ESCUELA POLITÉCNICA NACIONAL

ESCUELA DE POSGRADO EN INGENIERÍA Y CIENCIAS

RUTAS MÁS CORTAS AL INTERIOR DEL SISTEMA DE TRANSPORTE MASIVO DE PASAJEROS DE GUAYAQUIL

PROYECTO PREVIO A LA OBTENCIÓN DEL GRADO DE MAGÍSTER EN INVESTIGACIÓN OPERATIVA

MIGUEL A. FLORES SÁNCHEZ

DIRECTOR: DR. LUIS MIGUEL TORRES

JULIO, 2006

DECLARACIÓN

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

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

MIGUEL A. FLORES SÁNCHEZ

CERTIFICACIÓN

Certifico que el presente trabajo fue desarrollado por Miguel A. Flores Sánchez, bajo mi supervisión.

Dr. Luis Miguel Torres

DIRECTOR DE PROYECTO

Agradezco a todo el claustro de profesores del Departamento de Matemática y personal administrativo, especialmente al Dr. Luis Miguel Torres, por su paciencia y enseñanza, durante mi formación en la maestría y el desarrollo de mi proyecto de graduación,
así también agradesco la colaboración de La Muy Ilustre Municipalidad de Guayaquil por haber facilitado los datos operacionales de tres de las siete troncales que constituyen el sistema METROVIA.

DEDICATORIA

A Miguel A. Flores Calle y Nelly Sánchez Correa mis adorados padres y a mis queridas hermanas Nelly y Liliana Flores Sánchez.

Índice general

Índice de Tablas

Índice de figuras

Resumen

La Ilustre Municipalidad de Guayaquil, a través de la Dirección Municipal de Transporte y con el apoyo del PNUD, ha desarrollado el Plan de Transporte de la ciudad de Guayaquil, que tiene como objetivo fundamental mejorar el nivel de servicio del transporte público urbano. Uno de los aspectos fundamentales de este plan es la conformación de una red de corredores troncales de transporte automotor de elevada capacidad, operados en vías exclusivas y alimentados por buses convencionales integrados física, operativa y tarifariamente.

Dentro de este contexto, y buscando facilitar la adopción rápida del nuevo sistema de transporte por parte de la ciudadanía, esta tesis tiene como objetivo principal el desarrollo a nivel de prototipo, de una aplicación informática para la determinación de rutas más cortas. Diseñada para ser utilizada vía Internet, está aplicación ayudará a los usuarios a planificar mejor sus viajes y obtener un mayor provecho del sistema.

La determinación de rutas óptimas es formulada matemáticamente como un problema de caminos más cortos con restricciones de recursos $(SPPRC)$ y se propone un algoritmo de solución, basandose en una extensión del algoritmo de Dijkstra para la versión clásica de este problem que reside en un servidor Web. A través de un formulario de Internet, el usuario ingresa los siguientes datos de entrada para el prolema:

- Origen del desplazamiento
- Destino del desplazamiento
- Fecha de viaje
- Hora de salida

Estos datos son transmitidos al servidor, donde también reside una base de datos que contiene diversa información de la red de transporte; en particular

- Rutas de las líneas
- Planes de frecuencia de las líneas
- Paradas de las troncales y alimentadores

A partir de esta información, el módulo de cálculo determina la mejor ruta de transportación, incluyendo horarios de embarque y transbordos. Finalmente la respuesta se transmite de regreso al usuario a través de una página Web generada dinámicamente.

Este algoritmo ha sido implementado como una aplicación informática diseñada de tal forma que el ingreso de datos y la entrega de resultados se realicen a través de una interfaz apropiada para ser utilizado en Internet.

1. Conceptos preliminares

Existen muchos problemas prácticos concernientes a áreas tan diversas como la logística, el transporte, la gestión de inventarios, el diseño de redes de telecomunicaciones, la bioinformática, el control de la cadena productiva, entre otros, pueden ser formulados matemáticamente como problemas de optimización sobre grafos. En conjunto, ellos constituyen el objeto de estudio de la Optimización Combinatoria.

El lector interesado puede encontrar en [4] una introducción didáctica a este campo de la matemática aplicada, mientras que [13] contiene una descripción exhaustiva los problemas de investigación más comunes, así como los principales resultados. Por otro lado en [1] se presentan numerosas aplicaciones prácticas.

1.1. Grafos

En esta sección se introducen algunos conceptos básicos de la teoría de grafos, y se describe la notación utilizada en el resto de la tesis.

Definición 1.1 (Grafo)**.** Dado un conjunto no vacío y finito V, un *grafo no dirigido* sobre V es un par ordenado $G = (V, E)$ donde E es un multi-conjunto (es decir, un conjunto que admite elementos repetidos) de la forma $E \subseteq \{\{v, w\} : v, w \in V\}$. Los elementos de V se llaman *nodos* o *vértices* y los elementos de E son las *aristas* del grafo.

Ejemplo: Los conjuntos $V = \{a, b, c, d, e, f, g\}$ y $E = \{\{a, b\}, \{c, a\}, \{b, c\}, \{b, e\}, \{b, d\}, \{c, d\}, \{f, c\}, \{d, e\}, \{d, f\}, \{e, f\}, \{e, g\}, \{g, f\}\}$ definen un grafo. Este puede

representarse esquemáticamente como en la Figura 1.1

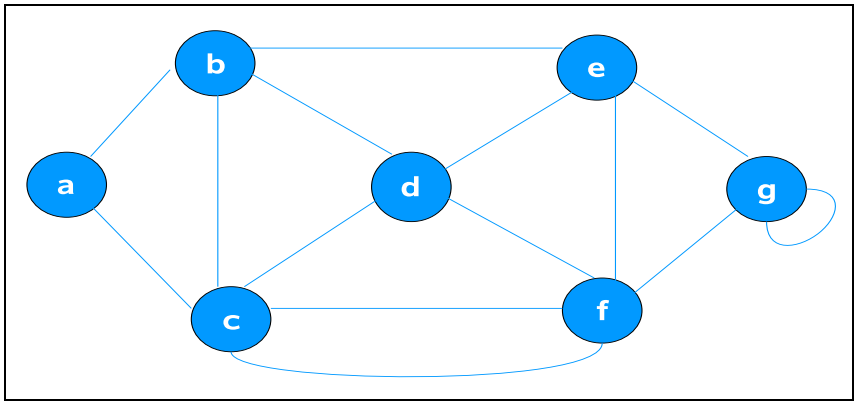


Figura 1.1: Un grafo no dirigido

Cada arista está formada por un par de nodos conocidos como sus *extremos*. En adelante, cuando no haya riesgo de confusión, escribiremos simplemente $e = ij$ para referirnos a la arista $\{v, w\}$, con $v, w \in V$. Decimos además que e es incidente a los nodos v, w.

Dos aristas e y f se denominan *paralelas* si tienen los mismos extremos; una arista de la forma $e = ii$, con $v \in V$ es un *lazo*. En el ejemplo de la Figura 1.1 hay dos aristas paralelas que contienen los nodos c y f; y gg es un lazo. Un grafo sin aristas paralelas y sin lazos se conoce como *grafo simple*.

Dado un grafo $G = (V, E)$, cualquier otro grafo $G' = (V', E')$ tal que $E' \subseteq E$ y $V' \subseteq V$ se conoce como un *subgrafo* de G. La Figura 1.2 muestra un ejemplo. Si además se cumple que $V' = V$, entonces G' se denomina un subgrafo generador. En estos casos emplearemos la notación $G[E']$ para referirnos a G' (Ver Figura 1.3).

Un subgrafo $G' = (V', E')$ se denomina *grafo inducido* por V' (y será denotado en adelante como $G[V']$) si E' contiene todas las aristas con ambos extremos en V', es decir, si $E' = \{\{v, w\} : v \in V', w \in V' \bigwedge ij \in E\}$ (Ver Figura 1.3).

Si $A \subseteq E$, el grafo $G \backslash A := (V, E \backslash A)$ se obtiene al eliminar de G las aristas que

G G'

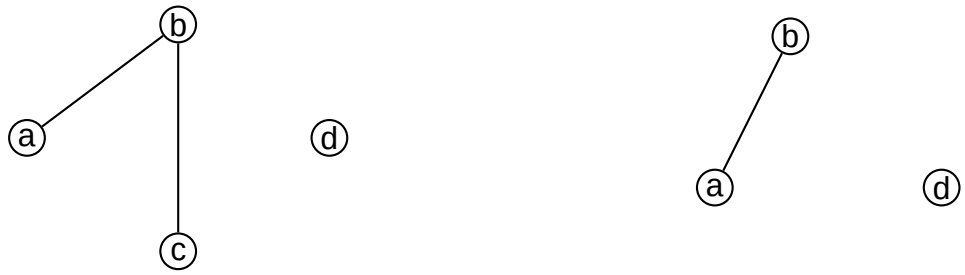
Figura 1.2: Un grafo $G(V, E)$ y un subgrafo $G'(V', E')$ de G. Notar que $V' \subseteq V$ y $E' \subseteq E$.

están en el conjunto A. En el caso particular de la eliminación de una sola arista $e \in E$, denotaremos por $G \backslash e$ al grafo resultante. Si $B \subseteq V$, el grafo $G \backslash B := G[V \backslash B]$ es el grafo inducido por $V \setminus B$, obtenido al eliminar de G todos los vértices de B, así como todas las aristas incidentes a ellos; usaremos $G \backslash v$ para referirnos al grafo $G \backslash \{v\}$, con $v \in V$.

Definición 1.2 (Digrafo)**.** Dado un conjunto V, un *grafo dirigido o un digrafo* sobre V es un par ordenado $D = (V, A)$ donde $A \subseteq V \times V$. Los elementos de V se llaman *nodos* o *vértices* del grafo y los elementos de A se conocen como *arcos*.

Si $a = (v, w)$ es un arco, entonces v es el extremo inicial de a y w es su extremo final. Se dice además que a es incidente a v y w. Al igual que para el caso de no dirigidos, denotaremos al arco $(v, w) \in A$ simplemente por ij. Todos los conceptos introducidos en las páginas anteriores pueden trasladarse al caso de digrafos.

Ejemplo: Los conjuntos $V = \{a, b, c, d, e, f, g\}$ y $E = \{(a, b), (c, a), (b, c), (b, e), (b, d-, (c, d), (f, c), (d, e), (d, f), (e, f), (e, g), (g, f)\}$ definen un digrafo. En la Figura 1.4 se presenta una representación esquemática.



$G[E']$
$E' = \{\{a, b\}, \{b, c\}\}$

$G[V']$
$V' = \{a, b, d\}$

Figura 1.3: Un subgrafo generador $G[E']$, y un grafo inducido $G[V']$ a partir del grafo G de la Figura 1.2

Dado un grafo $G = (V, E)$ dos nodos $v, w \in V$ se dicen *vecinos* o *adyacentes*, si $ij \in E$. Se define a la *vecindad* de un nodo $v \in V$ como el conjunto $N(v)$ de vértices de V adyacentes a v. En otras palabras, $N(v) = \{w \in V : ij \in E\}$. El concepto de vencindad se define de forma similar para digrafos, aunque en este caso es conveniente precisarlo de forma más detallada. Dado un digrafo $D = (V, A)$, si $ij \in A$ decimos que los nodos v y w son *vecinos*, y además que v es el *antecesor* de w, y w el *sucesor* de v. Para $v \in V$ definimos tres conjuntos:

- la vecindad de los sucesores: $N^+(v) := \{w \in V : w$ es un sucesor de $v\}$
- la vecindad de los antecesores: $N^-(v) := \{w \in V : w$ es un antecesor de $v\}$
- la vecindad (global) de v: $N(v) := N^+(v) \cup N^-(v)$.

El grado $d(v)$ de un nodo $v \in V$ es la cantidad de aristas incidentes a v. Para digrafos se distinguen además siguiendo el esquema descrito en el párrafo anterior el grado entrante $d^-(v)$ (cantidad de aristas cuyo extremo final es v) y el grado saliente $d^+(v)$ (cantidad de aristas que tienen a v como extremo inicial. Un digrafo

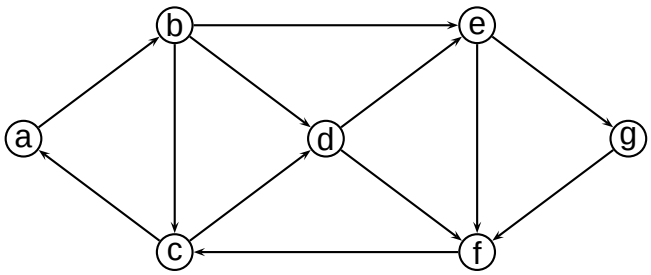


Figura 1.4: Un grafo dirigido

que satisface $d^-(v) \leq 1$, para todo nodo $v \in V$ se conoce como *arboresencia*.

Dado un grafo G (dirigido o no), una *cadena* P es una sucesión alternada de nodos y aristas (o arcos) de la forma $v_0, e_1, v_1, e_2, v_2..., e_k, v_k$, donde: $v_{v-1}, v_v \in V$ son los extremos de la arista e_v, $\forall v \in \{1, 2, .., k\}$.

Si P es una cadena en un grafo dirigido, los arcos de P que tienen la forma $e_v = (v_{v-1}, v_v)$ se conocen como arcos ”hacia adelante”, los otros arcos se conocen como arcos ”hacia atrás”. Una cadena cuyos arcos son todos ”hacia adelante”, se denomina *cadena orientada*.

En general, una cadena puede contener nodos y arcos repetidos. Una cadena que no contiene aristas o arcos repetidos se conoce como *sendero*. Un camino P es un sendero en el que todos los arcos son ”hacia adelante”. Si además P no contiene nodos repetidos, se conoce como *camino elemental* (De manera análoga se define un sendero elemental). Notar que cuando el grafo no contiene arcos (o aristas) paralelos un camino (sendero, cadena, etc) puede escribirse simplemente como una secuencia de nodos, sin riesgo de confusión. Un sendero (o camino) se dice cerrado si se cumple que $v_0 = v_k$. Un sendero cerrado elemental se llama ciclo, un camino cerrado elemental es un circuito.

Definición 1.3 (Longitud de un camino)**.** Dado un grafo dirigido $D = (V, A)$ y una función de costos sobre los arcos,

$$\begin{aligned} c :& A \longrightarrow \mathbb{R} \\ & e \longrightarrow c(e) := c_e, \end{aligned} \tag{1.1}$$

La longitud de un camino $P : v_0, e_1, v_1, e_2, v_2..., v_k$, se define por

$$c(P) = \sum_{v=1}^{k} c_{e_v} \tag{1.2}$$

En las secciones siguientes supondremos siempre que $V = \{1, 2, 3, ..., n\}$ y $|E| = m$. Además, cuando hablemos de un grafo será de uno no dirigido, a menos que se especifique explicitamente lo contrario.

1.2. Complejidad de Algoritmos

Un marco teórico comúnmente aceptado para el estudio de la eficiencia de los algoritmos es el análisis de complejidad. El mismo consiste en evaluar el número de operaciones requeridas por un algoritmo para resolver una determinada instancia de un problema, como función de la cantidad de memoria requerida para almacenar dicha instancia en un computador (llamada también la *longitud de codificación* de la instancia). Debido a que es casi imposible predecir con exactitud el número de operaciones requeridas para una instancia arbitraria, se emplea una notación asintótica. Exponemos a continuación, a breves rasgos, las ideas fundamentales de la teoría de complejidad. Información más detallada puede encontrarse en [9], [13], entre otros.

Definimos un *problema* como una pregunta general que depende de varios parámetros de entrada y para la cual se pide seleccionar una respuesta de entre un conjunto

de respuestas posibles. Un problema que admite como respuestas posibles "si" o "no" se conoce como *problema de decisión*. Una *instancia de un problema* Π se obtiene al asignar valores concretos para los parámetros Π.

Toda instancia puede almacenarse en la memoria de un ordenador como una sucesión de símbolos de un alfabeto. En la práctica, el alfabeto empleado contiene solo dos símbolos: 0 y 1. El número de símbolos necesarios para almacenar una instancia I se conoce como *longitud de codificación* y suele representarse por $[I]$. Como ejemplo calcularemos la longuitud de codificación para un grafo cuando es almacenado en un esquema de listas (en la siguiente sección se detalla más sobre las formas de almacenamiento o representaciones computacionales de grafos).

Ejemplo: Dado un grafo dirigido $D = (V, A)$ donde la cardinalidad del conjunto de vertices es $\mid V \mid = n$ y la del conjunto de arcos es $\mid A \mid = m$, tenemos que el almacenamiento consiste en definir listas de incidencia para cada nodo del grafo, por lo tanto necesitamos almacenar primero los n nodos y para cada nodo almacenar los nodos incidentes a él (nodos que son incidentes al mismo arco). Se tiene que para almacenar un número natural k la longuitud de codificación es $\lceil logk \rceil$, es así como que la longitud de codificación para almacenar un grafo es igual a la suma de la longitud de codificación de los n nodos la cual es igual a $n * \lceil logk \rceil$ mas la longitud de codificación de los arcos representados por listas de incidencia que sería $m * \lceil logk \rceil$, es decir $n * \lceil logk \rceil$ + $m * \lceil logk \rceil$.

Dadas dos funciones $f, g : \mathbb{N} \to \mathbb{R}$, decimos que

- $f \in O(g)$ si y sólo si $\exists n_0 \in \mathbb{N}, c \in \mathbb{R}$ tal que $f(n) \leqslant c * g(n), \forall n \geqslant n_0$
- $f \in \Omega(g)$ si y sólo si $\exists n_0 \in \mathbb{N}, c \in \mathbb{R}$ tal que $f(n) \geqslant c * g(n), \forall n \geqslant n_0$
- $f \in \Theta(g)$ si y sólo si $f \in O(g) \wedge f \in \Omega(g)$

Un algoritmo de solución ALG para un problema Π se dice *polinomial* (o eficien-

te) si para cada instancia I de Π, con longitud de codificación suficientemente grande, se cumple que el número de operaciones requeridas por ALG para resolver I está acotado por un polinomio en $[I]$. Formalmente, sea $k_{ALG}(I)$ el número de operaciones requeridas por ALG para resolver una instancia I de Π. Asociamos a ALG la función:

$$t_{ALG} : \mathbb{N} \Rightarrow \mathbb{R}$$
$$n \Rightarrow t_{ALG} := \max \{k_{ALG}(I) : [I] = n\}$$

Decimos que ALG es polinomial si existe un polinomio $p : \mathbb{N} \to \mathbb{R}$ tal que $t_{ALG} \in O(p)$.

Un problema Π es polinomial si existe un algoritmo polinomial de solución para Π. El conjunto (o clase) de todos los problemas polinomiales se representa po $\mathcal{P}$. Hay problemas computacionales difíciles para los cuales hasta la actualidad no se conocen algoritmos polinomiales de solución. Aunque no se ha demostrado que tales algoritmos no existan, la teoría de complejidad presenta un argumento fundamental para suponer que esto es así. Describimos este resultado a continuación.

Un problema Π de decisión pertence a la clase NP si existe un problema polinomial de decisión Π' con la propiedad de que para cada instancia de I de Π con respuesta "sí", existe una instancia I' de Π' que satisface:

- La respuesta de I' es "sí"
- $[I']$ está acotada por un polinomio en $[I]$

En otras palabras,toda instancia de respuesta "$s\acute{i}$" de Π tiene un *certificado* asociado a ella, cuya validez puede ser verificada por un algoritmo polinomial. Está definición es tan amplia que la mayoría de problemas de optimización combinatoria conocidas pueden ser asociados a problemas de decisión pertenecientes a NP. En particular $P \subseteq NP$.

Cook demostró en 1971 [3] que existe un problema de decisión (conocido como problema de satisfactibilidad SAT) con la propiedad de que todos los problemas de la clase NP pueden ser reducido a SAT en un número polinomial de operaciones. En consecuencia, de existir un algoritmo polinomial SAT, el mismo podría utilizarse para definir un algoritmo polinomial para cada problema de NP y se tendría $P = NP$. Un problema con esta característica se dice $NP - completo$. Desde entonces, se ha demostrado que la mayoría de los problemas computacionales difíciles son (o pueden asociarse a) problemas $NP-$completos: el problema del circuito hamiltoniano, el problema del agente viajero, problemas de empacamiento, particionamiento y recubrimiento de conjuntos, e incluso problemas de caminos más cortos sobre grafos con circuitos de costo negativo.

Aunque el sentido común sugiere que $P \neq NP$ y que por tanto no existen algoritmos polinomiales para ninguno de estos problemas, tal conjetura todavía no ha podido ser demostrado y constituye uno de los siete problemas del milenio[2].

1.3. Representación computacional de grafos

Los algoritmos que trabajan sobre grafos deben contar con una estructura de datos apropiada para la representación de los datos. En esta sección vamos a describir cuatro formas posibles de almacenar grafos (dirgidos o no) en el computador. El lector intersado puede referirse a [14] para más información detallada en torno a este tema.

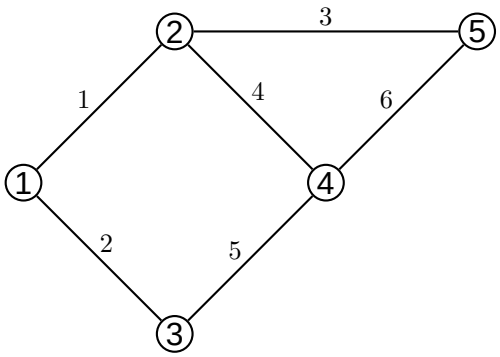


Figura 1.5: Grafo no dirigido. Las aristas han sido numeradas arbitrariamente como mecanismo de referencia.

1.3.1. Lista de arcos o aristas

Este tipo de representación consiste en mantener un arreglo de m pares de enteros en el intervalo $\{1, 2, ..., n\}$, donde cada elemento del arreglo corresponde a una arista (o un arco) del grafo.

Ejemplo: El grafo de la Figura 1.5 puede representarse mediante los arreglos $E[1] = \{1, 2\}, E[2] = \{3, 1\}, E[3] = \{2, 5\}, E[4] = \{2, 4\}, E[5] = \{3, 4\}, E[6] = \{4, 5\}$.

1.3.2. Matriz de adyacencia

Un grafo no dirigido se puede representar como una matriz cuadrada $H = (h_{ij})$ de orden n, donde los elementos de H son de la forma:

$$h_{ij} = \begin{cases} 1 & \text{si } v \text{ y } w \text{ son nodos adyacentes}, \\ 0 & \text{caso contrario}. \end{cases} \tag{1.3}$$

Ejemplo: La representación como matriz de adyacencia del grafo de la Figura 1.5 es la siguiente:

$$H = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 \end{pmatrix}$$

Notar que H es una matriz simétrica, pues la adyacencia es un concepto no dirigido. En el caso de digrafos, los elementos de H se suelen definir de la siguiente manera:

$$h_{ij} = \begin{cases} 1 & \text{si } (v, w) \text{ es un arco del digrafo}, \\ 0 & \text{caso contrario}. \end{cases} \tag{1.4}$$

1.3.3. Matriz de incidencia

El concepto de incidencia puede emplearse para definir una nueva representación computacional de un grafo $G = (V, E)$ mediante una matriz $K = (k_{ij})$ de orden $n \times m$, cuyos elementos son de la forma:

$$k_{ij} = \begin{cases} 1 & \text{si la arista } w \text{ es incidente al nodo } v, \\ 0 & \text{caso contrario}. \end{cases} \tag{1.5}$$

Ejemplo: La matriz de incidencia para el grafo de la Figura 1.5 es:

$$K = \begin{pmatrix} 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \end{pmatrix}$$

Notar que K tiene exactamente dos elementos con el valor de 1 en cada columna. En el caso de grafos dirigidos, K se define de la siguiente manera:

$$k_{ij} = \begin{cases} -1 & \text{si } v \text{ es el nodo inicial de } w, \\ 1 & \text{si } v \text{ es el nodo final de } w, \\ 0 & \text{en los demás casos.} \end{cases} \tag{1.6}$$

1.3.4. Listas de adyacencia

La representación computacional más ampliamente empleada en la práctica (debido a la eficiencia en el acceso a los datos) es la de listas de adyacencia. Consiste en almacenar, para cada nodo $v \in V$, un arreglo o una lista L_v con los nodos de la vecindad $N(v)$. En el caso de los grafos dirigidos, pueden considerarse alternativamente listas para los sucesores, antecesores (o para ambos). Generalmente, la gran mayoría de algoritmos para problemas de optimización en grafos (incluyendo aquellos desarrollados en la presente tesis) emplean esta representación.

Ejemplo: Para el grafo de la Figura 1.5, las listas de adyacencia son: $L_1 =< 2, 3 >$, $L_2 =< 1, 4, 5 >$, $L_3 =< 1, 4 >$, $L_4 =< 2, 3, 5 >$ y $L_5 =< 2, 4 >$

1.4. Problemas de caminos más cortos

En esta sección presentamos conceptos relacionados con el problema que sirve como base teórica para la aplicación estudiada en esta tesis: el problema de caminos más cortos. Su definición formal se presenta en la sección 1.4.1, mientras que las secciones 1.4.2 y 1.4.3 tratan acerca de los algoritmos de solución más comunes. La sección 1.5. describe resultados en torno a una generalización importante del problema: la búsqueda de caminos más cortos con restricciones de recursos.

1.4.1. Introducción

En su versión clásica más sencilla, el problema del camino más corto(SSP- Shortest Path Problem) puede formularse como sigue a continuación.

Definición 1.4 (Problema del camino más corto (SPP))**.** Dado un digrafo $D = (V, A)$, una función $c : A \longrightarrow \mathbb{R}$ de costos sobre los arcos, y dos nodos $r, s \in V$, el *problema del camino más corto* consiste en encontrar el camino de menor longitud que empieza en r y termina en v.

Este problema puede formularse como un problema entero:

$$\text{mín} \sum_{(v,w)\in A} c_{vw} X_{vw} \tag{1.7}$$

$$\sum_{(v,w)\in A} X_{vw} \leq 1 \qquad \forall v \in V \tag{1.8}$$

$$\sum_{(v,w)\in A} X_{vw} - \sum_{(w,v)\in A} X_{wv} = \alpha_v \qquad \forall v \in V,\ donde \tag{1.9}$$

$$\alpha_v = \begin{cases} 1 \text{ ,si } v = r \\ -1 \text{ ,} v = s \\ 0 \text{ ,} v \in V \setminus \{r, s\} \end{cases} \tag{1.10}$$

$$\sum_{(v,w)\in \delta(S)} X_{vw} \leq \mid S \mid -1 \qquad \forall S \subseteq V \tag{1.11}$$

$$X_{vw} \in \{0, 1\} \tag{1.12}$$

Usando una variable binaria X_{vw} para cada arco $(v, w) \in A$, de la forma que

$$X_{vw} = \begin{cases} 1 \text{ ,si arco } (v,w) \text{ es usado en la solución} \\ 0 \text{ ,caso contrario} \end{cases}$$

donde $\delta(S) := \{vw \in A : v \in S, w \in S\}$ se conoce como el conjunto de arcos internos a $S \subseteq V$.

Tenemos que 1.7, representa la función objetivo minimizar el costo de viajar de r a s, es decir, se cumple el objetivo al tener la ruta de menor costo de un conjunto de viajes o rutas factibles. Para que una ruta sea factible debe cumplir que:

1. En la solución no debe existir una ruta con nodos repetidos, es decir que pase dos veces por el mismo lugar para eso definimos 1.8.

2. El grupo de ecuaciones 1.9 nos garantiza que se forma un camino o ruta desde r a s, si $v = r$ entonces $\alpha_r = 1$ y con esto sólo hay un arco que sale de r y con $v = s$ tenemos $\alpha_r = -1$, es decir sólo un arco o camino que llega a s, para el resto de los nodos se tiene $\alpha_r = 0$ y con esto se destruye la posibilidad de tener ciclos que formen parte de un camino.

3. Pero aún no son suficiente las anteriores restricciones para garantizar un camino factible ya que existe la posibilidad de tener dos conjuntos de arcos que cumplan las restricciones antes definidas y que no exista una conexión entre estos conjuntos, es decir que no exista un arco que los una. La ecuación 1.11 no permite que el camino o ruta este desconectado..

Cuando no existen circuitos negativos la solución al problema de caminos más cortos es un camino elemental.

1.4.2. Algoritmos de solución para el SPP

Los algoritmos de solucón para el SPP emplean la idea de potencial, definida a continuación.

Definición 1.5 (Potencial)**.** Dado un digrafo $D = (V, A)$, una función $c : A \longrightarrow \mathbb{R}$ de costos sobre los arcos, y un nodo $r \in V$, una función $y : V \longrightarrow \mathbb{R}$ se denomina un *potencial factible*, si se satisfacen las dos condiciones siguientes:

- $y_r = 0$
- $\forall e = (v, w) \in A$, $c_e \geq y_w - y_v$

Como lo muestra el siguiente resultado, los potenciales están íntimamente relacionados con los caminos más cortos:

Teorema 1.1. *Si y es un potencial factible y P un camino de la forma $P : r = v_0, e_1, v_1, e_2, v_2 ..., v_k = s$, entonces se cumple que: $c(P) \geq y_s$*

Demostración:

$c(P) = \sum_{v=1}^{k}(c_{e_v}) \geq \sum_{v=1}^{k}(y_{v_v} - y_{v_{v-1}}) = y_{v_k} - y_{v_0} = y_s - y_r = y_s$.

Corolario 1.1. *Si P es un camino desde r hasta s tal que el costo de P es $c(P) = y_s$, entonces P es un camino de longitud mínima.*

La observación anterior constituye la idea fundamental del siguiente algoritmo, publicado en [11] por Ford (1956), que se basa en construir un potencial factible y una arborecencia de caminos para los cuales la desigualdad $c(P) \geq y_s$ se satisfaga con la igualdad.

Algoritmo Ford

{Inicialización}

2: $y_r := 0$;

$y_v := +\infty, \forall\, v \in V \setminus \{r\}$;

4: $p_v := -1, \forall\, v \in V$; {Lazo principal}

mientras exista $e = (u, w) \in A$ tal que $c_e < y_w - y_u$ **hacer**

6: $y_w := c_e + y_u$;

$p_w := u$;

8: **fin mientras**

El algoritmo mantiene dos arreglos y, p de dimensión n. El arreglo y se emplea para construir un potencial factible, mientras p almacena una arborescencia de caminos que parten desde r. Inicialmente, el ”potencial”y es definido como cero sobre el nodo de partida y $+\infty$ sobre los demás nodos (para las operaciones aritméticas supondremos que $+\infty + x = +\infty, \forall x \in \mathbb{R}$). En cada iteración, el algoritmo çorregeün arco que viole la segunda condición de la definición de potencial factible, donde corregir un arco significa cambiar el valor de y en el nodo de llegada para que esta condición se satisfaga con igualdad. El algoritmo termina cuando no existen más arcos por corregir.

Las componentes distintas a -1 del arreglo p definen una arboresencia con raíz en r, representada como un vector de predecesores que indica para cada nodo $v \in V \setminus \{r\}$, cuál es su antecesor p_v en el camino que va desde r hasta v. Notar que este predecesor es único por que $d(v) \leq 1$ dentro de la arborecsencia.

Cuando el algortimo termina, si lo hace, y es un potencial factible y además, del resultado que presentamos a continuación, se sigue que para cada nodo $v \in V \setminus \{r\}$, la longitud del camino de r a v almacenado en p es exactamente y_v. El Colorario

1.2. nos permite concluir entonces que p define una arborecsencia de caminos más cortos.

Teorema 1.2. *Si D no contiene circuitos de costo negativo, en cada iteración del algoritmo de Ford se cumple:*

(I) Si $y_v < +\infty$, entonces y_v es el costo de un camino elemental de r a s

(II) Si $p_v \neq -1$, entonces p define un camino de r a s de costo menor o igual a y_s.

Para una demostración del teorema anterior, se hace referencia, por ejemplo, a [11]. Para demostrar la validez del algoritmo de Ford basta, por tanto, con demostrar que el mismo termina después de un número finito de pasos. Cuando D no contiene circuitos negativos y c toma valores enteros, esto es una consecuencia del siguiente lema:

Lema 1.1. *Si D no contiene circuitos de costo negativo y c toma valores únicamente enteros, entonces el algoritmo de Ford termina después de $O(\alpha n^2)$ iteraciones donde $\alpha := \max\{2|c_e| : e \in A\}$.*

Demostración: Sea $v \in V$. Notar que el costo de cualquier camino elemental de r a v es un valor entero del conjunto $N := \{\frac{-(n-1)\alpha}{2}, \frac{-(n-1)\alpha}{2} + 1, ... - 1, 0, 1, .. \frac{(n-1)\alpha}{2}\}$. El Teorema 1.2 (v) nos dice que y_v debe ser siempre igual a un valor de N y como y_v es corregido siempre hacia abajo, se sigue que y_v puede ser corregido máximo $|N| - 1 = \frac{2(n-1)\alpha}{2} = (n-1)\alpha \leq \alpha n$ veces. Debido a que en cada iteración el valor de y es corregido para algunos de los n nodos de D, el algoritmo termina después de máximo αn^2 iteraciones.

La siguiente instancia (Figura 1.6), propuesta en [10] por Edmons (1965), demuestra que la cota superior de αn^2 iteraciones puede ser alcanzada en ciertos casos:

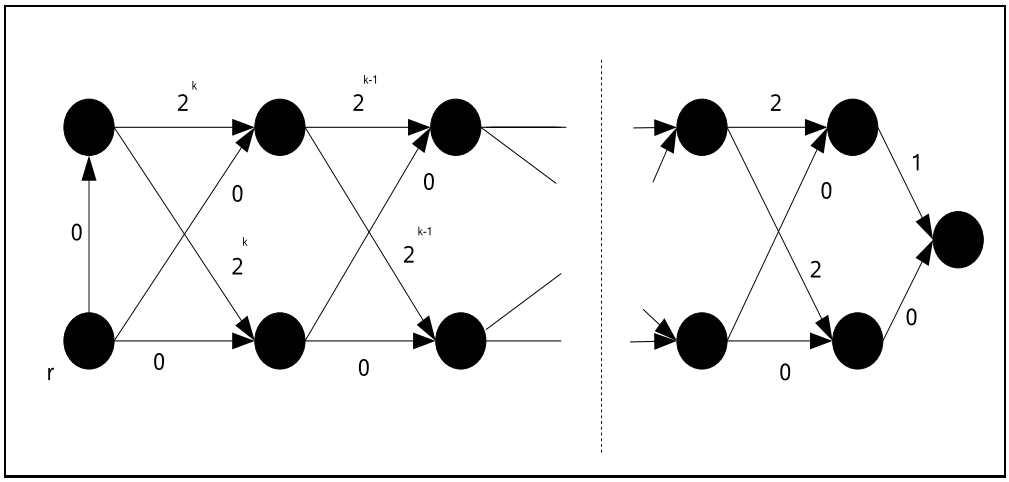


Figura 1.6: Un Grafo G_k

El algoritmo de Ford llega a requerir más de 2^k iteraciones para resolver el problema de caminos más cortos, con r como el nodo de partida, para esto hay que forzar el algoritmo a resolver el mismo problema en G_{k-1} dos veces, y emplear luego inducción matemática.

El ejemplo anterior demuestra que, en un sentido estricto, el algortimo de Ford no es polinomial debido a que el número de operaciones puede llegar a ser proporcional a α, pero la longitud de codificación de una instancia del SPP es de $O(mlog\alpha)$.

Observar además que si D contiene circuitos negativos, entonces el algoritmo no termina.

Algoritmo de Ford - Bellman

Bellman (1958) introdujo en [12] ciertas correcciones al algoritmo de Ford con las cuales éste detecta la existencia de ciclos negativos y alcanza un orden estrictamente polinomial de $O(nm)$. El algoritmo de Ford - Bellman se basa en procesar los arcos en ”barridos”sucesivos. Es decir, ningún arco es corregido dos veces antes de haber chequeado por lo menos una vez todos los demás arcos del grafo. Para expli-

car con mayor detalle la idea detrás de este algoritmo, presentamos a continuación algunos resultados preliminares.

Definición 1.6 (Sucesiones incrustadas)**.** Dadas dos sucesiones finitas $(a_i)_{v\in I}$, y $(b_j)_{w\in J}$ con $|J| \leq |I|$, decimos que b está incrustada en a, si existe una función estrictamente creciente $f : J \longrightarrow I$ tal que $b_w = a_{f(w)}\ \forall w \in J$. Es decir, si los elementos de b aparecen dentro de a en el mismo orden.

Teorema 1.3. *Sea S la sucesión formada por todos los arcos procesados por el algoritmo de Ford hasta la iteración $w - ésima$, sea P un camino de r hasta $v \in V$ y $E(P)$ la sucesión que contiene todos los arcos de P. Si $E(P)$ está incrustada en S, entonces $c(P) \geq y_v$.*

Demostración: Para $v \in V$ denotaremos por y_v^v el valor de y_v después de la $v-ésima$ iteración. Sea $P : r = v_0, e_1, v_1, e_2, v_2..., e_k, v_k = v$ como la sucesión $E(P) := e_1, e_2, ..., e_k = v$ ésta incrustada en S, entonces existen números naturales $q_1 < q_2 < ... < q_k$ tal que e_v fue corregido en la iteración $q_v, \forall v \in \{1, 2, .., k\}$. Luego $y_{v_v}^{q_v} = y_{v_{v-1}}^{q_v} + c_{e_v} \leq y_{v_{v-1}}^{q_{v-1}} + c_{e_v}$, pues el valor de $y_{v_{v-1}}$ sólo puede haber disminuido entre la iteración q_{v-1} y q_v. De aquí se sigue:

$$c(P) = \sum_{v=1}^{k}(c_{e_v}) \geq \sum_{v=1}^{k}(y_{v_v}^{q_v} - y_{v_{v-1}}^{q_{v-1}}) = y_{v_k}^{q_k} - y_{v_0}^{q_0} = y_{v_k}^{q_k} = y_{v_k}$$

Corolario 1.2. *Si para todo $v \in V\backslash\{r\}$, la sucesión de arcos correspondientes a un camino más corto de r a v aparece como sucesión incrustada en S, entonces y define un potencial factible.*

La idea principal del algoritmo de Ford-Bellman es construir una sucesión $S := S_1, S_2, ..., S_{n-1}$ donde S_v es subsucesión que contiene todos los arcos de A y corresponde al v-ésimo ”barrido”. Notar que cualquier camino elemental en D tiene sus

arcos incrustados en S. En particular, si D no contiene circuitos negativos, todos los caminos más cortos de r a $v \in V \setminus r$ tienen sus arcos incrustados en S, asý que el algortimo obtiene un potencial factible en máximo $n-1$ barridos, como consecuencia del Corolario 1.2.

Algoritmo Ford - Bellman

{**Inicialización**}

2: $y_r := 0$**;**

$y_v := +\infty$**,** $\forall\ v \in V \backslash \{r\}$**;**

4: $p_v := -1$**,** $\forall\ v \in V$**;**

$v := 0$**;**

6: **mientras** $v < n$ **y** y **no es potencial factible hacer**

$v := v + 1$ **;**

8: **para** $(v, w) \in A$ **hacer**

si $y_w > y_v + c_{vw}$ **entonces**

10: $y_w := y_v + c_{vw}$**;**

fin si

12: **fin para**

fin mientras

Al finalizar el algoritmo se cumple una de las siguientes condiciones:

- y es un potencial factible, entonces p define una arborescencia de caminos más cortos de r a los demás nodos
- $v = n$, entonces D contiene(al menos) un circuito de costo negativo, pues se han realizado $n - 1$ barridos sin obtener un potencial factible.

Como el número de barridos está limitado a n y en cada barrido se corrigen m arcos, el algoritmo de Ford-Bellman alcanza un orden de complejidad de $O(mn)$.

Este es el mejor orden de complejidad obtenido para el SPP en grafos generales, pero puede ser mejorado para instancias del SPP que presentan una estructura especial.

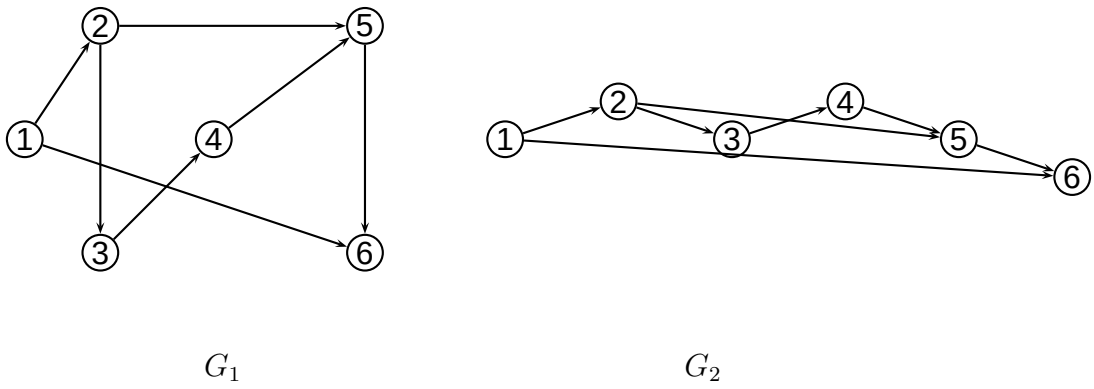
1.4.3. Casos especiales

Algoritmo de caminos más cortos para grafos acíclicos

Definición 1.7 (Grafos Acíclicos)**.** Un digrafo se dice acíclico si no contiene un circuito como subgrafo.

Si $D = (V, A)$ es un digrafo acíclico, sus nodos pueden ordenarse en una sucesión $v_1, v_2, ..., v_n$ con la siguiente propiedad:
$\forall\ v, w: \quad (v_v, v_w) \in A \Longrightarrow v < w$, tal ordenamiento se conoce como ordenamiento topológico.

Esto equivale a afirmar que el grafo puede dibujarse de tal forma que todos sus arcos vayan de "izquierda a derecha". En la Figura 1.7 el digrafo G_2 representa un ordenamiento topológico del digrafo G_1.



G_1 G_2

Figura 1.7: Ordenamiento topológico.

Como en todo grafo acíclico existe al menos un nodo con grado de entrada igual

a cero, empleando inducción matemática puede obtenerse el siguiente resultado:

Teorema 1.4. *Un ordenamiento topológico puede construirse en* $O(m)$ *interacciones.*

Definición 1.8 (Procesar un nodo)**.** Sea $v \in V$, usaremos la expresión ”procesar v”para designar el siguiente bucle de operaciones. Para todo $(v, w) \in V$, si $y_w > y_w + c_{vw}$ entonces corregir (v, w).

El siguiente algoritmo resuelve el problema del camino más corto sobre un grafo acíclico en $O(m)$ operaciones.

Algoritmo para grafos acíclicos

{**Entrada**}

2: $D = (V, A)$ **acíclico;**

$c : A \to \mathbb{R}$**;**

4: Calcular un ordenamiento topológico $v_1, v_2, ..., v_n$ **para** D**;**

Inicializar $y_s := 0,\ y_v := +\infty,\ \forall\, v \in V \setminus s$

6: para $v := 1, 2, 3, ..., n$ **hacer**

Procesar v_v**;**

8: fin para

Notar que cualquier camino en D está contenido en la sucesión de arcos procesados por el algoritmo. Del Teorema 1.4 se sigue entonces que tras la primera barrida, y representa un pontencial factible.

Costos no negativos

Otro caso particular en el que el problema de caminos más cortos puede resolverse de manera más eficiente es cuando ningún arco del digrafo tiene costo negativo. Es posible emplear entonces el siguiente algoritmo propuesto en [8] por Dijkstra (1959).

Algoritmo de Dijkstra

{**Inicialización**}

2: $y_r := 0$**;**

$y_v := +\infty$**,** $\forall\ v \in V \backslash \{r\}$**;**

4: $p_v := -1$**,** $\forall\ v \in V$**;**

$S := V$**;**

6: **mientras** $S \neq \phi$ **hacer**

Elegir $v \in S$ **con** y_v **mínimo ;**

8: **Procesar** v**;**

$S = S \backslash \{v\}$ **;**

10: **fin mientras**

Como en cada iteración el conjunto S disminuye en un nodo, después de n iteraciones se cumple $S = \phi$ y el algoritmo termina.

Sea y_v^w, con $0 \leq w \leq n$, el valor de y_v luego de la w-ésima iteración y sea q_v la iteración en la que v es procesado por el algoritmo, notar que $y_v^0 \leq y_v^1 \leq ... \leq y_v^n$. Además de la definición del algoritmo puede demostrarse que luego de que v ha sido procesado, el valor de y_v no vuelve a ser modificado, es decir, que $y_v^n = y_v^{q_v}$. Por lo tanto, al terminar el algoritmo,y define un potencial factible, pues para cualquier arco $vw \in A$ se tiene $y_v^{q_v} + c_{vw} = y_w^{q_v}$ y de las anteriores observaciones se sigue que $y_v^n + c_{vw} \geq y_w^n$.

Eficiencia del algortimo de Dijkstra

En una imlementación sencilla, el algoritmo requiere de $O(n)$ operaciones para la inicialización de y y p, $O(m)$ para corregir arcos, y $O(n)$ para actualizar S. Adicionalmente, $O(n^2)$ operaciones son requeridas para encontrar los nodos con valores

mínimos de y_v ($O(n)$ en cada iteración). Como $O(n^2)$ domina a $O(n)$ y $O(m)$, se sigue que el algoritmo de Dijkstra resuelve el problema de caminos más corto en $O(n^2)$ operaciones. Este orden de complejidad puede mejorase empleando mejores estructuras de datos para la selección del mínimo (por ejemplo, colas de prioridad [14]).

1.5. Problemas de caminos más cortos con restricciones de recursos

En algunas aplicaciones prácticas (pricipalmente en el campo de enrutamiento de vehículos), no sólo nos interesa encontrar el camino más corto entre dos nodos, sino también cumplir con ciertos requerimientos adicionales tales como ventanas de tiempo para la llegada, capacidad de transporte de los vehículos, consumo de combustibles, etc. En estos casos el problema del camino más corto aumenta su grado de complejidad y la generalización del mismo se conoce como problema de caminos más cortos con restricciones de recursos (SPPRC-Shortes Path Problem With Resource Constraints). Un caso específico importante que se presenta comúnmente consiste en la inclusión de restricciones en cuanto al tiempo en el cual es permitido visitar los nodos. La versión correspondiente del *SPP* se conoce en este caso como el problema de caminos más cortos con ventanas de tiempo (SPPTW-Shortes Path Problem With Time Windows).

Sean $D = (V, A)$ un grafo dirigido y $R = \{1, 2, ...k\}$ un conjunto cuyo elementos denominaremos en adelante *recursos*. Asociadas a R y D, se definen las siguientes funciones:

- Una función de *costo* $c : A \longrightarrow \mathbb{Q}$ sobre todos los arcos de D.

- Una función *consumo* de recursos $t^r : A \longrightarrow \mathbb{Q}$ para cada recurso $r \in R$.
- Una función *límite inferior* $a^r : V \longrightarrow \mathbb{Q}$ y una función *límite superior* $b^r : V \longrightarrow \mathbb{Q}$ para cada recurso $r \in R$. Para cada nodo $v \in V$, el intervalo $[a_v^r, b_v^r]$ se conoce como la ventana asociada al r -ésimo recurso.

Consideremos ahora un camino $P : v_0, e_1, v_1, e_2, v_2 ..., v_k$ entre dos nodos v_0 y $v_k \in V$. Para cada recurso $r \in R$ y cada nodo v_h visitado por P, el *nivel* $T_{v_h}^r$ de r en v_h está definido recursivamente por la siguiente fórmula:

$$T_{i_h}^r = \text{máx}\left\{T_{i_{h-1}}^r + t_{i_{h-1},i_h}^r, a_{i_h}^r\right\}, \tag{1.13}$$

para $1 \leq h \leq k$ el nivel inicial $T_{v_0}^r$ se selecciona arbitrariamente dentro de la ventana del r -ésimo recurso para el primer nodo. El camino P es llamado factible si es posible seleccionar $T_{i_0}^r$ de tal forma que para cada recurso $r \in R$, y cada nodo i_h se cumple que $a_{i_h}^r \leq T_{i_h}^r \leq b_{i_h}^r$.

El *SPPRC* consiste en encontrar un camino factible de costo mínimo desde un nodo de origen s hasta un nodo de destino t. Este problema puede formularse como el modelo de programación matemática indicado a continuación.

$$\text{mín} \sum_{(v,w)\in A} c_{vw} X_{vw} \tag{1.14}$$

$$\sum_{w\in V} X_{vw} - \sum_{w\in V} X_{wv} = 1 \qquad , v = s \tag{1.15}$$

$$\sum_{w\in V} X_{vw} - \sum_{w\in V} X_{wv} = 0 \qquad , \forall v \in V \setminus \{s,t\} \tag{1.16}$$

$$\sum_{w\in V} X_{vw} - \sum_{w\in V} X_{wv} = -1 \qquad , v = t \tag{1.17}$$

$$X_{vw}(T_v^r + t_{v,w}^r - T_w^r) \leq 0 \qquad \forall (v,w) \in A, \forall r \in R \tag{1.18}$$

$$a_v^r \leq T_v^r \leq b_v^r \qquad \forall v \in V, \forall r \in R \tag{1.19}$$

$$X_{wv} \in \{0,1\} \qquad \forall (v,w) \in A \tag{1.20}$$

Usando una variable binaria X_{vw} para cada arco $(v,w) \in A$, de la forma que

$$X_{vw} = \begin{cases} 1 \text{ ,si arco } (v,w) \text{ es usado en la solución} \\ 0 \text{ ,caso contrario} \end{cases}$$

La modelación del SPPRC se basa en construir un camino factible de s a t, las ecuaciones desde 1.15 hasta 1.17 como en el caso del SPP restringue el esapcio de solución para construir una sola ruta de s a t, mientras si alguna de las funciones de recursos es estrictamente positiva, entonces la restricción 1.18 impide la formación de subcircuitos en la solución. Cabe notar que este modelo no es lineal por la restricción 1.18 pero se puede linealizar transformando esta restricción. 1.19 nos garantiza que se cumpla la restricciones de recursos.

1.5.1. Algoritmos de solución para el SPPRC

Multiplicando por un factor escalar apropiado a t^r, a^r y b^r podemos asumir que estos valores son enteros para todo $r \in R$. Si esto es posible, se puede reducir el SPPRC al problema clásico de la ruta más corta SPP sobre un nuevo grafo $\tilde{D} = (\tilde{V}, \tilde{A})$ con un mayor número de nodos definido como sigue:

Creamos algunas copias $\hat{v}_1, \ldots, \hat{v}_{M(v)}$ de cada nodo $v \in V$. Estas copias corresponden a todas las combinaciones de los valores permitidos para los niveles de recursos en v. En adelante, estas combinaciones serán llamadas *estados* del nodo v. Como a^r y b^r son valores enteros, el número de estados de v es finito, y está dado por:

$$M(v) = \sum_{r=1}^{k} (b_v^r - a_v^r + 1),\ \forall v \in V$$

Por último, para cada arco $(v, w) \in A$, creamos arcos con el mismo costo c_{vw} entre todos los pares $(\hat{v}_\kappa, \hat{w}_\eta)$ de estados de v y w que satisfagan la siguiente condición: Si $T^r_{\hat{v}_\kappa}$ y $T^r_{\hat{w}_\eta}$ son los niveles del recurso $r \in R$ asociados a los estados $\hat{v}_\kappa$ y $\hat{w}_\eta$, entonces $T^r_{\hat{v}_\kappa} + t^r_{vw} = T^r_{\hat{w}_\eta}$, $\forall\, r \in R$.

Dados dos nodos $s, t \in V$, es evidente que para cada camino factible de s a t en D, va a existir un camino dirigido con el mismo costo entre dos estados $\hat{s}_\kappa$ y $\hat{t}_\eta$ en $\tilde{D}$, y viceversa. Por lo tanto, un algoritmo para el SPP clásico puede ser aplicado sobre $\tilde{D}$ para resolver el $SPPRC$. Sin embargo la aplicación de esta idea conlleva dos dificultades principales.

La primera tiene que ver con la función de costo: cuando al $SPPRC$ se lo encuentra como un subproblema dentro de algoritmo de solución a modelos de enrutamiento de vehículos, el grafo $\tilde{D}$ suele contener circuitos de costo negativo y tal como fue señalado en la sección , el SPP es entonces NP-diícil. Es preciso notar,

por otra parte, que si existe almenos un recurso de consumo estrictamente positivo, $\tilde{D}$ es acíclico por construcción. Un ejemplo comuún lo constituye el recurso tiempo en SPPTW.

La segunda dificultad tiene que ver con el tamaño del grafo auxiliar $\tilde{D}$. El número de nodos del mismo puede ser astronómicamente grande, debido a que estos representan todas las posibles combinaciones de los niveles de recursos admisibles. Una solución a esto consiste en no crear el grafo $\tilde{D}$ explicitamente, sino generar sus nodos y arcos ”sobre la marcha”, conforme estos son necesitados por el algoritmo. Los nodos de $\tilde{D}$ suelen mantenerse como conjuntos de *etiquetas* asociados a nodos del grafo original D.

Específicamente, consideremos un camino factible P_κ en D que inicia en el origen s y llega a un nodo $v \in V$ consumiendo $T^r_{v,\kappa}$ unidades de cada recurso $r \in R$, y con un costo de $c_{v,\kappa}$. Se asocia a P_κ una etiqueta en v de la forma $f(v,\kappa) = (c_{v,\kappa}, T^1_{v,\kappa}, \ldots, T^k_{v,\kappa})$. Las etiquetas pueden emplearse para reducir considerablemente la cantidad de caminos a explorar, como veremos a continuación.

Definición 1.9. Dadas dos etiquetas $f(v,\kappa)$ y $f(v,\eta)$, decimos que $f(v,\kappa)$ *domina* $f(v,\eta)$ si se cumple lo siguiente:

$$c_{v,\kappa} \leq c_{v,\eta},$$

$$T^r_{v,\kappa} \leq T^r_{v,\eta},\ \forall r \in R.$$

Un etiqueta es *eficiente* si no existe otra que la domine. Un camino que usa sólo etiquetas eficientes se llama un *camino eficiente*. Se puede observar que, si existe un camino factible para P_κ entre el origen s y cualquier nodo $v \in V$, entonces existe un camino eficiente de s a v con costo menor o igual a $c_{v,\kappa}$. Por lo tanto, al resolver una instancia del $SPPRC$, solo hace falta considerar etiquetas eficientes.

Denotamos por Q_v el conjunto de todas las etiquetas en el nodo v y sea $EFF(Q_v)$ el subconjunto de todas las etiquetas eficientes en Q_v. A continuación presentamos los tres algoritmos de solución más comunes para $SPPRC$, los cuales se basan en el cálculo de $EFF(Q_t)$ por medio de programación dinámica. El camino más corto de s a t es entonces obteniendo de la etiqueta con el menor costo en este conjunto.

Dada una etiqueta $f(v,\kappa) = (c_{v,\kappa}, T^1_{v,\kappa}, \ldots, T^k_{v,\kappa})$ en un nodo v, y un nodo $w \in N^+(v)$ sea $NEXT(f(v,\kappa), w)$ una función que retorna la etiqueta $f(w,\eta)$ en w definida por:

$$c_{w,\eta} = c_{v,\kappa} + c_{vw}$$

$$T^r_{w,\eta} = \max\{T^r_{v,\kappa} + t^r_{vw},\ a^r\},\ \forall r \in R.$$

Si $f(w,\eta)$ es infactible porque $T^r_{w,n}$ no está dentro de la ventana admisible, para algún $r \in R$ entonces convenimos que el valor retornado por $NEXT(f(v,\kappa), w)$ es un conjunto vacío.

En este contexto, Desrosiers, Pelletier y Sounmis (1983) propone en [7] el siguiente algoritmo.

Algoritmo (Corrección de Etiquetas)

{**Inicialización**}

2: $Q_s := \{(0, T^1_{s,\alpha}, \ldots, T^k_{s,\alpha})\}$**;**

$Q_v := \{(\infty, a^1, \ldots, a^k)\}\ \forall v \in V \setminus \{s\}$**;**

4: $L := \{s\}$**;**

mientras $L \neq \emptyset$ **hacer**

6: **Seleccionar un nodo** $v \in L$**;**

{**Procesar nodos sucesores**}

8: **para** $w \in N^+(v)$ **hacer**

$Q'_w := EFF(\cup_\kappa \{NEXT(f(v,\kappa), w)\} \cup Q_w)$**;**

si $Q'_w \neq Q_w$ **entonces**

$Q_w := Q'_w$**;**

$L := L \cup \{w\}$**;**

fin si

fin para

{**Reducción de** L}

$L := L \setminus \{v\}$**;**

fin mientras

Q_s es inicializado con alguna etiqueta válida $f(s, \alpha)$. Normalmente, esta etiqueta es parte de los datos de entrada para una instancia del problema, caso contrario Q_s debe inicializarse con todas las etiquetas (no dominadas) correspondientes a estados iniciales posibles.

El algoritmo mantiene una lista L de nodos por procesar. Cada vez que un nodo v es procesado, todas las posibles nuevas etiquetas para cada sucesor $w \in N^+(v)$ son generadas. Antes de añadir una nueva etiqueta al conjunto Q_w de etiquetas del nodo w, chequeamos que la misma no sea dominada por alguna otra etiqueta de Q_w, operación que puede requerir de un tiempo considerable si la etiqueta es añadida w es reinsertada en la lista de nodos por procesar. La ejecución del algoritmo termina cuando la lista L está vacía.

Si alguna de las funciones de consumo de recursos toma únicamente valores positivos, puede emplearse la misma idea del algoritmo de Dijkstra para resolver el $SPPRC$ más eficientemente. El algoritmo definido de esta manera se conoce como algoritmo de fijación de etiquetas (label setting) y es propuesto por Desrochers y Soumis (1988a) en [15]. El mismo mantiene una lista L de nodos a ser tratados,

y define en cada nodo un conjunto P_v de etiquetas *permanentes* cuyo valor no será modificado en iteraciones, futuras. Supondremos en adelante (sin restricción de la generalidad) la función de consumo asociada al primer recurso es estrictamente positiva. El algoritmo de fijación de etiquetas puede formularse como sigue a continuación:

Algoritmo (Fijación de Etiquetas)

{Inicialización}

$Q_s := \{(0, T^1_{s,\alpha}, \ldots, T^k_{s,\alpha})\}$;

$Q_v := \emptyset, \forall v \in V \setminus \{s\}$;

$P_v := \emptyset, \forall v \in V$;

mientras $\cup_{v \in V}(Q_v \setminus P_v) \neq \emptyset$ **hacer**

Seleccionamos una etiqueta $f(v, \kappa)$ de $\cup_{v \in V}(Q_v \setminus P_v)$ para la cual $T^1_{v,\kappa}$ sea mínima ;

{**Procesar la etiqueta** $f(v, \kappa)$}

para $w \in N^+(v)$ **hacer**

$Q_w := EFF(Q_w \cup NEXT(f(v, \kappa), w))$;

$P_v := P_v \cup f(v, \kappa)$

fin para

fin mientras

Se inicializa Q_s de la misma forma que en el algoritmo de Correcíon de Etiquetas. Adicionalmente, se mantiene un conjunto P_v de etiquetas permanentes para cada nodo $v \in V$. En cada iteración, se selecciona una etiqueta $f(v, \kappa)$ del conjunto $\cup_{v \in V}(Q_v \setminus P_v)$ para la cual el valor de $T^1_{v,\kappa}$ sea mínimo y se procesa esta etiqueta, la cual consiste en generar todas las posibles nuevas etiquetas(no dominadas). La etiqueta $f(v, \kappa)$ es luego aadida al conjunto de etiquetas ”permanentes”, lo que signi-

fica que no volverá a ser procesada. El algoritmo termina cuando todas las etiquetas son permanentes.

Desde el punto de vista de la complejidad, el paso crítico dentro de este algoritmo es la selección de la siguiente etiqueta a ser tratada, lo que comprende una comparación de los valores de $T^1_{v,\kappa}$ entre todas las etiquetas $f(v,\kappa)$(de todos los nodos) que todavía no hayan sido marcadas como permanentes . Para acelerar esta búsqueda, se emplea la idea de crear *buckets* (”baldes”). Un bucket es una lista no ordenada que contiene todas las etiquetas cuyos valores para el nivel del primer recurso están dentro de un intervalo específico. El $h-$*é*simo bucket, por ejemplo, podría contener las etiquetas $f(v,\kappa)$ para los cuales $hw \leq T^1_{v,\kappa} < (h+1)w$, donde w es una constante llamada el ancho de bucket y usualmente definida por:

$$w := \min_{vw \in A} t^1_{vw}$$

Por lo tanto, cuando una etiqueta perteneciente a un bucket B es tratada, todas las nuevas etiquetas creadas corresponderán a buckets que vienen después de B. Así la búsqueda de la etiqueta con el mínimo valor de $T^r_{w,n}$ puede restringirse al primer bucket que contenga etiquetas sin procesar. El uso de buckets requiere de un costo computacional extra para insertar las nuevas etiquetas creadas en los buckets correctos pero aún así esto resulta mucho más eficiente que mantener un conjunto con todas las etiquetas ordenadas.

La idea de usar buckets está presente también en un tercer tipo de algoritmos para el SPPRC, conocidos como algoritmos de arrastre de etiquetas (label pulling) y que fúeron propuestos por Desrochers (1986) en [6] (ver también [16]).

Algoritmos de arrastre de etiquetas (Label pulling)

Una desventaja que presentan los dos métodos descritos anteriormente es el hecho de que requieren calcular $EFF(Q_w)$ varias veces durante cada iteración (cada vez que se crea una etiqueta nueva para algún $w \in N^+(v)$). La determinación del conjunto de etiquetas no dominadas requiere de comparaciones lexicográficas sucesivas y puede torbarse computacionalmente muy costosa cuando la cantidad de recursos aumenta. Para hacer frente a esto, un tercer grupo de algoritmos construye las etiquetas de forma inversa. En cada iteración, los antecesores de un nodo w son examinados y sus etiquetas se .ªrrastran "hacia w en la forma usual. Como todas las nuevas etiquetas son creadas en w, unicamente el conjunto Q_w debe ser actualizado durante la iteración.

Algoritmo (Arrastre de Etiquetas)

{Inicialización}

2: $Q_s \leftarrow \{(0, T^1_{s,\alpha}, \ldots, T^k_{s,\alpha})\}$;

$\pi_s \leftarrow b^1_s$

4: $Q_v \leftarrow \emptyset, \forall v \in V \setminus \{s\}$;

$\pi_v \leftarrow a^1_v, \forall v \in V \setminus \{s\}$;

6: **mientras** exista $w \in V$ con $\pi_w < b^1_w$ **hacer**

8: Seleccionar $w \in V$ con el mínimo π_w;

{Procesar el nodo w}

10: **para** $v \in N^- w$ **hacer**

$Q_j \leftarrow Q_w \cup (\cup_\kappa NEXT(f(v,\kappa), w))$;

12: **fin para**

$Q_w \leftarrow EFF(Q_w)$

14: $\pi_w \leftarrow \min\{b_w^1, \pi_w + w\}$

fin mientras

Como en el último caso, w es el mínimo valor de la función t^1 sobre A. En cada nodo w, se mantiene una marca π_w que indica el conjunto de etiquetas permanentes dentro de Q_w. Estas son precisamente aquellas cuyo valor concerniente al primer recurso es menor que π_w. En efecto, puede demotrarse por inducción matemática que todas las etiquetas nuevas creadas sobre w satisfacen $T_{w,k}^1 \geq \pi_w$

En cada iteración, el nodo w con el menor valor de π_w es seleccionado. Examinamos sus antecesores, el algoritmo intenta crear nuevas etiquetas en Q_w. Luego, se eliminan etiquetas dominadas y el valor de π_w es icrementado en w.

Después de un número finito de iteraciones, los valores π_v alcanzan el límite superior b_v^1 para cada nodo $v \in V$, y el algoritmo termina.

2. Caso de Estudio: Rutas más cortas en la METROVIA de Guayaquil

2.1. Introducción

En esta sección se describe el Sistema de Transporte Masivo de Guayaquil y se establecen los parámetros fundamentales que servirán de base para formular matemáticamente el problema de determinar las rutas más cortas al interior de la red de transporte y describir un algoritmo de solución. La información presentada en esta sección sobre la estructura del Sistema de Transporte METROVIA ha sido tomada de [5], documento que fue facilitado y elaborado por la Ilustre Municipalidad de Guayaquil.

El proyecto tiene como objetivo, desarrollar un sistema informático que sea utilizado a través del Internet para dar a conocer al usuario, los mecanismos más cortos de desplazamiento al interior de la red de transporte urbano, facilitando con esto que el pasajero pueda planificar mejor sus desplazamientos y obtener así un mayor provecho del sistema.

Sistemas similares de información se han venido utilizando en otros países desde hace algunos aos, por ejemplo la red de trenes de la *Deutche Bahn* en Alemania utiliza una pagina Web para que los turistas y ciudadanos en general planifiquen sus viajes desde su hogar.

El presente proyecto de investigación persigue tres objetivos concretos:

1. El desarrollo de un modelo matemático y su implementación mediante un al-

goritmo construido en C++ para resolver el problema de determinar las rutas óptimas de desplazamiento al interior del sistema de transporte masivo urbano. La contribución del proyecto desde el punto de vista científico es la aplicación de un algoritmo de optimización para determinar las rutas más cortas con restricciones de recursos $(SPPRC)$, el cual es modificado y adaptado para nuestro caso de estudio (Sistema de Transporte METROVIA).

El algoritmo tomará como entradas los siguientes datos del usuario:

- Origen de desplazamiento
- Destino de desplazamiento
- Fecha del viaje
- Hora de salida

2. El diseño e implementación de una base de datos en MYSQL, que almacene la estructura del sistema de transporte METROVIA.
3. El diseño e implementación de una página Web, para efectuar las consultas de los viajes al interior del sistema METROVIA.

Cabe recalcar que la implementación del sistema informático ha sido realizado bajo una plataforma de código abierto (sin costos de licencias), donde se conjugan los siguientes herramientas: PHP, C++, MYSQL, MYSQLADMIN, LINUX.

En la Figura 2.1 se presenta el esquema de funcionamiento y los componentes antes mencionados.

La interfaz gráfica para el usuario ha sido desarrollada en el lenguaje PHP, para las consultas de los viajes al interior del sistema METROVIA a través de una página Web. Esta página recibe los parámetros de consulta (estación llegada, salida, etc.), los cuales son enviados a un programa desarrollado en C++ que reside en el servidor. Este programa determina las rutas más cortas mediante un algoritmo de

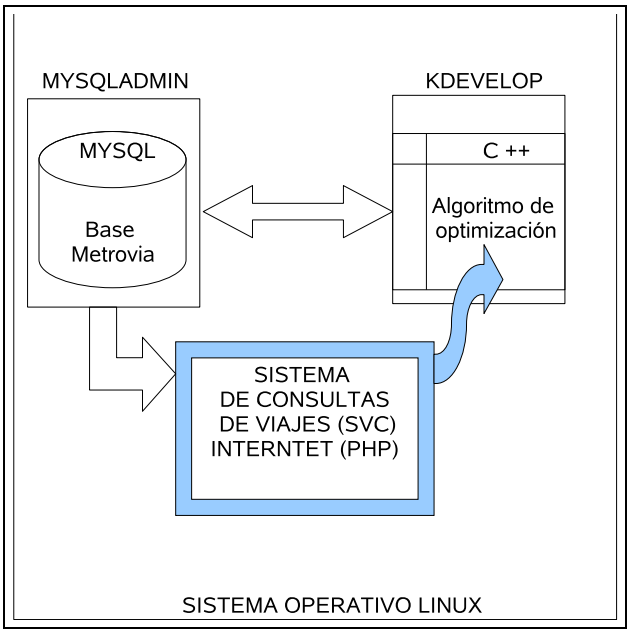


Figura 2.1: Esquema de funcionamiento del sistema informático

optimización, para el cálculo el algoritmo se consulta además información a una base de datos diseñada e implementada en MYSQL que contiene la estructura del sistema de transporte (frecuencia de las lneas, etc.). Una vez encotrada la solución, el algoritmo almacena la respuesta en la base de datos. Finalmente la base es consultada por la aplicación vía Web y se genera una tabla donde el usuario puede observar sus allternativas de desplazamiento, las mismas que consiten de líneas y paradas con sus respectivos tiempos de llegadas que debe conciderar para la fecha planificada de su viaje.

2.2. El Sistema de Transporte Masivo de Pasajeros de Guayaquil

El Plan de Racionalización del Transporte Público Masivo de la ciudad de Guayaquil ha sido concebido por el Municipio con un horizonte de 20 años (2000-2020), período en el cual se implementarán siete troncales de transporte con sus respectivas líneas alimentadoras. Durante este período tendrán que coexistir las dos modalidades de transporte urbano: la convencional y la Metrovía. La red actual de servicios convencionales irá reduciéndose en la medida que se vayan incorporando al Sistema Integrado. En la Figura 2.2 se presenta la configuración de las siete troncales planificadas para la red de transporte.

La primera fase del Sistema Integrado consiste en la implementación de tres troncales, las cuales han sido la base de partida para el desarrollo del prototipo de nuestro sistema informático para consultas de viajes.

2.2.1. Primera fase de Sistema Integrado

La primera fase del sistema está constituida por tres líneas troncales, cuyos buses circularán en carriles para su uso exclusivo, es decir, separados del resto del tráfico. Estos carriles permitirán que las personas que viajan en bus tengan preferencia en la circulación y por lo tanto lleguen más rápido a su destino.

A lo largo de la vía existirán paradas que serán cerradas y cubiertas para proteger a los pasajeros del sol y de la lluvia. Para ingresar a estas paradas las personas primero deberán cancelar su pasaje y luego estar listos para abordar su bus en forma cómoda y rápida. Las personas por lo tanto no tendrán que pagar al chofer del bus, sino a su entrada a la parada. Estas paradas se denominan de preembarque.

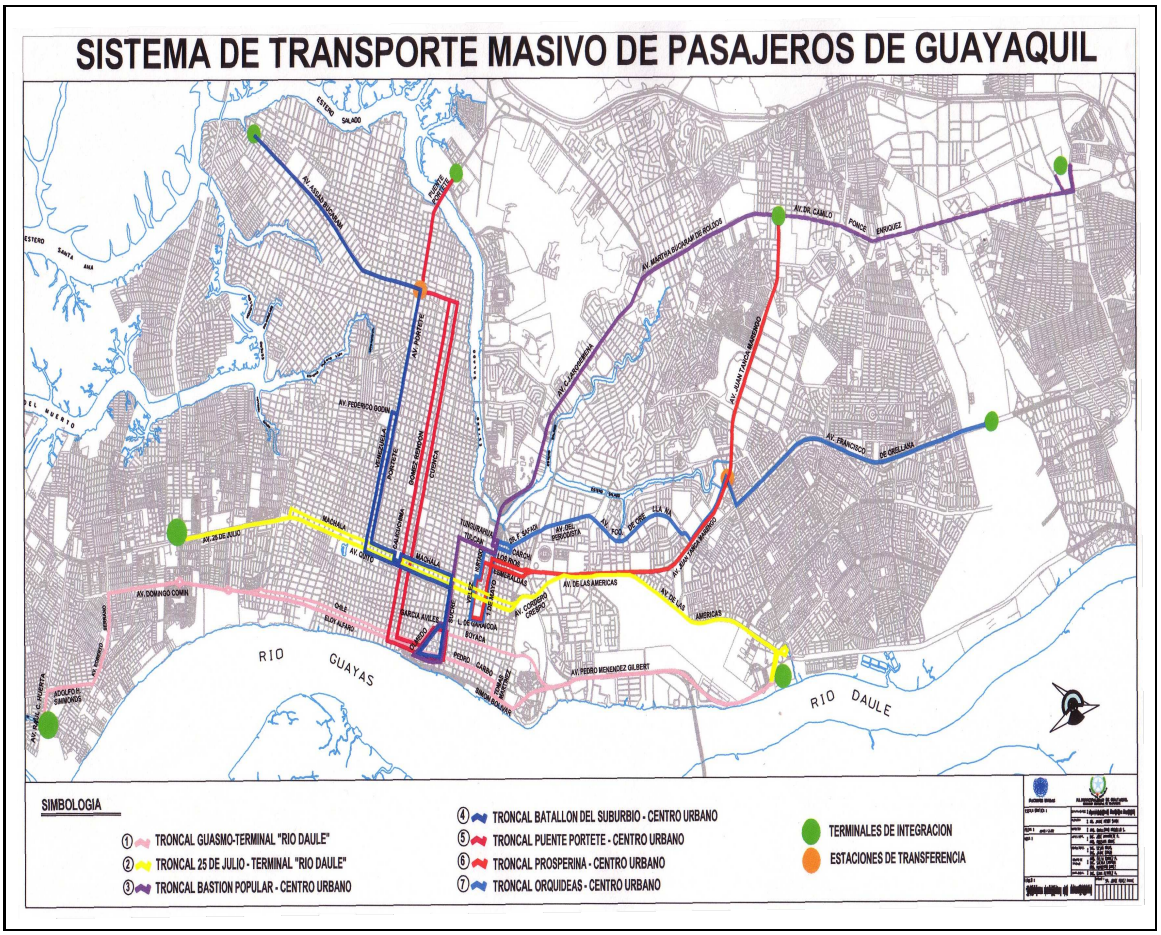


Figura 2.2: Esquema de las siete troncales

En los extremos se construirán grandes estaciones terminales (llamadas Terminal) a donde llegarán todos lo buses tanto de la línea troncal, como ”buses alimentadores”, que serán buses comunes encargados de transportar a los usuarios entre las estaciones terminales y los barrios aledaños. Las personas podrán realizar trasbordos dentro de la terminal, sin pagar otro pasaje.

Se ha dividido las 24 horas del día en fajas horarias y se le ha dado una denominación (P1=hora pico en la maana, M=maana, P2=hora pico en la tarde, etc., ver la Tabla 2.2), por ejemplo la faja P1 es definida de 6:00 am a 9:00 am.

Primera Troncal ”Terminal El Guasmo - Terminal Río Daule”

La primera línea troncal o primer corredor a operar corresponde a la línea que se extenderá entre las terminales de transferencia El Guasmo y Río Daule (Ver Figura 2.2), con una longitud aproximada (ida y vuelta) de 31,65 kilómetros, en los que se implantarán 28 paradas de preembarque, distanciadas aproximadamente 400 metros entre sí. En la Tabla 2.1 se presenta la ruta o recorrido de la Troncal 1.

En la Troncal 1 el tiempo total del recorrido de ida y vuelta de un bus articulado es ha estimado en 91 minutos, y como existen 45 paradas a lo largo de la linea podemos asumir el tiempo promedio de recorrido entre paradas será de 2.02 minutos (sin considerar el tiempo de espera en cada estación). La Tabla 2.2 se presentan otros datos operacionales de la troncal, específicamente los tiempos entre llegadas de los buses articulado por frajas de horario.

En la Tabla 2.2 se tiene información del tiempo entre llegada por fajas de horario, por ejemplo tiempo entre llegadas de buses es de 1.9 minutos, siempre que se tome el bus de 6:00 a 9:00 am y su denominación es P1. La Tabla 2.3 contiene una descripción del recorrido, en orden de paradas.

En la troncal 1 funcionan dos líneas las cuales se denominan Diametral, la una tiene el recorrido de Norte-Sur y la otra de Sur-Norte, cabe notar que son los mismos articulados que satisfacen la demanda de usuarios en las dos líneas. Como se observa en la Figura 2.3 las paradas o estaciones se clasifican por tipo simple en el caso de que pertenesca a una vía unidireccional, y doble cuando es bidireccional en este caso esta estación es usada para las dos lineas. Cuando una linea biene de estaciones simples y llega a una estación de tipo doble esta se cambia de tipo y se denomina de Integración. Finalmente tenemos dos tipos de parada, la de transferencia que son utilizadas para cambiar de una troncal a otra (por ejemplo la parada

Recorrido Troncal 1	
IDA	REGRESO
Terminal El Guasmo	Terminal Río Daule
Avenida Las Esclusas	Benjamín Rosales
Av. Adolfo Simonds	Avenida P. Menéndez
Av. Serrano	Túnel, bajo el cerro del Carmen
Domingo Comín	Calle Boyacá
Eloy Alfaro	Av. Olmedo
Pedro Carbo	Calle Chile
Rocafuerte	Rosa Borja de Icaza
Tomás Martínez	Avenidas Domingo Comín
Malecón Simón Bolívar	Av. Serrano
Túnel bajo el cerro Santa Ana	Av. Adolfo Simonds
Pedro Menéndez	Avenida Las Esclusas
Benjamín Rosales	Terminal de El Guasmo
Terminal Río Daule	

Tabla 2.1: Recorrido de la Troncal 1 por vías transitadas en la ciudad

Datos Operacionales						
Faja	6-9	9-12	12-15	15-17	17-20	20-00
Denominación	P1	M	P2	T	P3	N
Tiempo	1.9	2.8	3.2	2.8	1.9	4.2

Tabla 2.2: Tiempo de operaciones de la Troncal 1

Recorrido Troncal 1 NORTE-SUR	
NOMBRE	TIPO
Terminal Río Daule	Terminal
Las Banderas	Doble
Liceo Naval	Doble
Fae	Doble
Atarazana	Integración
Hosp. Luis Vernaza	Simple
Boca Nueve	Simple
Catedral	Simple
Av. Olmedo	Transferencia
La Providencia	Simple
El Astillero	Simple
Hosp.Leon Becerra	Simple
Barrio Centenario	Simple
Barrio Cuba	Simple
Caraguay	Doble
Cdla. 9 de Octubre	Doble
Pradera 1	Doble
Pradera 2	Doble
Tulipanes	Doble
Guasmo Central	Doble
Floresta 1	Doble
Floresta 2	Doble
Guasmo Norte	Simple
Guasmo Sur	Doble
Terminal Guasmo	Terminal

Recorrido Troncal 1 SUR -NORTE	
NOMBRE	TIPO
Terminal Guasmo	Terminal
Guasmo Sur	Doble
Guasmo Norte	Simple
Floresta 2	Doble
Floresta 1	Doble
Guasmo Central	Doble
Tulipanes	Doble
Pradera 2	Doble
Pradera 1	Doble
Cdla. 9 de Octubre	Doble
Caraguay	Doble
Barrio Cuba	Simple
Barrio Centenario	Simple
Hosp.Leon Becerra	Simple
El Astillero	Simple
La Providencia	Simple
Plaza de Integración	Simple
Biblioteca	Simple
Jardines del Malecon	Simple
Las Peñas	Simple
Atarazana	Integración
Liceo Naval	Doble
Las Banderas	Doble
Terminal Río Daule	Terminal

Tabla 2.3: Recorrido por paradas de la Troncal 1

Atarazana es una estación de transferencia, por que permite que los usuarios de la troncal 3 pasen a la troncal 1) y las paradas denominadas Terminal, estas son en donde inician o finalizan los recorridos de las dos líneas dentro de la troncal y llegan los buses alimentadores.

Segunda troncal ”Terminal 25 de Julio Terminal Río Daule”

La segunda troncal se denomina Terminal 25 de Julio - Terminal Río Daule, porque parte desde el intercambiador de tráfico, ubicado en la intersección de la vía perimetral y Avenida 25 de Julio. La ruta sigue las Avenidas 25 de Julio, Av. Quito, Avenida de las Américas para unirse con la Terminal Río Daule, ubicada al frente de la terminal terrestre de pasajeros.

En el retorno, los buses saldrán desde la Terminal Río Daule hacia la Avenida de las Américas en dirección sur siguiendo su recorrido por la Avenida Machala hasta empatar con la Avenida 25 de Julio y de allí hasta la Terminal del mismo nombre.

Esta troncal tiene una extensión aproximada (ida y vuelta) de 26,30 kilómetros con 34 paradas de pasajeros y 73 intersecciones semaforizadas en todo su recorrido.

En la Troncal 2 el tiempo total del recorrido de un bus articulado de ida y vuelta es de 77 minutos, y dado que existen 50 paradas, podemos estimar el tiempo promedio de recorrido entre paradas en 1.54 minutos. La Tabla 2.4 presenta otros datos de operacionales relevantes y la Tabla 2.5 describe las paradas visitadas a los largo del recorrido.

Datos Operacionales						
Faja	6-9	9-12	12-15	15-17	17-20	20-00
Denominación	P1	M	P2	T	P3	N
Tiempo	0.8	1.0	1.0	1.0	0.8	1.4

Tabla 2.4: Tiempo de operaciones de la Troncal 2

Recorrido Troncal 2 SUR -NORTE	
NOMBRE	TIPO
Terminal 25 de Julio	Doble
La Pradera	Doble
IESS	Doble
Los Almendros	Doble
Las Acacias	Doble
Francisco Segura	Simple
Chavez Franco	Simple
Plaza de Artes	Simple
Venezuela	Simple
Capwell	Simple
Hosp. Niño	Simple
Mercados	Simple
Victoria	Simple
9 de Octubre	Simple
M.A.G.	Doble
Cementerio	Doble
Cuartel Modelo	Doble
Plaza Dañin	Doble
Juan Tanca Marengo	Doble
Aeropuerto	Doble
Simón Bolivar	Doble
Terminal Río Daule	Terminal

Recorrido Troncal 2 NORTE-SUR	
Terminal Río Daule	Terminal
Aeropuerto Internacional	Doble
Simón Bolivar	Doble
Aeropuerto	Doble
Juan Tanca Marengo	Doble
Plaza Dañin	Doble
Cuartel Modelo	Doble
Cementerio	Doble
M.A.G.	Doble
9 de Octubre	Simple
Victoria	Simple
Mercados	Simple
Hosp. Niño	Simple
Capwell	Simple
Venezuela	Simple
Plaza de Artes	Simple
Francisco Segura	Simple
25 de Julio	Doble
Las Acacias	Doble
Los Almendros	Doble
IESS	Doble
La Pradera	Doble
Terminal 25 de Julio	Doble

Tabla 2.5: Recorrido por paradas de la Troncal 2

Tercera troncal ”Terminal Bastión Popular - Centro”

La tercera troncal tiene una extensión aproximada de 31,14 kilómetros. Parte del sector del Mercado de Víveres en la Vía a Daule (Camilo Ponce Enríquez) hacia el centro de la ciudad. El recorrido de vías que transita la troncal 3 está indicado en la Tabla 2.6, y en la Tabla 2.7 se presentan las paradas que visita está troncal. A lo largo de la misma se presentan 42 intersecciones semaforizadas.

Recorrido Troncal 3	
IDA	REGRESO
Terminal Bastión Popular	Avenida Olmedo
Av. Camilo Ponce	Sucre
Av. Juan Tanca Marengo	Tulcán
Av. Martha Bucaram de Rold	Primero de mayo
Av. Carlos Julio Arosemena	Av. 9 de Octubre
Puente 5 de Junio	Puente 5 de Junio
Av. Nueve de Octubre	Av. Carlos Julio Arosemena
Calle Tulc	Kilómetro 41/2 de la Vía a Daule
Calle Sucre	Av. Martha Bucaram
Av. Olmedo	Terminal Bastión Popular

Tabla 2.6: Recorrido de la Troncal 3 por vías transitadas en la ciudad

En la estación Av. Olmedo las personas podrán hacer transferencias hacia la Troncal número uno para desplazarse hacia los sectores del Guasmo y la Terminal Terrestre sin necesidad de pagar otro pasaje.

El tiempo total del recorrido de un bus articulado de ida y vuelta es de 90 minutos, con 49 paradas y un tiempo promedio de recorrido entre paradas de 1.84 minutos.

Recorrido Troncal 3 NORTE-CENTRO	
NOMBRE	TIPO
Terminal Bastión	Terminal
California	Doble
Bastión Popular	Simple
La Filantropica	Doble
Fuerte Huancavilca	Doble
La Florida	Doble
Academia Naval	Doble
Colegio Americano	Doble
Prosperina	Doble
Dolores Sucre	Doble
Cerros de Mapasingue	Doble
Mapasingue	Doble
Los Ceibos	Doble
Federación de Guayas	Doble
28 de Mayo	Doble
Vista Grande	Doble
Bellavista	Doble
Universidad Católica	Doble
Ferroviaria	Doble
Barrio Orellana	Doble
Vicente Rocafuerte	Doble
Los Rios	Doble
La Victoria	Doble
Mercado Central	Doble

Recorrido Troncal 3 CENTRO-NORTE	
Av. Olmedo	Transferencia
Biblioteca	Simple
El Castillo	Simple
Mercado Central	Doble
La Victoria	Doble
Los Rios	Doble
Vicente Rocafuerte	Doble
Barrio Orellana	Doble
Ferroviaria	Doble
Universidad Católica	Doble
Bellavista	Doble
Vista Grande	Doble
28 de Mayo	Doble
Federación de Guayas	Doble
Los Ceibos	Doble
Mapasingue	Doble
Cerros de Mapasingue	Doble
Dolores Sucre	Doble
Prosperina	Doble
Colegio Americano	Doble
Academia Naval	Doble
La Florida	Doble
Fuerte Huancavilca	Doble
La Filantrópica	Doble
Bastión Popular	Simple
California	Doble
Terminal Bastión	Terminal

Tabla 2.7: Recorrido de la Troncal 3 por orden de paradas

Datos Operacionales						
Faja	6-9	9-12	12-15	15-17	17-20	20-00
Denominación	P1	M	P2	T	P3	N
Tiempo	1.0	1.2	1.2	1.2	1.0	1.6

Tabla 2.8: Tiempo de operaciones de la Troncal 3

3. Algoritmo de Solución

3.1. Planteamiento general

La red de transporte urbano masivo de Guayaquil, puede ser representada mediante un grafo dirigido $D = (V, A)$, donde los elementos de V son las paradas de las troncales y alimentadoras y los elementos de A representan los enlaces de transporte que conectan a las paradas(un nodo para cada sentido de circulación.

El problema de determinar las rutas más cortas al interior del sistema de transporte Metrovia puede formularse como un problema de caminos más cortos en D, considerando una restricción adicional asociada a las líneas de transporte: si se llega a un nodo intermedio v empleando un arco correspondiente a una línea l, entonces para salir de v debe emplearse también un arco de A^l, o pagar un costo adicional que simboliza el tiempo requerido para el cambio de línea.

Esta nueva restricción puede modelazarse al crear $\mid L \mid$ copias del conjunto de nodos y definir un digrafo distinto $D^l = (V^l, A^l)$ para cada línea $l \in L$. Conectamos finalmente todas las $"copias"$ de una parada entre sí con arcos cuyos costos indican los tiempos de espera para cambiar de línea. Obtenemos así un nuevo grafo $\hat{D} := (\biguplus_{l \in L} V^l, \biguplus_{l \in L} A^l \uplus B)$, donde B es el conjunto de todos los arcos de $"cambio\ de\ l\acute{i}neas"$. El problema queda reducido a un problema (clásico de caminos más cortos en $\hat{D}$.

Ejemplo:

La Figura 3.1 nos da una ilustración de como modelar el problema de tener un

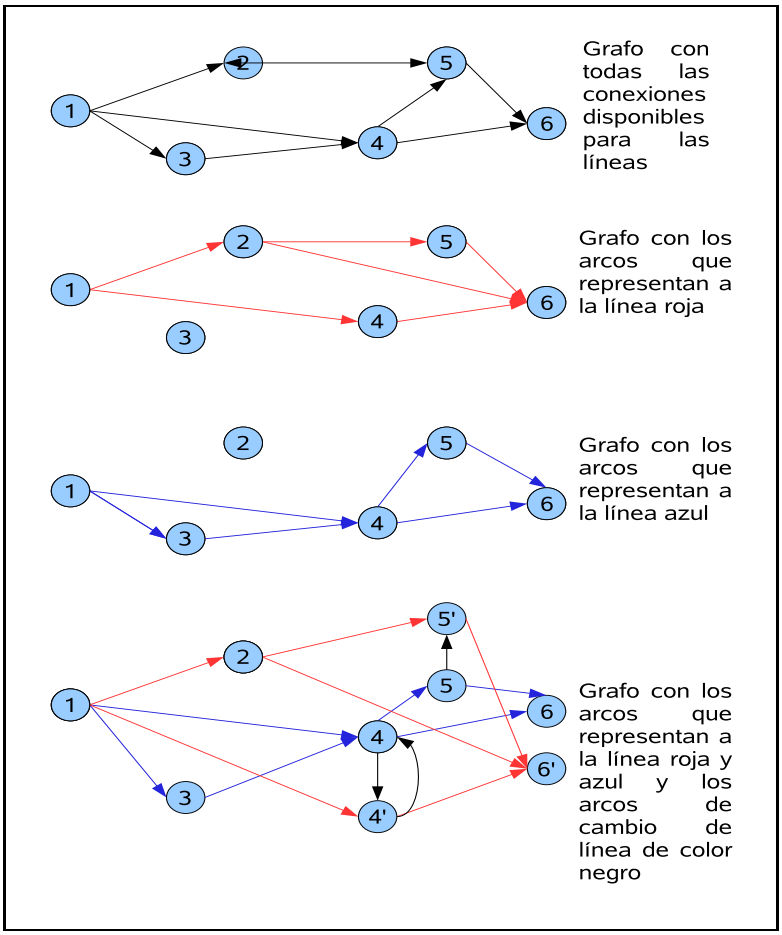


Figura 3.1: Digrafo que representa todos los posibles cambios de línea

sistema de transporte con seis estaciones y dos líneas, el cual es representado mediante un grafo. Como se puede observar tenemos que por las estaciones 4 y 5 pasan las dos líneas y las dos llegan a la estación seis. En el momento de cambiar de línea se toma un arco de color negro cuyo costo es el tiempo de espera de un bus de la otra línea.

El algoritmo de solución que hemos desarrollado no trabaja sobre el grafo $\hat{D}$ explícitamente sino que emplea la idea de etiquetas en los nodos introducida por los algoritmos para el SPPRC. Para $v \in V$, la etiqueta (c_κ, l_κ) registra el costo c_κ(es decir, el tiempo total de recorrido) de un camino desde el nodo de partida hasta v

y la línea l_κ a la que pertenece el último arco de este camino. El algoritmo opera entonces en una forma similar a la de los algoritmos de fijación de etiquetas (o al algoritmo de Dijkstra para el SPP clásico); tal como se describe a continaución:

Algoritmo de optimización

{Inicialización}

Ingresar s, t;

Crear cola de prioridad P;

Generar etiquetas de los sucesores de t y guardar etiquetas en P;

mientras Exista etiquetas en P o llegar a procesar el nodo t **hacer**

  Seleccionamos una etiqueta de P mediante la operación top, para la cual se tiene que sea mínima(la de menor tiempo de viaje);

  {**Procesar la etiqueta**}

  para un sucesor del nodo de la etiqueta seleccionada **hacer**

    si hay cambio de línea **entonces**

      $tiempo_{viaje_j} := tiempo_{espera} + tiempo_{viaje_{j-1}} + tiempo_{viaje}$;

    fin si

    $tiempo_{viaje_j} := tiempo_{viaje_{j-1}} + tiempo_{viaje}$;

    Guardar la etiqueta del nodo sucesor en P

  fin para

  Eliminar etiqueta seleccionada de P

fin mientras

Las etiquetas se mantienen en una cola de prioridad P, donde los elementos están ordenados de menor a mayor tiempo de viaje. Al realizar la operación Top extrae de la cola la etiqueta con menor tiempo de entre todas las etiquetas generadas para todos los nodos. El algoritmo termina cuando se llega al nodo s (nodo de

llegada) o cuando la cola de prioridad esta vacía, en cuyo caso no existe una ruta factible en el sistema para desplazarse desde t hasta s.

Al implementar un algoritmo de este tipo, mantener en la memoria principal del computador la estructura completa del grafo tendría un impacto negativo sobre la eficiencia del cálculo. Por esta razón se ha diseñado una base de datos para almacenar la configuración de la red METROVIA. Esta base es consultada durante la ejecución del algoritmo, con lo que se consigue mantener en la memoria únicamente los nodos(paradas) y arcos(vías) necesarios para la resolución de la instancia actual.

3.2. Detalles de implementación

Se ha implementado un sistema de información de consultas de viajes al interior del sistema de transporte METROVIA, el mismo que consiste de tres módulos:

- Una interfaz para el usuario, desarrollada en PHP, desde donde se pueden realizar las consultas de viajes a través de un formulario de Internet. La dirección de la página Web para realizar las consultas es www.math.epn.edu.ec$\backslash \sim$mflores.
- Una base de datos en `MySQL` que contiene la información correspondiente a la estructura de la METROVIA (líneas, paradas, frecuencias, etc.)
- Una aplicación en `C++` para realizar el cálculo de las rutas más cortas utilizando el algoritmo descrito en la sección anterior.

3.2.1. Interfaz del usuario

En este prototipo del sistema de información se tiene que ingresar un nombre de usuario y clave, los cuales son creados en la base de datos, en este caso se

tiene un usuario master con clave root, en la Figura 3.2 se muestra la página web de ingreso.

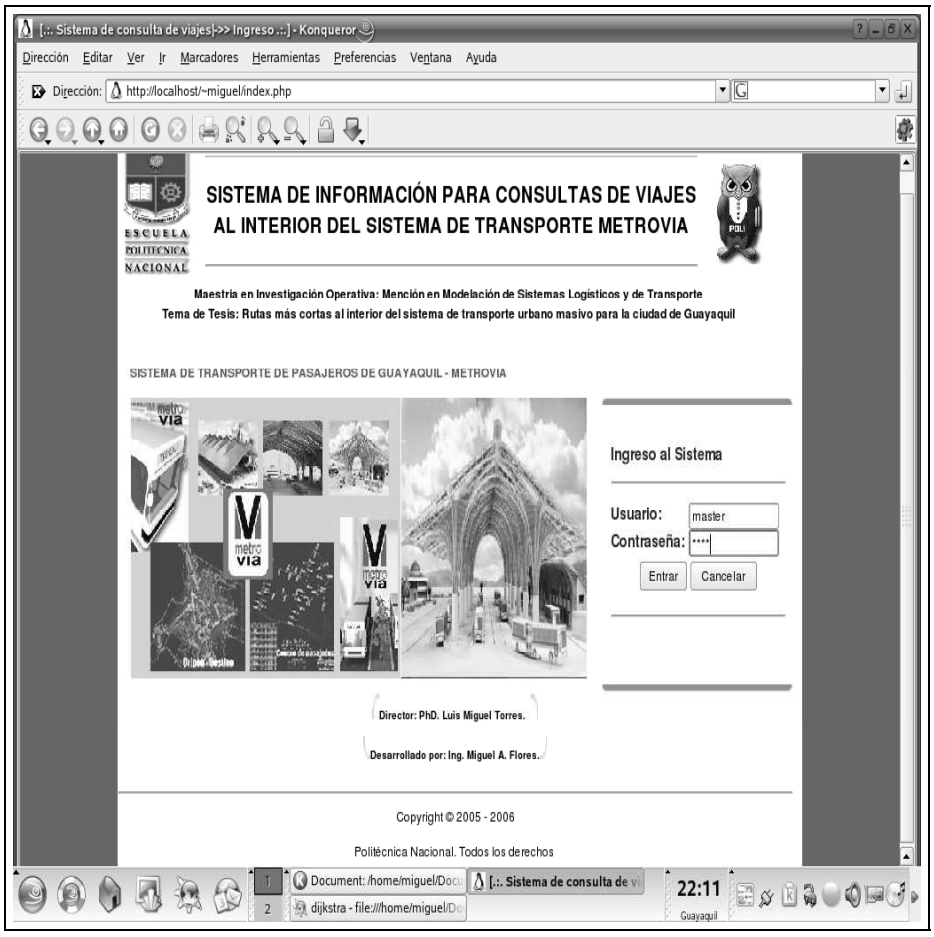


Figura 3.2: Página de Ingreso al sistema de información

El sistema de información tiene un menú que me permite navegar por tres páginas: una de información sobre el sistema, la otra para consultar los datos operacionales del sistema de transporte que están almacenados en la base, por último, la página de consultas de viajes. En la Figura 3.3 se presenta la pagina Home, que contiene información sobre el propósito del sistema de consultas de viajes (SCV).

La Figura 3.4 presenta la página de consultas a la base, desde esta página se tiene acceso a la base de datos para consultar datos concernientes a la estructura

Figura 3.3: Página Home del sistema de información

de la METROVIA: líneas, las tablas de fajas, paradas, etc.

Por ejemplo, al hacer clic en el botón que esta junto a la etiqueta ”tabla fajas ”se genera una consulta que recupera los registros de esta tabla y los presenta en una página web (ver Figura 3.5).

La página pricipal del sistema de información es la que se muestra en la Figura 3.6, la cual es utilizada para el ingreso de los parámetros de viaje del usuario. En ésta se ingresa el origen y destino del viaje, así como la fecha y la hora del viaje.

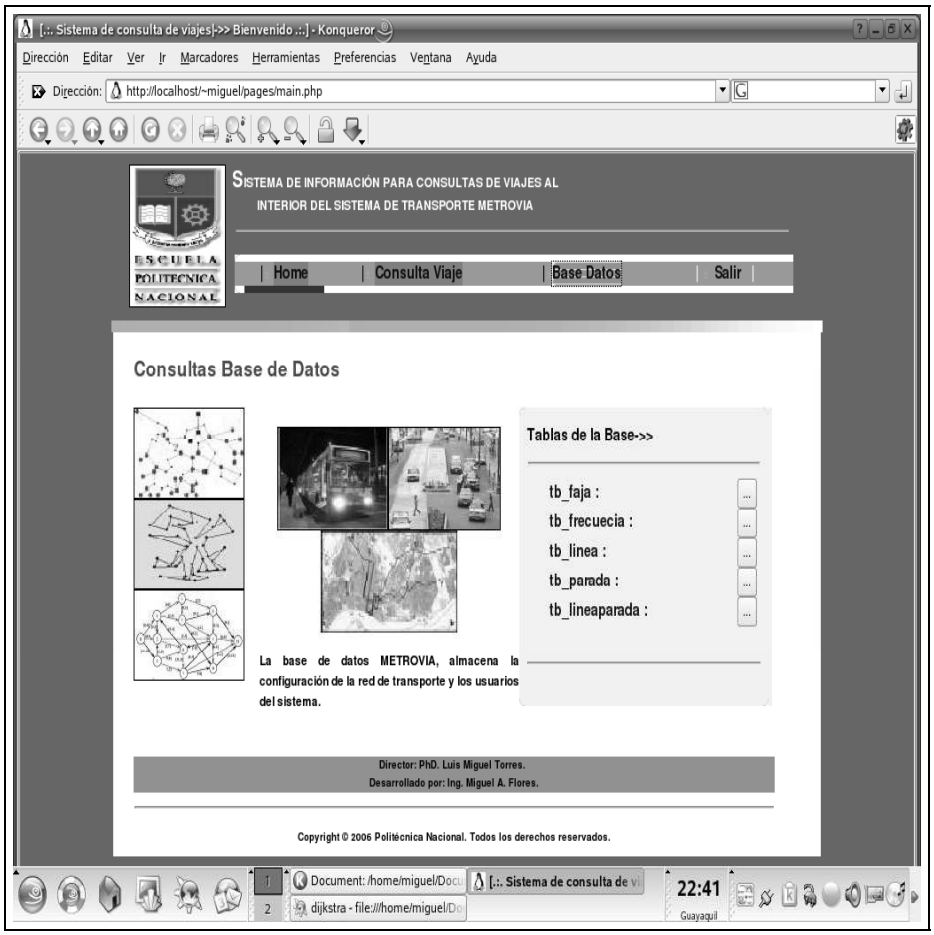


Figura 3.4: Página de consultas a la base de datos

3.2.2. Base de datos

La base de datos METROVIA que mantiene la configuración de la red de transporte ha sido implementada en `MYSQL` y para la administración de la misma se utiliza `mysqlcc (mysql control center)`, en la Figura 3.7 se presenta las tablas que contituyen la base de datos.

Se tiene una tabla fajas, en la que se almacena las fajas horarias, la cual está relacionada con la tabla frecuencia, donde se almacena el tiempo promedio que realiza una línea de parada a parada y la frecuencia con que llegan los buses en cada parada, por esta razón se tiene una relación con la tabla que mantiene las líneas,

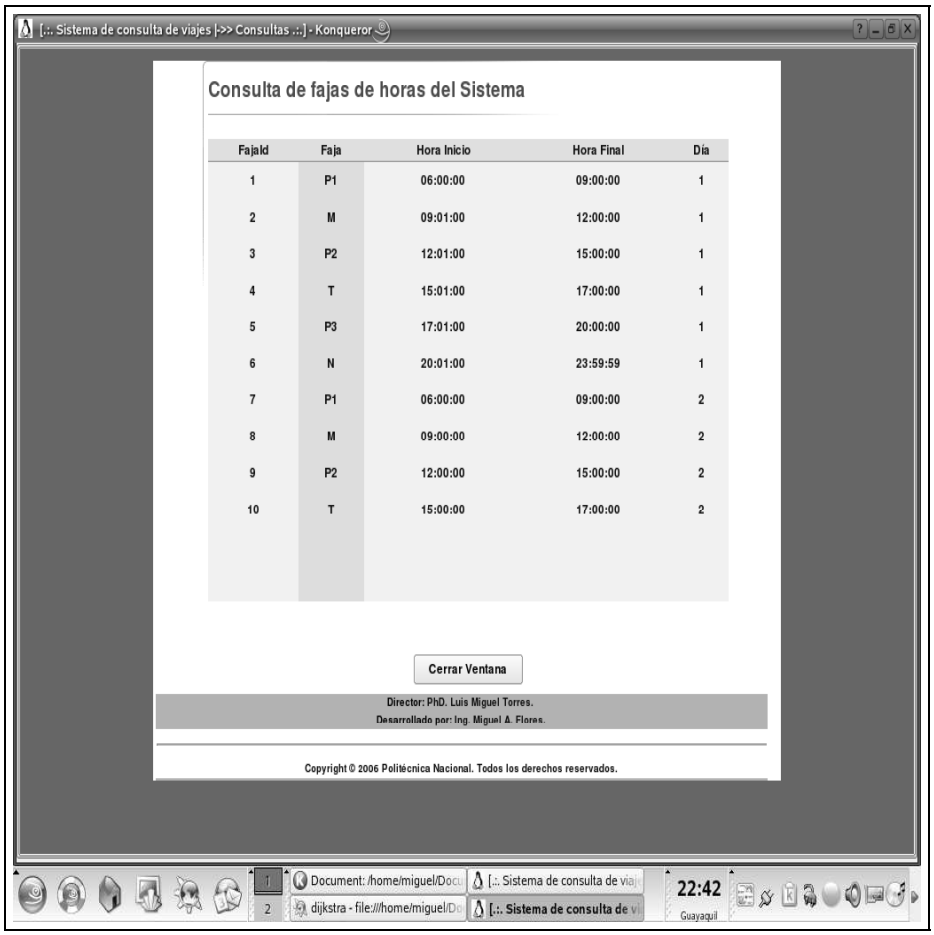


Figura 3.5: Página de consultas a la base de datos

asi en estas tres tablas se tiene por cada línea su frecuencia y tiempo de viaje por faja, pero también se creó la relación de las paradas con las líneas, para esto se ha tiene una tabla parada que almacena a todas las paradas del sistema de transporte, para poder hacer efectiva la relación línea a parada se tiene que construir una tabla linea-parada que este relacionada con la tabla parada y línea, en la Figura 3.8 se presenta las relaciones de la base de datos.

Para que el algoritmo de optimización realice consultas de la configuración de la red de transporte y almacene la respuesta a la solución de encontrar las rutas más cortas con restricciones de recurso, se ha utitilizado la librería `mysql++` para

Figura 3.6: Página de consultas de viajes

poder conectarnos a la base de datos y efectuar consultas y grabaciones de datos desde C++, en tiempo de ejecución del algoritmo. El primer paso es conectarse a la base de datos mediante la identificacíon del servidor donde se encuentra la base de datos, el nombre y contraseña del usuario, así como el nombre de la base de datos, para lo cual se utiliza el siguiente código.

```
Connection DB("www.math.epn.edu.ec","mflores","mflores","metrovia");
```

En nuestro caso nuestra base de datos se llama metrovia y se encuentra alojado en el servidor del departamento de Matemática de la EPN y se a creado un usuario

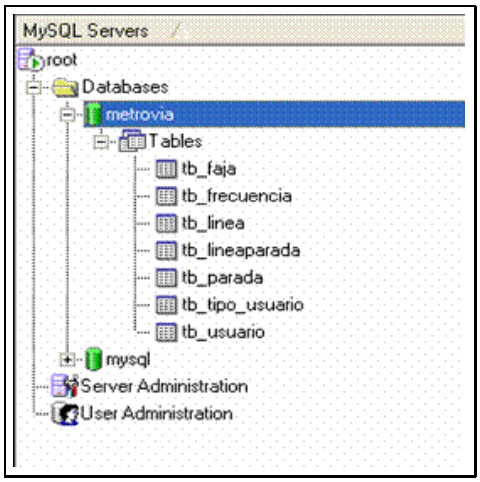


Figura 3.7: Implementación de la base en el motor de base de datos MySql

con nombre y contrasea mflores, con los unicos permisos de consultar y guardar datos en la base.

Una vez que se tiene la concección a la base, para esto primero hay que definir un script de consulta el cual es almacenado en un objeto de tipo Query, después se declara un objeto del tipo Result para almacenar los datos de la consulta, y para recorrer cada dato se utiliza un objeto del tipo Row, a continuación se presenta la sentencia sql utilizada para consultar las paradas del sistema METROVIA y la declaración de los objetos antes mencionados.

```
Query query=DB.query();
query<< "select ParadaId,Nombre from tb_parada";
Result res0=query.store();
Row row0;
```

El siguiente código es el que se utiliza para guardar los resultados del algoritmo en una tabla con nombre stock en la base de datos, especificamente se guarda las estación de salida, de llegada, la línea y el tiempo de viaje hasta la parada de llegada.

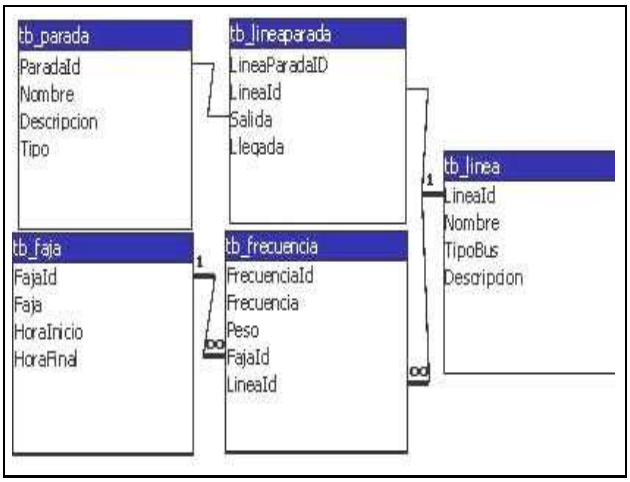


Figura 3.8: Esquema de la relación de las tablas que representan la estructura del sistema METROVIA

```
query << "insert into %5:table values (%0q, %1q, %2q, %3)";

query.def["table"] = "stock";
```

A continuación se presenta las propiedades de las tablas creadas para almacenar la configuración de la red de transporte Metrovía, así como sus relaciones en la base de datos METROVIA

Campo	Tipo	Nulo	Predeterminado
FajaId	int(11)	No	
Faja	char(2)	S	NULL
HoraInicio	time	S	NULL
HoraFinal	time	S	NULL
Dia	int(1)	S	NULL

Tabla 3.1: Tabla Faja

Campo	Tipo	Nulo	Predeterminado
FrecuenciaId	int(11)	No	
Frecuencia	float	S	NULL
Peso	float	S	NULL
FajaId	int(11)	S	NULL
LineaId	int(11)	S	NULL

Tabla 3.2: Tabla Frecuencia

Campo	Tipo	Nulo	Predeterminado
LineaId	int(11)	No	
Nombre	varchar(50)	S	NULL
TipoBus	varchar(20)	S	NULL
Descripcion	varchar(50)	S	NULL

Tabla 3.3: Tabla Línea

Campo	Tipo	Nulo	Predeterminado
LineaParadaId	int(11)	No	
LineaId	int(11)	S	NULL
Salida	int(11)	S	NULL
Llegada	int(11)	S	NULL

Tabla 3.4: Tabla LineaParada

Campo	Tipo	Nulo	Predeterminado
ParadaId	int(11)	No	
Nombre	varchar(100)	S	
Descripcion	varchar(50)	S	NULL
Tipo	varchar(50)	S	NULL

Tabla 3.5: Tabla Parada

3.2.3. Módulo de optimización

La implementación del algoritmo se realizó en C++, empleando el copilador de libre distribución GNUCC (versión 3.3.5), el cual forma parte de la distribuciones más comunes del sistema operativo Linux (En nuestro caso, trabajamos con Suse Linux versión 9.3). La comunicación con el módulo de interfaz del usuario se lleva a cabo por medio de parámetros de líneas de ejecución, mientras que para la comunicación con la base de datos se emplean fuciones de las bibliotecas mysql y mysql++. Debido a que estas bibliotecas no forman parte de las bibliotecas estándares del C++, es preciso su instalación y configuración en el sistema donde va ser ejecutada la aplicación.

Estructura de datos

Para la implementación del algoritmo de fijación de etiquetas se utilizaron clases de contenedor pertenecientes a la biblioteca STL (Standard Template Library), tales como, colas de prioridad (pqueue),listas (list), vectores (vector). Sedefinieron además clases específicas para almacenar la red, las cuales se presentan a continuación.

La representación computacional elegoda para el digrafo (que se genera dinámicamente durante la ejecución del algoritmo) es de listas de arcos (ver sección 1.3.1). Para esto se ha definido la clase digrafow, que esta conformada por la clase nodo y clase arco, descrito más adelante.

```
class digrafow
{
   public:
   int n,m;
```

```
   int id_mfinal, id_nfinal;
   vector<nodo> V;
   digrafow();
   ~digrafow();
   void  asignar (int n1);
   void  agregar_arco(int i, int j, double w, double f,int l);
   void  agregar_marca(int mid,int i, int j, int m, int l,double f);
};
```

Las varaibles miembro `n` y`m` almacenan el número de nodos y el número de arcos, respectivamente. El vector `V` contiene a los nodos del digrafo. Adicionalmente, la clase contiene cuatro funciones, entre ellas un destructor de la clase, una función que asigna la cantidad de nodos, y dos funciones que agregan marcas (etiquetas) en los nodos. Finalmente, al finalizar el algoritmo se guarda en las variables `id_mfinal`, `id_nfinal` información para la rescontrucción del camino más corto, en la sección de cálculo de viajes se da una mejor explicación.

A continuación se presentan las clases arco y nodo.

```
class arco
{
 public:
    int inicio;
    int final;
    double peso;
    double frecuencia;
    int linea;
    arco(int x, int y, double w, double f,int l)
```

```
    {
      inicio=x;
      final=y;
      peso=w;
      frecuencia=f;
      linea=l;
     }
  };
```

En la clase arco se define los nodos(estaciones) inicio y final, así como el costo (tiempo de viaje entre estaciones), la frecuenca(tiempo mínimo de espera hasta que llegue una línea) y una función constructora de la clase.

```
class nodo
{
 public:
    int id;
    list <arco> L;
    list <marca> M;
  };
```

Para cada nodo, se mantiene una lista de arcos adyacentes y una lista de marcas(etiquetas) que se generan durante la ejecución del algoritmo. También se ha definido la clase marca, que consta del nodo al que pertenece la etiqueta, el nodo y marca antecesores, así como la línea con la que se llega al nodo procesado. Esta información nos permitirá construir la ruta más corta cuando finalice el cálculo. Por último, la variable `t` registra el tiempo de viaje acumulado hasta el momento.

```
class marca
{
  public:
    int marcaid;
    int nodo_procesado;
    int nodo_antecesor;
    int marca_antecesor;
    int linea;
    double t ;
    marca(int ide=-1,int ma=-1,int no=-1,int mara=-1,int l=-1,
    double f=10000)
    {
     marcaid=ide;
     nodo_procesado=ma;
     nodo_antecesor=no;
     marca_antecesor=mara;
     linea=l;
     t=f;
    }
};
```

Para administrar la cola de prioridad se ha definido una clase order que compara dos marcas, donde el criterio de ordenamiento es el tiempo de viaje, es decir la marca que tiene menor viaje de tiempo es la más eficiente.

```
class order
{
```

```
public:
  bool operator () (marca a, marca b)
   {
     return   (a.t > b.t);
    }
};
```

Cálculo de viajes

Para la explicación del cálculo de viajes, se hara una ilustración mediante un ejemplo. El cual consiste en determianr la ruta más corta entre dos estaciones, donde el origen del viaje es El Terminal 25 de julio y destino o estación de llegada es El Terminal Río Daule, el viaje esta planificado para realizarce a las 15:45 horas del 14/06/06. En la Figura 3.9 se presenta la pagína de consultas de viajes.

Los datos de desplazamiento al momento de realizar clic en el boton aceptar, son transmitidos a un servidor Web que está alojado en la EPN, donde también se encuentra el algoritmo de optimización. El código para enviar los datos vía Web se presenta a continuaación.

```
$dia=$HTTP_GET_VARS['dia'];
$salida = $HTTP_GET_VARS['salida'];
$llegada = $HTTP_GET_VARS['llegada'];
$hora=$HTTP_GET_VARS['hora'];


system("./dijkstra $salida $llegada $dia $hora");
```

La instrucción system, al ejecutarse se escibe una linea de comando en el shell del servidor que ejecuta el algoritmo de optimización con los parámetros de entrada,

Figura 3.9: Página de consultas de viajes

que en este caso son los datos de desplazamiento, el código utilizado para recibir estos datos en C++ es el siguiente.

```
s=atoi(argv[1]);
r=atoi(argv[2]);
d=atoi(argv[3]);
Time dtime =argv[4];
```

Todos los datos son convrtidos en tipos de datos que son manejados en C++, para poder realizar el cálculo de la ruta más corta desde el origen al destino del viaje. El algoritmo interactúa con la base de datos metrovia para generar los sucesores de

cada nodo procesado, para esto se han utilizado las siguientes lineas de código.

```
Query query2=DB.query();
query2 << "select f.frecuencia,f.Peso,f.LineaId,lp.Llegada"
       <<" from tb_frecuencia f,tb_faja fa,tb_lineaparada lp"
       <<" where f.LineaId=lp.LineaId and f.FajaId=fa.FajaId"
       <<" and fa.HoraInicio<="
       <<"'"
       <<dtime
       <<"'"
       <<" and fa.HoraFinal>="
       <<"'"
       <<dtime
       <<"'"
       <<" and Dia="
       <<d
       <<" and lp.Salida= "
       <<nodoid;
       Result res2=query2.store();
       Row row2;
```

Se consulta la frecuencia de las líneas dependiendo de la hora del viaje la misma que depende de la faja de horario, el valor f.frecuenca es almacenado en la variable f, y f.Peso que representa el tiempo de viaje entre estaciones es almacenado en la variable w y la línea que brinda el servicio es obtenido por el campo LineaId y almacenado en la variable l. Finalmente el campo lp. Llegada genera las estaciones(nodos) sucesores. En la primera iteración se tiene como nodo de partida a

s(estación de origen del viaje), el cual es almacenado en la variable nodoid, esta variable ira tomano valores correspondientes a los nodos a ser procesados y terminara este proceso cuando tome el valor de r(estación de destino del viaje). A continuación se presenta el código generado en C++ que resuelve el problema de calcular las rutas más cortas con restricciones de recurso al interior del sistema METROVIA.

```
if (nodoid==s)
   {
    //hora de salida esperando un tiempo f
    //(iniciamos el recorrido en el instante f y recorremos w)
     tiempo=f+w;

    //(elaboro una marca para el nodo fin)
     G.agregar_marca (k,fin, ini,0, l,tiempo);

    //creo una copia de la marca elaborada del nodo final
     marca ma(k,fin, ini, 0, l,tiempo);

    //agrego en la cola de prioridad la copia de la marca
     pq.push(ma);

    }
else
    {
     //verificar marca en el nodo fin por medio de la linea
     //(no hay linea, ya linea menor o mayor en tiempo)
```

```
if (la==l)//considera cambio de lineas
   tiempo=tiempo+w;//hora de llegada al nodo fin
else
   tiempo=tiempo+w+f;//hora de llegada al nodo fin

   list<marca>::iterator em=G.V[claves[fin]].M.begin();
   int leti;//linea etiquetada
   //tiempo de la marca con la linea que se repite
   double tiempom;
   leti=0;

   while ((leti!=l) & ( em!=G.V[claves[fin]].M.end()))
      {
        leti=em->linea;
        tiempom=em->t;
        em++;
      }
   if (em!=G.V[claves[fin]].M.end())
     {
       if (tiempo<tiempom)
         {
           em->t=tiempo;
           //creo una copia de la marca elaborada del
           marca ma(k,fin, ini, mai, l,tiempo);
           //agrego en la cola de prioridad la copia de la marca
           pq.push(ma);
```

```
            }
         }
        else
         {
           if (leti!=l)
             {
              //(elaboro una marca para el nodo fin)
               G.agregar_marca (k,fin, ini,mai, l,tiempo);
               //creo una copia de la marca elaborada del
               marca ma(k,fin, ini, mai, l,tiempo);
              //agrego en la cola de prioridad la copia de la marca
               pq.push(ma);
             }
          }
}
```

Una vez que se termina el cálculo del viaje más optimo, se envía a la base metrovia y se genera dínamicamente una página Web que presenta las paradas, líneas y tiempo de viajes(Ver Figura 3.10).

El resultado de la planificación del viaje para nuestro ejemplo que era realizar un viaje de una terminal a otra, que pertenercen a la troncal 2 del sistema METROVIA, es tomar la línea 1, la cual atraviesa todas las paradas de la troncal en un tiempo de 29.4 mínutos.

Figura 3.10: Página de consultas de viajes

4. Pruebas Computacionales

En este capítulo se cosntruyen algunas instancias para poder analizar el tiempo de ejecución del algoritmo de optimización implementado para resolver el problema de rutas más cortas al interior del sistema METROVIA. Una instancia del problema viene definida básicamente por la estructura del digrafo, es decir número de nodos (paradas), arcos (vías) y además número de líneas. En la base de datos que mantiene la estructura de la red de transporte, se tiene una instancia del problema con 77 paradas que representan los nodos, 141 vías que conectan a las paradas que representan los arcos, y tres líneas.

Para realizar las pruebas computacionales, del tiempo de ejecución se ha utilizado una máquina con procesador intel centrino(versiíon de pentium para lapton), con 512 de RAM y sistema operativo Linux. En el caso de la instancia definida sólo por las tres troncales, el tiempo promedio de ejecución del algoritmo es de menos de 1 segundo. A continuación se presenta parte del codígo escrito para determinar el tiempo de corrida del algoritmo.

```
srand(time(0));
time_t t1=time(0);
//
Aqu se escribe el algoritmo de optimizacin
//
time_t t2=time(0);
double dif=difftime(t2,t1);
```

```
cout<<"Tiempo de corrida: "<<dif<<" segundos"<<endl;
```

Para utilizar el código anterior se debe definir la librería `time.h`. Como se puede ver se declara dos variables $t1, t2$ del tipo `time_t`. La variable $t1$ almacena el instante en que inicia la corrida del algoritmo mientras $t2$ almacena el instante en el que el algoritmo termina, así en una variable `diff` del tipo `double` se tiene que se guarda la diferencia entre estos dos tiempos, lo que representaría que la variable `diff` guarda el cálculo del tiempo de ejecución del algoritmo.

Como se nota una instancia y el tiempo de ejecución va a crecer de modo que aumente la estructura de la red y por ende los datos en la base, es decir el número de nodos, arcos y líneas.

Se ha desarrollado un algoritmo para correr algunas instancias, considerando un grafo con sus nodos, arcos y líneas generado aleatoriamente. A continuación se presenta la función que nos permite generar el grafo.

```
void generar_grafo(int n1, int k1, int r1)
{
  int mi=r1-(r1*0,1);//minimo valor de los arcos
  int ma=r1+(r1*0,1);//maximo valor de los arcos
  int dif=ma-mi;

  Query queryl=DB.query();//crear objeto query lineas
  Query queryp=DB.query();//crear objeto query paradas
  Query querylp=DB.query();//crear objeto query linea-parada
  cout<<endl<<"Generar grafos"<<endl;
  try {
queryl.execute("drop table tb_linea");
}
catch (mysqlpp::BadQuery&) {
// ignore any errors
}
queryl << "create table tb_linea  (LineaId bigint)";
queryl.execute();
queryl << "insert into %5:table values ( %0q )";
queryl.parse();
queryl.def["table"] = "tb_linea";

try {
queryp.execute("drop table tb_parada");
}
```

```
catch (mysqlpp::BadQuery&) {
// ignore any errors
}
queryp << "create table tb_parada  (ParadaId bigint)";
queryp.execute();
queryp << "insert into %5:table values (%0q)";
queryp.parse();
queryp.def["table"] = "tb_parada";

//insertar las paradas
for (int i=1; i<=n1; i++) // n1 numero de lineas
  {
    int par=i;
    queryp.execute(par);//insertar paradas
   }

try {
queryp.execute("drop table tb_lineaparada");
}
catch (mysqlpp::BadQuery&) {
// ignore any errors
}
  querylp << "create table tb_lineaparada"
          querylp << "(LineaParadaId bigint,LineaId bigint"
          querylp << ", Salida bigint, Llegada bigint)";
  querylp.execute();
```

```
  querylp << "insert into %5:table values (%0q,%1q,%2q,%3q)";
  querylp.parse();
  querylp.def["table"] = "tb_lineaparada";

int ID=1;
for (int i=1; i<=k1; i++) // k1 numero de lineas
  {
    int lin=i;
    queryl.execute(lin);//insertar lineas

    int r=(double(rand())/RAND_MAX)*dif+mi; //r numero de arcos
    cout<<"linea: "<< lin <<endl;
    cout <<"numero de arcos: " << r<< endl;
    arcos.resize(r+1,0);//arcos
    claves.resize(n1+1,0);// n1 numero de nodos en el grafo
    //inicializo las claves con cero
    for (int l=1; l<=n1; l++)
    {
       claves[l]=0;
       cout<<endl<<l<<" "<<claves[l]<<endl;
      }
    cout<<"nodos que forman lo arcos"<<endl;
     for (int i=1; i<=r+1; i++)
     {
                int j=((double(rand())/RAND_MAX)*(n1));
     if (j==0)
```

```
                j++;


  cout<<endl<<"inicia"<<endl<<endl;
  if (claves[j]==0)
    {  claves[j]=1;
          cout<<endl<<"primer"<<endl;
          cout<<endl<<j<<endl;
      }
   else
      {
      j=1;
      if (claves[j]==0)
         {
                        claves[j]=1;
                        cout<<endl<<"segundo"<<endl;
                        cout<<endl<<j<<endl;
      }
                     else
         {
           while (claves[j]==1)
            {
             j++;
              };
if (claves[j]==0)
                        {
                                  claves[j]=1;
```

```
                                }
                cout<<endl<<"tercero"<<endl;
                        cout<<endl<<j<<endl;
                       }
            }
          cout<<endl<<"me quedo con esto"<<endl<<j<<endl;
          arcos[i]=j;
         }
int ante,suce;
for (int i=1; i<=r; i++) // r numero de arcos
 {
    ante=arcos[i];
    suce=arcos[i+1];
    cout<<"Datos de las lineas y paradas";
    cout<<endl<<"numero registro "<<ID<<endl;
    cout<<endl<<"antecesor "<<ante<<endl;
    cout<<endl<<"Sucesor "<<suce<<endl;
    cout<<endl<<"Linea "<<lin<<endl;

    querylp.execute(ID,lin,ante,suce);//insertar linea parada
    ID++;
 }

cout<<endl<<"valores de las claves"<<endl;
for (int i=1; i<=n1; i++)
 {
```

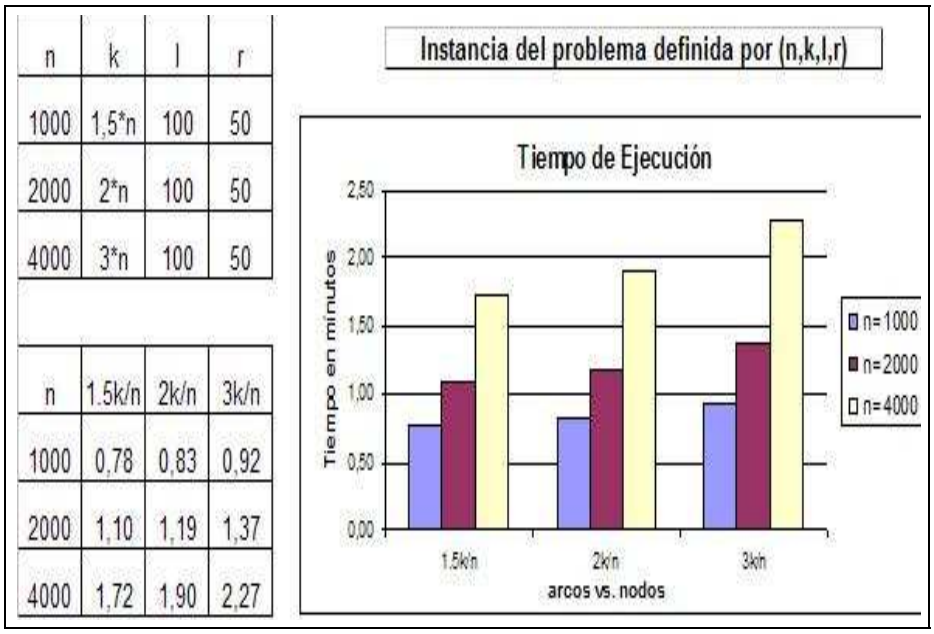
```
        cout<<endl<<claves[i]<<endl;
       }
   }

}
```

Este algoritmo trabajo sobre una base secundaria creada para realizar las corridas, en ella se guarda una instancia, es decir en la tabla tb_paradas se insertan las paradas, así como los arcos y líneas en las tablas tb_lineaparada y tb_linea respectivamente que se generan aleatoriamente, las otras tablas (tb_frecuencia, etc.) de la base que se utilizan en el algoritmo ya se encuentran definidas.

Se ha medido el tiempo de ecuación del algoritmo de optimización para algúnas instancias del problema de rutas más cortas considerando la restricción de cambio de línea. En la Figura 4.1 la primera tabla presenta los diferentes valores que toma n (cantidad de nondos en el digrafo), k (cantidad de arcos en el digrafo), l (cantidad de líneas en el digrafo) y r (longitud de la línea). La segunda Tabla presenta los tiempos de ejecución en mínutos del algoritmo para las instancias definidas. Se ha iniciado la evaluación del algoritmo con una instancia que varía el n y $k = \alpha * n$, donde α es factor de relación entre n y k ($\alpha = 1{,}5,\ 2,\ 3$), así también $l = 100$ y $r = 50$.

Por ejemplo se define la primera instancia con los valores $n = 1000$, $k = 1{,}5(1000)$, $l = 100$, $r = 50$, y se reporta un tiempo promedio para econtrar la solución para un para cualquiera de nodos de $0{,}78$ mínutos. En general se puede observar que un aumento en la cantidad de arcos en el digrafo constituye un aumento en el tiempo de procesamiento de la respuesta.

En las siguientes instancias de los problemas definidos para medir la eficiencia del algoritmo se consideran dos caso más, donde la cantidad de líneas l y longitud



n	k	l	r
1000	1,5*n	100	50
2000	2*n	100	50
4000	3*n	100	50

n	1.5k/n	2k/n	3k/n
1000	0,78	0,83	0,92
2000	1,10	1,19	1,37
4000	1,72	1,90	2,27

Figura 4.1: Tiempo de ejecución (arcos vs. nodos)

r de la misma varían también. En la Figura 4.2 se observa que el tiempo de corrida del algoritmo tiene una relación inversa con respecto a la cantidad de líneas y la cantidad de arcos en el grafo, es decir mientras existan menos líneas para recorrer y más arcos, se tiene que el tiempo de ejecución aumenta y viciversa.

Como últimas instancias se han definido en las cuales lalonguitud de las líneas varien con respecto a la cantidad de arcos en el digrafo (ver Figura 4.3.

Como se puede apreciar, el tiempo de ejecución tiene una relación creciente a medida de que el número de arcos aumenta y la longitud de las líneas disminuye.

n	k	l	r
1000	1,5*n	k/4	50
2000	2*n	k/8	50
4000	3*n	k/12	50

n	4k/l	8k/l	12k/l
1000	2,01	1,17	0,89
2000	3,98	2,32	1,76
4000	7,94	4,61	3,50

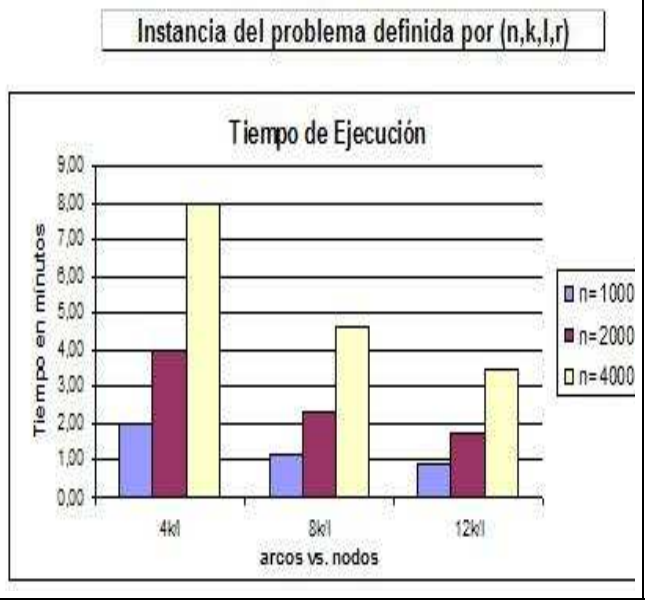


Figura 4.2: Tiempo de ejecución (arcos vs. líneas)

n	k	l	r
1000	1,5*n	k/4	k/5
2000	2*n	k/8	k/10
4000	3*n	k/12	k/15

n	5k/r	10k/r	15k/r
1000	2,14	2,06	2,03
2000	4,28	4,12	4,06
4000	8,55	8,23	8,13

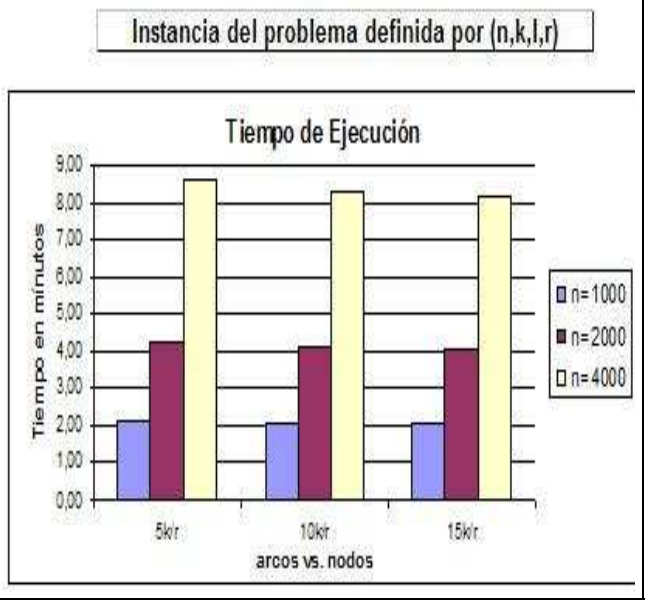


Figura 4.3: Tiempo de ejecución (arcos vs. longitud de la línea)

5. Conclusiones y Recomendaciones

5.1. Conclusiones

1. El problema de determinar las rutas más cortas al interior del sistema de transporte Metrovia puede formularse como un problema de caminos más cortos, considerando una restricción adicional asociada a las líneas de transporte: si se llega a una estación intermedia empleando una línea, entonces para salir de la estación a través de una línea diferente debe pagar un costo adicional que simboliza el tiempo requerido para el cambio de línea.
2. El algoritmo de solución que hemos desarrollado no trabaja sobre un grafo D explícitamente sino que emplea la idea de etiquetas en los nodos introducida por los algoritmos para el SPPRC. Para $v \in V$, la etiqueta (c_κ, l_κ) registra el costo c_κ (es decir, el tiempo total de recorrido) de un camino desde el nodo de partida hasta v y la línea l_κ a la que pertenece el último arco de este camino. El algoritmo opera entonces en una forma similar a la de los algoritmos de fijación de etiquetas (o al algoritmo de Dijkstra para el SPP clásico).
3. El algoritmo de solución reside en un servidor Web. A través de un formulario de Internet, el usuario ingresa los siguientes datos de entrada para el prolema: Origen del desplazamiento, Destino del desplazamiento, Fecha de viaje, Hora de salida.
4. Se ha implementado un sistema de información de consultas de viajes al interior del sistema de transporte METROVIA, el mismo que consiste de tres módulos:

- Una interfaz para el usuario, desarrollada en PHP, desde donde se pueden realizar las consultas de viajes a través de un formulario de Internet. La dirección de la página Web para realizar las consultas es: www.math.epn.edu.ec\ ∼mflores.
- Una base de datos en MySQL que contiene la información correspondiente a la estructura de la METROVIA (líneas, paradas, frecuencias, etc.)
- Una aplicación en C++ para realizar el cálculo de las rutas más cortas.

5. Se ha desarrollado a nivel de prototipo una aplicación informática para la determinación de rutas más cortas de tal forma que el ingreso de datos y la entrega de resultados se realicen a través de una interfaz apropiada para ser utilizado en Internet. Está aplicación ayudará a los usuarios a planificar mejor sus viajes y obtener un mayor provecho del sistema METROVIA.
6. Se ha diseado el algoritmo de optimización, la base de datos y la aplicación en Internet para las consulta de tal forma que si la estructura de la red Transporte cambía no se tenga que realizar cambios drásticos en el proceso de cálculo de viajes.
7. Nuestro trabajo es la primera aplicación para la planificación de viajes en Latino America, diseñada en plataforma Web.

5.2. Recomendaciones

1. Como completo del sistema de información de consultas de viajes se puede realizar una aplicación que sea utilizada para consultas de viajes através de mensajes celulares. La misma que tendría básicamente el mismo módulo de cálculo de viajes (algoritmo de optimización).
2. Se podría mejorar la interfaz del usuario mediante el desarrollo de un sistema de información geográfica, el cual sería otro módulo de nuestro sistema de información, donde la persona que consulta su viaje pueda visualizar gráficamente el recorrido del o los viajes a planificar.
3. El algoritmo desarrollado en ésta tesis, puede ser utilizado para casos generales de sistemas de transporte de pasajeros, para esto se tendría que considerar la estructura vial de toda la ciudad.
4. Actualmente los datos empleados para el desarrollo del proyecto de tesis considera los tiempos y frecuencias de las líneas para el caso en donde la demanda de pasajeros del sistema de transporte METROVIA es la misma para todos los días, se debería actualizar los tiempos y frecuencias considerando la demanda en feriados, sabado y domingos. Para esto se tiene que crear una tabla en la base de datos en la cual se mantenga una lista de las fechas del año donde se especifique que día es.
5. Para que el cálculo de los tiempos de viajes sea más exactos, se debería modificar el algoritmo para que determine las rutas más cortas considerando el mismo criterio de minimizar el tiempo de viaje pero calculandolo en tiempo real atraveés de la distancia y velocidad que lleva la línea.

Bibliografía

[1] Hand books in Operations Research and Managment ScienceÑorth Holland. *Network routing, volumen 8*. Amsterdam: Elsevier Science B.V., 1995.

[2] Claymath. Problemas del milenio. `http://www.claymath.org/millennium/`, 2000.

[3] S.A. Cook. The complexity of theorem proving procedures. *In Porc. Of the 3er Annual ACM Symposium of the Theory of Computing*, 16:151–158, 1971.

[4] William Cook, William Cunningham, William Pulleyblank, and Alexander Schrijver. *Combinatorial Optimization*. Wiley Interscience, 1998.

[5] Dirección de Transporte de Guayaquil. Información sobre el sistema de transporte masivo de pasajeros de la ciudad de guayaquil. *Muy Ilustre Municipalidad de Guayaquil*, 2005.

[6] Desrosiers. An algorithm for the shortes path problem with resource constraints. *Centre de recherche ur les transports, Montréal*, 421A, 1986.

[7] Pelletier y Sounmis Desrosiers. Plus court chemin avec contraintes d'horaries. *RAIRO, Inf. Théor*, 17:977–978, 1983.

[8] Dijkstra E. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[9] Garey and Johnson. *Computers and Intractability: A Guide to NP-completeness*. x, 1979.

[10] Edmons J. Paths, trees and flowers. *Canadian Journal of Mathematics*, 17:449–467, 1965.

[11] Ford L.R. Network flow theory. *RAND Corporation*, page 923, 1956.

[12] Bellman R.E. On a routing problemnegation as failure in the head. *Quarterly of Applied Mathematics*, 16:87–90, 1958.

[13] Schrijver. *Combinatorial Optimization*. Springer, 2003.

[14] Ronald L. River Thomas H. Cormen, Charles E. Leiserson, editor. *Introduction to Algorithms*. The MIT Press, Cambridge, Massachusetts, 2001.

[15] Desrochers y Sounmis. A generalized permanent labelling algorithm for the shortes path problem with windows. *Europ. J. on OR*, 26:191–212, 1983a.

[16] Desrochers y Sounmis. A reoptimization algorithm for the shortes path problem with windows. *Europ. J. on OR*, 35:242–254, 1983b.