Ernesto Ignacio Espinosa Chávez

Modelado y Optimización de Redes de Tráfico Vehicular Utilizando Técnicas de Programación Dinámica

2009

Universidad Autónoma de Querétaro
Facultad de Informática

Modelado y Optimización de Redes de Tráfico Vehicular Utilizando Técnicas de Programación Dinámica

Tesis
Que como parte de los requisitos para obtener el grado de

Ingeniero en Computación

Presenta

Ernesto Ignacio Espinosa Chávez

Querétaro, Qro.
28 de septiembre de 2009

Portada Interna de Tesis

Universidad Autónoma de Querétaro
Facultad de Informática
Ingeniería en Computación

Modelado y Optimización de Redes de Tráfico Vehicular
Utilizando Técnicas de Programación Dinámica

TESIS

Que como parte de los requisitos para obtener el grado de

Ingeniero en Computación

Presenta:
Ernesto Ignacio Espinosa Chávez

Dirigido por:
Dr. Jaime Rangel Mondragón

SINODALES

Dr. Jaime Rangel Mondragón Presidente	Firma
Dr. Arturo González Gutiérrez Secretario	Firma
M.C. Fidel González Gutiérrez Vocal	Firma
L.MAC. Elieth Velázquez Chávez Suplente	Firma
M.C. Guillermo Díaz Delgado Suplente	Firma

Nombre y Firma Director de la Facultad	Nombre y Firma Director de Investigación y Posgrado

Centro Universitario
Querétaro, Qro.
28 de Septiembre de 2009
México

RESUMEN

El desarrollo del presente trabajo se concentra en el modelado de una familia de algoritmos de orden de complejidad polinomial basada en la técnica de la Programación Dinámica, los cuales son capaces de resolver problemas de optimización computacionalmente intratables en el contexto de las redes de tráfico vehicular. Para obtener el objetivo planteado, se construyó un marco teórico en el contexto de la técnica de la Programación Dinámica, orientada a un enfoque de modelado basado en la Teoría de Grafos. Utilizando la estructura matemática conocida como dígrafo ponderado, es posible realizar un modelado de una red de tráfico vehicular, utilizando datos reales. Con este modelado, podemos comprobar la eficiencia desde el punto de vista del grado de acceso entre dos puntos cualesquiera de dicha red. Si el cómputo de las distancias Euclidianas entre cualesquier par de nodos del grafo tiende a 0 respecto al cómputo de las rutas óptimas entre todos los nodos se tiene que la red es ineficiente y es necesario realizar un re-diseño de la misma. Por otro lado cuando el cómputo de las rutas óptimas tiene una tendencia a 1 respecto al cómputo de las distancias Euclidianas se tiene que las rutas son eficientes. Por lo tanto, se necesita realizar un recálculo de las distancias tantas veces como el grafo cambie de manera experimental. ¿Cómo hacer para que el recálculo de las rutas sea eficiente? La investigación refiere a la aplicación de un algoritmo modelado bajo la técnica de Programación Dinámica ideado por Robert Floyd y Stephen Warshall conocido como *Algoritmo de Floyd-Warshall,* también conocido en inglés como *All-Pairs-Shortest-Path Algorithm*.

(**Palabras clave:** Programación Dinámica, Teoría de Grafos, Optimización, Algoritmo de Floyd-Warshall)

SUMMARY

The development of the present work concentrates on the modelling of an algorithmic family of polynomial order using a Dynamic Programming perspective, these algorithms are able to solve intractable optimization problems applied to traffic flow networks. A theoretical framework on Dynamic Programming, linked to the modelling approach based on Graph Theory, was built in order to reach the objective considered. The mathematical structure known as weighted digraph enables the possibility to model a traffic flow network dealing with real data. Then, we could analize the whole network efficiency from a point of view based on the access factor between any two vertices of the digraph. If the sum of all Euclidean distances approaches 0 about the sum of all shortest path distances between any two nodes in the digraph, the whole network is said to be inefficient, translated on a network re-design. On the other hand, if the sum of all Euclidean distances approaches 1 about the sum of all shortest path distances between any two nodes in the digraph, the network routes are said to be efficient. We need to re-compute all shortest path distances every time the digraph representing the traffic flow network is experimentally altered. The question here is how to make all shortest path distances computationally efficient. This research hinges to the application of the *All Pairs Shortest Path Algorithm*, also known as *Floyd-Warshall Algorithm*, which was modelled by Robert Floyd and Stephen Warshall using a Dynamic Programming approach.

(**Keywords:** Dynamic Programming, Graph Theory, Optimization, Floyd-Warshall Algorithm)

AGRADECIMIENTOS

Este trabajo no hubiera sido posible sin el apoyo desinteresado brindado por el Dr. Jaime Rangel Mondragón y el Dr. Arturo González Gutiérrez, así como la correcta orientación y asesoría brindada durante el desarrollo de la tesis.

ÍNDICE

LISTA DE FIGURAS

LISTA DE TABLAS Y ECUACIONES

I. INTRODUCCIÓN

La programación dinámica es un enfoque de optimización que transforma un problema complejo en una secuencia de problemas simples; Bradley (1977) señala como su característica esencial la naturaleza multietapa del procedimiento, es decir, tratar secuencialmente uno o más sub-problemas derivados simplificando así el cómputo del resultado final. Comparada con técnicas de optimización como la programación lineal, programación matemática y la programación entera, la programación dinámica provee un entorno general para abordar problemas de diversos tipos.

Con este entorno general, una gran variedad de técnicas de optimización pueden ser utilizadas para resolver problemas particulares o para dar una formulación general a un problema. En este sentido, la creatividad es un aspecto muy importante en el momento en el que se identifica el método adecuado para resolver un problema, aunque también muchas veces se necesita una visión más sofisticada para reformular la solución establecida.

Desde un punto de vista estructural el problema de tráfico vehicular que se trata en la presente investigación consiste en el diseño o re-diseño de redes de tráfico eficientes. Tales redes deben ser eficientes desde el punto de vista del grado de acceso desde un punto a otro punto en la red. Tal grado de acceso se encuentra en el rango abierto (0,1), donde aquellos grafos donde la sumatoria de las distancias totales de las rutas óptimas o rutas más cortas rebasan por mucho a la sumatoria de las distancias Euclidianas entre todos los pares de vértices en el grafo, resultando en un grado de acceso cercano a 0. Por otro lado, el grado de acceso será cercano a 1 cuando la sumatoria de las distancias totales de las rutas óptimas es muy similar a la suma de las distancias totales Euclideanas entre cualesquier par de puntos dentro de la red. Matemáticamente, el grado de acceso está dado por la ecuación 1:

$$A(i,j) = \frac{\sum_{\forall i,j\, i \neq j} d(i,j)}{\sum_{\forall i,j\, i \neq j} c(i,j)} \qquad (1)$$

donde c(i, j) representa la ruta óptima entre los puntos i y j, mientras que d(i, j) refiere a la distancia Euclideana entre los puntos i y j.

El cómputo de la matriz de distancias entre cualesquier par de nodos es necesaria para el proceso de búsqueda de una red de tráfico vehicular casi-óptima, cuando bajo ciertos criterios la red es modificada a través de un re-direccionamiento de las aristas. Es aquí donde la importancia del problema recae. Contar con un procedimiento eficiente como el que la Programación Dinámica ofrece repercute sustancialmente en el proceso de diseño o re-diseño de redes de tráfico vehicular.

El problema de las rutas más cortas entre todos los pares de nodos entonces consiste formalmente en encontrar una ruta óptima entre cada par de vértices en un grafo dirigido o digrafo G = (V, E), siendo V el conjunto de vértices y E el conjunto de aristas formado por el conjunto pares ordenados de vértices. Esto es, para cada par de vértices (i, j) (no necesariamente (i, j) $\in$ E), queremos encontrar un camino óptimo de i a j así como también uno de j a i. La solución que se plantea en este trabajo de investigación basada en la Programación Dinámica refiere al algoritmo de Floyd-Warshall (Weiss, 2007), el cual corre en tiempo $O(|V|^3)$.

Para la implementación de los algoritmos modelados, se hace uso de dos paradigmas de programación diferentes: El lenguaje de alto nivel *Mathematica*, que provee un enfoque de programación funcional, es decir, la utilización de funciones matemáticas, soportando también los paradigmas de la programación procedural y orientada a objetos; y el entorno de desarrollo *Java*, construida en el lenguaje orientado a objetos del mismo nombre,

ofreciendo las ventajas multi-plataforma y de escalabilidad e integración entre librerías. Así mismo, se presenta un componente de interconexión entre ambos entornos, el *J/Link*, permitiendo una comunicación basada en el envío de paquetes síncronos.

En el caso de *Mathematica*, es esencial en el modelado e implementación de los algoritmos la librería *Combinatorica*, que extiende a *Mathematica* con alrededor de 450 funciones de Combinatoria y Teoría de Grafos. Para esta investigación, se hace uso de las herramientas que nos permiten no sólo el modelado de grafos y sus estructuras derivadas, sino la graficación y la utilización de algoritmos sobre los mismos. En los ejemplos mostrados a lo largo del presente trabajo, se agregará el código *Mathematica* necesario para obtener las salidas correspondientes, con la pre-condición de haber cargado el paquete *Combinatorica* mediante la instrucción

```
Needs["Combinatorica`"]
```

Los ejemplos en *Mathematica* fueron desarrollados utilizando la versión 7.0, mientras que en el caso de *Java* las implementaciones se hicieron sobre la versión 6 del lenguaje.

II. LA TEORÍA DE GRAFOS

La teoría de Grafos es usada tanto en Matemáticas como en las Ciencias de la Computación para modelar relaciones entre objetos. Numerosas situaciones pueden ser descritas en términos de un diagrama con puntos y líneas uniendo ciertos pares de ellos. Escenarios cuyas conexiones son físicas, como circuitos electrónicos, vías de tráfico o estructuras moleculares, o escenarios con relaciones no tangibles, como lo son un organigrama, las relaciones interpersonales o el flujo de información en un sistema de cómputo, entre otras, pueden ser abstraídas y representadas matemáticamente en términos de la Teoría de Grafos (Gross y Yellen, 2003).

Un grafo G se define como el par G = (V, E), que consiste en el conjunto V $\neq \emptyset$ y el conjunto E compuesto por subconjuntos de V de dos elementos. Los elementos de V son llamados vértices o nodos, mientras que cada elemento e = {a, b} de E es llamado arista, con vértices-finales a y b. Se dice que los vértices a y b son incidentes con e y que a y b son adyacentes uno con el otro, escribiéndose e = ab (Jungnickel, 2007).

Otros autores agregan a la definición de grafo una función de incidencia ψG asociada a cada uno de los elementos de E, de tal forma que, teniendo el conjunto de aristas E = (e1, e2, e3, e4, ... , en), al aplicar la función ψG(ei), con ei $\in$ E, obtendríamos el par de vértices u, v $\in$ V, elementos adyacentes entre sí e incidentes con e (Bondy y Murty, 2008). Esta definición va más acorde a la estructura matemática derivada del grafo conocida como Multigrafo (Jungnickel, 2007), que será analizada más adelante.

Historia

La Teoría de Grafos tuvo su inicio cuando el matemático suizo Leonhard Euler (1707-1783) realizó en 1736 una publicación en la cual daba solución al problema de los 7 Puentes de Königsberg (Jungnickel, 2007). La ciudad de Königsberg, Prusia, estaba situada a ambos lados del río Pregel, y también incluía dos islas que eran conectadas a la ciudad por medio de puentes. La Figura 2.1 muestra el mapa de la ciudad de Königsberg, con los 7

puentes y el río Pregel a través de ella.

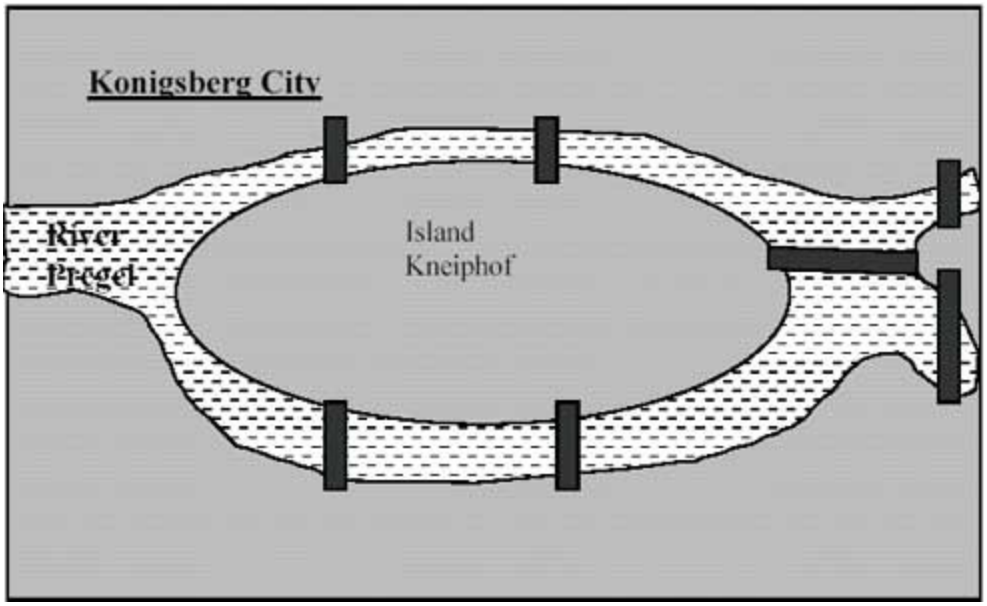


Figura 2.1 Mapa de la ciudad de Königsberg.

El problema consistía en formar un recorrido circular usando cada uno de los 7 puentes una sóla vez. Euler planteó el problema de tal manera que la forma de las islas y los puentes no tenía importancia: el elemento clave en el diagrama eran la conectividad de los puntos. Euler representó las dos islas y la parte norte y sur del río por medio de puntos (vértices), mientras que los puentes fueron representados por líneas uniendo a estos puntos. En la Figura 2.2 se muestra la representación final de Euler del mapa de Königsberg.

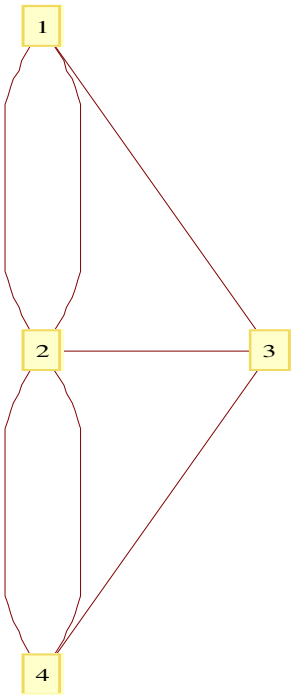


Figura 2.2 Grafo representativo de la ciudad de Königsberg.

```
KoenigsbergBridgesBis = {1 -> 3, 1 -> 2, 1 -> 2, 2 -> 3, 2 -> 4, 2 -> 4, 3 -> 4};
GraphPlot[KoenigsbergBridgesBis, VertexLabeling -> True, MultiedgeStyle -> 1/6,
 VertexCoordinateRules -> {1 -> {0, 1}, 2 -> {0, 0}, 3 -> {.5, 0}, 4 -> {0, -1}}]
```

En el grafo mostrado en la Figura 2.2, el vértice 2 representa la isla central, el vértice 4 la parte sur, el vértice 1 la parte norte y el vértice 3 la isla situada a la derecha. Además, existen 7 aristas representando a los 7 puentes del mapa.

Se inicia el trazo de un recorrido circular a través de los puentes de Königsberg tomando un vértice al azar, por ejemplo, el vértice 2, que tiene 5 aristas incidentes. Puesto que se busca un recorrido circular, habría que volver en algún momento al vértice 2, por lo que nos obliga a considerar las aristas en pares: una arista para salir del vértice y otra para regresar a él. Dado que el número de aristas incidentes en dicho vértice es impar, esto nos resulta imposible. Lo mismo pasa con cualquiera de los otros 3 vértices: 1, 3 y 4 tienen 3

aristas incidentes. Tomando cualquier vértice como inicio obtendríamos el mismo resultado que en el vértice 2.

Euler observó este fenómeno, dándose cuenta que siempre que se entra a un nodo o vértice por medio de un puente, al salir hacia otro vértice se usa otro puente, por lo que cada visita a un vértice implica que el número veces que se entra a un vértice sea el mismo número de veces que se sale. Esto obliga a que, para obtener un recorrido a través de un grafo (ya ni siquiera circular) el número de aristas incidentes en cada vértice sea par. En el caso del mapa de Königsberg, todos los vértices tienen un número non de aristas incidentes, por lo que es imposible obtener un recorrido utilizando cada puente una sóla vez.

Desde sus inicios, la Teoría de Grafos está claramente ligada a las redes de tráfico vehicular. Resulta evidente la conexión entre ambas: calles, avenidas, etc. representadas por aristas, mientras que sus interconexiones representadas por vértices. Euler en su solución al problema de los 7 Puentes de Königsberg realizó un modelado similar: las vías de transporte (puentes) fueron representadas por las aristas mientras que los nodos finales y la isla central que funje como intersección fueron representados por los vértices.

En los últimos tiempos, han surgido problemáticas debidas al rápido crecimiento de las ciudades y sus consecuencias sobre las redes de tráfico vehicular. Se dice que el incremento mundial del tráfico vehicular es de alrededor de 15% (Foulds, 1995), lo cuál obliga a los gobiernos a realizar un análisis sobre un posible re-diseño a las redes de vehículos. El diseño debe estar orientado a aspectos naturales de las ciudades y sus redes de transporte: la dirección del crecimiento de la ciudad y la manera de funcionar de la misma (ciudades industriales, comerciales, históricas, etc), la ubicación de centros de trabajo, centros comerciales, mercados, oficinas gubernamentales, hospitales, entre otros aspectos, brindan datos clave al realizar el modelado del flujo vehicular (Foulds, 1995; Dym, 2004).

Conceptos y definiciones

Dentro del contexto del modelado de redes de tráfico vehicular, existen diversas

estructuras matemáticas que nos sirven como herramientas para modelar más a fondo detalles de la red, como son la interconexión, la dirección, el tiempo de recorrido, etc.

Partiendo de que un grafo G = (V, E) consiste en el conjunto V $\neq \emptyset$ y el conjunto E compuesto por subconjuntos de dos elementos de V, se encuentra que un vértice v tiene un grado x, denotado por el número de aristas a las que es incidente. Como vimos en el modelado de Euler de los puentes de la ciudad de Königsberg, el grado de un vértice necesario para completar el recorrido circular utilizando los 7 puentes 1 sóla vez debe de ser par.

Cuando en un grafo G su conjunto de aristas E esta compuesto por todos los subconjuntos de V de 2 elementos, es decir, todos los vértices se encuentran conectados entre sí, se dice que el grafo es un grafo completo K_n, siendo n la cardinalidad de V. Un ejemplo de un grafo completo es el mostrado en la Figura 2.3, el cuál es un grafo completo K_5.

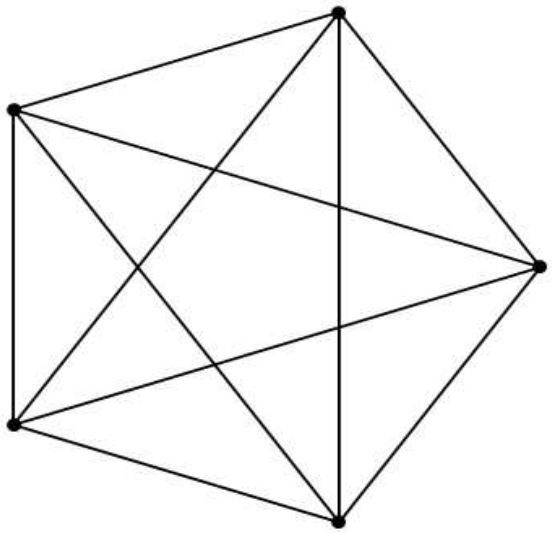


Figura 2.3 Grafo completo de 5 vértices. Su número de aristas es igual a 10.

```
G321 = CompleteGraph[5];
ShowGraph[G321]
```

Dado que para un grafo completo K_n todos sus vértices son adyacentes entre sí, resulta más completo de analizar por las combinaciones resultantes de todas sus aristas. Para cualquier grafo completo, su número de aristas será de $n \cdot (n-1)/2$, siendo n = |V|.

Comprobación: En un grafo completo K_n = (V, E), todos los vértices tienen grado n-1, ya que cada vértice se conecta con todos menos con el mismo. Si se suman todos los grados de los vértices estaríamos contando las incidencias totales de cada uno sobre sus aristas, lo que nos daría como resultado n (n-1). Al hacer esto se estaría contando el número de aristas dos veces (una arista representa la interconexión de 2 vértices, y se están contando los vértices por separado), por lo que al dividir este resultado entre dos se obtiene el número total de aristas.

Sería hasta cierto punto utópico que en una red de tráfico vehicular existiera un canal directo para cualquier par de vértices. Incluso las redes de tráfico vehicular no existirían: algún medio como la teletransportación permitiría ir a algún punto de una ciudad en instantes. En la vida real, esto no es así: para ir de un lugar a otro, generalmente hay que utilizar varios puntos intermedios.

Entonces, ¿para que sirve un grafo completo al analizar una red de tráfico vehicular? Una manera de analizar la eficiencia de una red de tráfico vehicular es desde el punto de vista del grado de acceso entre dos vértices cualesquiera de dicha red. Si se divide la sumatoria de las distancias en línea recta o Euclidianas de cualquier par de puntos entre la sumatoria de las rutas más cortas entre todos los vértices, se estará obteniendo el grado de acceso de la red. Cuánto mayor sea la sumatoria de las rutas más cortas respecto a la sumatoria de la distancias Euclidianas, el grado de acceso tenderá a 0, lo que indicaría que la red es ineficiente. En cambio, mientras más se asemeje al caso ideal (la sumatoria de las distancias Euclidianas), el grado de acceso tenderá a 1 y tendremos una red de tráfico vehicular eficiente. Es trivial observar que la sumatoria de las distancias Euclidianas entre cualesquier par de puntos de la red es la sumatoria de la ponderación de las aristas (i, j) del grafo completo K_n de la red de tráfico vehicular, tomando la ponderación como el valor de la distancia entre los vértices i y j en cuestión.

¿Es posible tener dos intersecciones unidas por dos aristas diferentes? De acuerdo a la definición formal de grafos mencionada en la introducción del capítulo no es así, pues en ella se menciona a las aristas como un conjunto. Bajo la definición de Georg Cantor, el término conjunto se define como la colección S de objetos definidos y distintos entre si, llamados los elementos de S (Dauben, 1979). Fraleigh (1994), por su parte, menciona a los conjuntos como un término primitivo en Matemáticas, es decir, es un concepto que a pesar de tener diversas definiciones (como la dada por Cantor), es más una noción que un concepto. Fraleigh también menciona que es imposible definir todos los conceptos matemáticos, puesto que la definición va acarreando ambiguedad limitada por nuestro vocabulario. Si se define un conjunto como una colección, inmediatamente nos preguntaríamos ¿qué es una colección?, y así sucesivamente. A pesar de ello, Fraleigh menciona verdades acerca de la noción de los conjuntos, entre ellas la característica de que sus elementos son distintos entre sí.

Para considerar un grafo a la solución del problema de los 7 Puentes de Königsberg dada por Euler, se generaliza la definición de Grafo bajo la estructura matemática conocida como Multigrafo. Para un Multigrafo G = (V, E), se redefinen las aristas como la familia (en lugar de un conjunto) E de subconjuntos de V de 2 elementos, agregando una función de incidencia para distinguir dos subconjuntos e_i y $e_j \in$ E cuyos elementos a y b $\in$ V son iguales. Por lo tanto, el multigrafo finalmente se define como una terna (V, E, ψ), donde ψ es la función o mapeo de incidencia, de tal manera que al tomar cualquier elemento e $\in$ E y aplicándole dicha función, ψ(e) daría como resultado el subconjunto {a, b}. Aquellas aristas e y e' cuyo resultado de la función de incidencia es el mismo, es decir, ψ(e) == ψ(e'), se dice que son paralelas entre sí. Se concluye entonces que el diagrama de Königsberg mostrado por Euler en 1736 es un Multigrafo (Jungnickel, 2007).

Es posible ver un Multigrafo en un modelado de una red de tráfico vehicular. Un ejemplo de ello son las carreteras libres y cuotas dentro del sistema de carreteras mexicano. El Gobierno de México está comprometido constitucionalmente a brindar libre tránsito a los ciudadanos a través de todo el país. Cuando el Gobierno decide construir una carretera con cobro de peaje, generalmente (porque hay contadas excepciones donde se viola esta regla)

se construye paralela a una carretera sin costo, denominada carretera libre. Bajo este esquema, considerando el mapa de carreteras de México como un grafo G = (V, E), teniendo a V como el conjunto de ciudades de México y a E como el conjunto de interconexiones directas entre las mismas, tendríamos entonces varios pares de aristas paralelas e_i y $e_j \in E$, por lo que se puede considerar al grafo G que representa el sistema de carreteras mexicano como un Multigrafo.

Cabe señalar que varios autores señalan esta definición de Multigrafo como la definición misma de Grafo, tomando la definición aquí presentada de un Grafo como un "Grafo Simple". En el presente trabajo se conservará el término mencionado como Grafo para los grafos en general y el Multigrafo para referirse a grafos con aristas repetidas.

Subgrafos

Otro concepto de la Teoría de Grafos es el subgrafo. Se tiene un grafo G = (V,E) y un conjunto $V' \subset V$. Entonces es posible tener un grafo G | V' = (V', E | V'), teniendo a E | V' como el conjunto de aristas e = {a, b} tales que tanto a como b pertenecen a V'. A este grafo se le llama subgrafo inducido, y se dice que es inducido por V'. En otras palabras, un grafo es un grafo inducido de otro grafo G = (V, E) cuando a partir de un subconjunto V' de V se toma como su conjunto de aristas todos los pares de aristas cuyos vértices incidentes estén en V'.

En la Figura 2.4, se muestra un grafo con 11 vértices y 22 aristas. Si se omiten en dicho grafo los vértices 8 y 10, reduciendo el conjunto de vértices al conjunto V' = {1, 2, 3, 4, 5, 6, 7, 9, 11}, el resultado será el grafo inducido por V' mostrado en la Figura 2.5.

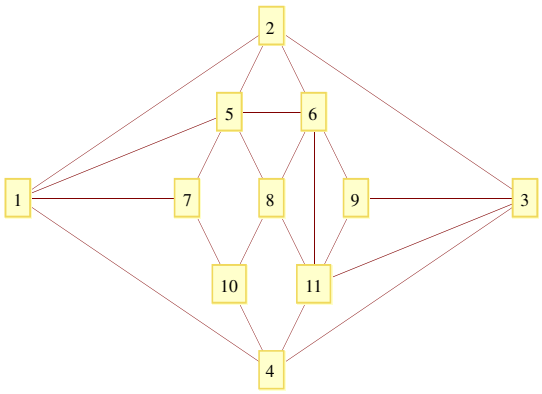


Figura 2.4 Grafo G de 11 vértices y 22 aristas.

```
E322 = {{1, 2}, {1, 5}, {1, 7}, {1, 4}, {2, 3}, {2, 5}, {2, 6}, {3, 4}, {3, 9},
     {3, 11}, {4, 10}, {4, 11}, {5, 6}, {5, 7}, {5, 8}, {6, 8}, {6, 9}, {6, 11},
     {7, 10}, {8, 10}, {8, 11}, {9, 11}};
G3222 = AddEdges[EmptyGraph[11], E322];
VU3222 = {1 -> {-3, 0}, 2 -> {0, 2}, 3 -> {3, 0}, 4 -> {0, -2},
     5 -> {-0.5, 1}, 6 -> {0.5, 1}, 7 -> {-1, 0}, 8 -> {0, 0},
     9 -> {1, 0}, 10 -> {-0.5, -1}, 11 -> {0.5, -1}};
GraphPlot[G3222, VertexLabeling -> True, VertexCoordinateRules -> VU3222]
```

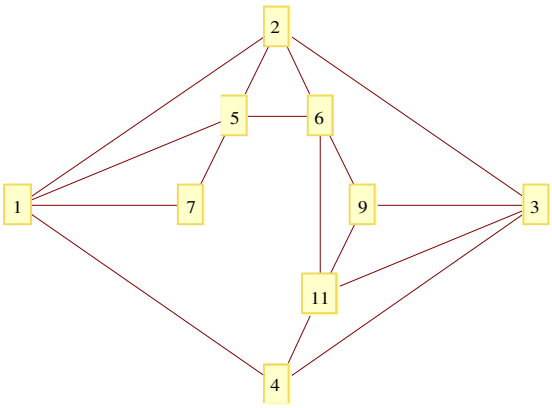


Figura 2.5 Grafo inducido de G.

```
E322 = {{1, 2}, {1, 5}, {1, 7}, {1, 4}, {2, 3}, {2, 5}, {2, 6}, {3, 4}, {3, 9},
     {3, 11}, {4, 10}, {4, 11}, {5, 6}, {5, 7}, {5, 8}, {6, 8}, {6, 9}, {6, 11},
     {7, 10}, {8, 10}, {8, 11}, {9, 11}};
VU322 = {{-3, 0}, {0, 2}, {3, 0}, {0, -2}, {-0.5, 1}, {0.5, 1}, {-1, 0}, {0, 0},
     {1, 0}, {-0.5, -1}, {0.5, -1}};
G322 = SetVertexLabels[ChangeVertices[AddEdges[EmptyGraph[11], E322], VU322],
     {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11}];
V323 = {1, 2, 3, 4, 5, 6, 7, 9, 11};
G323 = InduceSubgraph[G322, V323];
GraphPlot[G323, VertexLabeling -> True, VertexCoordinateRules ->
     {1 -> {-3, 0}, 2 -> {0, 2}, 3 -> {3, 0}, 4 -> {0, -2}, 5 -> {-0.5, 1},
      6 -> {0.5, 1}, 7 -> {-1, 0}, 9 -> {1, 0}, 11 -> {0.5, -1}}]
```

Ahora, no necesariamente vamos a utilizar todas las aristas del grafo original G = (V, E) cuyos vértices incidentes se encuentren ambos en V'. Entonces decimos que un subgrafo G' = (V', E') es aquel donde V' ⊂ V y E' ⊂ E | V'. En la Figura 2.6, se omiten no sólo los vértices 8 y 10, sino también diversas aristas del grafo original, reduciendo la cardinalidad de E a 9. Si el subgrafo en cuestión tiene que V' == V, entonces se le llama subgrafo de expansión (en inglés, spanning subgraph). Esto se muestra en la Figura 2.7.

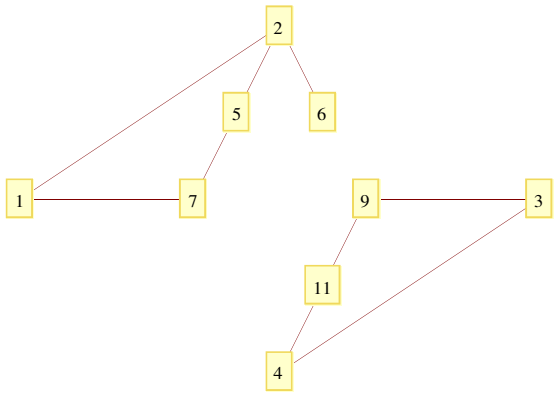


Figura 2.6 Subgrafo de G.

```
(* La definición de G323 es tomada de la Figura 2.5 *)
G324 = DeleteEdges[G323, {{6, 8}, {6, 9}, {1, 4}, {3, 9}, {1, 5}, {5, 6}, {2, 3}}];
GraphPlot[G324, VertexLabeling -> True,
     VertexCoordinateRules -> {1 -> {-3, 0}, 2 -> {0, 2}, 3 -> {3, 0},
           4 -> {0, -2}, 5 -> {-0.5, 1}, 6 -> {0.5, 1}, 7 -> {-1, 0},
           9 -> {1, 0}, 11 -> {0.5, -1}}]
```

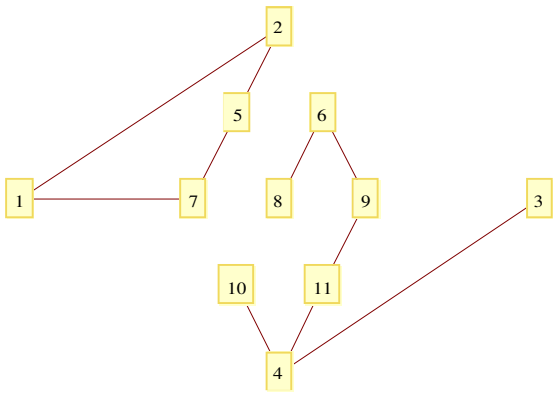


Figura 2.7 Subgrafo de expansión (spanning subgraph) de G.

```
(* La definición de G323 es tomada de la Figura 2.5 *)
G325 = DeleteEdges[G322, {{1, 5}, {5, 6}, {6, 11}, {2, 3}, {2, 6}, {3, 9},
     {7, 10}, {8, 10}, {8, 11}, {3, 11}, {5, 8}, {1, 4}}];
GraphPlot[G325, VertexLabeling -> True,
     VertexCoordinateRules -> {1 -> {-3, 0}, 2 -> {0, 2}, 3 -> {3, 0},
           4 -> {0, -2}, 5 -> {-0.5, 1}, 6 -> {0.5, 1}, 7 -> {-1, 0},
           8 -> {0, 0}, 9 -> {1, 0}, 10 -> {-0.5, -1}, 11 -> {0.5, -1}}]
```

¿En qué nos pueden servir estas estructuras para el modelado de redes de tráfico vehicular? Bajo mi punto de vista, entre las aplicaciones se pueden encontrar las siguientes. Si se tiene un grafo Q = (V, E) que representa a la ciudad de Querétaro en su totalidad, y se quiere analizar sólo cierto sector de la misma, habría que obtener el conjunto V' $\subset$ V que represente a las intersecciones de dicho sector a analizar. El grafo resultante sería un grafo que incluyera todas las calles cuyas intersecciones inicial y final estuvieran incluidas en V', es decir, el subgrafo inducido Q' = (V', E | V') para dicho sector de la ciudad. Si nosotros quisieramos analizar la ciudad desde el punto de vista físico, como pudiera ser el tráfico sobre todas las calles empedradas o sobre las avenidas de alta velocidad, se tendría que enumerar todas las calles con estas características dentro de un conjunto E' $\subset$ E. Lo que finalmente obtendríamos sería un grafo Q' = (V', E') donde V' $\subset$ V. Si mediante un algoritmo se calculan las rutas más cortas desde todos los vértices hacia un punto en particular, se tendría un sólo camino desde cada nodo hasta el nodo final, por lo que el conjunto de aristas E' dentro del grafo resultante sería un subconjunto de E, más no así con el conjunto de

vértices, que sería igual al conjunto original V. Es decir, el grafo resultante del cómputo del algoritmo sería un subgrafo de expansión de Q.

Conectividad, Ponderación y Digrafos

Si se tratara de obtener la ruta más corta entre cualquier par de vértices de un grafo G = (V, E) , o la ruta más corta desde cualquier vértice hasta un vértice específico, sería necesario que en dicho grafo G existiera un camino desde el vértice v_i hasta un vértice final v_n, con $v_i, v_n \in V$ y $v_i \neq v_n$.

Se podría entonces definir un camino (en inglés, walk) como la secuencia de aristas (e_1, e_2, e_3, e_4, ... , e_n) que forman una secuencia de vértices (v_0, v_1, v_2, v_3, v_4, ... v_n), de tal manera que $e_i = v_{i-1}v_i$ para toda i = 1, ..., n. Es decir, que para todos los aristas de la secuencia indicada, su vértice destino es igual al vértice origen del arista siguiente. Si tenemos que el vértice 0 de la secuencia es igual que el vértice n, es decir, que comienza y término en el mismo punto, se dice que el camino es cerrado (closed walk).

Si las aristas de la secuencia son diferentes entre si, al camino se le llama sendero (en inglés, trail). De la misma manera, si el sendero inicia y termina en el mismo vértice, es decir, el vértice 0 es igual al vértice n, se dice que el sendero es cerrado (closed trail). Si los vértices de la secuencia son diferentes entre sí, el sendero es denominado ruta (en inglés, path). Un sendero cerrado, de longitud mayor o igual a 3, en el cuál los vértices son diferentes entre sí, exceptuando que el vértice inicial es igual al vértice final, el sendero es denominado ciclo. La longitud de la secuencia está dada por el número de aristas utilizadas. Volviendo a la conectividad, un grafo G es conexo, si existe para cualquier par de vértices a y b un camino que empiece en el vértice a y término en el vértice b (Jungnickel, 2007).

Aplicando estas propiedades de los grafos a una red de tráfico en el escenario de búsqueda de ruta más corta, se tiene que para poder evaluar el costo del transporte de un vértice a otro dentro de la red, es necesario asignar un valor específico a cada arista, que represente dicho costo. Esta estructura matemática es conocida como Grafo Ponderado (en

inglés, weighted graph). Foulds (1995) define la ponderación de las aristas como una función w tal que dado un grafo G = (V, E) para cada valor de e $\in$ E existe un valor w $\in \mathbb{R}$. Es decir, asigna un número real a cada arista del grafo. Más adelante se mostrará de que manera y bajo que tipo de variables se puede ponderar un grafo.

¿Que pasaría si en una red de tráfico existieran vértices no conectados entre sí? Sería imposible trazar una camino entre ellos, por lo que la distancia o costo de la ruta se dice que es igual a ∞. En cambio si existe un camino entre ambos vértices, el costo del mismo será igual a la sumatoria de la ponderación de cada una de sus aristas.

En la búsqueda de caminos más cortos, el camino resultante necesariamente tiene que ser una ruta (path). Resultaría redundante pasar por el mismo nodo dos veces: esto nos indicaría que habríamos hecho un ciclo innecesario dentro de la ruta, por lo que, podríamos generalizar, que en la búsqueda de caminos óptimos vamos a obtener como resultado una ruta.

Dentro del modelado de las rutas y caminos más cortos, no siempre es cierto que teniendo un arista e = (x, y), sea posible “transitar” de x a y y viceversa. Esta diferencia es mostrada por la estructura matemática conocida como digrafo. En una ciudad, una calle que conecte varias intersecciones es representada por un conjunto de aristas (e_1, e_2, e_3, e_4, ... , e_n), las cuales forman una ruta comenzando por un vértice v_0 que representa el inicio de la calle, pasando por n vértices o intersecciones hasta su finalización. Si la calle es de un sólo sentido, la ruta inversa formada por las aristas (e_n, e_{n-1}, e_{n-2}, e_{n-3}, ... , e_1) sería inválida a pesar de tener conectados los vértices por dichas aristas. Es entonces cuando al conjunto de aristas E del grafo representativo de la ciudad se le asigna una dirección, y en cada arista e_i = xy se nombra a x como el nodo inicial y a y como el nodo o vértice final del arista. La dirección es indicada en una representación gráfica por una flecha. Esta misma adecuación es aplicable a los Multigrafos.

En la figura 2.8 se muestra un ejemplo de esto. Es posible hacer el recorrido iniciando en el vértice 3, pasando por el vértice 1 y finalizando en 2, más no viceversa.

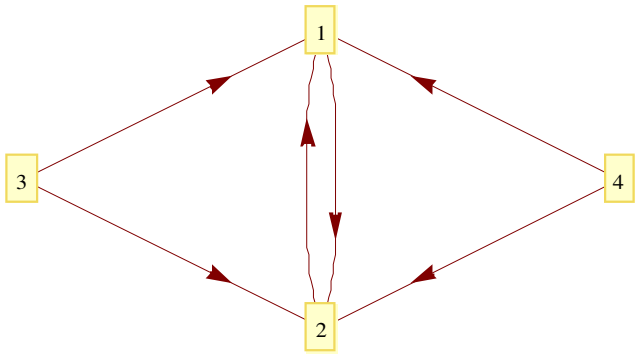


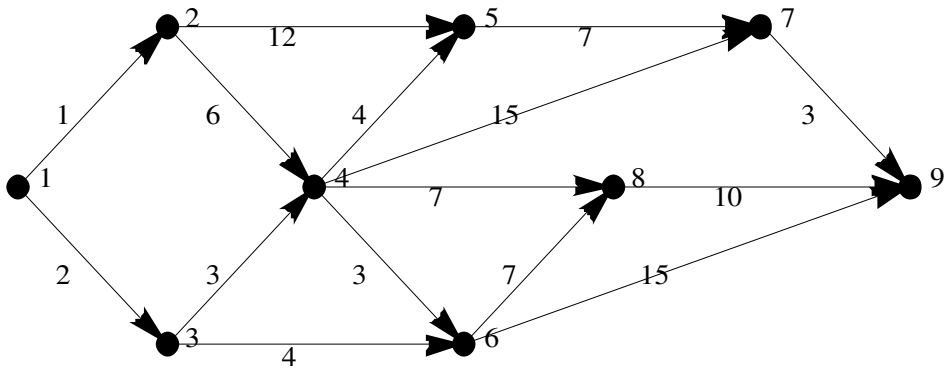
Figura 2.8 Digrafo de 4 vértices y 6 aristas.

```
GraphPlot[{1 -> 2, 2 -> 1, 3 -> 1, 3 -> 2, 4 -> 1, 4 -> 2},
          VertexLabeling -> True, DirectedEdges -> True]
```

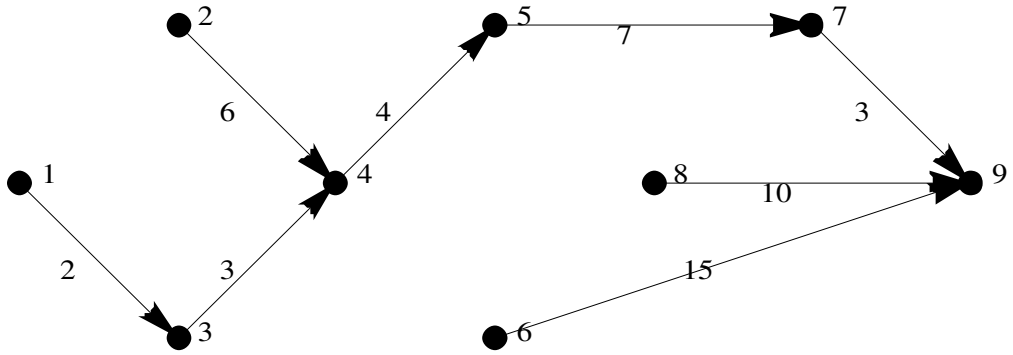
Árboles

Se le llama árbol a aquel grafo conexo en el cuál existe una única ruta entre cualesquier par de vértices del mismo. En el caso de la ejecución del algoritmo de Dijkstra sobre un grafo G = (V, E), el cual encuentra la ruta más corta desde un vértice al resto de los mismos, se tiene como resultado un subgrafo de expansión G, puesto que el conjunto de vértices de dicho subgrafo es igual a V. Por otro lado, al mostrar las rutas óptimas entre el vértice inicial s y el resto de los vértices en V, existe una sóla ruta entre dicho vértice inicial y el resto de los vértices, por lo que el grafo resultante es un árbol.

El concepto de árbol mínimo de expansión indica que el árbol de expansión resultante de dicho algoritmo contiene la sumatoria mínima de las aristas respecto al resto de árboles de expansión posible del grafo G. Un ejemplo de árbol mínimo de expansión se ilustra en la Figura 2.9, donde a partir del digrafo D de la Figura 2.9 (a) se obtiene el árbol dirigido de la Figura 2.9 (b), el cuál es el árbol mínimo de expansión de D.



(a)



(b)

Figura 2.9 A partir del digrafo D (a) se obtiene su árbol mínimo de expansión (b).

```
E351 = {{1, 2}, {2, 4}, {2, 5}, {1, 3}, {3, 4}, {3, 6}, {4, 5}, {4, 6},
     {4, 7}, {4, 8}, {5, 7}, {6, 8}, {6, 9}, {7, 9}, {8, 9}};
W351 = {1, 6, 12, 2, 3, 4, 4, 3, 15, 7, 7, 7, 15, 3, 10};
CR351 = {1 -> {0, 0}, 2 -> {1, 1}, 3 -> {1, -1}, 4 -> {2, 0}, 5 -> {3, 1},
     6 -> {3, -1}, 7 -> {5, 1}, 8 -> {4, 0}, 9 -> {6, 0}};
VU351 = {{0, 0}, {1, 1}, {1, -1}, {2, 0}, {3, 1}, {3, -1}, {5, 1}, {4, 0}, {6, 0}};
G351 = SetEdgeLabels[ChangeVertices[SetEdgeWeights[AddEdges[MakeDirected[
     EmptyGraph[9]], E351], W351], VU351], W351];
RG351 = ReverseEdges[G351];
ShowGraph[G351, VertexLabel -> True]
SPST351 = ChangeVertices[SetEdgeLabels[AddEdges[MakeDirected[EmptyGraph[9]],
     Edges[ShortestPathSpanningTree[RG351, 9]]], {2, 6, 3, 4, 7, 15, 3, 10}],
     VU351];
ShowGraph[SPST351, VertexLabel -> True]
```

De manera natural, y tal como lo hizo Euler, es posible modelar una red de tráfico vehicular por sus vías e interconexiones. Utilizando la estructura matemática conocida como grafo dirigido o digrafo, se agrega la dirección de las rutas de la red de tráfico vehicular. Otro dato inmediato para el modelado de una red de tráfico es la distancia entre los vértices conectados por las aristas. Dos vértices a y b incidentes en un arista e son gráficamente representados mediante una línea recta, que vendría siendo el arista e. La distancia Euclidiana entre ambos vendría siendo la ponderación del arista e, es decir, si entre el vértice A y el vértice B existe una distancia Euclidiana de 100 metros, la ponderación del arista e será de 100. Un ejemplo es la Figura 2.10, donde se muestra un grafo ponderado mostrando las distancias entre sus vértices como valor de las aristas.

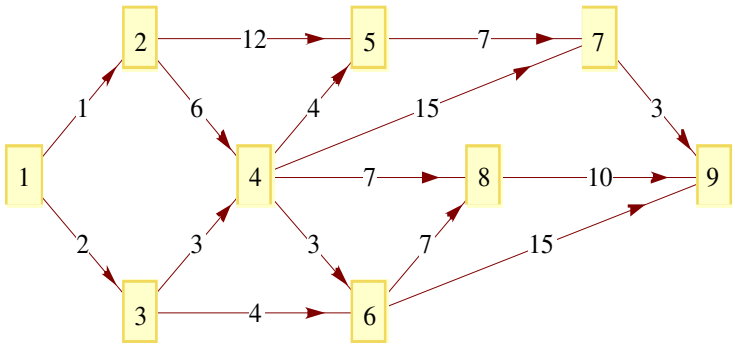


Figura 2.10 Digrafo ponderado de 9 nodos y 15 aristas.

```
E350 = {{1 -> 2, "1"}, {2 -> 4, "6"}, {2 -> 5, "12"}, {1 -> 3, "2"},
     {3 -> 4, "3"}, {3 -> 6, "4"}, {4 -> 5, "4"}, {4 -> 6, "3"},
     {4 -> 7, "15"}, {4 -> 8, "7"}, {5 -> 7, "7"}, {6 -> 8, "7"},
     {6 -> 9, "15"}, {7 -> 9, "3"}, {8 -> 9, "10"}};
CR350 = {1 -> {0, 0}, 2 -> {1, 1}, 3 -> {1, -1}, 4 -> {2, 0}, 5 -> {3, 1},
     6 -> {3, -1}, 7 -> {5, 1}, 8 -> {4, 0}, 9 -> {6, 0}};
GraphPlot[E350, VertexLabeling -> True, VertexCoordinateRules -> CR350]
```

Una variante a este modelado se da cuando se tienen dos vértices a y b adjacentes, con una distancia relativamente larga entre ambos y teniendo la forma del arista no como una línea recta sino como una curva. Una solución a este problema es ponderar el arista e

con la distancia real más no Euclidiana entre ambos vértices. ¿Cuál es la desventaja de esto? Que el tramo representado por el arista no es necesariamente igual en todo su recorrido. Existen diversas variables que pueden afectar al flujo del tráfico vehicular durante el tramo completo. Se propone entonces dividir el arista en aristas más pequeños, de tal forma que la suma total de las distancias Euclidianas de éstas aristas sea igual a la distancia real entre los vértices originales a y b.

En mi opinión, se tienen dos tipos principales de variables para realizar la división: una manera de dividirlo sería tomando en cuenta aspectos físicos, por ejemplo una curva cerrada en una avenida de alta velocidad o una carretera, que ocasiona un tráfico más lento dado que una velocidad relativamente alta podría ocasionar un accidente; la otra manera sería tomar en cuenta aspectos abstractos de la distribución de la ciudad, como pudiera ser la ubicación de uno de los principales centros comerciales de la ciudad o un edificio gubernamental en cierta parte del arista: este tipo de elementos, a pesar de no ser físicamente intersecciones, si fungen de cierta manera como tales, ya que son un destino común para los conductores, teniendo flujo de entrada y salida de autos en estos puntos intermedios. Nótese que el flujo vehicular sobre éstos vértices abstractos depende en gran manera de sucesos que no tienen nada que ver con la topología de la ciudad: un concierto, un partido de fútbol o la fecha límite para realizar un pago incrementarían el flujo vehicular hacia éstos nódos abstractos.

¿Es la distancia entre cualesquier par de vértices a y b un factor suficiente para el modelado de una red de tráfico vehicular? Todo depende del análisis que se piense realizar. Como se mencionó anteriormente, existen aspectos naturales en las ciudades que vuelven más compleja la valoración del arista. Existe una teoría completa, llamada Traffic Flow Theory (o Teoría del Flujo del Tráfico), que se encarga de modelar, analizar y predecir el comportamiento del tráfico vehicular, y contempla dos tipos de modelos. El primero, llamado modelado macroscópico, representa al tráfico vehicular a manera de un flujo arterial: asume que hay el suficiente número de carros en una vía de tal manera que el flujo de autos puede ser tratado como un fluído corriendo a través de un canal. Esto se logra expresando el flujo a partir de 3 variables principales: la frecuencia representada en términos de autos por unidad

de tiempo, la velocidad del flujo expresada por la distancia recorrida por unidad de tiempo, y la densidad del tráfico representada por el número de autos que caben en una línea de tráfico de determinada longitud. El segundo modelo, denominado modelado microscópico, está basado en la interacción entre los autos que circulan por una vía de tráfico, modelando las respuestas a estímulos dados por autos contiguos (Dym, 2004).

La Teoría del Flujo Vehicular presenta diversos problemas prácticos. Un ejemplo es el análisis de los patrones existentes en las redes de tráfico vehicular a distintos niveles (nacional, estatal o a nivel ciudad) que realizan los gobiernos para verificar que su eficiencia dadas las condiciones urbanas actuales. Se pueden realizar cambios en las redes de diversas formas, como puede ser el agregar una nueva calle o avenida en cierto punto de la ciudad, incrementar el número de carriles de alguna vialidad, introducción de sistemas de control de tráfico en ciertas zonas para reducir el flujo de autos o el tiempo que tardan en atravesar dicho sector, o la re-orientación de la dirección de algunas calles, entre otros. En el análisis, también es importante considerar las variables a través del tiempo, para proveer de soluciones a largo plazo. Entre los datos a recopilar, Foulds (1995) menciona los siguientes:

- ¿Cuál es la magnitud del flujo del tráfico? ¿Es homogenea o cambia dependiendo de ciertos factores?
- ¿Dónde se origina el tráfico y cuáles es el origen del mismo?
- ¿Es posible implementar otras modalidades de tráfico vehicular para incrementar la capacidad de los aristas?
- ¿Hacia dónde se dirige el tráfico vehicular? ¿Cuáles son los principales destinos del tráfico?
- ¿Cuáles son las capacidades actuales de las aristas? ¿Puede ser incrementada la densidad, es decir, el número de autos por línea de tráfico en ciertas avenidas?
- ¿Cuál es la velocidad actual del tráfico? ¿Cuáles son los tiempos estimados de recorrido entre cualesquier par de nodos dentro de la red? ¿De que dependen estas aproximaciones? ¿Sería posible incrementar estos tiempos añadiendo vialidades?

Investigar bajo que términos urbanos se definen las ciudades y realizar el cómputo

de éstas variables para predecir el comportamiento del Flujo Vehicular es algo que se encuentra fuera del alcance de ésta investigación. Pero, mediante técnicas de Programación Dinámica, podemos modelar algoritmos de orden de complejidad polinomial (eficientes), que permitan realizar un cómputo exhaustivo dentro del análisis de rutas. Con esto, se pueden obtener bases suficientes en la toma de decisiones acerca del re-diseño y optimización de las redes de tráfico vehicular a analizar.

III. ALGORITMOS ENFOCADOS AL MODELADO DE GRAFOS

Dentro del modelado de una red de tráfico vehicular, existen aspectos que requieren de especial atención al representar dicha red mediante un grafo, como lo es la ponderación de las aristas, donde se representan variables que influyen en el análisis de la red, así como términos de conectividad, utilizados para representar matemáticamente las rutas dentro de la red. Es trivial mencionar que, dentro de una red de tráfico vehicular, la distancia de un nodo o intersección a sí mismo es 0. Lo mismo para nodos inalcanzables: la distancia entre cualquier nodo y un nodo que se encuentre inalcanzable será igual a Infinito.

Aprovechando la ventaja multiplataforma de *Java*, las diversas herramientas open-source que existen para este lenguaje y el alto nivel de escalabilidad entre las mismas, podemos realizar un sencillo modelado utilizando las librerías propias del lenguaje más 2 frameworks: *JGraph* y *JGraphT*. Más adelante al abordar el caso de estudio se mostrará la interacción entre ambos paradigmas, la programación funcional de *Mathematica* y el paradigma orientado a objetos de *Java*.

JGraphT es una librería open-source para *Java* que modela a un grafo mediante el paradigma de orientación a objetos. Presenta interfaces para las diferentes estructuras matemáticas derivadas de un grafo, como son el digrafo, grafos ponderados, multigrafos, etc. Además, basa el control de aristas en un grafo en una interfaz llamada *EdgeFactory*, que viene a representar una función de incidencia entre dos vértices. Para su uso, estas interfaces pueden ser implementadas y extendidas o bien pueden utilizarse las implementaciones que provee la librería. Presenta además la implementación de una conexión con el otro framework, *JGraph*, el cual es una librería para la visualización de gráficos utilizando la librería *Swing*. Esto permite utilizar las imágenes generadas como páneles nativos *Java*, y presentarlos en una ventana ad-hoc con el sistema operativo utilizado. *JGraphT*, al proveer implementaciones de estructuras matemáticas como el digrafo y los grafos ponderados, permite realizar de manera natural el modelado de algoritmos bajo estas estructuras, ahorrando el tiempo de realizar el diseño e implementación de un prototipo de trabajo orientado a objetos.

Búsqueda en Anchura como Algoritmo de Conectividad

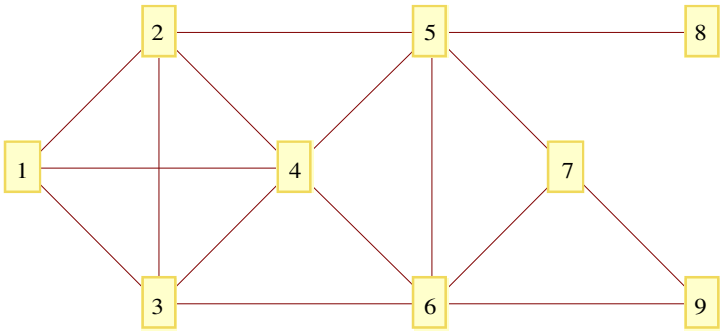
Teniendo un digrafo D, que represente una red de tráfico vehicular, si tenemos al menos un nodo inalcanzable dentro del digrafo, nuestra red se volverá completamente ineficiente desde el punto de vista del grado de acceso: teniendo al menos una distancia igual a infinito dentro las rutas óptimas convertirá la sumatoria en infinito, dando un grado de acceso igual a 0 al al dividir la sumatoria de las distancias Euclidianas de todos los pares de puntos entre infinito.

Esto quiere decir que el digrafo necesita al menos un camino entre cualquier vértice y el resto, es decir, es necesario que el digrafo en cuestión sea un componente fuertemente conexo (en inglés, Strongly Connected Component), es decir, que para cualquier par de vértices del digrafo exista un camino (walk) entre ellos.

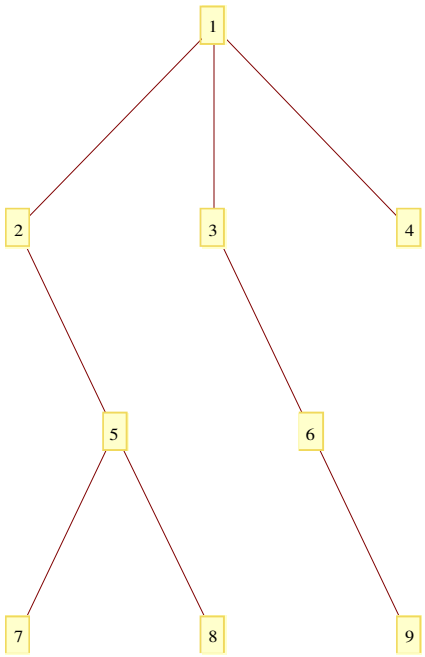
Jungnickel (2007) señala para estos propósitos al algoritmo de *Búsqueda de Primero en Anchura* o simplemente *Búsqueda en Anchura* (en inglés, *Breadth First Search*, también conocido por sus siglas *BFS*), el cual generalmente es usado para encontrar la ruta más corta entre un par de vértices en un grafo no ponderado G = (V, E), tratando de determinar la distancia en aristas utilizadas en un camino, es decir, la distancia |w| igual a la longitud de dicho camino.

Dicho algoritmo no utiliza ningún método heurístico de optimización, simplemente recorre el grafo partiendo desde un vértice arbitrario tomado como origen, tomando los vértices adyacentes a él, y así sucesivamente hasta terminar de recorrer por completo el grafo, por lo que su orden de complejidad es de O(|E|). Los vértices o nodos visitados son almacenados en una lista, esto con el fin de evitar un recorrido redundante en alguna iteración, de etiquetar los vértices para identificar bajo que nivel o distancia se encuentran los vértices recorridos respecto al vértice inicial e identificar el orden de recorrido de los vértices. Un ejemplo de esto se observa en la Figura 3.1. Partiendo de (a) se realiza una traza por todos los nodos adyacentes desde el vértice 1, mostrada jerárquicamente en (b). La jerarquía de los vértices muestra la distancia de dichos vértices al vértice inicial: los vértices 2, 3 y 4

son adyacentes con el vértice 1, por lo que su distancia es 1; los vértices 5 y 6 están a 2 aristas de distancia, mientras que los vértices 7, 8 y 9 se encuentran a 3.



(a)



(b)

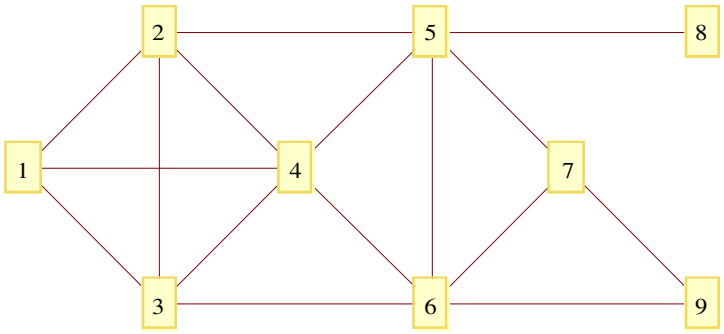
Figura 3.1 Los vértices del grafo en (a) se muestran jerarquizados en (b) por la distancia respecto al nodo arbitrario inicial 1.

En la Figura 3.2 se muestra a manera de pseudo-código el algoritmo BFS. La implementación bajo *Java* se encuentra en el Anexo A.1. *Mathematica* por su parte, en el paquete *Combinatorica*, posee una implementación bajo la función *BreadthFirstTraversal*. En

la Figura 3.3 se codifica el grafo de la Figura 3.1 (a) haciendo uso de dicho paquete. La primer salida muestra la distancia de cada vértice al nodo inicial 1, representada gráficamente en la Figura 3.1 (b). La segunda salida muestra el orden en el que fueron tomados los vértices en el recorrido.

```
Input: Grafo G = (V, E), Vertice inicial v

Q = EmptyQueue
M = EmptyMap

M[v] = 0
Q.push(v)

While |Q| > 0 Do
   v' = Q.pop()

   ForAll v'' adjacent to v'
      If M[v''] is undefined
         M[v''] = M[v'] + 1
         Q.push(v'')
      End If

   End For

End Do

Return M
```

Figura 3.2 Pseudo-código del algoritmo de *Búsqueda por Primero en Anchura*.

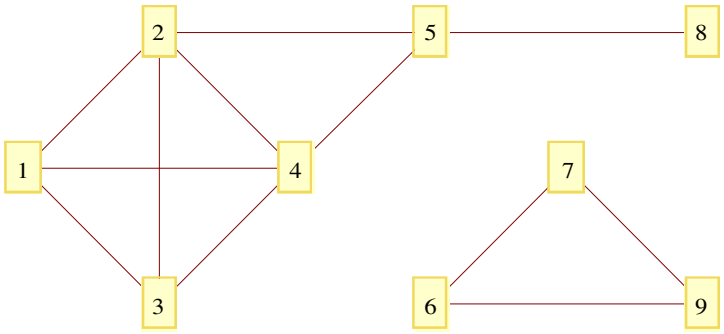



```
Out[1] = {0,1,1,1,2,2,3,3,3}
Out[2] = {1,2,3,4,5,6,7,8,9}
```

Figura 3.3 Utilización del algoritmo *BFS* para determinar si ungrafo es conexo.
La primer salida muestra las distancias de cada vértice al vértice origen 1.
La segunda salida muestra el orden en el que fueron tomados los vértices.

```
Needs["Combinatorica`"];
G403 = AddEdges[EmptyGraph[9], {{1, 2}, {1, 3}, {2, 3}, {2, 4}, {3, 4}, {4, 1},
     {4, 5}, {5, 6}, {5, 2}, {5, 7}, {6, 3}, {6, 4}, {6, 7}, {6, 9}, {8, 5},
     {9, 7}}];
GraphPlot[G403, VertexLabeling -> True, VertexCoordinateRules -> {1 -> {0, 0},
     2 -> {1, 1}, 3 -> {1, -1}, 4 -> {2, 0}, 5 -> {3, 1}, 6 -> {3, -1},
     7 -> {4, 0}, 8 -> {5, 1}, 9 -> {5, -1}}]
BreadthFirstTraversal[G403, 1, Level]
BreadthFirstTraversal[G403, 1]
```

Si en el grafo de la Figura 3.3 aislamos los vértices 6, 7 y 9, y aplicamos el algoritmo de *Búsqueda en Anchura* desde el vértice 1, estos 3 vértices no podrán ser alcanzados bajo ningún camino tendrán un valor de 0 en el orden de visita, mientras que la distancia a la cual quedan respecto al vértice 1 será de ∞. Caso contrario si cualquiera de los vértices 6, 7 o 9 es el vértice inicial en la búsqueda: el resto de los vértices no serán utilizados y la distancia respecto al vértice inicial será igual a ∞. La Figura 3.4 muestra el grafo con los vértices aislados y las 4 salidas del algoritmo de *Búsqueda en Anchura* tomando como vértices iniciales 1 y 6.



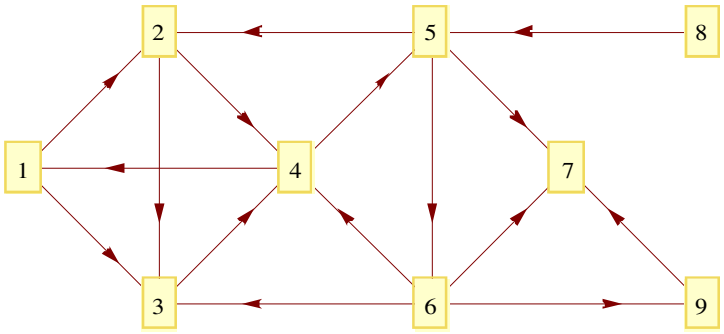

```
Out[1] = {0,1,1,1,2,∞,∞,3,∞}
Out[2] = {1,2,3,4,5,0,0,6,0}
Out[3] = {∞,∞,∞,∞,∞,0,1,∞,1}
Out[4] = {0,0,0,0,0,1,2,0,3}
```

Figura 3.4 Se aislan los vértices 6, 7 y 9 del grafo de la Figura 3.3 y se realiza la Búsqueda en Anchura desde los vértices 1 y 6.

```
G404 = DeleteEdges[G403, {{3, 6}, {4, 6}, {5, 6}, {5, 7}}];
CR400 = {1 -> {0, 0}, 2 -> {1, 1}, 3 -> {1, -1}, 4 -> {2, 0},
     5 -> {3, 1}, 6 -> {3, -1}, 7 -> {4, 0}, 8 -> {5, 1}, 9 -> {5, -1}};
GraphPlot[G404, VertexLabeling -> True, VertexCoordinateRules -> CR400]
BreadthFirstTraversal[G404, 1, Level]
BreadthFirstTraversal[G404, 1]
BreadthFirstTraversal[G404, 6, Level]
BreadthFirstTraversal[G404, 6]
```

Se puede concluir que para un grafo no dirigido finito, la *Búsqueda de Primero en Anchura* determina si dicho grafo es conexo bajo un orden de complejidad de O(|E|).

Lamentablemente, esta solución no es aplicable a un digrafo. En un grafo no dirigido resulta indiferente tomar un arista desde cualquiera de los dos vértices incidentes, puesto que las aristas no poseen dirección. Agregando dirección a las aristas, no es trivial el tomar las aristas adyacentes al vértice en uso en una iteración, sino que hay que tomar las aristas salientes de dicho vértice. Aplicando este algoritmo para el digrafo de la Figura 3.5 (que es el mismo que el de la Figura 3.3 pero con aristas dirigidas) da como resultado que no hay ningún camino desde el vértice 1 al vértice 8. Caso contrario en el sentido inverso: desde el vértice 8 encontramos que hay caminos para el resto de los vértices.




```
Out[1] = {0,1,1,2,3,4,4,∞,5}
Out[2] = {1,2,3,4,5,6,7,0,8}
Out[3] = {4,2,3,3,1,2,2,0,3}
Out[4] = {9,3,6,7,2,4,5,1,8}
```

Figura 3.5 Se dirigen las aristas del grafo de la Figura 3.3.La búsqueda en anchura muestra que no hay caminos desde el vértice 1 al 8, pero si viceversa.

```
G402 = AddEdges[MakeDirected[EmptyGraph[9]], {{1, 2}, {1, 3}, {2, 3}, {2, 4},
     {3, 4}, {4, 1}, {4, 5}, {5, 6}, {5, 2}, {5, 7}, {6, 3}, {6, 4}, {6, 7},
     {6, 9}, {8, 5}, {9, 7}}];
CR400 = {1 -> {0, 0}, 2 -> {1, 1}, 3 -> {1, -1}, 4 -> {2, 0}, 5 -> {3, 1},
     6 -> {3, -1}, 7 -> {4, 0}, 8 -> {5, 1}, 9 -> {5, -1}};
GraphPlot[G402, DirectedEdges -> True, VertexLabeling -> True,
     VertexCoordinateRules -> CR400]
BreadthFirstTraversal[G402, 1, Level]
BreadthFirstTraversal[G402, 1]
BreadthFirstTraversal[G402, 8, Level]
BreadthFirstTraversal[G402, 8]
```

Jungnickel señala que si al aplicar el algoritmo de *Búsqueda de Primero en Anchura* en cada vértice v de un digrafo G = (V, E) resultan siempre definidas las etiquetas de cada vértice, es decir, el conjunto V' de vértices etiquetados siempre es igual a V, estaremos teniendo un digrafo fuertemente conexo. En la Figura 3.6 presento el resultado de ejecutar la *Búsqueda en Anchura* para todos los vértices del digrafo de la Figura 3.5. Sólo tomando como vértice origen el vértice 8 es posible encontrar un camino al resto de los vértices. El orden de complejidad resultante de este procedimiento es de O(|V||E|)), dado que por cada vértice se recorren todas las aristas una vez.

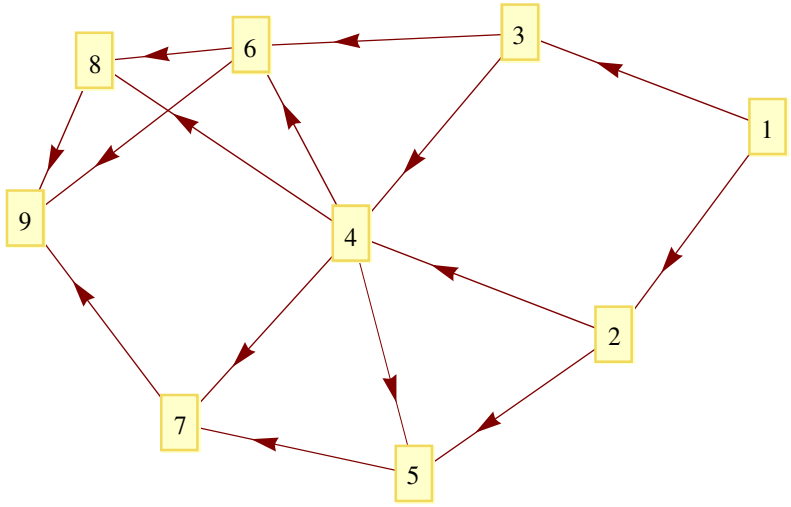
```
{{0,1,1,2,3,4,4,∞,5}, {2,0,1,1,2,3,3,∞,4},
 {2,3,0,1,2,3,3,∞,4}, {1,2,2,0,1,2,2,∞,3},
 {3,1,2,2,0,1,1,∞,2}, {2,3,1,1,2,0,1,∞,1},
 {∞,∞,∞,∞,∞,∞,0,∞,∞}, {4,2,3,3,1,2,2,0,3},
                {∞,∞,∞,∞,∞,∞,1,∞,0}}
```

Figura 3.6 Resultado de aplicar el algoritmo de *BFS* sobre todos los vértices del digrafo de la Figura 3.5. Sólo desde el vértice 8 es posible encontrar un camino hacia el resto de los vértices.

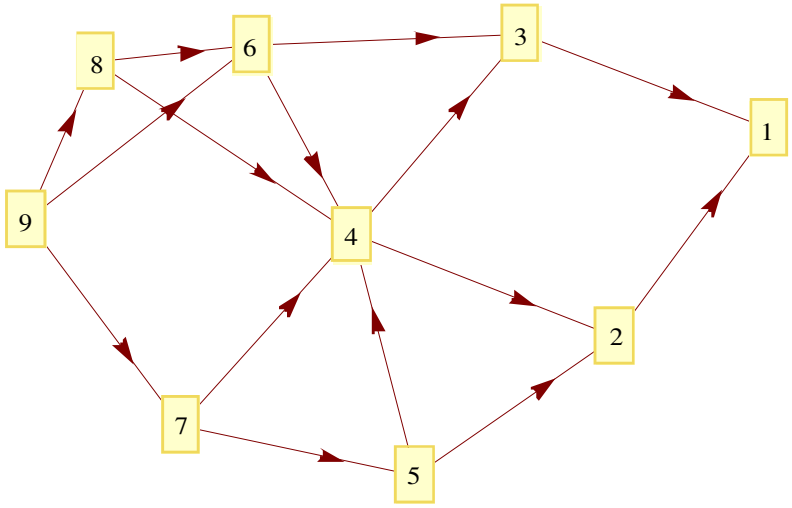
```
Table[BreadthFirstTraversal[G402, i, Level], {i, 9}]
```

Bajo mi opinión, utilizar este algoritmo sobre un digrafo tomando como raíz un vértice arbitrario v, es aplicable en el caso de que se busque la ruta más corta desde cualquier vértice en un digrafo a un vértice en específico, como en el algoritmo prototipo de la Programación Dinámica de Denardo que se mostrará en el capítulo siguiente. Si se invierte la dirección de las aristas de un digrafo, tomando un vértice destino f como raíz y aplicamos el algoritmo *BFS*, es posible determinar si existe un árbol de expansión con las rutas más cortas desde cualquier vértice al vértice destino f, ya que para cada vértice existiría al menos un camino al vértice destino f.

Dicha observación se presenta en la Figura 3.7. Se toma el digrafo (a) y se invierte la dirección de todas sus aristas, resultando en el digrafo (b). Al aplicar el algoritmo de *Búsqueda en Anchura* desde el vértice final 9, se encuentra que existe al menos un camino para el resto de los vértices, tal como lo indican las distancias resultantes en (c). Con este resultado es posible afirmar que para todas los vértices del digrafo existe al menos un camino hacia el vértice final f = 9. Es trivial también decir que existe un camino del vértice 9 a si mismo con distancia igual a 0.



(a)



(b)

```
Out[66] =  {3,3,2,2,2,1,1,1,0}
```

(c)

Figura 3.7 Resultado de la aplicación del algoritmo de *Búsqueda en Anchura* sobre el digrafo (a) desde el vértice 9, invirtiendo sus aristas (b). En (c) aparecen todos los vértices etiquetados.

```
E351 = {{1, 2}, {2, 4}, {2, 5}, {1, 3}, {3, 4}, {3, 6}, {4, 5}, {4, 6},
     {4, 7}, {4, 8}, {5, 7}, {6, 8}, {6, 9}, {7, 9}, {8, 9}};
W351 = {1, 6, 12, 2, 3, 4, 4, 3, 15, 7, 7, 7, 15, 3, 10};
G351 = SetEdgeWeights[AddEdges[MakeDirected[EmptyGraph[9]], E351], W351];
RG351 = ReverseEdges[G351];
GraphPlot[G351, DirectedEdges -> True, VertexLabeling -> True]
GraphPlot[RG351, DirectedEdges -> True, VertexLabeling -> True]
BreadthFirstTraversal[RG351, 9, Level]
```

Grafos cíclicos y acíclicos

Dentro del enfoque de las redes de tráfico vehicular, los ciclos dentro de una red son un aspecto fundamental, dado que permiten analizar el costo de rutas críticas. Por ejemplo, en un servicio de entrega de paquetes, el repartidor tendrá que optimizar su recorrido para regresar al lugar de partida utilizando de la mejor manera tiempo y energía. Esto hace que cobre importancia un algoritmo que se encargue de analizar los ciclos como aspecto de conectividad de un digrafo aplicado dentro del entorno de las redes de tráfico vehicular.

Para determinar si el digrafo a analizar es acíclico, Denardo (2003) menciona un método basado en la etiquetación de los vértices del grafo, donde dice que un digrafo acíclico consiste de N vértices etiquetados con los números enteros 1 a N, de tal manera para cualquier arista dirigida (i, j) i es menor a j. Para esto, se necesita que exista al menos un vértice v cuyas aristas ninguna termine en v, es decir, que la dirección de todas sus aristas emane de v.

El procedimiento es el siguiente: se toma dicho vértice y se etiqueta con el número 1. Posteriormente se eliminan sus aristas adyacentes y se busca el siguiente vértice que no tenga ninguna arista terminal. En caso de que en cualquier parte de este algoritmo recursivo (incluído el inicio) existan dos vértices con dichas características, la numeración es intercambiable. Pero en caso de que en algún punto de la recursión no exista ningún vértice con dichas características, estaremos encontrando un ciclo dentro del digrafo.

En la Figura 3.8 (a), se muestra un digrafo finito con 4 vértices y 5 aristas. Los nodos o vértices están ubicados en cada uno de los puntos cardinales. Por inspección, se encuentra que el vértice situado al norte no tiene aristas terminando en el, por lo que es etiquetado con el número 1, y se eliminan sus vértices adyacentes. El digrafo resultante se muestra en la Figura 3.8 (b), donde ahora, nuevamente por inspección, se observa que el vértice situado al oeste no tiene aristas terminando en el, por lo que lo es etiquetado con el número 2 y se eliminan sus aristas adyacentes. El digrafo resultante de la segunda iteración, mostrado en la Figura 3.8 (c), contiene sólo 1 arista, del vértice situado al sur hacia el vértice

situado al este, por lo cual los se etiquetan los vértices con los números 3 y 4 respectivamente y se finaliza el procedimiento. El digrafo resultante del procedimiento completo es el mostrado en la figura 3.9.

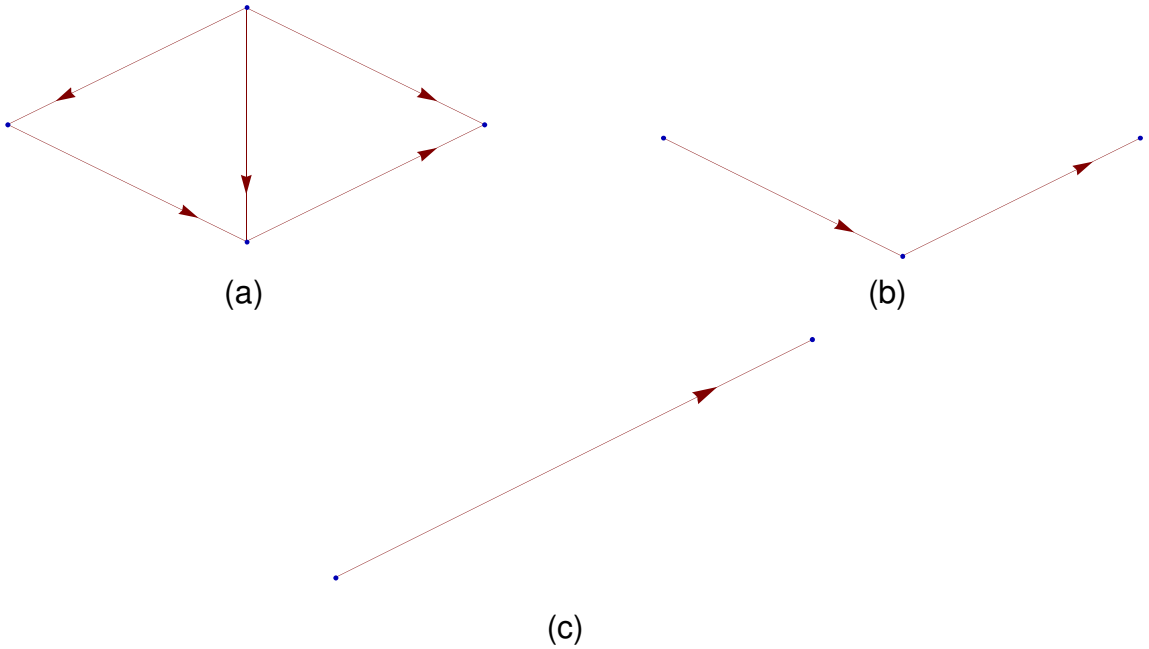


Figura 3.8 Iteraciones del algoritmo de etiquetado para determinar si el digrafo es acíclico.
(a) Digrafo inicial, (b) después de retirar el vértice norte, (c) después de retirar el vértice oeste.

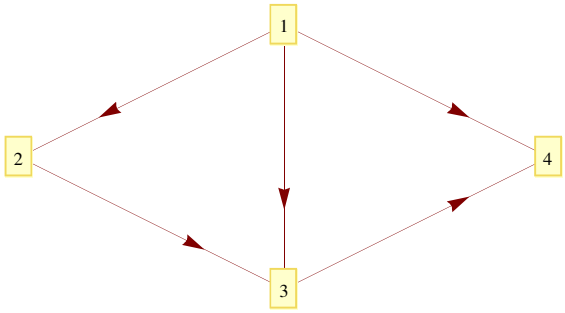


Figura 3.9 Digrafo etiquetado resultante.

```
E311 = {{1, 2}, {1, 3}, {1, 4}, {2, 3}, {3, 4}};
G311 = SetVertexLabels[AddEdges[MakeDirected[EmptyGraph[4]], E311], {1, 2, 3, 4}];
G312 = DeleteVertex[DeleteEdges[G311, {{1, 2}, {1, 3}, {1, 4}}], 1];
G313 = DeleteVertex[DeleteEdges[G312, {{1, 2}}], 1];
CR311 = {1 -> {0, 0.5}, 2 -> {-1, 0}, 3 -> {0, -0.5}, 4 -> {1, 0}};
Table[GraphPlot[i, DirectedEdges -> True, VertexCoordinateRules -> CR311],
     {i, {G311, G312, G313}}]
GraphPlot[G311, DirectedEdges -> True, VertexCoordinateRules -> CR311,
     VertexLabeling -> True]
```

El digrafo resultante de la Figura 3.9 cumple con lo indicado por Denardo, ya que para cualquier arista dirigida e = (i, j), i < j, siendo entonces un digrafo finito acíclico.

La implementación de dicho algoritmo bajo el lenguaje orientado a objetos *Java* se presenta en el Anexo A.2, mientras que el pseudo-código se muestra en la Figura 3.10. Tal como lo muestra el pseudo-código, en cada iteración se busca al menos un vértice sin aristas terminando en él. En caso de que en alguna iteración se encuentre un vértice sin dichas características, tendremos que el digrafo contiene un ciclo. Una vez etiquetados los nodos, se eliminan los vértices y sus correspondientes aristas para iniciar una siguiente iteración. Utilizando la herramienta de visualización *JGraph* se muestra el etiquetado resultante mediante el dump screen al panel nativo de *Java* en Figura 3.11

```
Input: Digrafo D = (V, E)

Index = 0
While |V| > 0 Do
   List L = EmptyList

   ForAll v in V
      If |IncomingEdges of v| == 0
         L.push(v)
      End If

   End For

   If |L| == 0
      Throw Error: "El Grafo no es acíclico"
   End If

   ForEach v in L
      v.Label = Index
      Index = Index + 1
   End For

   D.removeAll(L)

End Do
```

Figura 3.10 Pseudo-código del algoritmo de etiquetado de Denardo.

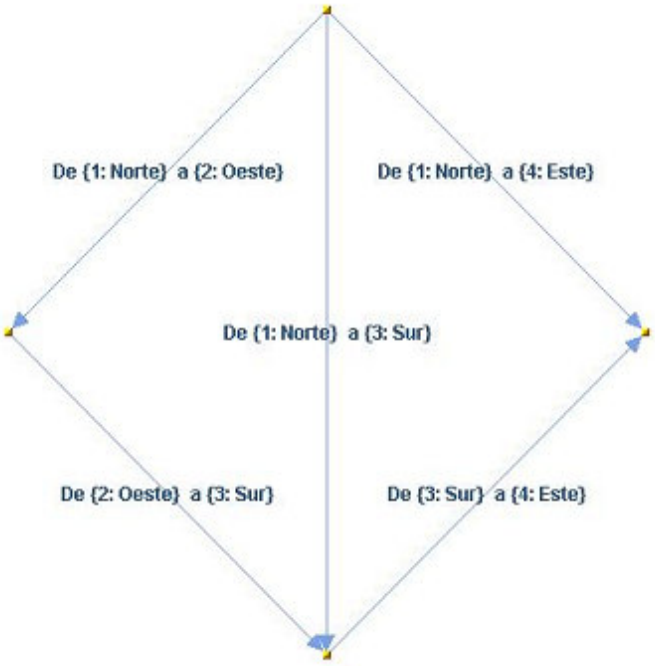


Figura 3.11 Digrafo etiquetado resultante y presentado bajo la librería *JGraph* en *Java*.

Otro ejemplo de la aplicación de este algoritmo es el digrafo acíclico de la Figura 3.12. Más adelante observaremos se hará uso de este etiquetado bajo algoritmos que hagan uso de la técnica de la Programación Dinámica.

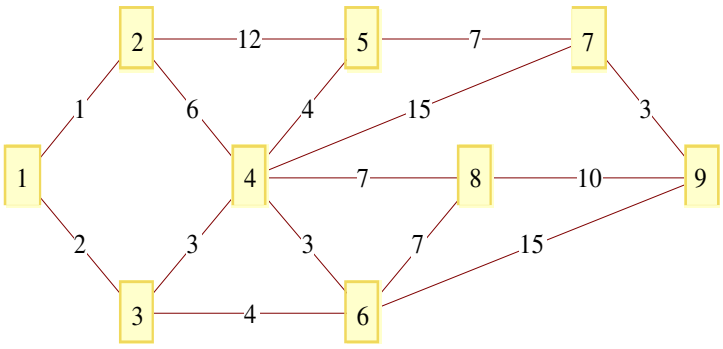


Figura 3.12 Digrafo acíclico etiquetado bajo el algoritmo de Denardo.

IV. PROGRAMACIÓN DINÁMICA

La programación dinámica es un enfoque de optimización que transforma un problema complejo en una secuencia de problemas simples; su característica esencial es la naturaleza multi-etapa del procedimiento (Bradley et. al., 1977). Esta naturaleza multietapa se traduce en un modelado recursivo, es decir, en modelar las funciones de tal manera que para obtener el resultado final la función se llama a sí misma hasta evaluar un punto final estático. Un ejemplo de esto es la serie de Fibonacci, definida en la ecuación 2:

F(0) = 0.
F(1) = 1.
F(n) = F(n-1) + F(n-2), para cualquier entero n > 1. (2)

Para obtener un número Fibonacci n > 1, la función se llama a sí misma hasta que n toma el valor de 0 o de 1. Una implementación simple en el lenguaje orientado a objetos *Java* es la observada en la Figura 4.1.

```
public static long recursiveFib(long n) {
    return (n == 0) ? 0 : (n == 1) ? 1 : recursiveFib(n-1) +
          recursiveFib(n-2);
}
```

Figura 4.1 Implementación recursiva en *Java* de la serie de Fibonacci.

Esta implementación en una sóla línea aparentaría ser eficiente en términos de la definición de la secuencia, pero no es así. El compilador, al simular la recursión, realiza cálculos redundantes de los valores de F(n-1) y F(n-2). Teniendo que calcular F(n), con n > 1, se ejecutaría una llamada recursiva a F(n-1) y a F(n-2). Al hacer el cálculo de F(n-1), nuevamente se invocaría el cálculo de F(n-2), adicional al cálculo ya hecho. En la figura 4.2 se muestra la traza recursiva e ineficiente de la implementación presentada.

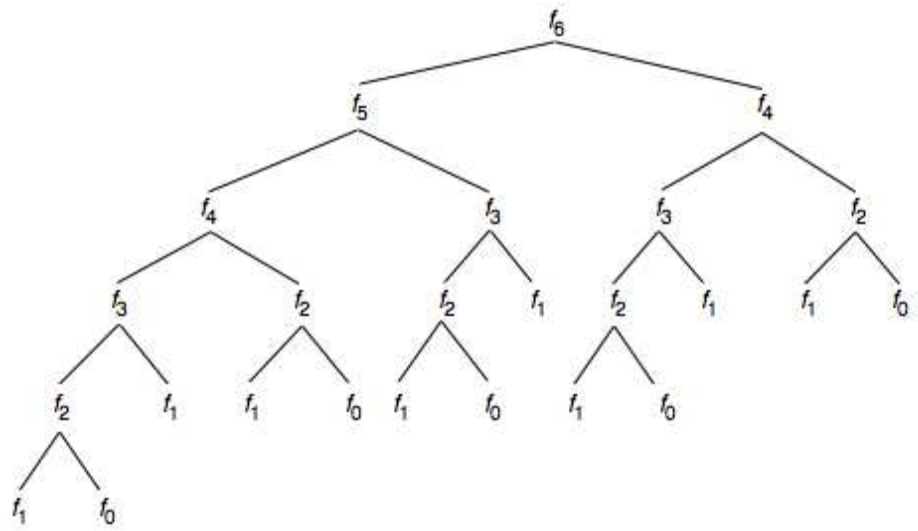


Figura 4.2 Traza recursiva de llamadas a la función recursiveFib.

Como vemos, el número de invocaciones resulta redundante. Para calcular tan sólo el sexto número Fibonacci, F(2) se calcula 5 veces, F(3) 3 veces, F(4) dos veces, etc. Una solución a esto lo brinda la técnica de la Programación Dinámica. Si se almacenaran en una tabla o registro los dos últimos valores calculados, sería suficiente para poder calcular cualquier número Fibonacci. La recursividad se transformaría entonces en una iteración, iniciando desde F(2) hasta F(n). Lo que iría calculando en cada iteración sería la suma de los dos números anteriores, inicializados en 1 (F(0) = F(1) = 1) siendo actualizados posteriormente con los nuevos valores calculados. El código *Java* de la Figura 4.3 muestra una implementación bajo dicho enfoque.

```
public static long linearFibonacci(long n) {
    long result;
    long last;
    long previousLast;

    if (n == 0) {
          return 0;
    }

    result = 1;

    if (n > 1) {
          last = 1;
          previousLast = 1;

          for (int i = 2; i <= n; i++) {
                result = last + previousLast;
                previousLast = last;
                last = result;
          }
    }

    return result;
}
```

Figura 4.3 Implementación lineal en *Java* de la serie de Fibonacci.

Haciendo una comparación de ambas implementaciones, se tomó el tiempo de ejecución para ambas, ejecutándo el algoritmo para los números Fibonacci 10, 20, 30, 40 y 50. Los resultados se muestran en la Tabla 4.1.

Tabla 4.1 Tiempos resultantes de la ejecución de las implementaciones de la serie de Fibonacci.

Valor de n	**Número Fibonacci resultante**	**Tiempo del algoritmo recursivo en milisegundos**	**Tiempo del algoritmo lineal en milisegundos.**
10	89	0	0
20	10946	0	0
30	1346269	47	0
40	165580141	6265	0
50	20365011074	769844	0

Confirme aumenta el número Fibonacci a calcular, el tiempo del algoritmo recursivo crece exponencialmente debido a la cantidad de operaciones redundantes, mientras que el tiempo del algoritmo lineal se mantiene. Este enfoque es uno de los fundamentos que sustentan a la técnica de la Programación Dinámica. Reescribiendo el algoritmo recursivo en uno no recursivo que vaya almacenando los valores calculados de los sub problemas en una tabla será más eficiente que calcular cada iteración los valores nuevamente (Weiss, 2007).

Problemas de Optimización

La optimización es un problema fundamental del ser humano que está siempre presente en todas sus actividades. Se pretende conocer la solución óptima a cada instancia de un determinado problema, ya sea minimizando los recursos o maximizando las ganancias. La Programación Dinámica es un enfoque de diseño de algoritmos de alto nivel que permite abordar problemas de optimización llamados NP-duros, o NP-completos cuando se tratan sus correspondientes problemas de decisión. La técnica de Programación Dinámica es inclusive aplicable a problemas que aunque posean algoritmos eficientes su orden de complejidad es polinomial con exponentes mayores a 3, como es el caso de los problemas que abordaremos en este investigación.

Un problema de optimización aplicable a la técnica de la Programación Dinámica es el siguiente. Se tiene un mapa de una ciudad típica modelado mediante un grafo. Los nodos representan los sitios donde se ubican las casas de los empleados y los diversos lugares de estacionamiento de sus centros de trabajo, marcados como nodos finales en el esquema de nodos; asimismo los nodos intermedios representan las intersecciones de calles a través de las cuales transitarán eventualmente para que los empleados partiendo desde sus hogares lleguen a sus centros de trabajo. La ciudad está diseñada para que cada empleado use exactamente 7 cuadras, es decir, 6 intersecciones para llegar a algún estacionamiento, lo que hace que la distancia total sea independiente de la ruta que tome. A pesar de esto, los empleados experimentan ciertos atrasos de tiempo en sus rutas. Los atrasos son indicados en las intersecciones, en tiempo en minutos. Podría asumirse que el retraso en las intersecciones es por los tiempos de cambio de luces en los semáforos ubicados en tal punto

de intersección, debido al tamaño y condiciones de las avenidas y cantidad del flujo vehicular. La ciudad esta representada en la Figura 4.4, donde aparecen los puntos intermedios y los retardos en dichos puntos. El problema de optimización consiste en minimizar la función objetivo planteada como la suma total de retraso originado por los tiempos que al empleado le consume cuando permanece en las intersecciones (Bradley et. al., 1977).

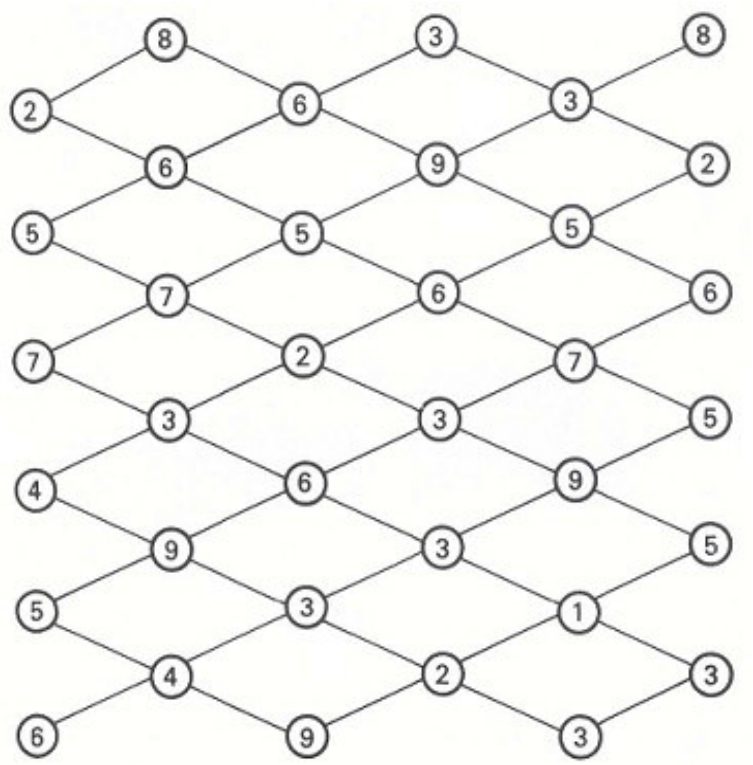


Figura 4.4 Los números en cada vértice representa el retardo en cada intersección.

En este caso no sólo es un vértice inicial y uno final, sino un conjunto de vértices iniciales y un conjunto de vértices finales, donde las posibilidades de movimiento son reducidas: al transitar a lo largo de la red, sólo existe la posibilidad de movimiento hacia la izquierda y hacia la derecha (excepto cuando se ha llegado al inicio y al final). De manera práctica, podemos asumir que las aristas que interconectan los puntos de retardo son dirigidas: la única posibilidad de movimiento es entonces, partiendo desde cualquiera de los 6 nodos iniciales, ir recorriendo los puntos de retardo o nodos hacia la derecha. Al momento de que los empleados terminan la jornada laboral y regresan hacia sus casas, se invierte la

dirección de las aristas, con lo que la posibilidad de movimiento es ahora hacia la izquierda.

Se agrupan en bloques los puntos de retardo de la ciudad, como lo muestra la Figura 4.5. Cada bloque de intersecciones es un nivel de avance a través de la red. Si un empleado atraviesa la ciudad para llegar a su trabajo, comenzaría tomando una intersección en el bloque de la extrema izquierda, avanzando gradualmente hasta llegar al bloque de la extrema derecha. De manera contraria sería cuando regresa a casa, la dirección del movimiento entre bloques sería de derecha a izquierda.

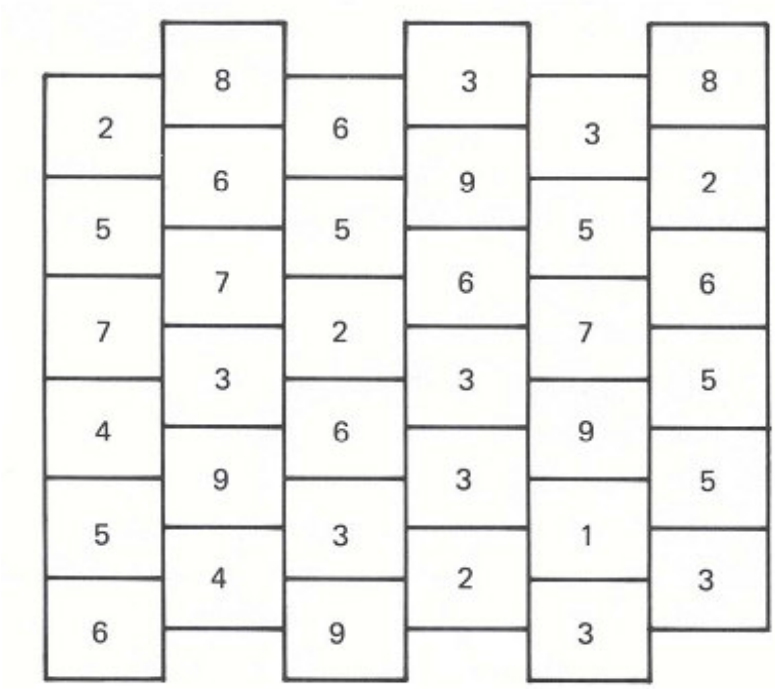


Figura 4.5 Modelado de la ciudad a manera de bloques. Cada caja corresponde a una intersección en la red. El retardo se indica en el número de cada caja.

Se resuelve ahora el problema enfocando la solución desde las casas de los empleados hacia los lugares de trabajo, resolviendo los caminos más cortos cuando el movimiento a través de la red es de izquierda a derecha. Es trivial observar que este problema es exactamente el mismo al de ir en sentido inverso, por lo que la solución a un problema aplicará al otro y viceversa. Se comienza tomando los bloques de derecha a

izquierda: esto brinda la posibilidad de ir almacenando los valores resultantes, evitando el recálculo completo cuando se les haga referencia en iteraciones más cercanas a los nodos inicio.

Dado que en la última columna el único movimiento por hacer es tomar una de las intersecciones, se asume cada valor como la ruta más corta desde el último bloque de intersecciones hasta el destino final, en este caso la fábrica. La Figura 4.6 ejemplifica esto al mostrar las intersecciones de la ciudad modelo a manera de grafo. Desde un nodo inicial abstracto, representando las casas de los empleados a manera de origen, se toma una intersección cualquiera, teniendo las aristas entre estos puntos ponderadas a 0. Posteriormente el retardo de cada intersección se propaga al resto de los grupos de vértices representativos de cada bloque de intersecciones, hasta llegar al último bloque, donde cada intersección propaga su retardo a manera de arista ponderada a un vértice final abstracto representando a la fábrica.

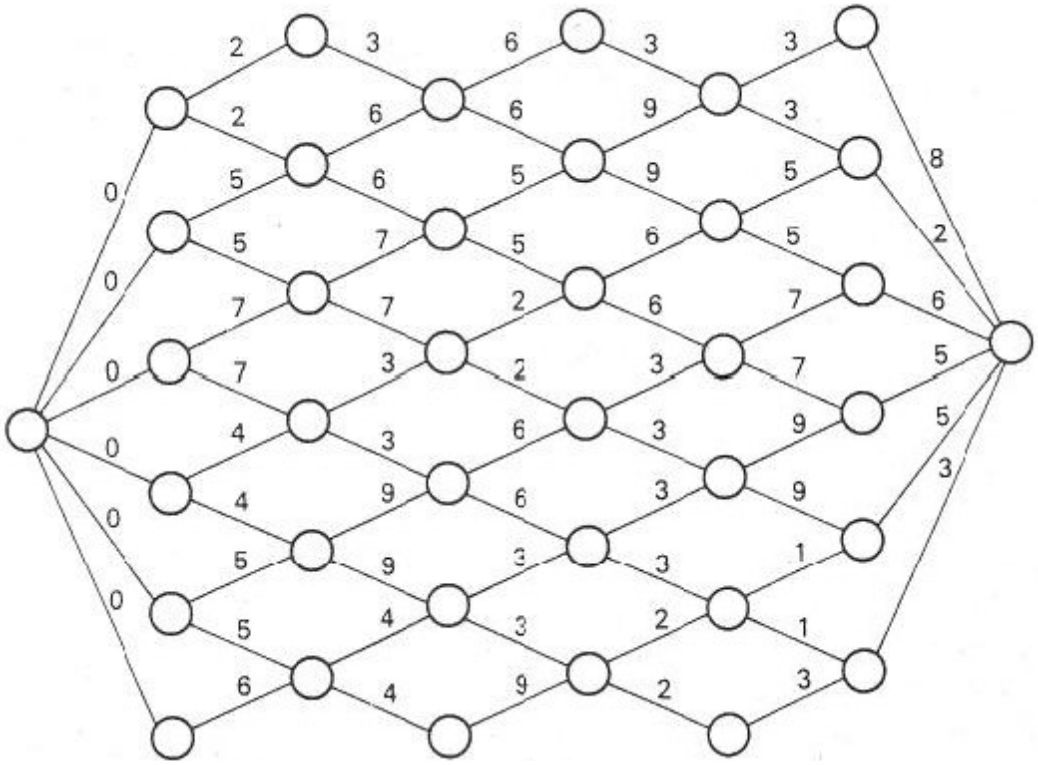


Figura 4.6 Modelado de la ciudad a manera de grafo. El retardo en cada vértice v es propagado a las aristas que son incidentes desde v a otros vertices de la red.

El procedimiento inicia tomando entonces la penúltima columna, puesto que para la última columna de intersecciones no existe elección alguna de valores, tomando entonces sus retardos como los valores óptimos. En la penúltima columna, para cada intersección se calcula su *movimiento óptimo* hacia la derecha. Por ejemplo, en la intersección extremo superior de dicha columna, con un retardo valuado en 3, existen dos opciones de movimiento: tomar la primer intersección de la última columna, con un retardo valuado en 8, o tomar la intersección inmediata siguiente, con un retardo valuado en 2. Si se suman los retardos entre la intersección actual y las dos posibilidades, se tiene que tomando la segunda intersección de la última columna hay un retraso total menor, valuado en 5, que en lugar de tomar la primer intersección, con un retraso total valuado en 11. Finalmente, para la intersección visitada se actualiza el valor del retardo con el retardo total óptimo, en el caso del ejemplo igual a 5 y almacenamos en dicha intersección una referencia a la intersección siguiente elegida. La Figura 4.7 muestra el procedimiento aplicado a todas las intersecciones de la penúltima columna.

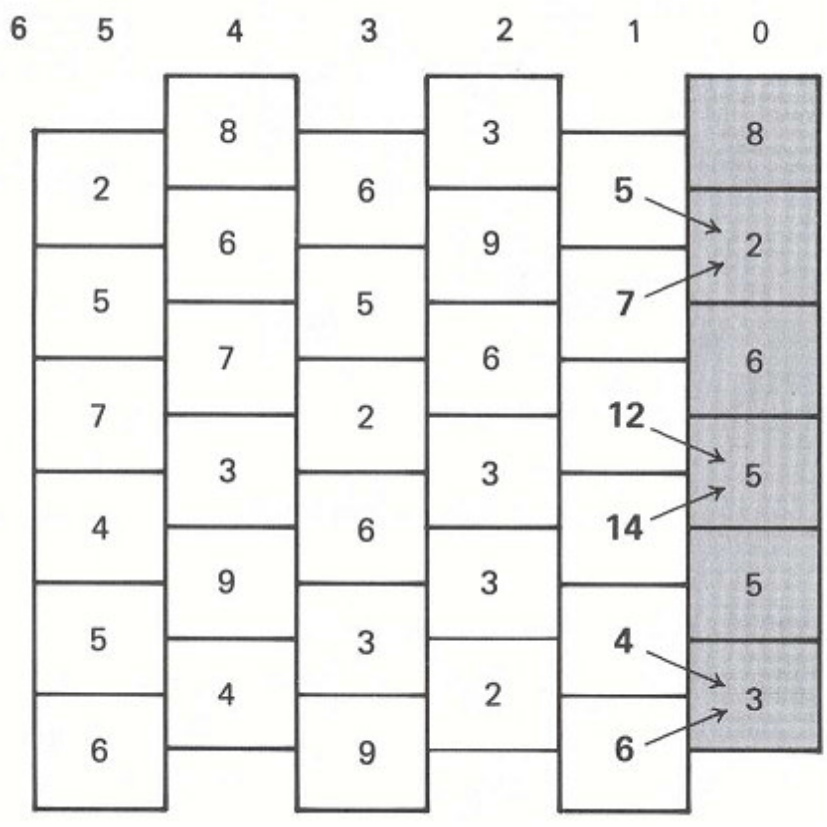
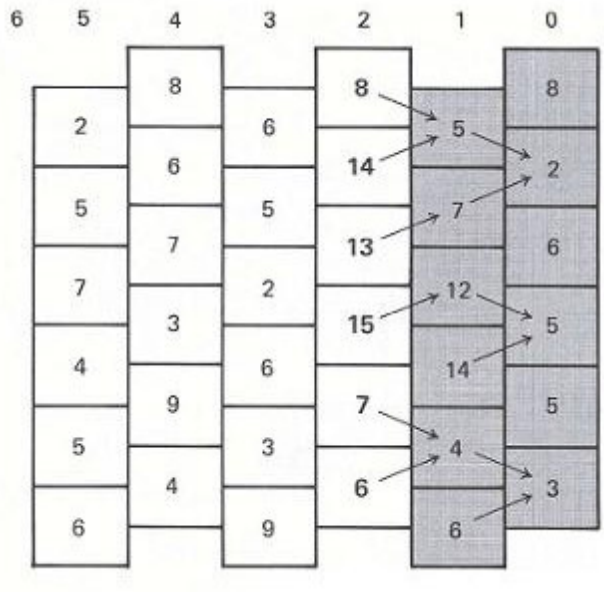
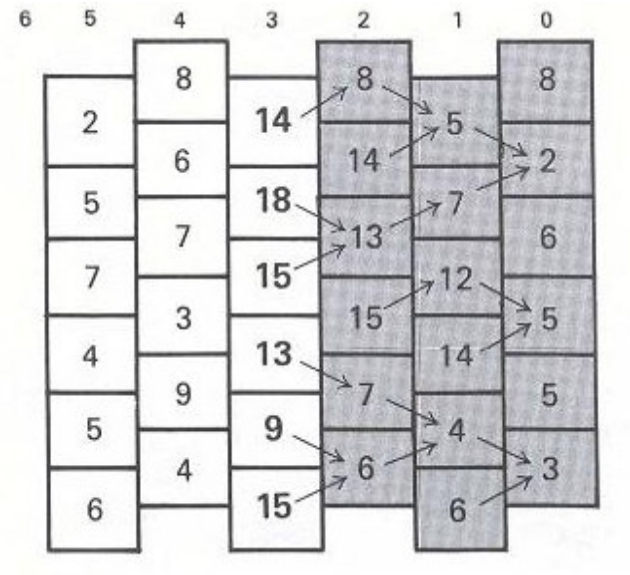


Figura 4.7 Inicia el procedimiento con la penúltima columna. En cada intersección se evalúa la dirección óptima, indicando con una flecha la elección y actualizando el retardo de la intersección con el retardo total.

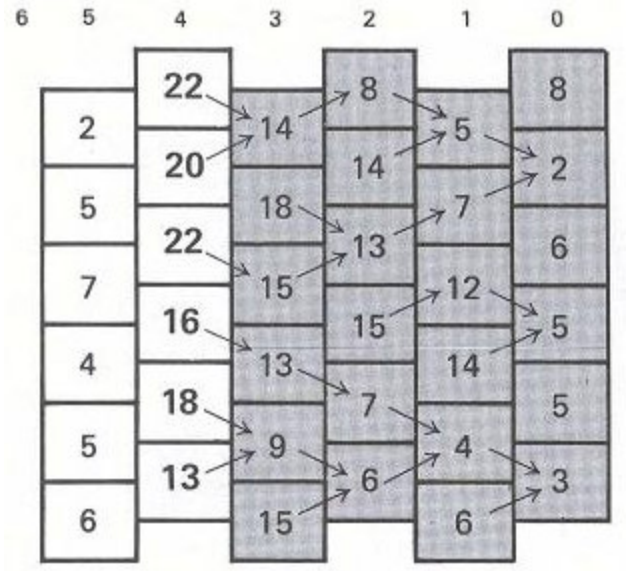
Para las siguientes iteraciones, el procedimiento es exactamente el mismo: verificar para cada intersección de la columna en curso su *movimiento óptimo* hacia la derecha. Con esto se consigue separar el problema principal de búsqueda de rutas óptimas en la red a un sub-problema o función objetivo evaluada en medio de una iteración: es por ello se menciona a la técnica de la Programación Dinámica como un procedimiento multi-etapa. La Figura 4.8 muestra el procedimiento completo: para la ante-penúltima columna, etiquetada con el número 2 se muestra en (a), la tercer y segunda columna y el resultado final son representados por (b), (c) y (d) respectivamente.



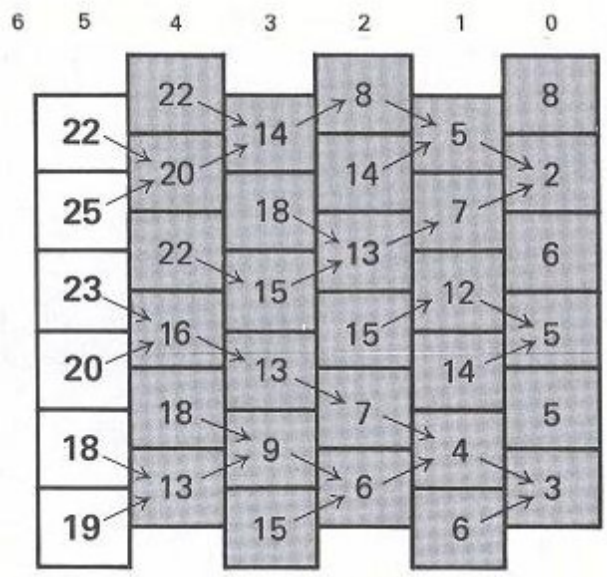
(a)


6 5 4 3 2 1 0
8 14 18 15 13 9 15
2 6 5 7 7 3 4 9 5 4 6
8 14 13 15 7 9 6
8 5 14 7 12 15 14 4 6
8 2 6 5 5 3

(b)


6 5 4 3 2 1 0
22 14 20 18 22 15 16 13 18 9 13 15
2 5 7 4 5 6
8 14 13 15 7 9 6
8 5 14 7 12 15 14 4 6
8 2 6 5 5 3

(c)



(d)

Figura 4.8. Procedimiento aplicado a la ante-penúltima columna (a), a la tercer columna (b), a la segunda columna (c) y a la primera, mostrando el resultado final (d).

Algoritmo Prototipo: Ruta más corta en un Digrafo Acíclico

Denardo (2003) señala como problema prototipo de la Programación Dinámica el encontrar la ruta más corta (o la ruta crítica) a través de un digrafo ponderado, finito y acíclico. Bajo el enfoque de modelado de la Teoría de Grafos, un digrafo brinda la idea de movimiento entre dos vértices: teniendo un arista e = (a,b) se dice que existe la posibilidad de movimiento del vértice a al vértice b, pero no necesariamente viceversa. Se le llama ponderado si para cada arista existe un valor asociado a la misma. Se sabe que, mediante el algoritmo de etiquetado del capítulo anterior es posible no sólo determinar si el digrafo en cuestión es acíclico, sino que cada arista dirigida (i, j) tiene la propiedad de que la etiqueta de su vértice origen i será menor que la del vértice destino j. Bajo esta propiedad, siempre se tendrá al menos un vértice final f sin aristas salientes. Este vértice final f expande los requisitos de dicho grafo, agregando la necesidad de la existencia de una ruta desde todos los vértices del grafo hasta el vértice final: esto puede ser determinado mediante el algoritmo de *Búsqueda en Anchura*.

En el escenario donde se calculan todos los caminos más cortos entre cualesquier par de nodos, si queremos evitar tener valoraciones iguales a infinito en rutas calculadas, es necesario que el grafo en cuestión sea fuertemente conexo (en inglés, Strongly Connected Component), es decir, que para cualquier par de vértices a y b, exista un camino (walk) entre ambos.

Para evitar desviar en este punto la atención del enfoque de la Programación Dinámica, se presenta un grafo con todos los requisitos anteriores cumplidos a fin de mostrar el algoritmo prototipo señalado por Denardo. Teniendo el digrafo mostrado en la Figura 4.9, buscando por inspección se tiene que la ruta más corta desde el vértice inicial 1 hasta el vértice final 9 pasa por los vértices {1, 3, 4, 5, 7, 9}.

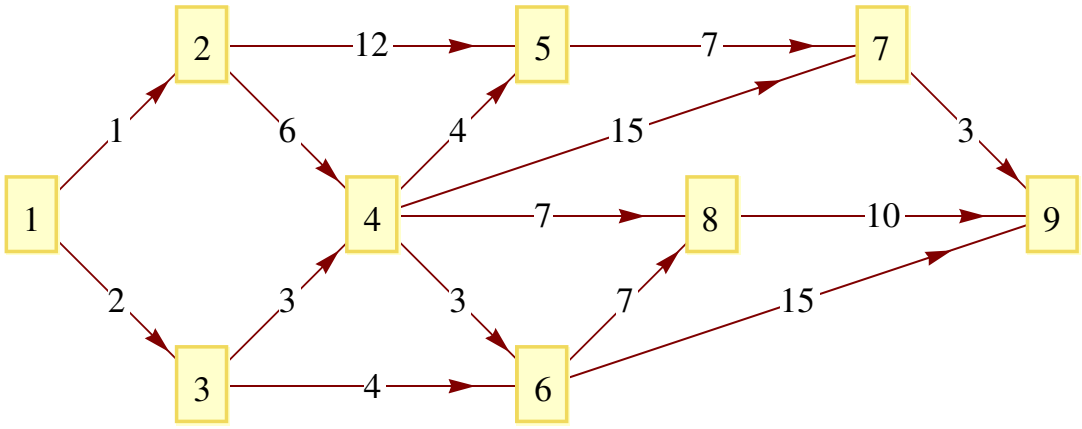


Figura 4.9 Digrafo ponderado finito y acíclico.

Se asume que el digrafo de la Figura 4.9 representa un sector de una ciudad, teniendo la ponderación de las aristas como la distancia en metros entre cada uno de los vértices, definiendo a d(i) como la distancia mínima de un vértice i al vértice final f, en este caso f = 9. Por ejemplo, la ruta más corta encontrada por inspección del vértice 1 al vértice 9, representada por d(1), es igual a 19, que es la sumatoria de las ponderaciones de las aristas {{1,3}, {3,4}, {4,5}, {5,7}, {7,9}}.

Estando en un vértice j intermedio en la ruta, se define a t(i, j) como la ponderación

del arista (i, j), de tal manera que t(i, j) + d(j) sería la distancia más corta desde el vértice i al vértice final f, pasando por el arista (i, j). Es decir, recorrer el arista (i, j) primero y posteriormente para ir de j a f utilizar la ruta más corta entre ambos vértices. Si se quisiera calcular la ruta más corta desde i al vértice final f que pase antes por un vértice j, habría que analizar la ponderación de cada una de las aristas salientes de i y posteriormente el valor de la ruta más corta de cada uno de los vértices adyacentes con i al vértice final f. Es decir, que para todo arista saliente de i (i, j), habría que obtener el valor mínimo de la suma entre t(i, j) y d(j). Entonces, podemos definir a d(i) por la ecuación 3, donde

$$di = \min_{j}\{tij + dj\},\ i \neq f \qquad (3)$$

d(i) es igual al valor mínimo entre la ponderación del arista (i, j) más el valor almacenado de la ruta más corta de j a f para todos los vértices j adyacentes con i, e i diferente a f.

Partiendo del digrafo a analizar y de un vértice final f, asignándole a d(f) el valor de 0, se comienza a iterar al resto de los vértices menores a f, comenzando con f – 1. En cada iteración, se toman todas las aristas (i, j) salientes del vértice en curso i, y a cada una se les aplica la ecuación 3, obteniendo el valor mínimo entre la suma de la ponderación del arista y el camino más corto ya calculado del vértice destino j de la arista hasta el vértice final f. Al terminar la iteración, se almacena el valor de d(j) del vértice en curso i. Con esto, se evita un recálculo innecesario de valores, almacenando los resultados de la ejecución de los sub-problemas recursivos en una tabla: con esto se está haciendo uso de una de las características principales de la técnica de la Programación Dinámica (Weiss, 2007). Al final, el algoritmo tendrá el árbol mínimo de expansión del digrafo analizado, donde existe un sólo camino desde cualquier vértice hasta el vértice final f y en cada vértice el valor de la ruta más corta hasta el vértice final f. A este árbol también de le llama *Árbol de rutas óptimas* (en inglés, *Shortest Path* Tree). Cabe señalar que si f < n siendo n el máximo valor de un vértice del digrafo a analizar, sólo se obtendrán las rutas y los valores de las mismas hasta ese punto intermedio de la red.

Tomando la implementación bajo *Java* presentada en el Anexo A.3, se trazó cada iteración, obteniendo los resultados mostrados en la Figura 4.10. El resultado final, tanto el Árbol Mínimo de Expansión del digrafo como la tabla de valores de las rutas más cortas de cada vértice hasta el vértice final f = 9, se muestran en la Figura 4.11.

```
Analizando vértice 8
 Calculando valor mínimo para f(8): Peso del arista (8, 9): 10 + f(9): 0 = 10
 Valor final para f(8): 10
Analizando vértice 7
 Calculando valor mínimo para f(7): Peso del arista (7, 9): 3 + f(9): 0 = 3
 Valor final para f(7): 3
Analizando vértice 6
 Calculando valor mínimo para f(6): Peso del arista (6, 8): 7 + f(8): 10 = 17
 Calculando valor mínimo para f(6): Peso del arista (6, 9): 15 + f(9): 0 = 15
 Valor final para f(6): 15
Analizando vértice 5
 Calculando valor mínimo para f(5): Peso del arista (5, 7): 7 + f(7): 3 = 10
 Valor final para f(5): 10
Analizando vértice 4
 Calculando valor mínimo para f(4): Peso del arista (4, 5): 4 + f(5): 10 = 14
 Calculando valor mínimo para f(4): Peso del arista (4, 6): 3 + f(6): 15 = 18
 Calculando valor mínimo para f(4): Peso del arista (4, 7): 15 + f(7): 3 = 18
 Calculando valor mínimo para f(4): Peso del arista (4, 8): 7 + f(8): 10 = 17
 Valor final para f(4): 14
Analizando vértice 3
 Calculando valor mínimo para f(3): Peso del arista (3, 4): 3 + f(4): 14 = 17
 Calculando valor mínimo para f(3): Peso del arista (3, 6): 4 + f(6): 15 = 19
 Valor final para f(3): 17
Analizando vértice 2
 Calculando valor mínimo para f(2): Peso del arista (2, 4): 6 + f(4): 14 = 20
 Calculando valor mínimo para f(2): Peso del arista (2, 5): 12 + f(5): 10 = 22
 Valor final para f(2): 20
Analizando vértice 1
 Calculando valor mínimo para f(1): Peso del arista (1, 2): 1 + f(2): 20 = 21
 Calculando valor mínimo para f(1): Peso del arista (1, 3): 2 + f(3): 17 = 19
 Valor final para f(1): 19
```

Figura 4.10 Iteraciones del algoritmo prototipo de Denardo. Se aplica la ecuación 3 en cada iteración, tomando el valor mínimo y almacenando dicho resultado evitando recálculos redundantes.

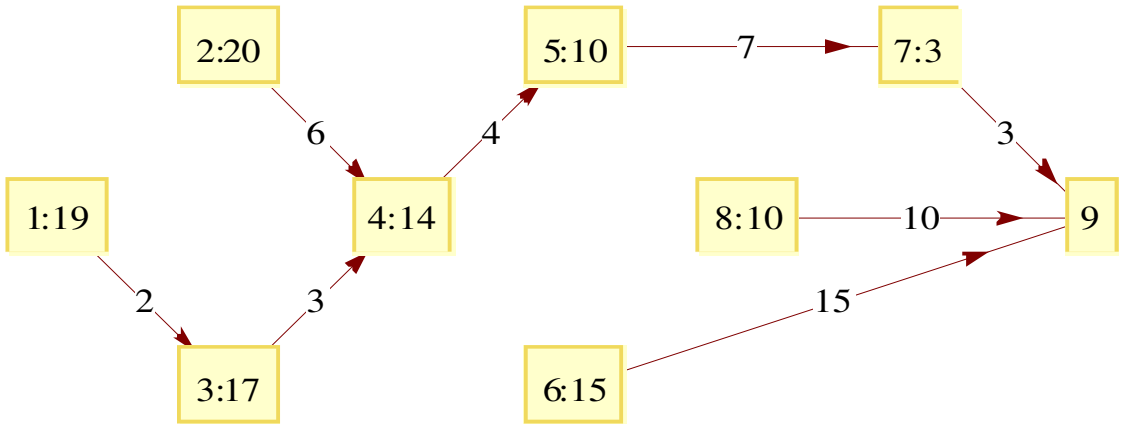


Figura 4.11 Árbol Mínimo de Expansión resultante. Hay un sólo camino de cada vértice al vértice 9, siendo estos caminos las rutas óptimas. En cada vértice se almacena el cómputo de la ruta óptima.

```
E350 = {{"2:20" -> "4:14", "6"}, {"1:19" -> "3:17", "2"}, {"3:17" -> "4:14", "3"},
     {"4:14" -> "5:10", "4"}, {"5:10" -> "7:3", "7"}, {"6:15" -> 9, "15"},
     {"7:3" -> 9, "3"},  {"8:10" -> 9, "10"}};
CR350 = {"1:19" -> {0, 0}, "2:20" -> {1, 1}, "3:17" -> {1, -1}, "4:14" -> {2, 0},
     "5:10" -> {3, 1}, "6:15" -> {3, -1}, "7:3" -> {5, 1}, "8:10" -> {4, 0},
     9 -> {6, 0}};
GraphPlot[E350, VertexLabeling -> True, DirectedEdges -> True,
     VertexCoordinateRules -> CR350]
```

En este escenario se separa el problema principal, que es la búsqueda de rutas óptimas en un digrafo finito, ponderado y acíclico hacia un vértice final f en la iteración de un problema más simple, que consiste en verificar el valor mínimo entre todas las aristas incidentes desde el vértice en turno i hacia otros vértices de la red, almacenando los valores calculados de las rutas más cortas desde el vértice en turno i al vértice final f, y trazando dichas rutas vía un árbol mínimo de expansión. En el escenario de la ciudad modelo separamos el problema de encontrar las rutas óptimas a través de la ciudad en la secuencia o iteración de la función objetivo del *movimiento óptimo* hacia un sentido dentro de la red, considerablemente más simple, optimizándolo almacenando los valores calculados en cada iteración (es por eso que estos procedimientos se realizan en sentido inverso) y además teniendo una referencia de las rutas elegidas.

La Figura 4.12 muestra el pseudo-código del algoritmo prototipo. Bajo el modelado

basado en el enfoque de la Programación Dinámica el problema principal se convierte en la iteración de la función objetivo que consiste en verificar el valor mínimo entre todas las aristas incidentes desde el vértice en turno i hacia otros vértices de la red. En cada iteración, se almacena el valor calculado de la ruta óptima desde el vértice en turno i hasta el vértice final f, agregando finalmente las aristas utilizadas a un digrafo T inicializado como (V, $\emptyset$), para así formar el árbol mínimo de expansión.

```
Input: Digrafo D = (V, E), Vertice final f

T = (V, EmptySet)
M = EmptyMap

M[f] = 0
v = f - 1

While v > 0 Do

   min = Infinity

   ForAll e in [OutgoingEdges of v]

      If (min >= weight(e) + M[target(e)])
         min = weight(e) + M[target(e)]
         edge = e
      End If

   End For

   M[v] = min
   AddEdge(T, edge)
   v = v - 1

End Do

Return {T, M}
```

Figura 4.12 Pseudo-código del algoritmo prototipo de la Programación Dinámica.

Se concluye entonces que separar un problema principal en una secuencia de problemas de decisión simples, almacenando los valores computados en cada iteración para evitar el recálculo de los mismos, es, en esencia, la definición de la técnica de la Programación Dinámica.

V. CASO DE ESTUDIO

Después de presentar un marco teórico acerca de los fundamentos de modelado de la Teoría de Grafos y la Programación Dinámica, se presenta la aplicación de los conceptos sobre un escenario con datos reales, modelando un sector de la red de tráfico vehicular de la ciudad de Querétaro mediante un digrafo ponderado. Aplicando algoritmos bajo el enfoque de la programación dinámica, se puede realizar un cálculo optimizado al analizar el grado de acceso de dicha red de tráfico vehicular. Estos algoritmos son el algoritmo de ruta más corta de *Dijkstra* y el algoritmo que calcula las rutas más cortas entre todos los pares de vértices dentro de un grafo, llamado *All Pairs Shortest Path* o *Algoritmo de Floyd-Warshall*.

Para el cómputo de resultados, se contó con 3 herramientas: la librería en *Mathematica* llamada *Combinatorica*, que brinda de algoritmos, funciones de modelado y graficación, complementada en este último aspecto por la función nativa de *Mathematica* conocida como *GraphPlot*; la segunda herramienta es la librería *Java* llamada *JGraphT*, que presenta un diseño orientado a objetos estructuras matemáticas derivadas del grafo: combinándola con la librería *JGraph*, es posible realizar la graficación de resultados vía paneles nativos de *Java*; la tercer herramienta es el framework de interconexión con *Java* que provee *Mathematica*: el *J/Link*.

Respecto a la instrucción *GraphPlot* y a la librería *Combinatorica*, estas herramientas fueron la base de la graficación. Al no ser herramientas de graficación de propósito general, sino herramientas diseñadas para mostrar estructuras matemáticas derivadas del grafo, es completamente natural el manejo de las mismas orientado matemáticamente al modelado de grafos, caso contrario de *JGraph*, que, siendo una herramienta de graficación de propósito general, requiere de un esfuerzo considerablemente mayor la presentación de resultados. *JGraphT*, librería si orientada completamente a grafos, provee un adaptador que permite graficar las estructuras de datos derivadas del grafo sobre *JGraph*. El adaptador esta implementado en la clase *JGraphModelAdapter*.

Mathematica también sirvió como una herramienta de verificación de resultados,

comparando los valores obtenidos en las ejecuciones de algoritmos con las implementaciones propias de los mismos, pudiendo detectar tempranamente códigos erróneos y oportunidades de mejora en los mismos. *Combinatorica* provee la implementación de los dos algoritmos base del caso de estudio: *Dijkstra* y *Floyd-Warshall*. No sólo los resultados de los algoritmos fueron comparados con las implementaciones propias: también el tiempo de ejecución fue un parámetro para decidir si la implementación se encontraba lista o no. Ejemplo de esto fue la implementación propia del algoritmo de *Floyd-Warshall*, detenida su ejecución en *Java* después de 25 minutos sin obtener resultado alguno, comparado con los menos de 3 minutos de ejecución en *Mathematica*, para posteriormente reducir el tiempo de ejecución a menos de 1 minuto.

La librería *JGraphT*, por su parte, tuvo un papel fundamental en la implementación propia de algoritmos: las estructuras de datos representativas de grafos y digrafos ponderados permitieron enfocar el desarrollo de la investigación en el modelado e implementación de algoritmos en lugar de la creación de un prototipo de trabajo orientado a objetos. *JGraphT*, al igual que *Combinatorica*, en su versión 0.8.1 provee tanto de la implementación del algoritmo de *Dijkstra* como del de *Floyd-Warshall*.

MathLink: Interconexión entre Mathematica y Java.

Uno de los experimentos realizados fue la interconexión entre dos paradigmas diferentes de programación: la programación orientada a objetos del lenguaje líder en este aspecto, como lo es *Java*, con la programación funcional basada en la definición de términos y formulaciones matemáticas, del lenguaje de alto nivel *Mathematica*. Esto fue logrado por una librería provista por *Mathematica* denominada *MathLink*, y su especialización hacia *Java* llamada *J/Link*.

En el caso del presente trabajo, el *J/Link* fue utilizado para tener el control del Kernel de *Mathematica* desde un hilo de ejecución en *Java*, brindando el acceso a la librería completa de funciones provista por *Mathematica*. *Java* es un lenguaje de un propósito mucho más general que *Mathematica*, con servicios orientados al manejo de bases de datos,

manejo de aplicaciones distribuidas, manejo de transacciones, programación orientada a aspectos, servicios Web, etc. Permitiendo que *Java* tenga acceso al poder de cómputo y la funcionalidad científica de *Mathematica* convierte al *J/Link* en la mejor herramienta para manejar el Kernel de *Mathematica* desde un entorno externo.

La librería que provee *Mathematica* es un archivo *JAR*, por lo que para ser utilizado en un entorno *Java* simplemente tiene que ser visible al compilador, es decir, las clases contenidas en la librería deben de estar en el *classpath* de la aplicación. Dentro del conjunto de clases provistas, la más esencial para la conexión es la interfaz *KernelLink*, que encapsula el funcionamiento del Kernel. Para obtener una instancia de la misma, hay que realizar una invocación al Kernel vía la clase de utilería *MathLinkFactory*. En el Anexo C.1 se presenta el código *Java* para generar una instancia del Kernel.

Para la comunicación con *Mathematica*, la interfaz *KernelLink* provee métodos para el envío de comandos y la recepción de resultados. Para el envío de comandos utilizamos 3 maneras en la presente investigación. La primera es bajo el método *evaluateToOutputForm*, que en la misma invocación obtiene la respuesta y la devuelve por medio de un *String*. De la misma manera funciona el método *evaluateToImage,* pero este devuelve un array de bytes representando a la imagen generada. El dump screen de la Figura 5.1 muestra una imagen generada y acoplada dentro de un panel nativo de Java. La tercer forma de comunicación utilizada es mediante el método *evaulate*. Este método, a diferencia de los otros dos, necesita una invocación posterior para obtener la respuesta del Kernel. Si posteriormente hacemos una invocación al método *discardAnswer,* la respuesta del Kernel es ignorada y el paquete desechado. En cambio, al invocar el método *waitForAnswer* la instancia de *KernelLink* toma el paquete respuesta y, mediante la sucesiva llamada a métodos *getter* nosotros podemos obtener el resultado final de la operación. La Figura 5.2 muestra un código donde se evalúa una suma, se mantiene el paquete respuesta por medio de la invocación a *waitForAnswer* y se obtiene la respuesta mediante la invocación al método *getInteger*.

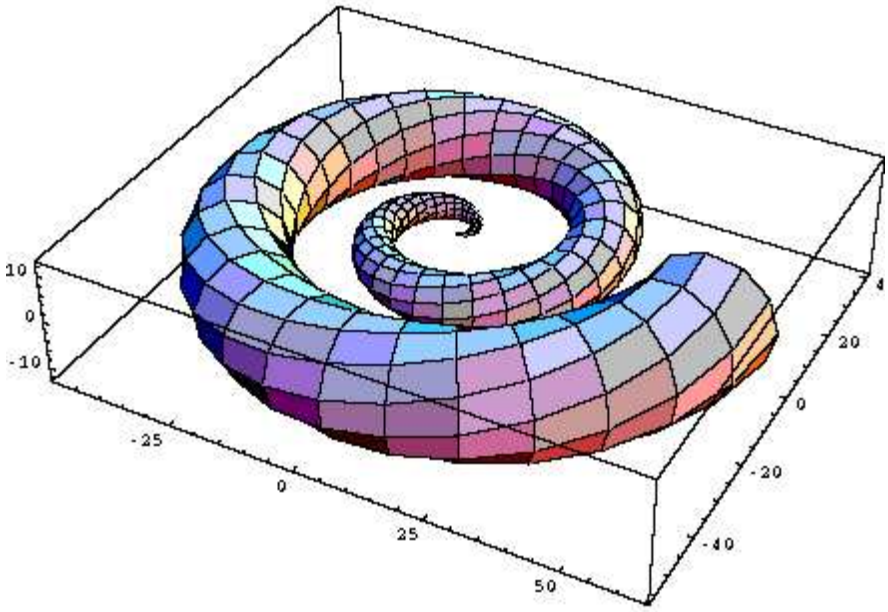


Figura 5.1 Imagen generada desde *Java* mediante el *KernelLink*.

```
package graphs.examples;

import graphs.utils.MathematicaUtils;
import com.wolfram.jlink.KernelLink;
import com.wolfram.jlink.MathLinkException;
import com.wolfram.jlink.MathLinkFactory;

public class JLinkTest {
      public static void main(String[] args) {
              KernelLink link;

              try {
                      link = MathematicaUtils.getKernelLink7Instance();
              } catch (MathLinkException e) {
                      System.out.println("Error al crear la instancia: " + e.getMessage());
                      return;
              }

              try {
                      link.discardAnswer();
                      link.evaluate("2+2");
                      link.waitForAnswer();
                      System.out.println("Resultado: " + link.getInteger());
              } catch(MathLinkException e) {
                      System.out.println("Excepción enviada al evaluar: " + e.getMessage());
              } finally {
                      link.close();
              }
      }
}
```

Figura 5.2 Código muestra de conexión con *Mathematica* vía *J/Link*.

Por último, al evaluar respuestas que vienen a manera de expresión matemática (como por ejemplo la matriz de resultados del algoritmo de *Floyd-Warshall* donde están expresadas las rutas más cortas entre todos los pares de nodos) son encapsuladas en la librería de *J/Link* por una clase denominada *Expr.* Para obtener una expresión como resultado a una evaluación, hacemos uso del método *getExpr* de la clase *KernelLink*. El objeto *Expr* resultante esta estructurado a manera de árbol: haciendo una doble iteración sobre la propiedad *args*, accesada vía el método *getArgs*, podemos leer valor a valor la matriz resultante del cómputo de las rutas más cortas entre todos los pares de nodos.

Modelado de la Ciudad de Querétaro

Se realizó el modelado de un sector de la red de tráfico vehicular de la ciudad de Querétaro, este sector se encuentra delimitado por las avenidas 5 de Febrero, Universidad, Corregidora y Constituyentes. El modelo consta de un digrafo ponderado, situando en cada vértice sus coordinadas geográficas obtenidas a través de un archivo GIS proporcionado por el IMT (Instituto Mexicano del Transporte). Las coordenadas geográficas usan el sistema de coordenadas geográficas UTM, por sus siglas en inglés Universal Transversal de Mercator.

El sistema de coordenadas UTM está basado en la proyección de Mercator, la cual es una proyección geográfica cilíndrica, representando la superficie esférica terrestre en un cilíndro. El sistema de coordenadas UTM, a diferencia de la proyección de Mercator, que es tangente al Ecuador, es tangente a un Meridiano y utiliza las coordenadas a nivel del mar de los puntos. La proyección geográfica funciona de la siguiente manera: imaginemos que metemos el globo terrestre en un cilindro y comenzamos a inflarlo hasta que ocupe el total del volumen del cilindro, imprimiendo el mapa en su interior. Al final, el cilindro se corta longitudinalmente, desplegando un mapa en este caso la proyección cilíndrica de Mercator.

El Anexo C.4 muestra el código para la construcción y graficación del digrafo representativo del sector seleccionado de la red de tráfico vehicular en *Mathematica*. Se muestran la construcción de las 736 aristas, con su respectiva ponderación, a través de las funciones *AddEdges* y *SetEdgeWeights.* Los vértices fueron numerados arbitrariamente por

números enteros, es en el momento de la graficación cuando se les es asignada su ubicación geográfica. La Figura 5.3 muestra el digrafo ponderado representativo del sector seleccionado de la red de tráfico vehicular de la ciudad de Querétaro.

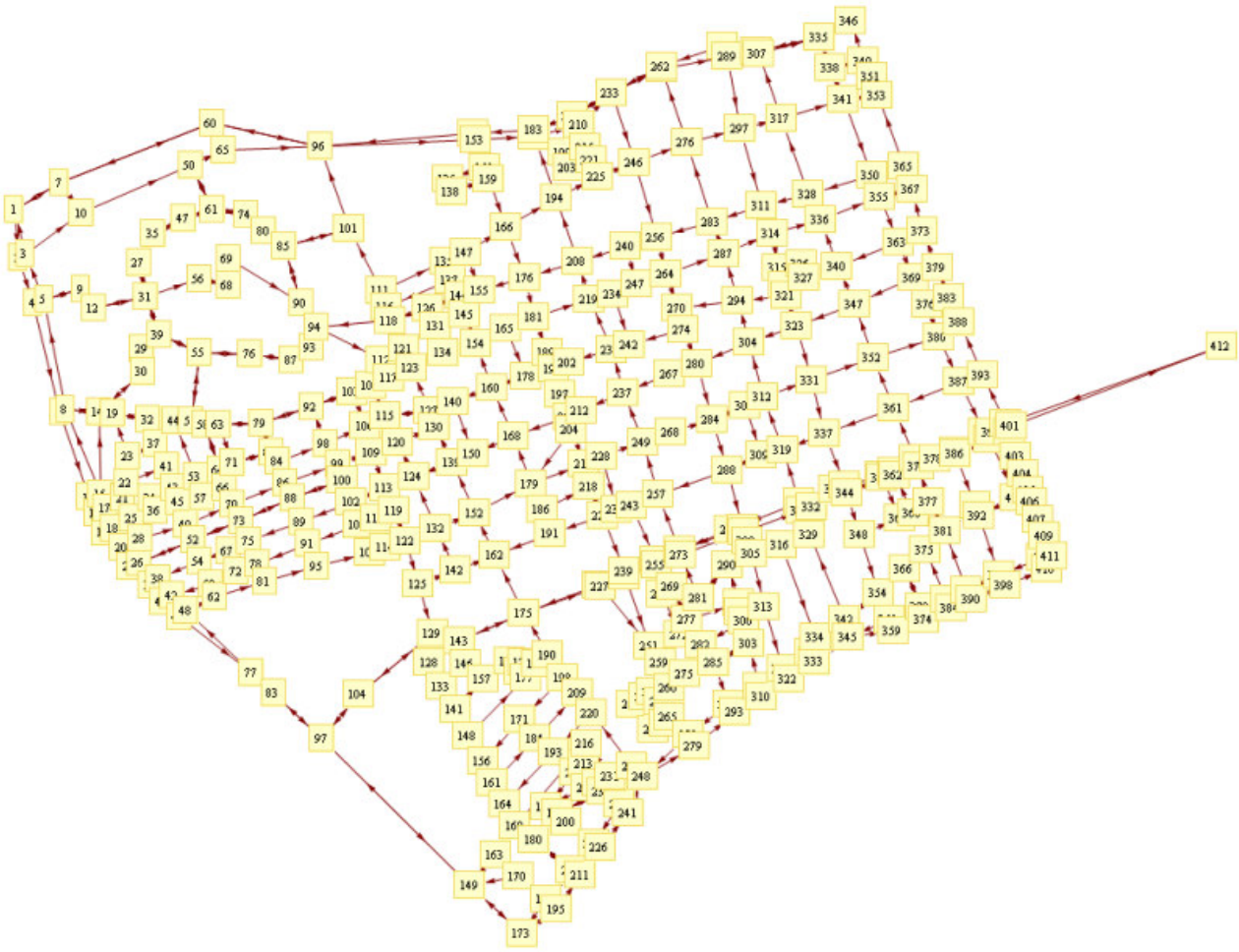


Figura 5.3 Digrafo ponderado representativo del sector de la red de tráfico vehicular de la ciudad de Querétaro delimitado por las avenidas 5 de Febrero, Constituyentes, Corregidora y Universidad.

Aplicación de los Algoritmos de Dijkstra y Floyd-Warshall

Jungnickel (2007) presenta a los algoritmos de Dijkstra y FloydWarshall como los algoritmos más eficientes para el cálculo de rutas óptimas dentro de una red. Ambos

algoritmos siguen el enfoque de la programación dinámica: presentan una función objetivo que es iterada sucesivamente hasta conseguir los resultados óptimos, para lograr esto en un tiempo eficiente se almacenan los valores hasta el momento computados dentro de una tabla.

El *Algoritmo de Dijkstra* es un algoritmo ideado por el científico holandés Edsger Dijkstra en 1959 que resuelve el problema de encontrar la ruta más corta desde un vértice inicial al resto de los vértices, siendo el algoritmo más popular y eficiente hasta la fecha (su orden de complejidad es de $O(|V|^2)$). Funciona sólo para aristas con ponderaciones no negativas, y puede ser adaptado para que encuentre la ruta óptima entre un par de vértices definidos. En el Anexo C.2 se presenta la implementación bajo *Java* del algoritmo. *Combinatorica* y *JGraphT*, por su parte, también presentan una implementación del algoritmo en sus respectivos enfoques de programación.

El Algoritmo de Dijkstra funciona de la siguiente manera: Se inicializan dos tablas, una para almacenar las distancias más cortas desde vértice inicial al resto de los vértices, y otra para almacenar la traza del árbol mínimo de expansión resultante. Dado que existirá una sóla ruta desde el vértice inicial al resto de los vértices, podemos almancenar en una tabla una relación entre cada vértice y un arista terminando en él: de esta manera empezando desde cualquier vértice podemos obtener la traza hasta el vértice inicial. Así mismo, inicializamos un conjunto de vértices V' igual al conjunto de vértices del grafo a analizar. Este conjunto nos servirá para mantener un histórico de los vértices ya visitados por las iteraciones. Se agrega a tabla de las distancias la distancia entre el vértice inicial y si mismo, o sea 0.

En cada iteración, se selecciona de la tabla de las distancias el vértice no visitado cuya distancia al vértice inicial sea la menor. Este vértice se marca como visitado, eliminandose de V', y se toman todas sus aristas salientes (en el caso de un grafo no dirigido, se toman todas las aristas para las que el vértice en cuestión es incidente). En este punto tenemos 2 vértices identificados dentro de la iteración: el vértice inicial s y el vértice visitado v, más el conjunto T de aristas salientes de v. Posteriormente se realiza una iteración sobre

los vértices u adyacentes desde v ubicados en el conjunto de aristas T: es en este punto donde se verifica la función objetivo, en este caso verificar si la distancia más corta desde el vértice s al vértice v más el valor de la arista ponderada (v, u) resulta en una distancia más corta que la ya almacenada desde el vértice s al vértice u. Es decir, si encontramos un camino más corto desde el vértice inicial s al vértice u, pasando antes por el vértice v, el camino desde s hasta u será actualizado en la tabla de rutas más cortas desde s hasta el resto de los vértices.

La consistencia del procedimiento recae en el punto de selección del vértice a visitar en cada iteración: seleccionando el vértice con el menor valor entre los vértices no visitados estaremos asegurandonos que la ruta más corta a dicho vértice ya fue calculada respecto a los vértices anteriores. Es por ello que este algoritmo no funciona para ponderaciones negativas en las aristas, dado que se asume que en iteraciones posteriores el agregar una arista al árbol de expansión resultante implica un aumento en el valor de la ruta más corta: un arista negativa en iteraciones posteriores decrementaría el valor de la ruta, lo que haría inválida la selección del vértice con el menor valor al inicio de alguna de las iteraciones pasadas.

Si se quisiera obtener la ruta óptima desde el vértice inicial s hasta un vértice específico e, simplemente tendríamos que detener las iteraciones cuando el valor del vértice visitado v fuera igual a e.

Por su parte, el *Algoritmo de Floyd-Warshall*, también conocido en inglés como *All Pairs Shortest Path Algorithm*, es un algoritmo ideado por los científicos estadounidenses Robert Floyd y Stephen Warshall en 1959. Basado en el enfoque de optimización de la técnica Programación Dinámica, encuentre la ruta más corta entre todos los pares de nodos de un grafo en una sóla ejecución en un orden de complejidad de $O(|V|^3)$.

El *Algoritmo de Floyd-Warshall* funciona de la siguiente manera. Sean los n vértices de G numerados del 1 al n. Sea también la función c(i,j,k) la longitud total de la ruta óptima de i a j que no tiene vértices intermedios mayores que k. Por definición tenemos los

siguientes escenarios, con los cuales se inicializa la tabla de rutas óptimas:

0, si a es igual a b,
∞, si a y b no se encuentran conectados por ninguna arista y
w(a,b), si existe una arista (a, b) se inicializa por su ponderación.

Por definición, también tenemos que c(i, j, n) es la ruta óptima entre los vértices i y j. ¿Cómo podemos determinar c(i, j, k) para cualquier k, k>0? Existen dos posibilidades: esta ruta podría o no tener a k como una vértice intermedio. Si no lo tiene entonces su longitud es c(i, j, k-1). Si lo tiene, entonces su longitud es c(i, k, k – 1) + c(k, j, k - 1). Así que finalmente, c(i, j, k) corresponde a aquella con el valor más pequeño. Formalmente, obtenemos la ecuación 4, donde

$$c(i, j, k) = \min \{ c(i, j, k-1), c(i, k, k-1) + c(k, j, k-1)\}, k > 0. \quad (4)$$

La recurrencia anterior formula la solución para un valor de k en términos de las soluciones para k-1. La obtención de soluciones para k-1 debería ser más fácil que la obtención de aquellas para k directamente (Weiss, 2007).

Para el caso de estudio en particular, se aplicó el algoritmo de Floyd-Warshall de 2 maneras diferentes: la primera fue ejecutando un programa Java que ejecutara la implementación del Anexo C.3 y la implementación brindada por la librería *JGraphT* sobre el digrafo ponderado representativo de la ciudad de Querétaro. Los tiempos de respuesta fueron 85734 milisegundos para la implementación de *JGraphT* y 55735 milisegundos para la implementación del Anexo C.3. Al utilizar tablas hash tanto para el acceso a la matriz de distancias óptimas como para el acceso a los elementos de grafo se logró un considerable decremento en el tiempo de ejecución.

La segunda aplicación del algoritmo fue mediante el *J/Link*. Se creó el grafo de acuerdo al código *Mathematica* del Anexo C.4 y se ejecutó la función *AllPairsShortestPath* de la librería *Combinatorica,* logrando un tiempo de 112473 milisegundos en la ejecución del

algoritmo y la comunicación con el Kernel. Para obtener la traza de las rutas óptimas fue necesaria una ejecución del algoritmo de *Dijkstra* provisto también por *Combinatorica* en la función *ShortestPath*.

Por último, se realizó el cómputo del grado de acceso del sector de la red de tráfico vehicular de la Ciudad de Querétaro analizado, con los siguientes resultados. La sumatoria de las distancias Euclidianas entre todos los pares de puntos fue de 184421698.29816588 metros, mientras que la sumatoria de las rutas óptimas fue de 278685684 metros. Esto nos resulta en un grado de acceso igual a 0.6617551919, lo que indica una eficiencia de la red del 66% respecto al grafo completo ideal que representa la interconexión de todos los pares de nodos en línea recta.

VI. CONCLUSIONES

Una vez construido un marco teórico guía para el modelado de estructuras y algoritmos basados en la Teoría de Grafos y la técnica de la Programación Dinámica, y aplicado al modelado de un sector de la red de tráfico vehicular de la ciudad de Querétaro utilizando los algoritmos de *Dijkstra* y *Floyd-Warshall*, se llegó a las conclusiones siguientes:

- Podemos hacer uso de un grafo completo $K_n = (V, E)$ en un grafo cuya cardinalidad de su conjunto de vértices $|V|$ es igual a n para modelar las distancias Euclidianas entre cualesquier par de vértices. Esto nos es útil al calcular el divisor de la fórmula del grado de acceso entre dos nodos cualesquiera dentro de un grafo, teniendo un cálculo con orden de complejidad $O((|V| * (|V|-1)) / 2) = O (|E|)$, dado que es tomada cada arista del grafo completo una vez.
- El algoritmo de *Búsqueda en Anchura*, o en inglés conocido por sus iniciales *BFS* (*Breadth First Search*) nos sirvió como herramienta para determinar si desde cualquier vértice en un digrafo es posible tener una ruta hacia un vértice destino f, invirtiendo la dirección de las aristas del digrafo y aplicando el algoritmo de *BFS* tomando dicho vértice f como raíz.
- En un algoritmo por definición recursivo, evitando el recálculo redundante almacenando el cálculo de valores intermedios en registros logra optimizar la solución a un problema aún cuando el algoritmo por definición tenga un orden de complejidad polinomial con exponentes mayores o iguales a 3, como es el caso del algoritmo de *Floyd-Warshall* usado en el presente trabajo. Dicho algoritmo nos permite un re-cálculo eficiente de las rutas más cortas con el objetivo de comprobar el grado de acceso entre los nodos de la red de tráfico vehicular analizada.
- Se modeló un sector de la red de tráfico vehicular de la ciudad de Querétaro, delimitado por las avenidas 5 de Febrero, Universidad, Constituyentes y Corregidora mediante un digrafo finito ponderado de 412 vértices y 736 aristas. La ponderación está dada por la distancia Euclidiana entre los vértices de una arista.
- Se ejecutó el algoritmo de *Floyd-Warshall* sobre el digrafo de la red de tráfico vehicular modelada, logrando un desempeño de menos de 1 minuto en el recálculo de las rutas

más cortas entre todos los pares de nodos. Esto bajo una implementación en *Java* optimizando el acceso a la matriz de almacenamiento de rutas ya calculadas entre los pares de vértices mediante tablas hash.

- Se logró realizar una conexión entre un lenguaje regido bajo el paradigma de la programación orientada a objetos (*Java*) con un lenguaje funcional basado en formulación matemática (*Mathematica*), mediante una librería basada en mensajes síncronos enviados desde un hilo de ejecución desde *Java* al core del lenguaje funcional, el *MathKernel* de *Mathematica*. Al combinar la alta escalabilidad entre librerías y el entorno multi-plataforma de *Java* y el lenguaje funcional orientado a un aspecto científico como lo es *Mathematica* nos permitió hacer uso de las ventajas de cada lenguaje.
- Se obtuvo el grado de acceso actual del sector de la red de tráfico vehicular de la Ciudad de Querétaro analizado, computado en 0.6617551919, es decir, es un 66% eficiente respecto al grafo completo ideal con todas las intersecciones conectadas entre si.

VII. REFERENCIAS BIBLIOGRÁFICAS

Bondy, A., Murty, U. S. R., *Graph Theory*, Estados Unidos: Springer, 2008.
Bradley, S. P., Hax, A. C., Magnati, T. L., *Applied Mathematical Programming.* Estados Unidos: Addison Wesley, 1977.
Dauben, J. W., *Georg Cantor: His Mathematics and Philosophy of the Infinite*, Boston: Harvard University Press, 1979.
Denardo, E. V., *Dynamic Programming, Models and Applications*, 1982 ed. New Jersey: Dover Publications, 2003.
Dym, C., *Principles of Mathematical Modeling*, 2da. ed. Estados Unidos: Academic Press, 2004.
Foulds, L. R., *Graph Theory Applications*, 2da. ed. Estados Unidos: Springer, 1995.
Fraleigh, J. B., *A First Course in Abstract Algebra*, 5ta. ed. Estados Unidos: Addison Wesley, 1994.
Gross, J. L., Yellen, J., *Handbook of Graph Theory.* Estados Unidos: CRC Press, Discrete Mathematics & Applications Series, 2003.
Jungnickel, D., *Graphs, Networks and Algorithms*, 3ra. ed. Berlin: Springer, 2007.
Weiss, M. A., *Data Structures and Algorithm Analysis in Java*, 2da ed. Estados Unidos: Addison Wesley, 2007.

ANEXOS

A. IMPLEMENTACION DE ALGORITMOS BAJO JAVA

1 Algoritmo de Búsqueda por Primero en Anchura (Breadth-First Search, BFS)

```
package commons.algorithms;

import java.util.HashMap;
import java.util.LinkedList;
import java.util.Map;
import java.util.Queue;

import org.apache.commons.collections.CollectionUtils;
import org.jgrapht.DirectedGraph;
import org.jgrapht.Graph;

public final class BFS {

	/**
	 * Implementación para un grafo no dirigido.
	 */
	public static <V, E> Map<V, Integer> breadthFirstSearch(Graph<V, E> graph, V start) {

		Map<V, Integer> breadthFirst;
		Queue<V> queue;
		V vertex;
		V source;
		V target;
		V adjacent;

		breadthFirst = new HashMap<V, Integer>();
		breadthFirst.put(start, 0);

		queue = new LinkedList<V>();
		queue.add(start);

		while (queue.size() > 0) {
			vertex = queue.poll();

			for (E edge : graph.edgesOf(vertex)) {
				source = graph.getEdgeSource(edge);
				target = graph.getEdgeTarget(edge);
				adjacent = vertex.equals(source) ? target : source;

				if (!breadthFirst.containsKey(adjacent)) {
					breadthFirst.put(adjacent, breadthFirst.get(vertex) + 1);
					queue.add(adjacent);
				}
			}
		}

		return breadthFirst;
	}

	/**
	 * Implementación para un digrafo.
	 */
	@SuppressWarnings("unchecked")
	public static <V, E> Map<V, Double> breadthFirstSearch(DirectedGraph<V, E> digraph,
		V start) {

		Map<V, Double> breadthFirst;
		Queue<V> queue;
		V vertex;
```

```
                V adjacent;
                V disjoined;
                Collection<?> disjoint;

                breadthFirst = new HashMap<V, Double>();
                breadthFirst.put(start, 0D);

                queue = new LinkedList<V>();
                queue.add(start);

                while (queue.size() > 0) {
                        vertex = queue.poll();

                        for (E edge : digraph.outgoingEdgesOf(vertex)) {
                                adjacent = digraph.getEdgeTarget(edge);

                                if (!breadthFirst.containsKey(adjacent)) {
                                        breadthFirst.put(adjacent, breadthFirst.get(vertex) + 1);
                                        queue.add(adjacent);
                                }
                        }
                }

                disjoint = CollectionUtils.disjunction(digraph.vertexSet(),
                        breadthFirst.keySet());

                for (Object object : disjoint) {
                        disjoined = (V) object;
                        breadthFirst.put(disjoined, Double.POSITIVE_INFINITY);
                }

                return breadthFirst;
        }
}
```

2 Algoritmo de etiquetado de Denardo

```
package commons.algorithms;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import graphs.structures.edge.DirectedWeightedEdge;
import graphs.structures.edge.factory.DirectedWeightedFactory;

import org.apache.commons.lang.builder.EqualsBuilder;
import org.apache.commons.lang.builder.HashCodeBuilder;

import org.jgrapht.DirectedGraph;
import org.jgrapht.graph.DefaultDirectedGraph;
import org.jgrapht.graph.DefaultDirectedWeightedGraph;


public final class AcyclicGraphs {
        /**
         * POJO que es utilizado para encapsular el etiquetado de un vértice.
         * @param <V> Clase del vértice.
         */
        public static class IntegerMap<V> {
                private Integer label;
                private V data;

                public IntegerMap(Integer label, V data) {
```

```
                this.label = label;
                this.data = data;
        }

        public IntegerMap() {

        }

        @Override
        public String toString() {
                return String.format("{%d: %s}", label, data.toString());
        }

        @Override
        public int hashCode() {
                HashCodeBuilder builder;

                builder = new HashCodeBuilder();
                builder.append(this.data);
                builder.append(this.label);

                return builder.toHashCode();
        }

        @Override
        public boolean equals(Object o) {
                EqualsBuilder builder;
                IntegerMap<?> other;

                if (this == o) {
                        return true;
                }

                if (!(o instanceof IntegerMap)) {
                        return false;
                }

                other = (IntegerMap<?>) o;
                builder = new EqualsBuilder();
                builder.append(this.data, other.data);
                builder.append(this.label, other.label);

                return builder.isEquals();
        }

        public Integer getLabel() {
                return label;
        }

        public void setLabel(Integer label) {
                this.label = label;
        }

        public V getData() {
                return data;
        }

        public void setData(V data) {
                this.data = data;
        }
}

/**
 * Función de etiquetado. Etiqueta los vértices del grafo, encapsulados
 * bajo la clase IntegerMap, de tal manera que para cualquier arista
 * dirigida (i, j) i < j.
```

```
         * @param <V> Clase del vértice.
         * @param graph Digrafo a etiquetar. Se hace uso de la clase
         * DirectedWeightedEdge para representar las aristas.
         * @return Digrafo etiquetado.
         */
        public static <V> DirectedGraph<IntegerMap<V>, DirectedWeightedEdge<IntegerMap<V>>>
                labelAcyclicGraph(DirectedGraph<V, DirectedWeightedEdge<V>> graph) {

                DefaultDirectedGraph<IntegerMap<V>, DirectedWeightedEdge<IntegerMap<V>>> labeled;
                DirectedWeightedFactory<IntegerMap<V>> factory;
                DirectedGraph<V, DirectedWeightedEdge<V>> cloned;
                List<V> vertexList;
                int counter;
                IntegerMap<V> labeledVertex;
                Map<V, IntegerMap<V>> mapping;


                cloned = new DefaultDirectedGraph<V, DirectedWeightedEdge<V>>(
                        graph.getEdgeFactory());

                for (V vextex : graph.vertexSet()) {
                        cloned.addVertex(vextex);
                }

                for (DirectedWeightedEdge<V> edge : graph.edgeSet()) {
                        cloned.addEdge(edge.getSource(), edge.getTarget());
                }

                counter = 0;
                vertexList = new ArrayList<V>();
                mapping = new HashMap<V, IntegerMap<V>>();
                factory = new DirectedWeightedFactory<IntegerMap<V>>();
                labeled = new DefaultDirectedWeightedGraph<IntegerMap<V>,
                        DirectedWeightedEdge<IntegerMap<V>>>(factory);

                while (true) {
                        vertexList.clear();

                        if (cloned.vertexSet().size() == 0) {
                                break;
                        }

                        for (V vertex : cloned.vertexSet()) {
                                if (cloned.incomingEdgesOf(vertex).size() == 0) {
                                        vertexList.add(vertex);
                                }
                        }

                        if (vertexList.size() == 0) {
                                throw new IllegalArgumentException("El grafo no es acíclico");
                        }

                        for (V vertex : vertexList) {
                                labeledVertex = new IntegerMap<V>(++counter, vertex);
                                labeled.addVertex(labeledVertex);
                                mapping.put(vertex, labeledVertex);
                        }

                        cloned.removeAllVertices(vertexList);
                }

                for (DirectedWeightedEdge<V> edge : graph.edgeSet()) {
                        labeled.addEdge(mapping.get(edge.getSource()),
                                mapping.get(edge.getTarget()));
                }

                return labeled;
        }
}
```

3 Algoritmo de la ruta más corta en un digrafo finito, ponderado y acíclico.

```
package commons.algorithms;

import java.util.HashMap;
import java.util.Map;
import java.util.Set;

import graphs.structures.edge.DirectedWeightedEdge;
import graphs.structures.edge.factory.DirectedWeightedFactory;

import org.jgrapht.graph.DefaultDirectedWeightedGraph;

/**
 * Implementación del algoritmo prototipo de la técnica de la Programación
 * Dinámica propuesto por Denardo. Este algoritmo encuentra la ruta más corta
 * dentro de un digrafo finito, acíclico y ponderado. El algoritmo es ejecutado
 * en el constructor.
 * @author Ernesto Ignacio Espinosa Chávez.
 */
public final class Denardo {

	private Map<Integer, Integer> distanceMap;
	private DefaultDirectedWeightedGraph<Integer, DirectedWeightedEdge<Integer>> tree;

	/**
	 * Obtiene el minimum spanning tree con las rutas más cortas desde cualquier
	 * vértice hasta el vértice final finalVertex.
	 * @param graph Grafo a analizar.
	 * @param finalVertex Vértice final f. Destino de los vértices menores a f
	 * a calcular la ruta óptima.
	 */
	public Denardo(DefaultDirectedWeightedGraph<Integer,
			DirectedWeightedEdge<Integer>> graph, Integer finalVertex) {

		Set<DirectedWeightedEdge<Integer>> outgoing;
		DirectedWeightedEdge<Integer> edge;
		DirectedWeightedEdge<Integer> newEdge;
		Integer minimal;
		Integer weight;

		tree = new DefaultDirectedWeightedGraph<Integer, DirectedWeightedEdge<Integer>>(
				new DirectedWeightedFactory<Integer>());
		distanceMap = new HashMap<Integer, Integer>();

		for (Integer v : graph.vertexSet()) {
			tree.addVertex(v);
		}

		distanceMap.put(finalVertex, 0);

		for (int i = finalVertex - 1; i > 0; i--) {
			outgoing = graph.outgoingEdgesOf(i);
			minimal = Integer.MAX_VALUE;
			edge = null;

			System.out.println("Analizando vértice " + i);

			for (DirectedWeightedEdge<Integer> e : outgoing) {
				weight = (int) graph.getEdgeWeight(e);
				System.out.println(String.format("\tCalculando valor mínimo " +
					"para f(%d): Ponderación del arista (%d, %d): " +
					%d + f(%d): %d", i, i, e.getTarget(), weight,
					e.getTarget(), distanceMap.get(e.getTarget())));

				if (minimal >= weight + distanceMap.get(e.getTarget())) {
					minimal = weight + distanceMap.get(e.getTarget());
```

```
                                edge = e;
                        }
                    }
                    System.out.println(String.format("\tValor final para f(%d): %d", i,
                            minimal));
                    distanceMap.put(i, minimal);
                    newEdge = tree.addEdge(edge.getSource(), edge.getTarget());
                    tree.setEdgeWeight(newEdge, edge.getWeight());
                }
            }

            /**
             * Obtiene la propiedad distanceMap.
             * @return Regresa el valor de la propiedad distanceMap.
             */
            public Map<Integer, Integer> getDistanceMap() {
                    return distanceMap;
            }

            /**
             * Asigna la propiedad distanceMap.
             * @param distanceMap Asigna el valor en la propiedad distanceMap.
             */
            public void setDistanceMap(Map<Integer, Integer> distanceMap) {
                    this.distanceMap = distanceMap;
            }

            /**
             * Obtiene la propiedad tree.
             * @return Regresa el valor de la propiedad tree.
             */
            public DefaultDirectedWeightedGraph<Integer, DirectedWeightedEdge<Integer>> getTree() {
                    return tree;
            }

            /**
             * Asigna la propiedad tree.
             * @param tree Asigna el valor en la propiedad tree.
             */
            public void setTree(DefaultDirectedWeightedGraph<Integer,
                    DirectedWeightedEdge<Integer>> tree) {
                    this.tree = tree;
            }
    }
```

B. UTILERÍAS JAVA DESARROLLADAS

1 Implementación de una clase representativa de un arista dirigida y ponderada.

```
package graphs.structures.edge;

import org.apache.commons.lang.builder.EqualsBuilder;
import org.apache.commons.lang.builder.HashCodeBuilder;

public class DirectedWeightedEdge<V> {

	private Double weight;
	private V source;
	private V target;

	@Override
	public String toString() {
		return String.format("De %s \n a %s", source, target);
	}

	@Override
	public boolean equals(Object obj) {
		EqualsBuilder builder;
		DirectedWeightedEdge<?> other;

		if (obj == this) {
			return true;
		}

		if (!(obj instanceof DirectedWeightedEdge)) {
			return false;
		}

		other = (DirectedWeightedEdge<?>) obj;
		builder = new EqualsBuilder();
		builder.append(this.weight, other.weight);
		builder.append(this.source, other.source);
		builder.append(this.target, other.target);

		return builder.isEquals();
	}

	@Override
	public int hashCode() {
		HashCodeBuilder builder;

		builder = new HashCodeBuilder();
		builder.append(this.source);
		builder.append(this.target);
		builder.append(this.weight);

		return builder.toHashCode();
	}

	public Double getWeight() {
		return weight;
	}

	public void setWeight(Double weight) {
		this.weight = weight;
	}

	public V getSource() {
		return source;
	}

	public void setSource(V source) {
		this.source = source;
```

```
	}

	public V getTarget() {
		return target;
	}

	public void setTarget(V target) {
		this.target = target;
	}
}
```

2 Implementación de un constructor de aristas dirigidas y ponderadas.

```
package graphs.structures.edge.factory;

import graphs.structures.edge.DirectedWeightedEdge;

public class DirectedWeightedFactory<V>
	implements WeightedFactory<V, DirectedWeightedEdge<V>, Double> {

	@Override
	public DirectedWeightedEdge<V> createEdge(V source, V target) {
		DirectedWeightedEdge<V> edge;

		edge = new DirectedWeightedEdge<V>();
		edge.setSource(source);
		edge.setTarget(target);

		return edge;
	}

	@Override
	public DirectedWeightedEdge<V> createEdge(V source, V target,
		Double weight) {
		DirectedWeightedEdge<V> edge;

		edge = this.createEdge(source, target);
		edge.setWeight(weight);

		return edge;
	}

}
```

3 Interfaz para un objeto Matriz.

```
package graphs.structures.matrix;

import java.util.List;

/**
 * Interfaz para una matriz de datos.
 *
 * @author Ernesto Ignacio Espinosa Chávez.
 *
 * @param <L> Clase de la etiqueta para las columnas y renglones.
 * @param <T> Clase de los elementos de la matriz.
 */
public interface Matrix<L, T> {

	enum MatrixElement {
		ROW, COLUMN;
	}
```

```
        /**
         * Obtiene el elemento indicado por las coordenadas x y y.
         * @param x Coordenada x.
         * @param y Coordenada y.
         * @throws IllegalArgumentException Si los índices recibidos como parámetros
         * no existen o no son válidos.
         * @return El elemento indicado.
         */
        T getAt(L x, L y) throws IllegalArgumentException;

        /**
         * Obtiene el elemento indicado (ROW o COLUMN) de la posición i.
         * @param i Posición del elemento.
         * @param element Renglón (ROW) o columna (COLUMN).
         * @throws IllegalArgumentException Si los índices recibidos como parámetros
         * no existen o no son válidos.
         * @return Lista que contiene el elemento indicado.
         */
        List<T> get(L i, MatrixElement element) throws IllegalArgumentException;

        /**
         * Coloca el elemento t en la posición i,j.
         * @param i Renglón de la matriz.
         * @param j Columna de la matriz.
         * @param t Elemento a colocar.
         * @throws IllegalArgumentException Si los índices recibidos como parámetros
         * no existen o no son válidos.
         */
        void put(L i, L j, T t) throws IllegalArgumentException;

        /**
         * @param i Renglón de la matriz.
         * @param j Columna de la matriz.
         * @param t Elemento a colocar.
         * @param b True si se coloca tanto en i,j como en j,i. Falso lo contrario.
         * @throws IllegalArgumentException Si los índices recibidos como parámetros
         * no existen o no son válidos.
         */
        void put(L i, L j, T t, boolean b) throws IllegalArgumentException;
}
```

4 Implementación de una matriz utilizando dos HashMap para el manejo de índices.

```
package graphs.structures.matrix;

import java.util.ArrayList;
import java.util.Collection;
import java.util.Collections;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

/**
 * Implementación de una matriz utilizando 2 HashMap para el manejo de índices.
 * @author Ernesto Ignacio Espinosa Chávez.
 */
public class MapMatrix<V, L> implements Matrix<V, L> {

        private HashMap<V, HashMap<V, L>> map;
        private Collection<V> labels;

        /**
         * Constructor. Inicializa los índices de la matriz indicados en el
         * parámetro labels.
         * @param labels Índices o etiquetas de la matriz.
         */
        public MapMatrix(Collection<V> labels) {
                this.labels = Collections.unmodifiableCollection(labels);
```

```
        map = new HashMap<V, HashMap<V, L>>();

        for (V label : this.labels) {
                map.put(label, new HashMap<V, L>());
        }
}

/** {@inheritDoc} */
@Override
public List<L> get(V i, MatrixElement element)
                throws IllegalArgumentException {

        List<L> elements;

        switch (element) {
                case ROW:
                        elements = new ArrayList<L>(this.map.get(i).values());
                        break;
                case COLUMN:
                        elements = new ArrayList<L>();

                        for (Map.Entry<V, HashMap<V, L>> entry : map.entrySet()) {
                                elements.add(entry.getValue().get(i));
                        }
                        break;
                default:
                        throw new IllegalArgumentException("El elemento requerido" +
                                        " es inválido.");
        }

        return null;
}

/** {@inheritDoc} */
@Override
public L getAt(V x, V y) throws IllegalArgumentException {
        return map.get(x).get(y);
}

/** {@inheritDoc} */
@Override
public void put(V i, V j, L t) throws IllegalArgumentException {
        map.get(i).put(j, t);
}

/** {@inheritDoc} */
@Override
public void put(V i, V j, L t, boolean b) throws IllegalArgumentException {
        map.get(i).put(j, t);
        if (b) {
                map.get(j).put(i, t);
        }
}

/** {@inheritDoc} */
@Override
public String toString() {
        StringBuilder builder;

        builder = new StringBuilder();
        builder.append(String.format("|%8s|", ""));

        for (V label : labels) {
                builder.append(String.format("|%8s|", label));
        }

        builder.append("\n");

        for (V label1 : labels) {
                builder.append(String.format("|%8s|", label1 + ">>"));
```

```
                        for (V label2 : labels) {
                                builder.append(String.format("|%8s|", this.getAt(label1,label2)));
                        }

                        builder.append("\n");
                }

                return builder.toString();
        }
}
```

C. IMPLEMENTACIONES PARA EL CASO DE ESTUDIO

1 Generador de instancias del Kernel de Mathematica

```
package graphs.utils;

import com.wolfram.jlink.KernelLink;
import com.wolfram.jlink.MathLinkException;
import com.wolfram.jlink.MathLinkFactory;

/**
 * Utilerías para la comunicación con Mathematica.
 * @author Ernesto Ignacio Espinosa Chávez.
 */
public final class MathematicaUtils {

        private static boolean System7Set = false;

        private static final String KERNEL_SOURCE =
                "C:\\\\Archivos de programa\\\\Wolfram " +
                "Research\\\\Mathematica\\\\7.0\\\\MathKernel.exe";

        private static final String LINK_DIR =
                "C:/Archivos de programa/Wolfram " +
                "Research/Mathematica/7.0/AddOns/JLink";

        public static KernelLink getKernelLink7Instance() throws MathLinkException {
                KernelLink link;
                String[] args;

                args = new String[] { "-linkmode", "launch", "-linkname", KERNEL_SOURCE };

                if (!System7Set) {
                        System.setProperty("com.wolfram.jlink.libdir", LINK_DIR);
                        System7Set = true;
                }

                link = MathLinkFactory.createKernelLink(args);
                link.discardAnswer();

                return link;
        }
}
```

2 Implementación del Algoritmo de Dijkstra

```
package commons.algorithms;

import java.util.HashMap;
import java.util.HashSet;
import java.util.Map;
import java.util.Set;

import org.jgrapht.Graphs;
import org.jgrapht.graph.DefaultDirectedWeightedGraph;
import org.jgrapht.graph.DefaultWeightedEdge;

/**
 * Implementación del algoritmo de Dijkstra. El algoritmo se ejecuta en el
 * constructor.
 * @author Ernesto Ignacio Espinosa Chávez.
 */
public final class Dijkstra<V, E> {
  public Dijkstra(DefaultDirectedWeightedGraph<V, E> graph, V start) {
        this(graph, start, null);
```

```
}

public Dijkstra(DefaultDirectedWeightedGraph<V, E> graph,
                V start, V end) {

        Set<V> vertices;
        V current;
        Set<E> edgesOf;
        V target;
        Double last;
        Double calculated;
        E newEdge;

        this.start = start;
        distances = new HashMap<V, Double>();
        tree = new DefaultDirectedWeightedGraph<V, E>(graph.getEdgeFactory());
        spanningTree = new HashMap<V, E>();

        for (V v : graph.vertexSet()) {
                tree.addVertex(v);
                distances.put(v, Double.POSITIVE_INFINITY);
                spanningTree.put(v, null);
        }

        distances.put(start, 0D);
        vertices = new HashSet<V>();
        vertices.addAll(graph.vertexSet());

        while (!vertices.isEmpty()) {
                current = getMinimal(distances, vertices);


                if (current.equals(end)) {
                        break;
                }


                vertices.remove(current);
                edgesOf = graph.outgoingEdgesOf(current);

                for (E edge : edgesOf) {
                        target = Graphs.getOppositeVertex(graph, edge, current);

                        last = distances.get(target);
                        calculated = distances.get(current) + graph.getEdgeWeight(edge);

                        if (calculated < last) {
                                distances.put(target, calculated);
                                newEdge = tree.addEdge(graph.getEdgeSource(edge),
                                                graph.getEdgeTarget(edge));
                                if (spanningTree.get(target) != null) {
                                        tree.removeEdge(spanningTree.get(target));
                                }
                                spanningTree.put(target, newEdge);
                        }
                }
        }
}

/**
 * Determina el siguiente vértice a utilizar como nodo intermedio. Decide
 * por el vértice no visitado aún con la distancia más corta en el mapa
 * de distancias.
 * @param <V> Clase del vértice.
 * @param distances Mapa de distancias más cortas.
 * @param unvisited Conjunto de vértices por visitar.
 * @return Vértice con las características indicadas.
 */
private static <V> V getMinimal(Map<V, Double> distances, Set<V> unvisited) {
        Double minimal;
        V vertex;
```

```
            minimal = Double.POSITIVE_INFINITY;
            vertex = null;

            for (V v : unvisited) {
                    if (distances.get(v) < minimal) {
                            minimal = distances.get(v);
                            vertex = v;
                    }
            }

            return vertex;
    }

    private DefaultDirectedWeightedGraph<V, E> tree;
    private Map<V, Double> distances;
    private V start;
    private Map<V, E> spanningTree;

    public Double getDistanceTo(V vertex) {
            return distances.get(vertex);
    }

    public Map<V, E> getSpanningTree() {
            return spanningTree;
    }

    public void setSpanningTree(Map<V, E> spanningTree) {
            this.spanningTree = spanningTree;
    }

    public V getStart() {
            return this.start;
    }

    public void setStart(V start) {
            this.start = start;
    }

    public DefaultDirectedWeightedGraph<V, E> getTree() {
            return tree;
    }

    public void setTree(DefaultDirectedWeightedGraph<V, E> tree) {
            this.tree = tree;
    }

    public Map<V, Double> getDistances() {
            return distances;
    }

    public void setDistances(Map<V, Double> distances) {
            this.distances = distances;
    }
}
```

3 Implementación del Algoritmo de Floyd-Warshall

```
package commons.algorithms;

import graphs.structures.matrix.MapMatrix;
import graphs.structures.matrix.Matrix;

import org.jgrapht.graph.DefaultDirectedWeightedGraph;
import org.jgrapht.graph.DefaultWeightedEdge;

/**
 * Implementación del algoritmo de Floyd-Warshall conocido como All Pairs
 * Shortest Path.
```

```
 * @author Ernesto Ignacio Espinosa Chávez.
 */
public final class FloydWarshall {

  public static <V, E> Matrix<V, Double> allPairsShortestPath(
                DefaultDirectedWeightedGraph<V, E> graph) {

        Matrix<V, Double> pathMatrix;
        E edge;
        Double value;
        Double last;
        Double calculated;

        pathMatrix = new MapMatrix<V, Double>(graph.vertexSet());

        for (V source : graph.vertexSet()) {
               for (V target : graph.vertexSet()) {
                      if (source.equals(target)) {
                             pathMatrix.put(source, target, 0D);
                      } else {
                             edge = graph.getEdge(source, target);
                             value = (edge != null) ? graph.getEdgeWeight(edge) :
                                    Double.POSITIVE_INFINITY;
                             pathMatrix.put(source, target, value);
                      }
               }
        }

        for (V k : graph.vertexSet()) {
               for (V i : graph.vertexSet()) {
                      for (V j : graph.vertexSet()) {
                             last = pathMatrix.getAt(i, j);
                             calculated = pathMatrix.getAt(i, k) + pathMatrix.getAt(k, j);
                             if (calculated < last) {
                                    pathMatrix.put(i, j, calculated);
                             }
                      }
               }
        }

        return pathMatrix;
    }
}
```

4 Construcción y graficación del Grafo representativo del sector seleccionado de la red de tráfico vehicular de la ciudad de Querétaro en Mathematica

```
Needs["Combinatorica`"];
g = EmptyGraph[412];
g = MakeDirected[g];

e = {{96, 60}, {96, 153}, {151, 96}, {183, 151}, {206, 183}, {206, 233}, {233, 206},
{233, 246}, {233, 262}, {233, 261}, {136, 138}, {158, 136}, {158, 159}, {199, 203},
{215, 199}, {215, 221}, {138, 159}, {159, 166}, {203, 221}, {221, 225}, {111, 135},
{111, 101}, {135, 147}, {135, 137}, {147, 166}, {147, 155}, {166, 194}, {166, 176},
{194, 225}, {194, 182}, {225, 246}, {246, 256}, {246, 276}, {116, 111}, {137, 116},
{118, 116}, {118, 94}, {126, 118}, {126, 131}, {144, 137}, {144, 126}, {155, 144},
{176, 155}, {176, 181}, {208, 194}, {208, 176}, {240, 208}, {240, 247}, {256, 240},
{256, 264}, {131, 145}, {131, 134}, {145, 144}, {123, 134}, {123, 121}, {123, 117},
{134, 154}, {154, 145}, {154, 165}, {165, 181}, {165, 178}, {181, 219}, {181, 189},
```

{219, 208}, {219, 234}, {234, 247}, {234, 242}, {247, 264}, {264, 287}, {264, 270},
{189, 192}, {127, 123}, {127, 115}, {140, 127}, {140, 150}, {160, 154}, {160, 140},
{178, 160}, {192, 178}, {192, 197}, {202, 192}, {232, 219}, {232, 202}, {242, 232},
{274, 242}, {274, 280}, {197, 201}, {139, 124}, {139, 130}, {150, 139}, {168, 160},
{168, 150}, {201, 168}, {201, 204}, {212, 201}, {212, 204}, {237, 232}, {237, 212},
{267, 237}, {267, 280}, {280, 267}, {280, 284}, {204, 179}, {204, 214}, {152, 139},
{152, 179}, {179, 168}, {179, 214}, {214, 228}, {214, 218}, {228, 249}, {228, 235},
{249, 237}, {249, 268}, {268, 284}, {284, 288}, {284, 301}, {186, 179}, {186, 218},
{218, 229}, {162, 152}, {191, 186}, {191, 162}, {229, 191}, {229, 239}, {235, 229},
{243, 235}, {243, 254}, {257, 249}, {257, 243}, {288, 257}, {288, 298}, {175, 162},
{175, 143}, {175, 227}, {239, 224}, {239, 238}, {254, 239}, {271, 257}, {271, 254},
{298, 271}, {298, 299}, {262, 233}, {286, 262}, {286, 289}, {306, 286}, {306, 335},
{335, 306}, {335, 346}, {335, 338}, {346, 335}, {338, 341}, {349, 346}, {349, 338},
{351, 349}, {276, 297}, {276, 261}, {297, 317}, {297, 311}, {317, 341}, {317, 307},
{341, 353}, {341, 350}, {353, 351}, {283, 256}, {283, 276}, {311, 283}, {311, 314},
{328, 317}, {328, 311}, {350, 328}, {350, 355}, {365, 353}, {365, 350}, {287, 283},
{287, 314}, {314, 336}, {314, 315}, {336, 328}, {336, 355}, {355, 367}, {355, 363},
{367, 365}, {315, 326}, {315, 320}, {326, 327}, {320, 321}, {327, 320}, {340, 336},
{340, 327}, {363, 340}, {363, 369}, {373, 367}, {373, 363}, {294, 287}, {294, 270},
{321, 294}, {321, 323}, {304, 280}, {304, 294}, {323, 304}, {323, 331}, {347, 340},
{347, 323}, {369, 347}, {369, 376}, {379, 373}, {379, 369}, {376, 380}, {383, 379},
{383, 376}, {301, 312}, {301, 309}, {312, 304}, {312, 331}, {331, 352}, {331, 337},
{352, 347}, {352, 380}, {380, 388}, {380, 387}, {388, 383}, {309, 288}, {319, 312},
{319, 309}, {337, 319}, {337, 343}, {361, 352}, {361, 337}, {387, 361}, {387, 394},
{393, 388}, {393, 387}, {324, 319}, {324, 292}, {343, 324}, {343, 344}, {370, 361},
{370, 360}, {394, 395}, {399, 393}, {399, 396}, {270, 274}, {1, 7}, {1, 2}, {7, 1},
{7, 60}, {7, 10}, {60, 96}, {60, 7}, {60, 65}, {3, 1}, {3, 10}, {10, 50}, {50, 65},
{50, 61}, {65, 96}, {61, 50}, {61, 47}, {61, 74}, {5, 3}, {5, 9}, {9, 5}, {9, 12},
{12, 9}, {12, 31}, {31, 12}, {31, 56}, {31, 39}, {31, 27}, {56, 68}, {69, 90}, {85, 90},
{85, 101}, {85, 80}, {68, 69}, {39, 31}, {39, 55}, {39, 29}, {90, 85}, {90, 94},
{55, 39}, {55, 51}, {55, 76}, {94, 90}, {94, 112}, {94, 93}, {112, 121}, {112, 117},
{8, 5}, {8, 14}, {14, 8}, {14, 19}, {14, 30}, {19, 14}, {19, 32}, {32, 19}, {32, 44},
{32, 37}, {44, 32}, {44, 51}, {51, 55}, {51, 44}, {51, 58}, {58, 51}, {58, 63}, {58, 64},
{63, 58}, {63, 79}, {63, 71}, {79, 63}, {79, 92}, {79, 82}, {92, 79}, {92, 103},
{103, 92}, {103, 108}, {103, 106}, {108, 103}, {108, 117}, {23, 19}, {23, 37}, {37, 41},
{15, 8}, {15, 14}, {22, 23}, {22, 41}, {41, 43}, {17, 15}, {21, 22}, {21, 17}, {34, 43},
{34, 36}, {43, 34}, {43, 53}, {43, 45}, {53, 44}, {53, 43}, {53, 64}, {64, 58}, {64, 53},
{64, 71}, {64, 66}, {71, 64}, {71, 82}, {82, 79}, {82, 71}, {82, 84}, {84, 82}, {84, 98},
{84, 86}, {98, 92}, {98, 106}, {106, 115}, {106, 109}, {115, 127}, {115, 108},
{115, 106}, {18, 17}, {18, 25}, {25, 21}, {25, 36}, {36, 45}, {45, 57}, {45, 49},
{57, 53}, {57, 66}, {66, 64}, {66, 70}, {20, 18}, {20, 28}, {20, 24}, {28, 25}, {28, 49},
{49, 70}, {49, 52}, {70, 66}, {70, 86}, {70, 73}, {86, 84}, {86, 99}, {86, 88}, {99, 98},

{99, 109}, {109, 120}, {109, 113}, {120, 115}, {120, 130}, {26, 20}, {26, 28}, {26, 52},
{52, 26}, {52, 73}, {52, 54}, {73, 70}, {73, 52}, {73, 88}, {73, 75}, {88, 86}, {88, 73},
{88, 100}, {88, 89}, {100, 99}, {100, 88}, {38, 26}, {54, 38}, {54, 59}, {67, 54},
{67, 72}, {75, 73}, {75, 67}, {75, 78}, {89, 88}, {89, 75}, {89, 91}, {102, 100},
{102, 89}, {113, 102}, {113, 119}, {124, 120}, {124, 113}, {42, 38}, {59, 42}, {59, 62},
{72, 59}, {78, 75}, {78, 72}, {78, 81}, {91, 89}, {91, 78}, {91, 95}, {105, 102},
{105, 91}, {110, 105}, {110, 114}, {119, 110}, {119, 122}, {48, 42}, {48, 62}, {81, 78},
{81, 95}, {95, 91}, {95, 107}, {107, 105}, {107, 114}, {114, 122}, {122, 132},
{122, 125}, {132, 152}, {132, 124}, {125, 142}, {125, 129}, {142, 162}, {142, 132},
{77, 48}, {77, 83}, {83, 77}, {83, 97}, {97, 83}, {97, 104}, {97, 149}, {104, 97},
{104, 129}, {129, 104}, {129, 143}, {129, 128}, {143, 175}, {143, 129}, {101, 96},
{101, 85}, {62, 81}, {27, 31}, {27, 35}, {35, 27}, {35, 47}, {47, 61}, {47, 35},
{74, 61}, {74, 80}, {80, 85}, {80, 74}, {29, 39}, {29, 30}, {76, 55}, {76, 87}, {87, 76},
{87, 93}, {93, 94}, {93, 87}, {30, 14}, {30, 29}, {128, 133}, {133, 146}, {133, 141},
{146, 143}, {141, 157}, {141, 148}, {157, 146}, {167, 157}, {167, 172}, {172, 174},
{172, 177}, {174, 167}, {148, 177}, {177, 185}, {185, 174}, {185, 190}, {190, 175},
{190, 185}, {156, 161}, {171, 156}, {171, 184}, {198, 190}, {198, 171}, {161, 184},
{184, 209}, {184, 193}, {209, 198}, {164, 161}, {193, 164}, {170, 149}, {170, 163},
{220, 209}, {220, 193}, {149, 97}, {149, 173}, {163, 149}, {169, 163}, {169, 180},
{216, 169}, {188, 169}, {207, 188}, {213, 207}, {217, 188}, {217, 207}, {223, 213},
{223, 217}, {223, 231}, {231, 216}, {231, 244}, {244, 220}, {196, 188}, {196, 230},
{230, 223}, {230, 196}, {180, 169}, {180, 200}, {180, 205}, {200, 196}, {200, 180},
{173, 149}, {173, 195}, {187, 170}, {187, 173}, {205, 180}, {205, 187}, {222, 200},
{222, 205}, {236, 230}, {236, 222}, {248, 244}, {248, 236}, {248, 279}, {224, 175},
{224, 227}, {260, 272}, {269, 260}, {251, 259}, {259, 272}, {259, 266}, {272, 259},
{272, 277}, {277, 269}, {277, 272}, {277, 281}, {277, 282}, {277, 295}, {281, 277},
{281, 290}, {281, 273}, {290, 281}, {290, 302}, {245, 253}, {250, 245}, {252, 250},
{258, 252}, {258, 266}, {253, 263}, {263, 265}, {265, 258}, {266, 258}, {266, 275},
{266, 278}, {275, 266}, {275, 285}, {282, 275}, {282, 285}, {295, 290}, {296, 295},
{296, 313}, {313, 296}, {313, 318}, {300, 282}, {300, 296}, {285, 275}, {285, 303},
{285, 291}, {303, 300}, {303, 285}, {278, 248}, {291, 278}, {308, 303}, {308, 291},
{318, 308}, {356, 339}, {360, 356}, {385, 370}, {305, 313}, {305, 316}, {316, 329},
{316, 334}, {329, 325}, {348, 364}, {348, 354}, {364, 368}, {368, 377}, {368, 362},
{377, 371}, {334, 330}, {342, 329}, {342, 334}, {354, 342}, {354, 358}, {366, 354},
{366, 372}, {375, 366}, {381, 377}, {381, 375}, {391, 381}, {392, 391}, {392, 397},
{402, 392}, {402, 408}, {330, 318}, {345, 342}, {345, 330}, {345, 359}, {358, 345},
{372, 366}, {372, 358}, {382, 375}, {382, 372}, {389, 381}, {389, 382}, {397, 389},
{408, 397}, {405, 402}, {405, 404}, {411, 408}, {411, 409}, {302, 305}, {409, 407},
{407, 406}, {406, 405}, {404, 403}, {403, 400}, {227, 251}, {227, 238}, {238, 251},
{238, 255}, {255, 273}, {273, 271}, {273, 299}, {299, 290}, {299, 302}, {299, 332},
{325, 324}, {325, 344}, {344, 348}, {344, 357}, {357, 364}, {357, 362}, {362, 378},
{371, 370}, {371, 386}, {386, 391}, {386, 395}, {395, 402}, {395, 401}, {400, 399},

```
{400, 401}, {412, 399}, {153, 158}, {153, 182}, {182, 210}, {210, 233}, {210, 215},
{261, 289}, {307, 335}, {289, 297}, {289, 307}, {2, 4}, {4, 6}, {6, 11}, {11, 13},
{13, 16}, {16, 20}, {24, 33}, {33, 40}, {40, 46}, {46, 77}, {195, 211}, {211, 226},
{226, 241}, {241, 248}, {279, 293}, {293, 310}, {310, 322}, {322, 333}, {333, 345},
{359, 374}, {374, 384}, {384, 390}, {390, 398}, {398, 410}, {410, 411}, {401, 412},
{339, 324}, {292, 271}, {378, 386}, {332, 344}, {130, 127}, {121, 118}, {117, 123},
{117, 108}, {396, 385}};

w = {288, 406, 400, 160, 105, 119, 119, 190, 147, 144, 35, 101, 31, 41, 59, 37, 102, 133,
62, 44, 185, 181, 56, 50, 124, 103, 151, 140, 121, 169, 103, 202, 145, 43, 182, 35, 192,
102, 48, 49, 91, 52, 122, 104, 174, 143, 131, 100, 84, 97, 80, 74, 49, 93, 53, 62, 87,
81, 87, 79, 134, 151, 96, 101, 64, 65, 137, 81, 160, 92, 46, 119, 114, 67, 145, 120, 105,
94, 73, 68, 42, 143, 117, 48, 144, 96, 61, 107, 102, 57, 137, 115, 152, 34, 37, 56, 115,
119, 130, 75, 75, 150, 174, 95, 143, 165, 131, 147, 49, 59, 113, 144, 137, 78, 111, 136,
94, 73, 136, 83, 121, 66, 155, 143, 155, 40, 40, 154, 142, 74, 191, 154, 168, 181, 211,
79, 12, 86, 154, 66, 174, 10, 147, 168, 31, 90, 166, 166, 92, 84, 92, 88, 110, 87, 44,
143, 203, 110, 207, 170, 179, 89, 209, 52, 148, 215, 137, 75, 209, 124, 172, 58, 196, 87,
87, 136, 130, 90, 73, 165, 85, 128, 63, 61, 47, 36, 29, 56, 128, 90, 170, 99, 111, 69,
129, 158, 126, 82, 146, 110, 130, 144, 106, 162, 166, 80, 101, 68, 88, 84, 57, 51, 135,
146, 133, 169, 144, 146, 179, 67, 128, 69, 96, 146, 58, 115, 143, 142, 192, 184, 146,
148, 62, 142, 193, 127, 22, 142, 64, 22, 139, 43, 57, 137, 124, 137, 425, 96, 288, 425,
77, 118, 182, 309, 94, 126, 251, 126, 79, 88, 127, 101, 101, 60, 60, 139, 139, 146, 102,
88, 78, 221, 153, 175, 66, 70, 102, 115, 57, 153, 74, 115, 179, 132, 74, 189, 53, 64, 52,
289, 87, 87, 37, 154, 37, 93, 93, 68, 65, 68, 44, 179, 44, 34, 34, 37, 124, 37, 110, 106,
110, 139, 78, 139, 113, 113, 50, 93, 50, 52, 119, 78, 65, 238, 213, 66, 114, 57, 39, 41,
52, 64, 33, 64, 63, 41, 154, 63, 65, 124, 65, 38, 48, 38, 104, 78, 104, 31, 31, 131, 57,
103, 115, 58, 69, 114, 88, 58, 53, 55, 58, 63, 63, 62, 61, 55, 64, 48, 46, 54, 52, 54,
56, 125, 132, 43, 46, 146, 46, 57, 148, 43, 62, 91, 72, 99, 74, 105, 61, 63, 154, 154,
132, 57, 46, 132, 146, 57, 43, 146, 139, 65, 43, 139, 64, 116, 67, 80, 56, 57, 59, 63,
65, 144, 61, 62, 141, 93, 63, 98, 81, 57, 102, 33, 76, 63, 55, 49, 61, 143, 61, 60, 140,
47, 77, 53, 84, 53, 82, 49, 141, 61, 141, 78, 44, 54, 85, 115, 109, 146, 102, 134, 109,
120, 234, 77, 77, 172, 172, 149, 542, 149, 249, 249, 73, 74, 181, 73, 227, 175, 136, 88,
84, 84, 88, 79, 88, 88, 63, 66, 63, 57, 62, 132, 109, 109, 60, 53, 60, 154, 62, 68, 85,
72, 61, 108, 74, 60, 82, 39, 25, 19, 35, 213, 44, 37, 37, 125, 37, 61, 145, 61, 71, 155,
160, 162, 60, 55, 60, 186, 129, 81, 61, 140, 542, 184, 105, 89, 61, 282, 95, 116, 31,
116, 49, 60, 30, 39, 105, 60, 174, 34, 131, 25, 131, 61, 96, 126, 33, 96, 184, 109, 93,
109, 126, 99, 89, 91, 54, 122, 39, 91, 156, 205, 8, 114, 33, 48, 88, 70, 88, 41, 94, 41,
63, 74, 141, 63, 115, 122, 115, 79, 85, 38, 23, 32, 44, 33, 20, 44, 44, 55, 132, 55, 79,
85, 56, 102, 23, 64, 64, 168, 127, 25, 79, 104, 112, 61, 104, 162, 131, 112, 91, 76, 126,
33, 106, 141, 79, 79, 256, 65, 112, 160, 37, 51, 110, 96, 39, 238, 89, 110, 74, 93, 108,
73, 86, 70, 102, 26, 163, 107, 169, 84, 46, 106, 118, 115, 108, 90, 140, 67, 164, 57, 93,
122, 36, 46, 24, 53, 34, 46, 48, 31, 49, 63, 202, 68, 194, 88, 67, 12, 175, 95, 25, 189,
```

```
21, 125, 111, 101, 115, 32, 113, 20, 102, 144, 97, 165, 66, 25, 17, 597, 75, 153, 120,
119, 60, 174, 164, 189, 81, 123, 290, 234, 45, 53, 60, 74, 51, 60, 237, 108, 87, 117, 99,
138, 80, 83, 81, 108, 86, 75, 60, 93, 116, 37, 584, 99, 143, 55, 97, 44, 82, 62, 52,
106};

g = SetEdgeWeights[AddEdges[g, e], w];

GraphPlot[g, DirectedEdges -> True, VertexLabeling -> True, VertexCoordinateRules ->
{1 -> {0, 1876}, 2 -> {10, 1752}, 3 -> {28, 1761}, 4 -> {48, 1635}, 5 -> {76, 1643}, 6 ->
{117, 1353}, 7 -> {118, 1947}, 8 -> {133, 1359}, 9 -> {173, 1672}, 10 -> {175, 1869}, 11
-> {194, 1131}, 12 -> {207, 1622}, 13 -> {213, 1090}, 14 -> {220, 1352}, 15 -> {225,
1139}, 16 -> {232, 1040}, 17 -> {238, 1102}, 18 -> {257, 1052}, 19 -> {257, 1349}, 20 ->
{280, 1003}, 21 -> {283, 1129}, 22 -> {291, 1170}, 23 -> {296, 1236}, 24 -> {299, 952},
25 -> {307, 1076}, 26 -> {325, 961}, 27 -> {325, 1738}, 28 -> {328, 1024}, 29 -> {332,
1518}, 30 -> {334, 1456}, 31 -> {343, 1651}, 32 -> {349, 1333}, 33 -> {356, 904}, 34 ->
{356, 1129}, 35 -> {362, 1814}, 36 -> {367, 1097}, 37 -> {367, 1270}, 38 -> {375, 921},
39 -> {377, 1554}, 40 -> {385, 861}, 41 -> {398, 1212}, 42 -> {412, 877}, 43 -> {414,
1157}, 44 -> {417, 1328}, 45 -> {427, 1118}, 46 -> {430, 821}, 47 -> {441, 1853}, 48 ->
{448, 837}, 49 -> {448, 1060}, 50 -> {459, 1992}, 51 -> {461, 1328}, 52 -> {467, 1021},
53 -> {472, 1184}, 54 -> {482, 966}, 55 -> {482, 1506}, 56 -> {482, 1698}, 57 -> {488,
1131}, 58 -> {495, 1320}, 59 -> {511, 905}, 60 -> {514, 2102}, 61 -> {516, 1879}, 62 ->
{522, 873}, 63 -> {532, 1320}, 64 -> {535, 1202}, 65 -> {545, 2031}, 66 -> {548, 1155},
67 -> {558, 992}, 68 -> {558, 1680}, 69 -> {558, 1750}, 70 -> {569, 1113}, 71 -> {569,
1220}, 72 -> {579, 940}, 73 -> {590, 1071}, 74 -> {603, 1865}, 75 -> {611, 1018}, 76 ->
{614, 1502}, 77 -> {618, 676}, 78 -> {632, 958}, 79 -> {642, 1323}, 80 -> {651, 1823}, 81
-> {653, 913}, 82 -> {669, 1249}, 83 -> {675, 623}, 84 -> {687, 1223}, 85 -> {703, 1782},
86 -> {705, 1168}, 87 -> {723, 1488}, 88 -> {724, 1129}, 89 -> {747, 1068}, 90 -> {747,
1635}, 91 -> {766, 1010}, 92 -> {774, 1367}, 93 -> {774, 1520}, 94 -> {787, 1572}, 95 ->
{789, 953}, 96 -> {796, 2041}, 97 -> {801, 505}, 98 -> {810, 1270}, 99 -> {845, 1218},
100 -> {855, 1176}, 101 -> {873, 1827}, 102 -> {879, 1118}, 103 -> {881, 1404}, 104 ->
{897, 620}, 105 -> {897, 1060}, 106 -> {915, 1317}, 107 -> {926, 987}, 108 -> {929,
1420}, 109 -> {931, 1249}, 110 -> {942, 1076}, 111 -> {956, 1666}, 112 -> {957, 1488},
113 -> {965, 1155}, 114 -> {968, 1003}, 115 -> {968, 1341}, 116 -> {969, 1624}, 117 ->
{978, 1440}, 118 -> {979, 1590}, 119 -> {991, 1097}, 120 -> {999, 1273}, 121 -> {1015,
1516}, 122 -> {1020, 1018}, 123 -> {1035, 1466}, 124 -> {1041, 1184}, 125 -> {1053, 907},
126 -> {1077, 1621}, 127 -> {1082, 1356}, 128 -> {1083, 704}, 129 -> {1090, 778}, 130 ->
{1096, 1314}, 131 -> {1098, 1577}, 132 -> {1099, 1050}, 133 -> {1113, 642}, 134 -> {1119,
1506}, 135 -> {1124, 1744}, 136 -> {1129, 1957}, 137 -> {1137, 1695}, 138 -> {1140,
1923}, 139 -> {1142, 1222}, 140 -> {1145, 1380}, 141 -> {1150, 580}, 142 -> {1150,
941}, 143 -> {1161, 760}, 144 -> {1163, 1653}, 145 -> {1174, 1605}, 146 -> {1176, 700},
147 -> {1176, 1765}, 148 -> {1183, 513}, 149 -> {1187, 124}, 150 -> {1195, 1243}, 151 ->
{1195, 2075}, 152 -> {1200, 1091}, 153 -> {1202, 2058}, 154 -> {1203, 1529}, 155 ->
```

{1213, 1668}, 156 -> {1222, 446}, 157 -> {1222, 661}, 158 -> {1226, 1986}, 159 -> {1237, 1957}, 160 -> {1244, 1416}, 161 -> {1247, 390}, 162 -> {1252, 981}, 163 -> {1254, 205}, 164 -> {1277, 337}, 165 -> {1281, 1569}, 166 -> {1281, 1831}, 167 -> {1291, 707}, 168 -> {1300, 1290}, 169 -> {1305, 279}, 170 -> {1314, 150}, 171 -> {1319, 554}, 172 -> {1319, 679}, 173 -> {1324, 0}, 174 -> {1326, 704}, 175 -> {1328, 831}, 176 -> {1331, 1700}, 177 -> {1333, 665}, 178 -> {1334, 1445}, 179 -> {1347, 1167}, 180 -> {1354, 242}, 181 -> {1355, 1598}, 182 -> {1355, 2067}, 183 -> {1355, 2083}, 184 -> {1358, 506}, 185 -> {1363, 698}, 186 -> {1376, 1099}, 187 -> {1386, 90}, 188 -> {1386, 330}, 189 -> {1389, 1508}, 190 -> {1391, 723}, 191 -> {1397, 1036}, 192 -> {1405, 1464}, 193 -> {1407, 471}, 194 -> {1412, 1907}, 195 -> {1414, 63}, 196 -> {1416, 312}, 197 -> {1423, 1398}, 198 -> {1430, 663}, 199 -> {1431, 2025}, 200 -> {1439, 288}, 201 -> {1444, 1340}, 202 -> {1444, 1482}, 203 -> {1444, 1986}, 204 -> {1452, 1306}, 205 -> {1453, 164}, 206 -> {1454, 2120}, 207 -> {1462, 418}, 208 -> {1468, 1742}, 209 -> {1469, 624}, 210 -> {1472, 2096}, 211 -> {1476, 152}, 212 -> {1478, 1356}, 213 -> {1485, 439}, 214 -> {1486, 1217}, 215 -> {1489, 2038}, 216 -> {1490, 492}, 217 -> {1492, 379}, 218 -> {1499, 1159}, 219 -> {1499, 1645}, 220 -> {1504, 573}, 221 -> {1504, 2004}, 222 -> {1511, 235}, 223 -> {1520, 390}, 224 -> {1520, 903}, 225 -> {1520, 1962}, 226 -> {1527, 223}, 227 -> {1528, 900}, 228 -> {1531, 1238}, 229 -> {1533, 1083}, 230 -> {1534, 369}, 231 -> {1555, 409}, 232 -> {1557, 1514}, 233 -> {1557, 2180}, 234 -> {1562, 1658}, 235 -> {1570, 1099}, 236 -> {1578, 337}, 237 -> {1588, 1403}, 238 -> {1591, 927}, 239 -> {1591, 939}, 240 -> {1594, 1781}, 241 -> {1601, 314}, 242 -> {1604, 1527}, 243 -> {1609, 1109}, 244 -> {1610, 434}, 245 -> {1610, 594}, 246 -> {1617, 1999}, 247 -> {1622, 1684}, 248 -> {1638, 406}, 249 -> {1638, 1275}, 250 -> {1643, 614}, 251 -> {1657, 744}, 252 -> {1663, 626}, 253 -> {1666, 529}, 254 -> {1672, 968}, 255 -> {1675, 954}, 256 -> {1675, 1805}, 257 -> {1678, 1138}, 258 -> {1682, 600}, 259 -> {1684, 704}, 260 -> {1687, 880}, 261 -> {1687, 2242}, 262 -> {1687, 2249}, 263 -> {1694, 547}, 264 -> {1699, 1710}, 265 -> {1707, 563}, 266 -> {1707, 637}, 267 -> {1709, 1451}, 268 -> {1712, 1301}, 269 -> {1714, 899}, 270 -> {1729, 1622}, 271 -> {1733, 994}, 272 -> {1735, 776}, 273 -> {1736, 982}, 274 -> {1743, 1566}, 275 -> {1749, 674}, 276 -> {1753, 2050}, 277 -> {1754, 813}, 278 -> {1758, 515}, 279 -> {1773, 486}, 280 -> {1780, 1477}, 281 -> {1784, 869}, 282 -> {1791, 748}, 283 -> {1818, 1845}, 284 -> {1819, 1332}, 285 -> {1823, 702}, 286 -> {1847, 2302}, 287 -> {1850, 1764}, 288 -> {1859, 1201}, 289 -> {1859, 2273}, 290 -> {1860, 956}, 291 -> {1862, 596}, 292 -> {1866, 1047}, 293 -> {1880, 574}, 294 -> {1886, 1640}, 295 -> {1888, 857}, 296 -> {1892, 834}, 297 -> {1892, 2086}, 298 -> {1898, 1052}, 299 -> {1901, 1042}, 300 -> {1902, 811}, 301 -> {1905, 1370}, 302 -> {1910, 1018}, 303 -> {1915, 751}, 304 -> {1915, 1533}, 305 -> {1922, 986}, 306 -> {1936, 2288}, 307 -> {1940, 2282}, 308 -> {1941, 642}, 309 -> {1947, 1241}, 310 -> {1949, 615}, 311 -> {1949, 1887}, 312 -> {1952, 1391}, 313 -> {1955, 848}, 314 -> {1976, 1816}, 315 -> {1991, 1727}, 316 -> {1999, 1007}, 317 -> {1999, 2113}, 318 -> {2003, 686}, 319 -> {2004, 1254}, 320 -> {2007, 1682}, 321 -> {2012, 1653}, 322 -> {2019, 660}, 323 -> {2039, 1575}, 324 -> {2046, 1118}, 325 -> {2052, 1097}, 326 -> {2052, 1737}, 327 -> {2060, 1701}, 328 -> {2070, 1916}, 329 -> {2073, 1035}, 330 -> {2075, 730}, 331 -> {2078, 1436}, 332 -> {2079, 1107}, 333 -> {2086, 706}, 334 -> {2089, 767}, 335 -> {2099, 2323}, 336 -> {2102,

```
1850}, 337 -> {2112, 1296}, 338 -> {2128, 2244}, 339 -> {2139, 1152}, 340 -> {2146,
1729}, 341 -> {2162, 2162}, 342 -> {2165, 815}, 343 -> {2165, 1163}, 344 -> {2170, 1141},
345 -> {2174, 769}, 346 -> {2180, 2367}, 347 -> {2191, 1632}, 348 -> {2204, 1035}, 349 ->
{2214, 2262}, 350 -> {2235, 1966}, 351 -> {2235, 2223}, 352 -> {2238, 1493}, 353 ->
{2251, 2173}, 354 -> {2253, 882}, 355 -> {2256, 1911}, 356 -> {2260, 1188}, 357 -> {2262,
1183}, 358 -> {2281, 813}, 359 -> {2291, 785}, 360 -> {2292, 1199}, 361 -> {2293, 1362},
362 -> {2294, 1189}, 363 -> {2304, 1792}, 364 -> {2308, 1077}, 365 -> {2319, 1989}, 366 -
> {2320, 947}, 367 -> {2340, 1929}, 368 -> {2343, 1090}, 369 -> {2343, 1701}, 370 ->
{2348, 1231}, 371 -> {2356, 1212}, 372 -> {2364, 848}, 373 -> {2367, 1821}, 374 -> {2372,
816}, 375 -> {2375, 996}, 376 -> {2380, 1630}, 377 -> {2385, 1120}, 378 -> {2399, 1231},
379 -> {2406, 1727}, 380 -> {2409, 1546}, 381 -> {2426, 1044}, 382 -> {2429, 866}, 383 ->
{2435, 1648}, 384 -> {2442, 844}, 385 -> {2452, 1252}, 386 -> {2453, 1244}, 387 -> {2464,
1430}, 388 -> {2464, 1585}, 389 -> {2482, 889}, 390 -> {2497, 868}, 391 -> {2505, 1109},
392 -> {2514, 1084}, 393 -> {2524, 1449}, 394 -> {2529, 1299}, 395 -> {2543, 1281}, 396 -
> {2548, 1299}, 397 -> {2567, 929}, 398 -> {2583, 904}, 399 -> {2584, 1323}, 400 ->
{2594, 1300}, 401 -> {2600, 1316}, 402 -> {2611, 1130}, 403 -> {2611, 1239}, 404 ->
{2630, 1193}, 405 -> {2643, 1148}, 406 -> {2652, 1118}, 407 -> {2669, 1073}, 408 ->
{2680, 975}, 409 -> {2686, 1030}, 410 -> {2692, 945}, 411 -> {2704, 980}, 412 -> {3147,
1523}}]
```