ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN
UNIVERSIDAD POLITÉCNICA DE CARTAGENA

Proyecto Fin de Carrera

"Diseño e implementación de la herramienta didáctica MatPlaNet Tool, para la planificación de redes WAN"

AUTOR: José Antonio Carrillo Conesa
DIRECTOR: Pablo Pavón Mariño

Diciembre / 04

Autor	José Antonio Carrillo Conesa
E-mail del Autor	joseantoniocarrilloconesa@hotmail.com
Director(es)	Pablo Pavón Mariño
E-mail del Director	pablo.pavon@upct.es
Título del PFC	"Diseño e implementación de la herramienta didáctica MatPlaNet Tool, para la planificación de redes WAN"
Descriptores	Planificación de redes WAN. Herramientas de planificación.

Resumen

El proyecto aquí presentado se centra en la planificación, diseño, y análisis topológico de redes WAN, mediante algoritmos normalmente centrados en teoría de Grafos, que dan solución a los problemas más típicos de un planificador de red. Los parámetros a optimizar, incluyen coste de la red, utilización, y retardo, así como fiabilidad de red. El entorno de trabajo para la programación y simulación para este proyecto es MATLAB, que proporciona las ventajas de modularidad y flexibilidad necesarias para la realización de este Proyecto, así como sus aplicaciones didácticas. Los logros obtenidos por el proyecto asimismo pretenden constituir un *toolbox* de planificación de red para MATLAB.

Titulación	Ingeniería de Telecomunicación
Intensificación	Planificación y gestión de Telecomunicaciones
Departamento	Tecnologías de la Información y las Comunicaciones
Fecha de Presentación	Diciembre– 2004

Índice

CAPÍTULO 1 INTRODUCCIÓN 7

1.1 Planificación de redes WAN 7

1.1.1 Introducción 7

1.1.2 Problemas de optimización en redes 9

1.1.2.1 Problemas de optimización o programación matemática 9

1.1.2.2 Variables asociadas a los problemas de optimización en redes 9

1.1.2.3 Clasificación clásica de los problemas de optimización de redes 11

1.1.3 Descripción del trabajo típico de un planificador 12

1.2 Aproximación a la teoría de grafos 13

1.3 Herramientas de planificación WAN 16

1.4 Objetivos del proyecto 18

CAPÍTULO 2 TEORÍA DE PLANIFICACIÓN EN REDES WAN 21

2.1 Problema del MST y SPT 21

2.1.1 Definición del problema MST 21

2.1.2 Análisis MST y definición del problema SPT 23

2.1.3 Solución intermedia: Prim-Dijkstra 24

2.2 Problema MST con capacidades (CMST) 25

2.2.1 Definición del problema 25

2.2.2 Esau-Williams 26

2.3 Problema de multispeed CMST 30

2.3.1 Definición problema 30

2.3.2 MSLA 31

2.4 Planificación de redes *Mesh* 34

2.4.1 Definición del problema 34

2.4.2 Estrategia MENTOR 35

2.4.2.1 Clustering: Selección de nodos para el *Backbone* 36

2.4.2.2 Creación de la topología inicial de la red de malla 38

2.4.2.3 Incorporación de enlaces directos para formar la red de malla 40

2.4.2.4 Topología de acceso al *backbone* 43

2.4.2.5 Esquema: conjunto de algoritmos MENTOR 44

2.4.2.6 Problema del encaminamiento en redes mesh 45

2.4.3 Propuesta de adaptación *multispeed* de AMENTOR 49

2.5 Generación de tráfico y tarifas. 51

2.5.1 Generadores de Tráfico 52

2.5.1.1 Generador Uniforme 52
2.5.1.2 Generador aleatorio simple 52
2.5.1.3 Generador en función de la población y la distancia 52
2.5.1.4 Añadimos *offset* y factor de escala. 53
2.5.1.5 Añadimos tráfico asimétrico 54
2.5.1.6 Añadimos generación de variaciones aleatorias 54
2.5.2 Normalizadores de tráfico 55
2.5.2.1 Normalizador total 55
2.5.2.2 Normalizador por filas 56
2.5.2.3 Normalizador por columnas 56
2.5.2.4 Normalizador por filas y columnas 56
2.5.3 Generadores de Tarifas 57
2.5.3.1 Generador Lineal 58
2.5.3.2 Generador Lineal a Trozos 58
2.5.3.3 Generador Constante a Trozos 59
2.5.4 Linealización de tablas de tarifas. 59
2.6 Evaluación de las soluciones 60
CAPÍTULO 3 DESCRIPCIÓN DE LA HERRAMIENTA 63
3.1 Preliminares 63
3.2 Bloques 65
3.2.1 Bloque generadores 65
3.2.2 Bloque algoritmos de planificación 69
3.2.2.1 Algoritmos del problema MST y SPT 69
3.2.2.2 Algoritmos del *capacitated* MST y *multispeed* CMST 71
3.2.2.3 Algoritmos MENTOR 72
3.2.3 Bloque análisis y visualización 76
3.2.4 Bloque Funciones generales 77
3.2.4.1 Herramientas de Chequeo y búsqueda en árboles 78
3.2.4.2 Herramientas de encaminamiento 80
3.2.4.3 Herramientas de forma 81
CAPÍTULO 4 RESULTADOS 83
4.1 Presentación 83
4.2 Contenidos 83
4.2.1 Generación de topologías, tráfico y normalización 83
4.2.1.1 Generación de una tabla de tráficos 83
4.2.1.2 Normalizaciones 85
4.2.1.3 Generación de tarifarios 88

4.2.1.4 Linealización ... 93
4.2.2 Algoritmos de optimización ... 95
4.2.2.1 MST y SPT ... 95
4.2.2.2 CMST y multispeed ... 101
4.2.2.3 MENTOR ... 108
CAPÍTULO 5 CONCLUSIONES Y LÍNEAS FUTURAS ... 115
APÉNDICE ... 119
A. Algoritmos de exploración de grafos: DFS ... 119
B. Algoritmo de búsqueda de ciclos en un grafo ... 120
C. Fiabilidad de una red ... 121
C.1. Algoritmo CheckConnected ... 121
C.2. Algoritmo CheckBiconnected ... 122
D. Búsqueda de subcomponentes ... 122
D.1. Conexos ... 122
D.2. Biconexos ... 122
BIBLIOGRAFÍA ... 125

Capítulo 1
Introducción

1.1 Planificación de redes WAN

1.1.1 Introducción

Con planificación de redes WAN nos referimos al conjunto de tareas que desembocan en el diseño de redes entre distintas localizaciones en un área extensa genérica. Esto implica tomar decisiones respecto a qué enlaces conectar, con qué características, y qué dispositivos mantener para conseguir una red de acuerdo a unas expectativas y restricciones impuestas.

Estas decisiones son múltiples y nada evidentes, y los diseños que de ellas se derivan pueden variar enormemente en función del proceso seguido para construirlos. Se habla a veces de que no existen claros ganadores, sino sólo claros perdedores. Por lo tanto, aunque la complejidad y extensión del espacio de soluciones sea demasiado grande como para pretender lograr el diseño perfecto, siguiendo las adecuadas reglas se evitará que caigamos en diseños mediocres o incorrectos.

El problema que nos aguarda implica pues adentrarse en un mundo muy extenso, por lo cual se intentará aquí mostrar unas nociones básicas para iniciarse en él. El punto de partida, la cuestión de fondo de cualquier problema de planificación de redes es "¿cómo puedo proveer a varios sitios de las comunicaciones que requieren de la forma más eficiente?"

Obviamente necesitamos datos acerca de las redes que tenemos que diseñar. Estos datos pueden ser de diversa índole, pero los básicos serían: la topología, el tráfico, el encaminamiento, los costes, y los requerimientos.

- La topología: La distribución de nodos y enlaces de la red.
- El tráfico: los flujos de tráfico que hay entre los nodos.
- El encaminamiento: las rutas que sigue cada flujo de tráfico entre dos nodos.
- Los costes: todos los posibles costes de construcción o alquiler de líneas, dispositivos de encaminamiento, y cualquier otro material relevante que puedan participar en la red.
- Requerimientos: de cualquier índole, p.e. de bajo coste, de rendimiento, de retardo, de fiabilidad.

De estos, la topología y los costes son datos fijos y relativamente fáciles de medir. El tráfico es el dato de donde obtendremos la mayor parte de la información útil para crear la red, un parámetro a estudiar con detenimiento. El encaminamiento es una función del tráfico respecto a los mecanismos de los dispositivos enrutadores de la red. Los requerimientos es el fin hacia el que debe de tender el diseño de la red.

La primera cuestión que hay que hacerse es respecto al tipo de tráfico que va a llevar la red. Hay dos tipos básicos: tráfico de voz y tráfico de datos.

Cuando tratamos con diseños con tráfico de voz , el objetivo es simple: minimizar el coste de la red mientras se mantienen las probabilidades de bloqueo, o el número de llamadas que encuentran señal ocupada, por debajo de un nivel específico.

La forma de enfrentar estos problemas suele ser sencilla, y pasa por informarse de las soluciones de que se pueden disponer (p.e. para la conexión telefónica entre dos oficinas, podríamos pensar en la red telefónica conmutada común, en líneas dedicadas, en la instalación de centralitas internas en cada una de las oficinas (PBX), etc...) y a qué precio saldría. Con esos datos, las decisiones pasan por elegir una solución o mezcla de ellas que aporte la calidad de servicio mínima requerida, al menor precio.

El principio de diseño de este tipo de redes sería tender a tener el mayor número de componentes bien utilizados posible, ya que aquí todo porcentaje de utilización bajo indica un desperdicio de capacidad, y por lo tanto un coste no óptimo. Gracias a los datos respecto al tráfico usual de llamadas, se pueden calcular las necesidades de una red que sea capaz de dar servicio a todos los clientes en la hora punta. Se calcula la conveniencia de contratar servicios o alquilar líneas respecto a si la media de llamadas efectuadas es alta (saldría más barato alquilar líneas) o si hay muy poco tráfico (bastaría con pagar lo que se consume). Se diseñan varias soluciones con distintos parámetros (una con el menor coste, otra con un coste un poco más alto pero mucho menor probabilidad de bloqueo, etc), y se elige el que más se ajuste a lo pedido.

Sin embargo, cuando tratamos con problemas de tráfico de datos, el asunto se complica. Para empezar, el tráfico de datos tiene unas características bastante diferentes al tráfico de voz:

Tráfico de Voz	Tráfico de Datos
Ancho de Banda fijo	Ancho de Banda variable
Duración de llamadas corta	Duración de llamadas larga
Una conexión por persona	Varias conexiones por persona
Sensibilidad extrema al retardo	Sensibilidad variable
Tolera ciertas pérdidas	Varía
Probabilidad de bloqueo	Retardo de los datos

Tabla 1-1. Comparación de tráfico de voz y datos

La diferencia más importante es la que aparece en último lugar en la tabla. En redes de voz, cuando se intenta efectuar una llamada y todas las líneas están ocupadas, no es posible realizarla, es decir, se bloquea o se pierde. En cambio en las redes de datos, los nodos disponen de colas en las que almacenan los paquetes que no pueden procesar en ese momento, retrasando la respuesta pero no perdiéndola como en las redes anteriores.

Por esta razón, en una red de voz, los enlaces con alta utilización son atractivos, y efectivos en cuanto al coste, porque están explotando el ancho de banda disponible en la más completa medida, y se está dando a las líneas un alto grado de servicio. Sin embargo, en una red de datos, el uso de enlaces con utilización muy alta es un desastre en tanto en cuanto el tráfico que usa esos enlaces sufre un terrible retardo, en detrimento de la calidad de servicio.

Las redes de datos guardan un equilibrio estrecho entre sus parámetros, menor retardo no significa mejor utilización, ni mejor utilización mejor coste. Una pequeña variación en un parámetro puede dar un gran grado de libertad para con los demás. Es este último tipo de redes en el cual nos vamos a centrar aquí. Por lo tanto, en adelante veremos la interesante parte del diseño y planificación de redes WAN que abarcan los <u>problemas de optimización de redes de datos</u>.

1.1.2 Problemas de optimización en redes

1.1.2.1 Problemas de optimización o programación matemática

El problema genérico de la programación matemática es encontrar para unas variables (x_1,x_2,... ,x_n) la solución óptima que maximiza o minimiza f(x_1,x_2,... ,x_n), sujeto a restricciones del tipo:

$$\left\{\begin{matrix} g_1(x_1, x_2, \ldots, x_n) \geq 0 \\ \vdots \\ g_n(x_1, x_2, \ldots, x_n) \geq 0 \end{matrix}\right\}$$

$$\left\{\begin{matrix} h_1(x_1, x_2, \ldots, x_n) = 0 \\ \vdots \\ h_n(x_1, x_2, \ldots, x_n) = 0 \end{matrix}\right\}$$

Ecuación 1-a. Restricciones para el problema clásico de optimización.

Existen varias posibles clasificaciones de este tipo de problemas:

- Clasificación 1:
  - Programación Finita: $x \in \Re^n$, con n finito.
  - Programación dinámica: si x tiene infinitas dimensiones.
- Clasificación 2:
  - Programación estocástica: los datos del problema son funciones de variable aleatoria.
  - Programación no-estocástica: determinista, el espacio de soluciones es más restringido.
- Clasificación 3:
  - Programación en la recta real: Cuando $(x_1, x_2, \ldots, x_n) \in \Re$.
  - Programación entera pura: Cuando $(x_1, x_2, \ldots, x_n)$ están en un rango discreto de valores.
  - Programación entera-mixta: Cuando algunas variables son reales y otras están en un rango discreto.

1.1.2.2 Variables asociadas a los problemas de optimización en redes

Estas serán las variables que utilicemos de ahora en adelante a lo largo de todo el proyecto. Tenemos variables asociadas a los distintos aspectos básicos del diseño de redes, es decir, variables asociadas a la topología, al tráfico, al encaminamiento, y al retardo.

- Variables de la topología
  - N ≡ Número de nodos a interconectar.

- M≡ Número de enlaces entre los nodos.
- C_{ij}≡ Capacidad de un enlace desde cuyos nodos terminales son el i y el j.
- L≡ Longitud media de paquete (mensaje). $L = \frac{1}{\mu}$ (bits)
- Tarifario≡ Matriz de costes de la forma Coste(i,j,C_{ij}): Coste de un enlace que va de i a j cuya capacidad es C_{ij}.

o <u>Variables asociadas al tráfico</u>

- TO≡ Matriz de tráfico ofrecido (medido en mensajes por segundo) NxN entre cada par de nodos, con componentes de la forma γ_{ij} , el tráfico generado por el nodo i que se dirige a j.
- $TO_{total} = \sum_i \sum_j \gamma_{ij}$
- Caudal≡ Es la cantidad de bps (bits por segundo) que se pueden introducir a la red en un punto de terminación de red, es decir, el ritmo al cual la red acepta información. También se le llama *cadencia*, *throughput*, o *tráfico* en general.

o <u>Variables de encaminamiento</u>

- $\prod_{ij}$≡ Camino, conjunto de nodos que se atraviesan para ir del nodo i al nodo j.
- TC≡ Matriz de tráfico cursado por cada enlace, con componentes de la forma

$$\lambda_{ij} = \sum_{kl} \gamma_{kl}$$

, donde *kl* son los pares de nodos cuyo camino $\prod_{kl}$ contiene el enlace del nodo i al nodo j (los flujos que pasan por ese enlace). Cada λ_{ij} es el flujo de mensajes que va por un enlace de i a j.

- $Hops_{ij}$≡ número de saltos del nodo i al nodo j≡ $\#\prod_{ij} - 1$.

o <u>Variables asociadas al retardo</u>

- Z_{ij}≡ retardo que sufre un paquete que va desde el nodo i al nodo j, es decir, tiempo medio que tarda un paquete en viajar desde i a j.
- T_{ij}≡ retardo medio que sufre un paquete en un enlace cuyos nodos terminales son el i y el j:

para M/M/1[1],
$$T_{ij} = \frac{L}{C_i - \lambda_{ij} \cdot L}$$

---

[1] En este documento se considerará válida la *hipótesis de Kleinrock*. Según esto, cada canal (enlace) corresponderá a un servidor aislado, con un patrón de llegadas poissoniano de media λ_{ij}, y un tiempo de servicio, distribuido exponencialmente, de media $\frac{1}{\mu \cdot C_{ij}}$.

- Retardo medio de la red, $\overline{T} = \sum_{ij} \frac{TO_{ij}}{TO_{total}} Z_{ij} = \sum_{ij} \frac{\lambda_{ij}}{TC_{total}} T_{ij}$

Además de estas variables, conviene definir ciertos conceptos:

- ***Backbone***: Es un subconjunto de nodos así como los enlaces que existen entre ellos, que forman la "columna vertebral" de una red. Es decir, es la parte troncal de una red más amplia, formada por los nodos más importantes y los enlaces de más capacidad. Es la parte que soporta el peso de la red, y por la que fluirá normalmente más tráfico.
- **Red de acceso:** Es el otro subconjunto de la red, el que se encarga de proporcionar los enlaces necesarios para que todos los nodos puedan acceder al *backbone.*

1.1.2.3 Clasificación clásica de los problemas de optimización de redes

o Asignación de capacidades (*CA*)

En este tipo de problemas se tienen como datos la matriz de tráfico (ofrecido), y los costes en función de las capacidades posibles, y consiste en encontrar la C_{ij} de cada enlace, teniendo por objetivo minimizar el retardo medio de la red, con la restricción de que la suma total de costes no pase de cierto umbral.

o Asignación de flujos (*FA)*

En este tipo de problemas se tienen como datos la topología de la red, y los costes en función de las capacidades posibles, y consiste en encontrar el tráfico cursado de cada enlace, teniendo por objetivo minimizar el retardo medio de la red.

o Asignación de capacidades y flujos (*CFA*)

En este tipo de problemas se tienen como datos la topología de la red, la matriz de tráfico y los costes en función de las capacidades posibles, y consiste en encontrar la C_{ij} de cada enlace, teniendo por objetivo minimizar el retardo medio de la red, y con la restricción de que la suma total de costes no pase de cierto umbral.

o Asignación de topología, capacidades y flujos (*TCFA*)

En este tipo de problemas se tienen como datos únicamente la matriz de tráfico y los costes en función de las capacidades posibles, y consiste en encontrar la topología que sigue la red, el encaminamiento, y la C_{ij} de cada enlace, teniendo por objetivo minimizar el retardo medio de la red, con la restricción de que la suma total de costes no pase de cierto umbral, o bien lo contrario, minimizar la suma de costes, manteniendo un umbral de retardo medio de la red máximo.

El problema más completo e interesante es el TCFA, y será del cual nos ocuparemos. Las características de este problema es que siempre se puede expresar como un problema de optimización, pero que no se puede usar el análisis clásico (ecuación 1-a), y aplicar el método de los multiplicadores de Lagrange para convertirlo en un problema sin restricciones, porque esto requeriría:

- Que la función objetivo sea derivable (continua, lineal).

- Que las restricciones sean derivables.
- Que las variables sean reales.

Ninguna de estas condiciones son característica en estos problemas. Por lo tanto hay que buscar otro método.

No existe un proceso que dé solución a problemas genéricos de optimización con variables enteras y funciones no lineales, como los característicos en optimización de redes, sino que cada problema tiene un estudio individual. Tampoco es posible encontrar métodos analíticos para llegar a la solución en estos problemas. El rango de soluciones para intentar evaluar la mejor entre todas las posibles por fuerza bruta, crece exponencialmente con el número de nodos. De hecho, la mayor parte de estos problemas son de clase NP-completa. Esto implica que los algoritmos conocidos capaces de encontrar la solución óptima, son de complejidad exponencial. Encontrar la solución óptima para tamaños de problema (p.e. número de nodos) medio-alto no es computacionalmente posible. Se hacen necesarios por tanto algoritmos basados en heurísticos, que ofrezcan soluciones subóptimas a este tipo de problemas, en un tiempo acotado. La planificación de redes se ha construido sobre este tipo de algoritmos, sobre el que existe un constante esfuerzo investigador.

Una buena forma de modelar las redes y los problemas de planificación asociados es mediante grafos. De esta forma pasaremos de un análisis clásico de los problemas de optimización (basados en funciones objetivos y restricciones), a expresar estos problemas mediante teoría de grafos. Esto será desarrollado en la siguiente sección, pero antes vamos a perfilar en el contexto de la planificación de redes, describiendo el trabajo típico de un planificador o diseñador de red.

1.1.3 Descripción del trabajo típico de un planificador

Un diseño de red es el alma de la construcción de redes. La labor de un planificador o diseñador de redes no consiste en trazar líneas sobre papel. El diseñador tiene que *crear* la estructura de la red, y posteriormente, decidir cómo disponer de los recursos y gastar el dinero.

En el fondo del diseño de redes para la vida práctica hay dos preguntas: ¿Cuánto tengo que gastar para tener una red que funcione?¿y qué tipo de mejoras sobre una red puede comprar X dinero? La respuesta a las dos depende del coste de los servicios básicos de red y de los componentes de red.

Antes de centrarse en diseñar redes tenemos que examinar la cuestión de evaluarlas. Si disponemos de los suficientes servicios o lo suficientemente baratos en la red pública, no hará falta diseñar ninguna red. Y si pretendemos mejorar esto mediante la construcción de una red propia, habrá que hacer una evaluación de las ventajas que compramos con este esfuerzo.

Para evaluar las redes se debe cuantificar en ellas tres características: coste, rendimiento, y fiabilidad. Elaborar una decisión respecto a estos datos es difícil, depende de las prioridades que haya en cada caso. Se pueden hacer planificaciones de red diferentes para el mismo problema, en las que el coste de unas a otras varíe, y asimismo varíe el rendimiento y la fiabilidad. Como se ha dicho antes, no hay claros ganadores.

El trabajo de un planificador de redes es un poco intuitivo. Como se ha visto, no hay métodos para llegar a soluciones perfectas, y en muchos casos los algoritmos se adaptan y "modulan" a mano. Trabajando con entradas de datos que muchas veces son parámetros, se sacan ristras de soluciones, decenas de diseños que encajan con las restricciones impuestas y en los cuales ninguno sobresale claramente sobre otro. Cuantos más diseños que cumplan las restricciones mínimas sean capaces de buscar los algoritmos, más probabilidad habrá de encontrar un algoritmo que se adapte a necesidades complementarias.

La consecuencia es que existe en este tipo de trabajo una necesidad de experimentar con algoritmos lo más rápidos posibles, y lo más "astutos posibles". Esto es, algoritmos que sean capaces de llegar rápidamente a soluciones correctas, sorteando las posibles soluciones no deseadas, perdiendo el mínimo tiempo posible en probar lo que no interesa, y siempre tendiendo a la solución perfecta por medio de continuas evaluaciones rápidas del estado temporal del diseño. En este proyecto se pretende presentar un pequeño compendio de los algoritmos más útiles y populares que existen a este respecto.

Las herramientas de trabajo más adecuadas serán no sólo las que mejor pensadas estén para satisfacer unos requisitos estándar, sino las que puedan moldearse fácilmente para permitir al diseñador buscar de forma más especifica el máximo rango de soluciones para un determinado problema. Por eso, las herramientas usadas serán un compendio de los algoritmos de optimización de redes que existan, para elegir qué algoritmo aplicar en cada caso, en qué orden y con qué finalidad. En la sección 1.3 se mostrarán algunos ejemplos de herramientas que pueden encontrarse actualmente.

1.2 Aproximación a la teoría de grafos

Vamos a dar una introducción al entorno sobre el cual nos vamos a mover. Los grafos son a la teoría de redes lo que el cálculo es a la física: sin las nociones de los grafos no podemos expresar ni investigar adecuadamente el tipo de problemas que se presentan, de forma que se han convertido en el *lenguaje* de las redes.

En esta sección se establecerá la mayoría de la notación que se puede encontrar posteriormente, así como las definiciones de los conceptos básicos de grafos que se usan para la creación de algoritmos. Para ser rigurosos matemáticamente antes de exponer esta teoría de grafos deberíamos dar un repaso a la teoría de conjuntos, pero esto escapa a la finalidad del documento y se ha omitido, aunque pueden encontrarse fácilmente buenos resúmenes sobre ello, p.e. en [1], (apéndice C).

Asimismo, directamente relacionada con la teoría de grafos, y respecto al problema que nos concierne, sería interesante una "Aproximación a la complejidad algorítmica". Nuevamente me remito a una referencia, [2], que vendrá muy bien a quien quiera profundizar en estos temas.

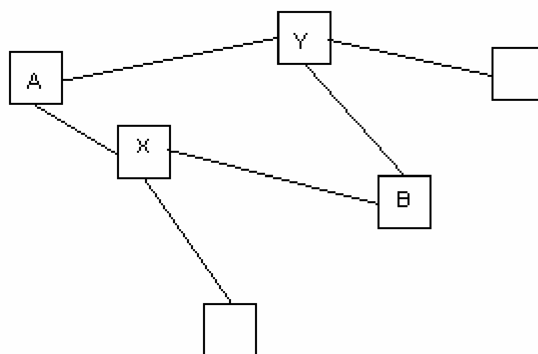


Figura 1-1. Un grafo

En la figura 1-1 tenemos un grafo cualquiera. Los cuadrados son llamados *vértices*, y las líneas *enlaces*. En redes cada vértice representará un *switch* o *router*, es decir un **nodo**, y cada enlace representará una **línea** de telecomunicaciones. En este proyecto consideraremos siempre

enlaces bidireccionales (esto repercute en las definiciones: por ejemplo la de conectividad, para la que, en el caso de unidireccionalidad, habría que especificar distintos tipos, siendo bastante más compleja). Se definen así los siguientes conceptos:

- Un **grafo** G consiste en un conjunto de vértices V y un conjunto de enlaces E. Cada enlace, $e \in E$, tiene dos puntos finales, $v_1, v_2 \in V$. El grafo G se denota como el par *(V,E)*.
- Un **lazo** es un enlace, $e \in E$, donde ambos puntos terminales son el mismo.

  Usaremos tanto la expresión puntos finales como terminales, o puntos extremos.

- Dos enlaces en un grafo son **paralelos** si tienen los mismos puntos terminales.
- Un grafo es **simple** si no tiene lazos o enlaces paralelos.

La gran mayoría de los grafos que se usan en redes son simples, aunque, en casos de estudios de fiabilidad se pueden encontrar enlaces paralelos.

- El **grado** de un nodo es el número de enlaces en el grafo que tienen a tal nodo como punto terminal.

  P.e. en la figura, el grado(A)=2, grado(X)=3.

- Dos nodos v_1 y v_2 son **adyacentes (vecinos)** si hay un enlace *e* que los tiene como puntos terminales.

  P.e. en la figura, A y B no son adyacentes, mientras B y X sí lo son.

- Un **camino** entre 2 vértices v_1 y v_2 consiste en un conjunto de enlaces $(e_1, e_2, ...e_n)$ tales que cada e_i y e_{i+1} tengan un punto terminal en común; v_1 sea un terminal de e_1, y v_2 un terminal de e_n.

  P.e. en la figura, un camino entre A y B podría ser (A,X),(X,B), o (A,Y),(Y,B).

- Un **ciclo** en el grafo G es un camino no vacío desde un vértice v_1 hacia él mismo.

  P.e. en la figura, un ciclo sería (A,X),(X,B),(B,Y),(Y,A).

- Un grafo está **conectado** (es **conexo**) si, dados dos nodos cualesquiera v_1 y v_2, hay un camino entre ellos.

El grafo de la figura está conectado, ya que podemos ir desde cualquier nodo a cualquier otro.

- Un **subgrafo** G' de un grafo G es un par *(V',E')* donde:

$$V' \subset V$$
$$E' \subset E$$

  Si $e \in E'$, entonces ambos terminales v_1 y v_2 pertenecerán a *V'*.

- Un **componente** de un grafo es un subgrafo conectado máximo de *V*.

  P.e:

  - Si $G=(V,\emptyset)$ es el grafo sin enlaces, entonces cada componente será un simple vértice v.
  - Si $G=K_n$ es el grafo completo de n vértices, es decir, habrá un enlace entre cada par de vértices o $\binom{n}{2}$ enlaces en total. Este grafo es conectado. El camino entre cualquier par de nodos es el enlace existente entre ellos, y todo el grafo es un solo componente.
  - En la siguiente figura, tenemos dos componentes: A,X,C,Z y B,Y,D.

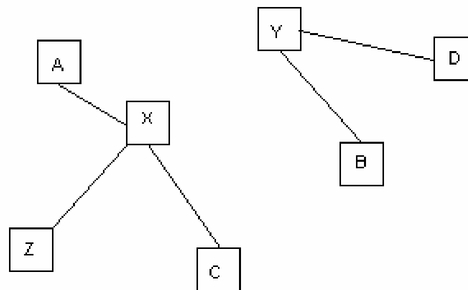


Figura 1-2. Grafo con 2 componentes

- Un **árbol** $T=(V,E)$ es un grafo conexo, simple y sin ciclos.
  - Se puede demostrar que cualquier árbol con n nodos tiene $n-1$ enlaces.
  - Un árbol $T=(V,E)$ es una **estrella** si sólo 1 nodo tiene grado mayor que 1.
  - Un árbol $T=(V,E)$ es una **cadena** si no tiene nodos de grado mayor que 2.

P.e. en la figura 1-2, considerados individualmente, el primer componente sería un árbol estrella, y el segundo un árbol cadena y estrella a la vez.

- Un **bosque** es un grafo conexo y simple.
- Un grafo G es **ponderado** si hay un número real asociado con cada enlace. El **peso** de un enlace e será denotado usualmente como $w(e)$. El grafo podrá ser denotado asimismo (G,w). Si G' es un subgrafo de G, entonces $w(G') \equiv \sum_{e \in G'} w(e)$.

A no ser que se diga específicamente lo contrario asumimos que los pesos con los que se ponderan los enlaces son positivos. Es decir, $w: E \rightarrow \Re^+$.

- Dado un conjunto de vértices $(v_1, v_2, \ldots, v_n)$, un ***tour*** T es un conjunto de n enlaces E tales que cada vértice v tiene grado 2 y el grafo es conexo.
- Un **tour TSP** se puede definir de varias formas:
  - Dado un conjunto de vértices $(v_1, v_2, \ldots, v_n)$ y una función de distancia $d: VxV \rightarrow \Re^+$, el ***traveling salesman problem*** es encontrar el *tour* T tal que

$$\sum_{i=1}^{n} d(v_{t_i}, v_{t_{i+1}})$$

,es un mínimo. En esta notación identificamos $v_{t(n+1)}$ con $v_{t(1)}$.

- Dado un grafo ponderado *G(V,E,w),* simple y conectado, encontrar el *tour* subgrafo de coste mínimo.

- Dado un grafo conexo *G=(V,E)*, el vértice *v* es un **punto de articulación** si quitándolo a él y a todos los enlaces para los que es un punto terminal, se desconecta el grafo.

  En un árbol, cualquier vértice con grado mayor que 1 es un punto de articulación.

- Si un grafo conexo *G=(V,E)* no tiene puntos de articulación, entonces el grafo es **2-conectado (biconectado, biconexo)**.
  - Supongamos que $G_1=(V_1,E_1)$ y $G_2=(V_2,E_2)$ son grafos biconexos con $V_1 \cap V_2 = \emptyset$. Sean $v_1, v_2 \in V_1$ y $v_3, v_4 \in V_2$. Entonces el grafo G con vértices $V_1 \cup V_2$ y enlaces $E_1 \cup E_2 \cup (v_1, v_2) \cup (v_3, v_4)$ es biconexo.
- Un **bloque** en un grafo es un subgrafo 2-conectado máximo de *V*.
  - Un bloque es la máxima porción de un grafo en donde la desconexión de cualquier nodo no implica la desconexión del bloque.
  - Si dos bloques están conectados (en un punto de articulación), cualquier enlace entre nodos (que no sean el punto de articulación) entre los dos bloques los convertirá en un solo bloque.
- La **fiabilidad** de una red es la probabilidad de que siga siendo conexa, ante la caída de determinados nodos y enlaces.

1.3 Herramientas de planificación WAN

En la actualidad existen varios lenguajes y paquetes para simular redes de computadoras los cuales los podemos dividir en 3 tipos, según [4]: Lenguajes de simulación de propósito general, lenguajes de simulación orientados a las comunicaciones, y simuladores orientados a las comunicaciones. Como ejemplos de los primeros tenemos: Arena, BONeS, NetDESIGNER, GPSS/H, MODSIM II, SES/workbench, SIMSCRIPT II.5. Como ejemplos de los segundos contamos con OPNET y DELITE. Algunos ejemplos de simuladores orientados a las comunicaciones pueden ser: BONeS PlanNet, COMNET III y NETWORK.

En general todas estas herramientas ayudan a modelar entornos de red para ser analizados, algunos utilizan scripts, otros cuentan con entornos gráficos, sin embargo tienen el inconveniente de ser comerciales, y no ofrecen su código libre para investigación y extensión.

De estos tipos de herramientas, las que más nos interesan para el objetivo de este proyecto son las de lenguajes de simulación orientados a las comunicaciones. A continuación vamos a presentar una breve descripción de las dos herramientas fundamentales de este grupo:

- OPNET:

Opnet es una plataforma de simulación y diseño de redes, creada por OPNET Technologies, Inc. proveedor de software de gestión para redes y aplicaciones. Su producto está altamente extendido, y tiene utilidad en un gran abanico de problemas de red. Es capaz de hacer cálculos de capacidades, auditar redes y configurar su gestión, modelado y simulación. Es un programa sofisticado válido para corporaciones, proveedores de servicio, y fabricantes de redes.

Proporciona un entorno fácil, en el cual se pueden elegir y configurar diversos tipos de redes así como sus componentes, de forma rápida y efectiva.

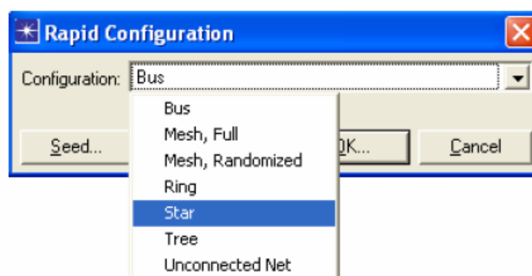


Figura 1-3. Un menú de configuración en OPNET

El usuario elige dichos componentes para crear un entorno de su elección en el que podrá hacer simulaciones de red, analizar comportamientos, probar posibles cambios para mejorar la red, etc...

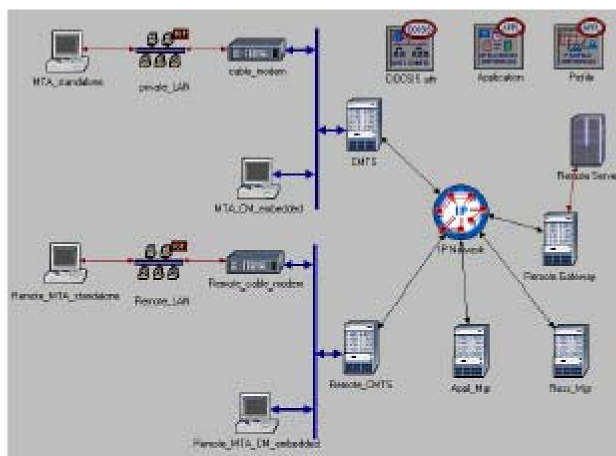
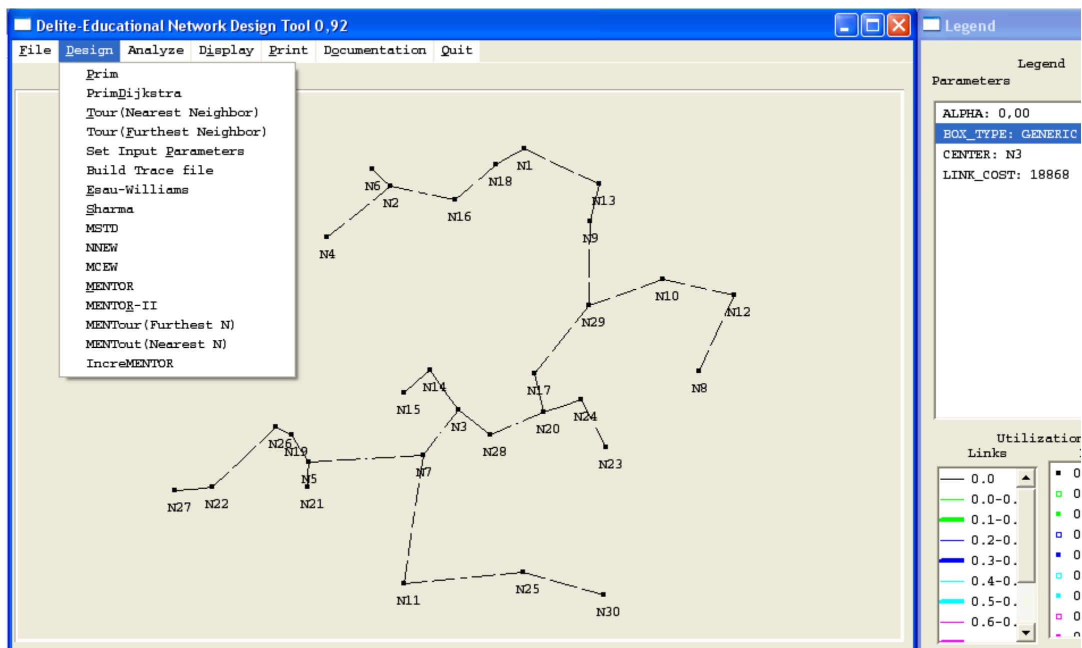


Figura 1-4. Entorno de trabajo en OPNET

Para una mejor descripción sobre las características y el procedimiento para adquirir esta herramienta, dirigirse a [6].

- Delite:

Delite es una herramienta creada por los autores de [1], como respuesta a la necesidad de poder resolver los problemas de redes que se estudian de forma computerizada. Es una herramienta de diseño de red no comercial (una de las pocas que pueden encontrarse gratuitamente). Implementa los algoritmos desarrollados en el texto sin entrar en las complicaciones de una herramienta sofisticada.



Está pensada para trabajar con redes de tamaño medio. El manejo es sencillo, si bien tiene algunos problemas de limitación de parámetros o interacción insuficiente con el usuario que hacen que a menudo no funcione correctamente para probar ejemplos de redes cuando no se lleva el suficiente cuidado en prepararlas.

La herramienta puede adquirirse en [7], y un pequeño manual de software en [8].

1.4 Objetivos del proyecto

Haciendo un breve análisis de las herramientas que se encuentran para la planificación y diseño de redes se observa que presentan ciertos inconvenientes que sería importante subsanar:

- O bien son comerciales, de pago (prácticamente todas).
- O bien no son demasiado buenas didácticamente (Delite), por diversas razones:
  - Inestabilidad en ejecución (p.e., caso de Delite).
  - Mala documentación y ayuda.
  - Falta de versatilidad, dificultad de modificación o integración.

- Falta de claridad a nivel de programación como para entender el funcionamiento del código, para controlar y saber lo que hace o modificarlo.

Por tanto, y especialmente a nivel universitario, interesa una herramienta:

- Que disponga de los principales algoritmos para planificación de redes.
- Que sea didáctica, y fácilmente modificable y extensible.
- Donde se vean claros los conceptos de planificación para los alumnos.
- Con código altamente reutilizable y legible (al más alto nivel posible).

De todo esto va deduciéndose que la herramienta debería ser programada en un lenguaje versátil, con un entorno que dé opción a la fácil edición de los diferentes módulos y algoritmos, así como la transformación y visualización de los resultados. En este proyecto, se ha optado por la herramienta MATLAB, que cumple todos los requisitos, además de ser ya una herramienta familiar y de uso extendido en la universidad.

Eligiendo MATLAB, los objetivos se concretan en el diseño de un módulo formado por una serie de funciones MATLAB, de algoritmos y utilidades para el tratamiento de los datos con los que se está trabajando. Es decir, sería la creación de un paquete o *toolbox*, un módulo de planificación de redes.

Concluyendo, todo esto deriva en una decisión, que ha formado el **objetivo del presente proyecto**:

➔ Crear una herramienta de planificación y diseño de redes mediante un paquete (*toolbox*) de funciones en MATLAB (lenguaje que cumple todos los requisitos), formado por funciones altamente reutilizables, fácilmente extensible, y con ayuda bien documentada.

A este *toolbox* se le ha bautizado con el nombre de MatPlaNet Tool (*MATLAB Network Planning Tool*).

Capítulo 2
Teoría de planificación en redes WAN

2.1 Problema del MST y SPT

Los problemas de redes suelen ser sencillos de expresar, sin embargo, sus resultados involucran tal conjunto de soluciones posibles que es muy difícil encontrar una de ellas óptima, ni de comprobar si realmente lo es. La complejidad de estos problemas suele ser de crecimiento exponencial. Las soluciones conseguidas en un tiempo aceptable dado un problema grande apuntan al uso de heurísticos. El procedimiento analítico, pues, no es muy popular, y es más efectivo expresar el problema como un grafo que como un problema de optimización, lo cual nos brinda la oportunidad de ayudarnos de herramientas ya estudiadas y desarrolladas.

La teoría de redes que se va a exponer aquí ha seguido esta tendencia y tiene pues una gran relación con la teoría de grafos, de la cual se ha proporcionado un breve repaso en la sección 1.2. La topología de las redes, considerada como un conjunto de nodos y enlaces con cierto coste, es representativa de una red y sobre ella se pueden operar con conceptos como tráfico, costes y utilización. Dicha topología puede ser representada como un grafo en el que cada línea entre dos puntos representa un enlace entre dos nodos, y tiene un peso que la define. Es decir, partiremos de grafos simples y ponderados.

Los problemas más completos que normalmente se presentan en la planificación de redes WAN vienen a la hora de intentar diseñar una topología en la que todos los nodos estén conectados, con un cierto nivel de fiabilidad en dicha conectividad, y se satisfagan sus necesidades de tráfico y retardo al mínimo coste posible. Estos problemas entrarían dentro de la clasificación como TCFA (*topology*, *capacity and flow assignment*). El primer problema a resolver, por lo tanto, es el MST.

2.1.1 Definición del problema MST

"Sea G(V,E,w) un grafo ponderado, simple y conexo. Llamamos *Minimum Spanning Tree* (MST), esto es, Árbol de Expansión Mínima, al subgrafo G'(V',E',w) conexo, que contenga todos los nodos de G con el menor coste total Σw(G')."

O bien:

"Dado un grafo donde cada enlace tiene un peso real, seleccionar un subgrafo conexo con el coste mínimo total".

En nuestro recorrido por los problemas de redes, nuestro primer objetivo, el más básico, es conseguir una red conectada al mínimo coste. Posteriormente podremos desarrollar soluciones que además tengan en cuenta otros conceptos como utilización, debilidad de la red, etc. Sin embargo, en principio, tenemos que partir de lograr una solución al MST.

Bien, para enfrentarnos a este problema partimos de dos algoritmos posibles:

- Kruskal
- Prim

Kruskal considera cada nodo un componente, y va recorriendo los enlaces posibles de menor a mayor coste, aceptándolos (y aumentando el correspondiente componente) en el caso de que los nodos que unan estén en componentes separados.

```
1. Chequear que el grafo es conexo. Si no lo es, abortar.
2. Ordenar los enlaces del grafo G en orden ascendente de
   pesos.
3. Marcar cada nodo como un componente separado.
4. Repetir hasta que se hayan aceptado |G|-1 enlaces:
   Sea e el enlace a examinar (con el orden anterior: se
   empieza por el de menor peso).
      • Si los 2 nodos de e están en componentes diferentes,
        unir esos componentes y aceptar el enlace.
      • Incrementar el número de enlaces aceptados en 1
        unidad.
```

Algoritmo 2-1. Kruskal

Prim procede de forma diferente. En vez de ordenar los enlaces al principio, empieza con un árbol consistente en un solo nodo y ningún enlace, y va añadiendo nodos buscando el enlace más barato posible entre cualquiera de los nodos q aun no están en el árbol y el último nodo que se ha añadido (nodo pivote).

Cada nodo tiene una etiqueta de coste. En cada iteración se actualizan las etiquetas respecto al nodo pivote, para ir viendo qué vecino de cualquiera de los nodos del árbol será más barato. Cada vez que se escribe una etiqueta de coste, se escribe también una etiqueta de procedencia indicando de donde viene. Esto permitirá que después de todo el proceso pueda reconstruir los enlaces correctos.

```
1. Inicializar poniendo todas las etiquetas a infinito.
2. Pivote = Nodo inicial, incluirlo en el árbol.
3. Buscar los vecinos del nodo pivote que no estén en el
   árbol.
4. Actualizar la etiqueta de cada los vecinos con el valor
   mínimo entre:
      • el coste respecto al pivote.
      • el coste que ya tienen (respecto a un pivote
        anterior).
5. Si todos los nodos están en el árbol, terminar.
   Si no, seleccionar como nuevo pivote el nodo con menor
   etiqueta fuera del árbol e incluirlo en el árbol.
```

Algoritmo 2-2. Prim

De esta manera se van incluyendo en el árbol todos los nodos, en orden de menor coste relativo al nodo pivote temporal. Al final del algoritmo, cada uno de los nodos del árbol habrá actualizado la etiqueta de todos sus vecinos si el camino entre ellos es más barato, por lo tanto, tendremos todos los caminos de mínimo coste.

La diferencia en uso de Kruskal y Prim es que el segundo necesita que le pasemos un nodo inicial. Sin embargo, se puede probar que:

- dado un grafo conexo, tanto el algoritmo de Prim como el de Kruskal nos conducen a una solución MST.
- Si esta solución es única, para *cualquier* nodo de comienzo que le pasemos al algoritmo de Prim, se construye el mismo MST.

2.1.2 Análisis MST y definición del problema SPT

Hemos visto dos algoritmos para conseguir MST a partir de un grafo conexo, y sabemos que es la solución de mínimo coste. Ahora bien, ¿es una solución buena en otros aspectos? Un análisis de las estructuras MST nos revela lo siguiente:

Los MST son buenos cuando:

- Los enlaces son altamente fiables. (ver fiabilidad de una red, en el apéndice C)
- Las redes pueden tolerar baja fiabilidad.
- El número de nodos es pequeño.

Y no son tan buenas redes cuando:

- El número de nodos es grande.

Características a tener en cuenta:

- Tanto Prim como Kruskal nos dan una solución óptima respecto al *coste*.
- La fiabilidad de una red en árbol desciende (la probabilidad de fallo aumenta) exponencialmente con el número de nodos.
- El número medio de saltos obtenido aumenta rápidamente con el número de nodos.
- Los MST tienden a tener caminos muy largos y poco directos.

Si nos fijamos, la mayoría de problemas provienen del hecho de construir caminos largos, de muchos saltos, entre las localizaciones, porque será esto lo que derive en poca fiabilidad (dependencia de demasiados enlaces intermedios), en alto tráfico (cada enlace llevará el tráfico de los nodos del enlace más todos los nodos más distantes que lo usen), poca directividad (por el hecho mismo de tener muchos saltos), etc.

A la vista de estos resultados, se puede deducir que se hace necesario algún otro tipo de red que compense las deficiencias de los MST, y el problema a atacar en primer lugar va a ser el número de saltos. Inmediatamente podemos fijarnos en otro tipo de problemas que se encargan precisamente de este aspecto. Son los problemas de búsqueda del Árbol de Camino mas Corto, o SPT (Shortest Path Tree), que se enuncia como sigue:

"Sea G(V,E,w) un grafo ponderado, simple y conectado y un nodo de inicio O. Llamamos *Shortest path Tree* (SPT), esto es, Árbol de Camino más Corto, al subgrafo G'(V',E',w) conexo, que contenga todos los nodos de G de forma que el camino entre el nodo inicio O y cualquiera de los nodos sea el menor posible."

Existe un algoritmo óptimo que nos puede ayudar a encontrar el SPT de un grafo. Este es, el de Algoritmo de Dijkstra, mostrado a continuación:

1. Inicializar poniendo todas las etiquetas a infinito.
2. Pivote = Nodo inicial, incluirlo en el árbol.
3. Buscar los vecinos del nodo pivote que no estén en el árbol.
4. Actualizar la etiqueta de cada los vecinos con el valor mínimo entre:
   - el coste respecto al pivote + *coste del pivote*.
   - el coste que ya tienen (respecto a un pivote anterior).
5. Si todos los nodos están en el árbol, terminar.

   Si no, seleccionar como nuevo pivote el nodo con menor etiqueta fuera del árbol e incluirlo en el árbol.

Algoritmo 2-3. Dijkstra

El funcionamiento de este algoritmo es muy parecido al de Prim, con una sola diferencia. A la hora de etiquetar los nodos vecinos del nodo pivote, se les asigna un coste que es un histórico de los costes hasta el pivote inicial. Es decir, se suma el coste posible del nuevo enlace al coste que traía el pivote. Esto provocará que siempre la elección del nodo de menor etiqueta se haga respecto al nodo inicial, por lo tanto este nodo tendrá mucho más peso en el diseño final, siendo usual un resultado de topología en estrella centrada en el nodo pivote inicial.

2.1.3 Solución intermedia: Prim-Dijkstra

Llegados a este punto, se han obtenido dos soluciones distintas al diseño de topologías básicas de red, pero ¿cuál elegir? Con observar un par de ejemplos, se da uno cuenta de que hay demasiada distancia entre una solución y otra. De esta manera era natural buscar una solución intermedia que aunara los resultados del MST (bajo coste) y el SPT (bajo número medio de saltos).

Teniendo en cuenta la similitud entre el algoritmo de Prim y el de Dijkstra, se pensó hacer un algoritmo intermedio en el cual el grado de prim o de dijkstra se controlara mediante otra variable α, que es el porcentaje de coste del pivote que se le suma al coste de cada nueva etiqueta. El resultado es el algoritmo de Prim-Dijkstra:

```
1. Inicializar poniendo todas las etiquetas a infinito.
2. Pivote = Nodo inicial, incluirlo en el árbol.
3. Buscar los vecinos del nodo pivote que no estén en el
   árbol.
4. Actualizar la etiqueta de cada los vecinos con el valor
   mínimo entre:
        • el coste respecto al pivote + α· (coste del pivote).
        • el coste que ya tienen (respecto a un pivote
          anterior).
5. Si todos los nodos están en el árbol, terminar.
   Si no, seleccionar como nuevo pivote el nodo con menor
   etiqueta fuera del árbol e incluirlo en el árbol.
```

Algoritmo 2-4. Prim-Dijkstra

El algoritmo está controlado por un parámetro α, cuyos límites son el 0 y el 1. Cuando α=1, se suma íntegramente y corresponde a un Dijkstra puro, y cuando α=0 no se suma en absoluto y corresponde a un Prim puro. Un α intermedio puede resultar en un algoritmo más en estrella que con Prim (con lo cual se reduce el número de saltos), pero más barato que Dijkstra (porque el nodo inicial no tendrá tanto peso en la elección de los enlaces).

2.2 Problema MST con capacidades (CMST)

2.2.1 Definición del problema

Un paso más allá de encontrar un árbol de coste mínimo, es buscarlo con ciertas exigencias respecto a los parámetros de capacidad de los enlaces. Usualmente nos encontramos en las redes de acceso con que hay establecido un caudal específico desde varios nodos hacia un nodo de acceso o nodo central, y asignada una capacidad máxima de enlace. Obviamente, en este caso se puede calcular con facilidad que no todas las combinaciones de enlaces cumplen esa restricción de capacidad, ya que habrá un límite de trafico que impedirá que se puedan unir varios flujos. Se hace necesario pues un algoritmo que sea capaz, no sólo de encontrar un diseño que cumpla las limitaciones de capacidad, sino que lo haga con un coste mínimo. A éste problema se le denomina CMST, es decir, búsqueda de un árbol de mínimo coste con restricciones de capacidad, y se enuncia de la siguiente manera:

"Dado un nodo central N0, un conjunto de otros nodos (N1, ..., Nn), el caudal que generan hacia el nodo central (w1,... wn), la capacidad posible de todos los enlaces W, y el coste de interconexión entre cada par de nodos para los enlaces de la capacidad W,

Encontrar el árbol que interconecte todos los nodos con coste mínimo, sin que el tráfico que circula por ningún enlace supere un determinado valor (p.e. 50% de W)."

Un algoritmo que satisface estas exigencias de forma sencilla es el Esau-Williams, que se describe a continuación.

2.2.2 Esau-Williams

El proceso es ir calculando una función tradeoff, que da una idea acerca de si interesa enlazar dos nodos. Para esto, va creando componentes o grupos de enlaces y asignándoles costes fijos a todos los enlaces dentro del mismo componente. La función tradeoff dice el coste que se ahorra cada nodo (componente) de llevar su tráfico por el enlace vecino más barato que tiene en puesto de llevarlo directamente al nodo central N0. El menor tradeoff, es el de más ahorro, así que si cumple la restricción (de que la suma de tráficos de uno y otro nodo no sobrepase W) se acepta ese enlace y los dos nodos del enlace forman un solo componente (o anexan sus componentes). Los componentes tienen un coste hacia el nodo N0 igual para todos sus nodos, que es el más barato de entre todos los nodos que forman el componente. Reiterando el proceso hasta que todos los nodos sean parte de un componente se consigue que todos los nodos vayan al central por el camino más barato que les permite la restricción de capacidad.

Para un nodo N_i, la función tradeoff se define así:

$$tradeoff(N_i) = \min_{j \notin componente_de_i} (coste(N_i, N_j)) - coste(componente(N_i), N0)$$

El algoritmo es el siguiente:

1. Calcular el *tradeoff* para todos los nodos.
2. Elegir el nodo N_i con menor *tradeoff*.
   - Si *tradeoff* ≥ 0, pasar a 3.
   - Si *tradeoff* < 0
     - Si γ (componente i) + γ (componente j) ≤ W
       - Aceptar el enlace.
       - Unir los dos componentes.
     - Si γ (componente i) + γ (componente j) > W
       - Eliminar el enlace.

   - Recalcular *tradeoff*.
   - Repetir 2.
3. Fin del Algoritmo: Conectar cada componente a N0

Algoritmo 2-5. Esau-Williams

<u>Un ejemplo del funcionamiento del Esau-Williams:</u>

Asumimos que W=3 y cada nodo tiene un peso w=1 (por lo tanto, estaremos construyendo caminos al nodo central de máximo 3 saltos).

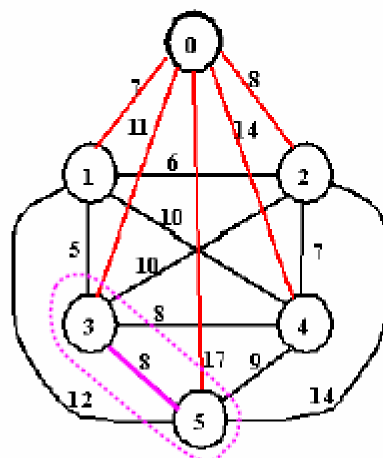


Figura 2-1. Algoritmo de E-W

[3]

Partiendo de la anterior topología, donde el central es el nodo 0, el cálculo de tradeoffs es como sigue:

Tradeoff(1)= mínimo del nodo 1 a cualquiera= 5 (al nodo 3)

coste del componente que contenga al nodo 1 hacia el nodo 0= 7

5-7=-2

Tradeoff(2)= 6-8=-2

Tradeoff(3)= 5-11=-6

Tradeoff(4)= 7-14=-7

Tradeoff(5)= 8-17=-9

El menor tradeoff es el del nodo 5. Chequeamos si el tráfico del componente de 3 (su nodo más próximo) mas el del propio nodo 5 sobrepasan la restricción.

$\Sigma w_i = w_3 + w_5 = 1+1=2$. → $2 \leq W$ →Ok

Puesto que cumple la restricción, aceptamos el enlace (5,3), que hemos marcado en la figura, y se sigue:

Nuevo cálculo de tradeoffs:

Tradeoff(1)= 5-7=-2

Tradeoff(2)= 6-8=-2

Tradeoff(3)= 5-11=-6

Tradeoff(4)= 7-14=-7

Tradeoff(5)= 9-*11*=-2 (aquí ha cambiado los dos costes: en el primero, ahora no se considera vecino el nodo 3, por lo tanto el más cercano es el 4, con coste 9; en el segundo, ahora se pone el coste más barato de entre los nodos del *componente (3,5)* hacia el nodo 0, esto es, el del enlace (3,0), que vale 11)

Menor tradeoff en el nodo 4. Chequeamos (4,2):

$\Sigma w_i = w_4 + w_2 = 1+1=2 \leq 3$ → Ok, incluímos (4,2)

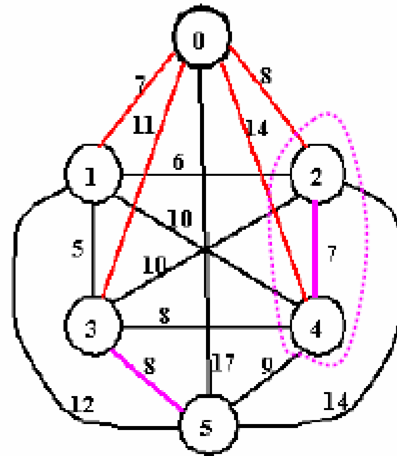


Figura 2-2. Algoritmo de E-W

[3]

Nuevo cálculo de tradeoffs:

Tradeoff(1)= 5-7=-2

Tradeoff(2)= 6-8=-2

Tradeoff(3)= 5-11=-6

Tradeoff(4)= *8-8*=0

Tradeoff(5)= 9-11=-2

Menor tradeoff en nodo 3. Chequeamos (3,1):

$\Sigma w_i = w_5 + w_3 + w_1 = 1+1+1 = 3 \leq 3$ → Ok, incluímos (3,1)

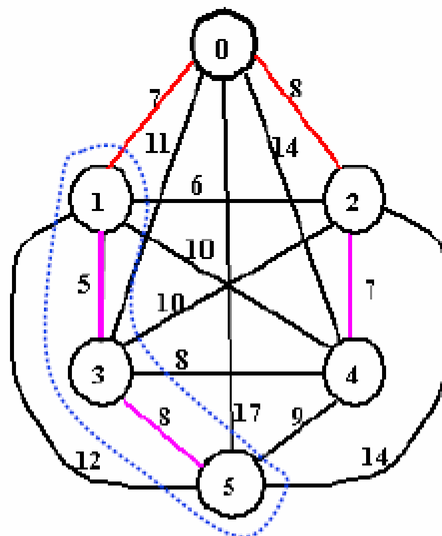


Figura 2-3. Algoritmo de E-W

[3]

Nuevo cálculo de tradeoffs:

Tradeoff(1)= *6-7*=-1

Tradeoff(2)= 6-8=-2

Tradeoff(3)= *8-7*=-1

Tradeoff(4)= 8-8=0

Tradeoff(5)= *9-7*=-2

Menor tradeoff en nodo 2. Chequeamos (2,1):

$\Sigma w_i = w_5+w_3+w_2+w_1=1+1+1+1=4 > 3$ → Not Ok → desechamos el enlace (2,1)

Tradeoff(2)= 8-8=2

Menor tradeoff en nodo 1. Desechamos el enlace (1,2) por la misma razón del anterior.

Tradeoff(1)= 7-7=0.

Menor tradeoff en nodo 3. Chequeamos (3,4):

$\Sigma w_i = w_5+w_4+w_3+w_2+w_1=1+1+1+1+1=5>3$ →Not Ok → desechamos el enlace (3,4)

Menor tradeoff en nodo 5. Chequeamos (5,4):

$\Sigma w_i = w_5+w_4+w_3+w_2+w_1=1+1+1+1+1=5>3$ →Not Ok → desechamos el enlace (5,4)

Tradeoffs finales:

Tradeoff(1)= 7-7=0

Tradeoff(2)= 8-8=0

Tradeoff(3)= 10-7=3

Tradeoff(4)= 8-8=0

Tradeoff(5)= 17-7=10

El algoritmo termina porque no quedan tradeoffs negativos, tenemos dos componentes: (1,3,5) y (2,4), que se enganchan ahora al nodo 0 mediante su enlace más barato:

Aceptamos (1,0) y (2,0).

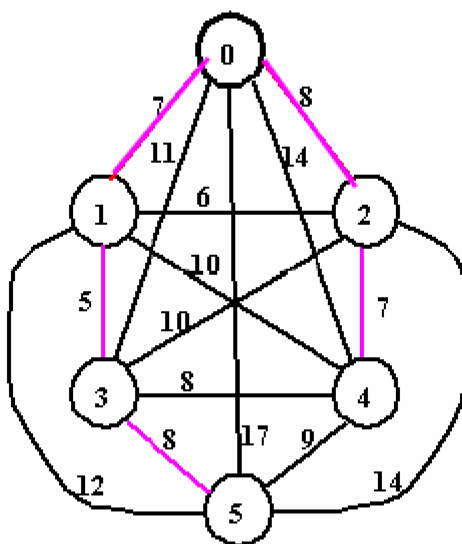


Figura 2-4. Algoritmo de E-W

[3]

Nota:

El algoritmo E-W no es el único que obtiene un CMST, sino tan solo es un heurístico que parece funcionar bien. La acreditabilidad del algoritmo se ha testeado ampliamente, con diferentes requerimientos de capacidad y tipos de tráfico. Un test típico es el "*1-exchange*", que comprueba si no existe ningún cambio de enlaces respecto a la solución dada, que abarate el coste sin violar las condiciones de capacidad. Esau-Williams tiene un excelente comportamiento frente a todos estos testeos, pero no son la prueba concluyente de que la solución que da sea la mejor.

Casos en los que se piensa especialmente que se podría encontrar una mejor solución son aquellos en los que devuelve líneas cruzadas. La solución especialmente este aspecto, viene de mano de otro algoritmo, el algoritmo de Sharma. Para mayor información sobre él, consultar [1]. (5.7)

2.3 Problema de multispeed CMST

2.3.1 Definición problema

Hemos tratado el CMST hasta ahora con una restricción de capacidad máxima por enlace igual para todos los enlaces. Pero esto no es el caso real, en el que tenemos cierta flexibilidad para poner un tipo de línea u otra en cada enlace. Es más natural que una red de acceso se construya usando varios tipos de línea que tienen diferente velocidad. El coste de un enlace viene marcado por un tarifario como el de la siguiente figura. En general, como norma práctica, el coste respecto a la capacidad del enlace crece aproximadamente en una proporción como su raíz cuadrada, es decir, el doble de capacidad costará $\sqrt{2}$ veces más. Los demás parámetros se explicarán con más detalle en la sección 2.5.3.

Tipo de línea	Coste Fijo	DIST_Cost1	DIST_Cost2	DIST1
9.6 Kbps	$200	$2.00	$1.40	300
56 kbps	$500	$5.00	$3.00	250
128 kbps	$750	$8.00	$4.40	350

Tabla 2-1. Tres típicos tipos de línea usados en diseños de acceso local

Si bien es cierto que en algunos casos se aconseja que una red sea homogénea, hay que tener en cuenta en normalmente la posibilidad de instalar enlaces con diferentes velocidades, así que necesitamos un nuevo algoritmo eficiente que haga todo lo anterior, pero incluya los mecanismos necesarios para tener en cuenta este concepto de *multispeed*, es decir, que construya árboles a bajo coste con enlaces de diferente capacidad.

El problema *multispeed* CMST es NP-Completo, es decir, El algoritmo encontrado en este caso para solucionar este nuevo problema es el MSLA, cuyo funcionamiento se describe a continuación.

2.3.2 MSLA

El algoritmo es parecido al Esau-Williams, con ciertas diferencias. La más importante, es que en el cálculo de los *tradeoffs* ahora se tiene en cuenta el coste de actualización de los enlaces del nodo destino, si la adición de tráfico requiere que se ponga una línea de más capacidad:

$$tradeoff(N_i, Nj) = coste(N_i, N_j, k) + upgrade(Nj, weight(i)) - coste(componente(N_i), N0, k)$$

, donde *upgrade* es el coste de añadir un tráfico igual a *weight(i)* a los enlaces que conectan j con el nodo central.

Para cada nodo, los *tradeOffs* (ahora hay uno por cada nodo destino posible, mientras antes solo era uno, el que iba hacia el nodo más próximo) representan las ganancias de pasar un enlace por otro nodo en vez de llevarlo directamente al nodo central. Ahora en cada enlace se tiene en cuenta en todo momento el tráfico que va a pasar por él, y con respecto a una utilización que se da como parámetro, una capacidad "sobrante" o *spare capacity*, que será la que se mida con respecto al tráfico que aporta un enlace, a la hora de decidir si hace falta mejorar el tipo de línea o no.

Mientras queden *tradeoffs* negativos, se van añadiendo enlaces, y cuando no quedan el algoritmo finaliza.

1. Asignar a todos los nodos una línea con la capacidad mínima necesaria k para llevar el γ del nodo al nodo N0.
2. Calcular todos los *tradeoff* para todos los nodos.
3. Elegir el nodo N_t con menor *tradeoff*.
   - Si *tradeoff* ≥ 0, pasar a 4.
   - Si *tradeoff* < 0
     - Aceptar el enlace y añadir al árbol definitivo.
     - Unir los componentes respectivos de los nodos del enlace.
     - Actualizar todas las *spare capacity* y los *weight* de los nodos que hay en el camino hacia el nodo N0.
   - Recalcular *tradeoff*.
   - Volver a 2.
4. Fin del Algoritmo: Conectar cada componente a N0

Algoritmo 2-6. MSLA

Es importante observar que en el MSLA no se hace un chequeo de si el enlace sugerido por el *tradeoff* sobrepasa una restricción de capacidad o no. La única restricción es la que nos dan los tipos de línea disponibles. En el cálculo de cada *tradeoff* viene incluida la condición de que si no encuentra líneas de suficiente capacidad para conectar los enlaces, queda descartado

(se pone a un valor infinito). El único problema puede darse solamente en el caso de no poder unir siquiera todos los nodos directamente al nodo central con ninguno de los tipos de línea disponibles, en este caso los parámetros iniciales del algoritmo son insuficientes, basta con dar opción a más tipos de línea.

Un ejemplo a grandes rasgos del funcionamiento del MSLA:

Tenemos una topología inicial tal y como se muestra en la figura, con unos pesos de cada nodo al centro N0. Disponemos de 3 tipos de líneas: a 9.6 Kbps, a 56Kpbs, y a 128Kbps, y la utilización es del 50%.

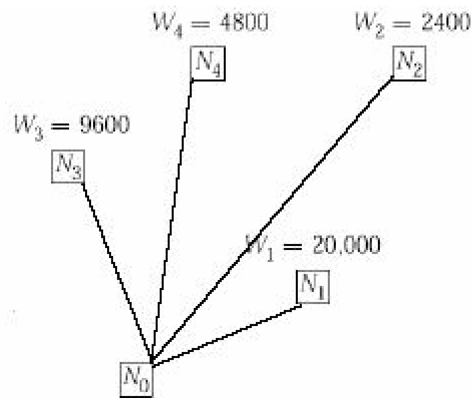


Figura 2-5. Algoritmo MSLA

[3]

Primer paso: Se calculan los tipos de línea mínimos que requiere cada nodo y su capacidad sobrante:

- spare_capacity(1) = 28000 – 20000 = 8000 (link type 1, 56 Kbps)
- spare_capacity(2) = 4800 – 2400 = 2400 (type (link type 0, 9.6 Kbps)
- spare_capacity(3) = 28000 – 9600 = 18400 (link type 1, 56 Kbps)
- spare_capacity(4) = 4800 – 4800 = 0 (link type 0, 9.6 Kbps)

Sites	Line	Spare capacity	Predecessors
N1	D56	8000	0
N2	D96	2400	0
N3	D56	18400	0
N4	D96	0	0

Tabla 2-2. Estado inicial del algoritmo MSLA

Segundo paso, se calculan los *tradeOffs*. Vamos a ver en este ejemplo solo los más interesantes. En principio, parece que N2 está muy lejos del centro y tiene a N4 cerca. Calculamos su *tradeoff.* Hay q tener en cuenta que N4 no tiene capacidad sobrante, por lo tanto habrá que convertir el enlace de N4 a N0 al siguiente tipo de línea, la notación sería:

upgrade (4, 2400) = c(4,0,1) – c(4,0,0)

y el tradeoff de 2 a 4 quedaría:

$Tradeoff_2(4) = c(2,4,0) + (c(4,0,1)-c(4,0,0)) - c(2,0,0) > 0$, -> no se coge.

Otro interesante es el que va de 4 a 3. Lo calculamos:

$Tradeoff_4(3) = c(4,3,1) + (0) - c(4,0,1) < 0$, -> Se comprueba que es el mejor tradeoff, además ni siquiera necesita actualización de línea, pues la *spare capacity* que tenia el enlace de 3 a 0 (18400) da cabida a todo el tráfico de N4 (4800). Ahora el tráfico de N3 será 9600+4800=14400, y la nueva capacidad sobrante, 28000-14400=13600.

Se vuelven a calcular *tradeoffs*, y el siguiente mejor es el que va del nodo 2 al 3, que sigue sin necesitar *upgrade*, puesto que 2400<13600.

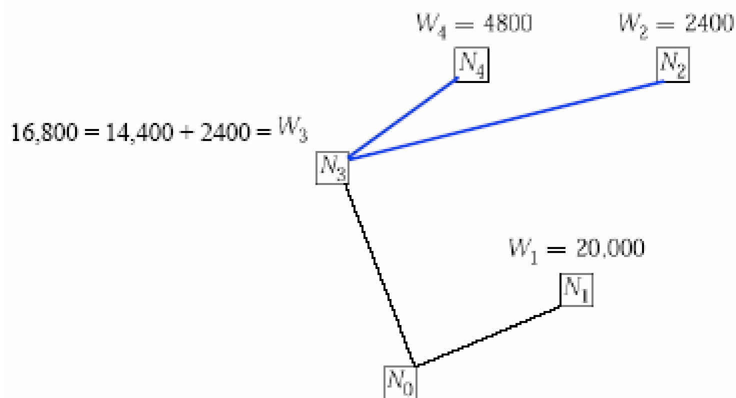


Figura 2-6. Algoritmo MSLA [3]

Finalmente, en el último paso, se conectará N3 a través de N1, el cual deberá actualizar su línea a D128K, ya que recoge un tráfico superior al que permitiría una línea de 56K con esa utilización.

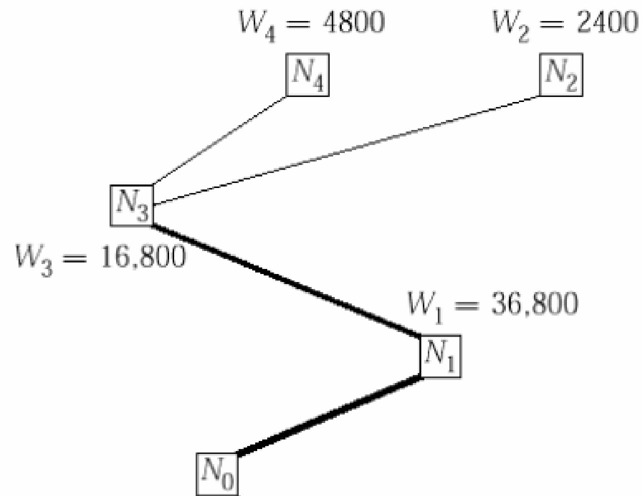


Figura 2-7. Algoritmo MSLA

[3]

Sites	Line	Spare capacity	Predecessors
N1	D128	37200	N0
N2	D96	2400	N3
N3	D56	11200	N1
N4	D56	0	N3

Tabla 2-3. Final del algoritmo MSLA

2.4 Planificación de redes *Mesh*

2.4.1 Definición del problema

En este apartado se expone el caso de que queremos diseñar un *backbone*, es decir, una serie de enlaces entre nodos importantes de la red en lo que a tráfico se refiere. En este caso la *mejor topología no está limitada a un árbol*, como puede verse en un ejemplo.

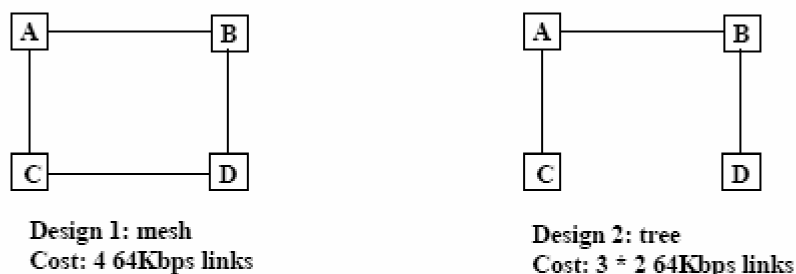


Figura 2-8. Comparación Mesh – Tree

[3]

En la figura tenemos 4 sitios, A, B, C y D. El patrón de tráfico es tal que cada nodo se comunica con sus dos nodos adyacentes, pero a través de la diagonal. Se disponen de enlaces de 64Kbps, y se asume una red de conmutación de paquetes, así que no se quieren cargar los enlaces por encima del 50% de utilización.

El diseño correcto para este problema es el rectángulo formado con el diseño 1. Sin embargo si nos restringimos a un árbol (diseño 2) tendremos 3 enlaces pero cada uno requerirá 2 líneas de 64 Kbps, ya que el tráfico entre los sitios q no están directamente conectados (C-D) requerirá un camino de 3 saltos. El coste en este caso aumenta, cuando en el primero además el número medio de saltos es inferior.

Las redes *mesh* o "tipo malla" se llaman así porque pasan por conseguir un *mesh-connected backbone* (un grupo de nodos importantes conectados en malla), y conforman un grado adicional de dificultad en cuando a lo que a planificación y diseño se refiere.

El diseño de redes *backbone* está regido por 3 objetivos o principios de diseño:

- Caminos directos entre origen y destino.
- Componentes con una buena utilización.
- Uso de líneas de alta velocidad para lograr una economía de escala (el coste del ancho de banda no es lineal).

En apariencia estos principios son mutuamente contradictorios, por ejemplo si todo el tráfico es directo, tendremos una malla de líneas de baja velocidad. Si todos los componentes tienen una gran utilización, usualmente construiremos árboles con demasiados saltos. Si la red sólo tiene líneas de alta velocidad, tendremos mucha capacidad de sobra en la periferia. Sin embargo, es posible aprender a reconocer diseños en donde estos principios son obedecidos, y desarrollar estrategias para conseguirlos.

Para un mayor detalle en ejemplos de diseño que siguen un principio de forma excesiva sin atender a los demás, dirigirse a [1], (sección 8.2).

2.4.2 Estrategia MENTOR

MENTOR significa *Mesh Network Topological Optimization and Routing*. Originalmente MENTOR fue concebido como un algoritmo para diseñar redes de multiplexores basados en división de tiempos. Sin embargo, una vez modificado, se halló que es útil con otros tipos de redes. El algoritmo MENTOR es ahora en realidad es un conjunto de algoritmos de baja complejidad y de alta calidad para el diseño de *backbones*.

Es la idea del MENTOR inicial lo que aprovechan las versiones posteriores para adaptarlo a diferentes tipos de redes. De esta manera, podría hablarse de que no hay un algoritmo MENTOR, sino una filosofía *o estrategia* MENTOR. La idea básica es empezar con un árbol que conecte todos los sitios y luego aumentar el diseño del *backbone*, es decir la estructura troncal (columna vertebral) de la red, con enlaces adicionales. De esta forma, podemos dividir los algoritmos MENTOR en 4 etapas:

- Clustering: Selección de nodos para el *Backbone*
- Creación de la topología inicial de la red de malla
- Incorporación de enlaces directos para formar la red de malla
- Topología de acceso al *backbone*

Las dos primeras etapas son comunes a todos los algoritmos MENTOR, y la tercera es donde el diseño se hace más específico para distintos tipos de red según el objetivo o las restricciones impuestas.

2.4.2.1 Clustering: Selección de nodos para el *Backbone*

En la primera etapa de todo diseño MENTOR, tenemos que dejar claro qué nodos pertenecen al *Backbone* y cuales no. El método original que se pensó para esta tarea fue el ***threshold clustering*** (clustering umbral). Otros procedimientos son el K-Medias clustering, y el *Automatic Clustering*.

Funcionamiento del *threshold clustering*:

Los sitios *Backbone* se entienden como puntos de agregación del tráfico en la malla de encaminamiento. Este procedimiento está fuertemente parametrizado por un radio RPARM y un límite de peso WPARM que determinan dicho *backbone*. Se parte de tener el peso de todos los sitios, normalizado:

$$NW(N_i) = \frac{W(N_i)}{C}$$

, donde el peso de un nodo es la suma del flujo que sale del nodo más el flujo que entra.

Los sitios que tengan suficiente tráfico, es decir:

$$NW(N_i) > WPARM$$

se convierten automáticamente en sitios *Backbone*. Todos los nodos restantes que no cumplan la restricción del peso y estén cerca de un sitio *backbone* se convierten en nodos pendientes de ellos, nodos finales (*end sites*). Aquí el que un nodo E esté cerca de un sitio significa que:

$$\frac{Cost[E][N_i]}{MAXCOST} < RPARM$$

$$MAXCOST = Max_{i,j} Cost[N_i][N_j]$$

Tras este paso, la clasificación de nodos puede quedar, por ejemplo, como en la figura siguiente: los cuadrados grandes representan sitios *backbone*, los pequeños sitios normales, y los círculos alrededor de los *backbones* indican el radio de pertenencia de los nodos finales que se encuentren dentro al *backbone* correspondiente. El triángulo pequeño indica el centro de masas, pero no representa a ningún nodo.

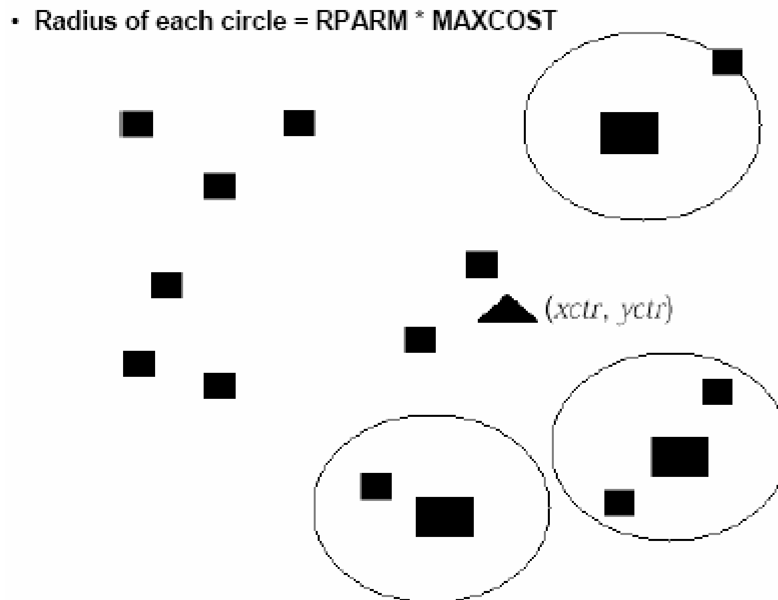


Figura 2-9. Threshold Clustering

[3]

Si tras todo eso, quedan sitios sin ser *backbone* o sin asignar como nodos finales a uno de los *backbones* (es decir, que escapan a los dos parámetros), se continúa eligiendo sitios *backbone* entre ellos, hasta que todos los nodos queden cubiertos. Para hacer esto se les asigna a cada sitio una figura de mérito. Este mérito se calcula con las coordenadas de los nodos, calculando primero un centro de pesos, que se define así:

$$xctr = \frac{\sum_{n \in N} x_n \cdot Weight_n}{\sum_{n \in N} Weight_n}$$

$$yctr = \frac{\sum_{n \in N} y_n \cdot Weight_n}{\sum_{n \in N} Weight_n}$$

El mérito de cada nodo se define entonces:

$$merit_n = \frac{1}{2}\frac{(\max_dist_ctr - dist_ctr_n)}{\max_dist_ctr} + \frac{1}{2}\frac{Weight_n}{Weight_\max}$$

Esta función de mérito da una idea de la proximidad de cada nodo al centro de pesos. De entre todos los nodos que faltan por clasificar, se escoge el de mayor mérito como nodo *Backbone*, y se vuelve a repetir el proceso de asignación de nodos finales a los *backbones*.

El resultado final será algo similar a la siguiente figura:

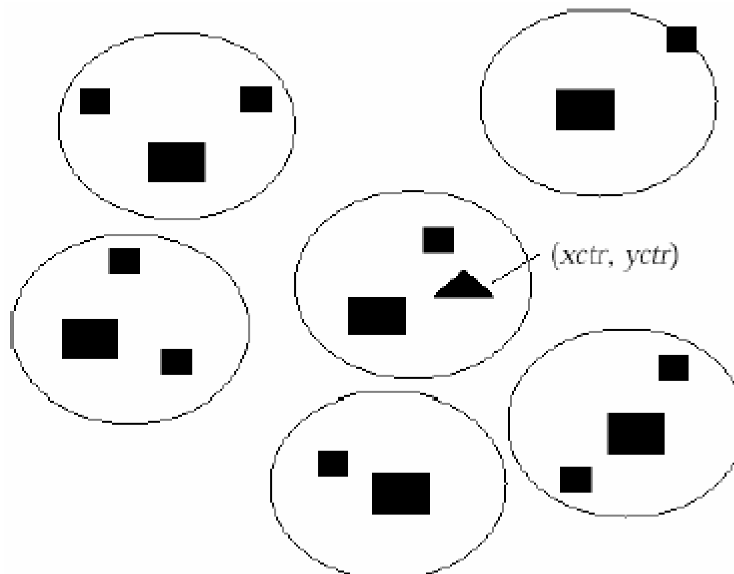


Figura 2-10. Fin del Threshold Clustering

[3]

Con esto, tenemos los nodos de la red clasificados en nodos *Backbone* y nodos finales, y la relación de asignación entre ellos.

Nota:

Hay casos en los que el threshold no trabaja bien: cuando todos los nodos tienen un tráfico muy parecido, por ejemplo. Sin embargo, es el más parametrizado, ideal para hacer un conjunto grande de diseños con parámetros graduales de forma que el planificador pueda seleccionar el diseño más conveniente. Para más información sobre los otros métodos dirigirse a [1], (sección 8.7).

2.4.2.2 Creación de la topología inicial de la red de malla

El siguiente paso es construir un árbol inicial que será el que lleve el tráfico que no vaya por enlaces dedicados. El árbol que se construye es un híbrido Prim-Dijkstra un tanto peculiar, llamado “Restricted Prim-Dijkstra”.

Antes de pasar a ello necesitamos calcular un nodo *backbone* para que sea el centro o mediana de la red. Esto se consigue seleccionando el de menor *momento*. El momento de un nodo se define como:

$$Moment(n) = \sum_{n' \in N} dist(n, n') * Weight_{n'}$$

En la red de la figura anterior, la mediana es el nodo *backbone* que cae dentro del círculo que contiene (xctr, yctr). Esta mediana es la que usaremos para ser el nodo inicial del que parta el algoritmo Prim-Dijkstra. Como ya se expuso en la sección 2.1.4, recordamos que este algoritmo viene parametrizado por un solo parámetro, α, entre 0 y 1, que indica su proximidad con un árbol de Prim, y uno de Dijkstra (típicamente una estrella).

La única peculiaridad de esta versión de Prim- Dijkstra aplicada a *Backbones*, es que debe cumplir la restricción: "sólo se permite que los puntos interiores del árbol sean los sitios *backbone*". En la práctica, para conseguir esto crearemos un árbol prim-dijkstra primero entre los *backbones* con inicio en la mediana, y luego anexaremos a él cada uno de los árboles prim-dijkstra con inicio en cada *backbone* y que engloban sus nodos terminales.

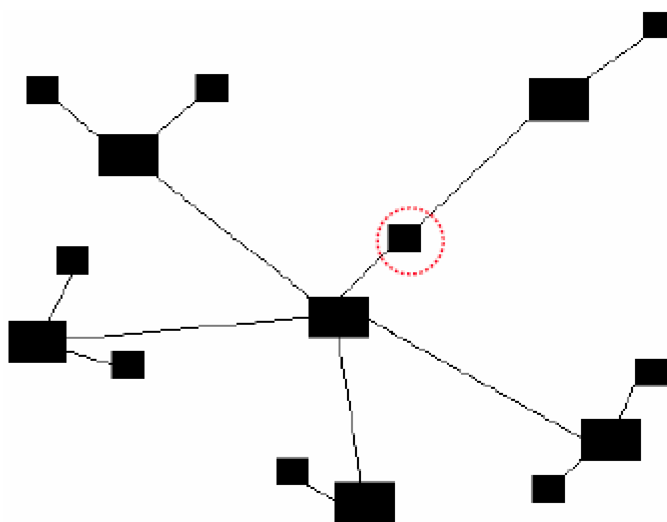


Figura 2-11. Prim-Dijkstra partiendo de la mediana

(notar que el nodo indicado no cumple la restricción)

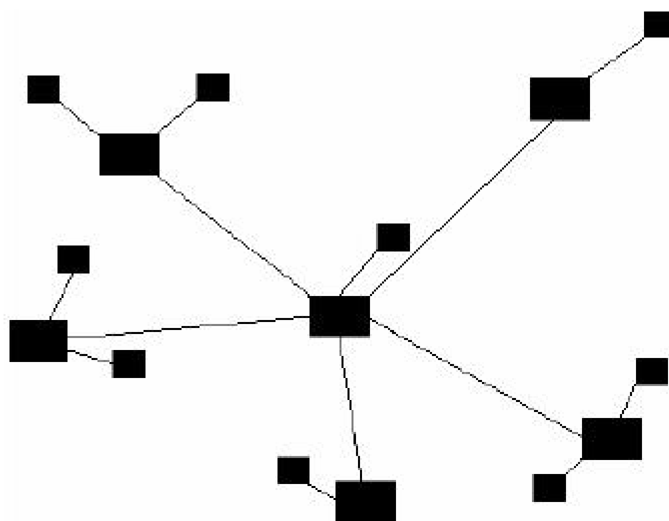


Figura 2-12. Árbol Restricted Prim-Dijkstra final

[3]

2.4.2.3 Incorporación de enlaces directos para formar la red de malla

Este es el paso más específico para cada tipo de MENTOR, de forma que según el diseño MENTOR que se elija, el procedimiento en este paso variará mucho. Describamos ahora las distintas posibilidades, y posteriormente se explicará el método escogido para este proyecto:

2.4.2.3.1 Incorporación de enlaces directos con ruteo basado en *homming*

En este procedimiento, creado para el primer MENTOR, se trabaja con todos los nodos del árbol Prim-dijkstra obtenido anteriormente.

El ruteo basado en *homming* requiere primero definir una secuencia de los pares de nodos. Esto se hace en un orden del nodo más externo hacia el más interno. Una vez secuenciados tendremos una lista con los pares de nodos que tienen mayor numero de saltos (*hops*) en primer lugar.

El segundo paso se conoce como "*homming*". Para cada par de nodos (N_1, N_2) que no son vecinos en el árbol, se selecciona un "*home*". Esto es, un nodo intermedio N_3 en la ruta de un nodo al otro, tal que, de todos los nodos intermedios posibles, la suma de costes $Cost(N_1,N_3)+Cost(N_3,N_2)$, sea la menor.

El tercer paso consiste en considerar cada par de nodos una sola vez. Se añadirá un enlace si éste va a transmitir el suficiente tráfico que lo justifique. Se trata de asegurar componentes bien utilizados, por lo que se añadirá al enlace entre N_1 y N_2 a la red si se cumple que:

$$util(enlace) > util_{\min}$$

Si no se cumple, entonces no se añade dicho enlace, y el tráfico de N_1 a N_2 se suma a los tráficos entre los nodos intermedios (entre los nodos extremos de este enlace y el "*home*").

La idea del algoritmo MENTOR es la de juntar tráfico que pueda ser usado para justificar enlaces que conecten nodos que están separados por varios hops del árbol. Si el tráfico entre N_1 y N_2 no puede justificar un enlace directo, éste se desvía a través del nodo "*home*" *H*.

El número de enlaces directos está gobernado directamente por el parámetro $util_{min}$ e indirectamente por el parámetro de construcción del árbol α. Mientras sea menor el valor de $util_{min}$, es más fácil para el algoritmo añadir enlaces directos. El efecto de la selección del valor de α es más sutil. Si α=1 entonces se construiría una estrella a partir de la mediana. En ese caso, se introducirán sólo los enlaces directos entre pares de nodos que tengan suficiente tráfico para justificar un enlace no añadido por otros nodos. Si α es pequeño (p.e. 0.2 ó 0.3) y el algoritmo de selección de nodos *backbone* ha introducido un pequeño número de nodos *backbone*, entonces habrá más tráfico que se logre reunir, más enlaces directos que formar.

Todo este algoritmo está diseñado, como se pensó el algoritmo MENTOR en un principio, para multiplexores. El algoritmo para ruteadores incluye asimismo esta parte, aunque además necesita de unos ajustes complementarios, y que dependen en gran parte del cargador de tráfico usado, que se describen en la sección 2.4.2.6.

2.4.2.3.2 Incorporación de enlaces directos en red de malla mediante ISP y enlaces "1-commodity".

Este procedimiento se creó con el algoritmo MENTOR II, aunque sus conceptos pueden utilizarse en los algoritmos posteriores, AMENTOR y MENTour.

El objetivo del MENTOR II al crearse fue obviamente mejorar el MENTOR, especialmente ciertos problemas que surgían con relación al robo de tráfico por los enlaces directos, así como mejorarlo para que fuera más adecuado para ruteadores sin que un planificador de red tuviera que hacer modificaciones en el código, ni tuviera que introducir mecanismos de *balanceo*. *(Ver ap. 2.4.2.6.3)*. El resultado del MENTOR II va a ser una serie de longitudes asignadas a cada nodo de la red, que son usadas por los mecanismos de encaminamiento de los routers que están preparados para ello. *(Ver ap. 2.4.2.6.1)*.

Aquí hay dos tipos de adición de enlaces: los directos, que se realizan sólo entre nodos *backbone*, para los cuales se calcula la longitud óptima de ruteo gracias al algoritmo ISP, y los *1-commodity*, que se dan cuando uno de los dos nodos del enlace es un nodo terminal.

Al principio mediante el algoritmo ISP se obtienen los datos de la red, que son las matrices de ruteo y el coste total por costes mínimos entre cualquier par de nodos. Estos dos datos se denominan sp_pred (ruteo) y sp_dist (costes entre cada par de nodos).

El segundo paso es incorporar los *enlaces 1-commodity* al diseño. Estos son como enlaces directos, pero asegurándose (por la longitud con que se diseñan), que sólo pasará por ellos el tráfico interno entre los dos nodos del enlace.

$$Length(L) = sp_dist_{tree} - 1$$

La forma de conseguir esto es ponerle una distancia igual a la distancia que tendrían que recorrer en el camino indirecto, menos uno. De ahí su nombre. Este ingenioso mecanismo evita que cualquier nodo que no sean ellos dos evite usar esa ruta porque siempre la verá más cara.

El tercer paso es considerar los requerimientos en el *backbone*. Se secuencian los pares de mayor a menor de acuerdo a la distancia de la ruta más corta (*shortest path distance*), y luego se estudia enlace a enlace mediante el algoritmo ISP el conjunto de pares de nodos *backbone* que podrían usar el nuevo enlace directo, y la longitud máxima que tendría que tener el enlace para usarlo (req_len). Una vez teniendo esta información, se elige la longitud que más interese, dejando fuera a los enlaces cuyo tráfico no se desea que pase por allí, y asimismo teniendo en cuenta la utilización con que se quedará el nuevo enlace, y el posible reordenamiento de la secuencia de pares.

Se puede notar que este proceso obtiene no solamente los enlaces y sus multiplicidades sino también las longitudes que necesita OSPF para recorrer la red. En resumen el algoritmo MENTOR II destaca por sus herramientas para "auditar" con un grado mayor de eficacia que en su predecesor, las longitudes necesarias para los parámetros respecto a utilización y topología que un planificador requiera.

2.4.2.3.3 Diseño aumentado del *backbone* para mayor confiabilidad

La confiabilidad se estudia en los dos últimos algoritmos MENTOR: el AMENTOR, y el MENTour. Una red es más confiable cuanto más segura sea desde el punto de vista de la conectividad ante la caída aleatoria de nodos. Esta confiabilidad se logra consiguiendo una red *backbone* biconexa (2-connected), de forma que si cae cualquier enlace o nodo, la red de *backbones* al menos siga conectada.

La diferencia entre uno y otro algoritmo estriba en que en el primero se pretende un aumento del *backbone* con coste total mínimo, partiendo de un árbol Prim-dijkstra de los *backbones*, y añadiendo los enlaces con coste mínimo sucesivos que vayan conectando la red. El

segundo sin embargo parte no de un árbol, sino de otra topología inicial, los tours TSP con árboles pendientes. Un tour es de por sí una estructura biconexa, por lo tanto la finalidad del proceso queda resuelta.

Funcionamiento del AMENTOR:

El *augmented*-MENTOR calcula cada posibilidad de conexión entre nodos *backbone* (reforzamiento de la conectividad), estimándola según una figura de mérito dependiente del coste y la distancia. Lo que se pretende es usar la mínima cantidad posible de enlaces extras, y que resulten lo más eficaces en coste. Para esto se hace uso de Teoría de Grafos (ver apéndice), que nos brinda pistas respecto a la mejor manera de conseguirlo:

- Un componente conectado (como es el *backbone* tras haber hecho el segundo paso del algoritmo mentor correspondiente), se compone de subcomponentes biconexos, y puntos de articulación que los unen.
- Si entre dos componentes biconexos separados por un solo punto de articulación se crea un enlace, pasan a convertirse en un solo componente biconexo.
- El punto anterior no es efectivo si los nuevos enlaces son entre los nodos que son puntos de articulación. Puede que en algún caso se forme un componente biconexo mayor, pero no en todos los casos.

Con estas ideas, el procedimiento es simple:

1. Se calculan los subcomponentes biconexos del grafo y los puntos de articulación.
2. Se calcula una figura de mérito entre nodos de distintos componentes, que no sean puntos de articulación. La figura de mérito es la siguiente:

$$\frac{\cos t \ (n_1, n_2)}{dist \ (C_1, C_2)}$$

   Esta figura nos dará el coste, por componente biconexo, del enlace.
3. Se crea dicho enlace, con el resultado de que tendremos un componente biconexo menos en la red, que habrá sido “engullido” por otro.
4. Si la red ahora es biconexa, se termina el algoritmo, si no, se vuelve al principio.

Funcionamiento del MENTour:

El algoritmo AMENTOR hace un buen trabajo al añadir algunos enlaces adicionales y hacer que el *backbone* sea biconexa si pocos enlaces adicionales satisfacen los requerimientos. Sin embargo, hay redes en donde no son suficientes pocos enlaces adicionales. El costo de una *backbone* biconexa construida a partir de dichas redes puede llegar a involucrar un gasto adicional considerable pues puede llegar representar un 10% a 20% de costo adicional en la red

Hay otra forma de producir *backbones* biconexas; se trata simplemente de empezar con ellos. Ésta es la idea del MENTour. Lo esencial de MENTour es que en lugar de construir un árbol Prim-Dijkstra como topología inicial, se construye un *tour* TSP (*Traveller Sales Person problem*), como está definido en la sección 1.2, para los nodos del backbone, y con árboles colgados para la red de acceso.

Existen diversas heurísticas para construir tours TSP's:

- o SIMP (*Simple Nearest-neighbour*) vecino simple más cercano,
- o TOUR_N (*Nearest-neighbour*) vecino más cercano
- o TOUR_F (*furthest neighbour*) vecino más lejano.

Esta construcción se controla con los parámetros (*algorithm*, *first_node*, α), donde algorithm controla el tipo de algoritmo (TOUR_N, TOUR_F O SIMP), first_node es el nodo del que parte dicho algoritmo, y α se usa después de que el tour esté construido, para conectar los sitios terminales a él, de forma que si un sitio *e* se quiere conectar al tour *backbone*, lo hará a través del nodo *b* tal que minimice la expresión:

$$dist(e,b) + \alpha * dist(b, median)$$

Para más información respecto a estos algoritmos, dirigirse a [1], (3.4).

2.4.2.4 Topología de acceso al *backbone*

Una vez que se tiene todo el grafo conectado, diferenciado el *backbone* con los parámetros de utilización y coste requeridos, queda la posibilidad de modificar el acceso al *backbone*. Recordemos que ahora mismo, salvo en el caso del MENTour, al *backbone* se accede mediante árboles Prim-Dijkstra parametrizados con un α igual al que se usa para crear el *backbone* inicial.

No hay dificultad alguna, por la flexibilidad del algoritmo, en especificar ahora otra forma de acceso si se requiere algo más específico. Este acceso puede ser de 3 formas:

- o Acceso Prim-Dijkstra con un α diferente.
- o Acceso con árboles Esau-Williams.
- o Acceso MSLA (*MultiSpeed Local Access*).

2.4.2.4.1 Acceso Prim-Dijkstra con un α diferente

Usualmente se denomina al primer modo de acceso *"en estrella"* cuando se enlazan directamente los nodos terminales con el nodo *backbone* que les corresponde de acuerdo al *cluster* en el que se encuentran (dicha creación de los *clusters* se da en la primera etapa de cualquiera de las versiones de MENTOR con cualquiera de los métodos de *clustering*).

Esto equivaldría a usar un prim-dijkstra con α=1. Por ello, para hacer este primer modo de acceso más general se ha optado aquí por generalizar, a fin de dar mayor versatilidad, el nombre del acceso, indicando que se puede elegir cualquier otro α diferente del usado inicialmente para la creación del *backbone* una vez que éste está definido.

Cuando se utiliza topología estrella para conectar a los nodos terminales con los nodos del *backbone*, muy probablemente el porcentaje de utilización de los enlaces sea demasiado bajo y esto vaya en contra de la premisa de tener componentes bien utilizados. Por eso, lo normal es usar un acceso de árbol mediante un α<1, o como los que resultan de los siguientes métodos.

2.4.2.4.2 Acceso con árboles Esau-Williams

Como se ha explicado (Sección 2.2), el algoritmo Esau-Williams es un algoritmo heurístico que resuelve problemas relacionados con la creación de un árbol de capacidad mínima (*Capacitated Minimum Spanning Tree* CMST), es decir, garantiza que el árbol que crea cumple con la restricción de capacidad.

Para tráfico homogéneo, el algoritmo Esau-Williams tiende a producir buenos resultados. Para un número muy grande de nodos, la tasa de fallo empieza a crecer.

2.4.2.4.3 Acceso MSLA (MultiSpeed Local Access)

Recordemos que el algoritmo MSLA construye árboles de acceso local *MultiSpeed* (Sección 2.3). Un acceso de este tipo puede ser conveniente cuando los requerimientos de capacidad en nodos diferentes se encuentran en rangos distintos. Es decir, si tenemos un grupo de nodos terminales por los cuales se necesita un tráfico de 10Kbps, y otros que requieren 100Kbps, la diferencia de rango de 10 nos indica que deberíamos usar líneas de diferentes capacidades para mejorar su utilización, con el consecuente ahorro en el coste para el diseño MENTOR final.

2.4.2.4.4 Consideraciones adicionales en el acceso local

Hay dos casos a considerar en el tema de acceso local:

- Primer caso:

  Supóngase que cada nodo terminal está conectado a un solo nodo *backbone*. Este caso se aplica cuando no se han usado los enlaces 1-commodity del MENTOR II. En este caso se tiene un problema de acceso local que puede ser optimizado con cualquiera de los algoritmos descritos anteriormente en esta sección.

- Segundo caso:

  MENTOR ha añadido un enlace entre un nodo *backbone* y un nodo terminal, o cuando MENTOR-II ha añadido un enlace 1-commodity.

  En este caso un nodo terminal *e* puede tener múltiples conexiones al *backbone*. Se podría tratar de añadir al nodo *e* a árboles de acceso múltiple, y cuidadosamente separar el tráfico que entra al *backbone* en *b1* del tráfico que entra en *b2*.

  Sin embargo, es mucho más fácil "promover" al nodo *e* para que sea parte del *backbone*. Esto es consistente con la idea de que el *backbone* de la red es la parte que está conectada en malla.

2.4.2.5 Esquema: conjunto de algoritmos MENTOR

Se resume a continuación en una tabla las posibilidades diferentes que ofrece el MENTOR tratado como un conjunto de algoritmos, de forma que se puedan ver claramente las distintas combinaciones que se pueden llevar a cabo:

Pasos	Algoritmos posibles	MI	MII	A-M	MT
1. Selección de nodos para el Backbone	*Threshold Clustering*	√	√	√	√
	K-medias *Clustering*	√	√	√	√
	Clustering Automático	√	√	√	√
2. Creación de la topología inicial de la red de malla	Árbol Prim-Dijkstra	√	√	√	
	Backbone a base de *tours*				√
3. Incorporación de enlaces directos para formar la red de malla	Enlaces directos con ruteo basado en *Homming*	√			
	Enlaces directos mediante ISP e incorporación de enlaces "*1-commodity*"		√	√	√
	Diseño aumentado del *Backbone* (biconexo) para mayor confiabilidad			√	
4. Topología de Acceso Local al Backbone	Prim-Dijkstra con α distinto	√	√	√	√
	Árboles Esau-Williams	√	√	√	√
	MSLA	√	√	√	√

Tabla 2-3. Conjunto de algoritmos MENTOR

[5]

- M: MENTOR
- MII: MENTOR II
- A_M: AMENTOR
- MT: MENTour

2.4.2.6 Problema del encaminamiento en redes mesh

Hemos visto que en redes mesh no hay un solo camino entre dos puntos. Es por ello importante la forma que tenga la capa de red de hacer el encaminamiento, porque esto afectará evidentemente a la eficiencia de nuestro diseño. Para dar una idea del problema de encaminamiento, primeramente se explicará de forma breve y sencilla de qué forma funciona la capa de red en una red de conmutación de paquetes, centrándonos únicamente en los mecanismos que nos afectan como diseñadores de red (sección 2.4.2.6.1). En la siguiente sección se dará una solución a la adaptación del MENTOR a este tipo de redes. En la última sección se exponen otros problemas, también directamente relacionados con el encaminamiento, a los que puede dar lugar el tráfico en el algoritmo MENTOR.

2.4.2.6.1 Cargadores de tráfico y protocolos de encaminamiento

Hay diversas clasificaciones generales de los protocolos de encaminamiento. Una clasificación está basada en si las decisiones de encaminamiento se realizan independientemente para cada paquete de datos enviado (servicio de datagramas) o si la ruta es determinada cuando se crea una conexión, de forma que todos los paquetes siguen el mismo camino (servicio de circuito virtual). Otra clasificación de estos protocolos está basada en qué nodos de la red seleccionan las rutas. En un protocolo de encaminamiento distribuido, cada nodo puede realizar tareas de enrutamiento. En un protocolo centralizado se asigna un nodo para seleccionar rutas para todo el tráfico.

Además, los protocolos de encaminamiento pueden clasificarse en función de si son capaces de responder a cambios en la densidad de tráfico o a la topología de la red. Así, se tienen algoritmos adaptativos que tienen en cuenta las propiedades dinámicas del tráfico y algoritmos estáticos, que toman las decisiones de encaminamiento independientemente de los cambios que se produzcan en la red. Existen varios protocolos:

- Border Gateway Protocol (BGP).
- Enhanced Interior Gateway Routing Protocol (EIGRP).
- Interior Gateway Routing Protocol (IGRP).
- Intermediate System-to-Intermediate System (IS-IS).
- Open Shortest Path First (OSPF).
- Routing Information Protocol (RIP).

El algoritmo más conocido para encaminamiento que se utiliza en ruteadores IP es OSPF. *Open Shortest Path First* es un método de ruteo en donde a cada enlace de una red se le da un a longitud y dichas longitudes se usan para seleccionar rutas. Cuando el control de la red necesita decidir cómo transmitir tráfico entre A y B, la red calcula la ruta más corta entre A y B y envía el tráfico sobre esa ruta. Si hay varias rutas más cortas, entonces, dependiendo de la implementación, la red escoge una o divide el tráfico entre varias. Aquí nos centraremos en este algoritmo, OSPF, como protocolo de encaminamiento. Para más información sobre los otros protocolos de encaminamiento puede dirigirse a [2].

Cuando se habla de la capa de red, además de los algoritmos de encaminamiento, hay que tener en cuenta que los routers y multiplexores están diseñados para funcionar con unos determinados cargadores de tráfico. Un cargador de tráfico es código, específico de una arquitectura, o específico de fabricante, que toma un diseño de red (bien producido por MENTOR, cualquier otro algoritmo, o a mano), y carga el tráfico de acuerdo con los algoritmos usados en la actual arquitectura y/o equipamientos. Hay varias posibilidades, entre las cuales las más significativas son:

o SRMH

Un cargador SRMH (*Single-route, minimum hop*) o de Ruta Simple de Mínimos Saltos para una red, basa sus rutas en la longitud en *hops* entre los nodos. Calcula una ruta simple (sencilla, o sola) de número de *hops* mínimo entre cada par de nodos y la usa para cargar el tráfico. Si el tráfico satura a la ruta fija, ésta es descartada. Si existen varias rutas con igual número de saltos, entonces se supone que el cargador es libre de escoger cualquier ruta, o dicha elección tendrá que ser tratada con algún mecanismo por el programador de los protocolos de los routers.

Los diseños creados con SRMH producen redes frágiles que no se adaptan a la capacidad de la red. Hasta ahora no existen algoritmos realmente efectivos para su implementación.

o FSMH

Un cargador FSMH (*flow-sensitive,minimum hop*) o Sensible al Flujo con Mínimos Saltos, toma un requerimiento de tráfico y lo carga en un camino de mínimos saltos usando sólo los enlaces con suficiente capacidad para llevar ese tráfico. La capacidad de los enlaces es calculada como la diferencia entre la capacidad real y el flujo que corre actualmente por el enlace. Si no se encuentra un camino, el tráfico es bloqueado.

Este cargador funciona generalmente (no siempre) mejor que el SRMH, y se utiliza sobre todo para trabajar con multiplexores.

o SRMD

Un cargador SRMD (*Single-route, mínimun distance*) o de Ruta Simple de Mínima Distancia para una red, usa longitudes asignadas a los enlaces para producir rutas. Calcula una ruta simple (sencilla, o sola) de distancia mínima entre cada par de nodos y la usa para cargar el tráfico. Si existen varias rutas, entonces se supone que el cargador es libre de escoger cualquier ruta. Si el tráfico satura a la ruta fija, ésta es descartada. Se supone que el mecanismo para descartar enlaces está fuera del control del diseñador de la red.

Este cargador funciona muy bien con OSPF, y se utiliza sobre todo para trabajar con routers.

Conclusiones:

Aunque MENTOR está preparado para hacer frente a redes con multiplexores, en este proyecto nos hemos centrado en las redes de encaminamiento de paquetes IP. Dentro de estos conviene tener el máximo control posible del ruteo, y esto se consigue utilizando cargadores SRMD, y utilizando la distancia que se les pasa como forma de controlarlo. Por lo tanto, utilizaremos la especificación OSPF con cargadores SRMD. En el caso de tener cargadores SRMH (o protocolo RIP, que también funciona por saltos), los problemas que se van a presentar y solucionar en estas secciones, no pueden abordarse.

La especificación OSPF especifica cuál tráfico es ruteado por un algoritmo de la distancia más corta. Por tanto, es posible reducir el problema a la siguiente pregunta ¿Cómo se diseñan redes para la distancia mínima? La solución a esto es asignar distancias iniciales a cada enlace equivalentes al coste del mismo, y modificarlas para satisfacer los requerimientos del ruteo.

Hay una diferencia importante entre redes a base de ruteadores y las redes a base de multiplexores. No se espera diseñar redes de ruteadores con el 100% de utilización. Más bien se desea diseñar para el 40% ó 45%. Como consecuencia, si el cargador SRMD hace algunos errores es posible tener un diseño con enlaces con utilización del 55% al 60%. Esta es más utilización de la que se desearía, pero no representa problema alguno. Los problemas se presentarían si el ruteo se dirige el tráfico hacia el enlace directo y entonces este enlace tendría que transmitir un mayor tráfico que para el cual fue diseñado.

Estos problemas se solucionan compensando la distancia de los enlaces directos (aumentándola), y en último extremo, *balanceando* varios enlaces, como se verá en los siguientes apartados.

2.4.2.6.2 Algoritmo MENTOR para ruteadores

Esta etapa complementa a la etapa correspondiente en la versión del algoritmo MENTOR para multiplexores. Este complemento se debe a que cuando se trabaja con ruteadores es

necesario añadir algunos cálculos que hagan posible tratar de encauzar el tráfico mediante "longitudes" que se le pasan a los routers, asignando las longitudes más largas a rutas en las que se desea encauzar menor cantidad de tráfico.

Supongamos que tenemos calculado el número de saltos entre dos nodos cualquiera en el árbol de prim-dijkstra (*hops*). Cuando se añade un enlace directo entre 2 nodos, se desea que envíe todo el trafico directo en lugar que tener al tráfico saltando por varios *hops* a través del árbol. Como consecuencia, se escogen los siguientes valores:

- Length(enlace (N_1,N_2) de árbol prim-dijkstra)=100
- Length(enlace directo)=100 +90*(hops(N_1,N_2)-1)

Con la asignación de estas longitudes, entonces el algoritmo de carga de tráfico escogerá enrutar el tráfico por enlaces directos entre dos nodos y al mismo tiempo evitar que se saturen dichos enlaces al dejar longitudes lo suficientemente altas para que atraigan el tráfico hacia el resto de los enlaces del árbol Prim-Dijkstra.

Veámoslo con un ejemplo:

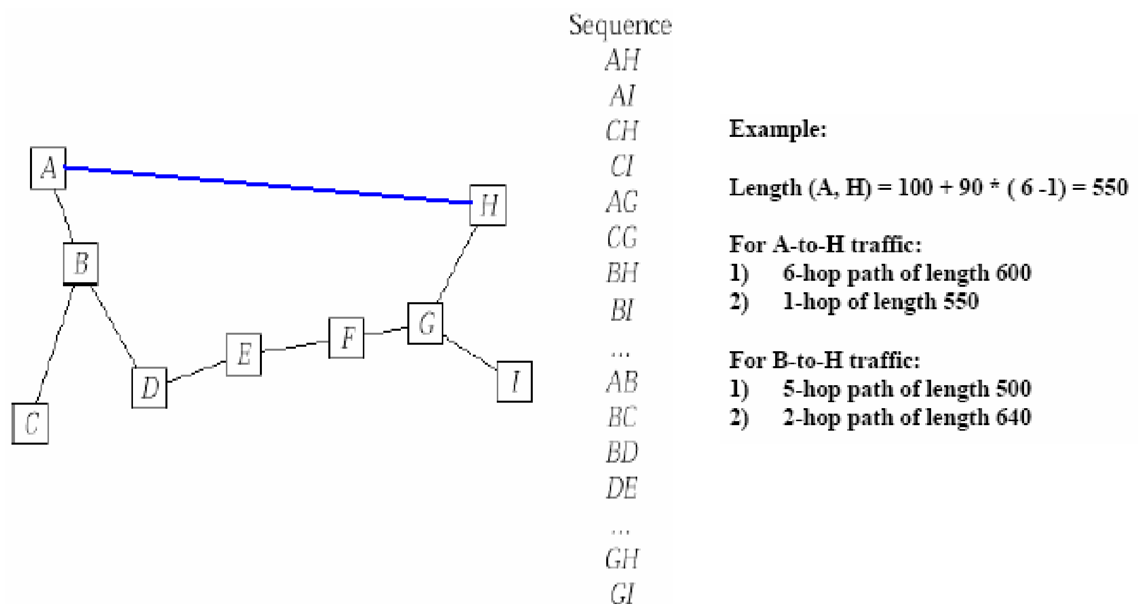


Figura 2-13. Ejemplo de enrutamiento respecto al tráfico

[3]

Al añadir el enlace A-H se piensa en aligerar el tráfico que todas las rutas mas largas (tal y como vienen ordenadas en *Sequence*) llevan por el camino original único de B a G. Pero lo que se quiere evitar es que por tener un número menor de saltos, se sobrecargue este enlace dejando el camino original con una baja utilización. (Esto es lo que pasaría asignando al enlace directo un valor igual a los demás).

Asignando longitudes a los enlaces tal y como se ha descrito, se comprueba en el cálculo de la figura que el tráfico de B a H seguirá enrutándose por el camino original en el árbol, ya que la longitud yendo por el enlace directo resulta más cara.

Al efectuar estos cambios hay que tener en cuenta que el cargador de tráfico con que funcionen los routers va a delimitar su correcto funcionamiento, puesto que van destinados a la capa de ruteo, en este caso se requeriría un cargador tipo SRMD o similar. Una breve explicación a este respecto se da en el siguiente apartado.

2.4.2.6.3 Problemas del MENTOR I relacionados con el tráfico de la red

Supóngase que el tráfico total de la red es T y que se tienen dos enlaces disponibles con capacidad C y 32C. Entonces podemos caracterizar el tráfico de varios valores de T como se muestra en la siguiente tabla:

T	Red
4C	MST de enlaces de baja velocidad
8C	árbol Prim-Dijkstra
16C	árbol con pocos enlaces directos
24C	Red de malla
64C	MST con enlaces de alta velocidad

Tabla 2-4. Tráfico según el tipo de red

Si la red es dispersa, entonces no hay demasiadas rutas y las redes MENTOR funcionan bien con el ruteo. Lo más probable es que la carga de los enlaces esté diseñada al 50% y que todas las rutas más cortas sean únicas. Las longitudes de estos enlaces podrían ser manejadas fácilmente para un cargador de tráfico OSPF.

Los problemas se presentan si se incrementa el tráfico en el siguiente rango. Cuando T=24C se pueden tener redes *backbone* bastante densa y podría llegarse al caso en donde al asignarse el tráfico utilizando pesos basados en *hops*, entonces el esquema de ruteo se rompe por completo ya que los enlaces podrían estar con un índice de utilización muy cercano o superior al 100%.

Para estos casos, es posible hacer estos diseños factibles por medio de un proceso complicado al que se le denomina "*balanceo*". Este proceso consiste en incrementar la longitud de los enlaces sobrecargados, acortar enlaces subutilizados, y, como último recurso, añadir y borrar capacidades de enlaces que resisten mediciones más sutiles. Sin embargo, tales procedimientos tienen a ser bastante difíciles de llevar a cabo y tienen un mayor riesgo de errores en los cálculos. Es más fácil introducir una nueva versión de MENTOR que tiene conocimiento de la capa de ruteo, dicha versión se conoce como MENTOR-II.

2.4.3 Propuesta de adaptación *multispeed* de AMENTOR

En esta sección se propone una extensión para *multispeed* del AMENTOR, con tres objetivos:

1) Probar que el concepto MENTOR no es un conjunto estático de algoritmos, sino una implementación *flexible* centrada en la idea de la diferenciación de un *backbone* dentro de una red.

2) Presentar una sugerencia respecto al *multispeed* en el conjunto de algoritmos MENTOR.

3) Aprovechar los conocimientos obtenidos con el resto de implementaciones anteriores a fin de crear un algoritmo más completo.

El punto central de la extensión es añadir al AMENTOR implementado la etapa de acceso local mediante MSLA, que se indicaba la tabla 2.4 [5] como una opción disponible (aunque no desarrollada). El problema es que si bien esta última etapa es bastante clara, hay que adaptar el resto del algoritmo para que pueda trabajar con varias velocidades. Se van a exponer ahora las decisiones tomadas para la adaptación de cada una de las etapas del AMENTOR, en su versión mediante el clustering de umbral (Threshold Clustering) hacia los requerimientos de la multivelocidad.

El concepto *multispeed* implica que ahora en el diseño se puede elegir una línea u otra para cada enlace. ¿En función de qué se eligen estas líneas? La forma general de decisión será intentar que elegir la más barata que cumpla las restricciones de utilización. Para cada nodo se mide aproximadamente un tráfico que irá destinado a los demás. Para cuantificar esto, hacemos la aproximación lógica de que cada nodo tiene un *peso*, que es el tráfico destinado a un nodo *central*, siguiendo un modelo parecido al que se da en la realidad (la estructura jerarquizada de las redes permite aproximar un nodo destino de la mayor parte del tráfico en cualquier sección de la red, sobre todo en los niveles de jerarquía inferiores) . Por lo tanto, tendremos tres variables extra: las capacidades de cada línea (*capacity*), la utilización máxima (*utilization*), y el peso de cada nodo hacia un nodo central (*weight*).

El primer paso es adaptar el *Threshold Clustering*. Este nos separará el conjunto de nodos *backbone* y nodos terminales, nos dará el nodo central respecto al cual irán destinados los pesos de los nodos, y la asignación de nodos terminales a cada nodo *backbone*

No hay que confundir aquí los pesos de los nodos que nos servirán para elegir el tipo de líneas (*weights*) de los pesos calculados por el *Threshold Clustering* para decidir la importancia y el flujo relativo de cada nodo respecto a los demás, que recordemos venían dados por la siguiente ecuación:

$$NW(N_i) = \frac{W(N_i)}{C}$$

Aquí se nos plantea la primera duda. Si tenemos varias capacidades disponibles, ¿con cuál normalizamos los pesos de los flujos relativos? Teniendo en cuenta que puede ser elegido cualquiera de los tipos de línea, lo más lógico es escoger aquel con mayor capacidad, ya que sólo de esta forma podemos asegurar que no obtengamos un "peso normalizado" mayor que la unidad a no ser que no haya capacidades disponibles suficientes para resolver el problema.

Aparte de este cambio, los únicos restantes son tener en cuenta en el cálculo del nodo *backbone* más próximo (respecto al coste) para cada nodo terminal que hay que elegir el más barato para cualquiera de los tipos de línea disponibles.

El próximo paso es crear la topología inicial de malla. Las opciones disponibles según la bibliografía utilizada en el tema se restringían a crear un árbol Prim-Dijkstra para todos los tipos de MENTOR, y la topología basada en *tours* para el MENTour. El problema de utilizar el algoritmo Prim-Dijkstra para varias velocidades es que los costes de cada enlace pueden ser varios (tantos como tipos de enlace), y eso requiere decisiones del tipo ¿escojo el árbol de mínimo coste sólo con las capacidades de líneas más baratas? ¿Tengo en cuenta el peso relativo de cada nodo backbone al escoger dicho tipo de línea? ¿tengo en cuenta la utilización, o la

uniformidad de líneas en la construcción del *backbone*?. Ante tal cantidad de preguntas que Prim-Dijkstra deja sin contestar, aquí se ha decidido ser coherente con el propósito del diseño, y se ha escogido otra opción, el MSLA.

Formando la topología inicial entre *backbones* con MSLA se puede decidir qué tipo de línea escoger en cada momento respecto a los parámetros de utilización, y los pesos de los nodos *backbone*, y posteriormente, en la etapa de acceso local se resolverá el problema respecto a los nodos terminales. Primero se calcula el nodo central por medio de los momentos, con el mismo procedimiento mostrado en la sección 2.4.2.2, ya que en dicha elección no influye para nada el hecho de tener distintas capacidades disponibles. Después, al pensar en el dimensionamiento que se va a llevar a cabo con el MSLA, recordamos que en una etapa posterior, a cada nodo backbone le va a llegar cierta cantidad de tráfico (caudal) de los nodos terminales que tiene asignados. Para tener el cuenta el peso de éstos en la red *backbone*, se suman los pesos de los nodos asignados a cada *backbone* (obtenidos en la etapa de clustering), al peso de dicho *backbone* en cuestión, para que el dimensionamiento MSLA sea lo más fiel posible a las necesidades de la red completa.

En la siguiente etapa, la propia del AMENTOR, el diseño aumentado se logra con algoritmos que corresponden más a teoría de grafos que a tráfico entre nodos, así que los cambios que implica el *multispeed* no determinan ninguna decisión importante. Recordemos que se calculaba la distancia entre cada par de nodos en función del número de bloques intermedios que les separaran, y se buscaba ir añadiendo al diseño las conexiones mínimas entre diferentes bloques. En estas conexiones el objetivo principal es obtener el mínimo coste, así que no se ha tenido en cuenta el tráfico, sobre todo teniendo en cuenta que el dimensionamiento del mismo ya es suficiente con el MSLA del *backbone*, y por lo tanto cualquier enlace adicional no implica ninguna necesidad extra en ese respecto[2].

En último lugar, queda el acceso local, que será resolver un problema MSLA para cada entorno del conjunto (*backbone*- nodos terminales asignados a él). Aquí se decidirá el dimensionamiento de cada enlace terminal, y uniendo esto a la solución obtenida en la etapa anterior respeto al *backbone* aumentado y dimensionado, habremos obtenido lo que se pretendía, un diseño total que cumple con las exigencias *multispeed*.

2.5 Generación de tráfico y tarifas.

En el diseño real de redes, el 90% de los recursos y el tiempo se dedican a la compilación la preparación adecuada de los archivos de datos. Con los generadores de tráfico y costes podemos reducir esto al 50% o quizá incluso al 25%.

Los generadores nos dan una solución que:

- Nos permite crear un ilimitado número de problemas de diseño interesantes para nuestros fines.
- Reduce la masa de problemas de unos millones a unos miles.
- Nos evitan las preocupaciones acerca de la seguridad de los datos reales así como de su privacidad.
- Ayudan a estudiar un rango más extenso de problemas que los de los sistemas reales.

[2] En este aspecto hay una cuestión que puede resultar útil respecto al ahorro de coste total del diseño. Esta cuestión se describirá en el capítulo 5, como una posible línea futura de desarrollo.

Estas ventajas los convierten en otra herramienta básica para cualquier planificador o diseñador de red. Se necesitan entre otras cosas para testear todas los mecanismos de resolución de redes y topologías, sin incurrir en los cuantiosos costes que supone el tratamiento de datos reales, pero sin apartarse de la forma real de los casos que puedan surgir a la hora de ponerlos en práctica.

2.5.1 Generadores de Tráfico

El fin de un generador de tráfico es crear una *matriz de tráfico*, es decir, una matriz que nos indique el tráfico ofrecido que fluye desde un nodo a cualquier otro nodo. Una matriz de tráfico será cuadrada de longitud igual al número de nodos, y las filas representan los nodos fuentes mientras las columnas los nodos destino del tráfico.

Los generadores de tráfico pueden ser muy sencillos, teniendo en cuenta que después de la generación se suelen pasar por un tratamiento de normalización. Su complicación depende del grado de realismo que se quiera simular. Vamos a ver aquí una pequeña evolución desde un generador básico a un generador bastante completo.

2.5.1.1 Generador Uniforme

En primer lugar, la solución evidente puede ser un generador de tráfico uniforme. Este es el modelo más simple, que define:

$$\gamma_{ij} = K \quad \forall i,j$$

Es evidente que el tráfico uniforme no es realista en absoluto en la gran mayoría de las redes. Es usado generalmente para funciones de testeo.

2.5.1.2 Generador aleatorio simple

Aquí el tráfico es aleatorio, es decir, tiene una cierta distribución entre un valor mínimo y otro máximo. Se escoge la distribución que más interese, puede ser uniforme, normal, etc.

$$\gamma_{ij} = U(\gamma_{\min}, \gamma_{\max}) \quad \forall i,j$$

Generador aleatorio simple con distribución uniforme

Sigue sin ser realista aunque es más interesante para los problemas de redes. Podría servirnos de ayuda más adelante una cierta aleatoriedad, pero aún faltan todos los parámetros que caracterizan el tráfico entre sitios reales.

2.5.1.3 Generador en función de la población y la distancia

El tráfico (Traf(i,j) entre i y j) está usualmente determinado por la población de los sitios (sea medida en personas, en computadoras, etc.) y la distancia entre ellos, siendo directamente proporcional a la población e inversamente proporcional a la distancia.

$$\gamma_{ij} = \frac{(Pop_i \cdot Pop_j)^{Pop_Power}}{(dist(i,j))^{Dist_Power}}$$

Aquí podemos controlar en qué medida afecta la población respecto a la distancia mediante los parámetros Pop_Power y Dist_Power, potencias de cada uno. Valores usuales suelen ser 1 para la potencia de la población, y 2 para la distancia (dividir por el cuadrado de la distancia, modelo "gravitacional"), o 3 para la distancia (modelo "magnético").

2.5.1.4 Añadimos *offset* y factor de escala.

La fórmula anterior empleada sobre una matriz de posiciones de nodos puede derivar en dos problemas:

- División por cero, cuando tenemos dos nodos distintos en la misma posición. Esto puede darse habitualmente si empleo datos de posición proporcionados por operadoras, que insertan los nodos en áreas.
- Tráfico poco realista, en ciudades de gran población (y abundante tráfico mutuo) separadas a gran distancia. El efecto de la distancia en este caso anularía el efecto de la gran población llevando el tráfico a factores poco realistas.

Se hace necesario algún mecanismo que haga el cálculo más independiente de las unidades, y la solución es normalizar cada factor de la división en cierta manera, y añadir un *offset*:

$$\gamma_{ij} = \alpha \cdot \frac{\left(\frac{Pop_i \cdot Pop_j}{Pop_\max^2} + Pop_{off}\right)^{Pop_Power}}{\left(\frac{dist(i,j)}{dist_\max} + Dist_{off}\right)^{Dist_Power}}$$

, siendo:

- α: El parámetro α es un factor de escala, se ajusta para dejar los márgenes de tráfico en valores deseados. Es una variable de normalización. (Ver ejemplo de normalización total, ap. 2.5.2.1)
- Pop_max= max{Pop_i}
- Dist_max= max{dist(i,j)}
- Pop_{off}: el *offset* para la población: evita ceros en el numerador, p.e, el tráfico de una población de 100000 habitantes a una de 10 habitantes, seria 0.
- $Dist_{off}$: el *offset* para la distancia: evita ceros en el denominador, p.e, ciudades diferentes a distancia cero.

2.5.1.5 Añadimos tráfico asimétrico

Los generadores vistos hasta ahora generan tráfico simétrico, cuando en la realidad se genera una cantidad de tráfico considerablemente mayor de unos nodos a otros (p.e., entre un servidor y un terminal). Para introducir este concepto de asimetría se usa una *matriz de niveles*.

Matriz de niveles: Matriz cuadrada de longitud **N**x**N** (donde **N** es el número de nodos de la topología), que contiene todos los factores de tráfico relativos entre distintos nodos.

Ejemplo de dos nodos, cliente y servidor:

$$L = \begin{pmatrix} 0 & 1 \\ 3 & 0 \end{pmatrix}$$

,donde L_{12}=1 representa el factor de tráfico aplicado de un cliente a un servidor, mientras L_{21}=3 sería el factor desde el servidor al cliente.

Introduciendo la matriz de niveles en el generador queda así:

$$\gamma_{ij} = \alpha \cdot \frac{Level(L_i, L_j) \cdot (\frac{Pop_i \cdot Pop_j}{Pop_\max^2} + Pop_{off})^{Pop_Power}}{(\frac{dist(i,j)}{dist_\max} + Dist_{off})^{Dist_Power}}$$

2.5.1.6 Añadimos generación de variaciones aleatorias

En ocasiones es de interés generar series de matrices de tráfico con variaciones aleatorias, controlando el grado de aleatoriedad, esto puede conseguirse simplemente añadiendo una componente de la siguiente forma:

$$1 - rf + 2 \cdot rf \cdot rand()$$

, donde rand() es una función que devuelve un número aleatorio entre 0 y 1, y el parámetro *rf* , que también estará entre 0 y 1, controla el grado de aleatoriedad:

Si *rf*=0 ➔ (1-*rf*+2**rf**rand())=1 ➔ no hay componente aleatoria.

Si *rf*=1 ➔ (1-*rf*+2**rf**rand())=2*rand() ➔ componente aleatoria entre 0 y 2.

De esta forma, podemos darle una variación al tráfico entre cada par de nodos que lo escale desde el 0% al 200% de su valor como máximo (rf = 1), o a valores intermedios (con un *rf* menor).

Un generador de tráfico completo incluirá todas las características posibles, se muestra aquí:

$$\gamma_{ij} = \alpha \cdot \frac{Level(L_i, L_j) \cdot (1 - rf + 2 \cdot rf \cdot rand()) \cdot (\frac{Pop_i \cdot Pop_j}{Pop_\max^2} + Pop_{off})^{Pop_Power}}{(\frac{dist(i,j)}{dist_\max} + Dist_{off})^{Dist_Power}}$$

Generador de Tráfico final

2.5.2 Normalizadores de tráfico

Los generadores de tráfico vistos crean una matriz de tráfico entre diferentes nodos en una proporción relativa más o menos realista y coherente. Sin embargo, para problemas concretos no basta con esto, se necesita poder especificar la cantidad de tráfico exacta que genera la red, o incluso especificar la parte de tráfico que será de entrada y la parte que será de salida de cada nodo. De esta tarea se encargan los normalizadores de tráfico.

Existen varios tipos de normalizadores. Los más importantes son:

- Normalizador total
- Normalizador por filas
- Normalizador por columnas
- Normalizador por filas y columnas

2.5.2.1 Normalizador total

El normalizador total es aquel en que se especifica el tráfico total ofrecido que debe correr por la red, es decir, lo que tiene que resultar de la suma:

$$T_{total} = \sum_{i,j} \gamma_{ij}$$

Para este normalizador bastará con multiplicar la matriz de tráficos por un factor α tal que sea:

$$\alpha = \frac{Tráfico_especificado}{Tráfico_generado}$$

Ejemplo de ajuste del parámetro α: Normalización total.

Tenemos 50 sitios enlazados mediante 85 enlaces E1 (2048 Mbps en cada dirección). El número medio de saltos es 2.75 y la utilización media 55%. ¿Qué α se debe escoger para el tráfico generado?

$$tráfico_total = \frac{\sum flujos_en_los_enlaces}{N^o\,Medio_saltos} = \frac{85 \cdot 2 \cdot 2048 \cdot 0.55}{2.75} = 6.9632\mathrm{e}+004$$

Si generamos un tráfico mediante estos generadores, para ajustarlo mediremos:

$$T = \sum_{i,j} \gamma_{ij}$$

, entonces sólo tendremos que escoger un $\alpha = \frac{6.9632\mathrm{e}+004}{T}$.

2.5.2.2 Normalizador por filas

Este tipo de normalizadores se usa cuando se ha especificado el tráfico total de salida de cada nodo ($TRAFOUT_i$), o lo que es lo mismo, el sumatorio de cada fila de la matriz de tráfico (recordemos que en una matriz de tráfico las filas representan las fuentes, y las columnas el destino del tráfico).

Esta normalización va a ser equivalente a hacer una normalización total para cada fila, es decir, se calculará un α_i diferente para cada fila que será:

$$\alpha_i = \frac{Tráfico_especificado_para_la_fila_i}{Tráfico_generado_en_la_fila_i} = \frac{TRAFOUT_i}{\sum_j \gamma_{ij}}$$

2.5.2.3 Normalizador por columnas

Este normalizador es exactamente lo mismo que el anterior para el caso de que se especifica el tráfico total de entrada de cada nodo ($TRAFIN_j$), así que igualmente se calcularán una serie de parámetros α_j de la siguiente forma:

$$\alpha_j = \frac{Tráfico_especificado_para_la_columna_j}{Tráfico_generado_en_la_columna_j} = \frac{TRAFIN_j}{\sum_i \gamma_{ij}}$$

2.5.2.4 Normalizador por filas y columnas

Este es el caso más completo de que conozco el tráfico total ofrecido por cada nodo i ($TRAFOUT_i$) y recibido por cada nodo ($TRAFIN_j$). Es decir, me especifican lo que debe sumar cada fila y cada columna de la matriz de tráfico. Ahora el ajuste es más complejo:

- Tengo N^2 incógnitas (un α para cada valor de la matriz de tráfico)
- 2*N ecuaciones (una para cada fila y para cada columna)
- Condición indispensable para que el problema tenga solución:

$$\sum_j TRAFOUT_j = \sum_i TRAFIN_i$$

Existen diferentes algoritmos para resolver este problema. El escogido aquí va acercándose a la solución iterando un proceso que va ajustando los α_{ij} tantas veces como se le especifique en un contador (*count*) hasta un límite de ajuste (*max_scale*). El código del bucle es el siguiente:

```
%Bucle para el normalizador completo
     while((count<100)&((max_scale<.999)|(max_scale>1.001)))
         count=count+1;
         trafficTableNorm=trafficTableNorm*scale;
         mat_tot=sum(sum(trafficTableNorm));
         scale=sum(TRAFOUT)/mat_tot;

         max_scale=0;
         for i=1:N
             row_scale(i)=TRAFOUT(i)/(sum(trafficTableNorm(i,:))*scale);
             col_scale(i)=TRAFIN(i)/(sum(trafficTableNorm(:,i))*scale);
             if (row_scale(i)>max_scale), max_scale=row_scale(i);end
             if (col_scale(i)>max_scale), max_scale=col_scale(i);end
         end
         for i=1:N
             for j=1:N
                    trafficTableNorm(i,j)=trafficTableNorm(i,j)*row_scale(i)*co
             l_scale(j);
             end
         end
         mat_tot=sum(sum(trafficTableNorm));
         scale=sum(TRAFOUT)/mat_tot;
     end
```

Algoritmo 2-7. Normalizador por filas y columnas

El proceso va ajustando dos escalas, la de filas y la de columnas, dependiendo del valor temporal de TRAFOUT y TRAFIN. Estas escalas se multiplican juntas (equivalente a un α_{ij}) por cada valor de la matriz en cada iteración. Para unas 100 iteraciones el resultado converge a la solución óptima con la suficiente precisión.

2.5.3 Generadores de Tarifas

El objetivo de un generador de tarifas es ser capaz de generar automáticamente tarifarios lo más cercanos posibles a los reales que ofrecen las operadoras. Se usan para no tener que introducir manualmente en la fase de pruebas estos valores en el programa de planificación.

La salida de un generador de tarifas, el tarifario, es una *matriz de costes*, es decir, una matriz que nos indique el coste que tendría conectar con una línea un nodo con cualquier otro nodo. Una matriz de costes será cuadrada de longitud igual al número de nodos, y en el caso que se estudia aquí, simétrica, ya que partimos de bidireccionalidad de líneas. Si existen varios (**k**) tipos de línea posibles, la matriz de costes será tridimensional de tamaño **N**x**N**x**k**, es decir, habrá una matriz cuadrada por cada tipo de línea (respondiendo a la forma Coste(i,j,C_{ij}), que se definía en la sección 1.1.2.2). De aquí en adelante, con el subíndice k nos referiremos al tipo de línea usado.

Los generadores de tarifas suelen ser muy sencillos, porque dependen casi por completo de un solo parámetro: la distancia entre los nodos a conectar. En general evolucionan linealmente con la distancia, aunque hay muchas excepciones, p.e. conexiones entre nodos concretos pueden ser más baratas por la ley de la oferta y la demanda de esas líneas (hay que tener en cuenta que la creación de un tarifario tiene una gran parte de competencia comercial). En todo caso estas excepciones quedan fuera de nuestro rango de conocimiento y no van a ser tratadas aquí. Nosotros veremos tres tipos básicos de generadores:

- Generador Lineal
- Generador Lineal a Trozos
- Generador Constante a Trozos

2.5.3.1 Generador Lineal

En este generador el coste depende de una función lineal con la distancia, en la que sólo tenemos dos parámetros, la pendiente de la recta (b) y el offset inicial (a):

$$C(i,j,k) = a_k + b_k \cdot dist(i,j)$$

Este es un generador muy básico, que puede ser usado para rangos muy pequeños de distancia. Sin embargo la pendiente de coste en la realidad va variando con la distancia, así que es más lógico usar los siguientes generadores cuando las distancias son medias o largas.

2.5.3.2 Generador Lineal a Trozos

En este generador el coste depende de una función que también es lineal con la distancia, pero cuya pendiente varía dependiendo del rango (o trozo) de distancias a que se encuentre el nodo destino. Los parámetros serán el offset inicial (fix), las sucesivas pendientes (b0, b1, b2,...) y las distancias a las que cambia la pendiente de la recta (d1, d2, d3...), como muestra la figura:

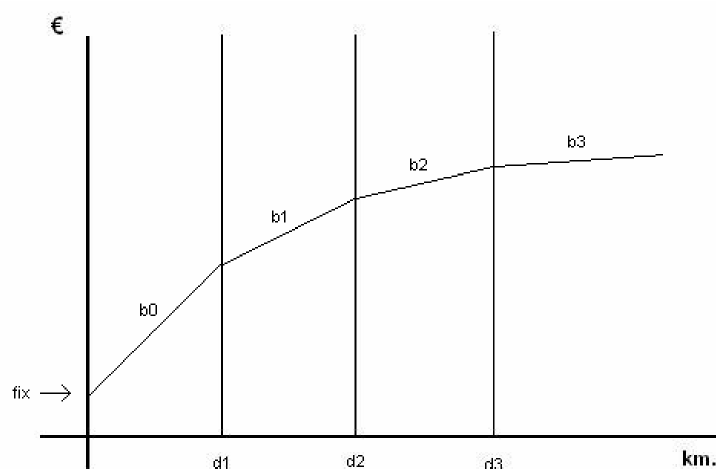


Figura 2-14. Generador lineal a trozos

A partir del último marcador de distancia, la pendiente se mantiene constante de forma indefinida para cualquier distancia superior. El número de parámetros puede ser variable dependiendo de cuantos cambios de pendiente haya, pero siempre será par (observar la figura).

Este generador está mejor adaptado a los costes reales de conexión, ya que es lo suficientemente flexible como para tener en cuenta el concepto de economía de escala, u otras variables de interés comercial y de marketing. En la mayoría de los casos las pendientes se irán haciendo más bajas conforme aumenta la distancia, debido al uso de estos conceptos.

2.5.3.3 Generador Constante a Trozos

Es parecido al anterior, pero el cobro en este se hace por pasos, como ha ocurrido con las tarifas telefónica durante mucho tiempo. Este sistema de costes deriva en unos presupuestos mucho mayores como normal general para conectar cualquier topología. Los parámetros serán el offset inicial (fix), los sucesivos costes constantes (c0, c1, c2,...) y las distancias a las que cambia un coste por el siguiente (d1, d2, d3...), como muestra la figura:

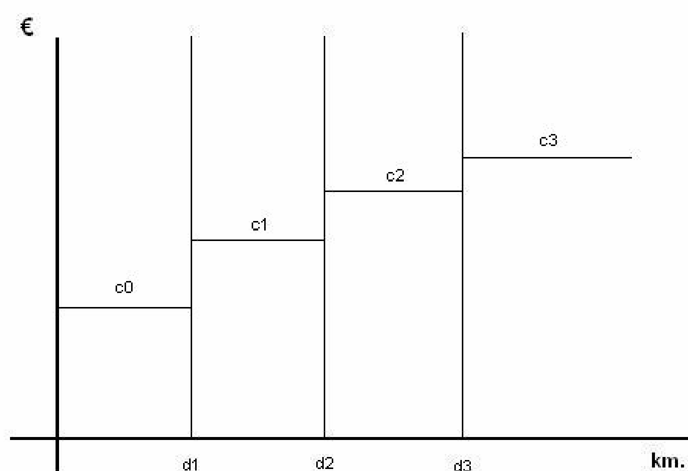


Figura 2-15. Generador constante a trozos

A partir del último marcador de distancia, el coste superior se mantiene constante de forma indefinida para cualquier distancia superior (aunque esta distancia estará usualmente limitada). El número de parámetros puede ser variable dependiendo de cuantos cambios de pendiente haya, pero en este caso, siempre será impar (observar la figura).

2.5.4 Linealización de tablas de tarifas.

A veces lo que se necesita es justo lo contrario, obtener una ecuación que me generalice el comportamiento de una tabla de costes ya generada, obteniendo sus parámetros. Puede ser necesario linealizar tablas para hacer una planificación posterior a partir de tablas de costes generadas con parámetros parecidos a las tablas existentes, para mejorarlas en algún aspecto.

Aquí vamos a ver la linealización de una tabla de forma que se ajuste a la que resultaría de un generador lineal a trozos. Recordemos que la ecuación de coste de este generador es:

$$C(i, j, k) = a_k + b_k \cdot dist(i, j)$$

Lo que nos interesa es minimizar una función *error cuadrático* definida:

$$Error_Cuadrático = \sum_{i=1}^{todos_los_enlaces} \left[\cos te_i - (a + b \cdot dist_i)^2\right]$$

El objetivo es encontrar un *a* y un *b* que minimicen la función anterior. Estos serán los parámetros buscados que definen la pendiente de una ecuación lineal del generador que dará lugar a valores semejantes a los de la tabla de costes dada. Cuanto más lineales sean los valores de la tabla original, en menor error incurrirá la linealización, y por tanto los valores después generados serán más coherentes.

2.6 Evaluación de las soluciones

A lo largo de este capítulo hemos ido viendo conceptos de la Teoría de Planificación, exponiendo problemas y las formas de enfrentarnos a ellos. Pero una vez terminado un algoritmo, una vez llevado a cabo el mecanismo que me soluciona un problema, ¿cómo compruebo hasta que punto han mejorado los parámetros que lo requerían? ¿Cómo compruebo que una topología solución es realmente mejor que la topología inicial? Para esto hay que introducir una última parte, que suele ser la primera en usar en el trabajo real de un planificador de redes, y que es imprescindible para evaluar todo el trabajo anterior: el *análisis de la red*.

Este análisis se fundamentará en aplicar las ecuaciones principales de teoría de redes para obtener de una topología, dadas sus matrices de costes y de tráfico (ofrecido), los siguientes datos:

- o Dadas las capacidades de las líneas usadas en el diseño, el tamaño medio de paquete, y el algoritmo de ruteo:
  - La matriz de encaminamiento.
  - El tráfico cursado en cada enlace.
  - La utilización real de cada uno de los enlaces.
  - El número de saltos de cada ruta, y el número medio de saltos.
  - El retardo en cada enlace, y el retardo medio de la red.

También habría opción a hacer otros cálculos interesantes con estos mismos datos:

- o Dados un retardo máximo, las capacidades de las líneas usadas en el diseño, y el algoritmo de ruteo, obtener los datos anteriores, y la longitud de paquete adecuada para que cumpla la restricción del retardo.
- o Dadas una utilización máxima, las capacidades de las líneas usadas en el diseño, y el algoritmo de ruteo, obtener los datos anteriores, y la longitud de paquete adecuada para que cumpla la restricción de utilización máxima.
- o Dados un coste, una utilización o el retardo máximos, la longitud de paquete media, y el algoritmo de ruteo, obtener también las capacidades a diseñar en los enlaces.

Para obtener todos estos datos lo principal es conseguir el primero, la tabla de encaminamiento que devuelve el mecanismo de ruteo, sin lo cual no podemos calcular la carga de tráfico que tendrá cada enlace, es decir, el trafico cursado. Una vez obtenida esta carga, los datos restantes son resultado de la directa aplicación de las fórmulas.

Teniendo en cuenta que nosotros tratamos con redes IP sobre ruteadores, según lo que se comentó antes (ver sección 2.4.2.6.1 en conclusiones respecto a los cargadores de tráfico y mecanismos de ruteo), el algoritmo que nos proporcione la tabla de encaminamiento será el OSPF. Este método se basa en asignar rutas por longitud mínima, donde la longitud es el coste que se le pasa, es decir, una matriz de costes (modificada o no en función de si se han aplicado estrategias MENTOR II para controlar la etapa de ruteo).

Para conseguir la distancia mínima entre todos los nodos de la red, hay varias formas, una es aplicar un algoritmo de camino mínimo entre cada pareja de nodos (hacer Dijkstra N·(N-1) veces), y la otra, más eficiente, es aplicar el algoritmo de Floyd, cuyo código se muestra a continuación:

```
%Algoritmo de Floyd.
for it=1:N     %n iteraciones
       % En cada iteracion, se pone de intermedio al de dicha iteracion
       for i=1:N                           %Para cada fila
              if (i==it)|(dist(i,it)==inf),continue;end
              for j=1:N               %se compara cada columna (cada valor) con lo que
                                       costaria si se pusiera de intermedio el nodo it
                     if (j==i)|(j==it), continue;end
                     if (dist(i,it)+dist(it,j)<dist(i,j))%y si es menor, se rutea
                                                          por alli y se cambia el
                                                          coste
                          dist(i,j)=dist(i,it)+dist(it,j);
                          routes(i,j)=routes(i,it);
                     end
              end
       end
end
```

Algoritmo 2-8. Floyd

Una vez conseguida la tabla de encaminamiento, se aplica la matriz de tráfico flujo por flujo, sumando a cada enlace por el que pasa el flujo el tráfico total que va desde el nodo inicial al final. Con esto obtendremos una matriz con el tráfico cursado total para cada enlace (λ). También se puede aprovechar este procedimiento para ir almacenando el número de saltos de cada ruta en una matriz (*hops*). Ya tenemos todas las variables necesarias para calcular los parámetros, mediante:

- Utilización media de cada enlace (o capacidad que tiene que tener un enlace):

$$Utilización_{enlace} = \frac{\lambda_{enlace}(paq/s) \cdot L(bits/paq)}{C_{enlace}(bps)}$$

, siendo L la longitud media de paquete.

-

- Tráfico ofrecido total en la red (en paq/s):

$$TO_{total} = \sum_{i,j} \lambda_{ij} \cdot hops(i,j)$$

- Número medio de saltos de la red:

$$\overline{Hops} = \frac{\sum_{i,j} \gamma_{ij} \cdot Hops(i, j)}{\sum_{i,j} \gamma_{ij}}$$

- Tiempo de Retardo en un enlace con nodos terminales i, j (en segundos):

$$T_{ij} = \frac{L}{C_i - \lambda_{ij} \cdot L}$$

- Tiempo de Retardo medio de la red (en segundos):

$$\overline{T} = \sum_{ij} \frac{TO_{ij}}{TO_{total}} Z_{ij} = \sum_{ij} \frac{\lambda_{ij}}{TC_{total}} T_{ij}$$

Capítulo 3
Descripción de la herramienta

3.1 Preliminares

En este capítulo pretende ser un pequeño "manual de usuario" de la herramienta. Se hará un repaso desde el comportamiento más general, hasta una breve descripción de cada una de las funciones de la herramienta, indicando sus parámetros de entrada y salida y cómo usarlos.

Respecto a los tipos de datos que vamos a manejar, las salidas de las funciones son en su mayor parte matrices o vectores, salvo quizá índices de contadores o booleanos para las funciones de comprobación. Los componentes básicos de una red (topología, tráfico, costes, encaminamiento) están representados en los siguientes datos:

- **N** *[Número entero]*: Representa el número de nodos total de la topología usada.
- **SitesTable** *[Matriz de Nx3]*: Matriz de sitios, contiene las características de los Switch_IBWR_PDBMrotSlot_SCWP_MEMCPYnodos. En nuestro caso, de los nodos nos interesa principalmente su localización en un *squareworld*, es decir, en un plano imaginario, y la población estimada de dichos nodos, en el supuesto en que representen ciudades (dicha población será un dato usado por la parte de generación de tráfico). Otros datos interesantes sobre los nodos pueden ser el tráfico de entrada o salida a cada nodo, pero esto se define fuera de esta matriz, en los vectores TRAFIN y TRAFOUT.

  Cada una de las N filas de esta matriz representa un nodo, y cada columna una característica:

  - VCORD: coordenada y (ordenadas) en el *squareworld*.
  - HCORD: coordenada x (abcisas) en el *squareworld*.
  - Population: Población estimada.
- **TrafficTable** *[Matriz de NxN]*: Matriz de tráfico, contiene todos los flujos que se dan en la red. Cada elemento de la matriz TrafficTable(i,j) representará el tráfico ofrecido estimado entre el nodo de origen i y el nodo destino j.

- **TariffTable** *[Matriz de Nx3]*: Tarifario, o matriz de costes. Contiene todos los costes entre cualquier par de nodos de la red. Cada una de las N filas de esta matriz representa un enlace, definido por un nodo origen y un nodo destino vecinos. Las columnas especifican dichos nodos, y el coste de realizar un enlace entre ellos:

  - Columna1: Nodo origen.
  - Columna2: Nodo destino.
  - Columna3: Coste del enlace entre el origen y el destino.

  A esta matriz también le denominaremos usualmente **LinksTable**, en el caso de contener solamente algunos de los costes entre nodos de la red. Esta es la matriz topología, que a diferencia de la TariffTable, representará el coste de los enlaces

que realmente existan en una red, mientras que TariffTable contendrá el coste de todos los potenciales enlaces que podría haber en dicha red.

NOTA: Acerca de la direccionalidad

En esta implementación no se considera unidireccionalidad de enlaces, suponemos que las líneas que se construyen son válidas en ambas direcciones. La tariffTable y la linksTable, por lo tanto, serán simétricas. Quién es el origen o el destino en las matrices de costes no tiene la mayor importancia. Sin embargo, se ha definido esta forma de representación de la tabla de costes porque sí da opción a la unidireccionalidad, a fin de posibles extensiones futuras.

La matriz de tráfico, sin embargo, sí es esencialmente unidireccional, porque el hecho de que un nodo envíe un flujo de datos a otro no implica que el destino envíe ese mismo tráfico al primero.

Las dos últimas matrices son datos fundamentales, que se usarán en el 90% de las funciones. Sin embargo, la forma de expresar cada una no siempre es ventajosa para el funcionamiento de los algoritmos que tratan con ellas, en muchos de ellos interesaría la forma contraria, p.e., la forma de la lista del tarifario en la matriz de tráfico, o una forma matricial para el tarifario (Matriz de topología). Es por ello que asociaremos cada una a dos tipos de formas de representación, definidos por las originales:

- Matriz tipo Traffic: Será una matriz cuadrada de NxN, donde Matriz(i,j) contenga el dato relacionado con el enlace que va desde el nodo i al nodo j.
- Matriz tipo Links: Será una matriz de Nx3 en que cada fila defina un enlace mediante las dos primeras columnas (origen, destino), y use la tercera columna para el dato relacionado con el enlace entre ese origen y el destino.

  Así, cuando sea necesario, usaremos matrices de tráfico tipo links, o LinksTables tipo traffic.

Otro detalle a considerar es cuando se da el caso de utilizar diferentes tipos de línea. Dichas matrices se ven modificadas entonces en lo siguiente:

- Matriz tipo Traffic MultiSpeed: Será una matriz cuadrada de NxNx*k*, donde Matriz(i,j,k) contenga el dato relacionado con el enlace que va desde el nodo i al nodo j y que usa el tipo de línea *k*.
- Matriz tipo Links MultiSpeed: Será una matriz de Nx4 en que cada fila defina un enlace mediante las dos primeras columnas (origen, destino), use la tercera columna para el dato relacionado con el enlace entre ese origen y el destino, y la cuarta columna para indicar el tipo de línea *k* de dicho enlace.

- **K** *[Número entero]*: representa el tipo de línea de un enlace, definido por la matriz *lineTypesTable.*
- **LineTypesTable** *[Matriz]*: Matriz que contiene los posibles tipos de línea disponibles para la red. Cada fila de la matriz está asociada a un tipo de línea, y las columnas (de número variable) las características de esa línea (ver sección 2.5.3).
- **RoutingTables** *[Matriz de NxN]*: Tabla de encaminamiento, o de ruteo. Contiene el nodo destino del siguiente salto que hay que dar desde un nodo para llegar a otro. Cada fila de esta matriz es la que conocen los dispositivos de ruteo, y mediante la cual deciden por donde encaminar cada paquete que les llega.

3.2 Bloques

La herramienta se compone de 40 funciones, que se pueden clasificar en varios bloques de acuerdo a la etapa de diseño o la necesidad circunstancial del problema.

3.2.1 Bloque generadores

El conjunto de funciones de este bloque son las que me generarán las matrices fundamentales, el tarifario, la matriz de topologías, y la matriz de sitios. Con estos datos iniciales seremos capaces de aplicar posteriormente los algoritmos de optimización y análisis que hagan falta.

Se hará uso de estos generadores siempre que necesitemos datos para simular, ahorrándonos el coste de comprar datos reales. Obviamente las matrices creadas serán más aproximadas a la realidad cuanto mejor ajustados estén los parámetros que las definen. Una solución de coste intermedio sería utilizar el procedimiento de comprar una pequeña muestra de datos, obtener de ellos los parámetros, y generar luego una gran cantidad de datos mediante las funciones de este bloque, suficiente para simular las redes.

Se han incluido en esta sección un generador de sitios, un generador de tráfico (el más avanzado de la teoría del capítulo dos), tres generadores de costes de diferentes tipos (lineal, lineal a trozos, constante a trozos), y los cuatro normalizadores de tráfico (total, por filas, por columnas y por filas y columnas). Aunque no sean un generador en sí mismo, los normalizadores se han incluido porque son parte esencial de la creación de la matriz de tráficos.

Todas estas funciones corresponden a la teoría del apartado 2.5.

1) [sitesTable] = randomSiteGenerator (numberOfNodes, popMax, seed)

Función:

Genera sitios, con una población aleatoria y unas coordenadas en *squareworld* aleatorias.

Parámetros:

- Entrada:
  - NumberOfNodes *[Número entero]* : Le indica al generador cuantos nodos crear.
  - PopMax *[Número entero]*: Le indica al generador el tamaño máximo en habitantes de las poblaciones de cada sitio.
  - Seed *[Número entero]* : Semilla que se le pasa al generador aleatorio.
- Salida:
  - **SitesTable**.

2) [trafficTable2, trafficTable] = trafficGenerator (sitesTable , levelIndicator , levelMatrix , alpha, rf, pop_offset, pop_power, dist_offset, dist_power, seed)

Función:

Genera la matriz de tráfico, utilizando la ecuación vista en teoría:

$$\gamma_{ij} = \alpha \cdot \frac{Level(L_i, L_j) \cdot (1 - rf + 2 \cdot rf \cdot rand()) \cdot (\frac{Pop_i \cdot Pop_j}{Pop_\max^2} + Pop_{off})^{Pop_Power}}{(\frac{dist(i,j)}{dist_\max} + Dist_{off})^{Dist_Power}}$$

Parámetros:

o Entrada:

- **SitesTable**.
- LevelIndicator *[Array de tamaño N]*: un vector columna, de tantas filas como sites, indicando cuál es el nivel de cada nodo de la SitesTable.
- LevelMatrix *[Matriz de Lmax x Lmax]*: Matriz que indica la proporción de tráfico de un nodo a otro según el nivel de ambos, que puede estar entre 1 y Lmax.
- Alpha *[Número Real]*: parámetro de escalado.
- Rf *[Número Real]*: componente aleatoria entre 0 y 1. En 0 no hay componente aleatoria, y en uno se tiene la máxima variación.
- Pop_Offset *[Número Real]* : *offset* para evitar una desviación grande debido relación de proporcionalidad entre poblaciones demasiado grande.
- Pop_power *[Número Real]*: potencia a la que se eleva el factor distancia.
- Dist_Offset *[Número Real]*: *offset* para evitar una desviación grande debido a poblaciones muy juntas.
- Dist_power *[Número Real]*: Potencia a la que se eleva el factor distancia.
- Seed *[Número Real]* : Semilla que se le pasa al generador aleatorio.

o Salida:

- TrafficTable2: **TrafficTable** tipo Traffic.
- TrafficTable: **TrafficTable** tipo Links.

3) [tariffTable] = costGenerator_linear (sitesTable,lineTypesTable)

Función:

Genera el tarifario, la matriz de costes, utilizando el tipo de línea de crecimiento lineal.

Parámetros:

o Entrada:

- **SitesTable**.
- **LineTypesTable** *[Matriz]*: Tabla de K filas y 2 columnas, con los parámetros a y b para cada linetype, que servirán para calcular el coste con la ecuación siguiente:

  $C(i,j,k) = a_k + b_k \cdot d$

  siendo d la distancia entre el nodo origen y el destino.

o Salida:

- **TariffTable**: tipo Traffic MultiSpeed.

4) [tariffTable] = costGenerator_piecewise_lin (sitesTable, lineTypesTable)

Función:

Genera el tarifario, la matriz de costes, utilizando el tipo de línea de crecimiento lineal a trozos.

Parámetros:

o Entrada:

- **SitesTable**.
- LineTypesTable *[Matriz]*: Tabla de K filas y un numero par de columnas, con los parámetros fix, distcost, dist1 y distcost2, dist2 y distcost3...etc, para cada linetype, que servirán para calcular el coste con la siguiente ecuación:

C(i,j,k) = fix + distcost · d   if d<=dist1

C(i,j,k) = fix + distcost · dist1 + (d-dist1) · distcost2   if d<=dist2

C(i,j,k) = fix + distcost · dist1 + (dist2-dist1) · distcost2 + (d-dist2) · distcost3   otherwise.

…or (d<=dist3...)

…

siendo d la distancia entre el nodo origen y el destino.

Es decir, hay un coste fijo, luego aumenta linealmente con distcost hasta la distancia límite dist1, luego aumenta linealmente con distcost2 hasta dist2... etc, hasta el ultimo límite, después de lo cual aumentará linealmente con distcost(n).

o Salida:

- **TariffTable** tipo Traffic MultiSpeed.

5) [tariffTable] = costGenerator_piecewise_const (sitesTable , lineTypesTable)

Función:

Genera el tarifario, la matriz de costes, utilizando el tipo de línea de coste constante a trozos.

Parámetros:

o Entrada:

- **SitesTable**.
- LineTypesTable *[Matriz]*: Tabla de K filas y un numero impar de columnas, con los parámetros fix, dist1, cost1, dist2, cost2, dist3, cost3... etc para cada linetype, que servirán para calcular el coste con la siguiente ecuación:

C(i,j,k) = fixk   if d<=dist1

C(i,j,k) = cost1   if d<=dist2

C(i,j,k) = cost2   if d<=dist3

C(i,j,k) = cost3   otherwise …or (d<=dist4...)

…

con d la distancia entre el nodo origen y el destino.

Es decir, hay costes fijos distintos en cada tramo entre dist(n) y dist(n+1), y luego hay un coste fijo final por mucho que aumente la distancia.

- Salida:
  - **TariffTable** tipo Traffic MultiSpeed.

6) [trafficTableNorm]=totalNormalizator (trafficTable, totalTrafficCarried)

Función:

Normaliza la matriz de tráfico con el normalizador total (se conoce el total del tráfico ofrecido) visto en teoría.

Parámetros:

- Entrada:
  - **TrafficTable** tipo Traffic.
  - TotalTraffiCarried *[Número real]*: Tráfico total que se debe inyectar a la red (en paquetes por segundo).
- Salida:
  - TrafficTableNorm *[Matriz traffic]*: Matriz de tráfico que está normalizada con un alpha homogénea, de forma que si se suma todo el trafico inyectado de la tabla normalizada, resulte exactamente TotalTraffiCarried.

7) [trafficTableNorm]=rowNormalizator(trafficTable, trafout)

Función:

Normaliza la matriz de tráfico con el normalizador por filas (se conoce el tráfico que sale de cada nodo) visto en teoría.

Parámetros:

- Entrada:
  - **TrafficTable** tipo Traffic.
  - Trafout *[Array vertical de tamaño N]*: Tráficos de salida deseada generados por cada nodo.
- Salida:
  - TrafficTableNorm *[Matriz traffic]*: Matriz de tráfico que está normalizada con un alpha diferente para cada fila, de forma que si se suma el tráfico que sale de un nodo i con destino a cualquier nodo, resulte exactamente "trafout(i)".

8) [trafficTableNorm]=columnNormalizator(trafficTable, trafin)

Función:

Normaliza la matriz de tráfico con el normalizador por columnas (se conoce el tráfico de entrada de cada nodo) visto en teoría.

Parámetros:

- Entrada:
  - **TrafficTable** tipo Traffic.
  - Trafin *[Array vertical de tamaño N]*: Tráficos de entrada deseada generados por cada nodo.
- Salida:
  - TrafficTableNorm *[Matriz traffic]*: Matriz de tráfico que está normalizada con un alpha diferente para cada columna, de forma que si se suma el tráfico que entra a un nodo i desde cualquier otro nodo, resulte exactamente "trafin(i)".

9) [trafficTableNorm]=rowColumnNormalizator(trafficTable, trafin, trafout)

Función:

Normaliza la matriz de tráfico con el normalizador por filas y columnas (se conoce lo el flujo que entra y sale de cada nodo) visto en teoría.

Parámetros:

- o Entrada:
  - **TrafficTable** tipo Traffic.
  - Trafin *[Array vertical de tamaño N]*: Tráficos de entrada deseada generados para cada nodo.
  - Trafout *[Array vertical de tamaño N]*: Tráficos de salida deseada generados por cada nodo.

- o Salida:
  - TrafficTableNorm *[Matriz traffic]*: Matriz de tráfico normalizada con un alpha diferente para componente, de forma que al sumar el tráfico que entra a un nodo i desde cualquier nodo, resulte exactamente "trafin(i), y además al sumar el tráfico que sale del nodo i con destino a cualquier nodo, de exactamente "trafout(i)".

3.2.2 Bloque algoritmos de planificación

El conjunto de funciones de este bloque conforman el núcleo central del trabajo. Son los algoritmos que solucionan problemas de optimización. Abarcan desde las soluciones básicas a los MST, pasando por los CMST y los CMST MultiSpeed al desarrollo de un conjunto de algoritmos más complejo, el MENTOR.

El uso de estos algoritmos es encontrar como norma general una topología de mínimo coste que cumpla unas expectativas. El mayor o menor número de expectativas dependerá de la complejidad del algoritmo. Obviamente en el campo que tratamos las mejores soluciones las harán los algoritmos más simples, mientras los más ambiciosos dependerán de más parámetros que pueden hacer variar mucho la solución.

Se ha dividido esta sección en 3 partes: algoritmos sobre el problema MST y SPT, algoritmos para CMST y *MultiSpeed*, y algoritmos MENTOR.

3.2.2.1 Algoritmos del problema MST y SPT

Estos son los algoritmos que solucionan los problemas básicos de planificación, expuestos en la sección 2.1. Se han incluido en este apartado los 2 algoritmos MST (Kruskal y Prim), un SPT y la mezcla (Dijkstra y Prim-Dijkstra). Todos ellos intentan interconectar un conjunto de nodos, optimizando en los primeros el coste, en el tercero el número de saltos, y obteniendo una solución intermedia en el último.

10)linksTable=kruskal(tariffTable)

Función: Obtiene el MST mediante el algoritmo de Kruskal.

Parámetros:

o Entrada:

- **TariffTable** tipo Traffic.

o Salida:

- **LinksTable** tipo Links.

`11)linksTable=prim(tariffTable,rootnode)`

Función: Obtiene el MST mediante el algoritmo de Prim.

Parámetros:

o Entrada:

- **TariffTable** tipo Traffic.
- Rootnode *[Número Entero]*: Nodo de comienzo del algoritmo.

o Salida:

- **LinksTable** tipo Links.

`12)linksTable=dijkstra(tariffTable,rootnode)`

Función: Obtiene el SPT mediante el algoritmo de Dijkstra.

Parámetros:

o Entrada:

- **TariffTable** tipo Traffic.
- Rootnode *[Número Entero]*: Nodo de comienzo del algoritmo.

o Salida:

- **LinksTable** tipo Links.

`13)linksTable=prim_dijkstra(tariffTable,rootnode,alpha)`

Función: Obtiene un punto intermedio entre un MST (por Prim) y un SPT (por Dijkstra).

Parámetros:

o Entrada:

- **TariffTable** tipo Traffic.
- Rootnode *[Número Entero]*: Nodo de comienzo del algoritmo.
- Alpha *[Número Real]*: debe estar entre 0 y 1, y tendera a una solución entre prim (0) y dijkstra(1).

o Salida:

- **LinksTable** tipo Links.

3.2.2.2 Algoritmos del *capacitated* MST y *multispeed* CMST

Estos son los algoritmos que dan el siguiente paso en complejidad para problemas de diseño de redes, expuestos en la sección 2.2 y 2.3. Se han incluido en esta sección un algoritmo para CMST (Esau-Williams), y otro para *multispeed* CMST (MSLA). Corresponden a una nueva restricción con respecto a los algoritmos anteriores: cumplir un límite de capacidad en los enlaces, y además en el segundo se tienen en cuenta las distintas velocidades de las líneas disponibles para la interconexión.

Se han incluido en esta sección un algoritmo para CMST (Esau-Williams), y otro para *multispeed* CMST (MSLA).

`14) [linksTable]=E_W (tariffTable,weights,W,central)`

Función: Obtiene una solución para el problema CMST, mediante el algoritmo de Esau-Williams.

Parámetros:

- o Entrada:
  - **TariffTable** tipo Links.
  - Weights *[Array vertical de tamaño N]*: Matriz columna de los pesos de cada nodo, es decir, del tráfico (caudal, en bps) que tienen que llevar al nodo central.
  - W *[Número entero]*: capacidad de los enlaces.
  - Central *[Número entero]*: índice del nodo central respecto al cual se hace el algoritmo.
- o Salida:
  - **LinksTable** tipo links.

`15)[linksTableMS]=MSLA (tariffTable, capacity , utilization, weight, central)`

Función: Obtiene una solución para el problema CMST *MultiSpeed*, mediante el algoritmo MSLA.

Parámetros:

- o Entrada:
  - **TariffTable** tipo Traffic *MultiSpeed*.
  - Capacity *[Array vertical de tamaño k]*: Matriz columna de las capacidades correspondientes a cada una de las linetypes con que está construida la tariffTable. K es ese número de linetypes, y es igual al tamaño de tercera dimensión de tariffTable.
  - Utilization *[Número real]*: Utilización máxima de los enlaces (entre 0 y 1).
  - Weight *[Array vertical de tamaño N]*: Matriz columna de los pesos de cada nodo, es decir, del tráfico (caudal, en bps) que tienen que llevar al nodo central.
  - W *[Número entero]*: capacidad de los enlaces.
  - Central *[Número entero]*: índice del nodo central respecto al cual se hace el algoritmo.
- o Salida:
  - **LinksTable** tipo links MultiSpeed.

3.2.2.3 Algoritmos MENTOR

MENTOR no es sólo un algoritmo, sino una filosofía, como se comentó y expuso en la sección 2.3. Estos son algunos de los algoritmos posibles que forman el conjunto de la filosofía MENTOR. Se ha implementado un algoritmo para cada etapa del diseño, orientando la implementación hacia el AMENTOR, que resuelve el problema con un *Aumento* del *backbone* de mínimo coste. Hay dos funciones que se corresponden a diseños AMENTOR completos, una básica (AMentor) y una extensión para *multispeed* (AmentorMS), y las funciones correspondientes a cada una de las etapas de los dos: para el AMentor básico se usan M_threshold, M_PrimDj, M_calculateDistanceBB, y M_connect. Para el AmentorMS, se modifican algunas de estas funciones para que puedan soportar distintas velocidades, y se usan finalmente: M_thresholdMS, MSLA (para el árbol inicial, en puesto de M_PrimDj), M_calculateDistanceBB, M_connectMS, y M_Msla, función esta última que se añade en la parte de acceso local.

Esta última extensión para *multispeed* del AMENTOR es la aportación del autor de la que se habla en la sección 2.4.3.

```
16) [linksSolution, linksBB, nextBB,median] = AMentor (tariffTable,
    trafficTable, sitesTable, Wparam, Rparam, C, L, alpha, graph)
```

Función: Clasifica los nodos según su importancia para crear una red backbone confiable (biconexa) y una red de acceso a ella desde todos los nodos terminales, con el mínimo coste.

Parámetros:

o Entrada:

- **TariffTable** tipo Links.
- **TrafficTable** tipo Traffic.
- **SitesTable**.
- Wparam *[Número Real]*: peso mínimo para considerar backbones los nodos dentro de la etapa Threshold Clustering.
- Rparam *[Número Real]*: radio máximo dentro del cual los nodos terminales cercanos a un nodo *backbone* se asocian a él.
- C *[Número Real]*: Capacidad de los enlaces.
- L *[Número entero]*: Tamaño medio de paquete.
- Alpha *[Número Real]*: Parámetro del árbol prim-dijkstra. Debe estar entre 0 y 1, y tendera a una solución entre prim (0) y dijkstra(1).
- Graph *[Número entero]*: parámetro que controla el grado de interacción de la función mediante la cantidad de *displays* de los diferentes estados del algoritmo:
    - 2: Se ven todos los displays. (cada paso del algoritmo).
    - 1: Se ven solo los más significativos: Estado inicial y final del algoritmo, *backbones*.
    - 0: Sin salida en pantalla..

o Salida:

- LinksSolution *[tipo Traffic]*: **LinksTable** de todos los enlaces del diseño resultantes de todo el proceso.
- linksBB *[tipo Traffic]*: **linksTable** entre los nodos que resultan ser backbone (el resto de la matriz estará a 0).

- NextBB *[Array vertical de tamaño N]*: Array vertical que indica el nodo backbone al que corresponde cada nodo pendiente. Es cero para los nodos que son backbone.
- Median *[Número entero]*: es la mediana, indica el nodo backbone con menor momento, el que tiene más nodos cercanos. . El momento de cada nodo viene dado por la ecuación descrita en la sección 2.4.2.2.

17)[linksSolution, linksBB, nextBB,median] = AMentorMS (tariffTable,trafficTable,sitesTable,Wparam, Rparam, capacity, L utilization, weight, graph)

Función: Igual que la anterior, pero considerando líneas con diferentes capacidades disponibles (adaptado para *multispeed*[3]).

Parámetros: Los mismos que en la función anterior, salvo alpha y C (no necesarios ahora), y añadiendo los siguientes en la entrada:

- Capacity *[Array vertical de tamaño k]*: Capacidad de cada uno de los tipos de línea (una para cada uno de los k tipos de línea en la tariffTable).
- Utilization *[Número Real]*: Utilización máxima de los enlaces, parámetro a pasarle al MSLA que se encarga del acceso local del algoritmo.
- Weight *[Array vertical de tamaño N]*: Matriz columna de los pesos de cada nodo, es decir, del tráfico (caudal, en bps) que tienen que llevar al nodo central que se decida durante la primera etapa del algoritmo.

18) [BB,nextBB,linksBBT,median] =M_threshold(trafficTable, tariffTable, sitesTable, Wparam, Rparam,C ,L)

Función: Clasificar el conjunto de nodos en dos subconjuntos: backbones, y nodos terminales.

Parámetros:

- o Entrada:
  - **TariffTable**.
  - **TrafficTable**.
  - **SitesTable**.
  - Wparam *[Número Real]*: peso mínimo para considerar backbones los nodos
  - Rparam *[Número Real]*: radio máximo dentro del cual entran los nodos anexos a los backbones, antes de necesitar otro backbone
  - C *[Número Real]*: Capacidad de los enlaces que vamos a usar para la clasificación.
  - L *[Número entero]*: Tamaño medio de paquete.

- o Salida:
  - BB *[Array booleano vertical de tamaño N]*: Array binario que indica los nodos que resultan ser backbone (1) o no (0).
  - NextBB *[Array vertical de tamaño N]*: Indica el nodo backbone al que corresponde cada nodo pendiente (terminal). Es cero para los nodos que son backbone.

[3] Diferencia para *graph>0*: el primer gráfico de la función, correspondiente a todos los posibles enlaces de la red backbone resultante tras haber aplicado *threshold clustering*, en este caso no muestra los costes de las líneas, ya que al haber varios tipos de línea se sobreescribiría el coste de cada uno. Por tanto, para hacerse una idea de esto hay que fijarse en la leyenda de colores.

- LinksBB *[Matriz traffic]*: es la tabla tariffTable (en formato traffic), pero conteniendo únicamente los nodos *backbone* (los demás están a cero).
- Median *[Número entero]*: es la mediana, indica el nodo backbone con menor momento, el que tiene más nodos cercanos. . El momento de cada nodo viene dado por la ecuación descrita en la sección 2.4.2.2.

19) `[BB,nextBB,linksBBT,median] =M_thresholdMS(trafficTable, tariffTable, sitesTable, Wparam, Rparam,C,L)`

Función: Igual que el anterior, pero adaptado para *multispeed*.

Parámetros:

o Entrada: Los mismos que en la función anterior. Para la capacidad C se le tiene que pasar el *max(capacity)* de los parámetros de AMentorMS.

o Salida:

- LinksBBT *[Matriz traffic MultiSpeed]*: es misma LinksBBT pero *multispeed*.

20) `[linksTable]=M_primDj(tariffTable,rootnode,nextBB,alpha)`

Función: Creación inicial de la topología de la red de malla, es decir, la malla inicial entre backbones y un Prim-Dijkstra entre cada backbone y sus nodos pendientes.

Parámetros:

o Entrada:

- **TariffTable**.
- Rootnode *[Número entero]*: Nodo del que se parte para el cálculo del algoritmo, en este caso, la mediana calculada en M_threshold.
- NextBB *[Array vertical de tamaño N]*: Indica el nodo backbone al que corresponde cada nodo pendiente (terminal). Es cero para los nodos que son backbone.
- Alpha *[Número real]*: Parámetro del árbol prim-dijkstra. Debe estar entre 0 y 1, y tenderá a una solución entre prim (0) y dijkstra(1).

o Salida:

- **LinksTable** tipo links: conjunto de enlaces resultantes.

21) `[distances,routingTables,blocks]=M_calculateDistanceBB(linksBBT, BB)`

Función: Encontrar en los nodos del backbone los bloques (componentes 2-connected), los *cutVertex* (puntos de articulación), y la tabla de encaminamiento (llamando a OSPFrouting) entre ellos. Devuelve unas distancias que servirán para elegir por qué nodos conectar entre sí los bloques diferentes.

Parámetros:

o Entrada:

- LinksBBT *[Matriz traffic]*: es la tabla tariffTable (en formato traffic), pero conteniendo únicamente los nodos *backbone* (los demás están a cero).
- BB *[Array vertical de tamaño N]*: Array binario que indica los nodos que resultan ser backbone (1) o no (0).

o Salida:

- Distances *[Matriz de NbxNb, siendo Nb el número de nodos backbone que haya]*: matriz de distancias en términos de número de componentes, existentes entre dos nodos backbone cualquiera.

- RoutingTables *[Matriz de NxN]*: Matriz de encaminamiento, que indica para cualquier nodo el siguiente salto que tiene que dar para llegar a cierto destino.
- Blocks *[Matriz de NxN]*: Matriz que contiene los nodos pertenecientes a cada bloque. Los elementos no cero de cada fila de esta matriz forman un bloque.

22)[linksBBT]=M_connect(linksBBT,distances,blocks,tariffTable)

Función: Añadir un enlace entre 2 nodos de diferentes componentes según cierto merito calculado con las distancias, y con el coste del tarifario.

Parámetros:

o Entrada:

- LinksBBT *[Matriz traffic]*: es la tabla tariffTable (en formato traffic), pero conteniendo únicamente los nodos *backbone* (los demás están a cero).
- Distances *[Matriz de NbxNb, siendo Nb el número de nodos backbone que haya]*: matriz de distancias en términos de número de componentes, existentes entre dos nodos backbone cualquiera.
- Blocks *[Matriz de NxN]*: Matriz que contiene los nodos pertenecientes a cada bloque. Los elementos no cero de cada fila de esta matriz forman un bloque.
- **TariffTable.**

o Salida:

- LinksBBT *[Matriz traffic]*: es la nueva tabla de enlaces entre nodos *backbone* tras la adición del enlace.

23)[linksBBT]=M_connectMS(linksBBT,distances,blocks,tariffTable)

Función: Añadir un enlace entre 2 nodos de diferentes componentes según cierto merito calculado con las distancias, y con el coste del tarifario.

Parámetros:

o Entrada: Los mismos que en la función anterior.

o Salida:

- LinksBBT *[Matriz traffic MultiSpeed]*: es misma LinksBBT pero *multispeed*.

24)[linksTable]=M_Msla(tariffTable, nextBB, capacity, utilization, weight)

Función: Obtiene una solución para el problema de acceso local de los nodos terminales a los backbone del MENTOR, resolviendo el problema como un CMST *MultiSpeed*, mediante el algoritmo MSLA.

Parámetros:

o Entrada:

- **TariffTable.**
- NextBB *[Array de N]*: Indica el nodo backbone al que corresponde cada nodo pendiente (terminal). Es cero para los nodos que son backbone.
- Capacity *[Array vertical de tamaño k]*: Capacidad de cada uno de los tipos de línea (una para cada uno de los k tipos de línea en la tariffTable).
- Utilization *[Número Real]*: Utilización máxima de los enlaces.
- Weight *[Array vertical de tamaño N]*: Matriz columna de los pesos de cada nodo, es decir, del tráfico (caudal, en bps) que tienen que llevar al nodo central.

o Salida:

- LinksTable *[Matriz traffic]*. Topología de la red de acceso local.

3.2.3 Bloque análisis y visualización

Las funciones de este bloque corresponden a la parte del trabajo encargada de examinar los resultados obtenidos con los algoritmos de las partes anteriores. En muchos casos son funciones utilizadas por éstos, para retroalimentar el proceso o simplemente ir mostrando el estado de las topologías.

Se han incluido en esta sección un visualizador de topologías, un extractor de características de diseños, y un linealizador que obtiene los parámetros que linealizarían una tabla de tarifas dada (parámetros útiles para generar posteriormente tablas parecidas, como se comentaba en el bloque de generadores).

25)displayGraph (sitesTable,linksTable,K,et,al,col)

Función: Muestra la topología de la red (nodos y enlaces) gracias al parámetro sitesTable que contiene las posiciones X e Y de los nodos, y la tariffTable que contiene los enlaces. Éstos siguen un código de colores en función del coste relativo (opcional).

Parámetros:

o Entrada:

- **SitesTable**.
- **LinksTable** tipo Traffic.
- K *[Número Entero]*: número de línea a dibujar (selecciona la tercera dimensión de la tariffTable). Si se pone a cero, dibuja todas las líneas de todos los tipos de línea que haya en el tariffTable.
- Et *[Número Entero]*: parámetro de etiquetado, que puede ser uno de los siguientes:
  - et=0: no se etiqueta.
  - et=1: se etiquetan solo los nodos(mostrando un numero).
  - et=2: se etiquetan solo los enlaces(mostrando su coste).
  - et=3: se etiquetan ambos.
- Al *[Número Entero]*: parámetro *Arrow-line,* indica si los enlaces se muestran mediante flechas (unidireccionales) o líneas simples (bidireccionales):
  - Al=1: se muestran las flechas.
  - Al=0: se muestran líneas.
- Col *[Número entero]*: indica si usa el mismo color (0), colores diferentes según el coste de la línea (1) o colores diferentes para cada linetype (2).

Leyenda de colores: se usan 8 colores diferentes en función del coste respecto al coste máximo (el mayor de los encontrados en tariffTable):

- Blanco: El coste de la línea es inferior al 12.5% del coste máximo.
- Cyan: El coste de la línea esta entre el 12.5% y el 25% del coste máximo.
- Azul: El coste de la línea esta entre el 25% y el 37.5% del coste máximo.
- Verde: El coste de la línea esta entre el 37.5% y el 50% del coste máximo.
- Amarillo: El coste de la línea esta entre el 50% y el 62.5% del coste máximo.

- Rojo: El coste de la línea esta entre el 62.5% y el 75% del coste máximo.

- Magenta: El coste de la línea esta entre el 75% y el 87.5% del coste máximo.

- Negro: El coste de la línea esta entre el 87.5% y el 100% del coste máximo.

NOTA: Si una línea tiene coste 0, no se dibuja.

26) [HopsMean, Tmean, U, T] = calculateParams(trafficTable, trafficPerLink, hops, C, L)

Función: Calcula los parámetros más característicos de un diseño.

Parámetros:

o Entrada:
  - **TrafficTable** tipo traffic.
  - TrafficPerLink *[Matriz de NxN]*: Matriz de Tráfico Cursado, indica cuál es el tráfico cursado por cada enlace.
  - Hops *[Matriz de NxN]*: Matriz con el total de saltos que hay que dar desde cualquier nodo para llegar a cualquier otro, tal y como está diseñado el ruteo.
  - C *[Número entero]*: Capacidad de los enlaces.
  - L *[Número entero]*: Longitud media de paquete

o Salida:
  - HopsMean: número medio de saltos.
  - Tmean: Retardo medio de paquete.
  - U: utilización de cada enlace.
  - T: retardo medio de cada enlace.

27)[a,b]=lineariz(table,graph)

Función: Obtiene los parámetros que linearizan una función bidimensional.

Parámetros:

o Entrada:
  - Table *[Array vertical de Nx2]*: tabla de N filas con 2 columnas tipo (coste, dist).
  - Graph *[booleano]*: Parámetro para indicar si se muestra una figura de la linearización (1), o no (0).

o Salida:
  - ‘a’ y b’: parámetros que linearizan la tabla de la forma: Coste(i)= a + b*dist(i)

3.2.4 Bloque Funciones generales

El conjunto de funciones de este bloque son de propósito general, en su mayor parte funciones usadas como herramientas para los algoritmos del segundo bloque, pero que también tienen utilidad fuera de ellos, para obtener distintas características de una red.

Se ha dividido esta sección en 3 partes: herramientas de chequeo y búsqueda en árboles, (relacionadas con los conceptos de grafos que han sido necesarios para la implementación de los algoritmos), herramientas de encaminamiento, y conversores de forma.

3.2.4.1 Herramientas de Chequeo y búsqueda en árboles

Se han incluido en esta sección funciones de chequeo de ciclos y de conectividad, de búsqueda de componentes y bloques en un árbol, así como las funciones internas usadas por ellas.

28) `[check]=checkConnected(linksTable)` **(versión rigurosa)**

Función: Chequea si un grafo está conectado.

Parámetros:

- o Entrada:
  - Table*[Matriz traffic o links]*: Matriz topología.
- o Salida:
  - Check*[Booleano]*: devuelve 1 si la tabla esta conectada y 0 si no.

29) `[check]=checkConnect(linkstable)` **(versión propia)**

Función: Chequea si un grafo está conectado.

Parámetros:

- o Entrada:
  - **LinksTable** tipo Traffic o Links.
- o Salida:
  - Check*[Booleano]*: devuelve 1 si la tabla esta conectada y 0 si no.

`30)[components]=findComponents(linkstable , tableType)`

Función: Encuentra los componentes (subgrafos conexos) que hay en un grafo.

Parámetros:

- o Entrada:
  - **LinksTable** tipo Traffic o Links.
  - TableType *[Número Entero]*: indica en qué formato se le pasa la linksTable (y en qué formato devolverá los componentes):
    - 1: tipo traffic (NxN).
    - 0: tipo links (Nx3).
- o Salida:
  - Components *[Matriz de tipo (traffic)xn o (links)xn]*: Matriz de subcomponentes conexos de la linksTable.

    Las dos primeras dimensiones son las mismas que en el formato del linksTable introducido, la tercera dimensión es para agrupar los distintos componentes.

`31)[blocks]=findBlocks(linksTable, type, begin)`

Función: Encuentra los bloques (componentes biconexos) que hay en un grafo.

Parámetros:

- o Entrada:

- **LinksTable** tipo Traffic o Links.
- TableType *[Número Entero]*: indica en qué formato se le pasa la linksTable:
    - 1: tipo traffic (NxN).
    - 0: tipo links (Nx3).
- Begin *[Número entero]*: un nodo de comienzo de búsqueda (opcional).

o Salida:

- Blocks *[Matriz de NxN]*: Matriz que contiene los nodos pertenecientes a cada bloque. Los elementos no cero de cada fila de esta matriz forman un bloque.

`32)[check, cycles]=checkCycle(linksTable)`

Función: Chequea si un grafo tiene algún ciclo.

Parámetros:

o Entrada:

- **LinksTable** tipo Traffic o Links.

o Salida:

- Check: devuelve 1 si hay un ciclo y 0 si no.
- Cycles *[Matriz de NxNxNumCycles]*: devuelve en las 2 primeras dimensiones cada componente en el cual se encuentra un ciclo, en el formato *traffic* (NxN), la tercera dimensión es para agrupar los distintos ciclos.

`33)[check, cutVertex]=checkBiconnected(linksTable)`

Función: Chequea si un grafo es biconexo.

Parámetros:

o Entrada:

- **LinksTable** tipo Traffic.

o Salida:

- Check *[Booleano]*: devuelve 1 si el grafo es biconexo y 0 si no.
- CutVertex *[Array vertical de N booleanos]*: Devuelve los puntos de articulación. Para cada nodo indica si es (1) o no(0) un punto de articulación.

  Obviamente, si check=1, no habrá cutVertex (estará a cero todo el array).

`34)[parent,cycle,edges,number]=DFSforest (LinksTable, [begin])`

Función: Realiza una búsqueda *Deep First Search* en un grafo, devolviendo la jerarquía relativa entre los nodos (quién es padre de quién), el orden del rastreo, y si el grafo presenta ciclos.

arámetros:

o Entrada:

- **LinksTable** tipo Traffic o Links.
- Begin *[Número entero]*: un nodo de comienzo de búsqueda (opcional).

o Salida:

- Parent *[Array vertical de tamaño N]*: Muestra los padres de cada nodo.

- Cycle *[Booleano]*: devuelve 1 si hay un ciclo y 0 si no.
- Edges *[Matriz]*: Matriz de dos columnas, donde cada fila es un enlace por el que se pasa en el recorrido del rastreo del bosque.
- Number *[Array vertical de tamaño N]*: Orden con el que se pasa por cada nodo desde begin. (orden con el que se han procesado los nodos).

`35)[ncv]= countCutVertex(start, end, routingTables, cutVertex, ncv)`

Función: Calcula el número de puntos de articulación entre dos nodos.

Parámetros:

- Entrada:
  - Start *[Número entero]*: nodo de inicio de el camino
  - End *[Número entero]*: nodo destino de el camino
  - **RoutingTables**.
  - CutVertex *[Array booleano vertical de tamaño N]*: Para cada nodo indica si es (1) o no(0) un punto de articulación.
  - Ncv *[Número entero]*: Es un valor usado internamente para la recursividad de la función, así que como uso general siempre se iniciará a 0.
- Salida:
  - Ncv *[Número entero]*: número de puntos de articulación intermedios entre el nodo de inicio y el nodo destino.

3.2.4.2 Herramientas de encaminamiento

Se han incluido en este apartado las funciones para obtención de la tabla de encaminamiento de un diseño mediante OSPF, y la que calculan el tráfico cursado y *hops* a partir del tráfico ofrecido y dicha tabla de encaminamiento.

La segunda bien podría tratarse como una función de análisis, pero se ha incluido aquí desde el punto de vista de que es una herramienta general de transformación de tráfico mediante encaminamiento, que aprovecha este seguimiento de la tabla de encaminamiento para devolver opcionalmente el número de saltos. Además, tanto una como otra serán herramientas necesarias para cualquier análisis (como se verá en el capítulo 4), pero no son funciones analíticas en sí.

`36)[routingTables, links] = OSPFrouting (Cost, tableType)`

Función: Crea la tabla de encaminamiento a partir de los "costes" o longitudes que se le pasen, que representan los gastos de recorrido entre los nodos.

Parámetros:

- Entrada:
  - Cost *[Matriz tipo Traffic o Links]*: Matriz con el coste directo entre cada par de los nodos a enrutar (coste sólo entre los nodos que tienen un enlace directo).
- Salida:
  - **RoutingTables**.

- Links *[Matriz NxN]*: Matriz con el coste total para moverse entre cada par de los nodos (coste de cualquier nodo a cualquier otro) pasando por sus rutas mínimas (calculadas mediante Open Shortest Path First).

37)[TrafficPerLink, Hops]=trafficLoad(trafficTable, routingTables)

Función: Transforma el tráfico ofrecido de la tabla de tráficos en tráfico cursado, es decir, en el tráfico total que pasa por cada enlace, para un encaminamiento dado. Además calcula el número de saltos entre cada par de nodos.

Parámetros:

o Entrada:

- **TrafficTable**.
- **RoutingTables**.

o Salida:

- TrafficPerLink *[Matriz de NxN]*: Matriz de tráfico cursado, indica en cada enlace usado en el ruteo cuál es el tráfico total.
- Hops *[Matriz de NxN]*: Matriz con el total de saltos que hay que dar desde cualquier nodo para llegar a cualquier otro, tal y como está diseñado el encaminamiento.

3.2.4.3 Herramientas de forma

Estas son funciones internas usadas en la mayor parte de las demás funciones, cuya utilidad es ahorrar líneas de código automatizando los procedimientos de conversión de forma de las matrices utilizadas. La más utilizada es la función convert, que es capaz de pasar rápidamente una matriz de costes o de tráfico entre sus posibles formatos, y las otras sirven para hacer matrices cuadradas o simétricas, formas a menudo requeridas por los algoritmos de las funciones del resto de los bloques.

38)[out]=convert (table, con)

Función: convierte matrices de tráfico o matrices de costes entre sus formatos posibles: *traffic*, *links*, *traffic multispeed,* y *links multispeed.*

Parámetros:

o Entrada:

- Table *[Matriz de NxN] o [Matriz de Nx3]*:
- Con: parámetro de conversión hacia:
  - 0 para convertir a *[Matriz de Nx3]*, (*links*)
  - 1 para convertir a *[Matriz de NxN]* (*traffic*)
  - 2 para convertir de *links multispeed* a *traffic multispeed.*
  - 3 para convertir de *traffic multispeed* a *links multispeed.*

o Salida:

- Out: La tabla convertida.

39)[out]=makeSquare (matrix,[F])

Función: devuelve una matriz cuadrada a partir de una matriz rectangular, rellenando con ceros las filas o columnas que le falten para serlo.

Parámetros:

- o Entrada:
  - matrix *[Matriz de MxN]*.
  - F *[Número entero]*: número de filas (y columnas) hacia el cual se quiere cuadrar (opcional).
- o Salida:
  - Out: *[Matriz de KxK]* , donde K es el max{M,N}, si no se ha utilizado el parámetro F.
    o bien *[Matriz de FxF]*, si se especifica.

40)[out]=makeSimetric(matrix)

Función: devuelve una matriz simétrica a partir de cualquier matriz cuadrada.

Parámetros:

- o Entrada:
  - matrix *[Matriz de MxM]*: Matriz cuadrada de entrada.
- o Salida:

  Out: *[Matriz de MxM]* : Matriz simétrica.

Capítulo 4
Resultados

4.1 Presentación

Este capítulo complementa al tercero en cuanto al aprendizaje del manejo de la herramienta, puesto que siempre es una rápida manera de asimilar las posibilidades de que se dispone el poder ver un conjunto de los problemas posibles a resolver. Aquí se van a exponer ejemplos de uso de la herramienta, mientras se comentarán los resultados que ésta devuelve.

El propósito de este capítulo es tanto didáctico, como para presentar los resultados esperados del trabajo para el cual se han diseñado todo el conjunto de algoritmos. Se comprobará la coherencia de dichos resultados con la teoría de redes, así como aspectos que puedan deducirse de los ejemplos, que nos ayuden a elaborar mejores diseños.

Los resultados se presentarán en el mismo orden que se han expuesto los conocimientos en el capítulo 2, para que el contraste sea más claro, y en todo momento se especificará que secuencia de comandos de ejecución de las funciones del capítulo 3 que se necesitan para llevarlos acabo. Así pues para un seguimiento claro de los ejemplos será bueno consultar dichos capítulos, lo cual servirá también para afianzar el conocimiento de la herramienta, siguiendo el objetivo didáctico de este proyecto.

Comenzamos con la presentación.

4.2 Contenidos

Se ha clasificado los tipos de problemas en dos básicamente:

- Generación de topologías, tráfico y normalización: Corresponde a toda la parte de generadores del capítulo 2.5.
- Algoritmos de optimización: Corresponde a todo el resto de la teoría (capítulo 2).

4.2.1 Generación de topologías, tráfico y normalización

4.2.1.1 Generación de una tabla de tráficos

<u>Problema número 1</u>

Crear un conjunto de nodos, con sus respectivas poblaciones, y generar una tabla de tráficos entre dichos nodos, asignándoles un nivel aleatorio entre 1 y 10, y teniendo en cuenta que la matriz de niveles es:

$$levelMatrix = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 2 & 2 & 2 & 2 & 2 & 2 \\ 1 & 1 & 1 & 2 & 2 & 3 & 3 & 3 & 3 & 3 \\ 1 & 1 & 1 & 2 & 2 & 3 & 3 & 5 & 5 & 5 \\ 1 & 1 & 3 & 3 & 3 & 3 & 4 & 5 & 5 & 5 \\ 1 & 1 & 4 & 4 & 4 & 4 & 4 & 5 & 6 & 6 \\ 2 & 4 & 4 & 4 & 4 & 4 & 4 & 5 & 6 & 6 \\ 5 & 5 & 5 & 5 & 5 & 5 & 5 & 5 & 6 & 7 \\ 5 & 5 & 5 & 6 & 6 & 6 & 6 & 6 & 6 & 7 \\ 6 & 6 & 6 & 6 & 7 & 7 & 7 & 7 & 7 & 1 \end{pmatrix}$$

La matriz nos indica que, por ejemplo, el tráfico entre un nodo de nivel 5 con otro de nivel 3 será multiplicado por levelMatrix(5,3)=3 en un sentido, y por levelMatrix(3,5)=2 en el otro sentido. Esta matriz de niveles está guardada en la variable levelMatrix.

Hay muchos parámetros no especificados, así que aquí escogemos los restantes (semillas, *offsets*, potencias, poblaciones, etc...) al azar. Una secuencia de comandos que resuelve el problema es la siguiente:

```
levelIndicator=ceil(rand(10,1)*10);
[sitesTable]=randomSiteGenerator(10,10000,4);
[trafficTable] = trafficGenerator(sitesTable,levelIndicator,levelMatrix,1,0.65,10,1,50,1,2);
```

- Con el primer comando creamos un array de 10 elementos aleatorios entre 1 y 10, que serán los niveles correspondientes a cada nodo.
- Con el segundo, creamos una tabla de 10 sitios de un máximo de 10000 habitantes, con la semilla 4 (por ejemplo).
- Con el tercero, generamos una tabla de tráficos con la sitesTable generada, el levelIndicator creado en la primera línea, y los parámetros:
    i. Alpha=1 (sin escalar).
    ii. Rf=0.65 (grado de aleatoriedad moderado).
    iii. PopOff=10 (mínimo de 10 habitantes por población).
    iv. PopPower=1.
    v. DistOff=50 (mínimo de 50 km. entre poblaciones).
    vi. DistPower=1.
    vii. Seed=2 (semilla aleatoria escogida).

La salida del programa:

sitesTable =

89	43	6297
25	82	3951
57	66	9319
14	64	5285
12	84	7125
53	94	2070
48	41	2199
47	83	9382
50	48	9576
23	1	6843

trafficTab le =

0	1.5050	0.8056	0.5755	1.2670	0.3545	0.3658	0.6550	0.3637	0.9157
0.8094	0	1.7915	1.3137	1.0039	0.7846	1.4301	1.6991	1.0529	0.6589
2.0432	0.7857	0	1.4286	1.3752	1.5879	1.4035	0.7896	2.1614	0.4445
0.9212	0.6816	1.6263	0	1.7457	0.4669	0.6257	0.4245	0.7923	0.9282
0.7281	1.5169	0.5442	1.7080	0	1.4661	0.7378	0.5471	1.7204	0.7713
0.6208	0.5373	0.8518	0.7607	0.5837	0	0.3982	0.4692	0.5918	0.4601
0.2087	1.2443	0.7520	0.8635	1.5995	0.8314	0	0.1971	0.6593	0.3095
0.6348	1.0062	0.3846	0.9609	0.6445	0.3492	0.4829	0	0.2739	0.7875
0.5132	1.0646	1.6932	0.9279	1.0937	0.2609	0.4937	0.2086	0	0.2657
0.6131	1.8771	1.0689	0.5540	1.5991	1.0128	0.5095	1.0370	0.9290	0

Si observamos, hemos creado una tabla de sitios (un ejemplo, el sitio 1 estaría colocado en la posición (89,43) del plano, y tendría una población de 6297 habitantes); y una tabla de tráficos, que se comprueba que es asimétrica, y que están cercanos a la unidad. Posteriormente podremos terminar de perfilar la *trafficTable* mediante las normalizaciones que me llevarán los datos de tráfico a niveles realistas.

4.2.1.2 Normalizaciones

Problema número 2

Hacer diferentes normalizaciones de la tabla de tráficos anterior de forma que obtenga en cada una de ellas un objetivo de los siguientes:

- *Un tráfico total de 1200 paq/s.*
- *Un tráfico de entrada de cada nodo correspondiente a los valores de la tabla siguiente:*

100	150	120	150	200	200	150	120	150	100

Tabla 4-1. TRAFIN

- *Un tráfico de salida de cada nodo correspondiente a los valores de la tabla siguiente:*

150	100	120	200	140	210	120	150	120	130

Tabla 4-2. TRAFOUT

- *Un tráfico que tenga simultáneamente los tráficos de entrada y salida anteriores.*

La secuencia de comandos que resuelve el problema es la siguiente:

```
%normalizaciones
Ttotal=1200;
TRAFOUT=[100;150;120;150;200;200;150;120;150;100];
TRAFIN=[150;100;120;200;140;210;1200;150;120;130];
disp('Normalizacion total');
trafficNormTotal=totalNormalizator(trafficTable, Ttotal)
disp('Normalizacion por filas');
trafficNormR=rowNormalizator(trafficTable,TRAFOUT)
disp('Normalizacion por columnas');
trafficNormC=columNormalizator(trafficTable,TRAFIN)
disp('Normalizacion por filas y columnas');
trafficNormRC=rowColumNormalizator(trafficTable,TRAFIN, TRAFOUT)
```

La salida del programa:

Normalizacion total

trafficNormTotal =

0	9.3232	9.9814	8.9124	7.8489	2.7453	3.7764	20.289	3.7548	28.362
4.1787	0	18.497	6.7816	5.1826	4.0506	7.3825	21.052	5.4355	6.8029
31.643	4.0562	0	22.125	7.0992	24.592	7.2456	14.674	33.474	11.473
7.1335	3.5185	16.791	0	9.0122	3.6154	4.8452	9.8621	6.1354	35.937
3.7585	7.8307	5.619	8.8175	0	7.5685	3.8089	6.7786	8.8813	7.9633
4.8076	2.7739	8.7949	11.782	3.0133	0	3.0833	10.901	4.5825	10.688
3.2323	7.7085	4.6585	8.9157	9.9087	8.5836	0	6.1055	10.211	3.1953
39.326	10.389	15.883	39.686	6.6546	14.423	7.4783	0	16.971	40.653
7.9482	6.5954	20.979	19.161	6.7752	2.6933	7.6467	19.381	0	8.2311
23.738	48.451	27.59	21.449	41.276	39.212	19.727	40.151	35.97	0

Normalizacion por filas

trafficNormR =

0	9.8146	10.507	9.3821	8.2625	2.89	3.9754	21.358	3.9526	29.857
7.8979	0	34.96	12.818	9.7954	7.6558	13.953	39.789	10.273	12.858
24.281	3.1125	0	16.978	5.4475	18.87	5.5599	11.26	25.686	8.8041
11.048	5.4493	26.006	0	13.958	5.5994	7.5042	15.274	9.5025	55.659
12.318	25.663	18.415	28.897	0	24.804	12.483	22.215	29.106	26.098
15.912	9.181	29.11	38.995	9.9735	0	10.205	36.08	15.167	35.377
7.7551	18.495	11.177	21.391	23.774	20.594	0	14.649	24.499	7.6663
24.648	6.5111	9.9547	24.873	4.1708	9.0398	4.6871	0	10.636	25.479
11.993	9.9517	31.655	28.912	10.223	4.0638	11.538	29.244	0	12.42
7.9774	16.282	9.2719	7.2082	13.871	13.178	6.6296	13.493	12.088	0

Normalizacion por columnas

trafficNormC =

0	9.2634	9.2999	12.074	11.355	5.3637	6.9724	20.399	3.5926	24.05
4.9839	0	17.234	9.1873	7.4977	7.9139	13.631	21.166	5.2008	5.7687
37.74	4.0302	0	29.974	10.271	48.047	13.378	14.754	32.029	9.7292
8.5081	3.4959	15.645	0	13.038	7.0636	8.9458	9.9154	5.8705	30.474
4.4828	7.7804	5.2354	11.945	0	14.787	7.0325	6.8152	8.4978	6.7526
5.734	2.7561	8.1944	15.961	4.3594	0	5.6927	10.96	4.3846	9.0635
3.8551	7.6591	4.3404	12.078	14.335	16.771	0	6.1385	9.7701	2.7095
46.904	10.322	14.799	53.764	9.6274	28.18	13.807	0	16.238	34.473
9.4798	6.5531	19.547	25.959	9.8017	5.262	14.118	19.486	0	6.9798
28.312	48.14	25.706	29.058	59.715	76.612	36.423	40.367	34.417	0

Normalizacion por filas y columnas

trafficNormRC =

0	11.243	7.7457	11.582	13.584	7.1084	7.1083	17.731	4.1043	19.778
10.386	0	23.407	14.372	14.627	17.103	22.661	30.001	9.6889	7.7361
25.29	2.5651	0	15.078	6.4429	33.391	7.1519	6.7249	19.188	4.1956
15.891	6.2019	19.045	0	22.797	13.683	13.33	12.597	9.8025	36.629
14.515	23.928	11.049	29.034	0	49.656	18.167	15.01	24.598	14.071
22.263	10.164	20.737	46.519	15.845	0	17.634	28.945	15.22	22.647
8.4444	15.935	6.1966	19.86	29.395	38.098	0	9.1459	19.132	3.8194
29.216	6.1068	6.0079	25.139	5.6138	18.204	6.8615	0	9.0422	13.819
15.526	10.194	20.866	31.915	15.029	8.9381	18.448	21.708	0	7.3568
8.4463	13.641	4.9984	6.5074	16.677	23.704	8.669	8.1916	9.1793	0

Ahora podemos hacer alguna comprobación al azar para ver si realmente cumple lo especificado, por ejemplo:

- Sumar todos los tráficos de la tabla trafficNormTotal (tiene que dar 1200).
- Sumar la fila 2 de la tabla trafficNormR (tiene que dar 150 según la tabla).
- Sumar la columna 3 de la tabla trafficNormC (tiene que dar 120 según la tabla).

- Sumar la fila 6 y la columna 10 de la tabla trafficNormRC (tiene que dar, en el primer caso 200, en el segundo 130 según las tablas).

Comandos y resultados:

```
%comprobacion de las normalizaciones:

fprintf('\nSuma de la matriz de trafico normalizado total: %f',sum(sum(trafficNormTotal)));

fprintf('\nSuma de la fila 2 en la matriz de trafico normalizado por filas: %f',sum(trafficNormR(2,:)));

fprintf('\nSumas de la columna 3 en lamatriz de trafico normalizado por columnas: %f',sum(trafficNormC(:,3)));

fprintf('\nSuma de la fila 6 y columna 10 en la matriz de trafico normalizado por filas y columnas: %f y
%f',sum(trafficNormRC(6,:)),sum(trafficNormRC(:,10)));
```

➔

```
Suma de la matriz de trafico normalizado total: 1200.000000

Suma de la fila 2 en la matriz de trafico normalizado por filas: 150.000000

Sumas de la columna 3 en lamatriz de trafico normalizado por columnas: 1200.000000

Suma de la fila 6 y columna 10 en la matriz de trafico normalizado por filas y columnas: 199.972714 y 130.050794
```

Como se ve, los resultados son bastante razonables. No obstante, la exactitud de la última normalización depende del número de bucles con que se programe la función. Si se requiere, dicha exactitud puede aumentarse variando la línea:

```
while((count<100)&((max_scale<.999)|(max_scale>1.001)))
```

Bien aumentando el *count* máximo, o las cifras de la precisión en la escala se lograría un resultado más exigente.

4.2.1.3 Generación de tarifarios

Problema número 3

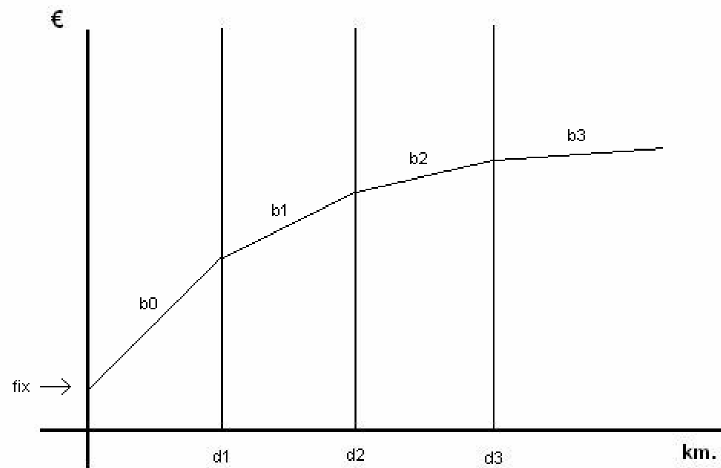
Generar 3 tarifarios para el conjunto de nodos anterior, de forma que el coste respecto a la distancia entre los nodos aumente:

- *1) De forma lineal $C(i,j,k) = a_k + b_k \cdot d$, con los parámetros de linealidad siguientes para cada tipo de línea:*

	A	B
Línea 1	30	10
Línea 2	40	7

Tabla 4-3. LineTypesTable lineal

- *2) De forma lineal a trozos, como en la figura siguiente:*

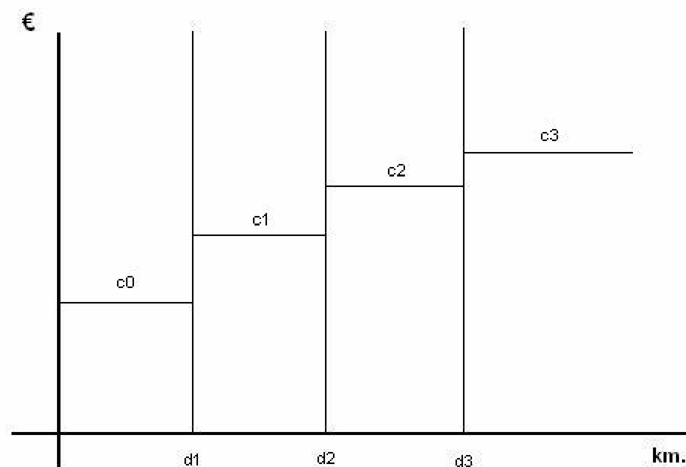


Es decir, crear un tarifario para 3 líneas diferentes de 4 trozos cada una, con los parámetros para cada línea:

	Fix	B0	D1	B1	D2	B2	D3	B3
Línea 1	10	4	15	3	25	2	30	1
Línea 2	5	5	15	4	30	3	45	2
Línea 3	0	7	20	6	30	5	45	4

Tabla 4-4. LineTypesTable lineal a trozos

- *De forma constante a trozos, como en la figura siguiente:*



Es decir, crear un tarifario para 2 líneas diferentes de 6 trozos cada una, con los parámetros para cada línea:

	C0	D1	C1	D2	C2	D3	C3	D4	C4	D5	C5
Línea 1	10	10	20	20	30	30	40	40	50	50	60
Línea 2	5	10	10	20	20	30	30	40	40	50	50

Tabla 4-5. LineTypesTable constante a trozos

La secuencia de comandos que resuelve el problema es la siguiente:

```
%generadores de tarifas
linetypesTable=[30 10;40 7];
disp('Tabla de tarifas con el generador lineal');
[tariffTable]=costGenerator_linear(sitesTable,linetypesTable)

linetypesTable=[10 4 15 3 25 2 30 1;5 5 15 4 30 3 45 2;0 7 20 6 30 5 45 4];
disp('Tabla de tarifas con el generador lineal a tramos (para 4 tramos):');
[tariffTable2] = costGenerator_piecewise_lin(sitesTable,linetypesTable)

linetypesTable=[10 10 20 20 30 30 40 40 50 50 60; 5 10 10 20 20 30 30 40 40 50 50];
disp('Tabla de tarifas con el generador constante a trozos (con 5 trozos):');
[tariffTable3] = costGenerator_piecewise_const(sitesTable,linetypesTable)
```

La salida del programa:

Tabla de tarifas con el generador lineal.

tariffTable(:,:,1) =

$1.0e+003*$

0	0.7795	0.4241	0.8088	0.9024	0.6543	0.4405	0.6100	0.4232	0.8123
0.7795	0	0.3878	0.2410	0.1615	0.3346	0.5001	0.2502	0.4520	0.8402
0.4241	0.3878	0	0.4605	0.5147	0.3128	0.2957	0.2272	0.2231	0.7636
0.8088	0.2410	0.4605	0	0.2310	0.5220	0.4405	0.4108	0.4240	0.6664
0.9024	0.1615	0.5147	0.2310	0	0.4520	0.5908	0.3801	0.5535	0.8673
0.6543	0.3346	0.3128	0.5220	0.4520	0	0.5624	0.1553	0.4910	1.0072
0.4405	0.5001	0.2957	0.4405	0.5908	0.5624	0	0.4501	0.1028	0.5017
0.6100	0.2502	0.2272	0.4108	0.3801	0.1553	0.4501	0	0.3813	0.8844
0.4232	0.4520	0.2231	0.4240	0.5535	0.4910	0.1028	0.3813	0	0.5720
0.8123	0.8402	0.7636	0.6664	0.8673	1.0072	0.5017	0.8844	0.5720	0

tariffTable(:,:,2) =

0	564.6265	315.8568	585.1917	650.6472	476.9817	327.3413	446.0000	315.2344	587.6130
564.6265	0	290.4396	187.6652	132.0706	253.2416	369.0745	194.1590	335.4133	607.1728
315.8568	290.4396	0	341.3254	379.2654	237.9899	225.9946	178.0616	175.1925	553.4871
585.1917	187.6652	341.3254	0	180.6983	384.4256	327.3413	306.5521	315.7680	485.4773
650.6472	132.0706	379.2654	180.6983	0	335.4133	432.5621	285.1000	406.4151	626.0802
476.9817	253.2416	237.9899	384.4256	335.4133	0	412.6473	127.7097	362.6841	724.0329
327.3413	369.0745	225.9946	327.3413	432.5621	412.6473	0	334.0833	90.9608	370.1893
446.0000	194.1590	178.0616	306.5521	285.1000	127.7097	334.0833	0	285.8984	638.0803
315.2344	335.4133	175.1925	315.7680	406.4151	362.6841	90.9608	285.8984	0	419.4232
587.6130	607.1728	553.4871	485.4773	626.0802	724.0329	370.1893	638.0803	419.4232	0

Tabla de tarifas con el generador lineal a tramos (para 4 tramos):

tariffTable2(:,:,1) =

```
        0 334.7866 192.6325 346.5381 383.9413 284.7038 199.1950 267.0000 192.2768 347.9217
 334.7866        0 178.1084 106.5701  62.6118 156.8524 223.0425 112.1363 203.8076 359.0988
 192.6325 178.1084        0 207.1859 228.8659 146.4214 137.8533  98.3385  95.8792 328.4212
 346.5381 106.5701 207.1859        0 100.5985 231.8146 199.1950 187.3155 192.5817 289.5584
 383.9413  62.6118 228.8659 100.5985        0 203.8076 259.3212 175.0571 244.3800 369.9030
 284.7038 156.8524 146.4214 231.8146 203.8076        0 247.9413  60.1199 219.3909 425.8759
 199.1950 223.0425 137.8533 199.1950 259.3212 247.9413        0 203.0476  39.1204 223.6796
 267.0000 112.1363  98.3385 187.3155 175.0571  60.1199 203.0476        0 175.5133 376.7601
 192.2768 203.8076  95.8792 192.5817 244.3800 219.3909  39.1204 175.5133        0 251.8133
 347.9217 359.0988 328.4212 289.5584 369.9030 425.8759 223.6796 376.7601 251.8133        0
```

tariffTable2(:,:,2) =

```
        0 484.6799 265.8568 502.3072 558.4119 409.5557 277.3413 383.0000 265.2344 504.3826
 484.6799        0 240.4396 128.7602  70.7647 203.2416 317.0638 136.1817 285.4133 521.1481
 265.8568 240.4396        0 291.3254 325.7989 186.2742 172.5653 117.7847 114.5057 475.1318
 502.3072 128.7602 291.3254        0 120.7980 330.2220 277.3413 256.5521 265.7680 416.8377
 558.4119  70.7647 325.7989 120.7980        0 285.4133 371.4818 235.1000 349.0701 537.3545
 409.5557 203.2416 186.2742 330.2220 285.4133        0 354.4120  67.6498 311.5863 621.3139
 277.3413 317.0638 172.5653 277.3413 371.4818 354.4120        0 284.0833  41.4005 318.0194
 383.0000 136.1817 117.7847 256.5521 235.1000  67.6498 284.0833        0 235.8984 547.6402
 265.2344 285.4133 114.5057 265.7680 349.0701 311.5863  41.4005 235.8984        0 360.2199
 504.3826 521.1481 475.1318 416.8377 537.3545 621.3139 318.0194 547.6402 360.2199        0
```

tariffTable2(:,:,3) =

```
        0 724.4665 363.4893 753.8453 847.3531 599.2596 381.5363 555.0000 362.5113 757.3043
 724.4665        0 323.5480 153.1403  92.0706 265.0940 445.1064 164.2726 394.2209 785.2469
 363.4893 323.5480        0 403.5114 459.6648 239.4113 218.8479 138.0616 135.1925 708.5530
 753.8453 153.1403 403.5114        0 141.1970 467.0366 381.5363 348.8675 363.3497 611.3961
 847.3531  92.0706 459.6648 141.1970        0 394.2209 535.8030 315.1571 498.4501 812.2574
 599.2596 265.0940 239.4113 467.0366 394.2209        0 507.3533  87.7097 435.9772 952.1898
 381.5363 445.1064 218.8479 381.5363 535.8030 507.3533        0 392.1309  50.9608 446.6991
 555.0000 164.2726 138.0616 348.8675 315.1571  87.7097 392.1309        0 316.4117 829.4004
 362.5113 394.2209 135.1925 363.3497 498.4501 435.9772  50.9608 316.4117        0 517.0332
 757.3043 785.2469 708.5530 611.3961 812.2574 952.1898 446.6991 829.4004 517.0332        0
```

Tabla de tarifas con el generador constante a trozos (con 5 trozos):

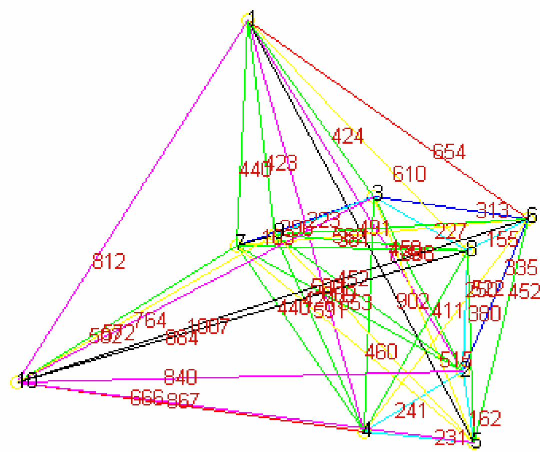
tariffTable3(:,:,1) =

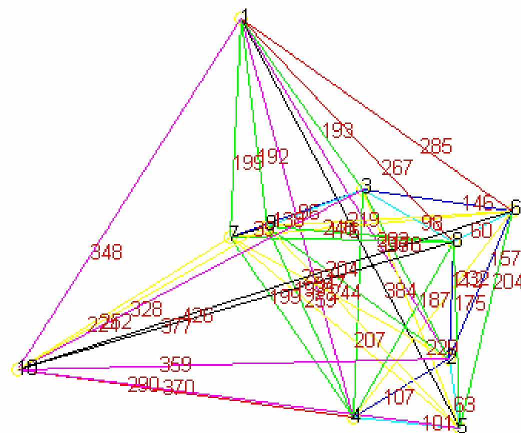
```
  0  60  40  60  60  60  50  60  40  60
 60   0  40  30  20  40  50  30  50  60
 40  40   0  50  50  30  30  20  20  60
 60  30  50   0  30  50  50  40  40  60
 60  20  50  30   0  50  60  40  60  60
 60  40  30  50  50   0  60  20  50  60
 50  50  30  50  60  60   0  50  10  50
 60  30  20  40  40  20  50   0  40  60
 40  50  20  40  60  50  10  40   0  60
 60  60  60  60  60  60  50  60  60   0
```

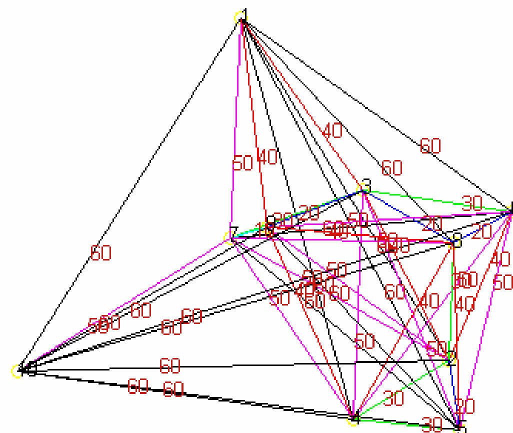
```
tariffTable3(:,:,2) =

   0  50  30  50  50  50  40  50  30  50
  50   0  30  20  10  30  40  20  40  50
  30  30   0  40  40  20  20  10  10  50
  50  20  40   0  20  40  40  30  30  50
  50  10  40  20   0  40  50  30  50  50
  50  30  20  40  40   0  50  10  40  50
  40  40  20  40  50  50   0  40   5  40
  50  20  10  30  30  10  40   0  30  50
  30  40  10  30  50  40   5  30   0  50
  50  50  50  50  50  50  40  50  50   0
```

Estos tarifarios pueden dibujarse mediante un *displayGraph*:

```
figure;
displayGraph (sitesTable,tariffTable,1,3,0,1); title('TariffTable lineal');
figure;
displayGraph (sitesTable,tariffTable2,1,3,0,1); title('TariffTable lineal a trozos');
figure;
displayGraph (sitesTable,tariffTable3,1,3,0,1); title('TariffTable constante a trozos');
```







Aquí se ha utilizado la función *displayGraph* con las opciones de mostrar sólo el primer tipo de línea, etiquetado completo, no dibujar flechas (los costes no son directivos), colorear. Si recordamos un poco el código de colores de la función displayGraph, puede comprobarse que los tarifarios generados con las funciones lineales son semejantes (con la diferencia de que el lineal a trozos nos permite especificar con mucho mejor detalle), y que con el generador constante a trozos (de filosofía parecida al "cobro por pasos"), hay un mayor número de enlaces de alto precio con respecto al resto de los enlaces (muchas más líneas oscuras y negras en la figura).

4.2.1.4 Linealización

Problema número 4

Linealizar la tabla de tarifas adjunta, basada en 6 distancias y sus respectivos costes, y crear con sus parámetros una tabla equivalente para un conjunto de 15 nodos de población máxima 5000 habitantes.

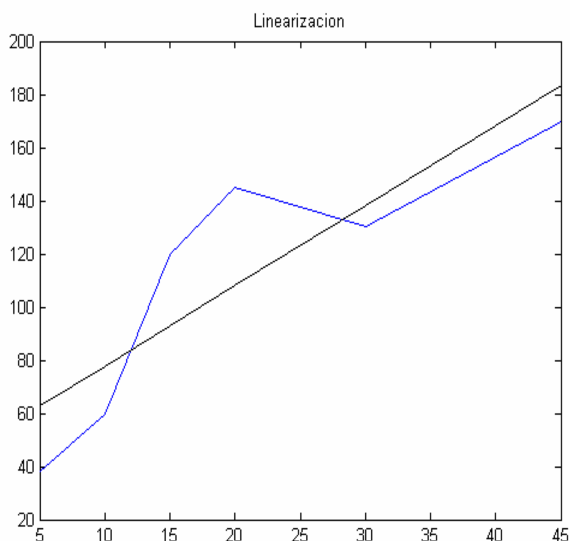
Dist	Cost
5	38
10	60
15	120
20	145
30	130
45	170

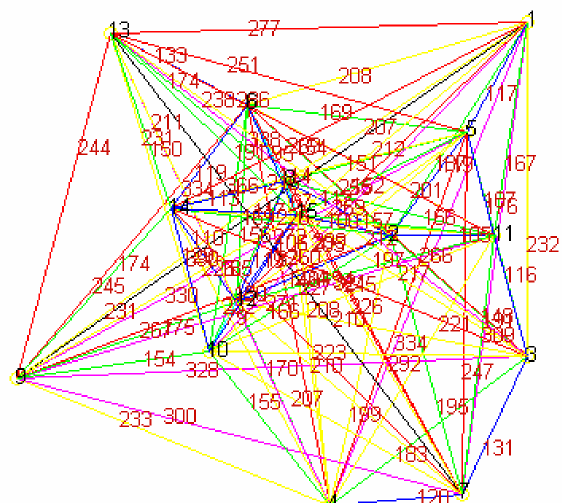
La secuencia de comandos que resuelve el problema es la siguiente:

```
table=[38 5; 60 10;120 15; 145 20; 130 30; 170 45];
[a,b]=lineariz(table,1);
[sitesTable]=randomSiteGenerator(15,5000,1);
[tariffTable]=costGenerator_linear(sitesTable,[a,b])
figure;
displayGraph (sitesTable,tariffTable,1,3,0,1); title('TariffTable lineal con los parametros lineales de la tabla anterior');
```

- Primero creamos la tabla. El formato en que tiene que estar la tabla, es (coste, distancia), como podemos observar haciendo "help lineariz".
- Linealizamos, poniendo a 1 el parámetro *graph*, para ver la salida gráfica de la linealización.
- Con el tercer comando generamos 15 nodos de 5000 habitantes como máximo cada uno, con la semilla aleatoria 1.
- Le pasamos al generador lineal de tarifas la *sitesTable* creada, y como *lineType*, los parámetros a y b resultantes de la linealización.
- Visualizamos con el displayGraph.

La salida del programa:





Mediante este sencillo ejemplo hemos obtenido una tabla de tarifas relacionada con unos parámetros reales. Este será un buen método para ahorrar costes a la hora de planificar tarifarios que se ajusten a los costes reales, sin tener que comprar o medir un número excesivo de datos, ilustrando la filosofía común de los generadores[4].

4.2.2 Algoritmos de optimización

4.2.2.1 MST y SPT

Problema número 5

A partir de un conjunto de 10 nodos de población máxima 10000, y un tarifario lineal a trozos generado con los datos de la tabla de líneas 4.4 del problema 3, encontrar las siguientes soluciones de coste y saltos mínimos:

- *Un MST mediante el algoritmo de Kruskal*
- *Un MST mediante el algoritmo de Prim, empezando en un nodo cualquiera.*
- *Un SPT mediante el algoritmo de Dijkstra, empezando en un nodo cualquiera.*

Representar y evaluar las soluciones de cada algoritmo. Buscar una solución intermedia entre el coste y el número de saltos mínimos.

Primeramente nos creamos la sitesTable y la tariffTable, como se ha hecho en los problemas anteriores. Para cambiar la topología respecto a los 3 primeros problemas, vamos a poner otra semilla aleatoria en el generador de sitios:

```
[sitesTable]=randomSiteGenerator(10,10000,1);
linetypesTable=[10 4 15 3 25 2 30 1;5 5 15 4 30 3 45 2;0 7 20 6 30 5 45 4];
[tariffTable] = costGenerator_piecewise_lin(sitesTable,linetypesTable)
```

Seguidamente resolvemos el problema. Para obtener los MST y SPT pedidos disponemos de las funciones kruskal.m, prim.m, y dijkstra.m. Vamos a escoger como nodo de inicio, por ejemplo, el 2.

Para evaluar el coste, ya que las linksTable que devuelven estas funciones son simétricas, habrá que sumar todos los costes y dividirlos entre dos (solo hace falta una línea entre *un* origen y un destino, mientras que en la tabla vendrán *dos* líneas por cada enlace, por ser simétrica). Seguidamente vamos a mostrar la topología resultante.

La secuencia de comandos que resuelve el problema es la siguiente:

```
MSTtable=kruskal(tariffTable(:,:,1));
```

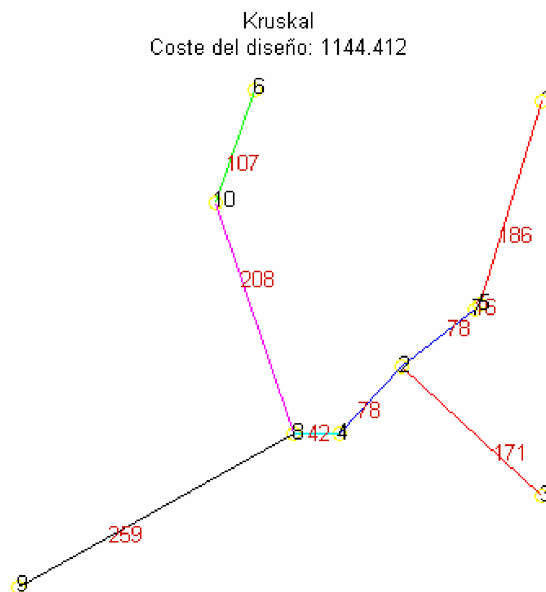
[4] Obviamente en este ejemplo se utiliza un linealizador básico, que sólo valdrá para entornos pequeños. Para una mayor exactitud, en entornos reales se deberá utilizar un linealizador a trozos, seguido de un generador lineal a trozos. Esto se nombrará como propuesta en el capítulo 5, en líneas futuras.

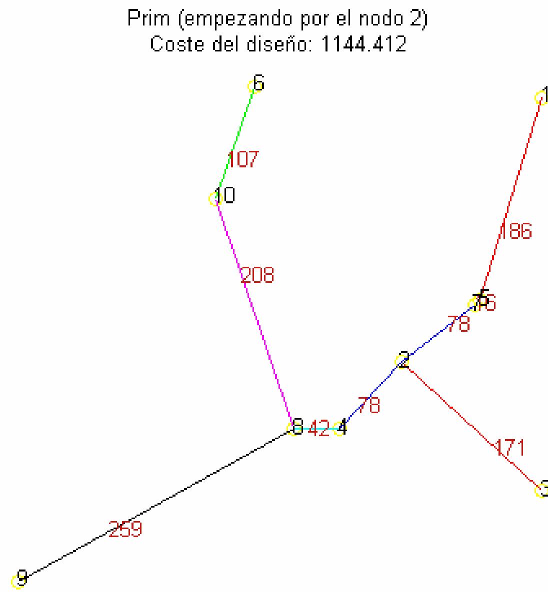
```
figure
displayGraph (sitesTable,convert(MSTtable,1),0,3,0,1)
title(sprintf('\nKruskal\nCoste del diseño: %.3f',sum(MSTtable(:,3))/2));

startNode=2;
MSTtable2=prim(tariffTable(:,:,1),startNode);
figure
displayGraph (sitesTable,convert(MSTtable2,1),0,3,0,1)
title(sprintf('\nPrim (empezando por el nodo %i)\nCoste del diseño: %.3f',startNode,sum(MSTtable2(:,3))/2))

SPTtable=dijkstra(tariffTable(:,:,1),2);
figure
displayGraph (sitesTable,convert(SPTtable,1),0,3,0,1)
title(sprintf('\nDijkstra (empezando por el nodo %i)\nCoste del diseño: %.3f',startNode,sum(SPTtable(:,3))/2))
```

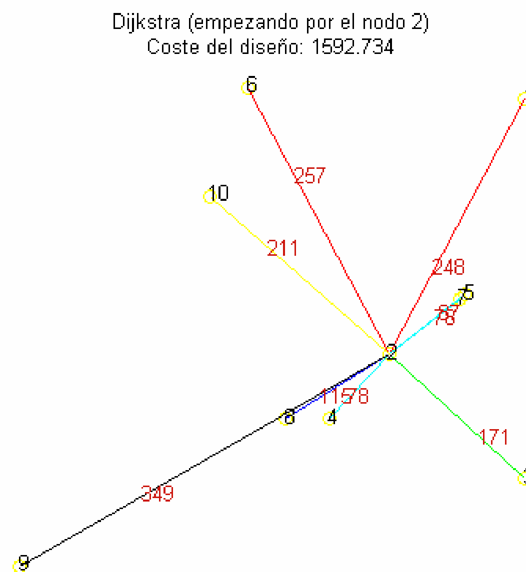
Comentarios de uso:

- Al OSPFrouting se le pasa la matriz topología, y el parámetro 0 que le indica que está en formato Links. Como solo funcionan para un tipo de línea se ha especificado para K=1 (el primer tipo de línea), mediante la entrada *tariffTable(:,:,1)*. La salida de los algoritmos es una linksTable tipo Links.
- El displayGraph funciona con LinksTable tipo Traffic, luego hay que convertirla haciendo uso de la función *convert(linkstable,1)*. (Ver sección 3.1, preliminares respecto a los tipos de datos manejados).
- El coste se obtiene sumando la tercera columna de las linksTable (correspondiente a los costes de cada línea) y dividiendo el resultado por dos, según se ha explicado anteriormente.

La salida del programa:







Aquí observamos, que tal y como se esperaba según la teoría, tanto el algoritmo de Prim como el de Kruskal nos llevan a la misma solución, con la ventaja en caso de Kruskal de que no hace falta especificar ningún nodo de inicio[5].

---

[5] Se puede comprobar que efectivamente, variando el nodo de comienzo, siempre se llega a la misma solución. Un posible código para comprobarlo es el siguiente:

```
for i=1:10
   startNode=i;
   MSTtable2=prim(tariffTable(:,:,1),startNode);
   figure;  displayGraph (sitesTable,convert(MSTtable2,1),0,3,0,1)
   title(sprintf('\nPrim (empezando por el nodo %i)\nCoste del diseño: %.3f',startNode,sum(MSTtable2(:,3))/2))
end
```

También se observa que el coste es considerablemente más alto en el caso de Dijkstra, porque lo que este último algoritmo pretende minimizar es el número de saltos. Para evaluar el número medio de saltos de cada diseño vamos a hacer uso de funciones de encaminamiento (OSPFrouting y trafficLoad), que nos calculan el número de saltos y el tráfico cursado para un algoritmo de ruteo, el OSPF.

Para cada *linksTable* solución, el código para obtener la media de saltos sería:

```
[trafficTable] = trafficGenerator(sitesTable,levelIndicator,levelMatrix,1,0.65,10,1,50,1,2);
trafficTable=totalNormalizator(trafficTable, 1200);
routingTables = OSPFrouting (linksTable,0);
[trafficPerLink, hops]=trafficLoad(trafficTable, routingTables);
[HopsMean, Tmean, U] = calculateParams(trafficTable, trafficPerLink , hops, 128000, 32)
```

Comentarios de uso:

- Al OSPFrouting se le pasa la matriz topología, y el parámetro 0 que le indica que la linksTable está en formato Links.
- Generamos una matriz de tráfico ofrecido con los mismos parámetros que en el primer problema, ya que necesitaremos el tráfico ofrecido para obtener con él el número medio de saltos[6], cuya ecuación es:

$$\overline{Hops} = \frac{\sum_{i,j} \gamma_{ij} \cdot Hops(i,j)}{\sum_{i,j} \gamma_{ij}}$$

- Mediante trafficLoad, al que le pasamos el tráfico ofrecido anterior, y la tabla de ruteo, obtenemos el tráfico cursado en cada enlace y el número de saltos de cada flujo posible entre cualquier par de nodos
- Finalmente, para calcular los parámetros de utilización, retardo, y número medio de saltos, usamos la función *calculateParams*, a la que le pasamos el tráfico ofrecido, el cursado, los saltos (*hops*) y dos parámetros:
  - i. C=128000: Capacidad de todos los enlaces. Suponemos una capacidad de 128Kbps.
  - ii. L=32: Tamaño medio de paquete, que se ha elegido de 32 bits.

La salida del programa:

- Para Prim y Kruskal*:*

```
HopsMean =
2.9668
Tmean =
2.6648e-004
```

[6] Si sólo nos interesaran los saltos entre cada par de enlaces, y no el número medio, podríamos ahorrarnos la función calculateParams y la generación de una matriz de tráfico, pasándole a la función *trafficLoad* una matriz cualquiera (por ejemplo, una matriz unidad de la misma longitud que la tabla de ruteo).

U=

0	0	0	0	0.024143	0	0	0	0	0
0	0	0.03165	0.072591	0	0	0.064249	0	0	0
0	0.038605	0	0	0	0	0	0	0	0
0	0.088399	0	0	0	0	0	0.072631	0	0
0.031981	0	0	0	0	0	0.03625	0	0	0
0	0	0	0	0	0	0	0	0	0.015556
0	0.046871	0	0	0.052974	0	0	0	0	0
0	0	0	0.10103	0	0	0	0	0.030509	0.053344
0	0	0	0	0	0	0	0.02419	0	0
0	0	0	0	0	0.027782	0	0.077282	0	0

- Para Dijkstra:

HopsMean =

1.8517

Tmean =

2.5975e-004

U=

0	0.024143	0	0	0	0	0	0	0	0
0.031981	0	0.03165	0.036602	0.023904	0.027782	0.016589	0.037409	0.030509	0.038642
0	0.038605	0	0	0	0	0	0	0	0
0	0.024015	0	0	0	0	0	0	0	0
0	0.015018	0	0	0	0	0	0	0	0
0	0.015556	0	0	0	0	0	0	0	0
0	0.015934	0	0	0	0	0	0	0	0
0	0.048185	0	0	0	0	0	0	0	0
0	0.02419	0	0	0	0	0	0	0	0
0	0.074806	0	0	0	0	0	0	0	0

La matriz de utilización nos indica la utilización de cada enlace del diseño, correspondiente a cada valor no cero de la matriz. Vemos que en el caso de Dijkstra los únicos enlaces utilizados son los que van o vienen del nodo 2. También vemos que al considerar tráfico asimétrico, los valores de utilización en un mismo enlace varían para un sentido y para otro, podemos interpretar que esta sería la utilización correspondiente a cada enlace *simplex* en un diseño direccional.

Los resultados nos muestran que el número medio de saltos disminuye considerablemente en el diseño mediante Dijkstra, como era de esperar, y el tiempo de retardo medio también es un poco más bajo. Concluimos que mediante este algoritmo, se invierte una cierta cantidad de coste en conseguir estas características, para ser más exactos, en este ejemplo se han invertido 1592.734-1144.412= 448.322 unidades monetarias (un 40% extra del coste mínimo) en bajar el número medio de saltos de 2.96 a 1.85 (un 37.5%) y reducir el retardo medio de 0.266 ms. a 0.259 ms. (6.73 nanosegundos, un 2.53%).

Lo siguiente que se puede buscar, tal y como pide el problema, es una solución intermedia. Esta solución viene de mano del algoritmo de Prim-Dijkstra. Como ejemplo vamos a elegir un α intermedio (por ejemplo, 0.3), y empezaremos en el mismo nodo de inicio que en los algoritmos anteriores.

Secuencia de comandos:

alfa=0.3;

startNode=2;

linksTable=prim_dijkstra(tariffTable(:,:,1),startNode,alfa)

figure

displayGraph (sitesTable,convert(linksTable,1),0,3,0,1)

title(sprintf('\nPrim-Dijkstra con alpha=%.2f (empezando por el nodo %i)\nCoste del diseño: %.3f',alfa,startNode,sum(linksTable(:,3))/2));

routingTables = OSPFrouting (linksTable,0);

[trafficPerLink, hops]=trafficLoad(trafficTable, routingTables);

[HopsMean, Tmean, U] = calculateParams(trafficTable, trafficPerLink , hops, 128000, 32)

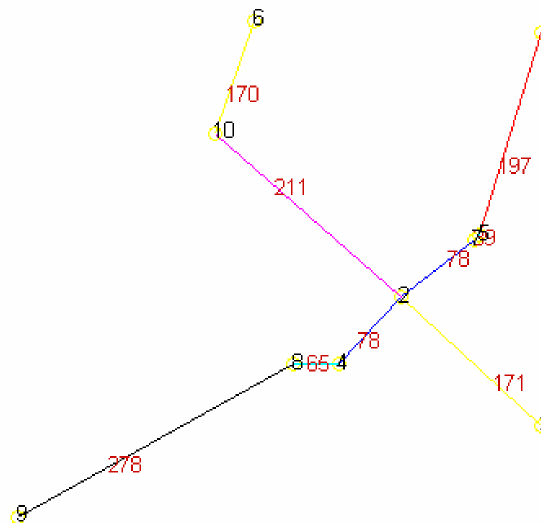
La salida del programa:

HopsMean =

2.7472

Tmean =

2.6424e-004



U=

0	0	0	0	0.024143	0	0	0	0	0
0	0	0.03165	0.077187	0	0	0.064249	0	0	0.053344
0	0.038605	0	0	0	0	0	0	0	0
0	0.069057	0	0	0	0	0	0.059039	0	0
0.031981	0	0	0	0	0	0.03625	0	0	0
0	0	0	0	0	0	0	0	0	0.015556

0	0.046871	0	0	0.052974	0	0	0	0	0
0	0	0	0.063495	0	0	0	0	0.030509	0
0	0	0	0	0	0	0	0.02419	0	0
0	0.077282	0	0	0	0.027782	0	0	0	0

En este caso el coste del diseño es 1288.229, sólo 143.8170 unidades monetarias por encima del coste mínimo (un 12.5%), y el número medio de saltos es 2.74 frente a 2.96 en los MST. Se concluye que son valores intermedios, pero más ajustados al diseño MST que al SPT. Si se quiere lograr un diseño más intermedio, habría que subir el α. Vamos a probar con algunos valores, que enumeramos en una tabla:

α	Coste del Diseño	HopsMean	Tmean
0 (MST)	1144.412	2.96	2.6648e-004
0.3	1288.229	2.74	2.6424e-004
0.5	1398.074	2.74	2.6424e-004
0.7	1520.446	2.74	2.6424e-004
0.8	1567.131	2.3667	2.6304e-004
0.9	1589.203	2.1615	2.6127e-004
1 (SPT)	1592.734	1.85	2.5975e-004

Se observa que en este diseño, y empezando con el nodo 2, el coste varía progresivamente con el α, mientras que los parámetros de número medio de saltos y retardo no varían en un amplio rango de α (desde 0.3 a 0.7), y después varían muy rápidamente para valores altos de α hacia la solución SPT. En esos valores altos, el diseño no interesa porque con un poco más de coste se obtienen mejores resultados. Por lo tanto, de elegir un valor intermedio, el más razonable es α=0.3, que produce los mismos resultados que la gran mayoría de valores medianos, con el mínimo coste.

4.2.2.2 CMST y multispeed

Problema número 6

Dado el conjunto de nodos, tarifarios y tráficos del ejercicio anterior, tener en cuenta los siguientes datos:

- *Pesos en cada nodo:*

1	2	3	4	5	6	7	8	9	10
900	20400	1500	52000	500	2200	1130	440	2490	240

- *Capacidades asociadas a las lineTypes dadas:*

LineType 1	9600 bps
LineType 2	56000 bps
LineType 3	128000 bps

Obtener con ellos un diseño CMST y un diseño CMST multispeed, utilizando como nodo central el 6, y cumpliendo las restricciones de capacidad en el primer caso para una W=128000, y las de utilización en el segundo para el 60% máximo. Variar seguidamente los parámetros a fin de comprender el funcionamiento de los algoritmos con respecto a sus restricciones.

Analizar estos diseños.

Los datos que da el problema, en código son:

```
weights=[900;20400;1500;52000;500;2200;1130;440;2490;240;];
capacity=[9600;56000;128000];
utilization=0.6;
W=128000;
central=6;
```

Para obtener el diseño CMST utilizaremos el algoritmo de Esau-Williams, mientras para el *multispeed* CMST utilizaremos MSLA.

La secuencia de comandos que resuelve el problema es la siguiente:

```
linksTable=E_W (tariffTable(:,:,1),weights,W,central);
figure
displayGraph (sitesTable,convert(linksTable,1),0,3,0,1);
title(sprintf('\nEsau-Williams\nCoste del diseño: %.3f',sum(linksTable(:,3))/2));

linksTable2=MSLA(tariffTable,capacity,utilization,weights,central);
figure
displayGraph (sitesTable,convert(linksTable2,1),0,3,0,1);
title(sprintf('\nMSLA\nCoste del diseño: %.3f',sum(linksTable2(:,3))/2));

figure
linksTable3=[linksTable2(:,1) linksTable2(:,2) linksTable2(:,4)];
displayGraph (sitesTable,convert(linksTable3,1),0,3,0,1);
title(sprintf('\nMSLA\ndibujo de los tipos de linea'));
```

Comentarios de uso:

- Para el Esau-Williams, al igual que hacíamos en los MST y SPT, utilizamos solo una de las posibles k de la tariffTable, en este caso la 1. El uso de esta función, salvo por los nuevos parámetros W (capacidad máxima de los enlaces) y *weights* (pesos), es igual a las de todos los problemas MST y SPT.
- Además de representar la salida normal del MSLA, se ha optado aquí por representar una segunda figura en la que se dibuje el tipo de línea que lleva cada enlace. Si recordamos los tipos de datos, la *linksTable multispeed* anexaba una cuarta columna que indicaba la *k* del respectivo tipo de línea utilizado, así pues se ha construido una *linksTable3* variando el orden de las columnas, para poder representar esto.

La salida del programa:

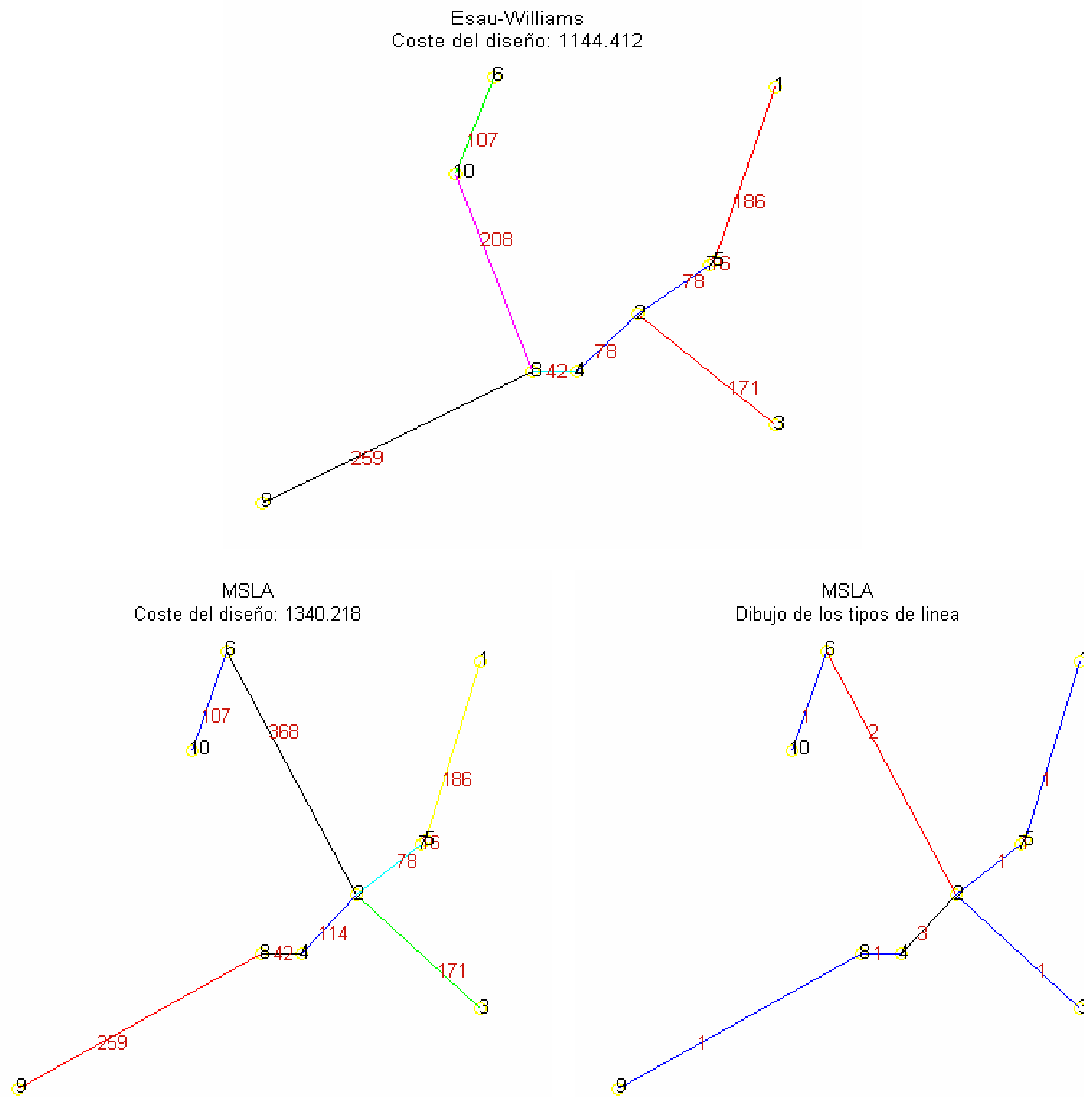


Figura 4-1. Diseños E-W y MSLA

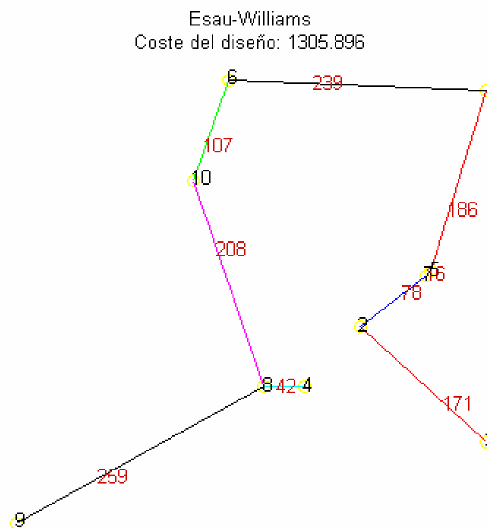
Como se puede observar, el CMST obtenido es el mismo diseño que nos devolvía el algoritmo de Prim y de Kruskal. Esto tiene una explicación simple: la suma de pesos de todos los nodos es menor que la capacidad máxima de la línea, por lo tanto pueden conectarse todos los nodos entre sí siguiendo una filosofía de coste mínimo, y luego conectarse al nodo central (el 6) por a través del nodo que les resulte más barato (el 10), sin incurrir en todo el proceso en un sobrepasamiento de los parámetros restrictivos.

El MSLA sin embargo tiene un comportamiento diferente. El algoritmo va desplazando los pesos de los nodos en el orden que corresponde a la búsqueda del mejor *tradeOff*, de manera que va buscando en cada iteración la conexión más barata, eligiendo el tipo de línea mínimo que requiere esta conexión, y partiendo de un nuevo problema para la siguiente iteración. El

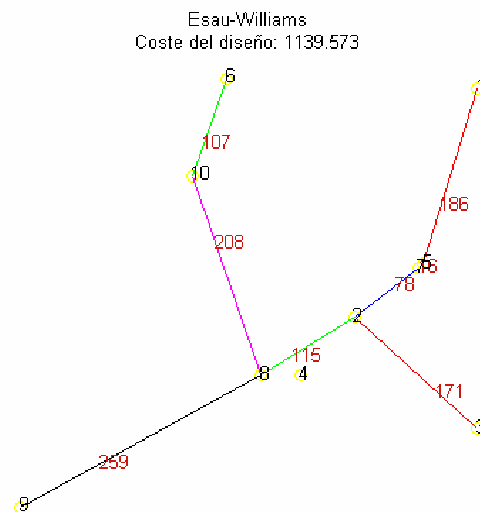
conjunto de salidas posibles al tener en cuenta tres tipos diferentes de líneas nos llevan a diseños bastante diferentes respecto al algoritmo Esau-Williams.

Para comprobar el correcto funcionamiento del Esau-Williams en este diseño, para ver como maneja los datos de entrada, tendríamos que reducir el parámetro W, o bien aumentar los pesos. Aquí vamos a mostrar las dos opciones, para que se vea con más claridad:

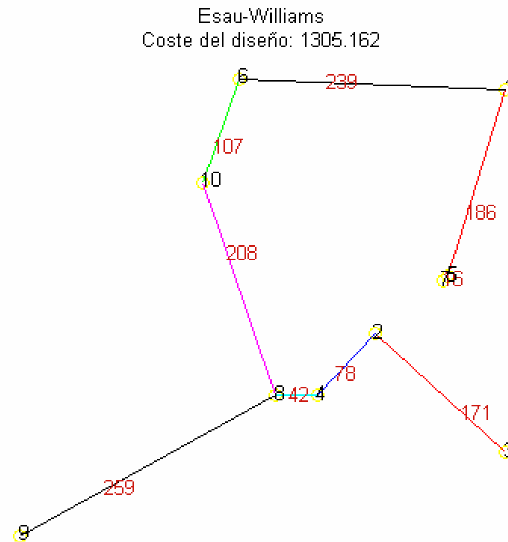
- Esau-Williams con W=64000:



- Esau-Williams con W=50000:



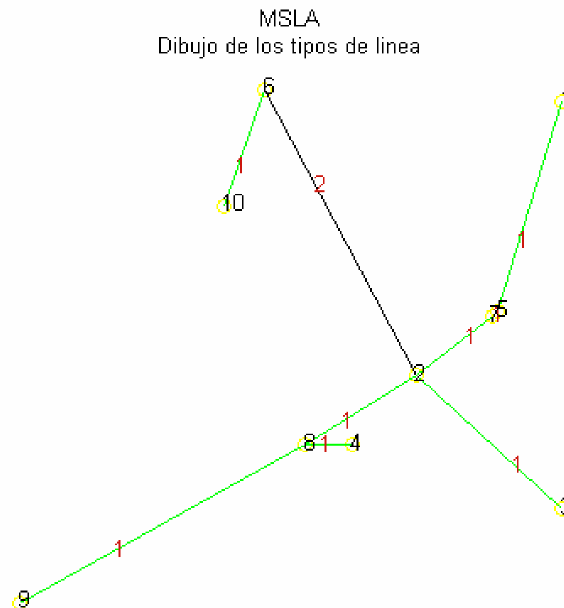
- Esau-Williams con aumentando el peso del nodo 5 a 64000 bps, con el W original:



En el primer caso, la restricción de W (64Kbps) impide que el caudal del nodo 2 (muy grande) se sume con todos los demás nodos, teniendo que "descargar" en el nodo central antes de hacerse más grande. En el segundo caso esta restricción se ha endurecido (50Kbps) hasta sobrepasar el límite que le permite al algoritmo conectar el nodo 4 (de un caudal de 52Kbps por sí solo), así que éste queda fuera del diseño. En el tercero, al haber aumentado mucho el caudal del nodo 5 tenemos una situación parecida a la primera, el tráfico del nodo 5 tiene que correr por una rama distinta a la del nodo 2 y el 4, porque juntos sobrepasarían rápidamente la capacidad W máxima.

Ahora vamos a observar el comportamiento del MSLA con respecto a los cambios en sus parámetros. Partiendo de los parámetros iniciales, veamos qué sucede en los siguientes casos:

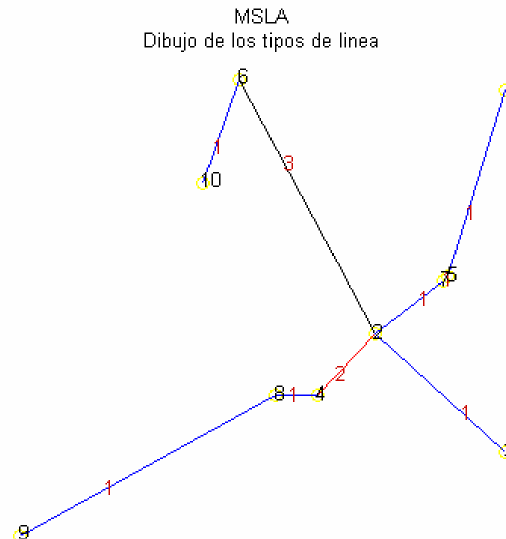
- Reducimos el peso del nodo 4 a 5200 bps:



Aquí se observa como la relajación del caudal en el nodo 4 han provocado dos cambios con respecto al MSLA que mostrábamos al principio: primeramente, el enlace que conecta el

nodo 4 con el resto ya no necesita una línea de tipo 3 (más cara, y con mayor capacidad: 128Kbps), sino que le basta una línea de tipo 1 (9.6 Kbps). En segundo lugar, el nodo 4 ya no está directamente conectado al 2, sino que su pequeño caudal puede sumarse sin prisas al del nodo 8 antes de llegar al nodo 2 y al central.

- Partiendo de este diseño, reducimos la utilización máxima de 0.6 a 0.3:



En este caso volvemos a tener un enlace de k=2 entre el nodo 4 y el 2, y además el enlace entre el 2 y el central (6) ha tenido que aumentar su tipo de línea a k=3. La filosofía general se va deduciendo: cuanto mayor es el caudal de un nodo con respecto a los tipos de línea, dicho nodo intentará aumentar su tipo de línea, o bien añadir su tráfico en último lugar, a fin de que su caudal no se sume con los de los nodos circundantes y se sobrepasen las restricciones.

- Reducimos aún más la utilización hasta un 0.15.

```
??? Error using ==> msla
There isnt a capacity enough for the weight of node 2.
```

A diferencia del Esau-Williams, que si no puede cumplir las restricciones para un nodo deja éste fuera del diseño, el MSLA, de una mayor complejidad y rango de soluciones, se ha programado de forma que indique la carencia. En este caso nos dice que la capacidad del nodo 2 (20400) excede la capacidad máxima utilizable. De hecho, se comprueba que:

$$128000*0.15<20400$$

Para terminar este problema vamos a hacer un examen de características de los diseños. En este caso interesará sobre todo la utilización de cada enlace. Probamos pues con los parámetros originales (correspondientes a la figura 4.1) el siguiente código:

```
routingTables = OSPFrouting (linksTable,0);
[trafficPerLink, hops]=trafficLoad(trafficTable, routingTables);
```

```
[HopsMean, Tmean, U] = calculateParams(trafficTable, trafficPerLink , hops, 128000, 32)


routingTables = OSPFrouting (linksTable2,0);
[trafficPerLink, hops]=trafficLoad(trafficTable, routingTables);
[HopsMean, Tmean, U] = calculateParams(trafficTable, trafficPerLink , hops, 128000, 32)
```

La salida del programa es:

- Para el Esau-Williams

```
HopsMean =
2.1010
Tmean =
2.6264e-004
U=
0         0         0         0         0.021905  0         0         0         0         0
0         0         0.027532  0         0         0         0.059025  0.05883   0         0
0         0.033179  0         0         0         0         0         0         0         0
0         0         0         0         0         0         0         0         0         0
0.030191  0         0         0         0         0         0.031855  0         0         0
0         0         0         0         0         0         0         0         0         0.012566
0         0.040216  0         0         0.048978  0         0         0         0         0
0         0.078185  0         0         0         0         0         0         0.029017  0.043484
0         0         0         0         0         0         0         0.019527  0         0
0         0         0         0         0         0.026864  0         0.068955  0         0
```

- Para el MSLA

```
HopsMean =
2.9809
Tmean =
2.6496e-004
U=
0         0         0         0         0.024143  0         0         0         0         0
0         0         0.03165   0.077187  0         0.053344  0.064249  0         0         0
0         0.038605  0         0         0         0         0         0         0         0
0         0.069057  0         0         0         0         0         0.059039  0         0
0.031981  0         0         0         0         0         0.03625   0         0         0
0         0.077282  0         0         0         0         0         0         0         0.038642
0         0.046871  0         0         0.052974  0         0         0         0         0
0         0         0         0.063495  0         0         0         0         0.030509  0
0         0         0         0         0         0         0         0.02419   0         0
0         0         0         0         0         0.074806  0         0         0         0
```

En el Esau-Williams lo más característico es que la utilización es muy baja, lo que confirma la posibilidad de efectuar el enlace completo de todos los nodos, resultando un diseño igual a los MST de anteriores problemas. El número medio de saltos y el retardo son relativamente bajos por esto mismo.

En el MSLA tampoco se observa una gran utilización, ya que cuando ésta aumenta se cambia de tipo de línea (enlace 4-2). El enlace que tiene mayor utilización es el (6-2), lo cual es lógico porque recoge el caudal de todos los nodos salvo el 10 para llevarlo al central (6). El diseño es peor que el anterior, en lo que se refiere a número de saltos y retardo. Esto es así por el comportamiento en la manera de buscar soluciones que comentamos antes del algoritmo. Se puede concluir que, para problemas de capacidades en los que no se requiera *multispeed*, será mejor utilizar el Esau-Williams al MSLA.

4.2.2.3 MENTOR

Problema número 7: AMENTOR vs AMENTOR *MultiSpeed*

Dado el conjunto de nodos, tarifarios y tráficos del ejercicio anterior, tener en cuenta los siguientes datos:

- *Pesos en cada nodo:*

1	2	3	4	5	6	7	8	9	10
4900	400	3900	2400	500	2400	900	4400	24900	2400

- *Capacidades asociadas a las lineTypes dadas:*

LineType 1	9600 bps
LineType 2	56000 bps
LineType 3	128000 bps

- *Longitud media de paquete: 32 bits*
- *Wparam=0.06*
- *Rparam=0.8*
- *Alpha=0.3*

Se debe obtener con ellos cuatro diseños AMENTOR, 3 simples (uno con cada tipo de línea), y uno multispeed (con los 3 tipos de línea a la vez), teniendo en cuenta en este último una restricción de utilización máxima de los enlaces al 50%. Comparar los resultados de uno u otro algoritmo.

Variar los parámetros para aprender cómo se comportan en el diseño final.

La secuencia de comandos que resuelve el problema es la siguiente:

```
weights=[4900;400;3900;2400;500;2400;900;4400;24900;2400];
capacity=[9600;56000;128000];
utilization=0.5;
L=32; %bits por paquete
Wparam=0.06;
Rparam=0.8;
alfa=0.3;
```

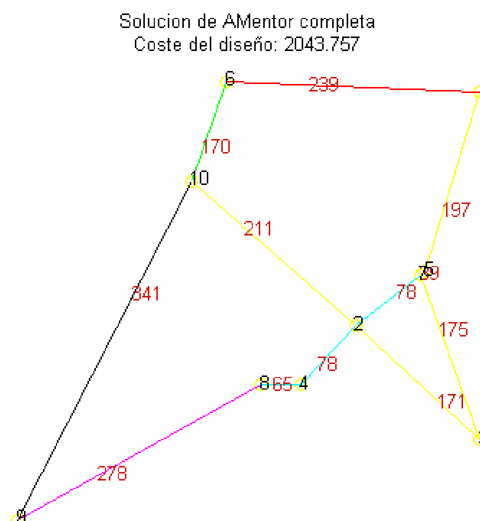
```
for k=1:3
    C=capacity(k);  %la capacidad fija de los AMentor normales será según la k elegida
    [linksSolution,linksBB,nextBB,median]=AMentor(tariffTable(:,:,k),trafficTable,sitesTable,Wparam, Rparam, C, L, alfa, 1);
end
 [linksSolution2,linksBB2,nextBB2,median2]=AMentorMS(tariffTable,trafficTable,sitesTable,Wparam, Rparam, capacity, L, utilization, weights, 1);
% % Mostramos el dibujo de los tipos de linea para el AMentorMS
links=convert(linksSolution2,3);
links=[links(:,1) links(:,2) links(:,4)];
figure;
displayGraph (sitesTable,convert(links,1),0,3,0,1);title(sprintf('\nAMENTOR MultiSpeed\nDibujo de los tipos de linea'));
```

Comentarios:

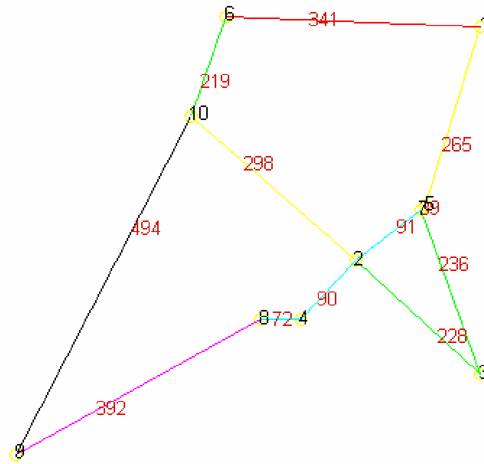
- Los 3 primeros diseños AMENTOR se obtienen mediante un bucle *for* en el que cada vez va cambiando el tipo de línea *k*, variando dos parámetros: tariffTable(:,:,*k*), y la capacidad C que va en función de *k*. Esto se hace así para comparar los diseños con el resultado posterior del *multispeed*.
- Para visualizar mejor los resultados del AMentorMS se ha convertido la solución a fin de poder mostrar una figura de los tipos de línea, tal como se hizo en el problema del MSLA.
- En ambos casos se ha elegido el parámetro 1 para los gráficos, que muestra sólo los más importantes. De estos haremos una selección ahora.

Salida del programa:

- *Para el AMENTOR con k=1*

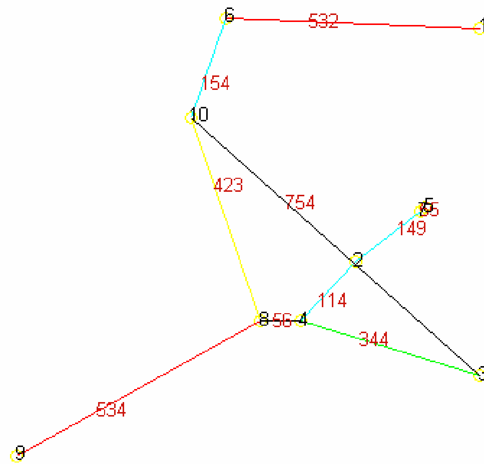


- *Para el AMENTOR con k=2*

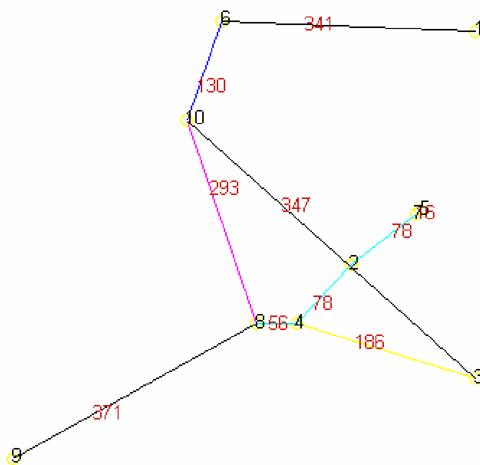


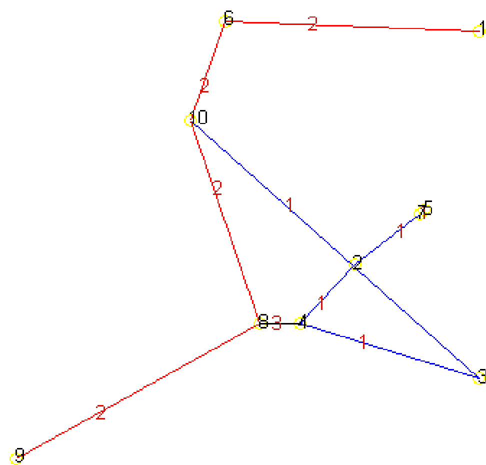
:

- *Para el AMENTOR con k=3*



- *Para el AMENTOR multispeed*





Como primer comentario, hay que tener en cuenta que para cada uno de los diseños simples, el tamaño del *backbone* varía mucho. De hecho, la restricción del Wparam=0.06 provoca que en el primer y segundo caso todos los nodos sean backbone, ya que la capacidad es menor, y el algoritmo de selección *threshold clustering* calcula unos pesos normalizados mayores. Recodemos la ecuación dada en la teoría (sección 2.4.2.1):

$$NW(N_i) = \frac{W(N_i)}{C}$$

La biconexión resulta por tanto más cara, ya que tiene que conseguir un bloque con más nodos. Por el contrario, en el tercer solo unos pocos nodos (3,4,8 y 10) pertenecen al *backbone*. Sin embargo, el uso de líneas de *k*=3, (capacidad 128000) en el tercer diseño, lo encarece respecto a los dos primeros, a pesar de lo anterior.

En el último diseño, como podemos observar el AMENTOR *multispeed* combina distintos tipos de línea, tanto en el *backbone* como en los nodos terminales, para lograr una solución que resulta más barata.

Podemos concluir pues, que el esfuerzo invertido en el AMENTOR *multispeed* ha merecido la pena.

Problema número 8: Parámetros de selección del *backbone* en AMENTOR

Dados los datos del ejemplo anterior, variar los parámetros para comprender se comportamiento e influencia en el diseño final.

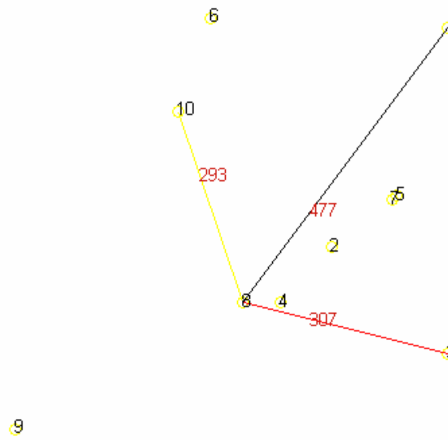
Existen múltiples combinaciones de datos que nos llevan a diferentes diseños. En el problema anterior hemos observado la importancia de la capacidad de los enlaces, y la ventaja de usar el algoritmo *multispeed*, ya que aúna las ventajas del resto de diseños.

El método de trabajo de un diseñador de red con la herramienta MENTOR suele pasar por sacar ristras de datos variando los parámetros iniciales. Los parámetros varían sobre todo el conjunto *backbone*, motor del diseño, ya que determina los costes principales del resto del diseño: el de aumentarlo –AMENTOR- para que sea biconexo (un backbone mayor será más caro de aumentar), y el de los nodos terminales que se le enganchen en la etapa de acceso local.

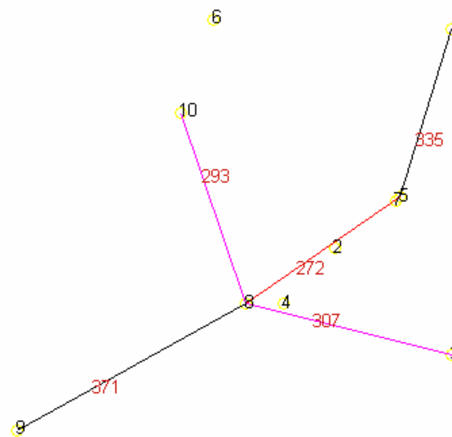
Para ver el comportamiento del resto de los parámetros vamos a presentar algunos ejemplos. En los siguientes casos se muestra el estado del algoritmo cuando se ha creado una topología inicial entre los nodos *backbone*, de forma que veremos directamente la repercusión de los diferentes datos en la selección de éste.

En este conjunto de ejemplos, veremos que variando Wparam y Rparam respecto a una misma capacidad, se puede obtener un subconjunto *backbone* mayor o menor. El parámetro *alpha* controlará la distancia entre diseño MST y SPT, y ya nos es común.

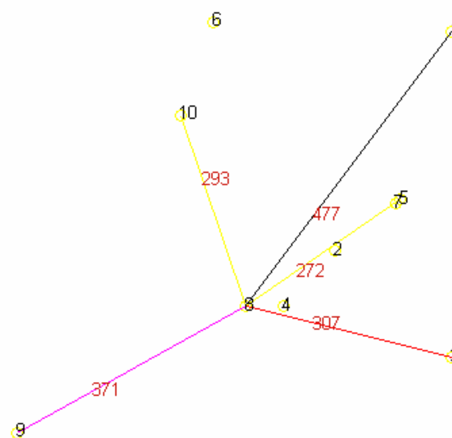
- AMENTOR básico con *k*=2, *alpha*=0.3, Wparam=0.15, y Rparam=0.5:



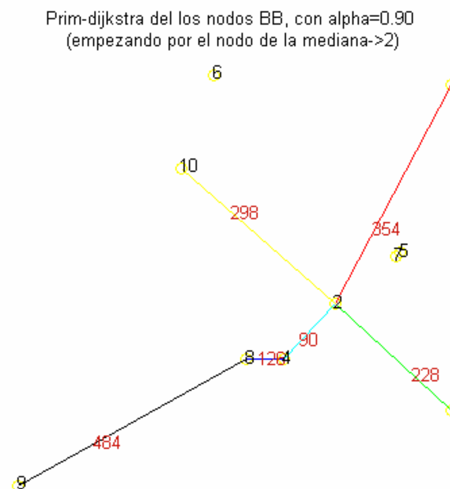
- AMENTOR básico con *k*=2, *alpha*=0.3, Wparam=0.15, y Rparam=0.3:



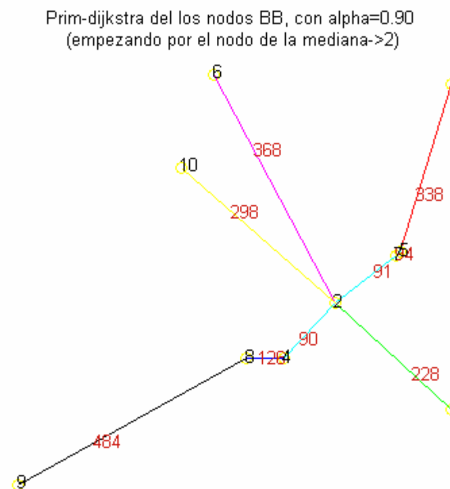
- AMENTOR básico con *k*=2, *alpha*=0.9, Wparam=0.15, y Rparam=0.3:



- AMENTOR básico con *k*=2, *alpha*=0.9, Wparam=0.1, y Rparam=0.3:



- AMENTOR básico con *k*=2, *alpha*=0.9, Wparam=0, y Rparam=0.3:



Si observamos, disminuyendo el Wparam, o aumentando el Rparam se captan más nodos backbone. En el primer caso porque un mayor número de nodos tiene un peso normalizado que supere la barrera de Wparam al estar ésta más baja, y en el segundo, porque una vez escogidos los primeros nodos *backbone*, si el radio de pertenencia de nodos terminales se acorta, habrá que volver a elegir *backbones* entre los nodos que quedan, una y otra vez hasta completarlos todos (segunda etapa del *threshold clustering*). Como cada *backbone*, con radio pequeño, agrupa a escasos nodos terminales, todos acabarán siendo *backbones*.

Si lo que hacemos es por el contrario aumentar Wparam para captar menos nodos *backbone*, hay un punto de inflexión en el cual quedarán muchos nodos sueltos, con lo que pasarán a la segunda etapa del algoritmo. En esta etapa no se elige *backbones* a los de mayor peso normalizado (sacado de la tabla de tráfico), sino que también influye la posición relativa.

Otro parámetro que podemos variar es la *longitud media de paquete*, L. Los resultados estarán directamente relacionados con la variación de capacidad que observábamos en el problema anterior, pues al variar L, varían los pesos calculados a partir de la trafficTable para el

proceso de selección de clustering, de forma inversamente proporcional a la variación de capacidad, ya que:

$$W(N_i) = \left(\sum_j \gamma_{ij} + \gamma_{ji} \right) \cdot L$$

, con lo cual,

$$NW(N_i) = \left(\sum_j \gamma_{ij} + \gamma_{ji} \right) \cdot \frac{L}{C}$$

, donde se ve dicha proporcionalidad inversa. Se deduce entonces que aumentando la longitud media de paquete, aumentará el número de *backbones* iniciales seleccionados.

Capítulo 5
Conclusiones y líneas futuras

El objetivo de este Proyecto Fin de Carrera, ha sido el diseño e implementación de una herramienta de planificación y diseño de redes de área extensa, para aplicaciones didácticas. A lo largo de la realización del mismo, se han desarrollado los conceptos de este campo de forma paulatina, comenzando por el contexto de la planificación WAN, y terminando por exponer problemas prácticos en los cuales la herramienta desarrollada obtiene resultados. El procedimiento a lo largo de toda la exposición del proyecto, así como el diseño de la herramienta ha pretendido tener un trasfondo didáctico. Se pretenece que la herramineta desarrollada, pueda servir de apoyo a asignaturas en el campo de la planificación., o en general por cualquier investigador interesado sobre el tema.

Como lenguaje para el desarrollo de la herramienta, bautizada como *MatPlaNet Tool,* se escogió el entorno MATLAB, ya que se ajustaba a ciertas condiciones: un lenguaje versátil, ampliable y fácilmente editable, con opción a trabajar con los datos que se van obteniendo y visualizarlos en todo momento. La experiencia de utilización de MATLAB ha sido positiva, pues el modo comando da la suficiente flexibilidad para trabajar de forma muy cómoda, y el tratamiento de matrices que incluye es especialmente beneficioso (más teniendo en cuenta que la mayoría de datos que se manejan en redes son matrices) a la hora de hacer una programación sencilla, sin necesitar manejar variables de memoria, punteros u estructuras de datos complejas, como hubiera sido necesario con otro lenguaje. En definitiva, ha permitido manejar con mayor claridad didáctica los conceptos de planificación de redes. La contrapartida de MATLAB es, sin embargo, una cierta lentitud de cálculos con respecto a otros lenguajes, debida al alto nivel con el que se programa. De todas formas, el programa da la posibilidad de compilar sus funciones y "batchs" en .m hacia ejecutables, así como utilizar funciones programadas en C, lenguaje muy común y adecuado para el tema de algorítmica. Por lo tanto, en general, MATLAB ha cumplido sobradamente los objetivos que se esperaban de él.

Otro punto del proyecto ha sido la introducción y la asimilación del mundo de la planificación y del diseño de redes. En el primer capítulo se habló del entorno, el contexto extenso y múltiple que suponen estas redes. En primer lugar hubo que delimitar nuestro rango de trabajo a las redes de tráfico de datos, (más concretamente nos concentramos en las redes de paquetes sobre IP), en las cuales se resolvía que había un equilibrio difícil entre sus parámetros más importantes: utilización de los enlaces, retardo, y coste de llevar a cabo el proyecto. Se deducía de esto el tipo de problema al que nos enfrentábamos, de optimización, en el que dadas ciertas restricciones se intenta obtener el coste más bajo o alguna otra característica de diseño. Nos centramos en los problemas de asignación de topología, capacidades y flujos (*TCFA*), y a su tratamiento mediante el lenguaje matemático de la teoría de grafos. Tras dar un breve repaso a lo que conlleva el trabajo de un planificador y las herramientas con las que cuenta para su tarea, se corroboró la decisión de obtener una herramienta didáctica que aunara los principales algoritmos de planificación.

El siguiente paso fue hacer un estudio de cuáles de los algoritmos y herramientas de diseño y optimización de redes contendría la herramienta. A lo largo del segundo capítulo, se hizo un recorrido por los problemas más comunes a solucionar a la hora de planificar una red. Desde los problemas más básicos, en que el diseño consistía tan sólo en conectar un conjunto de nodos al mínimo coste, se fueron aumentando las exigencias (número de saltos, pesos relativos de los nodos, utilización, asignación de múltiples capacidades y tipos de líneas disponibles, fiabilidad, etc) a la par que aumentaba la complejidad de los algoritmos que trataban dichos

problemas. De esta manera se completó lo que sería el entorno de actuación de la herramienta, mientras quedaban fundamentados los procedimientos y la base teórica de la teoría de redes, pilares sobre los cuales sostener los algoritmos que la herramienta tenía que incluir.

Siguiendo esta teoría, se programó la herramienta, de forma que cada función correspondiera a un algoritmo teórico, y los parámetros de entrada y salida de las diferentes funciones tuvieran tipos de datos en común como para poder combinarlas en cualquier problema dado. Aunque las funciones en su mayoría fueron destinadas a el bloque central de algoritmos de planificación, la parte de optimización de diseños no fue la única, ya que aparejados a ella surgieron los generadores de datos (generación y tratamiento inicial de las matrices de tráfico y costes), las herramientas de visualización y transformación de unidades y tipos, y un importante porcentaje relativo a algoritmos de teoría de grafos, usados a menudo en cuanto las exigencias del diseño iban aumentando. El "producto estrella" de la herramienta, se podría decir que es el AMENTOR (formada por un conjunto de sub-funciones), que da una solución altamente exigente al diseño optimizado de topologías de red. La experiencia de utilización de la herramienta llevó a rediseñar ésta última, obteniendo el AMENTOR *Multispeed*, aportación del autor, que supone una mejora en el conjunto de algoritmos MENTOR por la adición del soporte para diferentes capacidades y tipos de línea.

La descripción del total de las funciones, así como sus parámetros de entrada y salida se adjunta a modo de manual de usuario en el capítulo tercero del proyecto. Además, siguiendo el camino didáctico, para completar la instrucción en el manejo de la misma, y pensando en la pretensión lógica de dar una solución a problemas de redes reales, se ha creado una serie de problemas que de una forma rápida dan un recorrido por la mayoría de funcionalidades que puede desarrollar esta herramienta desde el punto de vista práctico. Esto último conforma el capítulo cuarto del proyecto.

Una idea importante es que MatPlaNet Tool no pretende ser en ningún momento un compendio completo de la teoría de redes para la planificación, sino una herramienta de diseño modular y altamente ampliable, que suponga un inicio firme con el cual adentrarse en este extenso mundo. De hecho, hay varias sugerencias acerca de las líneas futuras posibles con las que ampliar las funcionalidades disponibles:

- Crear el algoritmo de Sharma, para el tratamiento de las líneas cruzadas en la solución al problema CMST.
- Crear los tratamientos de *clustering K-Medias* y *Automatic Clustering* para el conjunto MENTOR.
- En el AMENTOR *Multispeed*, se podría conseguir un ahorro mayor de coste si se tiene en cuenta la etapa de aumento del *backbone* en el dimensionamiento de los enlaces, ya que se eligen tipos de línea en función de los pesos iniciales de cada nodo (antes de la etapa AMENTOR propia), pero los enlaces añadidos para formar una red biconexa, y la bajada de utilización en dicha red que estos enlaces provocan no repercuten en la elección de las líneas. Esta cuestión de dimensionamiento podría estudiarse y ajustar mejor la utilización por medio de cambios de tipo de línea posteriores a la tercera etapa del MENTOR.
- Construir un linealizador de funciones a trozos, cuyos parámetros sean directamente transformables en las variables de entrada de la función del generador de tarifarios lineal a trozos. De esta forma se conseguiría un resultado bastante más realista al analizar datos de un tarifario dado y poder generar nuevas tarifas de acuerdo con él. Para conseguir el linealizador, se debería programar un mecanismo que detecte si el ángulo de la linealización sufre un cambio brusco a partir de ciertos índices de una tabla, y terminar allí el trozo para empezar otro nuevo con nueva pendiente.

- o Elaborar algoritmos que manejen tablas de topologías unidireccionales, es decir, con enlaces *simplex*, lo que conlleva aumentar el concepto de conectividad (conectividad fuerte o débil) y reprogramar el 90% de las funciones a fin de adaptarlas a ello.
- o Modificar la función *displayGraph* para el tratamiento unidireccional de enlaces. Además de utilizar la flecha (función *arrow.m*), se debería añadir algún mecanismo para poder distinguir el color de cada línea unidireccional cuando hay dos que se sobreimprimen en un enlace.
- o Elaborar una nueva versión del MENTOR que aproveche los conceptos de líneas 1-*commodity* y el ruteo basado en *homming* del MENTOR II, y la fiabilidad del bloque *backbone* del AMENTOR.

Éstas serían algunas de las posibilidades, aunque el rango de ampliaciones puede ser todo lo extenso que se quiera. Recordemos que nuestro contexto es el de tráfico de datos de datagramas IP, con ruteado OSPF. Un simple cambio de algoritmo de encaminamiento requeriría, por ejemplo, la reprogramación completa del MENTOR. Esto nos da una idea de la cantidad de adiciones que se pueden hacer en la herramienta, a fin de prepararla para un entorno de planificación más general.

Concluyendo, en este proyecto, tras un inicial viaje instructivo del mundo de la planificación WAN, se ha implementado la herramienta de diseño y planificación de redes MatPlaNet Tool, altamente reutilizable y extensible, mediante un paquete de funciones correspondientes a los principales algoritmos y herramientas de que puede hacer uso un planificador de red, en un lenguaje que pretende ser claro y conciso, para ajustarse fácilmente a entornos didácticos.

Apéndice

A. Algoritmos de exploración de grafos: DFS

Uno de los problemas fundamentales de grafos es encontrar una forma sistemática de recorrer cada enlace y cada vértice. De hecho, en la mayoría de los algoritmos básicos se necesita hacer este recorrido de alguna manera, para distintas aplicaciones, como pueden ser:

- Contar el número de vértices y enlaces de un grafo.
- Obtener o validar los contenidos de cada enlace y/o vértice.
- Identificar los componentes conexos de un grafo.
- Buscar los caminos entre dos vértices, o los ciclos si existen.

Para lograr *eficiencia*, se tiene que asegurar no dejar ningún enlace fuera del recorrido, ni examinar algún enlace repetidamente. La pista es marcar cada vértice cuando lo visitemos y llevar un recuento de los que aún no hemos explorado. El orden en el cual exploramos depende de la estructura de datos usada para guardar los enlaces por los que hemos pasado y sus vecinos aún no explorados. Hay dos posibilidades importantes:

- Cola: Almacenando los vértices en una cola FIFO, recorremos en primer lugar los vértices que supimos inexplorados primero. El recorrido radia lentamente en torno a nuestro punto de partida, definiendo una búsqueda tipo "primeramente a lo ancho", o *Breadth-first Search* (BFS).
- Pila: Almacenando los vértices en una pila LIFO, recorremos los vértices llegando primero hasta el fondo de la primera línea de exploración hasta que haya que volver por los enlaces ya descubiertos para recorrer las otras opciones que se dejaron por el camino. El recorrido es más rápido, se aleja muy velozmente del punto de inicio, definiendo una búsqueda tipo "primero a lo profundo", o *Depth-first Search* (DFS).

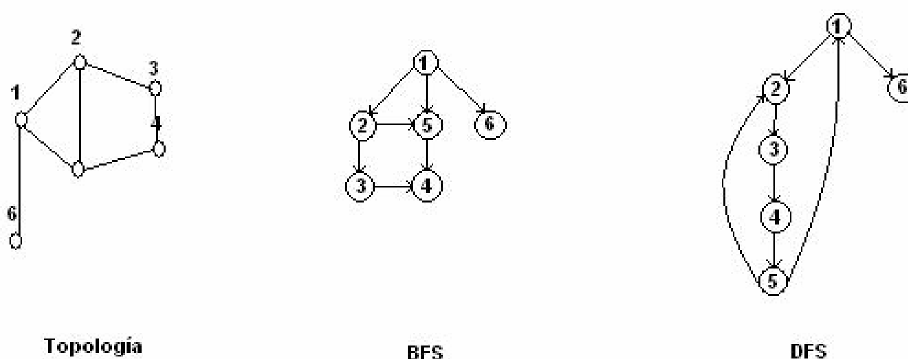


Ilustración 1. Búsquedas BFS y DFS

En este proyecto se ha utilizado como algoritmo de búsqueda el DFS, aprovechando el concepto de recursividad, con lo cual se evita tener que usar explícitamente una pila, de forma que resulta más sencillo. Para más información sobre el BFS puede acudir a [10].

El algoritmo utilizado para la búsqueda DFS utiliza una variable estado(nodo) que indica si está explorado o no (booleano), y un array padre(nodo) que indica el nodo desde el cual se ha explorado (opcional). El esquema general sería el siguiente:

```
Función DFS (G, u)
1. Poner estado(u) = ‘explorado’.
2. Procesar vértice u:
       Para cada v vecino de u:
              Procesar enlace (u,v)
                  Si estado(v) == ‘no explorado’, entonces
                       a. Padre(v)=u
                       b. DFS(G,v)
```

Algoritmo A 1. Búsqueda DFS

Este algoritmo se implementa en el proyecto como función DFSforest.m.

B. Algoritmo de búsqueda de ciclos en un grafo

En muchos de los algoritmos de redes se requiere que el grafo sea acíclico, por lo tanto un algoritmo de detección de ciclos en un grafo se hace indispensable. Hay muchos algoritmos posibles. Aquí se ha implementado de dos formas diferentes:

- Como opción en la función de búsqueda DFS:

  Aprovechando el orden de búsqueda, se puede comprobar que cuando se realiza la exploración, si se encuentra un nodo que ya he marcado antes, y no es el padre del nodo del que provengo, entonces existe un ciclo, ya que desde ese nodo se puede llegar a él mismo siguiendo el camino.

  El problema de esta forma es que requiere que el grafo esté conectado, propiedad que no siempre se da.

- Como función propia:

  Aquí se soluciona lo anterior separando en primer lugar los diferentes componentes que tenga el grafo. Para cada componente se hace la siguiente comprobación:

"Si el número de enlaces del componente es mayor o igual al número de nodos, hay un ciclo".

Vemos que esta conclusión es también sencilla y lógica, ya que si un conjunto de nodos están conectados todos mínimamente, es un árbol, una propiedad de los cuales era que el número de enlaces sería igual al de nodos menos uno (ver sección 1.2). Si hay un enlace más, este forma obligatoriamente dos caminos (el que había y uno nuevo) que van a un mismo nodo.

C. Fiabilidad de una red

La fiabilidad de una red es un importante concepto que tiene relación con el grado de conexión que haya entre sus elementos o nodos, en la introducción a la teoría de grafos en la sección 1.2 se dice que la **fiabilidad** de una red es la probabilidad de que los nodos que están activos estén conectados por enlaces que están en funcionamiento. Pero en términos prácticos, ¿qué quiere decir esto?

Quiere decir que una red n-conectada será más fiable cuanto mayor sea n. La razón es la siguiente: que una red sea n-conectada significa que como mínimo se necesitan romper n enlaces o nodos para conseguir desconectar la red.

Una red conectada es lo mínimo que se necesita para que haya comunicación entre todos los nodos. Una red 2-conectada da opción a que en caso de error, rotura o asalto a la seguridad de la red, la desactivación de un enlace o un nodo y todos los enlaces de alrededor no desconecten el resto de la red. Una red 3-conectada sería doblemente segura, y así sucesivamente.

En este proyecto se ha buscado un grado de conexión de red hasta 2-conectada, y para ello se han dispuesto algoritmos que lo consiguen, pero el primer paso de estos algoritmos, será comprobar el estado de la conexión de la red, por lo tanto se han creado dos funciones de chequeo. El funcionamiento se muestra a continuación:

C.1. Algoritmo CheckConnected

En la mayoría de los algoritmos de redes se requiere que el grafo conexo, o al menos poder separar las partes conexas de un grafo, por lo tanto es necesario un algoritmo de detección de conectividad simple. De los metódicas posibles, aquí se han implementado de dos formas diferentes:

- Aprovechando la función de búsqueda DFS:

  Dicha función me devuelve un vector de padres de la forma parent(u). Si observamos, al recorrer un grafo conectado todos los nodos tendrán un padre menos el primero. Esta propiedad se cumple si y sólo si el grafo es conexo. Por lo tanto, el algoritmo checkConnected es tan simple como comprobar si el grafo cumple esto.

  Esta metódica viene programada en la función checkConnected.m del proyecto.

- Otra opción, una función propia:

  A fin de ganar en velocidad, el autor programó un algoritmo que cambia el concepto sutilmente, de forma que no necesita ejecutar una búsqueda DFS completa, guardando datos de padres. La exploración se realiza de otra forma: se tiene una tabla de vecinos, se empieza con cada vecino de la lista (inicialmente uno cualquiera), y se buscan los vecinos de éste. Los que no pertenezcan a la lista se añaden, y se sigue buscando. En cada iteración se tendrán vecinos nuevos, que habrá que examinar en busca de más

vecinos. De esta manera se logrará tener en la lista todos los nodos accesibles que existan en el grafo. La condición para que sea conexo es que el número total de vecinos diferentes sea el número total de nodos que hay. Si no llega a cumplir esa condición antes de que se acaben los nodos no examinados, el grafo no está conectado, porque no se ha podido acceder a algún nodo. Si por el contrario se cumple, el algoritmo devuelve conectado.

Esta metódica viene programada en la función checkConnect.m del proyecto.

C.2. Algoritmo CheckBiconnected

El método para chequear si es biconexo se basa en el algoritmo anterior, siguiendo la siguiente idea: *"En un grafo biconexo, no existen puntos de articulación"*. Esto se aprovecha de una forma muy sencilla: para cada nodo compruebo si quitándolo (él y todos los enlaces que dan a él) el grafo restante es conectado. Esta es la comprobación de si es o no un *cut-Vertex* (punto de articulación). Si alguno lo es, la función devuelve negativo, y además devuelve los nodos exactos que son *cut-vertex*.

Esta metódica viene programada en la función checkBiconnected.m del proyecto.

D. Búsqueda de subcomponentes

Directamente relacionado con las anteriores cuestiones, tanto para conectar un grafo como para trabajar con sus componentes conexas, se necesita un algoritmo de búsqueda que me indique qué subconjuntos del grafo son componentes *n-connected*. Se han implementado dos métodos para esto:

D.1. Conexos

Cuando se tienen que separar los componentes sencillos (1-connected), el procedimiento es parecido al de *checkconnect*, pero más complejo. La idea en que se basa es que cada nodo pertenecerá a un solo componente, a la totalidad de cuyos nodos se podrá acceder partiendo de uno entre ellos. Por lo tanto, el procedimiento aquí es encontrar, empezando desde cada nodo, todos los nodos a los que se puede llegar. Todos estos nodos formarán un componente. La búsqueda se repite partiendo cada vez de un nodo diferente que no pertenezca ya a un componente, hasta tener todos los nodos asignados a uno.

Esta metódica viene programada en la función findComponents.m del proyecto.

D.2. Biconexos

Cuando se tienen que separar los componentes bloques (2-connected), el procedimiento es ya bastante más complejo. Hay bastante literatura sobre el tema, pero el algoritmo escogido en este proyecto es una versión particularmente optimizada que presenta una profesora de matemáticas de la universidad de Illinois y Texas A&M, la Dr. Carla Purdy [11].

El algoritmo que propone es el siguiente:

```
Let T be empty and Count be 1.  Mark each vertex "unvisited".
Also initialize a STACK of edges, initially empty.
```

```
Choose an initial v in V arbitrarily.
Then call BICONSEARCH(v).


BICONSEARCH(v).
     mark v "visited"
     dfsnum[v] = Count
     ++Count
     LOW[v] = dfsnum[v]
     for each vertex w in adjlist(v) do
          if (v,w) is not on the STACK, push (v,w)
                              ( (v,w) is on STACK if either v < w
                              and w already "visited" or
                              v > w and w = parent[v]  )
          if w is "unvisited" then
                 add (v,w) to T; set parent(w) = v;
                 BICONSEARCH(w)
                 if LOW[w] >= dfsnum[v] then a biconnected
                              component has been found;
                              pop the edges up to and including
                              (v,w); these are the component
                 LOW[v] = min (LOW[v],LOW[w])
          else if w is not parent[v] then
                LOW[v] = min(LOW[v], dfsnum[w])
```

Esta metódica viene programada en la función findBlocks.m del proyecto.

Bibliografía

[1]: Robert S. Cahn, Wide Area Network Design (Morgan Kaufman)

[2]: Ravindra K. Ahuja, Thomas L. Magnanti, James B. Orlin, "Network Flows.Theory, Algorithms and Applications", Prentice Hall 1993.

[3]: Dr. Xiaobo Zhou, http://www.cs.uccs.edu/~zbo/teaching/CS622/CS622_Spring04.html

[4]: A. M. Law and M. G. McComas, "Simulation software for communication networks: the state of the art". IEEE Communication Magazine 1994 32:44-50.
[5]: http://www-cia.mty.itesm.mx/~avazquez/dredes/

[6]: http://www.**opnet**.com/

[7]: http://www.mkp.com/wand.html
[8]:http://www.kt.agh.edu.pl/~mwagrow/projektowanie%20sieci%20telekomunikacyjnych/DeLite/DeLite-README.htm
[9]: http://www.mathworks.com
[10]: http://www2.toki.or.id/book/AlgDesignManual/BOOK/BOOK4/NODE159.HTM:búsqueda BFS y DFS.

[11]: http://www.ececs.uc.edu/~cpurdy/ (lec26.html)

[11]: http://www.cs.dal.ca/~nzeh/Teaching/

[12]: www.cs.dal.ca/~nzeh/Teaching/ Fall%202003/3110/DFSApps.ppt

[13]: Algorithms in C, part 5, 3ª edition, Robert SedgeWick