ESCUELA POLITÉCNICA NACIONAL

FACULTAD DE INGENIERÍA ELÉCTRICA Y ELECTRÓNICA

PROPUESTA DE UN ALGORITMO DE SOLUCIÓN DIRECTA PARA RESOLVER EL PROBLEMA DE ASIGNACIÓN DE RUTAS Y LONGITUDES DE ONDA (RWA) UTILIZANDO LOS MÉTODOS DIJKSTRA Y FIRST-FIT

TRABAJO DE TITULACIÓN PREVIO A LA OBTENCIÓN DEL TÍTULO DE INGENIERO EN ELECTRÓNICA Y TELECOMUNICACIONES

MONTALVO POVEDA PABLO SEBASTIAN

DIRECTOR: DR. LUIS FELIPE URQUIZA AGUILAR

Quito, abril 2022

AVAL

Certifico que el presente trabajo fue desarrollado por Pablo Sebastian Montalvo Poveda, bajo mi supervisión.

DR. LUIS FELIPE URQUIZA AGUILAR

DIRECTOR DEL TRABAJO DE TITULACIÓN

DECLARACIÓN DE AUTORÍA

Yo, Pablo Sebastian Montalvo Poveda, declaro bajo juramento que el trabajo aquí descrito es de mi autoría; que no ha sido previamente presentada para ningún grado o calificación profesional; y, que he consultado las referencias bibliográficas que se incluyen en este documento.

A través de la presente declaración cedo mis derechos de propiedad intelectual correspondientes a este trabajo, a la Escuela Politécnica Nacional, según lo establecido por la Ley de Propiedad Intelectual, por su Reglamento y por la normatividad institucional vigente.

PABLO SEBASTIAN MONTALVO POVEDA

DEDICATORIA

A mis padres, que con esfuerzo me han traído por este camino y sin ellos nada hubiese sido posible.

AGRADECIMIENTO

A Dios, a mis padres, a mis hermanas, a mis amigos y a mis profesores, que pusieron su granito de arena en cada paso que la poli me puso en frente.

Al Ing. Marco Yacelga MSc. y al Dr. Luis Felipe Urquiza, quienes me guiaron de inicio a fin; sin su orientación este proyecto no hubiese sido posible.

ÍNDICE DE CONTENIDO

AVAL I
DECLARACIÓN DE AUTORÍA II
DEDICATORIA III
AGRADECIMIENTO IV
ÍNDICE DE CONTENIDO V
RESUMEN VII
ABSTRACT VIII
1. INTRODUCCIÓN 1
1.1. OBJETIVOS 1
1.2. ALCANCE 1
1.3. MARCO TEÓRICO 4
1.3.1. MULTIPLEXACIÓN POR DIVISIÓN DE TIEMPO (TDM) 4
1.3.2. WAVELENGTH DIVISION MULTIPLEXING (WDM) 5
1.3.3. SISTEMAS WDM GRUESA (CWDM) 7
1.3.4. CWDM EN REDES METROPOLITANAS 7
1.3.5. SISTEMAS WDM DENSA (DWDM) 7
1.3.6. DWDM EN REDES METROPOLITANAS 8
1.3.7. LA RED ÓPTICA DE TRANSPORTE COMO UN GRAFO 9
1.3.8. PROBLEMA DE ENRUTAMIENTO Y ASIGNACIÓN DE LONGITUDES DE ONDA (RWA) 11
1.3.9. ASIGNACIÓN DE RUTAS EN RWA 14
1.3.10. ALGORITMOS DE ENRUTAMIENTO DE CAMINO MÁS CORTO 16
1.3.11. ALGORITMO DE DIJKSTRA 17
1.3.12. EL PROBLEMA DE RUTAS DESBALANCEADAS CON EL ALGORITMO DE DIJKSTRA 19
1.3.13. ASIGNACIÓN DE LONGITUDES DE ONDA EN RWA 21
2. METODOLOGÍA 25
2.1. VARIABLES 28
2.2. DESARROLLO DEL PROGRAMA PRINCIPAL 29
2.3. DESARROLLO DEL MÓDULO DE FUNCIONES DIJKSTRA-FIRSTFIT 46
2.4. DESARROLLO DE LA SOLUCIÓN TEÓRICA 57

3. RESULTADOS Y DISCUSIÓN 66
4. CONCLUSIONES Y RECOMENDACIONES 76
4.1. CONCLUSIONES 76
4.2. RECOMENDACIONES 78
5. REFERENCIAS BIBLIOGRÁFICAS 79
ANEXOS I
ANEXO A I
ANEXO B III
ANEXO C V
ANEXO D XI
ORDEN DE EMPASTADO 1

RESUMEN

El objetivo del presente proyecto es resolver el problema de enrutamiento y asignación de longitudes de onda (RWA) de manera directa en redes de transporte ópticas con nodos interconectados en topología de anillo que disponen de conversores de longitudes de onda en todos los elementos de red y en todos sus enlaces. El desarrollo se realiza en los 4 capítulos resumidos a continuación.

En el capítulo 1 se presenta el problema de enrutamiento y asignación de longitudes de onda (RWA), y los algoritmos propuestos para la solución directa del problema, incluyendo el algoritmo de enrutamiento de camino más corto *Dijkstra* y el algoritmo de asignación de longitudes de onda *First-Fit*.

En el capítulo 2 se muestra la metodología usada para el desarrollo del algoritmo propuesto. El programa diseñado incluye un módulo de variables, en el cual se definen las variables de entrada del programa, incluyendo la matriz de topología omega, la matriz de flujos y el valor de la capacidad de la red. Adicionalmente, se desarrolla el programa principal y el módulo de funciones implementado para la solución propuesta. Este capítulo finaliza con la descripción del código implementado en el programa de optimización MiniZinc para obtener la solución óptima teórica para cada problema planteado.

El capítulo 3 describe las pruebas realizadas sobre el algoritmo de optimización propuesto y los resultados obtenidos para los mismos escenarios en el programa de optimización MiniZinc. Estos resultados son ordenados, tabulados y comparados, con el objetivo de determinar la factibilidad de utilizar el algoritmo propuesto en sistemas reales, optimizando una red óptica.

El capítulo 4 muestra las conclusiones determinadas en base a los resultados obtenidos en el capítulo previo.

PALABRAS CLAVE: *enrutamiento, redes ópticas, enrutamiento y asignación de longitudes de onda, RWA, optimización, Python, MiniZinc*

ABSTRACT

The objective for this degree project is solving routing and wavelength assignment problem (RWA) in a direct way in optical transport networks with ring topology interconnected nodes which dispose wavelength converters in all network elements and in all their links. The development is described in the 4 chapters described next.

Chapter 1 presents routing and wavelength assignment problem and algorithms purposed to solving this optimization problem. This chapter includes the description for Dijkstra's shortest path routing algorithm and First Fit wavelength assignment algorithm.

Chapter 2 shows used methodology for purposed algorithm development. Designed program includes a module for variables, where it is defined algorithm input variables, including topology matrix defined as *omega*, flows matrix defined as *flujos* and maximum links capacity defined as *capacidad*. Additionally, it is developed main program and function's module code. This chapter finishes describing optimization software code, developed on MiniZinc for finding theoretical optimum solution for each scenario.

Chapter 3 describes tests scenarios defined for purposed algorithm analysis and for the comparison using optimization software MiniZinc. These results are ordered, tabulated and compared, to determine the feasibility of using purposed algorithm on real systems, to optimize an optical network.

Chapter 4 shows conclusions found based on results obtained on previous chapter.

KEYWORDS: *routing, optical networks, routing and wavelength assignment, RWA, optimization, Python, MiniZinc*

1. INTRODUCCIÓN

1.1. OBJETIVOS

El objetivo general de este Proyecto Integrador es: Implementar un algoritmo capaz de obtener una solución directa para la asignación de rutas y longitudes de onda (RWA).

Los objetivos específicos de este Proyecto Integrador son:

- Describir las principales características técnicas del problema de enrutamiento y asignación de longitudes de onda (RWA).
- Definir un algoritmo capaz de encontrar una solución directa para el problema RWA.
- Diseñar un programa informático que permita resolver el problema y presentar los resultados en forma de matrices.
- Comparar y analizar los resultados del algoritmo implementado con las respuestas obtenidas en base a un software de optimización de uso general.

1.2. ALCANCE

El presente trabajo pretende implementar un algoritmo para RWA, utilizando un programa informático que realizará dos tareas: determinar la ruta para cada *lightpath* y asignar una longitud de onda adecuada para cada enlace; debe notarse que el algoritmo desarrollado se enfocará en el problema de enrutamiento de *lightpaths*; para la asignación de longitudes de onda se escogerá un algoritmo conocido [1].

La primera tarea utilizará el algoritmo de Dijkstra para el problema del camino más corto y busca encontrar un costo mínimo para un conjunto de nodos pertenecientes al grafo. Debido a que el algoritmo diseñado no utilizará todas las variables al mismo tiempo para el proceso de optimización, sino que realizará una iteración por cada *lightpath* utilizando la solución de Dijkstra, el resultado podría no ser el óptimo.

El costo de utilización de los enlaces de la red se calculará utilizando una función matemática lineal creciente definida por partes, la cual se indica en la Figura 1.1 y es típica para el problema de RWA [2]. La función de costo definida presenta los siguientes valores: *"fl"* es el flujo que atraviesa un enlace unidireccional, mientras que *"Dl (fl)"*, representa el costo de utilización del enlace. El flujo máximo que puede atravesar una fibra en cualquier

sentido se define como |*C*| (la capacidad de la fibra medida en número de *lightpaths* que pueden atravesar).

La segunda tarea consiste en asignar una longitud de onda adecuada para cada enlace, considerando una red con conversores de longitud de onda en todos los nodos y en todos sus enlaces. Esto permitirá concentrar el trabajo en el algoritmo implementado. Para ello se utilizará la solución para la asignación de longitudes de onda conocida como *first-fit* [1].

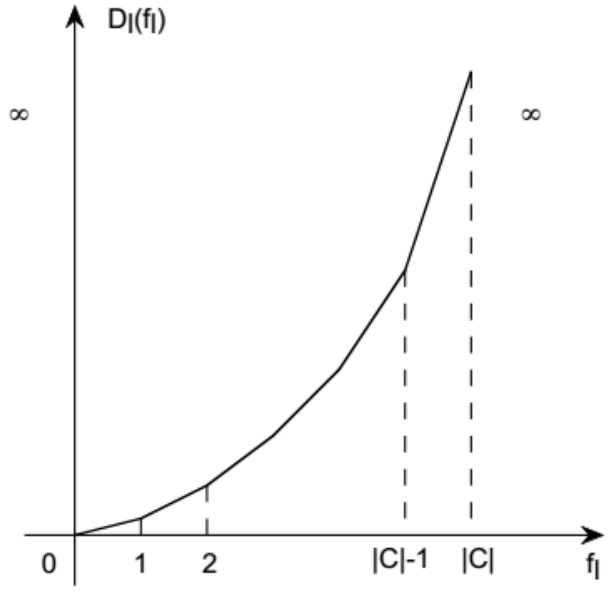

Figura 1.1. Función de costo, lineal definida por partes, para una red con capacidad máxima |C| **[2]**

El programa resultante mostrará los siguientes resultados:

Una matriz correspondiente al resultado del proceso de enrutamiento (con la información correspondiente a los nodos que el *lightpath* atraviesa desde el nodo origen hasta el nodo destino).

Una matriz con la información de las longitudes de onda asignadas a cada *lightpath* en cada enlace.

El costo total de la red según el resultado de la optimización. Este valor será calculado en base al número de *lightpaths* que atraviesan cada enlace y al costo correspondiente según la Figura 1.1.

Las topologías que se analizarán se definen como topologías en anillo como las mostradas en la Figura 1.2.

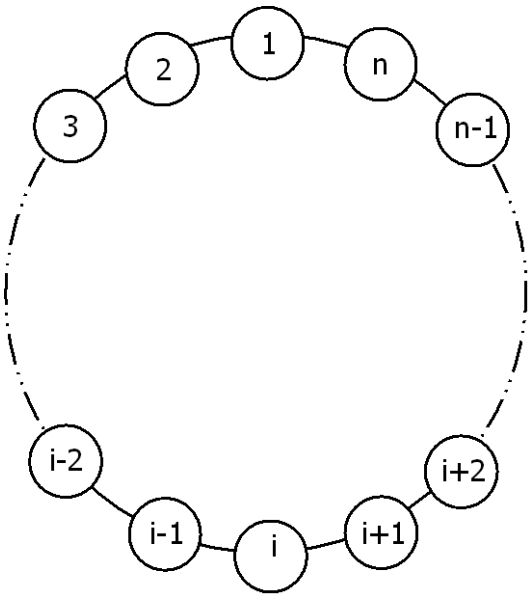

Figura 1.2. Topología de anillo

De hecho, se selecciona este tipo de conexión debido a su amplia utilización en redes ópticas metropolitanas.

Para analizar los resultados obtenidos con el algoritmo implementado, se realizará la comparación del costo total de la red obtenido con el valor que arroje un programa de optimización de uso general. Para ello se realizará la variación de dos parámetros: el número de longitudes de onda que se optimizan y el número de nodos en la red. Esto conlleva a cuatro escenarios diferentes basándose en las peores condiciones de cada caso:

- Una pequeña cantidad de longitudes de onda con una pequeña cantidad de nodos.
- Una pequeña cantidad de longitudes de onda con una gran cantidad de nodos.
- Una gran cantidad de longitudes de onda con una pequeña cantidad de nodos.
- Una gran cantidad de longitudes de onda con una gran cantidad de nodos.

De esta manera se analizará la efectividad del algoritmo propuesto en estos escenarios, contrastando la respuesta obtenida con los resultados teóricos, según el resultado del software de optimización de uso general.

1.3. MARCO TEÓRICO

La multiplexación de señales permite utilizar un medio de transmisión de manera compartida para la transmisión de varias comunicaciones simultaneas, optimizando de esta manera el uso del canal e incrementando su capacidad [3] [4]; cuando el medio de transmisión es óptico, se implementan fundamentalmente dos mecanismos: multiplexación por división en tiempo (TDM por sus siglas en inglés) y multiplexación por división en longitudes de onda [4].

1.3.1. MULTIPLEXACIÓN POR DIVISIÓN DE TIEMPO (TDM)

Los sistemas de multiplexación de señales por división de tiempo (TDM) envían la información de los abonados utilizando un canal multiplexado con velocidades mayores a las velocidades utilizadas por los suscriptores [5], como se muestra en la Figura 1.3 . En este sistema, cada usuario dispone de una o varias ranuras de tiempo (conocidas también como *time slot* o *slot*) para transmitir y solo puede enviar datos durante este intervalo [6] utilizando todo el ancho de banda del canal [7]; por este motivo, las señales no interfieren entre sí [5].

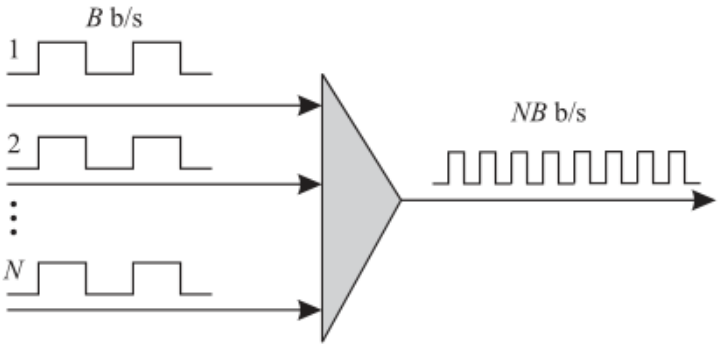

Figura 1.3. Multiplexación por división en tiempo (TDM) **[5]**

Con el objetivo de incrementar las velocidades de transmisión de información que permiten los sistemas TDM, las investigaciones se han concentrado es sistemas que permitan realizar la multiplexación y demultiplexación de las señales de manera óptica, los cuales se conocen como multiplexación óptica por división de tiempo (OTDM por sus siglas en inglés) [5]. Utilizando esta técnica, es posible alcanzar velocidades de 1 Tbps, las cuales son imposibles de alcanzar por medios electrónicos; es análogo a TDM, pero las señales que se multiplexan pertenecen al dominio óptico [8].

1.3.2. WAVELENGTH DIVISION MULTIPLEXING (WDM)

La técnica de multiplexación por división de longitudes de onda WDM es una tecnología diseñada para trabajar en el espectro óptico; asigna una o varias frecuencias específicas dentro del espectro óptico para cada señal de entrada al sistema; estas frecuencias son conocidas como longitudes de onda o lambdas (λ) [9] como se muestra en la Figura 1.4. De esta manera, un conjunto de portadoras puede ser transmitido sobre un único hilo de fibra, sin interferir entre ellas [5]; además, la división en canales de menor velocidad permite adaptar las señales ópticas a las velocidades de procesamiento electrónico de los usuarios finales [2]. Su funcionamiento es similar a FDM (Multiplexación por División de Frecuencia), pero se considera una tecnología diferente ya que la información viaja utilizando fotones en vez de ondas electromagnéticas.

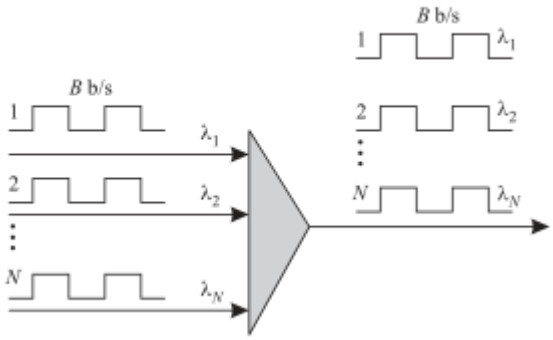

Figura 1.4. Multiplexación por división de longitudes de onda (WDM) **[5]**

Debido a la asignación de diferentes longitudes de onda para cada señal, todas pueden transmitir información en el canal al mismo tiempo, utilizando únicamente el ancho de banda asignado para esa comunicación, es decir, no utilizan todo el ancho de banda disponible en el canal para enviar información. Los sistemas WDM trabajan en conjunto con multiplexación TDM (Multiplexación por División de Tiempo) para incrementar al

máximo la capacidad de la fibra óptica; esta implementación permite obtener velocidades de 12.8 Tbps sobre un único hilo de fibra óptica [8], lo cual evidencia la gran capacidad de este tipo de sistemas.

Al considerar la confiabilidad del sistema, deben tomarse en cuenta dos aspectos: primero, la fibra óptica es un medio de transmisión confiable [7], pues es inmune a interferencia electromagnética (EMI) [8]; segundo, los sistemas WDM permiten incrementar en mayor medida la confiabilidad del sistema, pues, disminuyen el número de amplificadores ópticos requeridos, conforme aumenta el número de longitudes de onda multiplexadas en cada hilo de fibra óptica [8].

Estos sistemas poseen la capacidad de amplificar todas las longitudes de onda a la vez, sin la necesidad de convertirlas a señales eléctricas; además, pueden transportar varias señales de diferentes tipos de datos, diferentes velocidades y diferentes protocolos simultáneamente, sin interferencia entre ellas. Cada canal puede transportar tráfico homogéneo (todas las longitudes de onda llevan tráfico de una misma tecnología como SONET/SDH) o heterogéneo (diferentes longitudes de onda transfieren información de diferentes tecnologías, como SONET/SDH, TDM, IP, etc.). Por estos motivos, WDM puede considerarse un protocolo transparente [9] [4]. Por estos motivos, los sistemas WDM son ampliamente utilizadas en la actualidad para construir redes long-haul y metropolitanas [5].

Si se considera el número de longitudes de onda que se multiplexan en un canal, y el espaciamiento entre cada longitud de onda [10], se puede clasificar a esta tecnología en dos diferentes sub tecnologías: DWDM (WDM densa) y CWDM (WDM gruesa), las cuales se explican en la siguiente sección.

Los sistemas WDM alcanzan su máxima velocidad cuando se implementan en redes completamente ópticas (*all optical networks*), las cuales transmiten la señal como una onda de luz en todo su trayecto, evitando así el “cuello de botella” generado por la conversión entre una señal óptica a eléctrica y eléctrica a óptica.

Para atender las nuevas solicitudes de comunicación que ingresan a este tipo de redes, es necesario asignar a cada una los siguientes parámetros: la ruta física que el haz de luz atravesará desde su origen hasta su destino (conocido también como *lightpath*), y una longitud de onda para cada enlace que el *lightpath* atraviesa. Esta asignación influye en la disponibilidad de recursos para nuevas conexiones y, por consiguiente, en el número de nuevas solicitudes de conexión que se rechazan, las cuales representan un costo para el operador de la red.

1.3.3. SISTEMAS WDM GRUESA (CWDM)

Dentro de los sistemas ópticos WDM, se encuentra la multiplexación por división en longitudes de onda gruesa o *Coarse WDM* (CWDM, conocida también como multiplexación por división aproximada de longitud de onda) [11]. Este mecanismo se caracteriza por un espaciamiento entre canales ópticos más ancho comparado con los sistemas WDM tradicionales con una separación entre cada longitud de onda menor a 50nm, pero mayor a 1000 GHz (alrededor de 8 nm en la banda de 1550 nm y 5.7 nm en la banda de 1310 nm); con estas características, el sistema puede utilizar varias bandas del espectro óptico [12] [11].

La utilización de un espacio amplio entre cada longitud de onda evita que los láseres fabricados requieran dispositivos de control de temperatura, además de ser tolerantes a errores de sintonización. Conjuntamente, los filtros pasa-banda utilizados en la recepción funcionan sobre un amplio rango de frecuencias y son menos selectivos. Estas características permiten que los dispositivos de comunicación utilizados sean más sencillos de fabricar y menos costosos; el principal objetivo de este tipo de tecnología es la reducción de costos para obtener una mayor rentabilidad sobre la infraestructura de comunicaciones resultante [11].

Este tipo de tecnología ha sido diseñada para funcionar en sistemas de transporte metropolitanos y como plataforma integrada para una diversidad de clientes y servicios.

1.3.4. CWDM EN REDES METROPOLITANAS

Las corporaciones y proveedores de servicios de telecomunicaciones construyeron sus infraestructuras de redes metropolitanas sobre sistemas CWDM, debido a la escalabilidad que este tipo de redes provee con un bajo costo de inversión. Las topologías típicas para redes metropolitanas (las cuales coinciden para CWDM) son: anillo y punto a punto [13].

1.3.5. SISTEMAS WDM DENSA (DWDM)

Los sistemas de multiplexación por longitud de onda densa (DWDM por sus siglas en inglés) son parte de la tecnología WDM [11]. Utilizan un espaciamiento entre canales ópticos menor al utilizado en sistemas WDM tradicionales, con el cual, se obtienen un

número de longitudes de onda mayor o igual a 40 [4], y pueden utilizar una o varias bandas del espectro óptico [12].

En contraste con CWDM, los dispositivos fabricados para DWDM necesitan mayor precisión respecto a la longitud de onda generada, regenerada o filtrada; además, requieren de mecanismos de control para mantener estable la longitud de onda, los cuales son poco utilizados en dispositivos CWDM; en consecuencia, son más costosos; su ventaja frente a sistemas CWDM es que ofrecen mejores prestaciones en la distancia máxima del enlace, mejor la velocidad de transmisión y una mayor cantidad de longitudes de onda enviadas en cada hilo de fibra óptica.

Estos sistemas son utilizados típicamente en sistemas de largo alcance (*long-haul*) y metropolitanos [8].

Inicialmente los sistemas DWDM se utilizaban únicamente con topologías punto a punto. Hoy en día, se amplía el número de topologías utilizadas: punto a punto con capacidad de añadir o retirar información (*add-drop*), anillo, malla completa y estrella [4].

1.3.6. DWDM EN REDES METROPOLITANAS

Las redes *long-haul*, (de largo alcance), utilizan preferentemente sistemas de multiplexación DWDM, debido a su capacidad para llevar gran cantidad de información sobre largas distancias; además, estos sistemas empezaron a implementarse en una época en que las fibras ópticas escaseaban, utilizando estos sistemas para transportar una mayor cantidad de información sobre la red previamente instalada [9].

En sistemas DWDM, el aprovisionamiento de servicios, planeación y análisis de la red, reconfiguración de equipamiento, verificación de camino y de circuito, y la creación de nuevos servicios se vuelven sencillos. El implementar un nuevo servicio puede ser tan sencillo como habilitar una nueva longitud de onda sobre una conexión existente. De esta manera, se reduce el tiempo de despliegue de nuevos servicios sobre la red existente, reduciendo el tiempo en que se perciben ganancias sobre el servicio ofrecido utilizando la red instalada [9].

1.3.7. LA RED ÓPTICA DE TRANSPORTE COMO UN GRAFO

En sistemas de telecomunicaciones, se ha considerado en diferentes ocasiones la posibilidad de representar la red y todas sus conexiones como un grafo matemático, el cual pueda ser analizado utilizando la teoría de grafos. Este precisamente, ha sido el análisis utilizado en la resolución de problemas ampliamente conocidos en redes, como es el caso de: árbol de expansión (conocido también como *Spanning Tree* o SPT), problema del camino más corto (conocido como *Shortest Path First* o SPF), problema del viajante (*Traveling Salesman Problem* o TSP), entre otros.

Es evidente la ventaja de analizar una red de transporte como un sistema matemático, pues se pueden introducir conceptos y soluciones matemáticas ampliamente conocidas en la solución de un problema de telecomunicaciones. Además, el problema RWA se analiza como un problema de teoría de grafos. Por estos motivos, se mencionarán a continuación nociones generales de teoría de grafos, las cuales serán suficientes para entender el problema RWA (el cual será tratado en la siguiente sección) y comprender la solución propuesta.

En una gran cantidad de aplicaciones del mundo real, es conveniente describir a un problema como un conjunto de puntos y un conjunto de líneas que interconectan estos puntos [14] [15]. Estos puntos y sus interconexiones forman un gráfico que se conoce como grafo, dentro del cual los objetos interconectados se describen con el nombre de vértices y sus interconexiones como aristas o enlaces. Por ejemplo, un grafo puede representar un conjunto de ciudades (vértices) que se interconectan por medio de vías de transporte terrestre (aristas); con el análisis matemático de este sistema, es posible determinar la cantidad de tráfico que debe ser redirigido por vías alternas para evitar congestión. Este es solo un ejemplo de las aplicaciones posibles de la teoría de grafos en la solución de problemas del mundo real.

La formulación anterior es suficiente para comprender de manera general el concepto de teoría de grafos y sus principales componentes. Sin embargo, la definición matemática formal del problema no es tan intuitiva, pero es necesaria para una comprensión íntegra de la formulación; por este motivo, es presentada a continuación.

Un grafo $G = (V, E)$ es una estructura que consiste en un conjunto finito de vértices $V \neq \emptyset$ y un conjunto finito de enlaces o aristas E. Las aristas constan de un par de vértices, poseen la forma (u, v), y representan la unión de los vértices u y v. De esta manera, el grafo puede ser llamado “grafo simple finito”, lo cual significa que no posee “múltiples aristas” ni “lazos”;

las múltiples aristas representan varios enlaces interconectando el mismo par de vértices; por otro lado, los lazos representan enlaces que conectan los vértices consigo mismos [16].

Esta teoría diferencia a los grafos en dos grupos, en base a la directividad de sus enlaces. Esta característica, permite asignar a un enlace una dirección; esta cualidad es muy útil en problemas de tráfico o transporte, pues permite diferenciar una "vía de un solo sentido" de vías bidireccionales. Así, existen grafos no dirigidos (los cuales se consideran simplemente grafos) y los grafos dirigidos (conocidos también como digrafos). Los grafos dirigidos asignan una dirección a cada uno de sus enlaces; en este caso, para considerar enlaces bidireccionales es necesario utilizar dos enlaces diferentes entre los vértices correspondientes. Por otro lado, los grafos no dirigidos asumen que todos de sus enlaces son bidireccionales, por lo cual no necesitan de asignación de direcciones [17].

El presente trabajo procura aplicar su estudio en redes ópticas de transporte, por lo cual, es necesario traducir el problema propuesto para sistemas ópticos. Para ello, se define los vértices V como los nodos de una red óptica, los cuales pueden ser OADMs, OXCs u OLTs; las aristas E son representadas por los enlaces que interconectan los nodos, los cuales son enlaces de fibra óptica. Por otro lado, sería sencillo pensar en el problema de múltiples aristas como algo común en una red óptica, en la que se instalan varios hilos de fibra óptica entre dos equipos para incrementar la capacidad de transmisión de información entre ambos; sin embargo, para el presente trabajo, se considerarán estos múltiples enlaces como una sola conexión con una cantidad mayor de longitudes de onda WDM disponibles entre ellos. Esta suposición no se encuentra lejos de la realidad, pues se ha desarrollado mecanismos que permiten enlazar este tipo de conexiones y hacerlas trabajar como un solo enlace; este es el caso de los mecanismos de agregación de enlaces (que escapan al alcance del presente trabajo, por lo cual se deja a discreción del lector ampliar sus conocimientos respecto a este tema). Además, el problema de lazos en una red óptica puede ser tratado como inexistente (al considerar la definición de lazo para un grafo), pues no se realizan conexiones de un equipo consigo mismo. Finalmente, un grafo dirigido considera enlaces unidireccionales en la red (simplex), por lo cual transmite información en un solo sentido; por consiguiente, un grafo no dirigido considera enlaces bidireccionales (*half duplex* o *full duplex*); estas dos últimas consideraciones son factibles en las redes actuales, pero la capacidad de transmitir información de manera bidireccional es una característica que se ha extendido ampliamente, pues la instalación de un solo cable de fibra óptica permite una comunicación bidireccional (ya sea por el uso de técnicas de

multiplexación como WDM o por la instalación de dos hilos de fibra por cada cable), siendo una característica útil y deseable.

Por último, debe notarse que ya se ha mostrado un ejemplo de representación de grafos con anterioridad en el presente trabajo, pero no se lo ha denotado como tal; este es el caso de la imagen de la Figura 1.2, el cuál corresponde a una ilustración que cumple con la definición de grafos no dirigidos.

1.3.8. PROBLEMA DE ENRUTAMIENTO Y ASIGNACIÓN DE LONGITUDES DE ONDA (RWA)

El proceso de diseño de una red de transporte busca maximizar el número de comunicaciones que pueden ser establecidas utilizando la infraestructura disponible, incrementando las ganancias percibidas por el propietario de la red y, en consecuencia, disminuyendo el costo asociado al funcionamiento del sistema [5]. Se han propuesto diferentes métodos de optimización para una red de transporte óptica; uno de los más difundidos se conoce como: enrutamiento y asignación de longitudes de onda (*Routing and Wavelength Assignment* o RWA), el cual puede ser formulado de la siguiente manera [2] [18]:

En una red óptica, de la cual se conoce su estructura física y las conexiones requeridas de extremo a extremo, se busca determinar para cada *lightpath* los siguientes parámetros:

- La ruta de la comunicación, la cual se define como el conjunto de nodos que atraviesa desde el origen de la información hasta su destino, los cuales pueden ser elementos de acceso a la red de transporte (como OLTs) y se localizan en el borde del sistema.
- La longitud de onda asignada en cada enlace que el *lightpath* atraviesa, que representa la frecuencia que utilizará la comunicación en cada enlace. Además, debe considerarse que no puede utilizarse la misma longitud de onda, más de una vez en el mismo enlace óptico WDM.

Estos dos parámetros se seleccionan, debido a su gran importancia en el diseño de una red óptica de transporte WDM, pues su asignación influye en la probabilidad de bloqueo de las nuevas solicitudes de conexión [2].

Los estudios desarrollados acerca de este tema [2] [18], evidencian la posibilidad de analizar el problema con tres diferentes solicitudes de conexión por parte del cliente:

- Tráfico estático: este tipo de tráfico es conocido a priori, por lo cual, la solución del problema RWA analizará de manera global todas las variables e intentará determinar la solución de manera integral, tratando de minimizar los recursos utilizados por el sistema. En consecuencia, se asume que no existirán nuevas solicitudes de conexión.
- Tráfico incremental: las solicitudes de conexión arriban secuencialmente. En este caso, el *lightpath* permanecerá en la red por un tiempo indefinido.
- Tráfico dinámico: las solicitudes ingresan al sistema secuencialmente; a diferencia del tráfico incremental, la conexión se libera después de un tiempo y los recursos que utiliza pueden ser utilizados para una nueva solicitud.

La solución del problema RWA con un tráfico estático, trata de minimizar la cantidad de recursos utilizados por el sistema. En contraste, en los casos de tráfico incremental y dinámico, se trata de minimizar la probabilidad de bloqueo del sistema para nuevas solicitudes [18].

Además, debe considerarse la capacidad de conversión de longitudes de onda en los equipos de comunicación; de esta manera, en la mayoría de los estudios realizados se destaca la existencia de dos casos diferentes que constituyen dos problemas RWA diferentes, los cuales serán:

- Conversión completa de longitudes de onda, en el cual todos los equipos poseen la capacidad de convertir una señal recibida a otra frecuencia en el enlace de salida [1] [5].
- Sin conversión de longitudes de onda, en la que todos los equipos carecen de la capacidad de conversión de frecuencias. En este caso, un *lightpath* establecido debe utilizar la misma longitud de onda en todos los enlaces de todos los nodos de la red; esta condición, se conoce como restricción de continuidad de longitud de onda [1] [5].

Si bien, la conversión completa de longitudes de onda es deseable en la mayoría de los casos por la versatilidad que permite a la red [2], no siempre se puede aplicar por sus altos costos de implementación. Por estos motivos, una gran cantidad de redes poseen capacidad parcial de conversión de longitudes de onda o carecen de esta característica.

Existen a su vez, diferentes formulaciones matemáticas propuestas en base a variados estudios, los cuales difieren entre sí según el enfoque del problema (por ejemplo, minimizar el costo de utilización de la red, maximizar el número de conexiones establecidas, minimizar la probabilidad de bloqueo, entre otras) y los cambios en su función objetivo (determinados por el enfoque del problema).

La formulación matemática general de este problema [2] se muestra en la ecuación 1.1., Esta formulación está sujeta a restricciones de conservación de flujo y a cualquier otra restricción especial adicional.

$$\boldsymbol{minimizar} \qquad \sum_{l \in L} \boldsymbol{D_l(f_l)} \tag{1.1}$$

Para la formulación anterior se muestran las siguientes definiciones: l representa un enlace dentro del conjunto L de enlaces, f_l se considera como el flujo total sobre el enlace y $D_l(f_l)$ es la función de costo del enlace.

Si se considera una red con conversores de longitud de onda en todos sus enlaces, no existe limitación sobre las longitudes de onda que puedan ser asignadas en cada caso; por ello se elimina la restricción de continuidad de longitud de onda. [2]

El problema puede ser reformulado en base a esta suposición de la siguiente manera:

Se dispone de un grafo conexo, no dirigido $G = (V, E)$, donde V denota el conjunto de nodos y E denota el conjunto de enlaces. Cada enlace representa un par de fibras unidireccionales en direcciones opuestas. Los pares origen-destino (OD) se expresan como un par ordenado $w = (i, j)$ para distintos nodos i y j, que pertenecen al conjunto W. El tráfico de entrada para un par OD de w se representa con r_w, el cual es un entero no negativo, que simboliza el número de solicitudes de *lightpaths* del nodo i destinados al nodo j. P_w será el conjunto de caminos que los pares OD pueden utilizar y C es el conjunto de longitudes de onda disponibles en cada enlace. Además, se define x_p como el flujo para un camino $p \in P_w$ y toma un valor entero no negativo. [2]

Matemáticamente se expresa esta formulación en la Ecuación 1.2. [2]:

$$\begin{aligned} &\boldsymbol{minimizar} && \sum_{\boldsymbol{l \in L}} \boldsymbol{D_l(f_l)} \\ &\boldsymbol{sujeto\ a} && \sum_{\{\boldsymbol{p} | \boldsymbol{l} \in \boldsymbol{p}\}} \boldsymbol{x_p} \leq |\boldsymbol{C}|, \ \ \boldsymbol{para\ todo\ l} \in \boldsymbol{L}, \end{aligned} \tag{1.2}$$

$$\sum_{p \in P_w} x_p = r_w, \quad \textbf{para todo } w \in W,$$
$$x_p : \textbf{entero no negativo, para todo } p \in P_w, w \in W$$

La variable x_p puede tomar valores enteros no negativos [2]; por esta razón, el problema pertenece a programación lineal entera (ILP por sus siglas en inglés). La solución de este tipo de problemas se conoce como *NP-complete* [19], lo cual requiere una gran cantidad de recursos, e incrementa en complejidad con el aumento de variables analizadas [19]; esto significa que la complejidad de la solución incrementa con el aumento del número de nodos y enlaces en la red. Por este motivo, es necesario implementar un algoritmo alternativo, que permita encontrar de manera directa una solución para el problema de enrutamiento y asignación de longitudes de onda, reduciendo su complejidad. Se han propuesto diferentes formulaciones que permiten reducir la complejidad del problema. La más difundida, parte de la posibilidad de dividir la solución en dos subproblemas, considerando la asignación de rutas y longitudes de onda por separado [1] [5]. Este enfoque, permite reducir la complejidad del problema, con la desventaja de analizar las variables separado; en consecuencia, el resultado del proceso de optimización obtendrá un resultado diferente al óptimo. Sin embargo, bajo las restricciones mencionadas en la sección 1.2, se puede obtener un resultado igual o muy cercano al óptimo, lo cual permite pensar en este procedimiento, como un método factible para solucionar el problema RWA.

La nueva formulación, permite analizar los dos subproblemas por separado, y considerar diferentes soluciones para cada caso. De esta manera, se estudiará las posibles soluciones de cada problema en la siguiente sección.

1.3.9. ASIGNACIÓN DE RUTAS EN RWA

La asignación de rutas en RWA es el primer subproblema de la formulación propuesta; su solución consiste en asignar una ruta para el *lightpath* cuyo origen y destino es conocido con anterioridad. Al considerar el enunciado anterior, es fácil pensar en la utilización de algoritmos de enrutamiento para realizar el cálculo de la ruta; sin embargo, no es la única opción factible, como lo muestran la gran cantidad de estudios realizados en torno a este problema. Algunas posibles soluciones son:

- *Lightpath Topology Design* (LTD) [5], cuyo propósito general es encontrar la topología de *lightpath* extremo a extremo que minimice el número de recursos necesarios y, por consiguiente, reduzca el costo de operación de la red. Este

problema se considera en general como un problema matemático de gran complejidad, lo cual conlleva a considerar otras opciones.

- Algoritmos de enrutamiento de camino más corto [5], los cuales han sido ampliamente considerados en protocolos de red tradicionales; este es el caso de *Open Shortest Path First* e *Intermediate Sistem to Intermediate System* (conocidos también como OSPF o IS-IS respectivamente), los cuales fundamentalmente funcionan con algoritmos diseñados para encontrar una longitud mínima sobre un conjunto de nodos pertenecientes a un grafo.
- Métodos heurísticos [2], los cuales han sido los más difundidos, pues la mayoría de los estudios se han enfocado en buscar una solución matemática igual o muy cercana a la óptima. Esta solución busca determinar las rutas (y en algunos casos también las longitudes de onda) para el problema RWA, utilizando aproximaciones matemáticas y suposiciones válidas dentro de un cierto rango; esta solución permite reducir la complejidad matemática y computacional del problema. Aun así, su grado de complejidad es relativamente elevado, sobre todo al aplicarse a un problema con una gran cantidad de variables.

La utilización de algoritmos de enrutamiento ha sido nombrada en reiteradas ocasiones en los trabajos previos realizados acerca del tema [1] [5], pero su aplicación práctica ha sido poco estudiada. Para el presente trabajo, se propone el uso de estos algoritmos, cuyo funcionamiento es ampliamente conocido y son sencillos de implementar, en comparación con los otros dos métodos antes mencionados.

En este sentido, el estudio realizado en [20] merece mencionarse, pues compara la eficiencia en el uso de recursos entre el algoritmo de Dijkstra y un algoritmo propuesto por los investigadores (que utiliza métodos heurísticos). El resultado obtenido la solución propuesta en [20] muestra que el algoritmo de Dijkstra utiliza más recursos respecto al algoritmo propuesto en el estudio mencionado, reduciendo la eficiencia de la red. Sin embargo, el trabajo propuesto en el presente documento busca desarrollar una solución teórica con viabilidad para ser implementada en una red real con resultados satisfactorios; por este motivo, un incremento en los recursos utilizados puede ser un sacrificio necesario para obtener resultados prácticos utilizables para redes “en producción”. Para llegar a esta solución, el algoritmo desarrollado requiere un tiempo computacional bajo, con resultados de costo total de transporte de información sobre la red, iguales o cercanos a los valores mínimos obtenidos con optimización matemática.

1.3.10. ALGORITMOS DE ENRUTAMIENTO DE CAMINO MÁS CORTO

Una aplicación muy común de los grafos es la representación de redes para resolver problemas de tráfico o comunicación de datos. Para este tipo de problemas, resulta necesario el estudio de los caminos que interconectan los vértices; este es precisamente el objetivo de los algoritmos de camino más corto (shortest-path), los cuales determinan un buen camino, o inclusive el mejor camino, entre dos vértices de la red, dependiendo de las necesidades propias del problema. Además, debe notarse que el término "camino más corto" no se limita al significado de distancia más corta, sino que varía de acuerdo con el enfoque planteado, pudiendo ser: el camino más veloz, el de menor costo, mayor capacidad, entre otros [17].

Existen diferentes variaciones de estos algoritmos, las cuales se mencionan a continuación [21]:

- Problema del camino más corto con un único origen (*single-source shortest-paths*): este problema busca la ruta más corta entre un vértice s, el cual origina la información, hacia todos los demás vértices v, de la red. La solución de este problema puede resolver las demás variaciones del problema *shortest-path*, mostradas a continuación. Los algoritmos más conocidos para solucionarlo son: Dijkstra (el cual se tratará en la siguiente sección) y Bellman-Ford.
- Problema de camino más corto con un único destino (*single-destination shortest-paths*): busca la menor distancia hasta un único vértice t desde cada uno de los demás vértices v. Puede ser resuelto con algún algoritmo *single-source*, al invertir la dirección de cada enlace del grafo.
- Problema del camino más corto entre un par único (*single-pair shortest-path*): intenta determinar el camino más corto entre un par de nodos; es decir, se conoce cuál es el origen y el destino de la información. Puede resolverse utilizando la solución del problema *single-source* utilizando únicamente el camino deseado; esta solución será utilizada para resolver el problema, pues no se conocen algoritmos que puedan resolverlo de manera más veloz.
- Problema de camino más corto entre todos los pares (*all-pairs shortest-path*): busca determinar el camino más corto entre todos los nodos de la red. Ejecutar el algoritmo *single-source* una vez por cada enlace, puede resolver el problema más

rápidamente que un algoritmo diseñado para este caso, aunque existen ciertas implementaciones veloces; este es el caso del algoritmo de Floyd-Warshall.

Una forma muy sencilla de resolver el problema asignación de rutas en RWA, es asignar una ruta utilizando algoritmos de distancia mínima en la topología de la red, como es el caso de Dijkstra, Bellman-Ford, Floyd-Warshall, entre otros [5]. Se ha seleccionado el algoritmo de Dijkstra, debido a su sencillez y amplia difusión en gran cantidad de aplicaciones en sistemas de telecomunicaciones (forma parte de los protocolos OSPF, ISIS, entre otros).

1.3.11. ALGORITMO DE DIJKSTRA

El algoritmo propuesto en el presente documento procura utilizar el algoritmo de Dijkstra para la asignación de rutas en la resolución del problema RWA; por esta razón, se definirá en esta sección el algoritmo, haciendo énfasis en su funcionamiento y características, enfocadas en la solución de RWA.

El algoritmo de Dijkstra fue diseñado para resolver el problema del camino más corto con un solo origen (*single-source shortest-paths*) para un grafo ponderado, en el cual ningún peso correspondiente a los enlaces es negativo [21] [17]. Esta es su principal diferencia con el algoritmo de Bellman-Ford, el cual resuelve el mismo problema, para un grafo en el cual sus distancias pueden tomar valores negativos; por otro lado, el algoritmo de Bellman-Ford puede también resolver el problema de menor distancia para un grafo con pesos positivos en todos sus enlaces, pero la ejecución del algoritmo de Dijkstra utiliza un tiempo menor para obtener la respuesta; además, en una implementación real de una red, el significado del costo positivo o negativo puede ser definido según los requerimientos propios del problema; por estos motivos, el algoritmo de Dijkstra será el elegido frente a otros algoritmos.

Esta solución ha sido ampliamente utilizada en protocolos de enrutamiento para redes de información, por su capacidad de calcular la ruta para enviar paquetes a cualquier destino conocido localmente, sin la necesidad de consolidar la ejecución del algoritmo en toda la red; esto significa, que los protocolos intercambian información para conocer la topología de la red al inicio, pero la ejecución del algoritmo de camino más corto se realiza independientemente en cada dispositivo.

Una implementación general del algoritmo de Dijkstra se muestra a continuación. La explicación de este algoritmo se presenta posterior al mismo.

Procedimiento de Dijkstra $(G, w, s; d)$

(1) $d(s) \leftarrow 0, T \leftarrow V$;

(2) **for** $v \in V \backslash \{s\}$ **do** $d(v) \leftarrow \infty$ **od**;

(3) **while** $T \neq \emptyset$ **do**

(4) encuentre algún $u \in T$ tal que $d(u)$ sea mínimo;

(5) $T \leftarrow T \backslash \{u\}$;

(6) **for** $v \in T \cap A_u$ **do** $d(v) \leftarrow \min(d(v), d(u) + w_{uv})$ **od**

(7) **od**

Donde: (G, w) representa a la red, G es un grafo o un dígrafo con todas las longitudes o pesos (w) no negativas; $w(e)$ o w_{uv} corresponde al peso del enlace $e = (u, v)$, el cual representa la unión entre los nodos u y v. El vector d contiene los costos (o distancias) desde el nodo s (origen de la información) hasta los demás nodos. Además, el vector V contiene todos los nodos v que pertenecen al grafo; el vector T es temporal, en el cual se almacenan datos que cambiarán con la ejecución del algoritmo. Finalmente, la matriz A_v contiene la lista de adyacencia de todos los nodos de la red, lo cual significa, que contiene la información de todos los enlaces de la topología [15] [17].

Utilizando esta notación, se continúa con una descripción de cada línea del algoritmo:

(1) Asigna al valor 0 a la posición correspondiente a s dentro del vector d, y se copia la información del vector V dentro de la variable temporal T.

(2) Se establece para todos los demás valores del vector d el valor de ∞.

(3) Inicio del lazo **while**, que establece que se repetirá las sentencias contenidas entre (4) y (6), siempre y cuando, el vector T no se encuentre vacío.

(4) Encontrar algún valor u que pertenece al vector T, tal que $d(u)$ sea mínimo. El algoritmo considera valores que se suelen nombrar fijos (u), los cuales son considerados como referencia para el cálculo, por lo cual, después de cada iteración, estos valores fijos no serán tomados en cuenta para el cálculo del camino más corto.

(5) Se retira del vector T el nodo u, para que no se tome en cuentan en la siguiente iteración.

(6) Para todos los enlaces existentes en el nodo u y que también pertenecen al vector T, almacenar en el vector d, en la posición correspondiente al nodo v, el mínimo entre el valor actualmente almacenado en $d(v)$ y la suma $d(u) + w_{uv}$. De esta manera se garantiza que en el vector d se almacene el valor del costo menor para llegar al nodo u desde el nodo s.

(7) Final del lazo $\boldsymbol{while}$.

1.3.12. EL PROBLEMA DE RUTAS DESBALANCEADAS CON EL ALGORITMO DE DIJKSTRA

Para comprender el problema de rutas desbalanceadas al utilizar el algoritmo de Dijkstra en la resolución del problema RWA, es conveniente mencionar el caso propuesto en [17], en el cual se evalúa el desempeño de algoritmos de enrutamiento del tipo camino más corto (shortest path) para la resolución del problema. Las simulaciones desarrolladas en el trabajo mencionado determinan que es "inadecuada la utilización de los algoritmos de Dijkstra y Bellman-Ford, ambos en su forma estándar, pues producen carga dispersa, más rutas desbalanceadas sobre los enlaces de la red y sobrecarga innecesaria de los enlaces". Esta conclusión, permite notar que el principal problema de la utilización de estos algoritmos es la excesiva carga percibida en ciertos enlaces; este problema es un resultado esperado, pues los algoritmos mencionados no consideran el estado actual de la red, ni la carga actual sobre cada enlace, la cual se puede traducir en el estado actual de utilización de los recursos disponibles por la red. Sin embargo, el algoritmo propuesto en el presente trabajo trata de mitigar este inconveniente por medio de la aplicación del algoritmo de Dijkstra considerando la carga actual del enlace de la red.

Para cumplir con este objetivo, es necesario determinar un método factible que permita aplicar el algoritmo, considerando los recursos actualmente utilizados. Por consiguiente, se mencionarán dos puntos clave en la resolución del problema, que juntos permiten la adaptación del algoritmo:

- Una forma sencilla de resolver el problema de enrutamiento de *lightpaths*, es utilizar algoritmos de camino más corto para calcular la ruta de una conexión óptica a la vez, y repetir el proceso para cada nueva solicitud de comunicación [5]. De esta manera, se puede asegurar que el cálculo realizado para cada nueva solicitud que

ingresa al sistema se realice siempre utilizando información actualizada de todos los recursos disponibles y reservados en la topología actual.

- La función de costo asociada a cada enlace de la red debe ser seleccionada para considerar la carga actual sobre esa conexión. Si la función de costo incrementa conforme se incrementa el flujo del sistema, se escogerán caminos con enlaces subutilizados, y el algoritmo permitirá la disponibilidad de recursos para futuras conexiones [2].

La primera condición puede implementarse iterando el algoritmo de Dijkstra, para realizar el cálculo del camino más corto entre los dos extremos de la conexión una vez por cada *lightpath*.

La segunda condición necesita determinar una función adecuada para una red óptica; este es el caso de la función objetivo, propuesta en [2], la cual se muestra en la Figura 1.5. Esta función es adecuada para una red óptica, pues posee dos características que impactan significativamente en la naturaleza de la solución:

- Es una función convexa, monótonamente incremental y es lineal definida por partes. De esta manera, el costo de enrutar una nueva solicitud de conexión por un enlace l es mayor que el costo de enrutar los *lightpaths* anteriores en el mismo enlace, permitiendo la disponibilidad de recursos para las futuras solicitudes de conexión.
- Los puntos de corte de cada función de costo lineal ocurren en los puntos enteros $0, 1, \dots, |C|$. El costo para un flujo mayor a $|C|$ y menor a 0 es ∞, lo cual impone una restricción de la capacidad del enlace. Además, esta condición permite que el resultado del proceso de optimización sea entero, en el caso de realizar un proceso de optimización matemática, resolviendo el problema de programación lineal entera. Sin embargo, el presente trabajo trata de evitar la resolución directa de este problema, utilizando algoritmos más sencillos, lo cual puede fácilmente invitar al lector a considerar a esta función objetivo como inadecuada para este propósito; no obstante, se puede realizar ciertas modificaciones sobre la función de costo propuesta, para permitir obtener de manera más veloz los valores correspondientes a evaluar la función en los puntos de corte correspondientes. Para ello, es necesario determinar una función matemática, que debe ser sencilla de calcular, mediante la cual, se determinarán los valores correspondientes a la cantidad de flujo que atraviesa el enlace. Con este fin, se puede utilizar el proceso de interpolación o de regresión para determinar una función que permita calcular los puntos

correspondientes a la curva, incluyendo puntos más allá de los propuestos en el estudio realizado en [2]; esta característica puede resultar útil para determinar los puntos correspondientes a utilización superior del enlace, como es el caso de utilizar enlaces que soporten una mayor cantidad de longitudes de onda (como CWDM o DWDM).

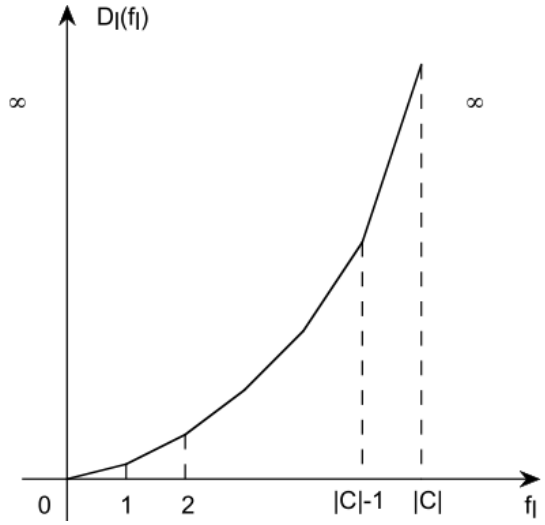

Figura 1.5. Función de costo convexa, monótonamente incremental y lineal definida por partes, para un enlace $\boldsymbol{l}$ con una capacidad $|\boldsymbol{C}|$, por la que atraviesa un flujo $\boldsymbol{f_l}$

Al considerar el tipo de tráfico que esta solución puede resolver, es fácil pensar en tráfico incremental y dinámico; sin embargo, la aplicación del algoritmo de manera iterada puede resolver el problema de tráfico estático de la misma manera, considerando un orden para los requerimientos de conexión extremo a extremo.

1.3.13. ASIGNACIÓN DE LONGITUDES DE ONDA EN RWA

La segunda parte de la resolución propuesta para el problema RWA, consiste en asignar una longitud de onda para la conexión óptica extremo a extremo. Para llevar a cabo esta tarea, deben considerarse dos restricciones [5]:

- No se debe asignar la misma longitud de onda para dos *lightpaths* en el mismo enlace. En efecto, una longitud de onda posee la capacidad de transmitir de transmitir un solo conjunto de información
- Si no existe disponibilidad de conversores de longitud de onda en un dispositivo de conmutación, la misma longitud de onda debe ser asignada en los enlaces del equipo que el *lightpath* atraviesa. En el caso de no disponer estos implementos en

toda la red, deberá asignarse la misma longitud de onda en todo el camino del *lightpath*; esta condición se conoce como restricción de continuidad.

Al considerar un sistema con tráfico incremental o dinámico, en el que las solicitudes de conexión ingresan al sistema una a la vez, se pueden utilizar métodos heurísticos para la asignación de rutas en cada enlace [18]; sin embargo, al pensar en la condición impuesta para la asignación de rutas, donde se iterará la asignación de rutas para lightpaths siguiendo algún orden para el caso de tráfico estático, es sencillo contemplar la misma solución para el segundo subproblema de RWA. De esta manera, si la red que se optimiza dispone de tráfico estático, se iterará la asignación de longitudes de onda utilizando un algoritmo basado en las propuestas mostradas en la presente sección.

Para la solución del problema de asignación de longitudes de onda debe diferenciar dos casos; el primero considera un solo hilo de fibra óptica para cada enlace de la red y el segundo considera cables con múltiples hilos de fibra en su interior; en las redes multifibra cada cable posee en su interior varios hilos de fibra; cada hilo posee la capacidad de llevar su propio conjunto de longitudes de onda WDM, funcionando independientemente entre sí.

El objetivo de la asignación de longitudes de onda en sistemas con enlaces de un solo hilo de fibra óptica es distribuir adecuadamente las longitudes de onda en cada uno de los enlaces; por otro lado, en sistemas de múltiples fibras el objetivo de la asignación de longitudes de onda es reducir el número de hilos utilizados para la comunicación [18].

Para cada caso han sido desarrollados métodos heurísticos, con el objetivo de determinar una asignación correcta de longitudes de onda; de esta manera, se muestra los métodos diseñados para redes con cables de un solo hilo de fibra óptica a continuación [18]:

- *Random* (aleatorio o R): este método realiza una asignación aleatoria de la longitud de onda asignada para una conexión, considerando el conjunto de las frecuencias disponibles.
- *First-Fit* (primera disponible o FF): en este caso, todas las longitudes de onda se numeran, y la asignación se realiza tomando la frecuencia con el menor número dentro del conjunto de lambdas no asignadas. La selección de un número para cada longitud de onda puede realizarse considerando a la frecuencia de menor longitud de onda con el menor número y a la mayor con el mayor número; sin embargo, no se restringe este método para su implementación. Por otro lado, su ejecución no requiere conocer el conjunto completo de las longitudes de onda disponibles, por lo

cual, el costo computacional de su utilización es menor que R. En la práctica, es el método preferido para implementar, pues requiere de una carga computacional reducida y su programación es sencilla, obteniendo resultados satisfactorios.

- *Least-Used* / SPREAD (menos utilizado/disperso o LU): esta solución busca la longitud de onda menos utilizada en la red y la asigna a la comunicación; de esta manera, se balancea la carga sobre todas las longitudes de onda. Para lightpaths que atraviesan una gran cantidad de nodos en la red, este método no es factible, pues existe la posibilidad de no encontrar una asignación realizable; en consecuencia, puede ser utilizado en sistemas con una cantidad reducida de nodos. Por otro lado, la implementación de este mecanismo obtiene resultados menos satisfactorios que el método R, con las desventajas de necesitar almacenamiento adicional, una mayor carga computacional y sobrecarga la red. Por estos motivos, no suele ser el método preferido en la práctica.
- *Most-Used* / PACK (más utilizado/empaquetado o MU): este método es opuesto a LU, pues asigna a las nuevas solicitudes las longitudes de onda más utilizadas. Su desempeño es muy superior a LU, y ligeramente superior a FF. Sin embargo, posee las mismas desventajas que LU.
- *Wavelength Reservation* (reservación de longitudes de onda o Rsv): en este caso, las conexiones con información del tipo *streaming* (en las cuales se produce un flujo constante de datos entre dos nodos de la red) son reservadas. De esta manera, cuando se espera una conexión entre dos nodos con tráfico de *streaming*, no se utilizará una longitud de onda asignada para este propósito, inclusive si esta frecuencia está disponible; usualmente, esta información utilizará múltiples saltos dentro de la red. Esta formulación reduce el bloqueo de conexiones de tráfico de múltiples saltos, pero aumenta el mismo parámetro para tráfico de un solo salto.
- *Protecting Threshold* (límite de protección o Thr): en este método, se asigna una longitud de onda para una conexión de un solo salto, únicamente si el número de longitudes de onda disponibles en la red supera un límite predefinido. Su objetivo, al igual que en el caso Rsv, es maximizar la disponibilidad de recursos para las conexiones de múltiples saltos, a costo de incrementar el bloqueo en conexiones de un solo salto.

Por otro lado, en redes con cables de fibra óptica de varios hilos, se utilizará los siguientes métodos [18]:

- *First-Fit multi-fiber* (primera disponible para múltiples fibras o FF-MF): este mecanismo es una implementación del método *first-fit* para redes con múltiples hilos de fibra óptica. En este método, las fibras y las longitudes de onda son numeradas y se escogerá el primer hilo con al menos una longitud de onda disponible.
- *Min-Product* (producto mínimo o MP): en este caso, el algoritmo determina el número de enlaces que utilizan una misma longitud de onda, y procura minimizar la función producto de este número en cada enlace. Su desempeño no iguala al de FF-MF e introduce una mayor carga computacional.
- *Least-Loaded* (menos cargada o LL): selecciona la longitud de onda con la mayor capacidad residual en el enlace más cargado a lo largo de la ruta. Su desempeño es superior a MU y FF-MF.
- MAX-SUM (suma máxima o $M\Sigma$): puede ser aplicado también al caso de cables con un solo hilo de fibra. Este método, asume que se conoce con anterioridad los flujos que se necesitan establecer en la red, por lo cual, considera tráfico estático o al menos una matriz de flujos estable. Su objetivo, es maximizar la capacidad de la red, después del establecimiento de los *lightpaths*, utilizando la ruta asignada por la solución del primer sub problema de RWA.
- *Relative Capacity Loss* (pérdida relativa de capacidad o RCL): este método, fue diseñado en base a $M\Sigma$ y busca minimizar la pérdida total de capacidad en una longitud de onda en específico.

El presente trabajo, procura determinar la efectividad del algoritmo desarrollado para la optimización de redes de transporte de manera directa. En consecuencia, el método seleccionado para la asignación de longitudes de onda debe ser sencillo de implementar, y debe obtener un resultado satisfactorio. Por este motivo, se selecciona el algoritmo *First-Fit* para la solución propuesta.

2. METODOLOGÍA

En esta sección se presentará el método de desarrollo el proyecto. Según lo definido en la sección 1, el objetivo del presente es el desarrollo de un algoritmo para encontrar una solución directa al problema RWA; con este propósito en mente, es lógico pensar en la implementación del algoritmo utilizando el desarrollo de un programa computacional utilizando los algoritmos y métodos definidos en la sección previa.

Para el desarrollo del algoritmo, es necesario definir la función de costo, de acuerdo con la formulación realizada en la Figura 1.1. En este caso, se utilizará el ejemplo mostrado en la Figura 2.1 [22]. En este ejemplo se muestra una función lineal definida por partes con los siguientes valores:

$$\begin{aligned} &\boldsymbol{D_l(f_l)} \\ &\boldsymbol{D_l(0) = 0} \\ &\boldsymbol{D_l(1) = 1} \\ &\boldsymbol{D_l(2) = 9} \\ &\boldsymbol{D_l(3) = 25} \end{aligned} \tag{2.1}$$

Con estos valores, es claro notar que pueden ajustarse a la siguiente función:

$$\boldsymbol{D_l(f_l)} = \begin{cases} \boldsymbol{(2 * f_l - 1)^2;\ \forall f_l > 0} \\ \boldsymbol{0;\ f_l = 0} \end{cases} \tag{2.2}$$

Con esta definición es sencillo obtener valores para un número de longitudes de onda mayor. La función de costo definida en la Ecuación 2.2. se implementará en el algoritmo desarrollado en el presente proyecto.

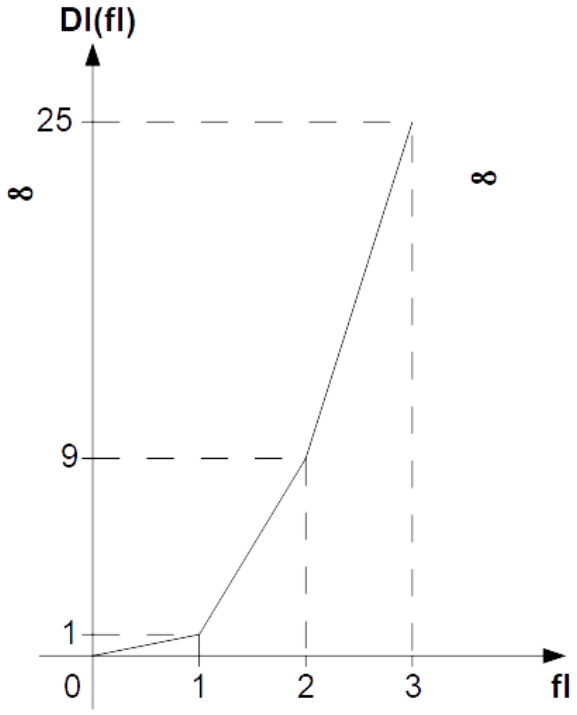

Figura 2.1. Función de costo, lineal definida por partes, para una red con capacidad máxima 3 **[22]**

Para el desarrollo del algoritmo, es necesario definir un lenguaje de programación adecuado de acuerdo con los requerimientos del problema planteado. En este contexto, se requiere un lenguaje de programación con la capacidad de realizar operaciones con matrices, pues la teoría de grafos y el problema de *Routing and Wavelength Assignment* se utilizan matrices para la definición del problema, su procesamiento y solución. Existe una variedad de lenguajes de programación que cumplen con esta característica, los cuales incluyen: Matlab (*Matrix Laboratory*) [23], R [24] y Python (con su paquete de software NumPy) [25]. Por supuesto, estos lenguajes son solo una muy pequeña parte de todas las opciones disponibles en el mercado que cumplen con las características que requiere el desarrollo del presente proyecto; sin embargo, se ha escogido estos lenguajes a manera de ejemplo, pues son los más representativos y los más conocidos para facilitar el desarrollo del algoritmo a manera de prueba de concepto (PoC), con lenguajes de programación ampliamente conocidos y muy sencillos de aplicar. De esta manera, se enfocará la solución del problema en el desarrollo e implementación del algoritmo propuesto, y no en la selección del mejor lenguaje de programación para su aplicación en entornos prácticos.

Teniendo en mente esta introducción a las opciones disponibles, se escoge el lenguaje de programación de Python con su paquete NumPy para el desarrollo del software del presente proyecto. Python fue creado en el año 1991 como un lenguaje de propósito general, interpretado y orientado a objetos [25]. Es un lenguaje de alto nivel con una estructura simple [26], que se ha crecido en un ecosistema maduro con el desarrollo de paquetes especializados [25] para una extensa variedad de propósitos [25]. Estas características hacen del lenguaje Python muy atractivo para el desarrollo de proyectos cuya finalidad es la ejecución de pruebas con algoritmos nuevos, permitiendo la implementación de código de manera rápida y la ejecución de pruebas de manera simple. Por defecto, el lenguaje de Python no incluye variables vectoriales o matriciales, considerando que las listas de Python no son consideradas dentro de esta clasificación pues su propósito y las operaciones que se pueden realizar sobre este tipo de información no corresponde completamente con vectores o matrices. Por este motivo, se desarrolló el paquete NumPy, el cual permite al usuario la posibilidad de crear variables a manera de matrices multidimensionales (conocidos como arreglos o *arrays* en inglés) y lo provee de un gran conjunto de funciones que permiten realizar operaciones sobre estos arreglos [25]. Los arreglos creados por este módulo permiten el almacenamiento de datos en forma matricial y la ejecución de operaciones sobre este tipo de variables de manera rápida [25]. Es necesario mencionar que el módulo NumPy no será el único utilizado en el desarrollo del presente proyecto, pero si será el más importante; adicionalmente, se mencionarán brevemente los otros módulos utilizados en el código implementado. El módulo "*os*" permite al usuario enviar comandos al sistema operativo sobre el cual se corre el software de Python, para ejecutar tareas que corresponden al sistema operativo; este módulo se usa para enviar un comando de borrado de consola hacia el sistema operativo, para presentar los datos y resultados de manera ordenada. Adicionalmente, se utilizará el módulo "*time*" para realizar el cálculo del tiempo de ejecución del algoritmo en el sistema. Finalmente, en las secciones 2.3 y 2.4 se implementarán módulos de programación desarrollados para el presente proyecto.

Ahora que se han determinado las herramientas que se utilizarán en el desarrollo del presente proyecto, es necesario determinar el proceso de desarrollo de la implementación. Según se mencionó en la sección 1, el problema de enrutamiento y asignación de longitudes de onda es comúnmente dividido en 2 etapas: enrutamiento de longitudes de onda y asignación de longitudes de onda. De esta manera, el algoritmo desarrollado deberá cumplir con la solución de estos dos problemas. A continuación, se presenta el desarrollo

de todos los métodos utilizados en cada función y en el método principal para el desarrollo del proyecto.

2.1. VARIABLES

Es necesario definir las variables que se utilizarán para el cálculo de la solución buscada. En el presente programa se requieren tres variables: matriz de topología, matriz de flujos y capacidad. Estas variables se describen a continuación.

- Matriz de topología: matriz de topología/costo. Contiene todos los costos entre los enlaces de la red. Si el costo es infinito, no existe conexión directa entre los dos enlaces. Debe entenderse las filas i y las columnas j como el costo de enviar información por el enlace $n[i,j]$. Cualquier costo diferente a infinito (Inf) indica una conexión existente entre el nodo origen i y el nodo destino j. El valor infinito (Inf) indica una conexión inexistente entre el nodo i y el nodo j. La diagonal de la matriz siempre tiene el valor cero, pues corresponde al costo de enviar información desde un nodo como origen al mismo nodo como destino. A continuación, se muestra la Ecuación 2.3. con un ejemplo de esta matriz.

$$\boldsymbol{\Omega} = \begin{bmatrix} \mathbf{0} & \mathbf{10} & \infty & \mathbf{3} \\ \mathbf{5} & \mathbf{0} & \mathbf{2} & \infty \\ \infty & \mathbf{1} & \mathbf{0} & \mathbf{7} \\ \mathbf{0} & \infty & \mathbf{0} & \mathbf{0} \end{bmatrix} \tag{2.3}$$

En la Ecuación 2.3. se muestra la representación de la red de topología anillo mostrada en la Figura 2.2. Considerando la numeración de filas y columnas a partir del cero (como se define en NumPy), se encuentran 4 nodos de la topología, interconectados como se define en la Figura 2.2. las filas representan el origen del enlace y las columnas representan el destino del enlace que interconecta dos nodos. Adicionalmente, el valor asignado en cada valor de la matriz corresponde al costo actual de transportar información por este enlace. De esta manera, el valor de 10 contenido en la fila 0, columna 1, indica que el costo de transportar información desde el nodo 0 al nodo 1 es de 10. Por otro lado, la fila 3, columna 1 posee un valor de infinito; esto determina que no existe un enlace con origen el nodo 3 y destino el nodo 1, como puede apreciarse en el ejemplo.

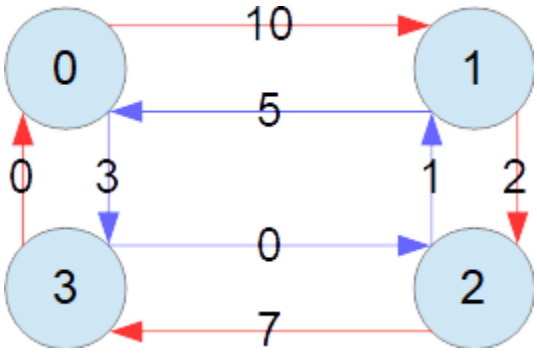

Figura 2.2. Topología en anillo ejemplo

- Matriz de flujos: contiene los flujos que se requieren instalar en el sistema. Debe entenderse por flujos a la cantidad de longitudes de onda (lambdas origen destino) que requieren ser instalados entre el nodo origen y el nodo destino. El valor de cero indica que no se requiere instalar longitudes de onda entre los nodos origen destino. La fila de la matriz indica el nodo origen de la comunicación requerida y la columna representa el nodo destino. La diagonal de la matriz es siempre cero, pues son valores de *lightpaths* que requieren ser instalados entre un nodo como origen y el mismo nodo como destino. La Ecuación 2.4. muestra un ejemplo de una matriz de flujo; se puede notar en la fila 0, columna 2 un valor de 2. Este valor indica el requerimiento de comunicar el nodo 0 con el nodo 2 utilizando un total de 2 *lightpath* para esta comunicación.

$$\mathbf{flujo} = \begin{bmatrix} \mathbf{0} & \mathbf{0} & \mathbf{2} & \mathbf{1} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{3} \\ \mathbf{0} & \mathbf{7} & \mathbf{0} & \mathbf{0} \end{bmatrix} \tag{2.4}$$

- Capacidad: esta variable corresponde a un número real entero positivo, que limita el número máximo de *lightpaths* que pueden atravesar cualquier enlace de la topología. Cuando el número de lightpaths que utilizan un enlace del sistema se iguala con el valor de la capacidad, se determina que el enlace ha llegado a su capacidad máxima y no puede admitir nuevas conexiones.

2.2. DESARROLLO DEL PROGRAMA PRINCIPAL

La tabla contenida en el Código 2.1, muestra la programación implementada en el programa principal. Su código invoca a las funciones desarrolladas para solucionar el problema planteado en el presente proyecto y muestra los resultados obtenidos; la

explicación de este código se desarrolla después de la tabla, y muestra de manera global el funcionamiento del algoritmo y el uso de cada una de las funciones implementadas en los módulos respectivos.

Código 2.1. Código del programa principal

```
(1) # coding=utf-8
(2) import numpy as np
(3) import DijkstraFirstFit as dj
(4) import os
(5) import time
(6) from variables import *
(7) os.system("cls")
(8) start_time = time.clock()
(9) flujoTemp = flujo.copy()
(10) omega1 = omega.copy()
(11) omega2 = omega.copy()
(12) m, n = omega.shape
(13) lambdasAssigned = np.empty((n, n), dtype = object )
(14) paths = np.zeros(n)
(15) x = np.zeros((n, n),dtype = float)
(16) pathSolution = False
(17) for i in range(0,m):
(18)     for j in range(0,n):
(19)         while (flujoTemp[i][j] != 0):
(20)             pathSolution = False
(21)             while pathSolution == False:
(22)                 d, path = dj.dijkstra(omega1, i)
(23)                 if np.all( d == Inf):
(24)                     print '\033[91m' + 'No existe una solucion factible' + '\033[0m'
(25)                     exit()
```

```
(26)              pathD = dj.dijkstraPath(path, j)
(27)          lambdasAssignedTemp, lambdasAssignedPathD = dj.ffLambdasAssigned(
pathD, capacidad, lambdasAssigned )
(28)              lambdasAssigned = lambdasAssignedTemp.copy()
(29)              x = dj.flujoIncremental(x, pathD)
(30)              omega1 = dj.costoIncremental(omega1, x)
(31)              omega2 = dj.costoIncremental(omega2, x)
(32)              if (np.any (x == capacidad)):
(33)                  omega1[x == capacidad] = Inf
(34)              flujoTemp[i][j] = flujoTemp[i][j]-1
(35)              if (np.any(paths != 0)):
(36)                  paths = np.vstack([paths, pathD])
(37)                          lambdasAssignedVector = np.vstack([lambdasAssignedVector,
lambdasAssignedPathD])
(38)              else:
(39)                  paths = pathD
(40)                  lambdasAssignedVector = [lambdasAssignedPathD]
(41)              pathSolution = True
(42) stop_time = time.clock()
(43) print '\n======================DATOS==========================='
(44) print "\nMatriz Omega = "
(45) print omega
(46) print "\nMatriz Flujo = "
(47) print flujo
(48) print "\nLa capacidad de los enlaces es: " + str(capacidad)
(49) print "\n\n"
(50)                                                                    print
'======================RESULTADOS==========================='
(51) print "\nMatriz x = "
(52) print x
```

```
(53) print '\nMatriz paths= '
(54) print paths
(55) print "\nlambdasAssignedVector"
(56) print lambdasAssignedVector
(57) print '\nMatriz omega1= '
(58) print omega1
(59) print '\nMatriz omega2= '
(60) print omega2
(61) print "\nMatriz de flujos resultantes = "
(62) print flujoTemp
(63) print 'Matriz lambdasAssigned'
(64) print lambdasAssigned
(65) costoT = dj.dijsktraCostF(omega2)
(66) print 'costoT = ',costoT
(67) print("--- %s seconds ---" % (stop_time - start_time))
```

Para entender el código fuente desarrollado, es necesario mencionar muy brevemente el formato de sangría definido por Python; este formato es equivalente al uso del carácter de abrir llave “{“ en otros lenguajes de programación como C, para delimitar el inicio de una función, lazo o condición y del carácter cerrar llave “}” para marcar el fin de estas secciones. En Python, se añade un espaciado en blanco al inicio de una línea para el cálculo del nivel de la sangría, utilizado para determinar la agrupación de las sentencias [27]; de esta manera, el iniciar una línea con un espacio en blanco, permite al sistema computacional calcular el nivel de identación de la línea. Este espacio en blanco puede ser definido usando espacios o tabulaciones al inicio de una línea.

Con la definición anterior en mente, se inicia el análisis de cada línea del programa. En el caso de funciones desarrolladas para el presente proyecto e importadas de librerías propias, serán definidas a partir de la siguiente sección.

La línea (1) se añade para permitir utilizar caracteres del idioma español en el código y en los comentarios del programa. Es necesario definirla al inicio del programa, con el símbolo # usado normalmente para añadir comentarios.

El código escrito desde la línea (2) hasta la línea (6) permite importar los módulos requeridos en el desarrollo del programa.

El inicio de un programa en Python donde se requiere importar librerías externas necesita definir la sentencia "*import"*, la cual incluye el nombre de la librería que se requiere importar; en el caso de la línea (2), se importa la librería NumPy que permitirá utilizar matrices y realizar operaciones matriciales entre las mismas. Adicionalmente, Python permite importar la librería con un alias para facilitar el desarrollo del código; con esta finalidad, se utilizó la opción "*as*" seguido con el alias utilizado para definir la librería. Se utiliza el alias "np", utilizado comúnmente para esta librería.

De la misma manera, se utiliza el comando "*import"* para importar librerías de la línea (3). En este caso, la librería importada no es propia de Python, y corresponde a la librería creada para el presente proyecto. Esta librería dispone de las funciones desarrolladas para el presente proyecto. Al igual que en el caso anterior, se define un alias para esta librería, utilizando las letras dj.

En la línea (4) se importa la librería "*os*", que permite correr comandos del sistema operativo desde Python. Se utilizará esta librería únicamente para limpiar la consola de comandos de Windows (CMD) antes de iniciar el cálculo actual.

La línea (5) permite importar la librería "*time*"; esta librería será utilizada para calcular el tiempo real utilizado por el algoritmo para finalizar el cálculo.

El código desarrollado en la línea (6) se utiliza para importar la librería de variables del programa. En este caso, se utiliza la sentencia "*from import"*. Esta sentencia permite importar todos los datos contenidos en esta librería sin la necesidad de usar el prefijo de librería para acceder a esta información. En este caso, se define de esta manera para importar las variables contenidas en este archivo; adicionalmente, es necesario mencionar que se definen las variables en un archivo diferente, pues para las pruebas requeridas se modificarán los datos de las variables para comparar los resultados.

Esta línea inicia con el programa. Utiliza la librería *"os"* y llama a la función "*system"*. Esta función permite enviar comandos al sistema operativo; los comandos se definen dentro de los paréntesis de la función y deben definirse dentro de comillas. El comando enviado corresponde a "cls" (*clear screen*); este comando se utiliza en el CMD de Windows para enviar a limpiar los datos mostrados en pantalla.

La línea (8) utiliza la librería "*time*" e invoca a la función "*clock()"*. Esta función permite obtener la hora actual del sistema. Esta información se almacena en la variable "*start_time*", y se utilizará al final del cálculo computacional para calcular el tiempo total que utilizó el algoritmo para finalizar el cómputo.

La línea (9) utiliza la variable flujo importada desde el archivo variables.py. Se utiliza la función "*copy()*", propia de la variable flujo del tipo matriz de NumPy; esta función permite copiar la matriz contenida en la variable flujo hacia la variable "flujoTemp". Esta copia permite operar sobre la variable "flujoTemp", sin perder la información contenida en la variable original "flujo". Se utilizará esta copia para disponer de la matriz original al final del programa, al imprimir los resultados operando la matriz "flujoTemp" en el programa.

De la misma manera, en las líneas (10) y (11) se utiliza la función *"copy()"* de la matriz "omega" para realizar dos copias de la matriz "omega", sobre las variables "omega1" y "omega2"; estas copias permitirán disponer de la información original de la matriz "omega" para imprimir al final del programa y también permitirán realizar operar y realizar recálculos en el programa principal.

En la línea (12), la función "*shape"* del objeto matricial de NumPy permite obtener las dimensiones de la matriz, como número de filas "m" y número de columnas "n". La matriz "omega" es siempre una matriz cuadrada; no obstante, debido al resultado entregado por la función, se requieren 2 variables de salida.

El código de la línea (13) permite crear la matriz cuadrada vacía "*lambdasAssigned*", de dimensiones nxn. Esta matriz se crea para almacenar los valores de las longitudes de onda asignados durante el cálculo del programa. Permite al programa conocer las longitudes de onda previamente asignadas, evitando reutilizarlas y se utilizará para imprimir los resultados al final del procesamiento.

En la fila (14) se crea la variable "*paths*", que almacena una matriz cuadrada de tamaño nxn conteniendo el número cero en todos sus valores. Esta variable almacenará todos los caminos escogidos por el programa para todos lightpaths instalados.

El código de la línea (15) permite Inicializar la variable "x" conteniendo una matriz de tamaño nxn, con todos sus valores en cero; esta matriz contiene la cantidad de flujos instalados en el cálculo actual para cada enlace. En consecuencia, si la fila 0, columna 3 de la matriz "x" tiene un valor de 2, significa que por el enlace con origen el nodo 0 y destino

el nodo 3 se han asignado un total de 2 longitudes de onda. El valor contenido en esta matriz no puede ser superior a la capacidad del enlace.

La variable "*pathSolution"* se define en la línea (16); es del tipo lógica o booleana, y permite realizar un lazo infinito que termina cuando se encuentra la solución o cuando se determina que no existe solución factible para el problema planteado. Se inicializa con el valor *False* o Falso, y su valor cambiará a *True* o verdadero si el programa finaliza el cálculo.

El lazo principal se define en las líneas (17), (18) y (19). La fila (17) permite recorrer todas las filas "i" de la matriz "Flujo", para realizar el cálculo de todos los *lightpaths* requeridos para el problema analizado. Inicia en cero, pues NumPy enumera desde este valor a la primera fila. El incremento no se define en la sentencia, indicando un aumento por defecto de 1 en cada iteración. El final del lazo es en el valor de m definido previamente como el tamaño de la matriz cuadrada de topología.

El código de la fila (18) permite recorrer todas las columnas "j" de la matriz "Flujo", para realizar el cálculo de todos los *lightpaths* requeridos para el problema analizado. Inicia en cero, pues NumPy enumera desde este valor a la primera columna. El incremento no se define en la sentencia, indicando un aumento por defecto de 1 en cada iteración. El final del lazo es en el valor de n definido previamente como el tamaño de la matriz cuadrada de topología.

En la fila (19) se crea el lazo de comprobación de final de cómputo. Permite determinar si la matriz "*flujoTemp*" dispone de *lightpaths* pendientes para calcular. En caso de encontrar algún valor diferente de cero en uno de sus valores, entonces determina que es necesario realizar un nuevo cálculo. En cada iteración el valor de la matriz "flujoTemp" reduce en 1 en la posición calculada.

En la fila (20) se define la variable "*pathSolution"* en el valor "*False*". Esta variable permite realizar un lazo infinito hasta terminar el cálculo dentro del lazo "*while"* de la siguiente línea. Cuando termina el cálculo respectivo, el programa sale del lazo de la línea (21) y vuelve al valor *False* para reiniciar el cálculo.

En el código definido en la línea (21), se inicia el lazo de cálculo de la solución para el cálculo del *lightpath* actual. Permite computar el camino más corto y la longitud de onda para el *lightpath* analizado en esta iteración.

La programación realizada entre las líneas (22) y (26) realiza el cálculo de enrutamiento de longitudes de onda, resolviendo el primer cálculo requerido para resolver el problema RWA.

La línea (22) utiliza la función “dijkstra” que se encuentra dentro del archivo de funciones importadas como “dj”. Esta función realiza el cálculo de camino más corto con origen “i” hacia todos los nodos de la topología, de acuerdo con la definición del algoritmo de Dijkstra definida previamente. El detalle de la programación de este algoritmo se describe en la siguiente sección, donde se detalla este archivo y el código desarrollado. La matriz “omega1” se ingresa en esta función como la matriz de topología. Los resultados obtenidos con el cálculo realizado son el vector “d” que contiene los costos acumulados de llegar al nodo correspondiente a la columna del vector con el origen *“i”* definido como variable de entrada. Adicionalmente, se obtiene como resultado el vector “path” que contiene los valores de los caminos escogidos para llegar al nodo “i” y todos sus posibles destinos.

La línea (23) contiene una sentencia de evaluación condicional “if”. Permite determinar si todos los valores contenidos en el vector “d” contienen el valor “Inf” (infinito). En caso de encontrar esta condición como verdadera, se determina que no existe una solución factible para el problema planteado, con las condiciones definidas para el mismo, de acuerdo con la definición del algoritmo de Dijkstra. Esta condición puede obtenerse en variados escenarios, pudiendo describir los más generales como: la capacidad del sistema no permite establecer todos los lightpaths requeridos, las condiciones iniciales del sistema no permiten establecer todas las longitudes de onda requeridas, entre otros. La evaluación de esta sentencia como verdadera, inicia las siguientes dos líneas de programa, las cuales imprimen en pantalla el texto “No existe una solución factible” y terminando el programa.

En la fila (24) se imprime en pantalla el texto “No existe una solución factible”. Los caracteres definidos al inicio y al final del texto permiten imprimir la frase en color rojo en el CMD de Windows.

La función “*exit*()” utilizada en la línea (25) permite terminar el programa y salir del mismo.

La función “dijkstraPath” contenida en el módulo “dj” se utiliza en la línea (26); esta función realiza el cómputo del camino seleccionado por el algoritmo para llegar al destino “j”. Para este cálculo se requiere ingresar a la función el destino “j” y la variable “*path”* calculada con el algoritmo de Dijkstra, la cual contiene los caminos escogidos para el cálculo actual. El resultado se almacena en el vector “*pathD*”, el cual contiene todos los saltos necesarios para alcanzar el destino “j” con origen “i”.

Las líneas (27) y (28) realizan el cálculo de la solución del problema *first fit*.

En la línea (27) se llama la función "*ffLambdasAssigned"* del código desarrollado; esta función realiza la asignación de una longitud de onda para cada uno de los enlaces involucrados en la comunicación. Recibe como variables el vector "*pathD*", el valor de la capacidad del enlace y la matriz "*lambdasAssigned*". Los resultados entregados por esta función corresponden a la matriz "*lambdasAssignedTemp"* que incluye las longitudes de onda asignadas para todos los enlaces y el vector "*lambdasAssignedPathD"*, el cual contiene las longitudes de onda asignadas para la ruta calculada en la iteración actual.

La línea (28) permite copiar el resultado obtenido en la variable "*lambdasAssignedTemp"* y asignarla a la variable "*lambdasAssigned"*.

El código de la línea (29) utiliza la función "flujoIncremental", diseñada para incrementar el contador de número de longitudes de onda que se encuentran actualmente asignadas en cada enlace. Recibe como argumentos la matriz "x" con los valores actuales y el vector "pathD", que contiene la ruta asignada para el presente enlace. El resultado se imprime en la variable "x" sobrescribiendo su valor actual.

En las líneas (30) y (31) se añade el código para incrementar el valor del costo contenido en las matrices "omega1" y "omega2". El nuevo valor de costo contenido en los enlaces calculados por el algoritmo de Dijkstra se calcula en base a la ecuación definida al inicio de la sección 2, $D_l(f_l) = (2 * f_l - 1)^2$. Se realiza este cálculo sobre las dos matrices, pues las dos deben contener el mismo valor de costo sobre cada enlace.

Las líneas (32) y (33) permiten realizar un recálculo de los valores contenidos en la matriz "omega1". En caso de encontrar cualquier valor en la matriz "x" en la línea (32), conteniendo un número de longitudes de onda que atraviesen un enlace igual a la capacidad máxima de los enlaces. En este caso, se utiliza la línea (33) para realizar el cambio del valor del costo contenido en la matriz "omega1" para el enlace respectivo, por el valor de infinito (Inf). La matriz "omega1" se utiliza para realizar el cálculo de rutas en el algoritmo de Dijkstra. Este cambio no se aplica sobre la matriz "omega2", pues esta no se utiliza para el cálculo; sin embargo, se utiliza para imprimir los resultados al final del algoritmo.

La línea (34) decrementa en 1 el valor del número de longitudes de onda que se necesitan instalar en la posición de la fila "i" y la columna "j". Este cambio permite ingresar a la siguiente iteración el nuevo número de longitudes de onda pendientes para instalar.

Las líneas (35) hasta (40) se utilizan para almacenar las rutas y las longitudes de onda seleccionadas en cada iteración. Se almacena los vectores calculados en cada iteración y

se realiza una operación de apilar (*stack*) sobre los vectores "*paths"* y "*lambdasAssignedVector"*. La variable "*paths*" contiene el cálculo de cada salto realizado con el algoritmo de Dijkstra para resolver el problema de enrutamiento (*routing*). Adicionalmente, la variable "*lambdasAssignedVector"* contiene todos los vectores asignados para cada una de las rutas seleccionadas. Esta información se almacena para presentar estos resultados al final del programa.

La sentencia de decisión "*if*" contenida en la línea (35) y su respectiva sentencia "*else*" escrita en la línea (38), permiten diferenciar entre un escenario donde las variables "*paths"* y "*lambdasAssignedVector"* han sido creadas previamente, de un escenario donde no se han creado. En caso de haber sido creadas previamente, se procede a ejecutar las líneas (36) y (37); caso contrario, se procede con las líneas (39) y (40). Las líneas (39) y (40) se añaden al programa para inicializar las variables "*paths*" y "*lambdasAssignedVector"* en la primera iteración. Para las siguientes iteraciones, el selector "*if"* escogerá las líneas (36) y (37) para realizar un proceso de apilar los nuevos vectores al final de la matriz; con este objetivo, se utiliza la función "*vstack*" del módulo NumPy.

La línea (41) se añade para cerrar el lazo definido en la línea (21), y regresar a los lazos anteriores, continuando con el cálculo. Al terminar todas las iteraciones requeridas por el programa, recorriendo todos los *lightpaths* necesarios para el cálculo, el programa sale del lazo principal y regresa a la primera jerarquía en la línea (42).

El código definido desde la línea (42) hasta la línea (67) permiten presentar los datos ingresados al programa y los resultados obtenidos durante el cálculo. En caso de llegar a ejecutar estas líneas de programación, se determina que el programa tiene una solución factible y se presentan los resultados. Un ejemplo del resultado obtenido con este código se presenta en la Figura 2.3.

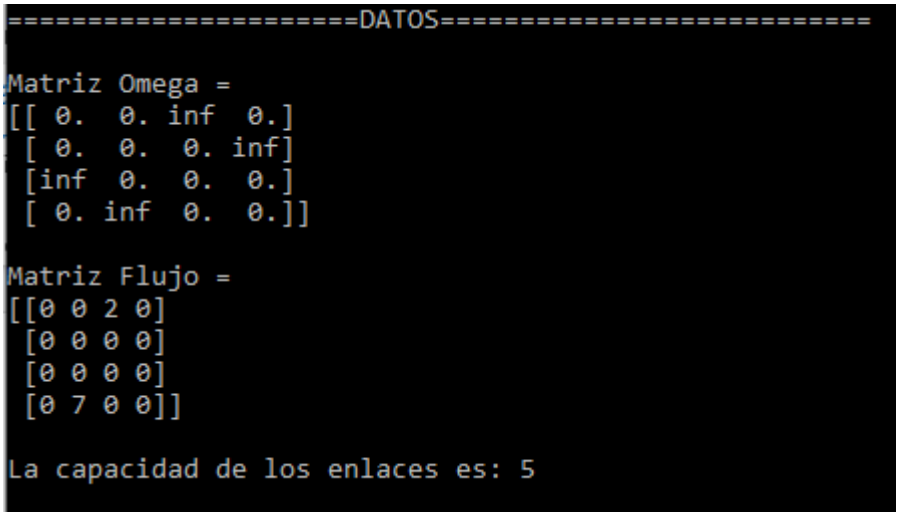

Figura 2.3. Resultados obtenidos - Datos

La línea (42) es la única que no imprime en pantalla los resultados encontrados. No obstante, se utiliza como dato de entrada para la línea (67), donde se imprime el tiempo que le tomó al programa encontrar una solución. En este caso, se repite el código de la línea (8), almacenando en la variable "*stop_time"* la hora actual del sistema.

Las filas encontradas desde el número (43) hasta (48) permiten imprimir en pantalla los datos ingresados para el cálculo realizado; estos datos incluyen la matriz "omega" en las líneas (44) y (45), la matriz de flujo en las líneas (46) y (47) y la capacidad máxima de los enlaces en la línea (48).

En la Figura 2.3 se muestra un ejemplo de los datos ingresados al programa. Puede notarse la matriz "omega" ingresada, con un tamaño 4x4 (es una matriz cuadrada según lo definido en la sección anterior). El valor de infinito contenido en la fila 0, columna 2, indica que no existe un enlace con su origen en el nodo 0 y destino el nodo 2. Este valor se repite para los enlaces 1-3, 2-0 y 3-1. Esta matriz se abstrae de la topología mostrada en la Figura 2.4. Se evidencia que esta red representa una topología en anillo, sin ninguna longitud de onda atravesando sus enlaces.

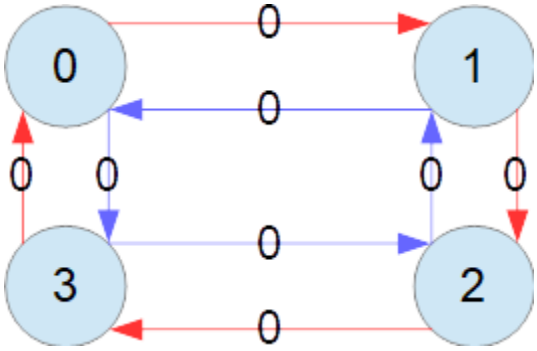

Figura 2.4. Resultados obtenidos – Datos – Topología de la matriz omega

La programación comprendida en las líneas (46) y (47) del Código 2.1, permite imprimir la matriz de "flujo" ingresada como dato al sistema. El resultado obtenido en un ejemplo realizado puede apreciarse en la Figura 2.3. en la fila 0, columna 2 se ingresa un valor de 2; este valor indica que se requieren instalar un total de 2 longitudes de onda que ingresen por el nodo 0 y con un destino el nodo 2. De manera análoga, se requieren un total de 7 longitudes de onda con origen el nodo 3 y destino el nodo 1.

La línea (48) permite imprimir en pantalla la capacidad máxima definida para los enlaces. En la Figura 2.3 se observa un ejemplo con una capacidad de 5; este valor referencia a un máximo de 5 longitudes de onda que pueden atravesar cualquier enlace en la red.

En el Código 2.1, se añade la línea (49) para agregar dos espacios al final de la impresión de los datos; este código permite separar de manera ordenada los resultados y los datos presentados en pantalla.

Las líneas del Código 2.1 comprendidas entre los números (50) y (64) permiten imprimir en pantalla los resultados obtenidos durante el cálculo del programa.

La matriz x se imprime en las filas (51) y (52). Esta matriz contiene el número de longitudes de onda que atraviesan cada enlace de la topología ingresada. En la Figura 2.5 se muestra un ejemplo del cálculo realizado utilizando los datos mostrados en la Figura 2.3; la matriz x obtenida contiene el número de longitudes de onda que recorren un enlace, de acuerdo con el cálculo realizado. Analizando la fila 0, columna 1, se determina que el enlace entre el nodo 0 con destino el nodo 1 tiene 5 longitudes de onda atravesando esa conexión. De la misma manera, en la fila 2, columna 1 se encuentra un valor de 3, correspondiente a un total de 3 longitudes de onda atravesando el enlace con origen el nodo 2 y destino el nodo 1.

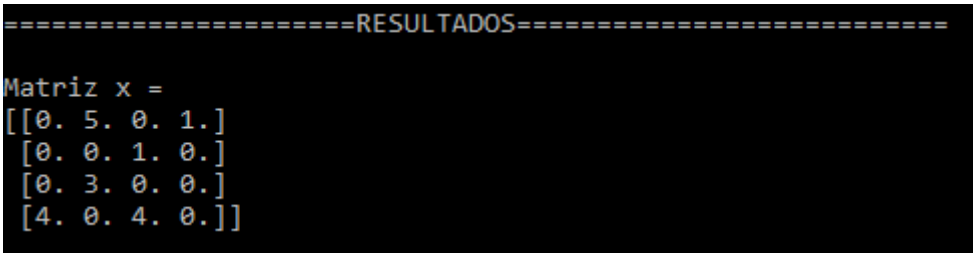

Figura 2.5. Resultados obtenidos – Matriz x

Las líneas (53) y (54) mostradas en el Código 2.1 se utilizan para imprimir los resultados contenidos en la matriz "*paths*". Esta matriz contiene las rutas seleccionadas por el algoritmo para cada uno de los "*lightpaths"* requeridos en la matriz "flujo". La Figura 2.6 muestra la matriz "paths" obtenida utilizando los datos mostrados en Figura 2.3. Se nota claramente la presencia de los números negativos -2 y -1 en todas las filas. Estos valores se utilizan en esta matriz como delimitadores, para la programación realizada; encontrar estos valores en esta matriz indica que el nodo anterior es el nodo destino de la comunicación establecida. De esta manera, se puede verificar la fila 0 que contiene los valores 0, 1 y 2. Estos son los resultados obtenidos al calcular la ruta para el primer flujo definido en la matriz de "flujo", con origen el nodo 1 y destino el nodo 2. En la topología definida en la Figura 2.4, puede observarse que existen 2 caminos posibles para la comunicación entre estos dos nodos; la primera utiliza el nodo 1 como nodo de paso y la segunda utiliza el nodo 3 con el mismo fin. En este caso, se selecciona el nodo 1. Le igual manera, se puede apreciar la fila 4 de la Figura 2.6; en esta ocasión se observa una ruta que atraviesa los nodos 3, 2 y 1. Esta información indica que el nodo 3 es el origen de la comunicación, el nodo 1 es el destino y se escogió el nodo 2 como nodo intermedio.

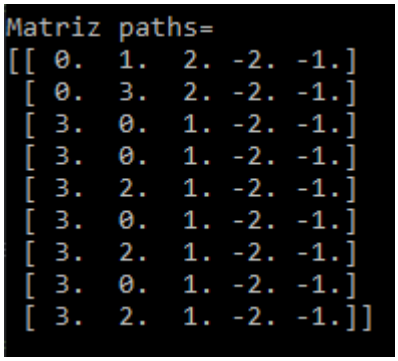

Figura 2.6. Resultados obtenidos – Matriz paths

Las líneas (55) y (56) mostrados en el Código 2.1 permiten imprimir los resultados de la matriz "*lambdasAssignedVector*". Esta matriz contiene las longitudes de onda utilizadas para cada comunicación. Los resultados de esta matriz deben contrastarse con la

información contenida en la matriz "*paths"*; el número de fila de esta matriz corresponde con el número de fila de la matriz "*paths*", y pertenece a la comunicación definida en el mismo número de fila de esta última matriz. La matriz mostrada en el ejemplo de la Figura 2.7 permite ilustrar estos resultados. Se nota nuevamente la presencia del número -1, utilizado como carácter delimitador, de la igual manera que en la matriz "*paths*"; adicionalmente, se nota la presencia de los caracteres de guion bajo al inicio y al final de cada longitud de onda. Este carácter permite delimitar una longitud de onda almacenada en la matriz "*lambdasAssigned"*, mostrada posteriormente en la presente sección; finalmente, se nota la presencia de caracteres de comilla simple (') al inicio y al final de cada posición de la matriz. Este carácter aparece debido al tipo de dato contenido en la matriz, pues se requiere utilizar el tipo de dato *string* para agregar el carácter guion bajo en la información contenida en la matriz.

La fila 0 de la matriz "*paths"* mostrada en la Figura 2.6 define una ruta que atraviesa los nodos 0, 1 y 2; en correspondencia, la fila 0 de la matriz "*lambdasAssignedVector"* define las longitudes de onda utilizadas para esa comunicación. Para el enlace existente con origen el nodo 0 y destino el nodo 1, se selecciona la longitud de onda 1; de igual manera, se selecciona la longitud de onda 1 para este lightpath, cuando atraviesa los nodos 1 y 2.

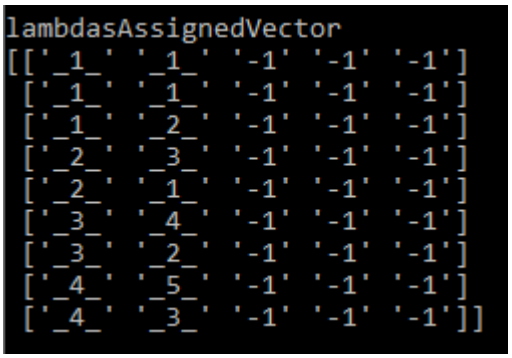

Figura 2.7. Resultados obtenidos – Matriz lambdasAssignedVector

Los resultados mostrados en la Figura 2.6 y en la Figura 2.7, pueden ilustrarse de mejor manera en la Figura 2.8. Los números mostrados en cada enlace corresponden a las longitudes de onda que atraviesan cada enlace; el valor de cero indica que no existen longitudes de onda atravesando ese enlace. Adicionalmente, los números contenidos en las flechas de color negro indican cuantas longitudes de onda requieren ingresar o salir de la red; si la flecha apunta hacia un nodo, indica que ese número de longitudes de onda requieren ser instalados ingresando por ese nodo; de manera similar, si la flecha sale del nodo, indica que ese nodo es el destino de ese número de longitudes de onda.

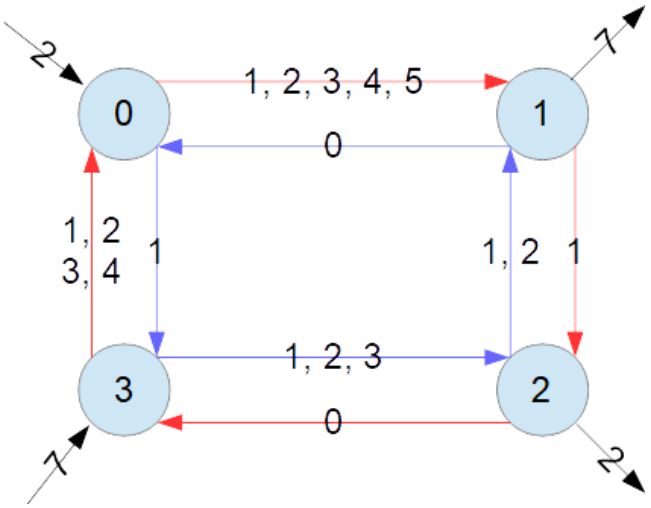

Figura 2.8. Resultados obtenidos – Matriz paths y matriz lambdasAssignedVector en la topología anillo.

La programación implementada en las líneas (57) y (58) del Código 2.1, permiten imprimir la matriz “omega1”. Esta matriz se muestra para contrastar los resultados de la matriz “omega1”, en comparación con la matriz “omega” original. La Figura 2.9 muestra el resultado obtenido para la matriz “omega1”, para el cálculo realizado con los datos definidos en la Figura 2.3. Este resultado puede contrastarse con la matriz “omega” mostrada en la Figura 2.3. La matriz original dispone de todos sus valores 0, con excepción de los valores en infinito que definen la ausencia de conexiones físicas entre dos nodos. El presente resultado muestra valores de 1, 25, 49 e infinito en posiciones previamente definidas como cero. Esta matriz se usa para el cálculo del enrutamiento en cada nueva iteración; el valor de 49 en la posición 3, 0 define un costo de 49 para atravesar el enlace con origen el nodo 3 y destino el nodo 0. En la Figura 2.5, se muestra el número de longitudes de onda que atraviesan este enlace, con un valor de 4. El valor equivalente de la matriz omega1 puede calcularse utilizando la fórmula definida en Ecuación 2.2.; de esta manera, el costo de llevar 4 longitudes de onda por un enlace se obtiene como $D_l(f_l = 4) = (2 * 4 - 1)^2 = 49$. Por otro lado, el valor de infinito encontrado en la fila 0, columna 1 no corresponde con el valor esperado; el número de longitudes de onda que atraviesan este enlace corresponde a 5, de acuerdo con lo definido en la Figura 2.5. Este valor debería obtener un valor de costo correspondiente a $D_l(f_l = 5) = (2 * 5 - 1)^2 = 81$; no obstante, se definió previamente que esta matriz es utilizada para realizar el cálculo de RWA y se requiere realizar el cambio del valor de 81 por un valor de infinito, para ingresar al recálculo del algoritmo de Dijkstra, pues el número de longitudes de onda que atraviesan el presente enlace es igual a la capacidad máxima del enlace. Este cambio se realiza para ingresar al

recálculo de Dijkstra una matriz de costo sin la posibilidad de seleccionar el enlace que ha alcanzado su capacidad máxima; este cambio se realiza en las líneas (32) y (33) mostradas en el Código 2.1.

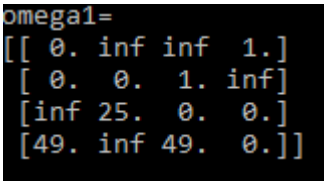

Figura 2.9. Resultados obtenidos – Matriz omega1

El código definido en las filas (59) y (60) del Código 2.1 permite imprimir la matriz "omega2". El resultado de esta matriz para los datos mostrados en la Figura 2.3, se muestra en la Figura 2.10. Se nota una gran similitud con la matriz "omega1" mostrada en la Figura 2.9, con un cambio en la posición 0, 1, donde no se encuentran valores de infinito diferentes a los definidos en la matriz "omega" original. Esta matriz se conserva para almacenar los costos reales calculados sobre la red, con la topología definida.

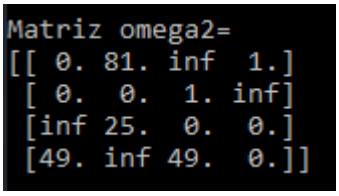

Figura 2.10. Resultados obtenidos – Matriz omega2

Las líneas de programación definidas en las filas (61) y (62) del Código 2.1 permiten imprimir la matriz de flujos resultantes. Esta matriz siempre se muestra en cero, pues se reduce en 1 el valor correspondiente en la matriz de flujo original, cuando se finaliza el cálculo de la ruta y longitudes de onda respectivos. El ejemplo de este resultado se muestra en la Figura 2.11; sus valores se comparan con la matriz "Flujo" mostrada en la Figura 2.3. Todas las longitudes de onda se han calculado y se ha reducido a cero todos los valores de la matriz correspondiente.

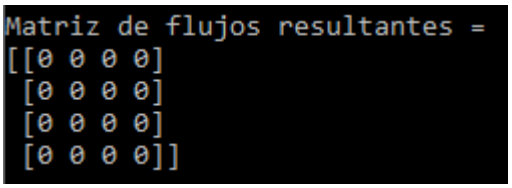

Figura 2.11. Resultados obtenidos – Matriz de flujos resultantes

El código implementado en las líneas (63) y (64) permiten imprimir los resultados obtenidos para la matriz "*lambdasAssigned*"; esta matriz contiene las longitudes de onda que atraviesan el cada enlace. Se utiliza el signo guion bajo para limitar una longitud de onda con la siguiente. La Figura 2.12 muestra los resultados obtenidos en la matriz "*lambdasAssigned*" con los datos definidos en la Figura 2.3. Estos resultados pueden compararse con los resultados mostrados en la Figura 2.8.

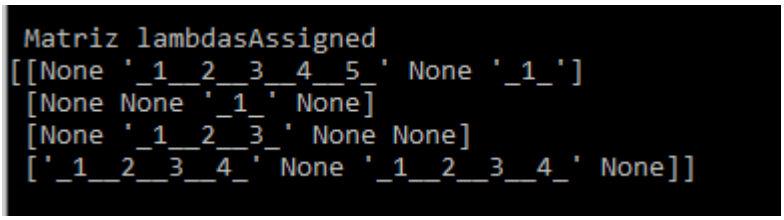

Figura 2.12. Resultados obtenidos – Matriz lambdasAssigned

Las líneas (65) y (66) mostradas en el Código 2.1 permiten realizar el cálculo e imprimir el costo total de transportar información por la red, de acuerdo con los resultados calculados para el problema RWA. La línea (65) utiliza la función "*dijsktraCostF*" perteneciente al módulo importado "dj". Esta función realiza el cálculo del costo total, de acuerdo con los valores contenidos en la matriz "omega2"; la Figura 2.13 muestra el resultado obtenido para el cálculo realizado con los datos definidos en la Figura 2.3. El valor 206 obtenido se calcula por medio de la suma de todos los valores de costo encontrados en la matriz "omega2", sin considerar los valores infinitos. El resultado de la matriz "omega2" puede observarse en la Figura 2.10. De esta manera, se calcula el costo total como: $D = 0 + 81 + 1 + 0 + 0 + 1 + 25 + 0 + 0 + 49 + 49 + 0 = 206$. Este valor se imprime en pantalla utilizando la línea (66) del programa.

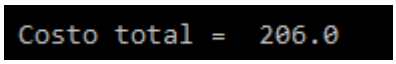

Figura 2.13. Resultados obtenidos – Costo total

Finalmente, la línea (67) contiene el código necesario para realizar el cálculo e impresión del tiempo total que el programa utilizó para realizar el cálculo. El tiempo se calcula desde la línea (8) del programa, posterior a la importación de librerías y al borrado de pantalla. A partir de este punto inicia la programación implementada para resolver el presente problema; el tiempo finaliza en la línea (42), previo a imprimir los resultados, pues la solución del problema planteado se encontró previo a este punto. En la Figura 2.14 se muestra el resultado obtenido para los datos definidos en la Figura 2.3. Este tiempo puede variar dependiendo de las características del sistema de cómputo que se utilice para

ejecutar el programa, incluyendo su capacidad de procesamiento, memoria disponible, otros procesos abiertos, etc. Adicionalmente, este valor cambiará cada vez que se ejecute el programa. El tiempo obtenido para el cálculo realizado corresponde a 6.47 ms.

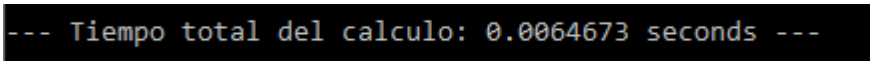

Figura 2.14. Resultados obtenidos – Tiempo total del cálculo

2.3. DESARROLLO DEL MÓDULO DE FUNCIONES DIJKSTRA-FIRSTFIT

El programa principal definido en la sección 2.2 permite entender el motivo de la implementación de las diferentes funciones implementadas en el módulo "*DijkstraFirstFit*".

Código 2.2. Código del módulo de funciones "*DijkstraFirstFit*"

```
(1) # coding=utf-8
(2) import numpy as np
(3) Inf=np.inf
(4) def dijkstra(omega, nodo):
(5)     m,n=omega.shape
(6)     d=np.empty(n)
(7)     d.fill(Inf)
(8)     known=np.zeros(n)
(9)     known[nodo]=1
(10)    path = np.empty(n)
(11)    path.fill(-1)
(12)    pivot=nodo
(13)    d[nodo]=0
(14)    while (np.any(d==Inf)):
(15)        for i in range (0,n):
(16)            if (known[i]!=1):
```

```
(17)                if d[i]<=(d[pivot]+omega[pivot][i]):
(18)                    pass
(19)                else:
(20)                    d[i]=d[pivot]+omega[pivot][i]
(21)                    path[i] = pivot
(22)            temp = d.copy()
(23)            while (known[np.argmin(temp)]==1):
(24)                temp[np.argmin(temp)]=Inf
(25)                if np.all( temp == Inf):
(26)                    break
(27)            if np.all( temp == Inf):
(28)                d = temp.copy()
(29)                return d, path
(30)                break
(31)            known[np.argmin(temp)]=1
(32)            pivot=np.argmin(temp)
(33)        return d, path
(34) def dijkstraPath(path, nodoD):
(35)        n = path.shape[0]
(36)        pathD=np.empty(n+1)
(37)        pathD.fill(-2)
(38)        pathD[0] = nodoD
(39)        for i in range (1, n+1):
(40)            if path[nodoD] != -1:
(41)                pathD [i] = path [nodoD]
(42)                nodoD = path [nodoD]
(43)                nodoD = nodoD.astype(int)
(44)            else:
(45)                pathD [i] = path [nodoD]
```

```
(46)             nodoD = path [nodoD]
(47)             nodoD = nodoD.astype(int)
(48)             break
(49)     pathD = pathD[::-1]
(50)     while pathD[n] != -1:
(51)         pathD = np.roll(pathD, 1)
(52)     return pathD
(53) def dijkstraCostF(omega1):
(54)     n,m = omega1.shape
(55)     costoT = 0
(56)     for i in range (0,n):
(57)         for j in range (0,m):
(58)             if (omega1[i][j] != Inf):
(59)                 costoT = costoT + omega1[i][j]
(60)     return costoT
(61) def flujoIncremental(x, pathD):
(62)     pathD = pathD.astype(np.int)
(63)     n = pathD.shape
(64)     for i in range (0, n[0] - 1):
(65)         if pathD[i+1] != -2 and pathD[i+1] != -1:
(66)             x[pathD[i]][pathD[i+1]] += 1
(67)         else:
(68)             break
(69)     return x
(70) def costoIncremental(omega1, x):
(71)     for i in range (0, omega1.shape[0]):
(72)         for j in range (0, omega1.shape[0]):
(73)             if x[i][j] != 0 :
(74)                 omega1[i][j] = (2*x[i][j] - 1) ** 2
```

```
(75)            else:
(76)                pass
(77)    return omega1
(78) def ffLambdasAssigned( pathD, capacidad, lambdasAssigned ):
(79)    [u] = pathD.shape
(80)    lambdasAssignedTemp = lambdasAssigned.copy()
(81)    lambdasAssignedPathD = np.empty((u), dtype = object )
(82)    lambdasAssignedPathD.fill('-1')
(83)    for i in range(0, u - 1):
(84)        for j in range(1, capacidad + 1):
(85)            test = '_' + str(j) + '_'
(86)            if pathD[i + 1] != -2 and  pathD[i + 1] != -1:
(87)                                                                    a = int(np.char.find(str(lambdasAssignedTemp[int(pathD[i])][int(pathD[i+1])]), test))
(88)                if a == -1:
(89)                    if  lambdasAssignedTemp[int(pathD[i])][int(pathD[i + 1])] == None:
(90)                        lambdasAssignedTemp[int(pathD[i])][int(pathD[i + 1])] = test
(91)                        lambdasAssignedPathD[i] = test
(92)                    else:
(93)                                    lambdasAssignedTemp[int(pathD[i])][int(pathD[i  +  1])] = lambdasAssignedTemp[int(pathD[i])][int(pathD[i + 1])] + test
(94)                        lambdasAssignedPathD[i] = test
(95)                    break
(96)    pathSolution = True
(97)    return lambdasAssignedTemp, lambdasAssignedPathD
```

La línea (1) del Código 2.2 se utiliza con el mismo propósito que en el código principal, para permitir ingresar caracteres españoles en los comentarios del programa. El código ingresado se muestra a continuación.

La línea (2) se requiere nuevamente para importar la librería NumPy con el alias “np”. Para el desarrollo del módulo “*DijlstraFirstFit”* no se requieren otras librerías, pues únicamente se definen funciones para realizar operaciones con las matrices y vectores ingresados como variables.

En la línea (3) se define la única constante de este módulo, correspondiente al valor de “Inf” que corresponde al valor de infinito de NumPy.

La función Dijkstra se desarrolla en el código escrito entre las líneas (4) y (33).

La línea (4) mostrada en el Código 2.2 define la función en Python, fijando el nombre de la función como “Dijkstra” y las variables de entrada como la matriz “omega” y el valor de “nodo”. La matriz “omega” ingresada en esta función corresponde a la matriz de costos y la variable “nodo” ingresada corresponde al origen “i” sobre el cual se realizará el cálculo.

Las líneas (6) y (7) permiten inicializar el vector “d”, con todos sus valores en infinito. Este vector almacenará los costos acumulados para alcanzar todos los nodos de la topología con origen “i”.

Las líneas (8) y (9) permiten inicializar el vector “*known”* con todos sus valores en cero, y modificar el valor del nodo “i” por 1. La columna fue seleccionada previamente como “pivot” para el cálculo definido en el algoritmo de Dijkstra y no puede ser seleccionado nuevamente para el cálculo. Las filas (10) y (11) permiten crear la variable “*path”* y llenar todos sus valores con -1; esta variable almacena los costos acumulados de llegar a todos los nodos de la red, con el origen el nodo “i”.

El código de la fila (12) permite asignar a la variable “pivot” el valor del nodo “i”. La variable “pivot” almacena el valor del nodo usado como pivot para cada iteración del algoritmo; este nodo se fija en cada nueva iteración y el algoritmo inicia fijando como nodo pivot al nodo origen “i”.

La fila (13) inicia el nodo origen “i” con el valor de cero en el vector “d”. Este valor define el nodo “i” como el origen del cálculo para el algoritmo de Dijkstra.

Las filas comprendidas entre el número (14) a la (32) comprenden el lazo “*while”* principal para el cálculo del algoritmo.

La fila (14) inicia el lazo “*while*” principal, que se repetirá si el vector “d” contiene algún valor infinito. Cuando no se tenga ningún valor en infinito se define que el cálculo terminó y que se termina el lazo “*while*”.

El lazo "*for"* incluido en la fila (15) permite recorrer todos los valores incluidos en el vector "d" y en la matriz "omega". El tamaño del vector "d" es "n" y el tamaño de la matriz omega es mxn siendo "m" igual a "n".

La sentencia de decisión "*if*" escrita en la fila (16) permite determinar si el nodo ha sido definido previamente como fijo; en caso afirmativo, no se realizará ningún cambio sobre el vector "d" y se continúa con el siguiente análisis. En caso negativo, se procede con el código de la fila (17).

El código implementado entre las filas (17) y (21) corresponde a la principal sentencia de decisión definida para el algoritmo de camino más corto de Dijkstra. El objetivo de este código es determinar si existe un nodo con un costo menor al costo definido previamente para alcanzar el destino "i" utilizando el nodo como origen, con el pivot como referencia. En caso de encontrar que el costo definido actualmente ($d[i]$) es menor al nuevo costo calculado usando el nodo origen con destino el nodo pivot más el costo de alcanzar el nodo "i" desde el nodo $pivot$ ($d[pivot] + omega[pivot][i]$), no se realiza ningún cambio sobre el vector "d", como se define en la fila (18); caso contrario, la sentencia ingresa a la programación definida en la línea (19), la cual permite ejecutar la programación contenida en las líneas (20) y (21). La programación escrita en la fila (20) permite reemplazar el costo de alcanzar el nodo "i" utilizando el camino anterior, por el costo de alcanzar el nodo "i" utilizando el camino definido por el nodo "pivot". Adicionalmente, se añade la fila (21) para almacenar el camino escogido como camino más corto, en la variable "*path*".

Las líneas de programación contenidas entre las líneas (22) y (32) permiten preparar las variables para la siguiente iteración. La línea (22) permite copiar la información contenida en el vector "d" hacia la variable "temp", sobre la cual se realizarán operaciones; de esta manera, se evita perder la información contenida en el vector "d" para entrar a la siguiente ejecución del lazo. El lazo "*while"* escrito en (23) permite reemplazar en el vector "d" los valores de costo de los nodos definidos previamente como fijos por un valor de infinito; este cambio evita seleccionar estos nodos como fijos o pivot para la siguiente iteración. En caso de determinar que todos los valores de todos los nodos en el vector "temp" han sido reemplazados por infinito, se determina que no existe solución factible para el problema planteado, utilizando las líneas (25) y (26) para salir del lazo "*while"* actual y proceder al análisis de la línea (27). La sentencia condicional "*if*" de la línea (27) determina si todos los valores contenidos en el vector "temp" son infinito; en este caso, se procede a copiar el valor del vector "temp" al vector "temp", regresando como resultado de la función este

vector “d” con todos sus valores en infinito y el valor actual del vector “path”. La línea (30) permite salir de la sentencia “*if*” actual. Las líneas (31) y (32) se ejecutan cuando se ha determinado que aún existe una solución factible al problema planteado, y cuando no se ha terminado la ejecución del algoritmo de Dijkstra; en este escenario, se determina cual es el nodo con el costo mínimo, definido en la variable “temp”; este nodo se utilizará como nodo “pivot” para la siguiente iteración del algoritmo, y se lo define como nodo conocido en el vector “known”, continuando con la siguiente ejecución del lazo principal. Al finalizar el cálculo del algoritmo de camino más corto, todos los valores del vector d que originalmente tenían en valor de infinito, han cambiado por un valor finito correspondiente al costo de alcanzar todos los nodos de la topología desde el nodo origen (variable “nodo”).

El Código 2.2 muestra el código implementado para desarrollar la función “*dijkstraPath*”, utilizada para encontrar el camino desde el nodo origen hacia el nodo destino, definido como “nodoD”, de acuerdo con los resultados del cálculo del camino más corto realizado por la función “dijkstra”. El objetivo de esta función es reconstruir de manera ordenada el camino entregado por la función “Dijkstra” para comunicar el nodo origen con el nodo destino. El nodo origen no se ingresa como variable para el cálculo, pues la variable “path” contiene el valor de -1 en nodo origen utilizado para el cálculo de dijkstra. Las variables ingresadas a la función corresponden al vector “*path*” obtenido como resultado de la función “Dijkstra” y el valor de “nodoD”, utilizado como nodo destino.

La línea (34) mostrada en el Código 2.2 define la función “dijkstraPath” y sus variables de entrada, como el vector “*path*” y el valor “nodoD”.

Las filas (35), (36), (37) y (38) del código permiten iniciar las variables “n” y “*pathD*”, que se utilizarán en el procesamiento respectivo. La fila (35) almacena en la variable “n” el valor del tamaño del vector “*path*”. Por otro lado, en (36) y (37) se inicia el vector “*pathD*” que almacenará el camino desde el origen hasta el destino; este vector se inicia con los valores -2, usando este valor como delimitador para el cálculo respectivo. Finalmente, el código escrito en (38) permite definir el valor del índice 0 del vector “*pathD*” con el valor del índice del nodo destino.

El lazo “*for*” definido en la línea (39) permite recorrer todo el vector “*pathD*”, evitando el primer valor, con un valor definido en la fila (38).

La programación implementada desde la fila (40) hasta la fila (48) permite reconstruir el camino, determinando todos los nodos requeridos para comunicar el origen con el destino. La línea (40) contiene la sentencia condicional “*if*” que permite determinar si se ha

alcanzado el final del cálculo; el procesamiento se realizará reconstruyendo el camino desde el destino hacia el origen, encontrando el nodo origen definido con el valor de -1; alcanzar este valor define el final del procesamiento, accediendo a la programación del condicional "*else"* en la línea (44) y ejecutando el programa contenido desde la fila (45) hasta la fila (48). Las filas (41) y (45) permiten almacenar en la variable "*path"* el siguiente salto, correspondiente al "nodoD" contenido en "*path*". El código contenido en las filas (42) y (46) reemplazan el valor del nodo destino por el siguiente salto contenido en el vector "*path*". La programación de las filas (43) y (47) permiten reemplazar el tipo de dato de la variable "nodoD" por un tipo entero (originalmente es un valor flotante); este cambio se requiere por el lazo condicional "*if*" de la fila (40), que usa este valor como índice del vector "*path*". Finalmente, la línea (48) utiliza la sentencia "*break"* para salir del lazo "*for*", al determinar el fin del cálculo alcanzando el nodo origen.

El código escrito en las filas (49), (50) y (51) permite cambiar la presentación de datos en el vector "pathD"; originalmente, el vector "pathD" es un vector columna con los valores ordenados desde destino en el índice cero hasta el origen; esta variable se requiere presentar como un vector fila, y con el nodo origen en la posición cero avanzando hacia el nodo destino. La fila (49) permite el cambio de un vector fila a un vector columna. Por otro lado, las filas (50) y (51) permiten invertir el orden los valores, permitiendo disponer del nodo origen en la posición 0.

La sentencia de la fila (52) permite retornar el vector "pathD" como respuesta para la función "dijkstraPath".

La función dijkstraCostF mostrada en el Código 2.2 se desarrolla con el propósito de calcular el costo final del resultado del enrutamiento calculado con el algoritmo implementado RWA. Esta función utiliza la matriz "omega1", que contiene los costos reales calculados para los resultados de enrutamiento definidos.

La línea (53) mostrada en la Código 2.2 permite definir la función "dijkstraCostF", seleccionando la matriz "omega1" como variable de entrada a la función.

Las filas (54) y (55) de programación permiten iniciar las variables "n", "m" y "costoT", que se utilizarán en el cálculo. Las variables "n" y "m" contienen respectivamente el número de filas y columnas que posee la matriz "omega1". El costo total ("costoT") se inicia con el valor de 0, sobre el cual se sumará los costos parciales de cada enlace.

La programación contenida entre las filas (56) y (59) permiten realizar el cálculo del costo total. Las filas (56) y (57) contienen los dos lazos “for” que recorren todas las filas y columnas de la matriz “omega1”. La fila (58) contiene la sentencia condicional “if” que determina si el valor contenido en la matriz es infinito; en este caso, este valor no se suma al cálculo realizado. Caso contrario, el programa accede a la línea (59) sumando al valor del costo total, el valor del costo del nuevo enlace analizado.

La línea (60) permite regresar como valor de retorno de la función, el número contenido en la variable “costoT”.

El Código 2.2 contiene el código implementado para desarrollar la función “flujoIncremental”. Esta función permite incrementar en 1 el valor del número de longitudes de onda contenidas en la variable x, para las nuevas longitudes de onda definidas como ruta en el programa principal, para el cálculo actual.

La línea de programación (61) mostrada en el Código 2.2, permite definir la función, con sus variables de entrada; la matriz x se utiliza como entrada de la función y será el resultado entregado por la función; por otro lado, el vector “pathD” se requiere para incrementar en 1 el número de longitudes de onda encontradas en los enlaces involucrados.

La programación encontrada en las líneas (62) y (63) inicializa las variables requeridas en la función. El vector “pathD” se convierte en tipo entero en la fila (62), para usar los valores contenidos en este vector como índices para el cálculo posterior. Adicionalmente, se crea la variable “n” que contiene las dimensiones del vector “pathD”.

Las filas contenidas entre (64) y (68) contienen el lazo principal. La fila (64) contiene el lazo “for” que recorre todos los índices del vector “pathD”. La sentencia “if” contenida entre las líneas (65) y (68) permite determinar si el lazo “for” ha llegado al final del análisis; esta condición se determina verificando el valor contenido en el vector “pathD” en el índice “i+1”; en esta posición se verifica la presencia de los valores -1 o -2, usados como delimitadores en este vector. Si no se encuentra uno de estos valores en el valor analizado, se determina que la ejecución de la función no ha terminado, y se procede con la línea (66). Caso contrario, se ejecuta la línea (67), que permite acceder a la función “*break”* contenida en la fila (68), que finaliza el lazo “*for*”. La línea (66) incrementa en 1 el enlace utilizado para interconectar el nodo contenido en el vector “pathD” en la posición “i” y el nodo de la posición i+1.

La fila (69) permite retornar la matriz “x” como resultado de la función.

El código contenido en Código 2.2 muestra la programación desarrollada para implementar la función “costoIncremental”. Esta función se define con el objetivo de realizar el cálculo del nuevo costo de cada uno de los enlaces de la red, considerando el número de longitudes de onda que recorren actualmente todos los enlaces y la función de costo definida en la Ecuación 2.2.

La línea (70) definida en el Código 2.2 permite definir la función “costoIncremental”, y sus variables de entrada. La matriz “omega1” se ingresa como dato de la función y contiene el costo actual de cada uno de los enlaces de la red; adicionalmente, será entregada como resultado de la función. La matriz “x” contiene el número de longitudes de onda que atraviesan actualmente todos los enlaces de la red; esta matriz se utiliza para realizar el cálculo de costo actual.

La programación realizada entre las líneas (71) y (76) contiene el lazo principal de la función. Las líneas (71) y (72) utilizan dos lazos “*for*” para recorrer todos los valores contenidos en las matrices “omega1” y x. La sentencia “*if*” mostrada en la línea (73) determina si el valor contenido en la matriz “x” es diferente de cero; en caso de contener un valor diferente de cero, se procede con el cómputo de la fila (74), la cual utiliza la Ecuación 2.2. Si el valor contenido en la matriz “x” es cero, la ecuación no aplica, y se mantiene el valor en 0.

La línea (77) permite devolver la matriz “omega1” como resultado de la función.

El Código 2.2 incluye la programación necesaria para crear la función “*ffLambdasAssigned*”. Esta función realiza el cálculo de la asignación de longitudes de onda, utilizando el algoritmo *first fit*. Este algoritmo enumera las longitudes de onda disponibles en cada enlace y las asigna en orden.

La línea de código (78) mostrada en la tabla contenida en Código 2.2 permite definir la función “*ffLambdasAssigned*”, y sus variables de entrada. La variable “*pathD*” contiene el camino desde el nodo origen hasta el nodo destino definido para la ruta respectiva; la variable capacidad contiene la capacidad máxima del enlace; por último, la variable lambdasAssigned contiene la matriz “lambdasAssigned” que dispone de todas las longitudes de onda asignadas previamente.

Las filas contenidas entre la línea (79) y (82) inician las variables requeridas para el cálculo realizado por la función. La fila (79) obtiene la variable u, que contiene el tamaño del vector “pathD”. La línea (80) copia la matriz “lambdasAssigned” en la variable

“lambdasAssignedTemp”, para manipular esta variable sin perder la información original. Las líneas (81) y (82) crean la variable “lambdasAssignedPathD” del tipo objeto, y la llena con el valor -1; esta variable almacena las longitudes de onda asignadas para el cálculo actual de ruta, de acuerdo con la información almacenada en el vector “pathD”.

Las líneas programadas desde el número (83) hasta el número (94) contienen el lazo principal del cálculo de FirstFit. La línea (83) contiene el lazo “*for*” con la variable “i”, utilizado para recorrer todo el vector “*pathD*”. La línea (84) contiene el lazo “*for”* con la variable “j”, utilizado para verificar todas las longitudes de onda disponibles en orden ascendente y determinar la primera longitud de onda libre.

La línea (85) crea la variable “*test*”, que almacena el valor de la longitud de onda analizada en el cálculo actual. Los símbolos de guion bajo se añaden al inicio y al final del número para marcar el inicio y fin de una longitud de onda.

La línea (86) contiene la sentencia “*if”* que permite determinar el fin del análisis; cuando el vector “*pathD*” contiene el valor -1 o -2 en la posición i+1, no se realizan cambios sobre las variables.

La fila (87) permite realizar el cálculo de la variable “a”; esta variable se utiliza para encontrar la posición de la longitud de onda almacenada en “*temp*”; en caso de encontrar esta longitud de onda almacenada en el enlace analizado, la variable a contiene el valor de la posición de esta longitud de onda; caso contrario, la función “*find”* devuelve un valor de -1 para la variable a. Este resultado se usará para asignar una longitud de onda disponible en el enlace, o para no realizar cambios e intentar con la siguiente longitud de onda.

La sentencia “*if*” de la fila (88) determina si la matriz “*lambdasAssignedTemp*” registra la longitud de onda analizada en la iteración actual (a diferente de -1) o si no se encuentra esta longitud de onda (a igual a -1). En caso afirmativo se procede con la línea (89), para seleccionar la longitud de onda analizada en el enlace respectivo. Caso contrario, se procede con la siguiente longitud de onda.

La programación contenida entre las líneas (89) y (94) permite asignar la longitud de onda actualmente analizada para el enlace respectivo. La fila (89) contiene la sentencia de decisión “*if*” que determina si el valor de longitudes de onda en el enlace analizado se encuentra vacío o si ya tiene almacenadas otras longitudes de onda; en caso de encontrar el valor vacío, se utiliza las líneas (90) y (91) para almacenar el valor de la longitud de onda contenida en la variable “*test*”, sobre la matriz *“lambdasAssignedTemp”* en el enlace

respectivo y sobre el vector "*lambdasAssignedPathD*" en la posición respectiva. Por otro lado, las líneas (93) y (94) permiten añadir la longitud de onda en la matriz "*lambdasAssignedTemp*" y en el vector "*lambdasAssignedPathD*", en las posiciones de los enlaces respectivos.

El código de la línea (95) permite salir del lazo "*for*" correspondiente a la variable "j", continuando con el análisis del siguiente enlace perteneciente a la ruta.

Finalmente, el código de la línea (96) permite retornar la matriz "*lambdasAssignedTemp*" y el vector "*LambdasAssignedPathD"* como resultado de la función.

2.4. DESARROLLO DE LA SOLUCIÓN TEÓRICA

El algoritmo propuesto las secciones 2.1, 2.2 y 2.3, permite encontrar la solución del problema RWA, utilizando los algoritmos planteados en el plan del presente trabajo de titulación; con el objetivo de determinar las ventajas de la solución formulada, es necesario compararla con los resultados obtenidos con un sistema computacional que obtenga una solución óptima o cercana a la óptima. Con este propósito en mente, es necesario obtener los resultados del proceso de optimización del sistema, utilizando sistemas computacionales de optimización de uso general, con los cuales comparar los resultados obtenidos.

Esta formulación requiere definir los resultados que permitirán comparar el funcionamiento del algoritmo planteado con los resultados obtenidos con el programa de optimización. Con esta premisa, se definen dos variables que permiten comparar el resultado obtenido entre los dos sistemas: el costo total del resultado obtenido y el tiempo de cómputo requerido para obtener esta solución. El costo del sistema muestra la eficacia del algoritmo, pues el objetivo del algoritmo es minimizar la función de costo del sistema; por otro lado, el tiempo computacional muestra la eficiencia del algoritmo.

El programa computacional seleccionado para realizar este cómputo corresponde a MiniZinc. Este sistema utiliza el lenguaje MiniZinc, diseñado para resolver problemas de optimización con restricciones y problemas de decisión sobre números enteros y reales [28]. El motivo de escoger el programa MiniZinc proviene de compartir el mismo objetivo de optimización requerido en el problema RWA planteado.

El lenguaje MiniZinc utiliza un formato de escritura muy sencillo, pues se asemeja mucho al planteamiento del problema matemático. Por este motivo, el primer paso para escribir el código del programa es plantear el problema matemático que se requiere optimizar. La formulación del problema para MiniZinc no es la misma en comparación con el planteamiento realizado para Python, pues MiniZinc no procesa matrices. Sin embargo, el programa permite la escritura del problema en manera de las fórmulas que lo describen. Para alcanzar este objetivo, se realizará el modelado del problema RWA descrito en las secciones 2.1, 2.2 y 2.3, mostrando en la presente sección la formulación del problema y sus ecuaciones, con el objetivo de ingresarlos en el programa.

La topología que se analizará se muestra en la Figura 2.15. Esta es la formulación original del problema planteado. La red dispone de 4 nodos conectados en topología de anillo, con el costo cero en todos sus enlaces; adicionalmente, los nodos 0 y 3 son fuente o *source* de la información y se marcan con la letra “s”; de manera equivalente, los nodos 1 y 2 son destino o *destination* de la información y se utiliza la letra “d” para nombrarlos. Los nombres mostrados sobre los enlaces se nombran con la letra x y corresponden al flujo entre un nodo y su destino; el flujo x01 corresponde al flujo que se origina en el nodo 0 y que alcanza al nodo 1 como destino.

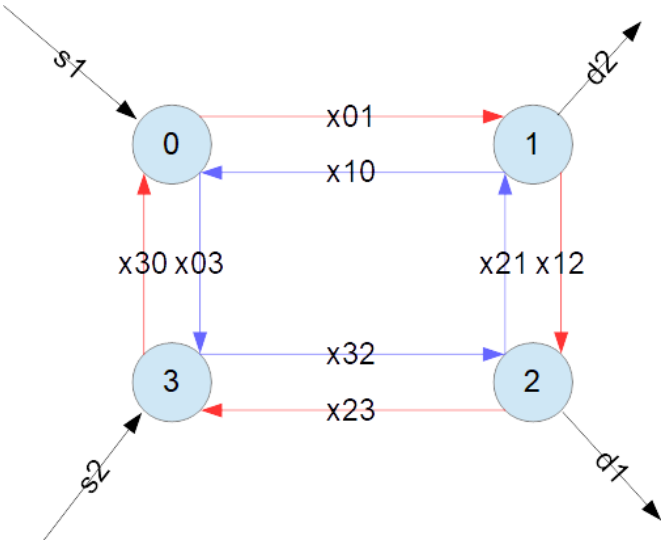

Figura 2.15. Solución teórica en MiniZinc - Topología

RESTRICCIONES: Es necesario considerar las siguientes restricciones aplicables sobre el presente problema:

1. Equilibrio del sistema: toda la información que ingresa a la red debe tener un destino y debe salir de la red. Esta restricción permite formular las siguientes ecuaciones.

$$s_1 = d_1 \qquad (2.5)$$
$$s_2 = d_2$$

Estas restricciones se han formulado previamente en la matriz de flujo mostrada en la Figura 2.3. se definió un nodo origen y un nodo destino para los datos, manteniendo el equilibrio del sistema.

2. Equilibrio de la información en la red: la información que circula por la red debe cumplir las mismas restricciones de equilibrio; esta restricción permite definir que la sumatoria de los flujos que ingresan a un nodo debe ser igual a la sumatoria de los flujos que salen del mismo. Esta restricción puede observarse en la formulación de la Ecuación 2.5.

$$\sum \boldsymbol{flujos - in} \; = \; \sum \boldsymbol{flujos - out} \qquad (2.6)$$

La Ecuación 2.5. permite encontrar las formulaciones mostradas en la Ecuación 2.7. aplicadas sobre cada nodo de la topología mostrada en la Figura 2.15. En el lado izquierdo del símbolo de igualdad se muestran los flujos que ingresan a cada nodo y en el lado derecho se definen los flujos que salen del mismo nodo.

$$\begin{aligned} s_1 + x_{10} + x_{30} &= x_{01} + x_{03} \; (\boldsymbol{para\ el\ nodo\ 0}) \\ x_{01} + x_{21} &= x_{10} + x_{12} + d_2 \; (\boldsymbol{para\ el\ nodo\ 1}) \\ x_{12} + x_{32} &= x_{21} + x_{23} + d_1 \; (\boldsymbol{para\ el\ nodo\ 2}) \\ s_2 + x_{03} + x_{23} &= x_{30} + x_{32} \; (\boldsymbol{para\ el\ nodo\ 3}) \end{aligned} \qquad (2.7)$$

Es necesario denotar que las ecuaciones definidas en la Ecuación 2.10. utilizan una gran cantidad de variables, y el número de variables incrementa considerablemente con el incremento del número de nodos de la red. Esto convierte a esta formulación una propuesta poco escalable, pues el programa incrementa considerablemente el procesamiento requerido, y su incremento está directamente relacionado con el número de variables y ecuaciones. Con el objetivo de disminuir la complejidad de las ecuaciones ingresadas al sistema, se realiza la siguiente formulación, considerando la topología de anillo utilizada.

Cualquier flujo requerido tiene un nodo origen (s) y un nodo destino (d). Los caminos posibles entre el nodo origen y el nodo destino se reducen a dos posibilidades: un camino que atraviese los nodos en sentido horario y un camino que atraviese la red en sentido antihorario. Con esta formulación, se puede graficar nuevamente la topología descrita en la Figura 2.15, y cambiar la formulación por la mostrada en la Figura 2.16.

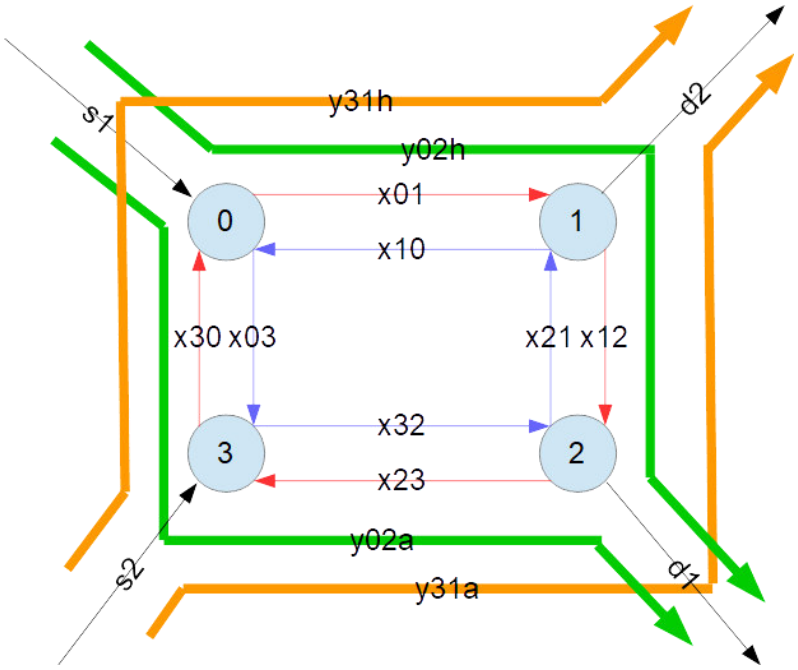

Figura 2.16. Solución teórica en MiniZinc – Topología con flujos horarios y antihorarios

La topología mostrada en la Figura 2.16 permite escribir las ecuaciones de equilibrio de la información en función de estas nuevas variables, simplificando el procesamiento requerido. Los super índices de las variables de salida y denotan el sentido horario (h) o antihorario (a) del flujo.

$$\begin{aligned} x_{01} &= {y_{31}}^{h} + {y_{02}}^{h} \\ x_{10} &= 0 \\ x_{12} &= {y_{31}}^{h} \\ x_{21} &= {y_{31}}^{a} \\ x_{23} &= 0 \\ x_{32} &= {y_{31}}^{a} + {y_{02}}^{a} \\ x_{30} &= {y_{31}}^{h} \\ x_{03} &= {y_{02}}^{a} \end{aligned} \tag{2.8}$$

Adicionalmente, las ecuaciones de equilibrio del sistema se pueden reescribir de acuerdo con la formulación mostrada en la Ecuación 2.9.

$$\begin{aligned} s_1 = d_1 &= {y_{02}}^{h} + {y_{02}}^{a} \\ s_2 = d_2 &= {y_{31}}^{h} + {y_{31}}^{a} \end{aligned} \tag{2.9}$$

3. Restricción de capacidad: el problema definido, mantiene la restricción de un número máximo de longitudes de onda que pueden atravesar cada enlace; esta restricción puede definirse con la siguiente inecuación.

$$f_l \leq |C| \tag{2.10}$$

Con base en esta formulación, se pueden definir las siguientes restricciones para el problema formulado con la Figura 2.15.

$$\begin{aligned} x_{01} &\leq C \\ x_{10} &\leq C \\ x_{12} &\leq C \\ x_{21} &\leq C \\ x_{23} &\leq C \\ x_{32} &\leq C \\ x_{30} &\leq C \\ x_{03} &\leq C \end{aligned} \tag{2.11}$$

4. Restricción de flujos negativos: esta restricción es similar a la condición 3. Es lógico pensar que el número de longitudes de onda que atraviesan un enlace debe ser un número negativo; no obstante, es necesario ingresar esta información al algoritmo para limitar los posibles resultados. Esta restricción puede definirse con las siguientes inecuaciones.

$$\begin{aligned} x_{01} &\geq 0 \\ x_{10} &\geq 0 \\ x_{12} &\geq 0 \\ x_{21} &\geq 0 \\ x_{23} &\geq 0 \\ x_{32} &\geq 0 \\ x_{30} &\geq 0 \\ x_{03} &\geq 0 \end{aligned} \tag{2.12}$$

FUNCION DE OPTIMIZACIÓN: en el lenguaje MiniZinc es necesario definir la función que se requiere optimizar en función de las variables seleccionadas. Con el objetivo de comparar el resultado obtenido con el programa MiniZinc con los resultados obtenidos con el algoritmo desarrollado, es necesario utilizar la misma función de costo definida para el algoritmo formulado. Esta función se observa en la Figura 2.1 y su formulación se define en la Ecuación 2.1.

Las definiciones realizadas en Ecuación 2.8., Ecuación 2.9., Ecuación 2.11. y Ecuación 2.12., permiten desarrollar el código definido en Código 2.3.

Código 2.3. Solución teórica en MiniZinc – Código fuente de MiniZinc

(1) int: capacidad = 5;
(2) int: S02=2;

(3) int: S31=7;
(4) var int: Y02h;
(5) var int: Y02a;
(6) var int: Y31h;
(7) var int: Y31a;
(8) var int: X01;
(9) var int: X10;
(10) var int: X12;
(11) var int: X21;
(12) var int: X23;
(13) var int: X32;
(14) var int: X30;
(15) var int: X03;
(16) constraint S02=Y02h+Y02a;
(17) constraint S31=Y31h+Y31a;
(18) constraint X01=Y02h+Y31h;
(19) constraint X10=0;
(20) constraint X12=Y02h;
(21) constraint X21=Y31a;
(22) constraint X23=0;
(23) constraint X32=Y31a+Y02a;
(24) constraint X30=Y31h;
(25) constraint X03=Y02a;
(26) constraint X01<=capacidad;
(27) constraint X10<=capacidad;
(28) constraint X12<=capacidad;
(29) constraint X21<=capacidad;
(30) constraint X23<=capacidad;
(31) constraint X32<=capacidad;

```
(32) constraint X30<=capacidad;
(33) constraint X03<=capacidad;
(34) constraint X01>=0;
(35) constraint X10>=0;
(36) constraint X12>=0;
(37) constraint X21>=0;
(38) constraint X23>=0;
(39) constraint X32>=0;
(40) constraint X30>=0;
(41) constraint X03>=0;
(42) var float: costo;
(43) costo =  if X01 != 0 then ((2*X01)-1)*((2*X01)-1) else 0 endif +
            if X10 != 0 then ((2*X10)-1)*((2*X10)-1) else 0 endif +
            if X12 != 0 then ((2*X12)-1)*((2*X12)-1) else 0 endif +
            if X21 != 0 then ((2*X21)-1)*((2*X21)-1) else 0 endif +
            if X23 != 0 then ((2*X23)-1)*((2*X23)-1) else 0 endif +
            if X32 != 0 then ((2*X32)-1)*((2*X32)-1) else 0 endif +
            if X30 != 0 then ((2*X30)-1)*((2*X30)-1) else 0 endif +
            if X03 != 0 then ((2*X03)-1)*((2*X03)-1) else 0 endif;
(44) solve minimize costo;
(45) output ["X12= ",show(X01), "\n"];
(46) output ["X21= ",show(X10), "\n"];
(47) output ["X23= ",show(X12), "\n"];
(48) output ["X32= ",show(X21), "\n"];
(49) output ["X34= ",show(X23), "\n"];
(50) output ["X43= ",show(X32), "\n"];
(51) output ["X41= ",show(X30), "\n"];
(52) output ["X14= ",show(X03), "\n"];
(53) output ["Y13h= ",show(Y02h), "\n"];
```

(54) output ["Y13a= ",show(Y02a), "\n"];
(55) output ["Y42h= ",show(Y31h), "\n"];
(56) output ["Y42a= ",show(Y31a), "\n"];
(57) output ["Costo de del camino óptimo = ", show(costo),"\n"];

El programa mostrado en el Código 2.3 es una forma sencilla de codificar las ecuaciones descritas previamente en el programa. El detalle del código se describe a continuación.

En el lenguaje MiniZinc, es necesario definir las constantes del programa como parámetros. Las líneas (1) hasta (3) del código fuente mostrado en el Código 2.3 contienen las constantes requeridas en el programa y corresponden al valor de la capacidad y los flujos. Los valores de estas constantes son del tipo entero, definiendo esta característica con la sentencia “int”. Adicionalmente, debe notarse que el lenguaje MiniZinc requiere terminar una línea de programación con el símbolo punto y coma e iniciar las líneas de comentarios con el símbolo de porcentaje.

El programa MiniZinc requiere definir las variables de decisión utilizadas en el programa. Estas variables corresponden a las variables de salida “Y”, y a las variables de entrada “X”. La definición de estas variables se encuentra entre las líneas (4) y (15) del código mostrado en el Código 2.3. Las variables de MiniZinc se declaran con la sentencia “var” al inicio de la línea, seguido del tipo de variable. En este escenario, las variables de entrada y de salida son del tipo entero, pues no se puede establecer una cantidad parcial de longitudes de onda entre el origen y destino.

El siguiente paso es definir las restricciones del programa. Las líneas contenidas entre las filas (16) y (41) contienen las restricciones mostradas en el Código 2.3. Se incluyen las restricciones definidas en Ecuación 2.8., Ecuación 2.9., Ecuación 2.11. y Ecuación 2.12.

La función de costo se define en las líneas (42) y (43) del Código 2.3. Se define la función de costo con el tipo flotante y se define la función de costo a optimizar. El lenguaje MiniZinc permite definir sentencias “*if*” dentro de las funciones; la función de costo mencionada en Ecuación 2.2. se define con estas sentencias.

El programa de optimización permite definir el objetivo de la optimización. Para el problema actual, es necesario minimizar la función de costo. La función de optimización mostrada en el Código 2.3 en la línea (44) muestra la función “*solve*” que define el objetivo de

optimización del programa, con el parámetro "*minimize"* que determina la minimización de la función costo.

Finalmente, el programa permite imprimir en pantalla los resultados obtenidos con la optimización calculada. Las líneas comprendidas entre (45) y (57) permiten imprimir en pantalla estos resultados; estas líneas de programación utilizan la función output para imprimir en pantalla el texto ingresado entre corchetes. Adicionalmente, la función "*show*" permite imprimir el valor contenido en una variable.

3. RESULTADOS Y DISCUSIÓN

La sección actual tiene el objetivo de analizar los resultados obtenidos con el algoritmo implementado en la sección 2, comparando con los resultados obtenidos con el programa de optimización MiniZinc; se realizará la comparación del costo total de la red obtenido con el valor que arroje un programa de optimización de uso general y el tiempo tomado para encontrar el resultado. El valor del costo de transmisión de información en cada enlace se obtiene con la fórmula definida en la Ecuación 2.2., en función del número de *lightpaths* que atraviesan esa conexión. El costo total de transmisión de información en la red se obtiene con la suma de todos los costos de todos los enlaces, con la solución ofertada.

La comparación de los valores de costo total de la red se realiza utilizando la variación de dos parámetros: el número de *lightpaths* que se optimizan y el número de nodos en la red. La Tabla 3.1 muestra cuatro escenarios diferentes planteados para comparar los resultados obtenidos; estas condiciones se aplican sobre el algoritmo planteado y se solucionan en el sistema de optimización, con el objetivo de analizar los costos totales resultantes y compararlos en la presente sección.

Tabla 3.1. Análisis de resultados - Parámetros de variación

No.	Cantidad de lightpaths	Cantidad de Nodos
1	10	4
2	10	20
3	50	4
4	50	20

Con las definiciones previas, se procede a definir las topologías, longitudes de onda de entrada y salida y la capacidad máxima de los enlaces. La información de las pruebas realizadas y sus resultados se adjuntan al presente documento, incluyendo los valores de costo resultantes y el tiempo utilizado por el algoritmo. Estos resultados han sido tabulados, e incluidos en el ANEXO . El procesamiento de estos resultados puede observarse en la Figura 3.1, en la Figura 3.2, la Figura 3.3 y la Figura 3.4; el análisis respectivo se muestra posterior a cada imagen, empezando por la descripción general de cada uno de los

resultados mostrados, y culminando posteriormente con el análisis comparativo de estos resultados.

La Figura 3.1 muestra los resultados del promedio del error entre los costos obtenidos con el algoritmo Dijkstra-FirstFit, respecto a los valores mínimos teóricos obtenidos con el programa de optimización MiniZinc. Estos valores definen un error relativo entre los costos resultantes con una desviación del 14.59% del valor real respecto al valor teórico. Este valor demuestra que los resultados obtenidos con el algoritmo no coinciden exactamente con los valores de costo mínimo obtenidos en la solución teórica. Esta solución corresponde con los valores esperados, para con la implementación del algoritmo de Dijkstra para el cálculo del costo mínimo, el cual puede obtener valores diferentes al óptimo.

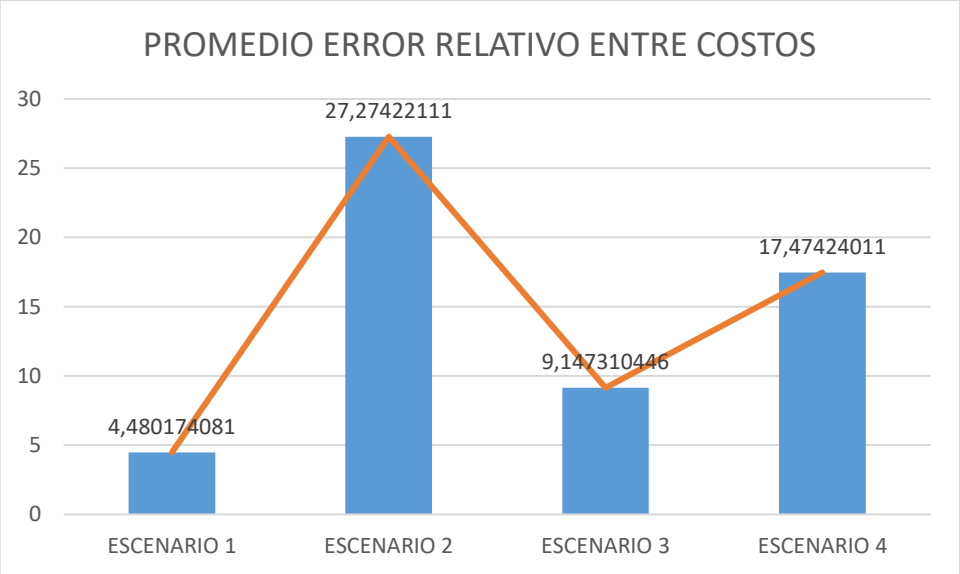

Figura 3.1. Análisis de resultados – Promedio de error relativo entre costos

- El escenario 1 se define con una topología de 4 nodos y un total del 10 *lightpaths* entre origen y destino. Muestran un valor de 4.48% de error entre los valores teóricos y los valores reales; el error obtenido para este escenario con una pequeña cantidad de nodos y una cantidad reducida de *lightpaths*, permite observar resultados cercanos a los valores teóricos en comparación con el algoritmo desarrollado.
- El escenario 2 utiliza una topología de 20 nodos y un total del 10 *lightpaths* entre origen y destino. Los valores de costo resultantes encuentran un 27.27% de error entre los costos teóricos y reales. En comparación con el escenario anterior, se incrementa el número de nodos en la red a un valor de 20 nodos. Con el incremento

de nodos en el sistema, se incrementa en gran medida los errores obtenidos con el algoritmo propuesto.

- El escenario 3 muestra un error relativo de 9.15% respecto a los valores teóricos obtenidos; estos resultados se obtienen con una red de 4 nodos y un total de 50 *lightpaths*. Este resultado muestra que el incremento del número de *lightpaths* entre origen y destino incrementa error obtenido para la optimización realizada; sin embargo, el incremento del número de *lightpaths* desde 10 *lightpaths* hasta 50 *lightpaths* resulta en un incremento de alrededor de 4% en el error, con resultados favorables.
- El escenario 4 obtiene un error relativo de 17.47% respecto a los valores teóricos obtenidos para el valor del costo. Estos resultados se obtienen con un número de 50 *lightpaths* distribuidos en 20 nodos. Con el incremento del número de nodos y del número de *lightpaths* se nota un incremento en el error obtenido para los valores de costo de la topología completa.

Los resultados mostrados en la Figura 3.1, muestran un incremento en el error del costo encontrado por el algoritmo, respecto al valor teórico calculado. El error incrementa con el incremento del número de nodos en la red y con el incremento del número de *lightpaths* en la red.

La Figura 3.2 muestra el promedio de diferencia porcentual en tiempo de cómputo. Estos resultados permiten observar que el tiempo de cómputo utilizado por el programa optimizador MiniZinc es 82408.69% más lento que el algoritmo desarrollado con los algoritmos Dijkstra-FirstFit. Esta resolución demuestra la factibilidad de usar el algoritmo propuesto para la solución de problemas RWA, por sus tiempos computacionales reducidos, en comparación con el programa de optimización MiniZinc. Durante las pruebas realizadas, se encontraron escenarios con un tiempo de cómputo mayor a dos días en el programa MiniZinc; en contraposición, el tiempo de cómputo del algoritmo propuesto se mantiene en el orden de los cientos de milisegundos en el escenario más complicado.

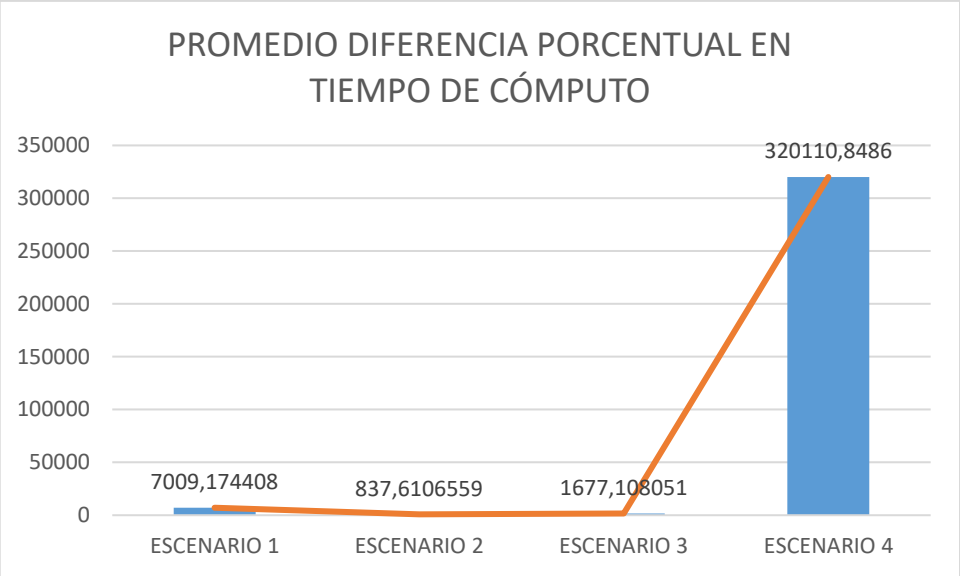

Figura 3.2. Análisis de resultados – Promedio de diferencia porcentual en tiempo de cómputo

- En el escenario 1, con un problema compuesto por 4 nodos y 10 *lightpaths*, los resultados muestran un tiempo promedio 7009.17% superior en el tiempo de cómputo del programa optimizador MiniZinc, respecto al tiempo obtenido por el algoritmo propuesto. Los resultados obtenidos permiten observar tiempos de cómputo menores a los obtenidos con la solución matemática del problema.
- En el escenario 2, con el cálculo de la solución del problema para 20 nodos con 10 *lightpaths*, se registra un tiempo promedio 837.61% mayor del cálculo realizado en el programa MiniZinc en comparación con el programa desarrollado.
- El escenario 3, propuesto con 4 nodos y 50 *lightpaths* entrega un tiempo de cómputo 1677.11% mayor en el programa optimizador, en comparación con el algoritmo propuesto.
- El escenario 4 delimitado para una red con 20 nodos y 50 longitudes de onda, muestra el tiempo de cómputo del programa optimizador supera en un valor promedio de 320110.85% al tiempo de cómputo encontrado por el programa desarrollado en Python. Al incrementar el número de variables, incluyendo el número de nodos en la red y el número de *lightpaths*, la complejidad del problema matemático incrementa considerablemente, acrecentando en la misma medida el procesamiento requerido por el programa MiniZinc para encontrar una solución.

Los resultados de la Figura 3.2, muestran que el algoritmo Dijkstra-FirstFit implementado en Python llega al resultado esperado en tiempos computacionales mucho menores (82408.69% más rápidos) en comparación con el sistema de optimización utilizado. Estos resultados manifiestan la factibilidad de utilizar este algoritmo en escenarios reales para la solución de problemas RWA.

La Figura 3.3 muestran los valores de tiempos promedio de solución en el programa optimizador MiniZinc. Los resultados obtenidos definen un tiempo de 136.99 segundos (aproximando a 137 segundos) en promedio para encontrar la solución en los escenarios estudiados; estos tiempos de procesamiento muestran tiempos de cálculo 2 minutos y 17 segundos en promedio para el programa optimizador, los cuales son muy elevados e inadecuados para su uso en escenarios reales, considerando tiempos de re-convergencia de tablas de enrutamiento en el orden de los milisegundos, requeridos para minimizar la afectación sobre un servicio de fibra óptica, con la capacidad de transporte de una gran cantidad de información y con una gran cantidad de suscriptores utilizando este servicio.

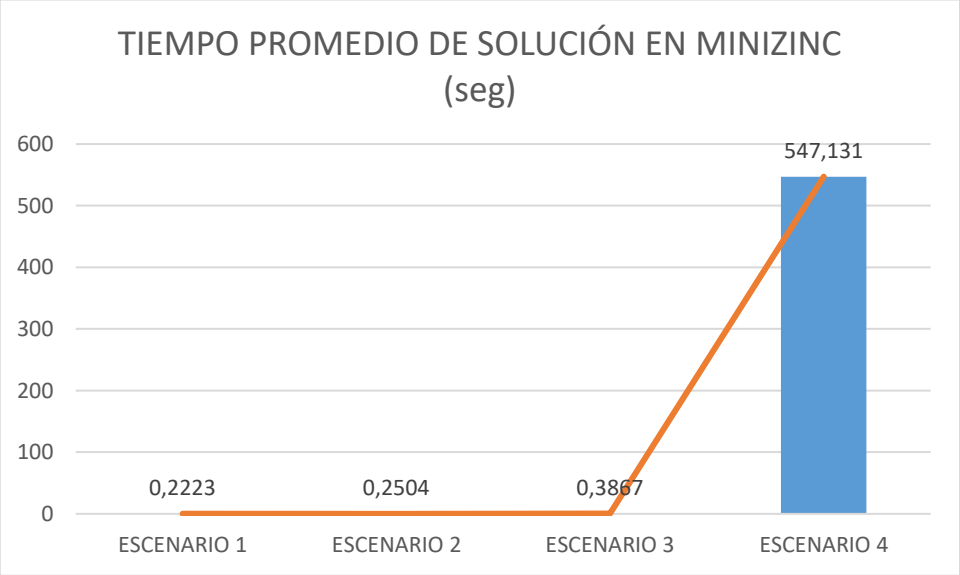

Figura 3.3. Análisis de resultados – Tiempo promedio de solución en MiniZinc

- En el escenario 1, analizando una red de 4 nodos con 10 *lightpaths*, el tiempo promedio de la solución del programa optimizador MiniZinc corresponde a 222.3ms. Este es el escenario más sencillo de todos los escenarios analizados.
- En el escenario 2, analizando una red de 20 nodos con 10 *lightpaths*, el tiempo promedio de la solución del programa optimizador MiniZinc corresponde a 250.4ms.

Se observa un incremento de 28ms respecto al escenario anterior, con un incremento considerable de número de nodos en la red.

- En el escenario 3, analizando una red de 4 nodos con 50 *lightpaths*, el tiempo promedio de la solución del programa optimizador MiniZinc corresponde a 386.7ms. Se observa un incremento de 136ms respecto al primer escenario, con un incremento considerable de *lightpaths* en la red.
- En el escenario 4, analizando una red de 20 nodos con 50 *lightpaths*, el tiempo promedio de la solución del programa optimizador MiniZinc corresponde a 547.13 segundos. El tiempo requerido para encontrar la solución al problema RWA en una red con una cantidad considerable de nodos y una cantidad similar de *lightpaths*, se incrementa considerablemente, mostrando su baja eficiencia para resolver este problema en escenarios reales.

Los resultados mostrados en la Figura 3.3 muestran que el programa de optimización MiniZinc obtiene resultados óptimos en tiempos computacionales del orden de las décimas de segundo. Estos tiempos pueden incrementarse considerablemente, dependiendo el problema planteado. Debe mencionarse que el sistema de optimización busca encontrar la solución óptima, analizando todos los posibles resultados; en consecuencia, un cambio pequeño en las condiciones iniciales puede llevar a escenarios en que el sistema de optimización necesite de tiempos del orden de los segundos, llegando a encontrar soluciones en horas o inclusive días.

La Figura 3.4 muestra los tiempos promedio de solución con el algoritmo Dijkstra-FirstFit Estos resultados definen un valor promedio de 55.98 ms en encontrar una solución utilizando el algoritmo Dijkstra-FirstFit en todos los escenarios. Este valor considera una convergencia total de la red, realizando un cálculo completo para todos los *lightpaths* requeridos.

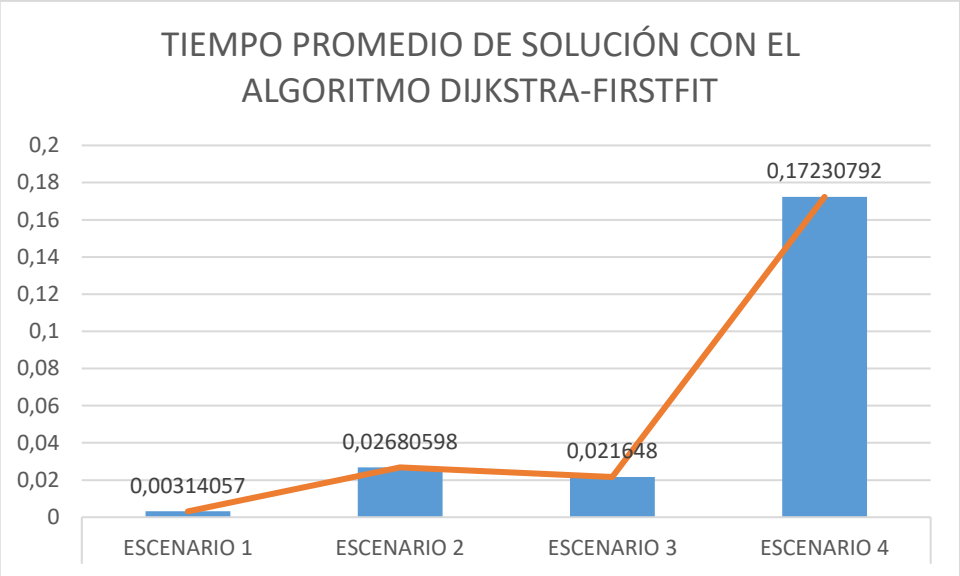

Figura 3.4. Análisis de resultados – Tiempo promedio de solución con el algoritmo Dijkstra-FirstFit

- El tiempo promedio de solución del algoritmo propuesto en el escenario 1, con 4 nodos y 10 *lightpaths*, corresponde a 3.14ms. Este tiempo muestra una gran velocidad para calcular el resultado, considerando una pequeña cantidad de nodos y una pequeña cantidad de longitudes de onda.
- El resultado obtenido para el escenario 2 se obtiene en una red con 4 nodos y un total de 50 *lightpaths*. Se incrementa el número de lightpaths en comparación con el escenario anterior, obteniendo el resultado de la optimización en un tiempo de 26.81ms.
- En el escenario 3, se incrementa el número de nodos a 20 en comparación con el escenario 1, y se mantiene el número de *lightpaths* en 10; con estos valores, el tiempo de convergencia del algoritmo en promedio corresponde a 21.65ms.
- El tiempo promedio de solución del algoritmo implementado en Python para el escenario 4, con 20 nodos y 50 *lightpaths*, corresponde a 172.31 ms. Se considera un incremento del número de nodos y del número de longitudes de onda respecto al escenario 1, manteniendo el tiempo de resolución en el orden de las décimas de segundo.

Los resultados mostrados en la Figura 3.4, muestran tiempos computacionales muy pequeños (en el orden de los milisegundos) para el algoritmo Dijkstra-FirstFit propuesto. Los tiempos de solución se incrementan con el incremento del número de *lightpaths* y el

incremento del número de nodos en la red; cualquiera de estos dos cambios, resultan en un incremento en el número de iteraciones que requiere el algoritmo para encontrar una solución al problema planteado.

Ahora que se ha terminado de describir los resultados obtenidos, es necesario comparar los resultados previamente definidos. A continuación, se muestra el análisis realizado, de acuerdo con los resultados obtenidos.

- En promedio, el error entre los costos resultantes del proceso de optimización es de 14.59%; este valor muestra que el algoritmo propuesto no obtiene resultados iguales al óptimo en todos los escenarios. Para analizar este resultado, es necesario recordar el algoritmo implementado; en el Código 2.1, en las líneas (17), (18) y (19) puede observarse el lazo implementado para calcular uno a uno todos los *lightpaths* requeridos en la red. Este lazo muestra tres características:
    - El análisis se realiza en orden ascendente, buscando entre todas las filas y en todas las columnas, un *lightpath* requerido entre origen y destino. Este análisis se realiza seleccionando una fila de forma ascendente, y seleccionando una columna dentro de esta fila de forma ascendente; al terminar con la fila analizada, se continua con la siguiente.
    - Al encontrar un valor de *lightpath* origen-destino en uno de los puntos analizados, se optimiza uno de los *lightpaths* requeridos y se continua con el siguiente *lightpath* con el mismo origen-destino hasta terminar con el requerimiento origen-destino; puede observarse este comportamiento en el Código 2.1, línea (34), donde se decrementa en 1 el valor contenido en la matriz flujo.
    - El análisis de la optimización de la siguiente iteración se realiza utilizando los resultados previamente calculados y asignados para realizar la optimización del siguiente *lightpath*; esta conducta puede visualizarse en el Código 2.1, en las líneas (22) y (30), donde se visualiza que la matriz “omega1” se actualiza en cada iteración y se vuelve a utilizar en el siguiente cómputo del lazo principal.

Estas tres características, difieren del comportamiento del programa de optimización MiniZinc, pues el programa de optimización utiliza todas las posibles respuestas, llegando al costo mínimo. El resultado obtenido en el valor de costo puede mejorarse al seleccionar

aleatoriamente el *lightpath* que se optimizará en cada iteración; este cambio permite obtener resultados diferentes para el mismo escenario, si se realiza un nuevo cálculo, mejorando probablemente el valor del costo resultante.

- La Figura 3.3 y la Figura 3.4 obtienen en promedio los valores de 136,99 segundos y 0,055975618 segundos para los resultados obtenidos con MiniZinc y con el algoritmo Dijkstra-FirstFit respectivamente. Estos valores se originan de la tabla de resultados globales mostrados en el ANEXO A y muestran que el valor del tiempo de cálculo en el programa MiniZinc no incrementa linealmente con el incremento del número de nodos en la red o con el incremento del número de *lightpaths* para optimizar; estos valores son resultados del cambio del problema planteado con pequeños cambios en el problema original. En algunos escenarios, se encuentra tiempos de cómputo muy elevados en comparación con escenarios similares; este no es el caso de los tiempos de cómputo requeridos por el algoritmo propuesto, notando un incremento del tiempo de solución, con el incremento de cualquiera de las dos variables. Parte del código implementado, se utiliza para almacenar la información necesaria para mostrar los cálculos realizados, como puede apreciarse en el Código 2.1 en la línea (31), donde se observa la aplicación de la función “costoIncremental” sobre la matriz omega2; estos resultados se requieren para el presente proyecto, pues se requiere visualizar claramente y comparar los resultados del proceso de optimización realizado. en un escenario real, se elimina esta programación, reduciendo el tiempo de cómputo de cada iteración.

Se pueden reducir tiempos de procesamiento, utilizando lenguajes de programación diferentes a Python, pues este lenguaje corresponde al tipo interpretado, utilizando un intérprete en el momento de la ejecución, para traducir el código fuente implementado y enviarlo al sistema computacional para su ejecución, a diferencia de otros lenguajes como C++, que utilizan un compilador que traduce el lenguaje fuente en código de máquina previo a la ejecución. Debe mencionarse que la selección del lenguaje de programación Python corresponde con la facilidad de probar el código propuesto y es ampliamente usado para pruebas de concepto (PoC) en nuevos algoritmos propuestos.

El cálculo realizado en todos los escenarios analizados corresponde a un cálculo integral de todos los *lightpaths*. En un escenario real, este cálculo corresponde a un análisis de todos los *lightpaths* sobre la red propuesta, pero no es el único escenario posible para una

red comercial, ni tampoco el escenario más común; las condiciones para encontrar este cálculo total incluyen, pero no se limitan a:

- Primera ejecución del algoritmo sobre toda la red; después de actualizar todas las tablas de topología y de flujos compartidas por todos los nodos de la red, es necesario realizar un cálculo completo de todos los *lightpaths* en todos los nodos, para encontrar las rutas respectivas.
- Cambio de todos los *lightpaths* origen destino, requiriendo un nuevo cálculo con las nuevas matrices.
- Falla completa de un nodo de la red, bloqueando una o varias rutas y requiriendo un nuevo cálculo de una gran cantidad de rutas en la topología.

Estos escenarios mostrados, no corresponden a condiciones de operaciones normales de la red, donde el operador puede requerir retirar o agregar servicios previamente adquiridos por los clientes; en el primer escenario, únicamente se requiere actualizar las matrices de topología y flujos, sin realizar cambios sobre los *lightpaths* previamente instalados, garantizando la continuidad de los servicios de los demás clientes. En el segundo escenario, al agregar servicios se requiere actualizar las matrices de topología y de flujo en todos los nodos en la red, considerando las matrices previamente calculadas como las condiciones iniciales del sistema, y los nuevos *lightpaths* como el cálculo nuevo requerido; de esta manera, si se requieren instalar 10 *lightpaths* nuevos sobre una red con 40 *lightpaths* con rutas previamente calculadas, será necesario emplear un tiempo computacional equivalente al cálculo de 10 *lightpaths* sobre la red, y no se requiere realizar un cálculo integral de todo el sistema. En consecuencia, en condiciones de operación normal de la red, el tiempo computacional requerido para cambios administrativos comunes en los servicios se reduce considerablemente.

Finalmente, debe notarse que la implementación actual del algoritmo no se limita a la topología en anillo, pudiendo ingresar matrices de topologías diferentes, como una red mallada completa o parcial. Sin embargo, el requerimiento de comparar los resultados con los valores teóricos, hacen necesario reducir la complejidad del problema, para su análisis en programas de optimización.

4. CONCLUSIONES Y RECOMENDACIONES

4.1. CONCLUSIONES

1. El algoritmo Dijkstra-FirstFit desarrollado en el presente proyecto de grado muestra la factibilidad de implementar algoritmos de optimización con soluciones directas para resolver el problema de enrutamiento y asignación de longitudes de onda (RWA) en redes ópticas. El programa implementado y los resultados obtenidos muestran que es posible implementar algoritmos de optimización sencillos como Dijkstra para solucionar problemas de optimización matemática aplicados sobre redes de información reales.

2. El escenario 1 y el escenario 3 analizados en la sección 3, muestran un error de 6.82% al comparar los resultados teóricos obtenidos con el programa de optimización MiniZinc y los resultados obtenidos con el algoritmo propuesto. Adicionalmente, analizando los mismos escenarios se encuentra un tiempo promedio 4343,14% superior, para encontrar la solución óptima en comparación con el tiempo de solución del algoritmo Dijkstra-FirstFit. Los tiempos y costos obtenidos, muestran la factibilidad de implementar el algoritmo en un escenario real para redes con una cantidad pequeña de nodos, con cualquier número de longitudes de onda.

3. Al comparar los promedios de error relativo entre costos obtenidos en los escenarios 1 y 3 de la sección 3 con los resultados obtenidos para los escenarios 2 y 4, es evidente el incremento del error obtenido al comparar los costos totales del algoritmo Dijkstra-FirstFit con los valores teóricos; El error promedio en el costo obtenido con 4 nodos en la red, corresponde a 6.82%, en comparación con el error de 22.37% obtenido con 20 nodos. Como resultado, la implementación actual del algoritmo no se recomienda para una red con una gran cantidad de nodos. El incremento del número de nodos en la topología se traduce en un incremento en el número de enlaces en la red, que a su vez incrementa el número de costos unitarios que aportan en el resultado total de la optimización realizada; al incrementar el número de nodos y seleccionar una ruta diferente a la óptima, este resultado incrementa el costo obtenido con el algoritmo desarrollado, en proporción al número de enlaces que utiliza la ruta seleccionada para el *lightpath*, obteniendo como resultado un costo mayor al óptimo teórico.

4. Los resultados obtenidos en la sección 3, muestran un tiempo promedio de 136 segundos para la convergencia del programa de optimización MiniZinc, en comparación con el tiempo promedio de 0.05598 segundos para la solución del programa con el algoritmo Dijkstra-FirstFit. Estos resultados muestran la factibilidad del uso del algoritmo implementado en sistemas reales, con un tiempo de cálculo mucho menor a los cálculos realizados con programas de optimización.

5. En la sección 3 se puede evidenciar un incremento en el tiempo computacional requerido por el algoritmo Dijkstra-FirstFit, para encontrar la solución. Al comparar los escenarios 1 y 2 con los escenarios 3 y 4, se obtiene un incremento porcentual de 223,84% al incrementar desde 10 a 50 el número de lightpaths. Adicionalmente, la comparación de los escenarios 1 y 3 con los escenarios 2 y 4, evidencia un incremento de 703,25% en tiempo computacional al incrementar desde 4 a 20 el número de nodos de la red. Estos resultados, suponen un incremento en el tiempo computacional requerido para resolver el problema RWA, debido al incremento en el número de iteraciones requeridas por el algoritmo para finalizar el cálculo.

6. El algoritmo Dijsktra-FirstFit obtiene un error promedio de 14.59% al comparar los costos totales obtenidos en relación con los valores teóricos. Este valor de error puede reducirse, seleccionando lightpaths de la matriz de flujos aleatoriamente en cada iteración. Debe considerarse que esta nueva implementación no funcionaría en una red donde todos sus nodos realizan el cómputo independiente de rutas y longitudes de onda y requiere una administración centralizada de la red, para mantener concordancia entre las rutas y longitudes de onda en todos los nodos de la red.

4.2. RECOMENDACIONES

1. Los tiempos de cómputo del algoritmo propuesto pueden reducirse eliminando código implementado para permitir mostrar y analizar los resultados obtenidos; este código es necesario para mostrar los resultados requeridos para el análisis de resultados, pero puede reducirse o eliminarse en implementaciones reales del código, permitiendo disminuir el tiempo de cómputo del algoritmo total, en una red real, mejorando el tiempo total de convergencia del algoritmo.
2. El algoritmo propuesto puede realizar el cálculo sobre redes de cualquier topología, sin limitarse a la topología en anillo; los resultados obtenidos por el programa desarrollado pueden probarse con la implementación actual del algoritmo para topologías diferentes, considerando que se requiere el desarrollo de una solución matemática simple que pueda ingresarse a programas de optimización de uso comercial, para disponer de datos teóricos para comparar.
3. La solución propuesta puede probarse en escenarios con redes que dispongan previamente de lightpaths instalados, cambiando las condiciones iniciales del sistema. Esta implementación puede permite verificar los resultados obtenidos en las operaciones normales de administración de la red, con la adición o la remoción de *lightpaths* en la red; estas condiciones pueden incluir la adición de un nuevo requerimiento de un grupo de lightpaths entre un nodo origen y un nodo destino, remoción de un grupo de *lightpaths*, disminución de la capacidad requerida para una conexión previamente instalada, incremento del número de *lightpaths* de una conexión, entre otros.

5. REFERENCIAS BIBLIOGRÁFICAS

[1] B. Mukherjee, Optical WDM Networks, University of California: Springer Science+Business Media, Inc., 2006.

[2] A. E. Ozdaglar y D. P. Bertsekas, «Routing and Wavelength Assignment in Optical Networks,» *LIDS Technical Reports,* nº P-2535, p. 26, 2001.

[3] E. Iannone, Telecommunication Networks, 6000 Broken Sound Parkway NW, Suite 300: CRC Press Taylor & Francis Group, 2012.

[4] S. V. Kartalopoulos, INTRODUCTION TO DWDM TECHNOLOGY, Murray Hil: Wiley-IEEE Press, 2000.

[5] R. Ramaswami, K. N. Sivarajan y G. H. Sasaki, Optical Networks A Practical Perspective, 30 Corporate Drive, Suite 400, Burlington, MA 01803, USA : ELSEVIER Inc. All rights reserved., 2010.

[6] E. Iannone, «Telecommunication Networks,» CRC Press`, Boca Raton, FL, 2012.

[7] A. S. TANENBAUM, Redes de computadoras, 53519 Naucalpan de Juárez, Edo. de México: PEARSON Educación, 2003.

[8] M. S. Jímenez, «Diapositivas Comunicaciones Ópticas EPN,» Quito, 2015.

[9] I. C. H. Cisco Systems, Introduction to DWDM for Metropolitan Networks, San Jose, CA 95134-1706, USA, 2000.

[10] UIT-T SECTOR DE NORMALIZACIÓN DE LAS TELECOMUNICACIONES DE LA UIT, Recomendación G.692 SERIE G: SISTEMAS Y MEDIOS DE TRANSMISION, SISTEMAS Y REDES DIFITALES. Características de los medios de transmisión - Características de los componentes y los subsistemas ópticos, UIT, 1999.

[11] UIT-T SECTOR DE NORMALIZACIÓN DE LAS TELECOMUNICACIONES DE LA UIT, Recomendación UIT-T G.694.2 SERIE G: SISTEMAS Y MEDIOS DE TRANSMISIÓN, SISTEMAS Y REDES DIGITALES. Características de los medios de transmisión - Características de los componentes y los subsistemas ópticos, UIT, 2004.

[12] ITU-T TELECOMMUNICATION STANDARIZATION SECTOR OF ITU, Recommendation ITU-T G.671, UIT, 2012.

[13] H.-J. Thiele y M. Nebeling, Coarse Wavelength Division Multiplexing, Technologies and Applications, Boca Raton, FL.: CRC Press, 2007.

[14] J. A. Bondy y U. S. Murty, Graph Theory, Primera ed., Springer, 2008.

[15] J. Xu, Theory and Application of Graphs, Norwell, Massachusetts: Kluwer Academic Publishers, 2003.

[16] T. Nishizeki y N. Chiba, Planar Graphs: Theory and Algorithms, Amsterdam: ELSEVIER SCIENCE PUBLISHERS, 1988.

[17] D. Jungnickel, Graphs, Networks and Algorithms, Augsburg, Germany: Springer, 2004.

[18] H. Zang, J. P. Jue y B. Mukherjeet, «A Review of Routing and Wavelength Assignment Approaches for Wavelength-Routed Optical WDM Networks,» *`,* p. 25, 1999.

[19] H. T. Jongen, K. Meer y E. Triesch, OPTIMIZATION THEORY, Boston: Springer Science + Business Media, Inc., 2004.

[20] G. M. Durães, K. D. R. Assis, A. F. Santos, A. C. B. Soares y W. F. Giozza, «Which of the Shortest Paths Should we Choose? A Proposal of Routing in the All-Optical WDM Networks Design,» *ICTON,* nº We.P.24, p. 4, 2010.

[21] T. H. Cormen, C. E. Leiserson, R. L. Rivest y C. Stein, Introduction to Algorithms, Cambridge, Massachusetts. London, England: The MIT Press, 2001.

[22] A. E. Ozdaglar y D. P. Bertsekas, «Routing and Wavelength Assignment in Optical Networks,» *LIDS Technical Reports,* nº P-2535, p. 26, 2001.

[23] S. R. Otto y J. P. Denier, An Introduction to Programming and Numerical Methods in MATLAB, London: Springer, 2005.

[24] W. N. Venables, D. M. Smith y R Core Team, An Introduction to R, 2021.

[25] A. Boschetti y L. Massaron, Python Data Science Essentials, Birmingham: Packt Publishing Ltd., 2015.

[26] J. V. Guttag, Introduction to Computation and Programming Using Python, Cambridge, Massachusetts; London, England: Massachusetts Institute of Technology, 2013.

[27] G. v. Rossum y Python development team, The Python Language Reference, 2020.

[28] K. Marriott y P. J. Stuckey, A MiniZinc Tutorial.

[29] L. Carvajal, Metodología de la Investgación Científica. Curso general y aplicado, 28 ed., Santiago de Cali: U.S.C., 2006, p. 139.

[30] R. T. Koganti y D. Sidhu, «Analysis of Routing and Wavelength Assignment in Large WDM Networks,» *The 9th International Conference on Future Networks and Communications (FNC-2014),* 2014.

[31] W. Tomasi, «Sistemas de Comunicaciones Electrónicas,» Pearson Educación, México, 2003.

[32] A. K. Dutta, N. K. Dutta y M. Fujiwara, WDM Technologies, 525 B Street, Suite 1900, San Diego, California 92101-4495, USA: Elsevier Academic Press, 2004.

[33] ITU-T TELECOMMUNICATION STANDARDIZATION SECTOR OF ITU, Recommendations ITU-T G.694.1. SERIES G: TRANSMISSION SYSTEMS AND MEDIA, DIGITAL SYSTEMS AND NETWORKS. Transmission media and optical systems Characteristics - Characteristics of optical systems, ITU-T, 2012.

ANEXOS

Los anexos incluyen los datos tabulados obtenidos en las pruebas realizadas el algoritmo propuesto y de la optimización realizada con MiniZinc. Adicionalmente, se incluye el código fuente de los 3 archivos generados para la implementación del algoritmo propuesto, incluyendo el archivo de variables, el código del programa principal y el módulo de funciones.

ANEXO A. Resultados de las optimizaciones realizadas con el algoritmo Dijkstra-FirstFit y con MiniZinc.

ANEXO B. Código fuente del archivo de variables.

El código fuente debe ser almacenado con el nombre variables.py. Este archivo puede ser modificado, de acuerdo con el problema planteado.

ANEXO C. Código fuente del programa principal.

Este programa debe ser almacenado con el nombre main_.py.

ANEXO D. Código fuente del módulo de funciones.

Este código debe ser almacenado con el nombre DijkstraFirstFit.py.

ANEXO A

Resultados de las optimizaciones realizadas con el algoritmo Dijkstra-FirstFit y con MiniZinc.

Escenario	Número de prueba	Número de nodos	Número de lightpaths	Capacidad	Costo resultante óptimo (MiniZinc)	Costo resultante calculado (dijkstra_ff)	Tiempo del cálculo óptimo (MiniZinc) (seg)	Tiempo de cálculo real (dijkstra_ff) (seg)	Error relativo entre costos	Diferencia porcentual de tiempo de cómputo
1	1	4	10	10	207	207	0,211	0,0037688	0	5498,599024
1	2	4	10	10	136	136	0,214	0,0033252	0	6335,703116
1	3	4	10	10	94	103	0,213	0,003087	9,574468085	6799,902818
1	4	4	10	10	56	56	0,209	0,003023	0	6813,661925
1	5	4	10	10	40	40	0,213	0,0030186	0	6956,251242
1	6	4	10	5	112	112	0,324	0,0031439	0	10205,6713
1	7	4	10	5	88	119	0,211	0,0030177	35,22727273	6892,080061
1	8	4	10	5	63	63	0,212	0,0029377	0	7116,529938
1	9	4	10	5	71	71	0,214	0,0030669	0	6877,729955
1	10	4	10	5	79	79	0,202	0,0030169	0	6595,614704
2	1	20	10	10	511	654	0,307	0,0274486	27,98434442	1018,454129
2	2	20	10	10	239	302	0,252	0,0287042	26,35983264	777,9203043
2	3	20	10	10	672	672	0,244	0,0260451	0	836,8364875
2	4	20	10	10	454	581	0,249	0,0255137	27,97356828	875,9462563
2	5	20	10	10	336	416	0,243	0,0244481	23,80952381	893,9422695
2	6	20	10	5	279	279	0,241	0,0252587	0	854,1266969
2	7	20	10	5	336	480	0,247	0,0249707	42,85714286	889,1592947
2	8	20	10	5	478	497	0,235	0,0266085	3,974895397	783,1764286

Escenario	Número de prueba	Número de nodos	Número de lightpaths	Capacidad	Costo resultante óptimo (MiniZinc)	Costo resultante calculado (dijkstra_ff)	Tiempo del cálculo óptimo (MiniZinc) (seg)	Tiempo de cálculo real (dijkstra_ff) (seg)	Error relativo entre costos	Diferencia porcentual de tiempo de cómputo
2	9	20	10	5	536	720	0,244	0,0289292	34,32835821	743,4384636
2	10	20	10	5	440	816	0,242	0,030133	85,45454545	703,1062291
3	1	4	50	100	3896	4032	0,416	0,0242133	3,490759754	1618,064039
3	2	4	50	100	5296	5712	0,268	0,0247057	7,854984894	984,7699114
3	3	4	50	100	2952	3560	0,579	0,0256058	20,59620596	2161,206445
3	4	4	50	100	2136	2416	0,903	0,023127	13,10861423	3804,527176
3	5	4	50	100	3632	3728	0,239	0,0235427	2,643171806	915,176679
3	6	4	50	25	3336	3360	0,399	0,0178765	0,71942446	2131,980533
3	7	4	50	25	5104	5151	0,25	0,023041	0,920846395	985,0223515
3	8	4	50	25	3512	3896	0,337	0,0187268	10,93394077	1699,559989
3	9	4	50	25	3008	3846	0,243	0,0180235	27,85904255	1248,239798
3	10	4	50	25	2869	2965	0,233	0,0176177	3,346113628	1222,533588
4	1	20	50	100	10968	13016	107	0,1752299	18,67250182	60962,63828
4	2	20	50	100	8424	10360	4924	0,1706354	22,98195632	2885584,916
4	3	20	50	100	8162	11063	300,121	0,1694857	35,54275913	176977,4762
4	4	20	50	100	8699	9230	0,627	0,1740597	6,104149902	260,2212344
4	5	20	50	100	10600	13128	91	0,1865117	23,8490566	48690,50483
4	6	20	50	25	10104	12688	31,238	0,1707693	25,57403009	18192,51511
4	7	20	50	25	9065	9258	1,825	0,1466932	2,129067843	1144,093114
4	8	20	50	25	3610	3610	2,806	0,2257715	0	1142,849518
4	9	20	50	25	11624	14693	0,425	0,151971	26,40227116	179,6586191
4	10	20	50	25	8438	9576	12,268	0,1519518	13,4866082	7973,612817

ANEXO B

Código fuente del archivo de variables.

```
# coding=utf-8
import numpy as np
#Definicion de la variable Inf que para la biblioteca numpy representa infinito
Inf=np.inf

#omega = matriz de topologia/costo. Contiene todos los costos entre los enlaces de la red. Si el costo es infinito,
no existe conexion directa entre los dos enlaces.
#Debe entenderse las filas i y las columnas j como el costo de enviar información por el enlace n[i, j]. Cualquier
costo diferente a infinito (Inf) indica una conexión
#existente entre el nodo origen i y el nodo destino j. El valor infinito (Inf) indica una conexión inexistente
entre el nodo i y el nodo j. La diagonal de la Matriz
#siempre tiene el valor cero.
omega=np.array([[0, 0, Inf, 0],
                [0, 0, 0, Inf],
                [Inf, 0, 0, 0],
                [0, Inf, 0, 0]])

#Flujo = matriz que contiene los flujos que se requieren instalar en el sistema. Debe entenderse por flujos a la
cantidad de longitudes de onda (lambdas origen destino)
#que requieren ser instalados entre el nodo orígen i y el nodo destino j. El valor de cero indica que no se requiere
instalar longitudes de onda entre los nodos origen
#destino. La diagonal de la matriz es siempre cero, pues son valores de lightpaths que requieren ser instalados
entre un nodo como origen y el mismo destino.
flujo=np.array([[0, 1, 2, 1],
                [1, 0, 1, 1],
                [0, 1, 0, 0],
                [0, 2, 0, 0]])
```

```
#Capacidad del enlace, corresponde al número máximo de longitudes de onda (lightpaths) que pueden instalarse en
cada enlace.
capacidad = 5
```

ANEXO C

Código fuente del programa principal.

```
# coding=utf-8
#La primera línea es requerida para utilizar caracteres non-ASCII (utilizados en el lenguaje español) sin generar errores en la ejecución del código.
#Importar librerías requeridas para la ejecución del presente programa.
import numpy as np #La librería numpy permite realizar operaciones matriciales de manera sencilla y eficiente.
import DijkstraFirstFit as dj #Se importa el módulo que contiene las funciones desarrolladas para el presente proyecto y se asigna el alias dj para su ejecución.
import os #Librería requerida para la ejecución de comandos del sistema desde el programa durante su ejecución.
import time #Librería utilizada para calcular el tiempo de ejecución del programa.
from variables import * #Módulo utilizado para importar del archivo variables, el cual contiene las variables para ejecución del programa.
#Las variables importadas desde el archivo variables_026 corresponden a:
#omega: matriz de topología/costo. Contiene el costo de todos los enlaces de la red, lo cual permite inferir la topología del enlace.
#flujo: matriz de flujos. Contiene los flujos que se requiere instalar entre los nodos del enlace. El flujo se describe como el número de lightpaths requeridos entre el nodo origen o destino.
#capacidad: la capacidad del enlace corresponde al máximo número de longitudes de onda que pueden cursar un enlace.
#Para más información sobre todas las variables, referirse a la descripción del archivo variables.py

#El comando os.system(command) permite enviar un comando command del sistema hacia el terminal que ejecuta el programa.
#En este caso, el comando enviado es cls (clear screen), el cual limpia el terminal CMD de Windows, eliminando toda la información previamente impresa en pantalla.
os.system("cls")

#start_time almacena el tiempo de inicio de ejecución del programa. Permitirá realizar el cálculo del tiempo total que le toma al programa llegar a una respuesta.
start_time = time.clock()
```

```
#Permite copiar la matriz flujo en la variable flujoTemp. Esta nueva matriz se utilizará para realizar operaciones
y evita que se pierda la información original.
flujoTemp = flujo.copy()

#Se copia la matriz omega sobre las matrices omega1 y omega2, para realizar operaciones sobre estas nuevas matrices
y no perder la informacion contenida en la
#matriz de costos original. La matriz omega1 se utilizará para realizar todos los cálculos del sistema y la matriz
omega2 se utilizará para realizar el cálculo
#de costos al final de la ejecución del programa.
omega1 = omega.copy()
omega2 = omega.copy()

#m almacena el número de filas en la matriz omega y n almacena el número de columnas.
m, n = omega.shape

#lambdasAssigned almacena la longitud de onda que ha sido asignada para cada comunicacion. Se inicializa con una
matriz vacia
lambdasAssigned = np.empty((n, n), dtype = object )

#paths = matriz que contiene el resultado de todas las rutas escogidas para los enlaces
paths = np.zeros(n)

#X es la matriz que contiene la cantidad de lambdas (flujos) que cursan el enlace X[i][j]
x = np.zeros((n, n),dtype = float)

#Permite inicializar la variable pathSolution
pathSolution = False

#Lazo principal, recorre todos los enlaces n,m
for i in range(0,m):
    for j in range(0,n):
```

```
        #Determina cuales son los flujos se deben analizar, si la matriz flujo contiene en este elemento 0, no
existe un flujo
        #requerido entre origen destino, por lo cual no se analiza.
        while (flujoTemp[i][j] != 0):

            pathSolution = False
            while pathSolution == False:
                #Calculo del camino mas corto utilizando el algoritmo de Djikstra. El algoritmo requiere una matriz
de topologia y
                #un destino para el computo. El resultado obtenido es d con los valores de costos acumulados y path
que contiene
                #los caminos entre el origen i y todos los posibles destinos (los demas nodos de la red).
                d, path = dj.dijkstra(omega1, i) #Dijkstra con origen "i"

                #En caso de obtener un vector "d" con todos sus valores en infinito (valor de inicializacion de la
variable), se
                #determina que para este calculo no existe solucion factible, con las condiciones iniciales
definidas; por este
                #motivo, el programa termina con este mensaje.
                if np.all( d == Inf):

                    print '\033[91m' + 'No existe una solucion factible' + '\033[0m'
                    exit()

                #En base al resultado anterior, encontrar la ruta desde "i" hasta "j" utilizando la función
“dijkstraPath”
                pathD = dj.dijkstraPath(path, j)

                #La función ffLambdasAssigned permite realizar la asignación de longitudes de onda y almacenarlos
en
                #la matriz “lambdasAssigned”Temp y en el vector “lambdasAssigned”PathD.
```

```
                lambdasAssignedTemp,    lambdasAssignedPathD    =    dj.ffLambdasAssigned(    pathD,    capacidad,
lambdasAssigned )
                #Permite copiar el resultado de la matriz “lambdasAssignedTemp a la matriz “lambdasAssigned”
                lambdasAssigned = lambdasAssignedTemp.copy()

                x = dj.flujoIncremental(x, pathD) #Incremeto del flujo (# de lambdas) que cursan el enlace i, j
                #La función “costoIncremental” permite incrementar el valor del costo contenido en las matrices
omega1 y omega2
                #de acuerdo a la función de incremento de costo definida.
                omega1 = dj.costoIncremental(omega1, x)
                omega2 = dj.costoIncremental(omega2, x)

                #Si la cantidad de longitudes de onda que atraviesan un enlace llego a su capacidad maxima, se
restringe
                #la seleccion de este enlace para los siguientes calculos, incrementando su costo a infinito
                if (np.any (x == capacidad)):
                    omega1[x == capacidad] = Inf

                #Resto 1 al numero de longitudes de onda que se requieren instalar sobre este enlace, y se continua
con
                #el calculo de la siguiente lambda
                flujoTemp[i][j] = flujoTemp[i][j]-1

                #paths almacena todos los caminos escogidos para cada calculo realizado. La sentencia if permite
determinar si
                #la matriz paths está vacía o si tiene información almacenada. En caso de encontrar que la matriz
no se
                #encuentra vacía (valor verdadero), se procede a usar la función vstack para apilar los resultados.
Caso
                #contrario, se inicia el vector con la primera ruta en la sentencia else.
                if (np.any(paths != 0)):
```

```
                    paths = np.vstack([paths, pathD])
                    lambdasAssignedVector = np.vstack([lambdasAssignedVector, lambdasAssignedPathD])
                else:

                    paths = pathD
                    lambdasAssignedVector = [lambdasAssignedPathD]

                pathSolution = True
#Permite almacenar en la variable stop_time la hora actual, para calcular el tiempo total del cálculo.
stop_time = time.clock()
#Las funciones print a continuación permiten imprimir los datos del problema y los resultados obtenidos.
print '\n======================DATOS==========================='
print "\nMatriz Omega = "
print omega
print "\nMatriz Flujo = "
print flujo
print "\nLa capacidad de los enlaces es: " + str(capacidad)
print "\n\n"
print '======================RESULTADOS=========================='
print "\nMatriz x = "
print x
print '\nMatriz paths= '
print paths
print "\nMatriz lambdasAssignedVector"
print lambdasAssignedVector
print '\nMatriz omega1= '
print omega1
print '\nMatriz omega2= '
print omega2
print "\nMatriz de flujos resultantes = "
print flujoTemp
print '\n Matriz lambdasAssigned'
```

```
print lambdasAssigned
#La función “dijkstraCostF” permite calcular el costo total de acuerdo con los resultados de costo
#almacenados en la matriz omega2
costoT = dj.dijkstraCostF(omega2)
#Las siguientes funciones permiten imprimir el resultado del costo total y el tiempo total
#del cálculo realizado por el algoritmo.
print '\n Costo total = ',costoT
print("\n--- Tiempo total del calculo: %s seconds ---" % (stop_time - start_time))
```

ANEXO D

Código fuente del módulo de funciones.

```
# coding=utf-8
#La primera línea es requerida para utilizar caracteres non-ASCII (utilizados en el lenguaje español) sin generar errores en la ejecución del código.
#Importacion de los framework con los que se trabajara
import numpy as np
import time
import os

#Definicion de la constante Inf que representara el infinito que puede comprender el framework numpy
Inf=np.inf

#Matriz que contiene los costos de los enlaces. En la diagonal es cero pues representa el costo de transportar informacion de un nodo hacia si mismo
#En el caso de que no exista conexion fisica entre dos nodos, en su correpondiente lugar se colocara Inf

def dijkstra(omega, nodo):

    #Se requiere los valores de m y n para determinar el tamano de los vectores
    m,n=omega.shape

    #El vector d almacenara los valores de los costos acumulados. El vector se inicializa con todos sus valores en Infinito
    d=np.empty(n)
    d.fill(Inf)
```

```
    #El vector known almacenara los nodos que fueron seleccionados como "fijos" en el algoritmo. Para el nodo que
se analizara este valor es de 1
    known=np.zeros(n)
    known[nodo]=1

    #Vector que almacenara el path desde el nodo analizado hasta el nodo destino
    path = np.empty(n)
    path.fill(-1)

    #El algoritmo utiliza un nodo como pivot en cada iteracion para analizar los nodos restantes que no fueron
seleccionados como "fijos"
    #Para la primera iteracion se selecciona al nodo que se analiza como el nodo pivot
    pivot=nodo
    d[nodo]=0

    #Lazo principal que realiza la optimizacion. La sentencia np.any(d==Inf) determina si dentro del vector d
existe algun elemento con su
    #valor igual a Inf, ya que se detendran las iteraciones cuando no exista ningun elemento igual a infinito
    while (np.any(d==Inf)):
        for i in range (0,n):

            #Permite determinar si el nodo analizado es seleccionado como "fijo", en cuyo caso no se realizara un
cambio sobre el vector d
            if (known[i]!=1):
                #Determina si se cumple la condicion d[i]<=(d[pivot]+omega[pivot][i]) (Revisar algoritmo de
Dijkstra)
                if d[i]<=(d[pivot]+omega[pivot][i]):
                    pass
                else:
                    #En caso de que sea falso, se actualiza el valor del vector d con el nuevo valor de costo
correspondiente y el valor del
                    #vector path con el nuevo path hacia ese nodo
```

```
                    d[i]=d[pivot]+omega[pivot][i]
                    path[i] = pivot


        temp = d.copy() #Es necesario copiar pues caso contrario ambas matrices o vectores estan enlazados
        #Determina si el minimo valor almanenado en el vector temp (que contiene la informacion del vector d), esta
determinado como "fijo"
        #en cuyo caso se modifica la informacion de temp correspondiente a este nodo a infinito y se recalcula
hasta obtener el valor minimo
        #de los nodos que no son fijos
        while (known[np.argmin(temp)]==1):
            temp[np.argmin(temp)]=Inf
            if np.all( temp == Inf):
                break


        #Al determinar el minimo, este nodo pasa a ser "fijo" y sera el nuevo pivot para la siguiente iteracion
        if np.all( temp == Inf):
            d = temp.copy()
            return d, path
            break
        known[np.argmin(temp)]=1
        pivot=np.argmin(temp)

    return d, path

#Esta funcion calcula el camino origen - destino; en el vector path, el valor -1 idenfifica al nodo origen para el
calculo de dijskstra, y nodoD contiene el indice del nodo
#destino. El vector “pathD” contiene el camino entre origen destino, identificado por el indice de los nodos origen
destino ordenados. El camino para llegar al destino termina
#cuando se encuentra el valor -2 en el vector, pues son valores no posibles. P.Ej.: “pathD” = [1, 3, 5, 7, -2, -2,
-2, -2], es el resultado de una red con 8 nodos, con el calculo
```

```
#del camino entre 1 -> 7, que atravieza los nodos 3 y 5 en ese orden
def dijkstraPath(path, nodoD):
    n = path.shape[0] #Tamano de la matriz path (primer valor)
    pathD=np.empty(n+1) #Crea la matriz “pathD”, de tamano n+1 y sin valores de inicializacion
    pathD.fill(-2) #Llena la matriz “pathD” con los valores -2, debido a que no es un valor posible dentro de los
posibles resultados (en los cuales -1 es posible)
    #x = np.where(path == -1) #x contiene el indice donde el vector path tiene el valor de -1; este valor se
encuentra en el nodo origen.
    pathD[0] = nodoD #El primer valor del vector resultado es el nodo destino del camino requerido
    for i in range (1, n+1): #Rango de analisis, que evita el primer nodo
        if path[nodoD] != -1: #Cuando se cumpla que path[nodoD] = -1, se ha llegado al nodo origen, por lo cual
debe terminarse el lazo, lo cual se contiene en el else
            pathD [i] = path [nodoD] #ingresa en “pathD” [i], el valor del siguiente salto.
            nodoD = path [nodoD] #Reemplaza el valor de nodoD con el nuevo nodo, para reconstruir el path
            nodoD = nodoD.astype(int) #Cambia el tipo de dato de nodoD por entero
        else:
            pathD [i] = path [nodoD] #Ingresa el valor de -1 en el ultimo salto
            nodoD = path [nodoD] #Reemplaza el valor de nodoD con el nuevo nodo, para reconstruir el path
            nodoD = nodoD.astype(int) #Cambia el tipo de dato de nodoD por entero
            break #Salida del lazo
    pathD = pathD[::-1] #Invierte el orden de los valores del vector “pathD”, para colocar a la izquierda al nodo
origen (indice [0]) y al extremo derecho al nodo destino

    while pathD[n] != -1: #Este lazo realiza un desplazamiento en el vector “pathD”, para obtener en “pathD”[0] el
valor del nodo origen y a la derecha el valor del nodo destino
        pathD = np.roll(pathD, 1)

    return pathD

#La función “dijkstraCostF” permite calcular el costo total del resultado calculado. Se ingresa como dato la matriz
omega1.
def dijkstraCostF(omega1):
```

```
    n,m = omega1.shape #permite almacenar las dimensiones n y m de la matriz omega1.
    costoT = 0 #Inicia la variable "costoT" con el valor de cero. Esta variable almacena el resultado de costo
total.
    #Los lazos for principales se muestran en las siguientes dos líneas.
    for i in range (0,n):
        for j in range (0,m):
            if (omega1[i][j] != Inf): #La sentencia if permite determinar si el valor almacenado en la matriz omega
es diferente de infinito, para proceder con la suma.
                costoT = costoT + omega1[i][j] #Permite incrementar el valor de la variable "costoT".
    return costoT #Permite devolver el valor de la variable "costoT" como resultado de la función.

#La función "flujoIncremental" permite incrementar los valores de la matriz x con un incremento de 1 en el número
de longitudes de onda que atraviesan el enlace correspondiente.
def flujoIncremental(x, pathD):
    pathD = pathD.astype(np.int) #Conversion de los valores de la matriz "pathD" al tipo entero int.
    n = pathD.shape #Almacena en la variable n el tamano del vector "pathD"
    for i in range (0, n[0] - 1): #Lazo de incremento del valor del flujo en cada enlace
        if pathD[i+1] != -2 and pathD[i+1] != -1: #Sentencia de decisión if que permite determinar el final del
cálculo de la función.
            x[pathD[i]][pathD[i+1]] += 1 #Incremento de 1 longitud de onda en cada enlace.
        else:
            break #Permite finalizar el lazo for
    return x #Devuelve el valor almacenado en x como resultado de la función.

#La función "costoIncremental" permite realizar el cálculo del costo de cada enlace considerando el número de
longitudes de onda actuales. La función de costo se calcula con
#la función (2x-1)^2
def costoIncremental(omega1, x):
    #Los lazos for principales mostrados en las siguientes dos líneas permiten recorrer toda la matriz de flujos y
toda la matriz omega para realizar un nuevo cálculo.
    for i in range (0, omega1.shape[0]):
        for j in range (0, omega1.shape[0]):
```

```
            if x[i][j] != 0 : #Sentencia condicional para determinar si el valor del número de longitudes de onda es cero, considerando la función de costo.
                omega1[i][j] = (2*x[i][j] - 1) ** 2 #Cálculo de la función de costo utilizando la fórmula (2x-1)^2
            else:
                pass #Si el número de longitudes de onda en un enlace es cero, no se realizan cambios sobre la matriz omega en este enlace.
    return omega1 #Permite regresar la matriz omega1 como resultado de la función.


#La función “ffLambdasAssigned” utiliza el algoritmo first fit para la asignación de longitudes de onda en el tramo de la ruta.
def ffLambdasAssigned( pathD”, capacidad, lambdasAssigned ):
    [u] = pathD.shape #Permite obtener el tamaño del vector “pathD”.
    lambdasAssignedTemp = lambdasAssigned.copy() #Permite copiar la matriz “lambdasAssigned” a una variable temporal para realizar operaciones sobre ella.
    lambdasAssignedPathD = np.empty((u), dtype = object ) #Crea un vector “lambdasAssignedPathD con sus variables del tipo caracter y todos sus valores vacíos.
    lambdasAssignedPathD.fill('-1') #Permite llenar todos los valores del vector “lambdasAssignedPathD con el valor -1.

    #El lazo principal permite recorrer todos los elementos de la matriz “lambdasAssignedTemp.
    for i in range(0, u - 1):
        for j in range(1, capacidad + 1):
            test = '_' + str(j) + '_' #La variable test almacena el valor a analizar, con los símbolos guión bajo al inicio y final del número.

            if pathD[i + 1] != -2 and  pathD[i + 1] != -1: #La sentencia if permite determinar el final del vector “pathD” delimitada por los valores -1 o -2

                #La variable a almacena la posición del valor almacenado en la variable temp. En caso negativo, el valor almacenado en a es -1; caso contrario, contiene
                #El valor de la posición correspondiente.
```

```
                a = int(np.char.find(str(lambdasAssignedTemp[int(pathD[i])][int(pathD[i+1])]), test))
                if a == -1: #La sentencia if permite determinar si no se ha encontrado la longitud de onda almacenada
en la posición correspondiente de la matriz.
                    if  lambdasAssignedTemp[int(pathD[i])][int(pathD[i + 1])] == None: #Permite determinar si la
posición respectiva se encuentra vacía.
                        lambdasAssignedTemp[int(pathD[i])][int(pathD[i + 1])] = test #Almacena la longitud de onda
almacenada en test en la posición del enlace respectiva.

                        lambdasAssignedPathD[i] = test #Permite almacenar en el vector lambdasAssignedPathD la
longitud de onda correspondiente.
                    else:
                        #Permite añadir la longitud de onda almacenada en la variable test a las longitudes de onda
actualmente almacenadas en el enlace correspondiente.
                        lambdasAssignedTemp[int(pathD[i])][int(pathD[i            +            1])]            =
lambdasAssignedTemp[int(pathD[i])][int(pathD[i + 1])] + test

                        lambdasAssignedPathD[i] = test #Permite almacenar en el vector lambdasAssignedPathD la
longitud de onda correspondiente.
                    break #Permite salir del lazo actual
    return lambdasAssignedTemp, lambdasAssignedPathD
```

ORDEN DE EMPASTADO