Universidad de Lima

Facultad de Ingeniería y Arquitectura

Carrera de Ingeniería de Sistemas

TRABAJO INTEGRADOR:

VECTOR DE DISTANCIAS

Chávarry Gutiérrez, Sebastián Alexander

Código: **20200498**

Profesor:

Carlos Martín Torres Paredes

Curso:

Redes de Computadoras

Sección: 602

Lima – Perú

3 de diciembre de 2021

UNIVERSIDAD DE LIMA
MCMLXV
SCIENTIA ET PRAXIS

TABLA DE CONTENIDO

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

ÍNDICE DE ANEXOS

I. INTRODUCCIÓN

En el presente trabajo se busca implementar uno de los protocolos de enrutamiento dinámico: Vector de Distancias, el cual hace uso del algoritmo Bellman-Ford para experimentar su comportamiento en 2 topologías diferentes. De esta forma, se podrá comparar su rendimiento durante la experimentación utilizando las métricas: Tiempo de Ejecución, Uso de Memoria y Número de Iteraciones. Además, se comparará el programa implementando con el de un compañero para analizar el diseño de experimentación más adecuado. Para la experimentación se ha seleccionado utilizar el lenguaje de programación "*Python*", debido que ofrece una versatilidad a la hora de programar, pues se trata de un lenguaje de alto nivel. En esa línea, facilitará la implementación de varias funciones en el programa de experimentación. Para ver el código fuente en cuestión, revisar el anexo adjunto en el que encontrará el enlace hacia un repositorio GitHub, donde además encontrará las instrucciones de instalación.

II. ESTADO DEL ARTE

2.1 Conceptos Teóricos

Para entender a cabalidad la experimentación es necesario entender el contexto en el que vamos a aplicar los algoritmos. El modelo OSI divide a los protocolos de red en 7 capas principales: Física, Datos, Red, Transporte, Sesión, Presentación y Aplicación. Para este experimento, la capa de Red es la más relevante, ya que de acuerdo con la investigación "*Research on the Performance of Dynamic Routing Algorithm*" presentada por Ming-Xin Yang et al (2009) la existencia de esta capa asegura la transmisión de la información entre computadoras. Esto se debe principalmente a que en esta capa se ejecuta la lógica de la manera en cómo los datagramas viajan de un router a otro, lo cual se conoce como enrutamiento de paquetes. Esta lógica varía según los protocolos estipulados en cada topología, pero generalmente se busca hallar la mejor ruta entre todos las posibles routers de una red asegurando así el mínimo costo de transporte posible.

Para traducir esto en términos de programación, Kurose (2017) elabora en su obra "*Redes de Computadoras: Un enfoque Descendente*" que hay que entender a las redes como grafos, donde los nodos representan a los routers, las aristas las conexiones y los pesos la longitud física del enlace. En Ciencias de la Computación un grafo es una Estructura de Datos que se expresa de la siguiente manera: $G = (N, E)$. Esta expresión representa que en un grafo G existen una N cantidad de nodos y una E cantidad de aristas, las cuales se encuentran conectadas a un nodo de N. Los grafos se pueden clasificar entre dirigidos y no dirigidos. Para el contexto de redes solo se trabaja con grafos no dirigidos, es decir, grafos cuyas aristas no tienen una dirección o sentido definido. Las aristas tienen pesos, que son definidos por el mismo usuario y al sumar los pesos de determinadas aristas se puede obtener una ruta. En ese sentido, la ruta más óptima sería aquella que tenga la menor suma total de los pesos de las aristas recorridas.

En el contexto de redes de computadoras, existen 2 tipos de enrutamiento: Estático y Dinámico. En el caso del primero, es el mismo administrador quien configura las rutas en cada router. Si bien asegura un camino seguro, cuando se trabajan con redes complejas resulta ineficiente. Siguiendo la línea de la investigacion de Ming-Xin Yang et al (2009) el enrutamiento estático no considera la carga actual de la red. Esto quiere decir, que si

se realizan cambios en la topología de una red, se tendría que recalcular el camino más corto manualmente. Por ello, es más útil utilizar un enrutamiento dinámico para realizar estos cálculos más eficientemente. Se usan 2 algoritmos principalmente: Vector de Distancias o Estado de Enlaces. Para esta investigación principalmente se ha utilizado el vector de distancias.

2.2 Algoritmo Vector de Distancias

De acuerdo con Rothkrantz (2009) *en su investigación "Dynamic Routing using the network of car drivers*" sigue la misma línea planteada anteriormente indicando que es más eficiente emplear algoritmos de enrutamiento dinámicos, pero no descarta que en la actualidad se sigan usando estos en sistemas de tráfico vehicular. Así introduce el funcionamiento de este algoritmo, el cual utiliza el algoritmo de Bellman-Ford para construir y mantener actualizada la representación de la red dentro del router para que así pueda guiar los paquetes en la ruta óptima posible. Un elemento que caracteriza a este algoritmo es que se mantiene una table de enrutamiento dentro de cada router, de tal forma que constantemente se va actualizando con cada iteración.

Min-Xin Yang et al (2009) profundiza más en este punto, indicando que cada router mantiene una tabla en la que los índices representan a cada router en la red y que a su vez cada router corresponde a un elemento de la tabla llegando a formar una matriz de distancias, la cual es actualizada constantemente mediante el intercambio de información con routers vecinos.

2.3 Trabajos Anteriores

Para elaborar la metodología de la experimentación se ha tomado como referente distintos trabajos anteriores que abordan esta temática. En primer lugar, se tomó "*An Experiment on the Performance of Shortest Path Algorithm*" de Chan et al (2016), el cual nos presenta un enfoque cuantitativo para comparar algoritmos de la ruta más corta. En este caso, cada algoritmo es ejecutado 20 veces para cada caso del experimento y luego se calcula el valor promedio. Si bien se comparan un conjunto de datos grande y pequeño, lo que se realiza para asegurar un punto común de comparación es evaluar cada algoritmo desde el mismo punto de origen y destino. Y así luego se compara el tiempo de ejecución y distancia recorrida total. Sin embargo, un aspecto criticable es la cantidad de métricas utilizadas, puesto que hubiera sido más interesante utilizar más métricas para así tener un

análisis más profundo de los algoritmos utilizados. A continuación, se puede apreciar sus resultados:

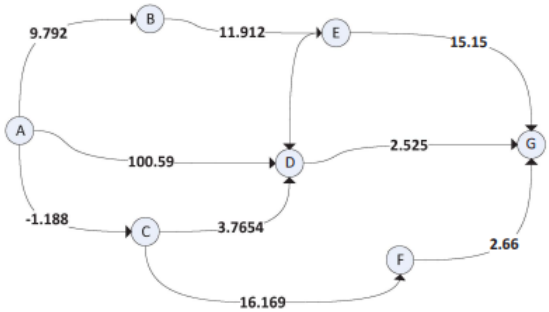
Figura 2.1:
Tabla de comparación de resultados de algoritmos de búsqueda del camino más corto

Algorithm	Small Sample Data		Large Sample Data	
	Running Time (nanosecond)	Total Distance (meter)	Running Time (nanosecond)	Total Distance (meter)
Dijkstra	55658	18240	321712	18240
Symmetrical Dijkstra	27632	18240	144080	18240
A*	43816	18240	322501	18240
Bellman-Ford	42237	18240	88816	18240
Floyd-Warshall	61974	18240	584213	18240
Genetic	863688	18240	865661	18240

Nota: De "Comparison of Performance of Algorithm to solve shortest path problem for large data versus small data cases". Por Chan, S. Y. M., Adnan, N. A., Sukri, S. S., & Wan Zainon, W. M. N. (p.3). https://core.ac.uk/download/pdf/78486805.pdf

Además, se ha tomado como referencia trabajos en otras áreas de la ingeniería tales como la investigación "*A Bellman-Ford Approach to Energy Efficient Routing of Electric Vehicules" presentada por Abousleiman y Rawashdeh* (2015), en la que cual se trata de implementar el algoritmo Bellman-Ford en el contexto de enrutamiento de autos eléctricos para así tener un uso más eficiente de la energía. En este caso, los valores de las aristas son calculadas a partir de ecuaciones físicas, la distancia, límite de velocidad y elevación de los nodos. Aquí se puede evidenciar como se puede mezclar ecuaciones con valores de tablas externos para implementar un cálculo adicional a ser considerado durante la ejecución del algoritmo de la ruta óptima. Así se genera un grafo con la respectiva cantidad de energía utilizada para desplazarse de nodo a nodo. En ese sentido, se tomará como referencia la forma en como estructuran sus grafos para extrapolarlo en el contexto de redes. Para ello, a continuación se muestra un ejemplo de sus grafos:

Figura 2.2:
Grafo con aristas de peso de consumo de energía en Joules



Nota. De "A Bellman-Ford approach to energy efficient routing of electric vehicles". Por Rami Abousleiman y Osamah Rawashdeh (p.3). https://doi.org/10.1109/itec.2015.7165772

Finalmente, se toma como referencia el trabajo "Analysis of algorithms for shortest path problema in parallel" presentado por Popa y Pospescu (2016), la cual ofrece un enfoque innovador y eficiente de los algoritmos de Dijkstra, Bellman-Ford, Floyd-Warshall y Viterbi utilizando programación paralela. Si bien es cierto que este enfoque difiere del que se va a emplear en este trabajo, se puede destacar su metodología de comparación de algoritmos. Lo que se realiza es implementar tablas, en las que se comparen el tiempo de ejecución de sus algoritmos utilizando una distinta cantidad de nodos, llegando a la conclusión que el tiempo de ejecución se ve directamente influenciado por la estructura del grafo y el algoritmo en sí. No obstante, no se llegará a utilizar topologías tan grandes como las que se plantean aquí, pero para futuros trabajos se podría considerar este enfoque, debido a que en redes existen topologías con cientos de routers. Por otro lado, un aspecto criticable de este es el cálculo del uso del CPU, porque están utilizando una herramienta externa, cuando esta métrica podría ser sencillamente implementada dentro del mismo programa y así obtener un resultado cuantitativo directo. A continuación, se muestran los resultados:

Figura 2.3:
Tabla con los tiempos de ejecución en segundos de los algoritmos implementados

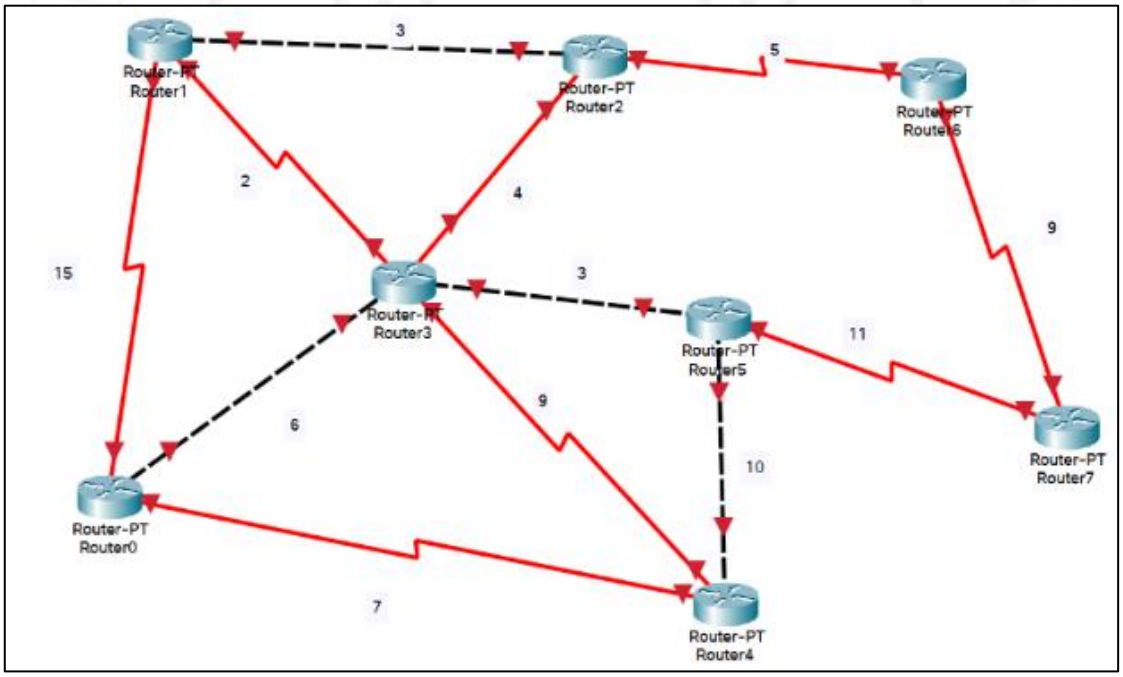
Number of nodes	Dijkstra's Sequential	Bellman Ford's Sequential	Dijkstra's Parallel	Bellman Ford's Parallel
250	0.01431	0.01749	0.04432	0.04551
400	0.04	0.04998	0.07432	0.07621
600	0.07321	0.08214	0.10873	0.11001
1000	0.09238	0.10326	0.13674	0.13882
1200	0.11787	0.13521	0.16672	0.16992
1500	0.15765	0.18532	0.19652	0.20671
1800	0.18021	0.21892	0.24001	0.25011
2000	0.20021	0.24136	0.26019	0.28732
2400	0.24532	0.2891	0.29115	0.31882
3000	0.29311	0.34122	0.32271	0.35611
3600	0.33427	0.39922	0.36511	0.38321
4000	0.43921	0.51021	0.48821	0.51221
5000	0.59043	0.62925	0.62991	0.65819

Nota. De “Analysis of algorithms for shortest path problem in parallel”. De Bodgan Popa y Dan Popescu. (p.3). https://doi.org/10.1109/carpathiancc.2016.7501169

III. DISEÑO DEL ESCENARIO DE EXPERIMENTACIÓN

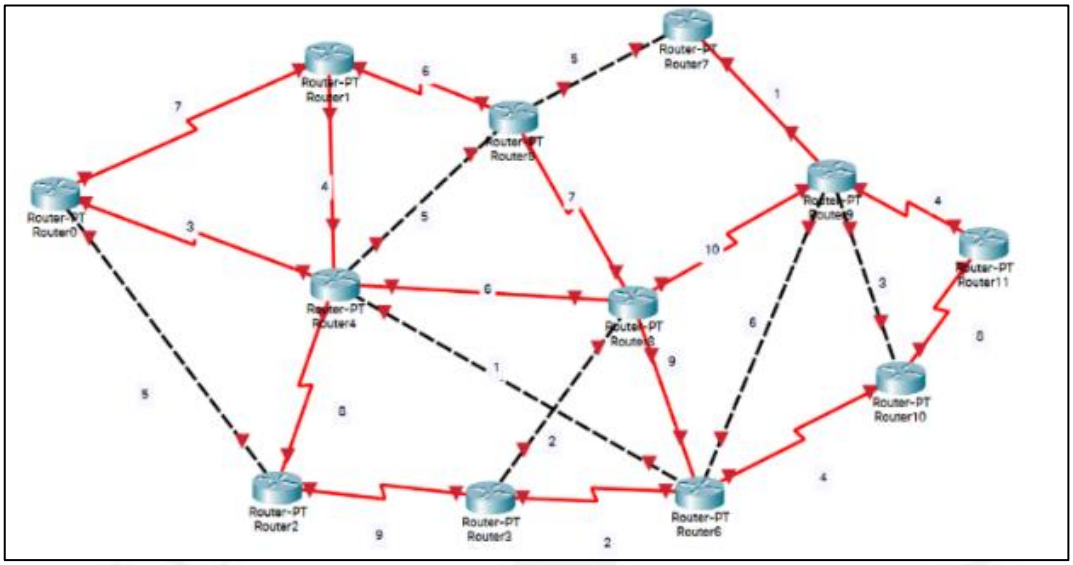
Para poder realizar una correcta experimentación de los algoritmos de enrutamiento se decidió experimentar con 2 escenarios distintos. Para ello, se utilizó la herramienta "Cisco Packet Tracer" para diagramar las conexiones entre routers. Para efectos del experimento no es necesario configurar cada router, porque no se pretende hacer un red funcional, solo visualizar correctamente los escenarios. Tampoco es necesario utilizar un tipo de router en específico, en este caso se empleó el Router-PT, dado que nos ofrecía varias interfases, lo cual permite tener varias conexiones entre routers. Pero, de todas formas se puede agregar manualmente más interfases en otros tipos de routers. El primer escenario se denomina "Topología de complejidad baja". Y así como se observa en la ilustración X, este escenario está conformado por 8 routers y 12 aristas no dirigidas. El segundo escenario se denomina "Topología de complejidad media". Y en este caso, el escenario está conformado por 12 routers y 22 aristas no dirigidas.

Figura 3.4:
Topología de Complejidad baja



Nota. Elaboración propia en Packet Tracer

Figura 3.5:
Tipología de complejidad media



Nota. Elaboración propia en Packet Tracer

IV. DISEÑO DE LA HERRAMIENTA DE EXPERIMENTACIÓN

En este capítulo se explicará la lógica del algoritmo a implementar, así como la lógica del experimento en sí con la finalidad de que el lector pueda entender la estructura del programa y pueda replicarlo en otros lenguajes de programación, en el siguiente capítulo se explicará a mayor profundidad la funcionalidad de cada función utilizada.

El protocolo de Vector de Distancia utiliza el algoritmo Bellman-Ford, el cual es un algoritmo dinámico que permite hallar el camino más corto en un grafo de manera similar al algoritmo de Dijkstra. Pero, lo que los diferencia es que este algoritmo puede trabajar con aristas de peso negativo evitando así ciclos negativos. En este caso estamos aplicando el paradigma Orientado a Objetos para implementar este algoritmo como una función dentro de la clase principal que representa un grafo. En ese sentido, el único parámetro que necesita recibir es el nodo fuente a partir del cual se hallará la ruta más corta, por lo demás esta función ya tendría acceso a la cantidad de nodos y la matriz de aristas. Así el pseudo código del algoritmo es el siguiente:

Figura 4.6:
Pseudocódigo del algoritmo Bellman-Ford

```
Clase Grafo:
    INICIO BellmanFord (Vertice Origen)

        INICIALIZAR infinito = Valor infinito
        INICIALIZAR distancias = nueva Lista [cantV] y ASIGNAR infinito a todos sus valores
        Asignar a distancias[Vertice Origen] = 0
        INICIALIZAR predecesor = nueva Lista [cantV] y ASIGNAR None a todos sus valores

        POR CADA nodo en el RANGO(cantV - 1):
            POR CADA arista en la matriz HACER:
                SI (distancias[u] != infinito) Y (distancias[u] + w < distancias[v]) ENTONCES:
                    distancias[v] = distancias[u] + w
                    predecesor[v] = u


        POR CADA ARISTA en la matriz HACER:
            SI (distancias[U] != infinito) Y (distancias[u] + w < distancias[v]) ENTONCES:
                IMPRIMIR('El Grafo contiene ciclos de peso negativo')
                ROMPER BUCLE
    FIN
```

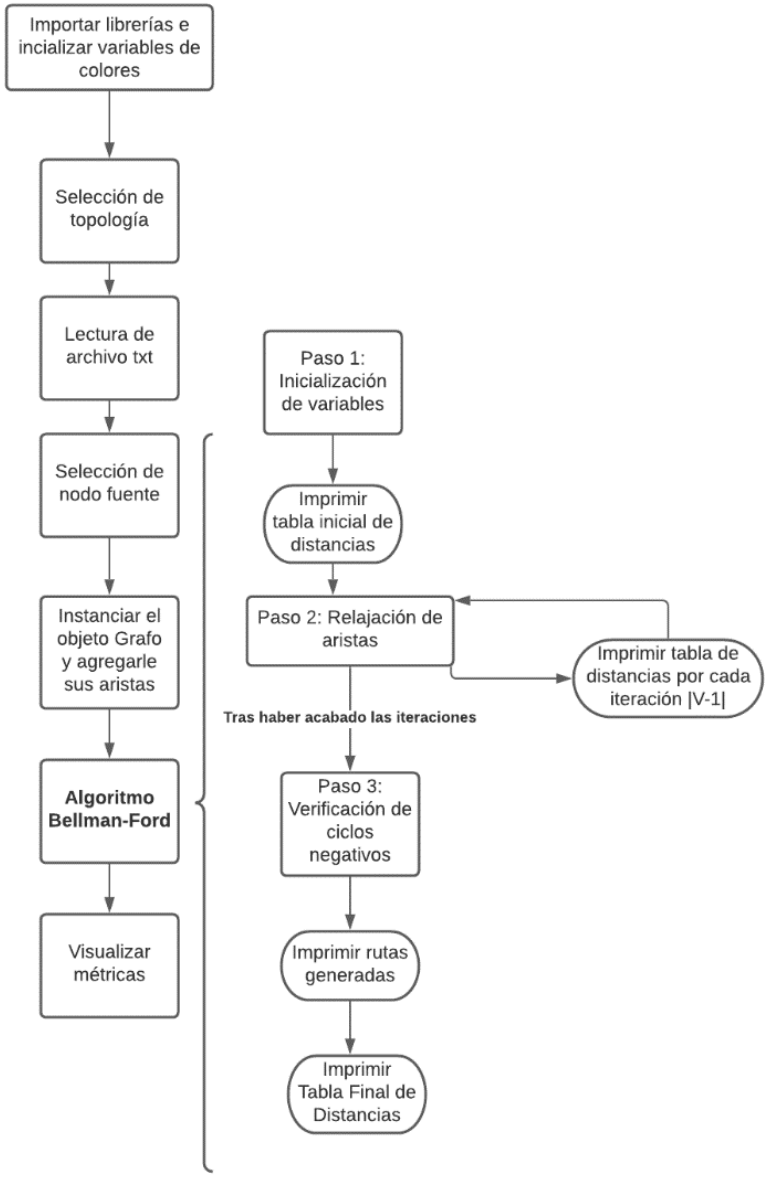
Nota. Elaboración propia en Spyder

Como se puede apreciar el algoritmo tiene 3 pasos principales. En primer lugar, se busca inicializar un arreglo de distancias (en este caso como se está trabajando en Python, se utiliza una lista), el cual represente la suma de pesos de aristas desde el nodo origen hasta los demás nodos para así calcular la ruta óptima. En este caso, se inicializa

la lista de tamaño igual a la cantidad de vertices y asigna el valor infinito a cada espacio. Luego, el valor de distancias en el índice del nodo origen es 0, dado que no se puede encontrar una ruta más corta hacia el mismo nodo. Además, se inicializa una lista de precedesor, el cual tendrá un tamaño igual a la cantidad de vertices e inicialmente se asignará el valor Null (en este caso en Python sería None) a todos los espacios. Esta lista permitirá más adelante rastrear y generar más adelante la ruta exacta del recorrido del algoritmo. En segundo lugar, se busca calcular las distancias más cortas. Lo que se realiza aquí es calcular la distancia entre los nodos para luego compararla con las distancias de otros nodos ya conocidos. Esto es lo que se conoce como "relajación de aristas", lo cual implica que conforme se va explorando el grafo, la arista entre un nodo origen 'u' está conectado a un nodo destino 'v' y se aplica la siguiente lógica: Si la suma del peso de la arista con la distancia recorrida del nodo origen es menor a la distancia recorrida del nodo destino, entonces se actualiza su valor con la suma previamente hecha. Y así sucesivamente se van relajando las aristas hasta encontrar la ruta más corta. Finalmente, el tercer paso permite la verificación de ciclos negativos, aplicando la misma lógica de la relajación. Esto se explica principalmente, porque si la suma de pesos es negativa, entonces ciertamente a la hora de realizar la comparación las distancias recorridas serían siempre menores.

En la implementación de este experimento no solo se busca obtener la tabla de distancias más cortas, sino permitir al usuario experimentar con distintos parámetros para así evaluar 3 métricas de comparación: Tiempo de ejecución, Cantidad de memoria usada y número de iteraciones. Para ello, se ha decidido la siguiente estructura de código:

Figura 4.7:
Flujograma del programa de experimentación



Nota. Elaboración propia en Lucidchart.

V. IMPLEMENTACIÓN DE LA HERRAMIENTA DE EXPERIMENTACIÓN

En este capítulo se busca explicar cada parte del código del programa implementado para que el lector pueda entender que se está realizando en cada segmento y pueda replicar la metodología en otros lenguajes de programación.

Antes de todo se crearon variables auxiliares, que representan los colores para imprimir de forma personalizada los resultados. Estas variables utilizan el código ASCII. También se importaron distintas librerías que serán explicadas en su debido momento. Primero, hay 2 componentes principales del código: La función principal y la clase 'Grafo'. En este experimento, se ha decidido utilizar el paradigma orientado a Objetos para así implementar esta estructura de datos. En este caso, la clase grafo tiene los siguientes métodos:

1. **Método Constructor**: En Python *__init__* es una palabra reservada que permite la construcción de objetos a partir de una estructura definida en la clase. Es decir, cada vez que se llame a este método, se estará creando un objeto a partir de esta clase y tendrá los atributos que se han establecido. En este caso, el usuario debe definir como parámetro de entrada la cantidad de vértices que va a tener cada grafo. Además, cada vez que se instancie un objeto, tendrá un atributo llamado *matriz* que será una lista vacía para almacenar las aristas. Para simplificar el proceso, no se está utilizando una matriz de adyacencia, sino simplemente esta lista almacenará listas que representan a las aristas. Por otro lado, hay que destacar que, en Python, todos métodos deben incluir como parámetro la palabra *self*, que hace hace referencia al mismo objeto y tiene la finalidad de permitir a los métodos acceder a los atributos y otros métodos de la clase. Sin embargo, el usuario a la hora de instanciar el objeto, no deben incluirlo.
2. **Agregar aristas**: Este método como su nombre indica sirve para agregar aristas que están representadas en forma de lista hacia la lista *matriz*. A diferencia de otros lenguajes de programación, donde el manejo de arreglos necesitaría la inclusión de una estructura iterativa, en Python este proceso es facilitado por la función *append()*, que permite agregar al final de una lista un elemento de manera dinámica, ya que las listas no tienen un tamaño definido. Así se podrá conectar los nodos y obtener un

grafo. Los parámetros que debe recibir son: u, v, w, los cuales representan al nodo origen, nodo destino y el peso de la arista respectivamente.

3. **Imprimir**: Este método tiene la función de imprimir las tablas de distancias desde un nodo fuente hacia los demás nodos del grafo indicando la suma de los pesos de las aristas recorridas para obtener el camino más corto. Es un método auxiliar que se utilizará al inicio del programa, se imprimirá cada vez que itere el algoritmo en un nodo para ver la evolución de la tabla conforme se va explorando el grafo y finalmente se imprimirá una vez se haya obtenido la ruta más corta. Para ello, recibirá como parámetros la lista *distancias* creada en el método de Bellman-Ford y el nodo *fuente* recibido en ese mismo método.
4. **Imprimir rutas**: En el método explicado anteriormente si bien sirve para ver cuanto costaría recorrer la ruta más corta, no indica cuales son los nodos por recorrer para obtener esa distancia. En ese sentido, este método cumple con la función de visualizar esta ruta indicando de forma gráfica los nodos por recorrer para llevar a todos los nodos del grafo. Para eso principalmente se utiliza como parámetro la lista *predecesor*, la cual incluye este recorrido de nodos. Para ello, se hace una iteración por cada nodo y se crea una variable auxiliar *current*, así se recorrer la lista y mientras que el valor no sea nulo (None en ese caso) se irá agregando los nodos a una variable *String* auxiliar que luego se imprimirá para visualizar el camino.
5. **Bellman-Ford**: La lógica de este algoritmo ya ha sido explicado previamente en el capítulo anterior. Sin embargo, para términos de experimentación se ha modificado la estructura original. En el paso 1, se ha uso de la librería *Numpy*, la cual es una librería predeterminada de Python que contiene distintas funciones de matemáticas. En este caso, se le utiliza para poder crear la variable *infinito* y así asignar dicho valor a toda la lista de *distancias*. En el paso 2 se ha agregado dos variables auxiliares: *iteracion*, que servirá para contar el número de iteraciones que se va a ejecutar el algoritmo, y la variable *nro_repeticiones* que será una variable global que almacenará el valor de *iteracion*, pero podrá ser accedida fuera de la función. Además, en este mismo paso se incluyen llamadas al método *imprimir* para visualizar como va evolucionando la tabla de distancias. Finalmente se imprime la tabla final y las rutas.

Luego, con respecto a la función principal, se encuentra estructurada de la siguiente manera:

1. **Selección de topologías**: El usuario va a ingresar por consola un número que va a representar el tipo de topología con el que se va a trabajar. En este caso 1 representa la topología de 8 nodos, mientras que 2 representa la topología de 12 nodos. Para asegurar un correcto funcionamiento del programa de ha agregado un manejo de errores de tal forma que en caso se ingrese un valor equivocado se va a mostrar en pantalla de nuevo un mensaje de advertencia así hasta que el usuario ingrese un valor correcto.
2. **Lectura de archivo txt**: Para este experimento se está trabajando con archivo txt que contienen datos sobre el grafo. En la primera línea se incluye el número de nodos y el resto del archivo indican datos sobre las aristas. Cada línea contiene el nodo origen, nodo destino y el peso de la arista en ese respectivo orden. Dado que el archivo txt contiene valores en formato *String* para la correcta lectura de los archivos, se utiliza 2 funciones: *rstrip()*, la cual sirve para eliminar los espacios blancos al final de cada String y *split()*, que permite dividir un String en una lista de String. Tras aplicar estas funciones al mismo tiempo, se obtiene una lista auxiliar *temp* que contiene listas de *String* que representan cada línea del archivo. En caso haya un error de lectura, el programa muestra en pantalla el error y cierra el programa. Para ello, se utilizan la librería *sys*, la cual permita manipular el ambiente de ejecución de Python.
3. **Selección del nodo origen**: Aquí también el usuario para ingresar por consola el nodo fuente. En caso el usuario ingrese un valor equivocado, se imprimirá una advertencia para que ingrese un número valido entre el rango de 0 y el número de nodos – 1. Para este punto, el programa ya habrá leído a partir del txt la cantidad de nodos con las que se van a trabajar.
4. **Instanciar Grafo y agregar aristas**: Aquí se hace llamado a la clase Grafo para instanciar un objeto grafo con la cantidad de nodos indicada en el archivo txt. Y luego se va haciendo llamado a su método *agregar Arista* para que conforme se va iterando en *temp* se vaya agregando aristas. Dado que esta variable contiene listas de valores String, es necesario castear dichos valores en valores enteros. Además, en este caso se aprovecha la función *enumurate()* de Python para así evitar considerar la primera línea del archivo. A diferencia de otros lenguajes, aquí no es necesario indicar el número de aristas, ya que Python va a iterar hasta el final del archivo, es decir, hasta que ya no haya otra línea más que leer.

5. **Visualizar métricas**: En este apartado se van a imprimir en pantalla las métricas obtenidas. En primer lugar, para el tiempo de ejecución se está utilizando la librería *timeit*, la cual permite calcula el tiempo de ejecución de una función en segundos. El primer valor que recibe es *stmt*, el cual es la función que se va a medir. En este caso se utiliza *lambda*, dado que esto permite la ejecución de la función Bellman-Ford en ese momento como si fuese una función anónima. También recibe el parámetro *number*, el cual indica el número de veces que se va a ejecutar. Luego, para calcular el uso de memoria en megabytes, se hace uso de la librería *psutil*, la cual permite recolectar información sobre la ejecución de progresos y utilización de recursos del sistema y *os*, la cual permite realizar operaciones en referencia al Sistema Operativo. Primero, se obtiene el ID del proceso actual usando *os.getpid()*. Luego, con ese datos, se puede obtener la información en memoria usando *memory_info()* obteniendo así el espacio en memoria en Bytes y luego se convierte a Megabytes al dividirlo entre 1024^2. Finalmente, se imprime el número de repeticiones obtenidos en el paso 2 del algoritmo de Bellman-Ford, este valor por teoría debería ser siempre |V - 1|, donde V represente el número de nodos.

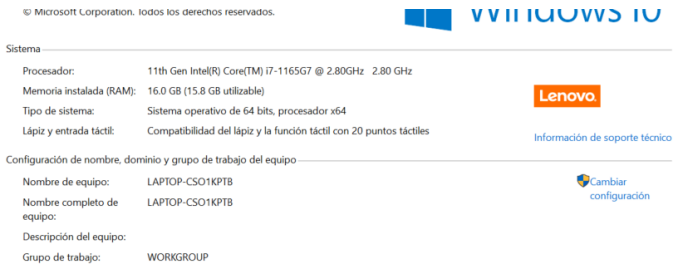
VI. EXPERIMENTACIÓN

En este capítulo se mostrarán los resultados obtenidos y se comparará el programa implementado con el algoritmo de Dijkstra realizado por otro compañero con la finalidad de comparar las métricas y analizar cual programa y algoritmo es más eficiente.

En primer lugar, hay que considerar que toda la experimentación fue realizada en una laptop personal que tiene las siguientes especificaciones:

Figura 6.8:

Especificaciones técnicas del computador en el que se realizará el experimento



Nota. Captura de pantalla.

Para la correcta comparación de los resultados del programa implementado se ha decidido tomar como criterios principales: Tiempo de ejecución, Uso de Memoria y Número de Iteraciones. En ese sentido, se ha ejecutado 10 veces el programa por topología considerando el mismo nodo de origen y se obtuvieron los siguientes resultados:

Tabla 6.1:

Tabla de resultados en la topología de complejidad baja

	Tiempo de ejecución (segundos)	**Uso de Memoria (MB)**	**Número de Iteraciones**
Intento 1	0.0034646	69.12109375	7
Intento 2	0.0027644	69.671875	7
Intento 3	0.0020791	69.78515625	7
Intento 4	0.0017038	69.81640625	7
Intento 5	0.0021876	69.83984375	7
Intento 6	0.0022638	69.875	7
Intento 7	0.0029951	69.9140625	7
Intento 8	0.0017033	69.91015625	7
Intento 9	0.0026075	69.96484375	7
Intento 10	0.0007418	70.12109375	7
Promedio	**0.0022511**	**69.80195313**	**7**

Valor Máximo	**0.0034646**	**70.12109375**	**7**
Valor Mínimo	**0.0007418**	**69.12109375**	**7**

Nota. Elaboración propia

Tabla 6.2:
Tabla de resultados en la topología de complejidad media

	Tiempo de ejecución (segundos)	**Uso de Memoria (MB)**	**Número de Iteraciones**
Intento 1	0.0403321	70.2734375	11
Intento 2	0.004682	70.3046875	11
Intento 3	0.0381715	70.34765625	11
Intento 4	0.0031562	70.38671875	11
Intento 5	0.0048458	70.3984375	11
Intento 6	0.0060969	70.43359375	11
Intento 7	0.0043079	70.44140625	11
Intento 8	0.0038708	70.45703125	11
Intento 9	0.0025197	70.46484375	11
Intento 10	0.0018275	70.4765625	11
Promedio	**0.01098104**	**70.3984375**	11
Valor Máximo	**0.0403321**	**70.4765625**	11
Valor Mínimo	**0.0018275**	**70.2734375**	11

Nota. Elaboración propia

Adicionalmente, se decidió comparar este programa con el algoritmo Dijkstra implementado por el compañero Fabrizio Figari, quien realizó su programa en Python y cuenta con métricas similares. Sin embargo, su diseño de presentación es diferente, ya que muestra el tiempo de ejecución en milisegundos, así que se tuvo que realizar la respectiva conversión. También hay que considerar que el uso de memoria que se muestra es la suma de memoria utilizada al utilizar el algoritmo en sus 2 topologías implementadas, las cuales son distintas a las de este experimento. Considerando ello se obtuvo los siguientes resultados:

Tabla 6.3:
Tabla de resultados del programa implementado por Fabrizio Figari

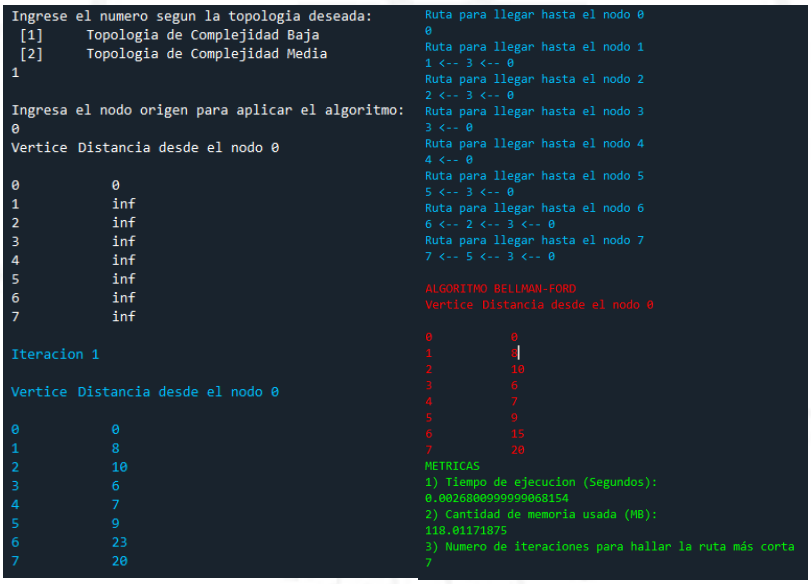
	Tiempo de ejecución en topología 1 (segundos)	**Tiempo de ejecución en topología 2 (segundos)**	**Uso de Memoria (MB)**
Intento 1	0.1034	0.1146	139.71
Intento 2	0.0986	0.3161	139.66
Intento 3	0.0966	0.1147	145.36
Intento 4	0.1384	0.1733	145.46
Intento 5	0.5241	0.1278	145.47
Intento 6	0.1373	0.1294	145.45

Intento 7	0.3135	0.1355	145.46
Intento 8	0.1087	0.1273	145.46
Intento 9	0.1994	0.2307	145.54
Intento 10	0.1386	0.5657	145.75
Promedio	**0.18586**	**0.20351**	**144.332**
Valor Máximo	**0.5241**	**0.5657**	**145.75**
Valor Mínimo	**0.0966**	**0.1146**	**139.66**

Nota. Elaboración propia

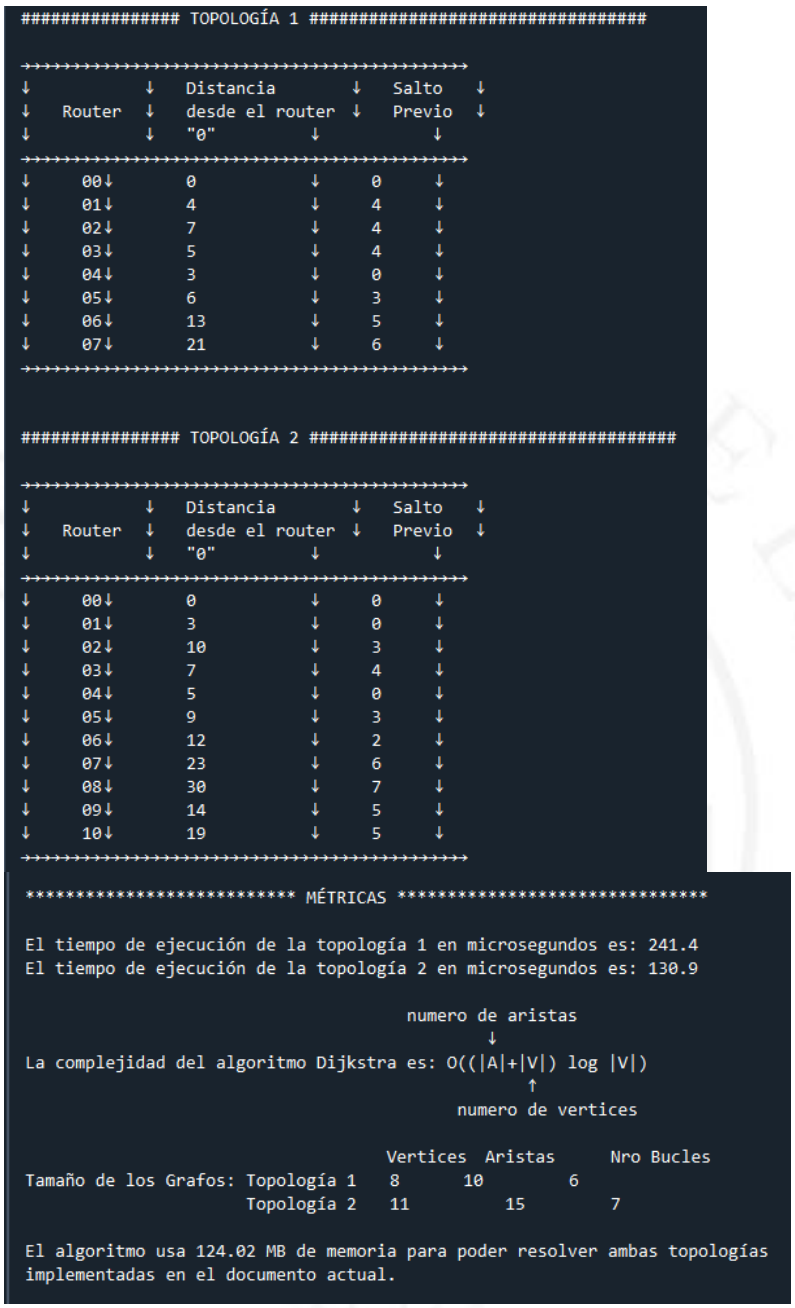
Dado que evidentemente los resultados van a ser distintos, debido a que se tratan de programas y topologías diferentes, se ha decidido considerar los 4 Principios del Diseño Interactivo para comparar los programas realizados con la finalidad de establecer una base para futuros proyectos, de tal forma que puedan considerar estos criterios a la hora de implementar experimentos. Se toma principalmente en cuenta el programa en sí al ser ejecutado por el usuario. Por lo que a continuación se adjunta algunas capturas de pantalla de la visualización de resultados de cada programa:

Figura 6.9:
Resultados en terminal del programa implementado en la experimentación



Nota. Captura de pantalla de terminal

Figura 6.10:
Resultados en terminal de programa implementado por Fabrizio Figari



Nota. Captura de pantalla de terminal

VII. ANÁLISIS COMPARATIVO

En este capítulo se busca interpretar los resultados obtenidos a partir de la experimentación realizada con la implementación de este reporte usando las 2 topologías propuestas. Así como comparar en términos de diseño las 2 implementaciones planteadas.

Con respecto al tiempo de ejecución hay que considerar el análisis de complejidad Big-O del algoritmo Bellman-Ford, el cual es O(V*E), donde V representa la cantidad de nodos y E la cantidad de aristas. En ese sentido, es evidente que con cuantos más nodos y aristas tenga una topología, más tiempo se va a demorar y así se demuestra en los resultados. En promedio la topología 2 se demora 0.00872994 segundos más que la topología 1. Con respecto al manejo de memoria, es interesante que la diferencia sea mínima de casi 1 MB. Se podría esperar que topología 2 tenga un mayor uso de memoria. Sin embargo, tras varios intentos se ha podido identificar un patrón y es que el uso de memoria al llamar el algoritmo es casi lo mismo. Esto quizás se puede explicar por la naturaleza de Python en sí, dado que se trata de un lenguaje interpretado y la memoria de Python maneja todas las estructuras de datos creadas. Los métodos y variables creadas en la cola de memoria son destruidas automáticamente cuando ya no están siendo utilizados. Mientras que, los objetos instanciados se crean en la pila de memoria, y cuando ya no está siendo utilizados son recogidos por el recolector de basura. En ese sentido, no habría mucha diferencia a la hora de ejecutar ambas topologías, porque el mismo Python estaría manejando de la misma forma la memoria de forma dinámica con la finalidad de optimizar el proceso. Así que la diferencia de tamaño no importaría. Finalmente, con respecto al número de iteraciones. Aquí en ambos casos el resultado se ha mantenido constante, así que eso demuestra la eficacia del código. Siempre sale el mismo valor, porque por teoría el algoritmo Bellman-Ford se repite |v - 1| veces, donde v representa la cantidad de nodos.

Con respecto a los resultados obtenidos de la experimentación del algoritmo de Dijkstra propuesta por Fabrizio Figari, se puede apreciar que se mantiene la misma tendencia que el tiempo de ejecución en una topología con mayor aristas y nodos sería el que más se demore. Sin embargo, un elemento a criticar es el uso de memoria, ya que se esperaría que esta métrica sea evaluada por cada topología a fin de obtener mejores

observaciones. De todas formas, este trabajo sirve de referencia para comparar los 4 principios del diseño interactivo, así como se evidencia en la siguiente tabla:

Tabla 7.4:
Tabla de comparación de programas de experimentación usando criterios de diseño

Criterios	Implementación de Sebastián Chávarry	Implementación de Fabrizio Figari
Visibilidad: Las funciones del programa deben ser claras y visible para el usuario de tal forma que no tenga que memorizarlas.	La secuencia del programa es clara, porque en pantalla se pregunta al usuario lo que tiene que ingresar, de tal forma que es bastante intuitivo para tanto usuarios novatos y expertos. No es necesario que el usuario tenga que modificar ninguna parte del código fuente y si quiere probar su propia topología solo debe seguir la estructura dada en las instrucciones.	Si bien es sencillo interpretar los resultados obtenidos a la hora de ejecutar el programa, si uno quiere cambiar los parámetros tiene que realizarlo de manera manual modificando el código fuente. Esto sería un problema para personas que desconocen el código, puesto que les forzaría a tratar de entender que se está realizando y que implica cada parámetro
Retroalimentación: La información que recibe un usuario de vuelta cuando ejecuta una acción.	A la hora de ingresar los datos y leer las topologías existe un buen manejo de errores, lo cual sirven de orientación para que el usuario sepa, porque no está funcionando el programa. Solo en caso no encuentre su topología el programa se cerraría. Por otro lado, las métricas, tabla de distancias y rutas mostradas en pantalla son claramente visibles y distintas por los colores utilizadas.	Se muestran los resultados claros en pantalla y la estructura del reporte es claro. Sin embargo, no se incluyen manejo de errores y tampoco se permite al usuario seleccionar los parámetros de entrada para experimentar con el algoritmo.
Restricciones: Formas de restringir ciertos tipos de interacción en el estado actual de un sistema	Si bien solo se tiene una sola pantalla, el hecho de que el programa te pida los valores de entrada de forma secuencial y con manejo de errores ciertamente resulta beneficioso para mantener un orden en el programa y asegurar que los valores ingresados sean correctos.	En cierta forma, restringe cualquier otra acción, pues solo permite al usuario visualizar los resultados. Sin embargo, a la larga podría resultar perjudicial, porque un usuario que quiera experimentar con distintas topologías no podría tener mucha libertad, debido a la naturaleza del programa.
Consistencia: Las interfases deben tener operaciones y elementos similares.	Se utilizan distintos colores para distinguir las partes del programa, lo cual es una ayuda visual útil. Y los valores ingresados en consola corresponden a los parámetros utilizados durante la experimentación, así que hay lógica en el programa y se mantiene una secuencia ordenada a la hora de ver los resultados.	El usuario solo puede correr el programa, no existe diversidad de operaciones para el usuario. Sin embargo, se mantiene un mismo color para los resultados, lo cual da una sensación de uniformidad.

Nota. Elaboración Propia

VIII. CONCLUSIONES

A partir del trabajo realizado se llega a la conclusión de la importancia de contar con una buena interfaz de experimentación, ya que al estar en un ambiente de Ingeniería es probable que los usuarios deseen probar distintos parámetros para así evaluar las métricas empleadas. Esto es un punto que el programa implementado sí cumple, porque permite al usuario cambiar el tipo de topografía y nodo fuente con el cual experimentar. De esta forma, se obtiene una mejor experiencia de usuario y al mismo se puede visualizar mejores resultados prácticos. En ese sentido, se resaltan las buenas prácticas de Ingeniería de Software, porque estas permiten asegurar una buena experimentación.

Por otro lado, las métricas utilizadas (Tiempo de ejecución, Uso de Memoria y Número de Iteraciones) resultan útiles para comparar el efecto del algoritmo en distintos escenarios. No obstante, como se ha evidenciado, puede ser que existan factores externos al programa que afecten en los resultados como el lenguaje de programación. Además, considerando el análisis Big-O de Bellman-Ford era inevitable que la topología 2 se demore más. En ese sentido, para futuras experimentaciones sería interesante utilizar estos parámetros de evaluación utilizando el algoritmo de Dijkstra. En este lamentablemente no se pudo hacer, debido a las diferencias de código en las implementaciones seleccionadas. No obstante, se cumplió el objetivo de comparar las métricas utilizando 2 topologías distintas para analizar el comportamiento del algoritmo Bellman-Ford y se compararon en cuestiones de diseño las implementaciones vistas.

Además, para futuras implementaciones sería bueno incluir un ejecutable o interfaz gráfica de tal forma que los usuarios no tengan que ejecutar el código desde un IDE. Esto facilitaría una mejor visualización de las topologías y los resultados obtenidos. Asimismo, en función a la experimentación y la literatura revisada, se recomienda que para futuros experimentos se definan y establezcan criterios de comparación factibles entre distintos algoritmos y escenarios, de tal manera que se mantenga una uniformidad en la experimentación.

IX. REFERENCIAS

Abousleiman, R., & Rawashdeh, O. (2015). A Bellman-Ford approach to energy efficient routing of electric vehicles. *2015 IEEE Transportation Electrification Conference and Expo (ITEC)*. Published. https://doi.org/10.1109/itec.2015.7165772

Chan, S. Y. M., Adnan, N. A., Sukri, S. S., & Wan Zainon, W. M. N. (2016). An experiment on the performance of shortest path algorithm. https://core.ac.uk/download/pdf/78486805.pdf

Ming-Xin Yang, Bing-Tong Wang, & Wen-Dong Guo. (2009). Research on the performence of dynamic routing algorithm. *2009 International Conference on Machine Learning and Cybernetics*. Published. https://doi.org/10.1109/icmlc.2009.5212662

Kurose, J. F., & Ross., K. W. (2017). Redes de computadoras: Un enfoque descendente (7ma. ed). Madrid: Pearson.

Popa, B., & Popescu, D. (2016). Analysis of algorithms for shortest path problem in parallel. *2016 17th International Carpathian Control Conference (ICCC)*. Published. https://doi.org/10.1109/carpathiancc.2016.7501169

Rothkrantz, L. J. M. (2009). Dynamic routing using the network of car drivers. *Proceedings of the 2009 Euro American Conference on Telematics and Information Systems New Opportunities to Increase Digital Citizenship - EATIS '09*. Published. https://doi.org/10.1145/1551722.1551733

X. ANEXOS

Anexo 1: Repositorio de código en GitHub:

https://github.com/sebastianperudev2001/ComputerNetworks-FinalProject-BellmanFord