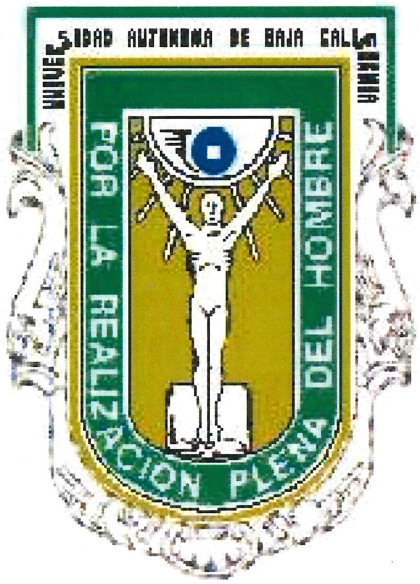
UNIVERSIDAD AUTÓNOMA DE BAJA CALIFORNIA

FACULTAD DE CIENCIAS



REGLAS DE TRÁNSITO EN UNA INTERSECCIÓN DE N CALLES

T E S I S

Que para obtener el título de

Licenciado en Matemáticas Aplicadas

presenta:

YADIRA PÉREZ GARIBAY

Ensenada, Baja California, México. Junio de 2006

UNIVERSIDAD AUTÓNOMA DE BAJA CALIFORNIA

FACULTAD DE CIENCIAS

REGLAS DE TRÁNSITO EN UNA INTERSECCIÓN DE N CALLES

T E S I S P R O F E S I O N A L

QUE PRESENTA

YADIRA PÉREZ GARIBAY

APROBADO POR:

DR. ALVARO ALVAREZ PARRILLA
Presidente del Jurado

FIS. ALFONSO NETZAHUALCOYOTL PAREDES CERVANTES
S E C R E T A R I O

DR. CARLOS YEE ROMERO
1ER. V O C A L

DEDICATORIA

Con amor a mis padres y hermanos...

Por su apoyo en todo momento.

Dios los bendiga siempre...

AGRADECIMIENTOS

De manera especial a *Dios* por estar conmigo en cada instante de mi vida...

A mis padres *Gabriel Pérez y Teresa Garibay* por el amor, apoyo y comprensión que me han dado siempre y enseñarme lo que no puedo aprender en un libro...

A mis hermanos *Belem, Iris Gabriela y Alan* por soportar mis regaños y travesuras...

A mi director de tesis Dr. Alvaro Alvarez Parrilla, por darme lata todo este tiempo... :-), pero también por darme consejos, brindarme su amistad y tenerme paciencia...

A las familias *Gaytan Garibay, Martínez Garibay, Pérez Gonzalez, Pérez Vargas, Pérez Garcia, Pérez Ledezma, Rosales Pérez, Echeverria Pérez y Rosalez Vazquez* por su apoyo moral y económico...

A mis maestros por compartir sus conocimientos, experiencias y tiempo dentro y fuera de un salón de clases...

Al Fis. Alfonso Paredes por el apoyo para que esta tesis se llevara acabo...

A mis amigos físicos, computólogos y por supuesto matemáticos, por todos los momentos buenos y malos que compartimos juntos y que gracias a dios aún compartimos...

RESUMEN de la Tesis de Yadira Pérez Garibay presentada como requisito parcial para la obtención de la Licenciatura en Matemáticas Aplicadas. Ensenada, Baja California, México, junio del 2006.

Reglas de tránsito en una intersección de n calles

En éste trabajo se desarrolló un algoritmo que permite encontrar y caracterizar las posibles "reglas de tránsito" válidas que pueden existir en una intersección de n calles, tomando en cuenta que las calles que llegan a dicha intersección pueden ser de un sentido o de doble sentido. Consideramos como "regla de tránsito" al hecho de que a cada entrada de la intersección, se le puede especificar a que calle no se le permite el acceso desde esta entrada. Las reglas de tránsito válidas determinan lo que llamamos configuraciones válidas, que reemplazan a la intersección y pueden ser utilizados para implementar las reglas de tránsito, sin necesidad de modificar los algoritmos de caminos mas cortos sobre grafos que ya existen.

Resumen aprobado:

Dr. Alvaro Alvarez Parrilla

Contenido

Figuras

Tablas

Capítulo 1

INTRODUCCIÓN

En la actualidad uno de los problemas más importantes que existen para empresas de transporte, es el de optimizar tiempo y distancia en el recorrido de un punto de la ciudad a otro. Este problema, en su forma más simple, ya tiene solución algorítmica en donde la ciudad se representa mediante un grafo, esto es, las aristas representan las calles de la ciudad y los vértices las intersecciones entre ellas (véase las figuras 1.1 y 1.2). Entre los algoritmos solución destaca el algoritmo de Dijkstra, ya que es el más eficiente que se conoce [5].

Por otro lado, en una ciudad existen reglas de tránsito que regulan el flujo vehicular en las calles. Estas reglas de tránsito nos indican situaciones tales como, si una calle es de un solo sentido, o si en una intersección de ellas está permitida la vuelta, o cual es la velocidad máxima permitida, etc.

Entre las reglas de tránsito, hay algunas que fácilmente se pueden tomar en cuenta utilizando los algoritmos basados en grafos. Por ejemplo, las reglas que indican la velocidad máxima permitida en una calle o la longitud de la misma, en cuyo caso se le asigna un peso a la arista que indique ya sea la velocidad permitida o la longitud,

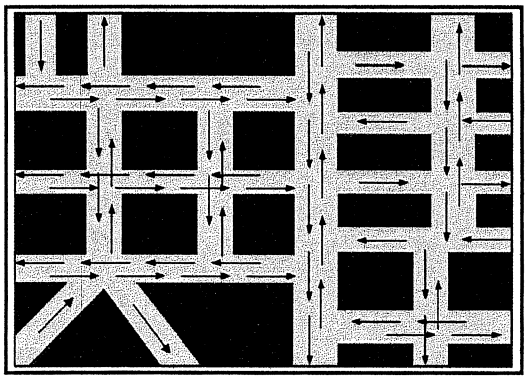


Figura 1.1: Representación de una ciudad con la dirección de sus calles.

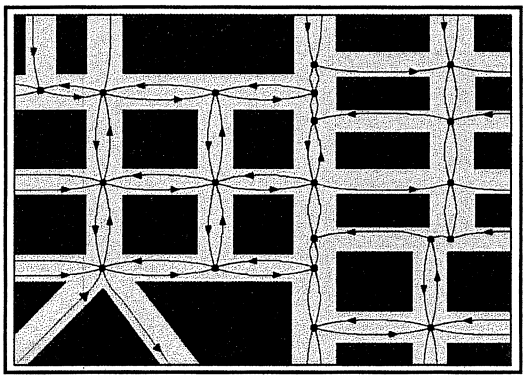


Figura 1.2: Representación de una ciudad mediante un grafo dirigido.

y estos casos se pueden representar por un grafo con aristas con pesos. Otro caso es el sentido de circulación de las calles, en donde se construye un grafo dirigido de manera que una calle de doble sentido se describe por medio de dos aristas; una por cada sentido, o ambos casos que se representan por medio de un grafo dirigido y con pesos (véase la figura 2.2).

Por otro lado existen reglas que, *a-priori*, no son fáciles de tomar en cuenta utilizando los algoritmos basados en grafos. En particular, en éste trabajo, nos enfocamos en una intersección de calles y consideramos algunas de las reglas de tránsito

que pueden surgir en dicha intersección: los sentidos de las calles y las posibilidad de no dar vuelta a otra calle.

Como se señaló anteriormente, los sentidos son fácilmente implementables en el grafo considerando un grafo dirigido, sin embargo, las reglas de tránsito que señalan la posibilidad de no dar vuelta a otra calle, no son fácilmente implementables. Una opción para tomar en cuenta estas reglas de tránsito que existen (no vuelta a la izquierda, no vuelta a la derecha, etc...) sería modificar el algoritmo, e incluir en el vértice correspondiente del grafo, la información de la regla de tránsito.

Esto pudiera llegar a funcionar relativamente bien, pero resulta más eficiente no tener que modificar al algoritmo y buscar la manera de solo modificar al grafo para que él mismo contenga la información de las reglas de tránsito dentro de su geometría.

Con esto en mente, en éste trabajo se desarrolló e implementó un algoritmo en el cual, dada una intersección de n calles, se determinan algunas reglas de tránsito que pueden llegar a existir sobre dicha intersección. Así mismo, el algoritmo determina la modificación que se le tiene que realizar al grafo que representa la intersección, para que contenga la información de la regla de tránsito que se quiere aplicar. En otras palabras se generan subgrafos que representan una intersección de n calles con k restricciones aplicadas, donde $k \leq n$.

En resumen, se obtienen todos los posibles subgrafos que pueden reemplazar al nodo central del grafo original que representa la intersección de n calles, de tal manera que:

1. el algoritmo de ruteo (Dijkstra) no necesita ser modificado,

2. el reemplazar la intersección por un subgrafo dado, modifica el comportamiento exactamente como si se le estuviera indicando una regla de tránsito dada en esa

intersección.

A los subgrafos modificados que cumplen con el criterio de señalar una regla de tránsito dada le llamaremos una *Configuración Válida.*

Asimismo, es importante que se estudie el aumento de la complejidad del grafo resultado de reemplazar los nodos centrales por los correspondientes subgrafos, para evaluar el crecimiento en la complejidad del grafo al aplicar la solución propuesta.

A continuación se muestra como se encuentra organizada la presente tesis: En el capítulo 2 se presentan antecedentes, definiciones y conceptos que se utilizan en el planteamiento y solución de este problema, posteriormente en el capítulo 3 se muestra la descripción del mismo, en el siguiente capítulo se presentan detalladamente los resultados, así como, subgrafos correspondientes a las configuaraciones válidas para el caso $n = 3$ que son parte del resultado del algoritmo solución y un análisis de complejidad del mismo. Finalmente se termina con el capítulo de conclusiones.

En los apéndices se muestran conceptos y definiciones sobre teoría de conjuntos utilizadas en el planteamiento y solución del problema, un análisis de complejidad del algoritmo de Dijkstra y la implementación en C++ del algoritmo.

Capítulo 2

ANTECEDENTES

En este capítulo, se muestra una pequeña introducción sobre Teoría de Grafos, así como definiciones que son utilizadas en el planteamiento de este problema. Además, se presenta una breve descripción del algoritmo de Dijkstra.

2.1 Antecedentes básicos de Teoría de Grafos

El origen histórico de la teoría de grafos, suele situarse en el conocido problema de "los siete puentes de Könisberg", donde se trata de encontrar un trayecto alrededor de una serie de puentes, de tal manera que cruce solamente una vez cada uno de ellos. En 1736, el matemático suizo, radicado en San Petersburgo, Leonhard Euler, publicó un artículo en el que resolvía el problema en el caso general. Este trabajo es considerado como el nacimiento de la Teoría de Grafos [4].

Los grafos son modelos matemáticos de numerosas situaciones reales: por ejemplo, un mapa de carreteras, un plano de la red de metro de una ciudad, un plano de un circuito eléctrico, un plano de red telefónica de una compañia, etc. Asimismo, un

algoritmo puede representarse mediante un grafo al que se llama diagrama de flujo del algoritmo [4].

Mencionado lo anterior, procedemos a dar una definición formal de grafo: Un *grafo* $G = (V, A)$ consiste en un par de conjuntos, donde $V = \{v_1, v_2, ..., v_N\}$ es un conjunto de puntos llamados *vértices* y $A = \{(v_i, v_j)\} \subseteq V \times V$ de pares no ordenados de vértices a cuyos elementos llamaremos *aristas.*

En algunos casos es necesario tener direcciones o sentidos en las aristas, por lo que se representan por pares ordenados de vértices tales que $(a, b) \neq (b, a)$, cuando $a \neq b$, donde a es el extremo inicial y b es el extremo final. A este tipo de grafo se le llama *Digrafo o grafo dirigido.* Cuando las aristas no son ordenadas, entonces hablamos de un *grafo no dirigido.* Si a cada arista se le asocia un *peso* o valor específico, el cual representamos como $w(a, b)$, hablaremos de un *grafo con pesos.*

Ejemplo 2.1.1. En la figura 2.1 se muestra un grafo no dirigido, donde los vértices son $V = \{a, b, c\}$ y las aristas son $A = \{(a, b), (b, c), (c, a)\}$.

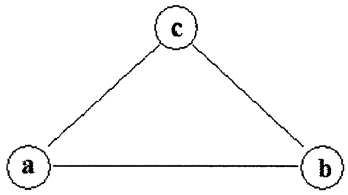


Figura 2.1: Grafo no dirigido.

Ejemplo 2.1.2. En la figura 2.2 se muestra un grafo dirigido y con pesos en sus aristas. En este caso $V = \{a, b, c\}$, $A = \{(a, b), (b, c), (c, b), (c, a)\}$ y los pesos están dados por: $w(a, b) = 30, w(b, c) = w(c, b) = 20$ y $w(c, a) = 50$.

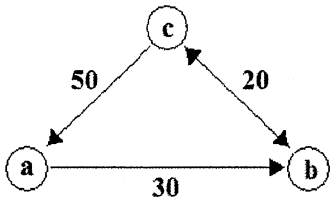


Figura 2.2: Grafo dirigido y con pesos en sus aristas 1.

Un *camino* W_{ab} entre los vértices a y b es una secuencia de vértices tales que

- el primer vértice de la secuencia es a y el último b,
- de cada uno de los vértices $a \neq b$, que forman el camino existe una arista desde el vértice sucesor.

Un *camino simple* W, es aquel en el que no se repite ninguno de sus vértices en él. El *camino más corto* entre dos vértices a y b, es un camino W_{ab} en un grafo con pesos que minimiza la suma de los pesos asociados a las aristas correspondientes a los vértices del camino.

Ejemplo 2.1.3. En el grafo que muestra la figura 2.3 se pueden determinar distintos caminos de un vértice dado a otro, por ejemplo, un camino W_{ae} compuesto por los vértices $\{a, b, e\}$ y las aristas $\{(a, b), (b, e)\}$. Este ejemplo es un camino simple.

Ejemplo 2.1.4. En el mismo grafo (ver figura 2.3), sea a el vértice de inicio y c el vértice de destino. Unas de las secuencias de vértices que me describen los diferentes caminos están dadas por los vértices $\{a, b, c\}$ el cual denominaremos como W_{ac} y W'_{ac} con los vértices $\{a, b, e, c\}$. De acuerdo a la suma de los pesos asociados a las aristas del grafo en cada camino tenemos los pesos 9 y 10 respectivamente, por lo que podemos decir que W_{ac} es el camino más corto entre ambos nodos.

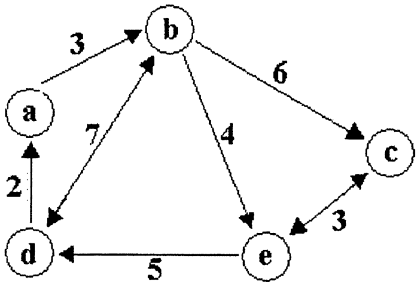


Figura 2.3: Grafo dirigido y con pesos en sus aristas 2.

Un *grafo conexo* es un grafo no dirigido de modo que para cualquier par de nodos existe al menos un camino que los une. Un *ciclo* es un camino que empieza y termina en el mismo vértice.

Dadas las definiciones anteriores podemos decir, que un *árbol* es un grafo no dirigido, conexo y sin ciclos.

Existen varios algoritmos que encuentran el camino más corto sobre un grafo, entre los que destacan el de Bellman-Ford, que funciona con aristas con pesos arbitrarios, y el de Dijkstra que solo funciona con pesos no negativos (pero que es más eficiente).

Como se planteó en la introducción, en el grafo que representa a una ciudad, los pesos de las aristas corresponden a las longitudes de las calles y/o a la velocidad máxima permitida por esas calles. Por lo tanto, estos pesos son no negativos, de allí que se utilice el algoritmo de Dijkstra para encontrar el camino más corto.

2.2 El algoritmo de Dijkstra y el camino más corto

El profesor Edsger Wybe Dijkstra, originario de Rotterdam, Holanda (1930-2002), anunció en 1959 un algoritmo que lleva su nombre y que le valió, en 1972, el premio Turing otorgado por la ACM (Association for Computing Machinery, EEUU) que

es considerado el equivalente al premio Nobel de la computación. El algoritmo de Dijkstra determina la ruta o camino más corto entre dos vértices de un grafo conexo, el vértice de origen a y el vértice de destino z [1].

La idea subyacente en este algoritmo consiste en ir explorando los caminos más cortos que parten del vértice origen y que llevan a *todos* los demás vértices. Cuando se obtiene el camino más corto desde el vértice origen al resto de vértices que componen el grafo, el algoritmo se detiene. Por lo tanto, al finalizar el algoritmo se obtiene el camino más corto del vértice origen a todos los demás vértices del grafo [1].

Para explicar el funcionamiento del algoritmo para determinar el camino más corto, primero presentamos los elementos que lo constituyen:

- Sea $G = (V, A)$ un grafo dirigido con pesos no negativos.
- Sean $a, z \in V$ los vértices de origen y de destino respectivamente.
- Sea un conjunto $C \subset V$, que contiene los vértices de V que determinan el camino más corto pero que todavía no se conoce.
- Sea un vector D, de dimensión igual al orden de V, y que "guarda" los pesos entre a y cada uno de los vértices de V.
- Sea finalmente P, otro vector de la misma dimensión que D, y que conserva la información sobre qué vértice precede a cada uno de los vértices en el camino.

Enseguida presentamos de manera breve el funcionamiento del algoritmo de Dijkstra para determinar el camino más corto entre los vértices a y z:

1 $C \leftarrow V$

2 Para todo vértice $i \in C$, $i \neq a$, se establece $D_i \leftarrow \infty$; $D_a \leftarrow 0$

3 Para todo vértice $i \in C$ se establece $P_i \leftarrow a$

4 Se obtiene el vértice $s \in C$ tal que no existe otro vértice $t \in C$ tal que $D_t < D_s$

5 Si $s = z$ entonces se ha terminado el algoritmo.

6 Se elimina de C el vértice $s : C \leftarrow C \backslash \{s\}$

7 Para cada arista $e \in A$ de peso l, que une el vértice s con algún otro vértice $r \in C$

8 Si $l + D_s < D_r$, entonces:

9 Se establece $D_r \leftarrow l + D_s$

10 Se establece $P_t \leftarrow s$

11 Se regresa al paso 4

Al terminar este algoritmo, en D_z se tendrá el peso mínimo entre a y z. Por otro lado, mediante el vector P se puede obtener el camino mínimo: en P_z estará y, el vértice que precede a z en el camino mínimo; en P_y estará el que precede a y, y así sucesivamente, hasta llegar a a el vértice de origen. Además, es de notar que P es un árbol.

Coloquialmente, el comportamiento del algoritmo se puede pensar de la siguiente manera: consideremos que las aristas del grafo son tubos por donde puede correr pintura y que los vértices son uniones de los tubos. Así, para entender el algoritmo,

se puede pensar que se inyecta pintura al grafo a partir del vértice de origen y la pintura avanza con velocidad uniforme por los tubos (por el grafo). Para saber el camino más corto del vértice origen a un vértice dado, lo que se tiene que hacer es identificar por cual de los tubos (aristas) llegó la pintura primero y regresarse por esta arista al otro vértice de la misma y repetir el proceso hasta llegar al vértice de inicio. De esta manera se encuentra la ruta más corta.

Capítulo 3

DESCRIPCIÓN DEL PROBLEMA A RESOLVER

En este capítulo, se describe el problema así como también la simbología y definiciones que se utilizan en el desarrollo de este problema.

En una ciudad existen diferentes tipos de calles entre las que sobresalen las de doble sentido y un solo sentido, es por esta razón que en éste problema las calles se representan por grafos dirigidos. Nos interesa lo que ocurre en una intersección de n calles así que le llamaremos a dicha intersección nodo central, y definimos a los tipos de calles de la siguiente manera:

- Doble sentido (D)
- Entrada al nodo central (E)
- Salida del nodo central (S)

Los tipos de calles los representamos como: 0 a las calles de doble sentido, 1 a las Entradas al nodo central y -1 a las Salidas del nodo central.

Las reglas o restricciones de tránsito se definen como la indicación de no acceso a la calle m si provenimos de una Entrada j, donde se cuentan las calles a partir de la Entrada j y en sentido de las manecillas del reloj.

Para una intersección de n calles existen n tipos de restricciones que se pueden aplicar a cada Entrada, esto es, considerando solo una restricción por Entrada, estas son:

1. No vuelta a la 1^{era} calle inmediatamente a la izquierda.

2. No vuelta a la 2^{da} calle inmediatamente a la izquierda.

3. No vuelta a la 3^{era} calle inmediatamente a la izquierda.

$\vdots$

$n-1$. No vuelta a la $(n-1)^{\text{esima}}$ calle inmediatamente a la izquierda.

n. Se permiten las vueltas a todas las calles.

Así pues, en total tendremos, a lo mas, k restricciones en una intersección, una por cada Entrada, donde $k = \#\text{Entradas} \leq n$.

Los tipos de restricciones en una intersección de n calles los representamos de la siguiente forma:

0 libre acceso a todas las calles.

1 no hay acceso a la primera calle inmediatamente a la izquierda.

2 no hay acceso a la segunda calle inmediatamente a la izquierda.

$\vdots$

$n-1$ no hay acceso a la $n-1$-esima calle inmediatamente a la izquierda.

Dada esta información, el problema es determinar configuraciones válidas en una intersección.

Entenderemos el concepto de *configuración válida* como aquella que cumple que: *"Todas las Salidas tengan por lo menos una Entrada, y que todas las Entradas tengan por lo menos una Salida, en donde se entiende que en ambos casos las Entradas o Salidas en cuestión no pueden ser ellas mismas"* (en éste trabajo no se contemplan las vueltas en "U"). Además, una configuración válida es un subgrafo que contiene la siguiente información:

1. Número de calles en una intersección.
2. Un representante de la clase de equivalencia de una permutación de los tipos de calles, en donde se ha tomado en cuenta la simetría.
3. Una permutación de los tipos de restricciones.
4. Una lista de todas las Entradas y Salidas del subgrafo.

Ejemplo 3.0.1. Configuración válida $= (n, [0, 1, \cdots, -1], (1, m, \cdots, 3)$, Entradas,Salidas)

Ejemplo 3.0.2. En la figura 3.1 se muestra una intersección de cuatro calles que cumple con la definición de una configuración válida.

Ejemplo 3.0.3. La figura 3.2, es un claro ejemplo de una intersección de cuatro calles que no cumple la definición de configuración válida, ya que no existe una Salida para la Entrada de la calle del tipo D.

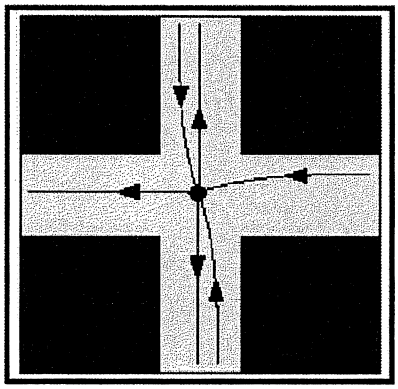


Figura 3.1: Configuración válida $= (4, [0, 1, 0, -1], (0, 0, 0), \text{Entradas, Salidas})$

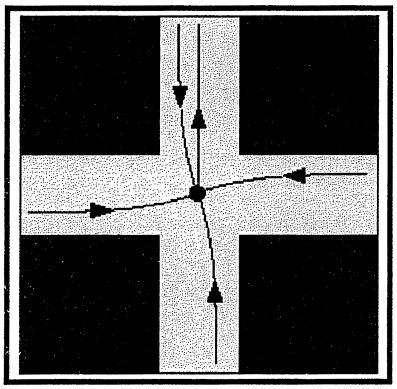


Figura 3.2: Configuración no valida para el caso $n = 4$.

En el capítulo 5, se muestra otro ejemplo de una configuración válida y una no válida con restricciones que se pretenden aplicar a una intersección de calles.

Capítulo 4

RESULTADOS

En éste capítulo se muestra en detalle el como obtener la solución al problema, así como algunos ejemplos y un análisis de la complejidad de la solución.

Al principio del trabajo, se pensó en obtener una solución puramente analítica, sin embargo al examinar el problema con mayor detenimiento, se vió que resultaba tener un grán número de casos que se pueden tratar de manera muy similar o metódica. De aquí que se propuso una solución algorítmica del problema.

4.1 La solución algorítmica

En las siguientes secciones nos podremos dar cuenta que la implementación del algoritmo se realizó diferente a la descripción del problema, esto se debe a la naturaleza de la solución. Además, se explican las rutinas principales de dicho algoritmo.

4.1.1 Descripción general del algoritmo

El algoritmo está diseñado para encontrar las restricciones que pueden existir en una intersección de n calles, donde n es fija en todo el algoritmo.

El algoritmo encuentra todas las combinaciones de calles que pueden ocurrir con los 3 distintos tipos de ellas, en una intersección dada.

Estas se agrupan en clases de equivalencia en donde se toma en cuenta que existen simetrías de rotación para evitar tener repeticiones.

Para cada clase de equivalencia se encuentran todos los posibles tipos de restricciones que puede tener (encontrando al mismo tiempo el subgrafo correspondiente).

4.1.2 Implementación del algoritmo

El algoritmo consiste principalmente de generación de tablas (restricciones, combinación de los tipos de calles y clases de equivalencia), y de tres etapas de revisiones:

1. El algoritmo inicia con la generación de n tablas de tipos de restricciones, una tabla para cada valor de k =#Entradas en la intersección,

2. enseguida se genera la tabla que contiene la combinación de los tipos de calles en una intersección, para la n y los 3 distintos tipos de calles,

3. también genera una tabla de clases de equivalencia para cada uno de los números de tipos de intersecciones,

4. para cada clase de equivalencia realiza la primera revisión,

5. si pasa la revisión se procede a elegir un elemento de la tabla de restricciones que corresponde al número de Entradas de esa clase de equivalencia,

6. se generan las permutaciones de esta restricción donde se tienen que considerar todas las simetrías posibles,

7. para cada permutación de restricciones, aplicadas a la clase de equivalencia en cuestión, se procede a realizar la segunda revisión y modificación del grafo (correspondiente a la clase de equivalencia),

8. finalmente si pasa la segunda revisión se procede con la tercera revisión,

9. si pasa la tercera revisión se acepta el subgrafo como una configuración válida.

Así se obtienen todas las configuraciones válidas para cada n dada.

4.1.3 Diagrama de flujo del problema

En el siguente diagrama de flujo se muestra la implementación del algoritmo anteriormente descrito, en este diagrama se puede observar de una manera mas clara las principales rutinas de este algoritmo (véase diagrama parte 1 y 2, figuras 4.1 y 4.2 respectivamente).

4.1.4 Rutinas principales

Generación de n tablas de tipos de restricciones

Se generan n tablas, ya que n es el número máximo de Entradas que podemos tener en una intersección de calles (una tabla para cada número de Entradas). En cada renglón de la tabla se especifica cuantas restricciones de cada tipo se aplicaran a una intersección con k Entradas. Recordando que los tipos de restricciones se definieron como: $0, 1, 2, \cdots, n-1$.

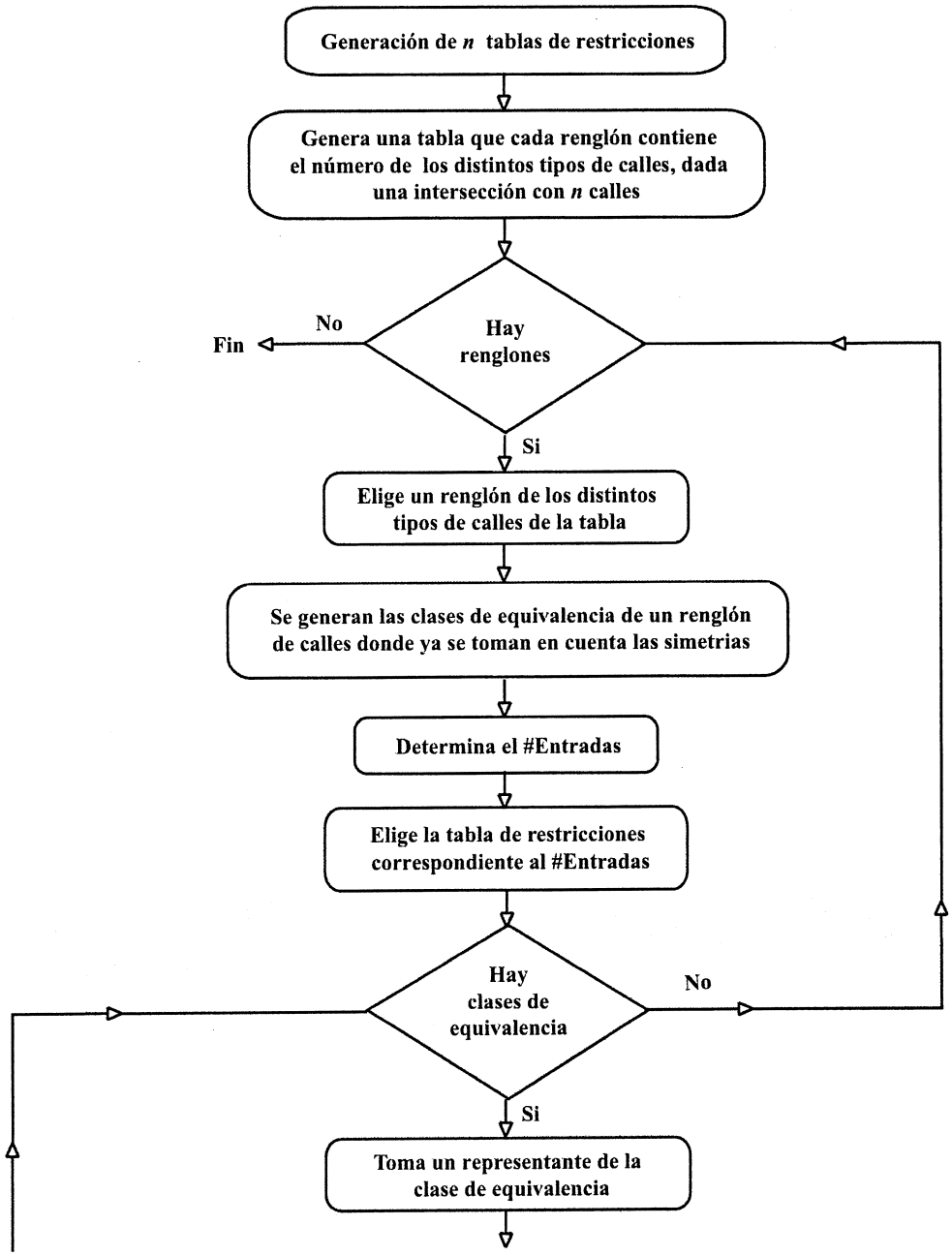


Figura 4.1: Diagrama de flujo. Parte 1

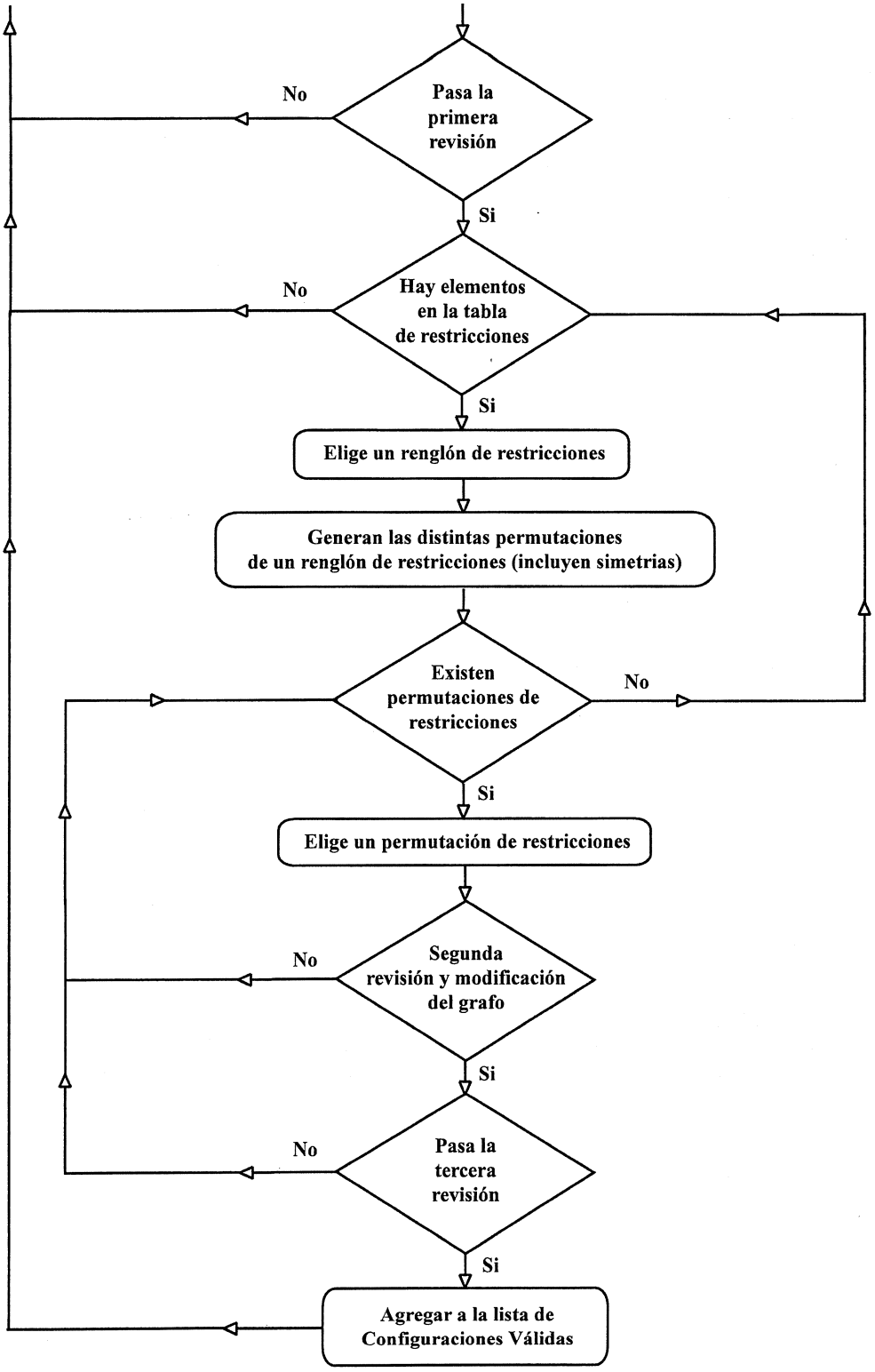


Figura 4.2: Diagrama de flujo. Parte 2

Ejemplo 4.1.1. Para k Entradas, véase la tabla I.

Tabla I: Tabla de restricciones que se genera en la intersección de n calles con k Entradas.

0	**1**	**2**	**.**	**.**	**n-1**
k	0	0	.	.	0
k-1	1	0	.	.	0
k-1	0	1	.	.	0
.	.	.	.	.	.
k-1	0	0	.	.	1
k-2	2	0	.	.	0
k-2	1	1	.	.	0
.	.	.	.	.	.
k-2	1	0	.	.	1
k-2	0	2	.	.	0
.	.	.	.	.	.
.	.	.	.	.	.
0	0	0	.	.	k

En el primer renglón de la tabla se muestran los tipos de restricciones ya anteriormente definidos, y en los renglones siguientes las cantidades de cada tipo de restricción. Cada renglón determina un conjunto de restricciones.

Para poder obtener la información de estos renglones es necesario saber n, el número de calles de una intersección y k el número de Entradas, con esta información se genera dicha tabla.

A continuación se describe explicítamente como se genera una tabla. El resto de ellas se generan con un proceso análogo, recordemos que son n tablas las que se generan.

Caso de n calles y k Entradas: Se tienen k objetos (las Entradas) que se requieren acomodar en n casillas (total de restricciones que se pueden aplicar = total de calles), por lo que se necesitan determinar los elementos de cada renglón

$(k_1, k_2, \cdots, k_n)$, por ejemplo, el renglón 1 formado por: k_1 objetos en la primera casilla, k_2 objetos en la segunda casilla, y k_n objetos en la última casilla, donde $k_1 + k_2 + \cdots + k_n = k$ y $k \leq n$, esto es para cada renglón.

El total de renglones que se obtienen para k objetos, en el caso de n casillas lo representamos como ${}_nR_k$ y se puede calcular de la siguiente manera :

Para $n = 3$ tenemos:

$$ {}_3R_k = C_k = \sum_{i=1}^{k+1} i = \frac{(k+1)(k+2)}{2} \tag{4.1.1} $$

$$ \text{para } k = 1, 2, 3. $$

Para $n = 4$

$$ {}_4R_k = \sum_{i=0}^{k} C_i = C_0 + C_1 + ... + C_k \tag{4.1.2} $$

$$ \text{para } 1 \leq k \leq 4. $$

Para $n = 5$

$$ {}_5R_k = \sum_{i_1=0}^{k} \sum_{i_2=0}^{i_1} C_{i_2} \tag{4.1.3} $$

$$ \text{para } 1 \leq k \leq 5. $$

Para $n = 6$

$$ {}_6R_k = \sum_{i_1=0}^{k} \sum_{i_2=0}^{i_1} \sum_{i_3=0}^{i_2} C_{i_3} \tag{4.1.4} $$

$$ \text{para } 1 \leq k \leq 6. $$

El caso $n = m$ está dado por:

$$ {}_mR_k = \sum_{i_1=0}^{k} \left(... \left(\sum_{i_{m-4}=0}^{i_{m-5}} \left(\sum_{i_{m-3}=0}^{i_{m-4}} C_{i_{m-3}} \right) \right) \right) \tag{4.1.5} $$

donde $1 \leq k \leq m$.

Si observamos la tabla notamos que cada renglón se puede pensar como un número de n dígitos en base $k+1$, por lo tanto los renglones forman un subconjunto de los números de n dígitos en base $k+1$. Por lo que en el algoritmo los renglones de esta tabla se generan tomando en cuenta esta observación, y tenemos que la tabla contiene a lo más $(k+1)^n - 1$ (se elimina el renglón de elementos ceros).

Notamos también que los renglones de la tabla, visto como un número en base $k+1$, tiene la característica de que sus dígitos suman k, por lo tanto son un subconjunto de los números de n dígitos en base $k+1$ que son múltiplos de k. Así el número de renglones en la tabla, ${}_nR_k$, es menor o igual que la cardinalidad del conjunto de números, en base $k+1$, que son divisibles entre k y que tengan n dígitos o menos.

En el algoritmo se aprovecha esto y se proceden a revisar precisamente los números entre k y $(k+1)^n - 1$, que son divisibles entre k. La manera de revisarlos es tomar el número en cuestión, representarlo en base $k+1$ y sumar sus dígitos, si suman k se acepta, de lo contrario no.

Generación de la tabla que contiene el número de distintos tipos de intersecciones de calles

Para generar los renglones de esta tabla es necesario conocer n, el número de calles en la intersección, y cuantos tipos de calles se consideran. Estas pueden ser de tres distintos tipos: Doble sentido, Entrada y Salida del nodo central y recordamos que están representadas por 0, 1 y -1 respectivamente.

Sin importar el número de calles en la intersección, los tipos de calles no cambian, por lo tanto el problema fué un caso particular al anterior, esto es, tenemos k objetos

y los queremos acomodar en t casillas. En este caso t representa los tipos de calles, por lo que $t = 3$, y k el número de calles de la intersección.

Para determinar el número total de renglones en esta tabla se aplica la ecuación (4.1.1), donde k representa el número de calles en la intersección. La información que nos proporciona cada renglón es la cantidad de calles que existen de cada tipo. Véase la tabla II.

Tabla II: Tabla de conjuntos de n calles que se genera en la intersección de n calles.

D	**E**	**S**
n	0	0
n-1	1	0
n-1	0	1
n-2	2	0
n-2	1	1
n-2	0	2
n-3	3	0
.	.	.
.	.	.

En el primer renglón de la tabla se muestran los diferentes tipos de calles D, E y S, y en los renglones siguentes la cantidad de cada tipo de calle.

Así, el problema se reduce a construir los vectores tridimensionales con dígitos enteros entre 0 y n, tales que la suma de los dígitos sea n. Esto es muy sencillo por enumeración. En esto se basa esta rutina del algoritmo.

Generación de las clases de equivalencia, donde se tome en cuenta la simetría

Para generar las clases de equivalencia de los tipos de calles, es necesario primero determinar el número de permutaciones que se tienen de dicha clase de equivalencia,

recordando que solo tenemos la cantidad de cada tipo de calles pero no el orden (esta es la información que viene en cada renglón de la tabla anterior).

Este problema se reduce a tener n objetos con n_1 del primer tipo, n_2 de segundo tipo, y n_k de un k-ésimo tipo, donde $n_1 + n_2 + \cdots + n_k = n$, así pues existen

$$ {}_nP_{n_1,n_2,\ldots,n_k} = \frac{n!}{n_1!n_2!\ldots n_k!} \tag{4.1.6} $$

permutaciones de los objetos dados. En este caso n es el número total de calles, $k = 3$ y n_1, n_2, n_3 son los diferentes tipos de ellas (que corresponden a D, E ,S respectivamente). Llamémosle P al número de permutaciones.

Se debe tomar en cuenta que existen permutaciones que son rotaciones de otras en algunos renglones de la tabla, así que, tenemos que eliminar este tipo de situaciones y de esta manera obtener las clases de equivalencia de cada renglón de la tabla.

Notemos que si n divide a P, entonces existe simetría de rotación en los tipos de calles que corresponden a este renglón de la tabla, por lo que

$$ h = \frac{P}{n} \tag{4.1.7} $$

corresponde al número de clases de equivalencia. Si por el contrario n no divide a P, entonces el número de clases de equivalencia es la parte entera de h más 1.

Ejemplo 4.1.2. Sea $n = 4$ y sea {2,2,0} un conjunto de tipos de calles, el cual nos dice que tenemos 2 calles del tipo D, 2 del tipo E y 0 del tipo S (este conjunto representa un renglón de la tabla II para el caso $n = 4$). Al realizar el cálculo de las permutaciones (con la ecuación 4.1.6) tenemos que $P = 6$. Las clases de equivalencia están dadas por [D,D,E,E] y [D,E,D,E], la primer clase de equivalencia representa

cuatro permutaciones y la segunda dos, podemos ver fácilmente éstas permutaciones con solo realizar rotaciones en cada clases de equivalencia. Por lo tanto, podemos notar que en éste caso n no divide a P por lo que $h = 1$ y en total tenemos 2 clases de equivalencia que se mostraron anteriormente.

Ya que se tiene el número de clases de equivalencia se procede a:

1. Primero construir un tipo de intersección como una n-tupla ordenada

$$(x_1, x_2, x_3, \cdots, x_{n-2}, x_{n-1}, x_n)$$

que contiene n_1 calles de tipo D, seguidas de n_2 calles de tipo E y de n_3 calles de tipo S. Este primer tipo de intersección ya es un representante de alguna de las clases de equivalencia.

2. A continuación se procede a examinar los últimos dos elementos de la n-tupla, esto es los elementos x_{n-1} y x_n. Si estos elementos son distintos, entonces se intercambian y ésta nueva n-tupla es un representante de otra clase de equivalencia distinta.

3. Enseguida se examinan los elementos que le siguen, esto es se examinan x_{n-2}, x_{n-1} de la n-tupla que se obtiene del paso anterior. Si son distintos se cambian.

4. Este proceso se continua hasta que se obtiene el total de clases de equivalencia h o hasta que se revisen los elementos x_2 y x_3 de la última n-tupla considerada.

5. Si se intercambian x_2 y x_3 y aún no se tiene el total de clases de equivalencia, regresamos de nueva cuenta a revisar los elementos x_{n-1} y x_n de la última n-tupla considerada.

6. El proceso se repite hasta obtener el total de clases de equivalencia.

De esta manera se generan todos los representantes de las clases de equivalencia.

Primera revisión

La primera revisión consiste en tomar una clase de equivalencia y verificar que para cada Entrada i al nodo central exista al menos una Salida j del nodo central ($i \neq j$).

Generación de las permutaciones de un renglón de restricciones, donde se incluyen las simetrías

Para generar estas permutaciones primero elegimos un renglón de restricciones en el cual tenemos la cantidad de restricciones de cada tipo, y es de tamaño n, este renglón lo convertimos a un renglón de tamaño #Entradas de esta manera tenemos la restricción que le corresponde a cada Entrada de una clase de equivalencia dada. Véase las líneas marcadas en el pseudo código.

Ahora si, para este nuevo renglón de restricciones (que tiene la información de las restricciones que le corresponden a cada Entrada de la clase de equivalencia que se esta examinando), utilizamos la ecuación (4.1.6) para determinar el total de permutaciones de dicho renglón pero para determinar estas permutaciones utilizamos el renglón original, es decir, el renglón que contiene la cantidad de cada tipo de restricción, donde en este caso n representa el número de Entradas y $n_1, n_2, \cdots, n_k$ representan la cantidad de cada tipo de restricción; en esta situación no es necesario eliminar simetrías (ya que el orden de las restricciones se aplican a cada Entrada y el orden de ellas se encuentra en la clase lateral).

Segunda revisión y modificación del grafo

Un tipo de intersección de n calles, lo podemos representar con $2n+1$ vértices, $2n$ vértices en los extremos $(V_1, V_2, \cdots, V_{2n})$, y un vértice central (V_{2n+1}) antes llamado nodo de intersección. Los vértices están ordenados a favor de las manecillas del reloj, el vértice central es el último en este orden. La cantidad de aristas va a depender de la clase de equivalencia y estas las representamos por un par ordenado de manera que (V_a, V_b) representa a la arista que va del vértice V_a al vértice V_b. Así pues, las aristas de Entradas están dadas de los vértices extremos (vértices impares) al vértice central, $\{(V_1, V_{2n+1}), (V_3, V_{2n+1}), \cdots, (V_{2n-1}, V_{2n+1})\}$ y las Salidas del vértice central a los extremos (vértices pares) $\{(V_{2n+1}, V_2), (V_{2n+1}, V_4), \cdots, (V_{2n+1}, V_{2n})\}$ (véase la figura 4.3).

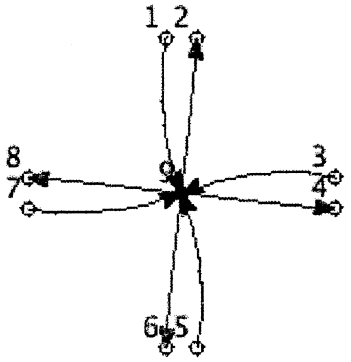


Figura 4.3: Grafo que representa una intersección de cuatro calles con la clase de equivalencia $[D, D, D, D]$ sin restricciones, las aristas rojas representan Salidas y las negras Entradas.

En la segunda revisión se intenta aplicar un renglón de permutaciones de restricciones al grafo que representa una clase de equivalencia, y al mismo tiempo se revisa si se puede aplicar.

La aplicación se realiza de la siguiente manera: Se toma el primer elemento del

renglón de permutaciones de restricciones (restricción que corresponde a la primera Entrada) y se intenta aplicar a la primera Entrada del grafo. Para esto, se revisa que exista la Salida m a la cual queremos aplicar dicha restricción, si la Salida existe se procede a modificar el grafo.

Modificación del grafo:

1.- Se crea un nuevo vértice central (V_{2n+2}) el cual recibe la llegada de la Entrada.

2.- Reemplazamos los vértices de la arista (V_1, V_{2n+1}) por (V_1, V_{2n+2}), de manera que ya no existe el acceso del primer vértice al vértice central que originalmente se tomó.

3.- Se generan nuevas aristas del vértice V_1 al resto de las Salidas (vértices externos), excepto si existe la Salida (V_{2n+1}, V_2) y al vértice de la restricción m, V_m.

Se repiten los pasos 2 y 3 para cada elemento j del renglón de permutaciones de restricciones y se aplica a cada una de las Entradas j de la clase de equivalencia. Para cada Entrada j en los pasos 2 y 3 se considera el mismo vértice central creado en el paso 1, es decir, al final del proceso tenemos dos nodos centrales.

Pero si el elemento j del renglón permutaciones de restricciones no se pudo aplicar, es decir, no se encontró la salida m, el grafo deja de ser modificado y pasamos a la siguiente permutación de restricciones. De esta manera evitamos tener que revisar los demás elementos del renglón de permutaciones que se está revisando.

Así pues, tenemos los grafos modificados de cada uno de los renglones de permutaciones de restricciones que pudieron ser aplicados a una clase de equivalencia.

Tercera revisión

En la tercera y última revisión se toma el grafo modificado y se revisa que sea una configuración válida, esto es que para cada Salida exista al menos una Entrada que no sea ella misma, y que para cada Entrada exista al menos una Salida, que de igual manera no sea ella misma. En otras palabras, se revisa que si la arista de Entrada es (V_j, V_p) entonces la arista de Salida no sea (V_p, V_j), ésta revisión es para ambos casos, ya que pueden existir Entradas para todas las Salidas, pero no existir Salidas para todas las Entradas o viceversa. En las figuras 4.4 y 4.5 se muestra un ejemplo de una configuración valida y de una no valida respectivamente de una intersección de cuatro calles, en ambos casos se tiene la misma clase de equivalencia pero se aplica distinta permutación de restricciones.

Ejemplo 4.1.3. En la figura 4.4 se muestra la siguiente clase de equivalencia [0, 0, 1, -1], por lo que tenemos tres Entradas a las cuales se le aplica la permutación de restricciones (1, 3, 0). En este caso, el subgrafo correspondiente a dicha clase de equivalencia con las restricciones antes mencionadas, es un ejemplo que cumple la definición una configuración válida.

Ejemplo 4.1.4. En la figura 4.5 de nueva cuenta se tiene la clase de equivalencia [0, 0, 1, -1], ahora con la permutación de restricciones siguiente (0, 3, 2). En este caso el subrafo que representa ésta situación no cumple con la definición de una configuración válida, ya que se cumple que para cada Entrada exista una Salida, pero no cumple que para cada Salida exista una Entrada como es el caso de la arista (9,2).

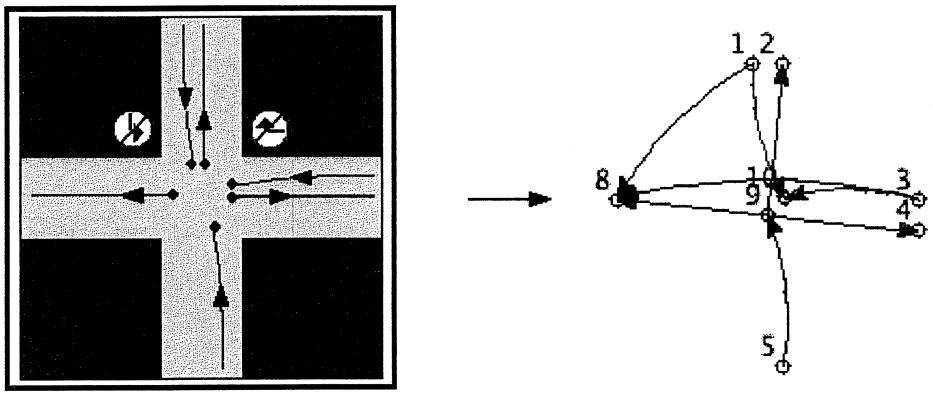


Figura 4.4: Subgrafo válido

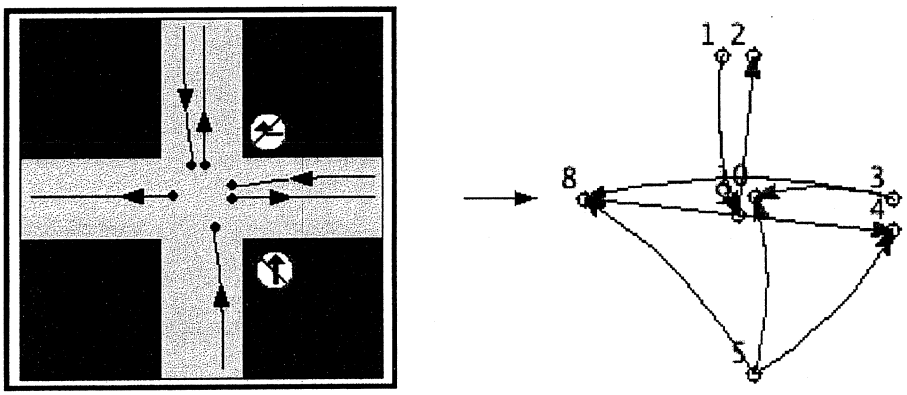


Figura 4.5: Subgrafo no válido

4.2 Resultados de la ejecución del algoritmo que encuentra las configuraciones válidas

En la tabla III se presentan resultados que se obtuvieron de la ejecución de la implementación en C++ del algoritmo (véase apéndice C). Se muestra el número de casos para la intersección de 3, 4, 5 y 6 calles.

Tabla III: Configuraciones válidas

n	Configuraciones Válidas
3	20
4	307
5	2 613
6	18 519

Se puede observar en esta tabla que el número de configuraciones válidas aumenta rapidamente en lo que parece ser una progresión geométrica como función de n. Estos resultados muestran el por qué de una solución algorítmica.

A continuación se presentan los subgrafos correspondientes a las configuraciones válidas para el caso $n = 3$ (véase la figura 4.6). En cada uno de los subgrafos se muestra la información siguiente:

1. Un vector de dimensión 3 con la cantidad de cada tipo de calles que se muestran en el subgrafo. En la primer componente se encuentran las calles del tipo D, en la segunda del tipo E y en la tercera las del tipo S.

2. Una clase de equivalencia de una permutación de los tipos de calles.

3. Una permutación de los tipos de restricciones aplicadas a dicho subgrafo.

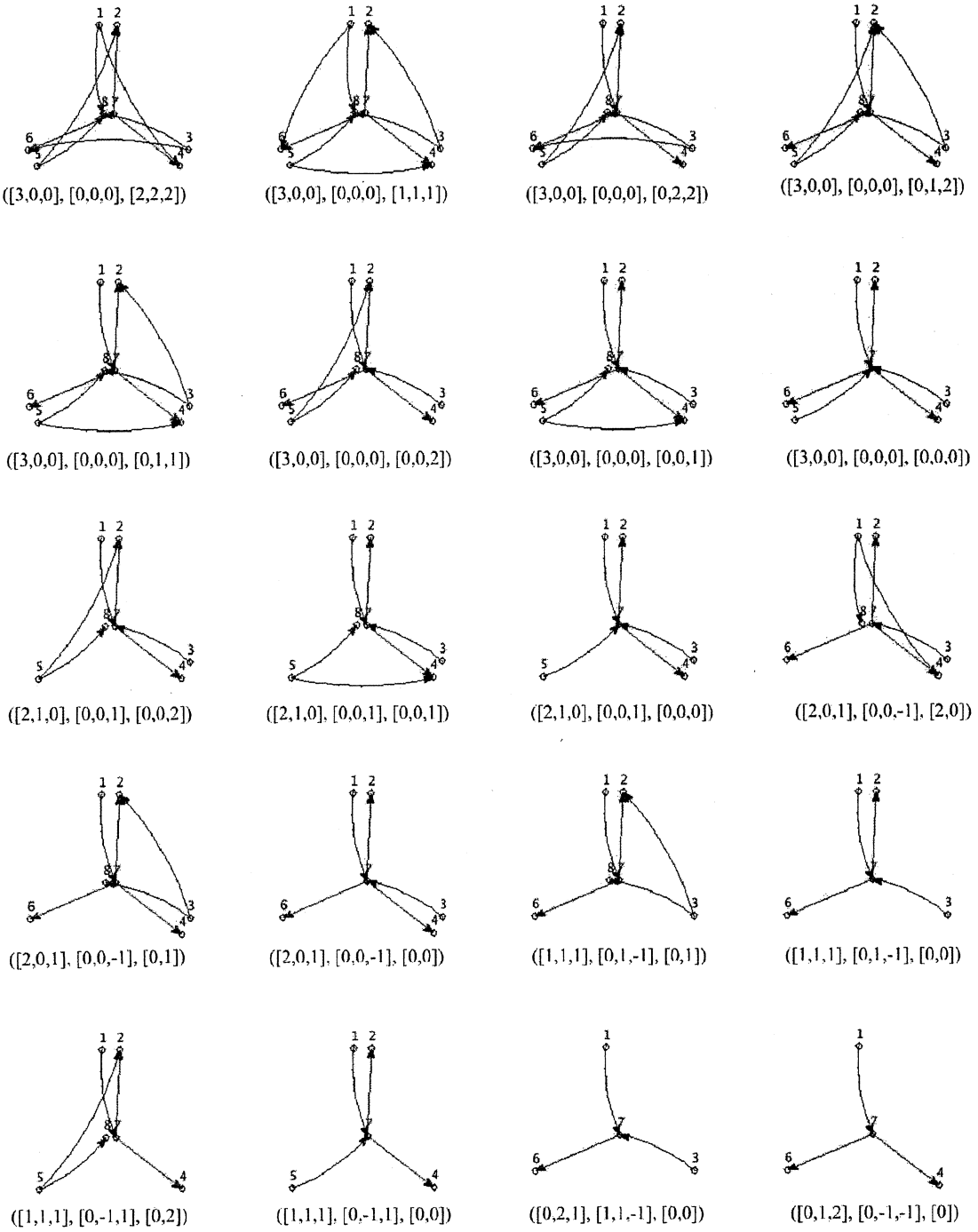


Figura 4.6: Configuraciones válidas para el caso de $n = 3$

Recordemos que los tipos de calles se representan como: 0 a las calles de doble sentido, 1 a las Entradas al nodo central y -1 a las Salidas del nodo central. En este caso, el total de tipos de restricciones que se pueden aplicar son 3 y se representan de la siguiente manera:

0 libre acceso a todas las calles.

1 no hay acceso a la primera calle inmediatamente a la izquierda (no vuelta a la izquierda).

2 no hay acceso a la segunda calle inmediatamente a la izquierda (no vuelta a la derecha).

Ejemplo 4.2.1. ([2, 1, 0], [0, 0, 1], [0, 0, 1])

Este ejemplo representa una intersección de dos calles del tipo 0 (doble sentido), una de tipo 1 (Entrada al nodo central) y cero del tipo -1 (Salidas del nodo central), esta información se muestra en el primer vector. En el segundo vector, se muestra la clase de equivalencia, es decir el orden de los tipos de calles. Por último, se muestra que en las dos primeras entradas de las calles del tipo 0, existe libre acceso a todas las calles y en la tercera calle que es del tipo 1, no esta permitida la vuelta a la izquierda.

4.3 Complejidad del algoritmo

La complejidad de este algoritmo se encuentra en el ciclo del análisis de los diferentes tipos de calles. En la siguiente tabla se muestran las partes principales de este ciclo que contribuyen para determinar la complejidad del algoritmo.

por lo que tenemos un tiempo de:

Tabla IV: En la siguiente tabla se muestran los ciclos que contribuyen a la complejidad de algoritmo.

Ciclos	**Tiempo de contribución**
Generación de la tabla de los diferentes tipos de calles	n^2
Cálculo de las permutaciones de los tipos de calles	$n!$
Clases de equivalencia	$(n-1)!$
Llenado de Entradas y Salidas de una clase de equivalencia	n
Primera revisión	n^2
Aplicación de la lista restricciones correspondiente	$\frac{(n+1)^n}{n}$
Cálculo de las permutaciones de los tipos de restricciones	$n!$
Tipos de restricciones	$n!$
Segunda revisión	n^3
Tercera revisión.	n^2

$$T(n) = C(n^2(n!)(n-1)!(n)n^2\tfrac{(n+1)^n}{n}(n!)(n!)n^3n^2)$$

$$T(n) = O(n^8(n!)^3(n+1)^n) \tag{4.3.1}$$

El resto del algoritmo tiene una complejidad de $O(n^3(n+1)^n)$, que se refiere a la generación de tablas de restricciones y a la tabla de los tipos de calles.

4.4 Cota superior en el número de vértices y aristas que el algoritmo añade al grafo original

Originalmente cada intersección de calles cuenta con un vértice y a lo más $2n$ aristas. Cuando se modifica una intersección, el algoritmo reemplaza al vértice de la intersección por a lo más $2(n+1)$ vértices y se *añaden* n^2 aristas, donde n representa el número de calles que existen en dicha intersección.

Considerando el caso más extremo donde se modifiquen todos los vértices de la ciudad, y dado que en promedio las intersecciones constan de 4 calles, entonces se tendrá que el grafo pasa de $\#V$ a $10\#V$ vértices y de $\#A$ aristas a $\#A + 16\#V$ aristas.

Así, tomando en cuenta que el algoritmo de Dikstra tiene complejidad $O(V^2)$ vemos que el tiempo de computo del camino más corto pasa de ser cV^2 a $100cV^2$.

Capítulo 5

CONCLUSIONES

Este algoritmo le da respuesta al problema planteado, esto es, encuentra las configuraciones válidas de restricciones que se pueden aplicar a una intersección de n calles. En particular:

1. Permite encontrar todos los casos, de una restricción por Entrada, que se presentan en una intersección de n calles. *Además el método utilizado puede ser generalizado para el caso de más restricciones por Entrada.*

2. Permite obtener los subgrafos modificados, para la implementación de un programa de ruteo. Esto facilita el desarrollo de un programa de edición que permita incorporar las restricciones en una intersección.

3. La complejidad del algoritmo es de $O(n^8(n!)^3(n+1)^n)$, donde n es el número de calles en la intersección.

4. El algoritmo solo se necesita realizar una vez para cada n. De esta manera y considerando que en una ciudad normal se tienen intersecciones con a lo más 6

calles, el requerimiento computacional no es muy grande.

5. Como se ve en la sección 4.4 el aumento en el número de vértices y aristas en el grafo de la ciudad crece como $2(n+1)$ por cada vértice y n^2 aristas de cero existentes por vértice. La importancia de esto es que aún con el aumento en el tamaño del grafo, el algoritmo de Dijkstra sigue siendo una opción muy viable para resolver el problema de ruteo tomando en cuenta las reglas de tránsito.

5.1 Proyectos futuros

- Se pretende generalizar el algoritmo para el caso de más de una restricción por Entrada.

- Se pretende incorporar el algoritmo a un proceso que solucione de manera eficiente el problema de ruteo y que permita incluir las reglas de tránsito en las intersecciones.

Apéndice A

CONCEPTOS Y DEFINICIONES SOBRE TEORÍA DE CONJUNTOS

A.1 Definiciones

A.1.1 Relación de equivalencia:

La relación binaria, $\sim$, sobre un conjunto A se dice que es una relación de equivalencia sobre A si para a, b, c cualesquiera en A:

1. $a \sim a$;
2. $a \sim b$ implica $b \sim a$;
3. $a \sim b$ y $b \sim c$ implica $a \sim c$.

La primera de estas propiedades se llama *reflexividad,* la segunda, *simetria* y, la tercera, *transitividad.*

A.1.2 Clase de equivalencia

Si A es un conjunto y $\sim$ es una relación de equivalencia sobre A, entonces la clase de equivalencia de $a \in A$ es el conjunto $\{x \in A : a \sim x\}$. Lo escribimos $\mathrm{cl}(a)$.

Teorema A.1.1. *Las distintas clases de equivalencia de una relación de equivalencia sobre A nos proporcionan una descomposición de A como una unión de subconjuntos mutuamente ajenos. Recíprocamente, dada una descomposición de A como unión de subconjuntos mutuamente ajenos y no vacios, podemos definir una relación de equivalencia sobre A para la que estos subconjuntos sean las distintas clases de equivalencia.*

Demostración. En [3] pag. 18. □

Apéndice B

ANÁLISIS Y COMPLEJIDAD DEL ALGORITMO DE DIJKSTRA

En el capítulo de antecedentes se presentó de forma breve la descripción y funcionamiento del algortitmo de Dijkstra para determinar el camino más corto entre dos vértices, en éste capítulo nuevamente se presenta éste algoritmo pero de una forma general. Además se presenta un análisis detallado de la complejidad del mismo.

B.1 Algoritmo de Dijkstra

El algoritmo de Dijkstra resuelve el problema de los caminos mas cortos en un grafo $G = (V, A)$ convexo dirigido y con pesos no negativos en sus aristas, por lo tanto, asumiremos que el peso $w(u, v) \geq 0$ para cada arista $(u, v) \in A$. Esto se lleva a cabo partiendo de un nodo $a \in V$ que es el vértice de origen. El algoritmo utiliza una

técnica llamada *relajación*, que decrementa de manera progresiva un *estimado del peso* $d(v)$, del camino más corto del vértice de origen a cada vértice $v \in V$. Asímismo, en este proceso de relajación se mantiene para cada vértice $v \in V$, un *predecesor* $\pi(v)$, que al final del algoritmo contiene la información de cual es el vértice que le precede en el camino más corto del vértice origen a v. Asi, los conjuntos V y $A_\pi = \{(\pi(v), v) \in A : v \in V \backslash \{a\}\}$ forman un árbol que contiene el camino más corto del vértice de origen a cada vértice de V.

Para comenzar la relajación se requiere inicializar los predecesores y el estimado del peso:

Inicializa (G, a)

1 Para cada $v \in V$

2 $\quad d(v) \leftarrow \infty$

3 $\quad \pi(v) \leftarrow$ **NULO**

4 $d(a) \leftarrow 0$

La relajación en v, requiere de w el peso de la arista (u, v):

Relajación (u, v, w)

1 Si $d(v) > d(u) + w(u, v)$

2 $\quad$ entonces $d(v) \leftarrow d(u) + w(u, v)$

3 $\quad\quad\quad\quad \pi(v) \leftarrow u$

El algoritmo de Dijkstra mantiene un conjunto S de vértices cuyo camino más corto del vértice origen ya se determinó. El algoritmo selecciona un vértice de $u \in$

$V \backslash S$ con el estimado $d(u)$ mínimo, añade u a S y relaja las aristas que salen de u. En la siguiente implementación se usará una cola de prioridad mínima Q de vértices, enfocada en los valores de $d(u)$.

Dijkstra (G, w, a)

1 Inicializa(G, a)

2 $S \leftarrow$ **NULO**

3 $Q \leftarrow V$

4 Mientras $Q \neq$ **NULO**

5 $u \leftarrow$ extrae el vértice de Q que tenga el peso estimado mínimo

6 $S \leftarrow S \cup \{u\}$

7 Para cada vértice v que se encuentra en los adyacentes de u

8 Relajación (u, v, w)

Por lo tanto, al finalizar el algoritmo, en $\pi(z)$ tendremos la distancia mínima entre a y cada uno de los vértices $z \in V \backslash \{a\}$ [2].

B.2 Análisis y complejidad

El algoritmo mantiene una cola de prioridad mínima Q llamando a tres operaciones de colas: *Insertar* (implícita en la linea 3), *Extraer-mínimo* (linea 5), y *Decrementar-llave* (implícita en *Relajamiento*, la cual es llamada en la linea 8). *Insertar* es invocado una vez por vértice, así como en *Extraer-mínimo*. Ya que cada $v \in V$ es

agregado al conjunto S exactamente una vez, cada arista en la lista de adyacencia es examinado en el ciclo de de las lineas 7-8 exactamente una vez durante el curso del algoritmo. Ya que el número total de arista en todas las lineas de adyacencia es $|A|$, hay un total $|A|$ iteraciones para este ciclo, y así un total de a lo más $|A|$ *Decrementar-llave* operaciones.

El tiempo de ejecución del algoritmo de Dijkstra depende de como se implemente la cola de prioridad mínima. En el caso de implementar la cola Q indexando los vértices del 1 a $|V|$, simplemente almacenamos $d(v)$ en la v entrada de un arreglo. Cada operación de *Insertar* y *Decrementar-llave* toma un tiempo de $O(1)$, y cada operación *Extraer-mínimo* toma un tiempo $O(V)$ (ya que tenemos que buscar en todo el arreglo), para un tiempo total de $O(V^2 + A) = O(V^2)$ [2].

Apéndice C

IMPLEMENTACIÓN EN C++ DEL ALGORITMO

C.1 Algoritmo para determinar Configuraciones Válidas

```
/*********************************************************************
*
*   (C) 2006 Yadira Perez Garibay
*
*    Algoritmo:
*   Este algoritmo determina configuraciones validas de grafos
*   dirigidos con algunas restricciones dadas.
*   Estos grafos representan la interseccion de $n$ calles, donde
*   la interseccion  esta representada por v\'ertices y las calles
*   por aristas.
*   Existen 3 tipos de calles: D = doble sentido, E = Entrada al
*   nodo y S = Salida al nodo (Entrada y Salida del nodo central).
*   Las restricciones pueden ser No vuelta a la derecha, izquierda,
*   seguir de frente, etc; dependiendo en numero de calles.
*
*********************************************************************/

#include <iostream>
#include <fstream>
#include <math.h>
#include <vector>
```

```
#include <map>
#include <set>
#include <list>
#include <iterator>
using namespace std;

typedef struct Arista
{
  long v;
  long w;
}arista;

typedef vector<long> listas;
typedef vector<listas> tablas;

typedef struct Conf
{
  long n;
  listas tipo_calle;
  listas restriccion;
  vector<arista> Ent;
  vector<arista> Sal;
} Configuracion;

//FUNCIONES
vector<Configuracion> Configuracion_valida(long ncalles);
vector<tablas> listas_restricciones(long ncalles);
listas genera_renglon( long dividendo, long base, long ncalles);
tablas conjunto_calles(long ncalles);
long multinomial(long n_total,listas denominador);
long fact(long n);
tablas clase_equivalencia(long ncalles, long P, listas permutacion_C);
void EntradasySalidas(long ncalles, listas permutacion,
         vector<arista> &Entrada, vector<arista> &Salida);
tablas crearlista_permutaciones(long P, listas conjunto_Rest,
         long nEntradas);
bool primera_revision( vector<arista>Entrada,  vector<arista>Salida);
bool segunda_revisionymodificacion_grafo(long ncalles, listas
   permutacion, vector<arista>&Entrada, vector<arista>&Salida);
bool tercera_revision( long ncalles, vector<arista>Entrada,
```

```
        vector<arista>Salida);
listas cambio_raro(listas conjunto_restricciones);
listas llenado_tipo_calle(listas conjunto_calle);

int main(int argc, char *argv[])
{
  if (argc != 2)
  {
    cerr << "Syntax: " << argv[0] << " <ncalles>" << endl;
    exit(-1);
  }
  long ncalles = atol(argv[1]);
  vector<Configuracion> lista_conf_validas; //<- contiene todas
        //las configuraciones validas
  lista_conf_validas = Configuracion_valida(ncalles);
  cout << "Termine con " << lista_conf_validas.size() <<
       " configuraciones validas" << endl;
  return(0);
}

/*********************************************************************
*   Rutina principal que determina una configuracion valida
*********************************************************************/

vector<Configuracion> Configuracion_valida(long ncalles)
{
  vector<arista> Entrada, EntradaVieja; // Entrada <- crear lista de
     //elementos (v,w)
  vector<arista> Salida, SalidaVieja;  // Salida <- crear lista de
     //elementos (v,w)
  listas conjunto_R, conjunto_C, generar_grafo, vector_restricciones;
  int ii, jj, kk, P1, P2, nEntradas, mm;
  tablas P_restricciones, tabla_calles, lista_restricciones;
  vector<tablas> tablas_restricciones;
     //configuracion_valida <- crear una estructura que contiene
     //(n,[0,1,1,..],(1,m,3,..),Entrada,Salida)
  Configuracion conf_valida;
  vector<Configuracion> lista_conf_validas; //   <- contiene  todas
     //las configuraciones validas
  conf_valida.n = ncalles;
```

```
tablas_restricciones = listas_restricciones(ncalles); //Genera todas
   //las tablas de restricciones del 1 al ncalles
tabla_calles = conjunto_calles(ncalles); //Genera la tabla de los
   //conjuntos de  calles
int tamtablacalles = tabla_calles.size();
for (ii = 1; ii <= tabla_calles.size(); ii++)
{
  conjunto_C = tabla_calles[ii-1];
  P1 = multinomial(ncalles, conjunto_C);
  tablas clase_lateral = clase_equivalencia(ncalles, P1,
   llenado_tipo_calle (conjunto_C));
     //Obtengo las clases de equivalencia

  cout << endl << "Cuando la calle es:" << endl;
  for(int ii=1;ii<=conjunto_C.size();ii++)
    cout << conjunto_C[ii-1] << " " ;
    cout << endl << "entonces P1="<< P1 << endl;
    int tamtabclase = clase_lateral.size();
  for(int ii=1; ii<=tamtabclase; ii++)
  {
    listas renglon3=clase_lateral[ii-1];
    int tamrenglon=renglon3.size();
    for (int jj=1; jj<=tamrenglon; jj++)
    {
      cout << renglon3[jj-1] << " " ;
    }
    cout << endl;
  }
  cout << endl;
  nEntradas =  conjunto_C[0] + conjunto_C[1];
  if (nEntradas != 0)
  {
    lista_restricciones = tablas_restricciones[nEntradas-1];
    for (jj = 1; jj <= clase_lateral.size(); jj++)
    {
      generar_grafo = clase_lateral[ jj-1 ];
      EntradasySalidas(ncalles, generar_grafo, Entrada, Salida);
         //se llenan las listas Entrada y Salida
      cout << "Usando la clase lateral" << endl;
      listas renglon3=generar_grafo;
```

```
int tamrenglon=renglon3.size();
for (int r=1; r<=tamrenglon; r++)
{
  cout << renglon3[r-1] << " " ;
}
cout << endl;
if (primera_revision(Entrada, Salida) == false)
{
  conf_valida.tipo_calle = generar_grafo;
  int tamlistarestricc = lista_restricciones.size();
  EntradaVieja.assign(Entrada.begin(),Entrada.end());
  SalidaVieja.assign(Salida.begin(),Salida.end());
  for (kk = 1; kk <= lista_restricciones.size(); kk++)
  {
    conjunto_R = lista_restricciones[kk-1];
    P2 = multinomial(nEntradas, conjunto_R);
    if( (ii-1) == 0 )
    {
      P_restricciones = clase_equivalencia(nEntradas, P2,
       cambio_raro(conjunto_R));
    }
    else
      P_restricciones = crearlista_permutaciones( P2,
       cambio_raro(conjunto_R), nEntradas);
        //aqui hace el cambio "raro"

    int tamprestricc = P_restricciones.size();
    for ( mm = 1; mm <= P_restricciones.size(); mm++)
    {
      vector_restricciones = P_restricciones[mm-1];
      Entrada.assign(EntradaVieja.begin(),EntradaVieja.end());
      Salida.assign(SalidaVieja.begin(),SalidaVieja.end());
      if (segunda_revisionymodificacion_grafo (ncalles,
       vector_restricciones, Entrada, Salida ))
      {
        if (tercera_revision( ncalles, Entrada, Salida ))
        {
          conf_valida.restriccion =  vector_restricciones;
          conf_validaEntassign(Entradabegin(),Entradaend());
          conf_validaSalassign(Salidabegin(), Salidaend());
```

```
                lista_conf_validas.push_back (conf_valida);
                   //Aniade esta configuracion a la lista de
                   //configuraciones validas
                cout << "Paso la tercera revision, lo que se guardo
                 es la siguiente informacion:" << endl;
                cout << "(" << conf_valida.n << ", [";
                for(int pp=1;pp<=conf_validarestriccionsize();pp++)
                  cout <<conf_valida.restriccion[pp-1] << " ";
                  cout << "], E={";
                for(int ii=1; ii<=conf_valida.Ent.size(); ii++)
                  cout << "(" << conf_valida.Ent[ii-1].v << ", " <<
                   conf_valida.Ent[ii-1].w << ")  ";
                  cout << "}, S={";
                for(int ii=1; ii<=conf_valida.Sal.size(); ii++)
                  cout << "(" << conf_valida.Sal[ii-1].v << ", " <<
                   conf_valida.Sal[ii-1].w << ")  ";
                  cout << "} )" << endl << endl;
              }
            }
          }
        }
      }
      else  //Si primera_revision(Entrada, Salida) == true
      break;
    }
  }
}
  return (lista_conf_validas);
}

/*********************************************************************
*   Se generan ncalles tablas que c/u contiene conjuntos de
*   restricciones, y dichos conjuntos contienen la cantidad del
*   tipo de restricciones; cada tabla corresponde al numero de
*   entradas que contiene el grafo.
*********************************************************************/

vector<tablas> listas_restricciones(long ncalles)
{
  vector<tablas> lista_num(ncalles);
```

```
  for (int kk = ncalles; kk >=  1; kk--) //Como el maximo
     //de Entradas = ncalles disminuiremos ncalles es
     //decir nEntradas
  {
    long nmax = (long)pow((double)(kk + 1),(double)ncalles) - 1;
    for (int i = kk ; i <= nmax; i += kk)
    {
      listas renglonRestricciones = genera_renglon(i, kk + 1, ncalles);
```

SE HACE MENCIÓN EN "COMPLEJIDAD DEL ALGORITMO"

```
      long suma = 0;
      for (int jj = 1; jj <= ncalles; jj++)
        suma += renglonRestricciones[ jj-1 ];
          if (suma == kk)
            lista_num[ kk-1 ].push_back(renglonRestricciones);
    }
  }
  return(lista_num);
}

/*********************************************************************
* Rutina que genera un renglon de la tabla de restricciones
*********************************************************************/

listas genera_renglon(long dividendo, long base, long ncalles)
{
  listas renglonRestricciones(ncalles);

  for(int ii=0; ii <= ncalles-1; ii++) //inicializa el vector con
     //ceros
  renglonRestricciones[ii]=0;
  for(int ii = ncalles-1; ii>=0; ii--)
  {
    long cociente = dividendo / base;  //Division entera
    long residuo = dividendo % base;  //residuo
    renglonRestricciones[ii] = residuo;
    if (cociente <= (base-1))
    {
      renglonRestricciones[ii-1] = cociente;
```

```
      break;
    }
    else
    dividendo = cociente;
  }
  return(renglonRestricciones);
}

/*********************************************************************
* Genera los conjuntos de los tipos de calles, los cuales nos
* dice la cantidad de cada tipo de calle que hay en el conjunto.
* (tabla de combinacion)
*********************************************************************/

tablas conjunto_calles(long ncalles)
{
  listas renglon_calles(3);  // vector que contiene la canti-
     //dad de cada tipo de calles ( dim 3)
  tablas lista_conjuntos;
     // Contiene  a todos los conjunto_calles

  for (int i = ncalles; i>= 0; i--)
  {
    for (int j = ncalles - i; j>= 0; j--)
    {
      renglon_calles[0] = i;
      renglon_calles[1] = j;
      renglon_calles[2] = ncalles - i - j;
      lista_conjuntos.push_back(renglon_calles);
    }
  }
  return (lista_conjuntos);
}

/*********************************************************************
* Calculo de permutaciones
*********************************************************************/
long multinomial(long n_total, listas denominador)
{
  long m = 1;
```

```
  for (int ii = 1; ii<= denominador.size(); ii++)
    m = m * fact(denominador[ii-1]);  //denominador,
  long P = fact(n_total) / m;
  return(P);
}

/*********************************************************************
* Rutina para calcular el factorial
*********************************************************************/
long fact( long n)
{
  long res=1;
  for (int i = 2; i<= n; i++)
    res = res * i;
  return(res);
}

/*********************************************************************
* Llena el vector de correspondiente al conjunto calle
*********************************************************************/
listas llenado_tipo_calle(listas conjunto_calle)
{
  listas permutacion; // crea un vector de tamanio #calles
  int i;

  for (i = 1; i <=conjunto_calle[ 0 ]; i++)
    permutacion.push_back(0);
  for (i = 1; i <= conjunto_calle[ 1 ]; i++)
    permutacion.push_back(1);
  for (i = 1; i <= conjunto_calle[ 2 ]; i++)
    permutacion.push_back(-1);
  return(permutacion);
}

/*********************************************************************
* Genera las distintas clases de equivalencia de un conjunto de
* calles
*********************************************************************/
tablas clase_equivalencia(long ncalles, long P,listas permutacion_C)
{
```

```
  int i, m, k;
  long lswap;
  tablas clase_lateral; // crea una lista de vectores
     //(lo que seran las clases de equivalencia)

  if (P <= ncalles)
    clase_lateral.push_back(permutacion_C);
  if (P > ncalles)
  {
    clase_lateral.push_back(permutacion_C);
    m = 1;
    if ((P % ncalles) == 0) // P mod ncalles == 0
      k = P / ncalles;
    else
      k = (P / ncalles)  + 1;
    for (i = permutacion_C.size(); i >= 3; i--)
    {
      if (permutacion_C[ i-1-1] != permutacion_C[ i-1 ])
        {
          lswap = permutacion_C[i-1]; //intercambiar i por i-1
             //en permutacion
          permutacion_C[i-1]=permutacion_C[i-1-1];
          permutacion_C[i-1-1]=lswap;
          clase_lateral.push_back(permutacion_C);
             //agregar permutacion a clase_equivalencia
          m = m+1;
          if (m == k) break;
        }
      }
      if ( (i == 3) && (m != k) )
        i = permutacion_C.size();
    }
    return(clase_lateral);
}


/*********************************************************************
* Al finalizar este procedimiento se llenan las listas de Entrada
* y Salida que representan una permutacion,esto es, depen-
* diendo de la permutacion, se aniade una arista a Entrada, Sa-
```

```
* lida o a ambos.
*****************************************************************/
void EntradasySalidas(long ncalles, listas permutacion,
 vector<arista> &Entrada, vector<arista> &Salida)
{
  arista aristaTemp;
  Entrada.clear();
  Salida.clear();

  for (int i = 2; i <= 2*ncalles; i+=2)
  {
    switch (permutacion[i/2-1])
    {
      case 0:
        aristaTemp.v = 2*ncalles+1;
        aristaTemp.w = i;
        Salida.push_back(aristaTemp);

        aristaTemp.v = i-1;
        aristaTemp.w = 2*ncalles+1;
        Entrada.push_back(aristaTemp);
        break;

      case 1:
        aristaTemp.v = i-1;
        aristaTemp.w = 2*ncalles+1;
        Entrada.push_back(aristaTemp);
        break;

      case -1:
        aristaTemp.v = 2*ncalles+1;
        aristaTemp.w = i;
        Salida.push_back(aristaTemp);
        break;
    }
  }
}
```

SE HACE MENCIÓN EN 4.1.4

```
/*Realiza el cambio raro de restricciones*/
listas cambio_raro(listas conjunto_restricciones)
{
  listas permutacion;// crear un vector de dim nEntradas

  for (int tt = 1; tt <= conjunto_restricciones.size(); tt++)
     //conjunto_restricciones.size() = ncalles
  {
    for(int ii = 1; ii <= conjunto_restricciones[tt-1]; ii++)
      permutacion.push_back(tt-1);
  } //Estas lineas de codigo son las del cambio "raro"
  return(permutacion);
}

/*********************************************************************
* Crea las permutaciones de un conjunto de restricciones
*********************************************************************/
tablas crearlista_permutaciones(long P, listas conjunto_Rest,
 //long nEntradas)
{
  tablas lista_permutaciones;
     //lista de permutaciones de restricciones
  long lswap;

  if (P == 1)
    lista_permutaciones.push_back(conjunto_Rest);
  if (P > 1)
  {
    lista_permutaciones.push_back(conjunto_Rest);
    int k = conjunto_Rest.size();
    for (int i = 1; i <= (P - 1); i++)
    {
      if (conjunto_Rest[k-1]  != conjunto_Rest[k -1-1])
      {
        lswap = conjunto_Rest[k-1];
           //intercambiar  k   por  k-1 en  permutacion
        conjunto_Rest[k-1]=conjunto_Rest[k-1-1];
        conjunto_Rest[k-1-1]=lswap;
        k  = k-1;
        lista_permutaciones.push_back(conjunto_Rest);
```

```
        if ( k == 1 )
          k = conjunto_Rest.size();
      }
    }
  }
  return(lista_permutaciones);
}

/*********************************************************************
* Realiza la primera revision de una clase lateral
*********************************************************************/

bool primera_revision( vector<arista> Entrada, vector<arista>
 Salida)
{
  bool eliminar = false;
  arista Ent, Sal;
  listas S(Salida.size());

  for (int ii = 1; ii <= Salida.size(); ii++)
    S[ii-1] = 0;
  if ( (Entrada.size() !=  0) && (Salida.size() != 0) )
  {
    for (int rr = 1; rr <= Entrada.size(); rr++)
    {
      Ent = Entrada[ rr-1 ];
      for (int tt = 1; tt <= Salida.size(); tt++)
      {
        Sal = Salida[ tt-1 ];
        if( ((rr-1) && (tt-1) == 0) && (Salida.size() == 1))
        {
          if ( (Ent.w == Sal.v) && (Ent.v +1 == Sal.w))
            eliminar = true;
            break;
        }
        if ( (Ent.w == Sal.v) && (Ent.v +1 != Sal.w))
          S[tt-1] = S[tt-1] + 1;
      }
    }
    for (int jj = 1; jj <= Salida.size(); jj++)
```

```
    {
    if (S[jj-1] == 0)
    {
      eliminar = true;
      break;
    }
  }
}
else
  eliminar = true;
return(eliminar);
}

/*********************************************************************
* Realiza una segunda revision al estar aplicando el verctor de
* restricciones y modificando el grafo
*********************************************************************/

bool segunda_revisionymodificacion_grafo( long ncalles, listas
 permutacion, vector<arista> &Entrada, vector<arista> &Salida)
{
  arista Ent, Sal, aristaTemp;
  bool encontro;
  long v_partida,v_partida_reloj, reloj, v_destino;

  long n  =  2*ncalles + 1;
  long v  =  ncalles;
  for(int ss =  1; ss <= permutacion.size(); ss++)
  {
    if (permutacion[ss-1] > 0)
    {
      Ent = Entrada[ss-1];
      v_partida = Ent.v;
      v_partida_reloj = (long)(Ent.v+1)/2;
      reloj = permutacion[ss-1];
      v_destino = (((v_partida_reloj + reloj - 1) % v) + 1) * 2;
      encontro = false;
         // Buscamos la arista de salida al v_destino
      for (int m = 1; m <= Salida.size(); m++)
      {
```

```
        Sal = Salida[m-1];
        if (Sal.w == v_destino)
        {
          Entrada[ss-1].v = v_partida;
          Entrada[ss-1].w = n+1;
          n++;
             //Crear aristas de v_partida a todos los nodos
             //restantes (menos al nodo central)
          for (int pp = 1; pp <= Salida.size(); pp++)
          {
            Sal = Salida[pp-1];
            if ( (Sal.w != (v_partida+1)) && (Sal.w != v_destino))
            {
              aristaTemp.v = v_partida;
              aristaTemp.w = Sal.w;
              Entrada.push_back(aristaTemp);
                 //agregar a   Entrada <- (v_partida, Salida.y)
            }
          }
          encontro = true;
          break; // salir del ciclo
        }
      }
    }
    if (encontro == false)
    {
      break;
    }  //Si vector_restricciones== 0 no hay modificaciones
  }
  return(encontro);
}

/*********************************************************************
* Tercera y ultima revision que verifica que para cada entrada
* existe al menos una salida y que para cada salida existe al
* menos una entrada. (Grafo modificado)
*********************************************************************/

bool tercera_revision( long ncalles, vector<arista> Entrada,
 vector<arista> Salida)
```

```
{
  arista Ent, Sal;
  map<long, int, less<long> > contador;

  for (int ii = 1; ii <= Salida.size(); ii++)
  {
    Sal = Salida[ii-1];
    contador[Sal.w] = 0;
  }
  for (int t = 1; t <= Entrada.size(); t++)
  {
    Ent = Entrada[ t-1 ];
    if (Ent.w == 2*ncalles + 1)
    {
      for (int tt = 1; tt <= Salida.size(); tt++)
      {
        Sal = Salida[tt-1];
        if (Ent.v +1 != Sal.w)
        contador[Sal.w] = contador[Sal.w] + 1;
      }
    }
    if (Ent.w < 2*ncalles + 1)
      contador[Ent.w] = contador[Ent.w] + 1;
         //Si Entrada.y > # calles +1 no se incrementa nada
  }
  for (int k = 1; k <= Salida.size(); k++)
  {
    Sal = Salida[k-1];
    if ( contador[Sal.w] == 0 )
    {
      return(false);
    }
  }
     //Ahora se asegurara que toda Salida tenga al menos una Entrada
     //contador.clear();
  for (int ii = 1; ii <= Entrada.size(); ii++)
  {
    Ent = Entrada[ii-1];
    contador[Ent.v] = 0;
  }
```

```
  for(int ii=1; ii<= Entrada.size(); ii++)
  {
    Ent = Entrada[ii-1];
    if(Ent.w > (2*ncalles+1) )
      contador[Ent.v]++;
    if(Ent.w < (2*ncalles+1) )
      contador[Ent.v]--;
  }
  for(int ii=1; ii<= Entrada.size(); ii++)
  {
    Ent = Entrada[ii-1];
    if (contador[Ent.v] == 1)
    {
      return(false);
    }
  }
  return(true);
}
```

Bibliografía

[1] *Algoritmo de Dijkstra.* Obtenido el 18 de enero de 2006 en http://www.alumnos.unican.es/uc900/Algoritmo.htm

[2] CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., STEIN, C: *Introduction to Algorithms*, Second Edition: MIT Press, Cambridge Massachussets, 2003.

[3] HERSTEIN, I. N., *Algebra moderna: grupos, anillos, campos, teoria de Galios*, 2^{da} ed. México: Trillas, 1990.

[4] M. ABELLANEAS - D. LODARES: *Análisis de algoritmos y teoría de grafos:* Macrobit - ra-ma, 1991.

[5] TOLEDO LÓPEZ, A. L., MONTIEL PÉREZ, J. Y: *Generador de rutas de viaje utilizando un sistema de información geográfica y procesamiento de video:* Revista Digital Universitaria, 10 agosto 2004, Vol 5 Num 7, pp. 1-16: http://www.revista.unam.mx/vol.7/num1/art03/art03.htm.