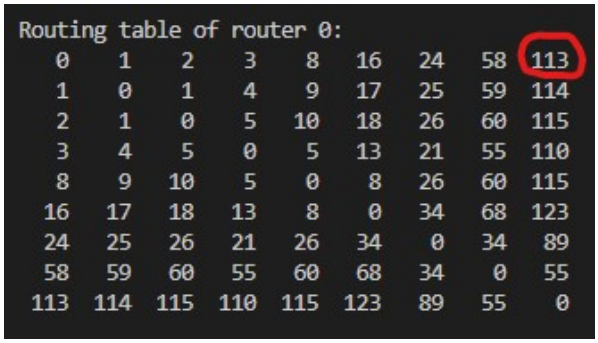

Figura 7: Distancia de R0 a R8 para Torus 2D

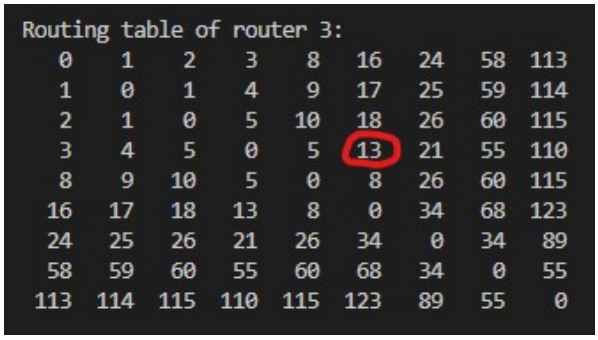

Figura 8: Distancia de R3 a R5 para Torus 2D

El camino que recorre es el siguiente:

Vértice	Distancia	Camino
0 -> 8	113	0 -> 3 -> 6 -> 8 ->
3 -> 5	13	3 -> 5 ->
2 -> 6	26	2 -> 0 -> 3 -> 6 ->
4 -> 5	8	4 -> 5 ->

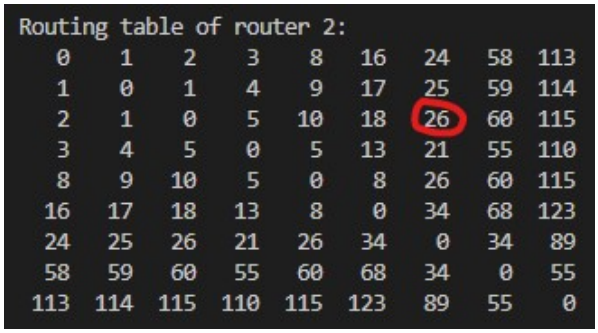

Figura 9: Distancia de R2 a R6 para Torus 2D

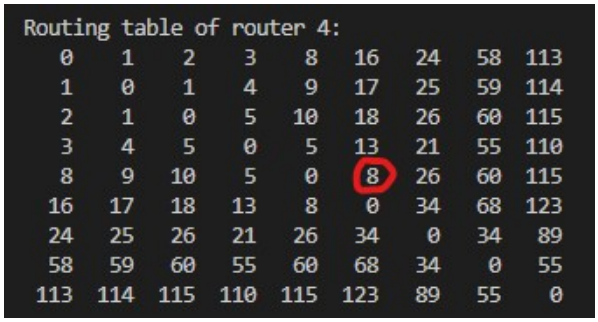

Figura 10: Distancia de R4 a R5 para Torus 2D

Como se puede ver en los resultados, las distancias calculadas por el Torus 1D son menores que las distancias obtenidas por el Torus 2D. Esto debido a que, al tener más aristas, el Torus 2D contiene pesos mucho mayores en comparación con el Torus 1D, dando como resultado que la distancia mínima obtenida para los puntos solicitados sean mayores en el caso del Torus 2D.