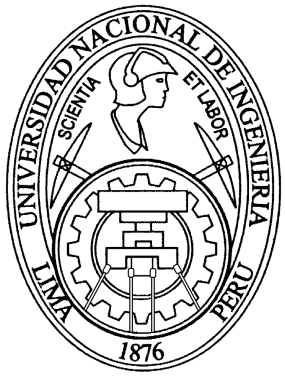
UNIVERSIDAD NACIONAL DE INGENIERÍA

FACULTAD DE CIENCIAS

ESCUELA PROFESIONAL DE MATEMÁTICA



TESIS PARA OPTAR EL GRADO EL TITULO PROFESIONAL DE

LICENCIATURA EN MATEMÁTICAS

TITULADA:

ALGORITMOS PARA CAMINOS MÍNIMOS

PRESENTADA POR

JOHANI OLIVOS IPARRAGUIRRE

LIMA-PERÚ

2009

Este trabajo se lo dedico a mis padres por darme la vida, a mis hermanos y familiares por acompañarme en ella y a Yuriana por darle el sentido y ser la razón

Agradezco a mi asesor, el profesor Lic. Rósulo Pérez, por todas las molestias que se tomo, por su apoyo y por la motivación en el tema.

Índice general

Resumen

En el presente trabajo se estudiará las diferentes aplicaciones prácticas que puede tener *El Problema de Caminos Mínimos* para problemas de optmización de flujos de redes.

Sea el caso en el que nos presenta un problema de decisión, el cual tiene como transfondo decidir cual podria ser la elección apropiada para optimizar un costo, buscando sea el menor posible apartir de una posición inicial hasta una determinada instancia final o destino, para el cual se presenterán mucho caminos por escoger. Para llegar a dicho resultado es necesario conocer las restricciones y condiciones con las que se deberá proceder para la modelación apropiada del problema.

Utilizando entonces la teoria de grafos se puede llegar a obtener un grafo apropiado que nos servirá para la aplicación de los algoritmos que estudiaremos para la obtención del resultado óptimo.

Serán entonces los *Algoritmos de Dijkstra* y el *Algoritmo de Bellman-Ford*, los que serán seleccionados para su respectivo estudio, analizando su implementación, las restricciones que se presentan para ciertos casos y los distintos comportamientos de su tiempo de ejecución, este último buscando que mejore con la utilización de diferentes estrategias.

Es importante recordar que el problema de Caminos Mínimos se presenta en muchas situaciones de aplicación real, como es el caso de transporte, telecomunicaciones, industria, aplicaciones geográficas y planeamientos, lo cual hace relevante su aprendizaje.

Introducción

Los Problemas de Caminos Mínimos son entre los problemas de optimización de flujos de redes, de los más estudiados, ya que tienen varios campos de aplicación.

Desde fines de 1950, más de dos mil trabajos científicos han sido publicados, muchos de estos en revistas especializadas y otros en conferencias, respecto al procedimiento general de optimización combinatoria sobre grafos.

Con la introducción de nuevas tecnologías, las nuevas herramientas posibilitaron el estudio de algunos casos que no podian ser vistos en profundidad debido a los infinitos cálculos que se realizaban, por lo cual la creación de algoritmos que fueran más eficientes ayudo a su estudio.

Es importante en un algoritmo la velocidad de procesamiento de datos, por lo cual se busca medir la complejidad del algoritmo, a partir de un número n de datos que se procesarán, con el cual hallaremos el tiempo que se tarda en dar una respuesta satisfactoria según las operaciones que realice el algoritmo.

Llamaremos entonces algoritmo polinomial a aquel algoritmo tal que el número de pasos para su ejecución crece a lo más tan rápido como un polinomio en $log(n)$, estos son considerados los mejores algoritmos; en caso de tomar una función exponencial, tendremos el algoritmo exponencial.

Luego de varios estudios, fue en 1959, Edsger Dijkstra quien entrego la mejor contribución a la informática, el algoritmo de caminos mínimos o también conocido como *Algoritmo de Dijkstra*, recibiendo el Premio Turing en 1972.

El algoritmo cumplía la función de determinar el camino más corto dado un vértice origen, al resto de vértices en un grafo dirigido y con pesos en cada arista. El algoritmo es un caso particular de la búsqueda de costo uniforme, y como tal, no funciona en grafos con aristas de costo negativo.

Ante tal hecho fue R.Bellman y Ford quienes presentaron su aporte para resolver dicho problema con un algoritmo que posee la idea similar del algoritmo de Dijkstra, solo que utilizando aristas de pesos negativos.

Los algoritmos de Caminos Mínimos tiene aplicaciones en campos como las telecomunicaciones, la investigación de operaciones, la robótica y los diseños. Es precisamente en este trabajo donde mostraremos algunas de estas aplicaciones para el caso del *algoritmo de Dijsktra* y el *algoritmo de Bellman-Ford* respectivamente.

En el capítulo 1 daremos los conceptos preliminares, las herramientas básicas que nos servirán para conocer el campo donde desarrollaremos el tema.

En el capítulo 2 nos enfocaremos a entender un poco acerca de la complejidad de un algoritmo, para poder conocer los tiempos de ejecución y las formas con la que se busca la eficiencia de estos. Además de analizar la clasificación que existe según los tipos de problemas que puedan contener una solución accesible; problemas P, NP y NP-Completo y NP-duro.

En el capítulo 3 veremos algo de estructura de datos, las bases de como iniciar la implementación de un algoritmo y como enfrentar mediante ciertos métodos algunos tipos de prob-

lemas buscando que su tiempo de ejecución sea la de mejor eficiencia, además veremos algunos algoritmos utilizados para la búsqueda de datos.

En el capítulo 4 nos enfocaremos ha analizar con profundidad el problema de caminos mínimos y como poder enfrentarlos, además de explicar los dos algoritmos principales que estudiaremos en este trabajo, como son el algoritmo de Dijkstra y el algoritmo de Bellman-Ford.

El capítulo 5 esta dedicado a las aplicaciones que se dan en algunos campos donde es importante el aporte de los algoritmos de caminos mínimos (Dijkstra y Bellman-Ford) además de la implementación e iteracción de los mismos.

Capítulo 1

Preliminares

1.1. Teoría de Grafos

Los grafos, los objetos que estudiaremos, resultan ser muy útiles para analizar problemas muy diversos, tales como *problemas de asignación de tareas*; buscando ocupar al mayor número de trabajadores, *construcción de redes*; buscando conectar todas las ciudades con el menor coste posible o *problemas de horarios*; buscando minimizar el número de horas necesario para programar, etc.

Los grafos son como veremos, un lenguaje, una notación que nos permite representar relaciones binarias; es decir entre pares de objetos; en un conjunto. En principio podriamos decir que un grafo es una colección de vértices y de aristas que unen estos vértices.

Definición 1.1 *Un grafo G es un par $G = (V, E)$ donde V es un conjunto finito llamado vértices o nodos y E es un multiconjunto (colecciones donde se permite la aparición repetida de los elementos) de pares no ordenados de vértices denotados por $\{x, y\}$ que se denomina aristas*

Denotamos por $V(G)$ al conjunto de vértices del grafo G y por $E(G)$ el conjunto de aristas del grafo G. Denotaremos además por $\mid V(G) \mid$ y $\mid E(G) \mid$ el número de vértices y el número de aristas respectivamente. Puesto que E es un multiconjunto es posible que existan pares repetidos, en este caso G tiene lados múltiples. Cuando un par no ordenado de E tenga el mismo vértice repetido diremos que el lado es un lazo. Si no hay lados múltiples ni lazos decimos que es un grafo simple. Los lados se denotan por pares ordenados, (u, v) denota el lado dirigido que tiene como vértice inicial a u y como vértice terminal a v.

En algunos casos usaremos los símbolos V, E para identificar el conjunto de vértices y el conjunto de aristas respectivamente.

Definición 1.2 *Un grafo simple $G = (V, E)$ consta de V, un conjunto no vacio de vértices y de E, un conjunto de pares no ordenados de elementos distintos de V. A esos pares se les llama aristas o lados*

Un grafo con varias aristas entre 2 vértices se llamará multigrafo

Definición 1.3 *Un multigrafo $G = (V, E)$ consta de un conjunto V de vértices, un conjunto E de aristas y una función f de E en $\{\{u, v\}/u, v \in V, u \neq v\}$. Se dice que las aristas e_1, e_2 son aristas múltiples o paralelas si $f(e_1) = f(e_2)$*

Los multigrafos definidos no admiten lazos (aristas que conectan un vértice consigo mismo)

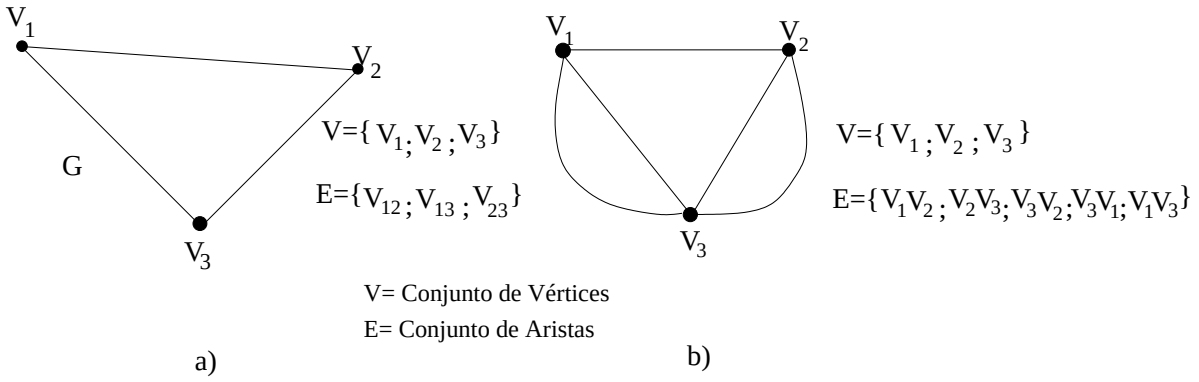


Figura 1.1: a) Grafo Simple - b)Multigrafo

Definición 1.4 *Un pseudografo $G = (V, E)$ consta de una conjunto V de vértices, un conjunto E de aristas y una función f en E en $\{\{u, v\}/u, v \in V\}$. Se dice que una arista e es un lazo si $f(e)=\{u, v\} = \{u\}$ para algún $u \in V$*

Definición 1.5 *. Un grafo dirigido $G = (V, E)$ consta de un conjunto V de vértices, un conjunto E de aristas, que son pares ordenados de elementos de V*

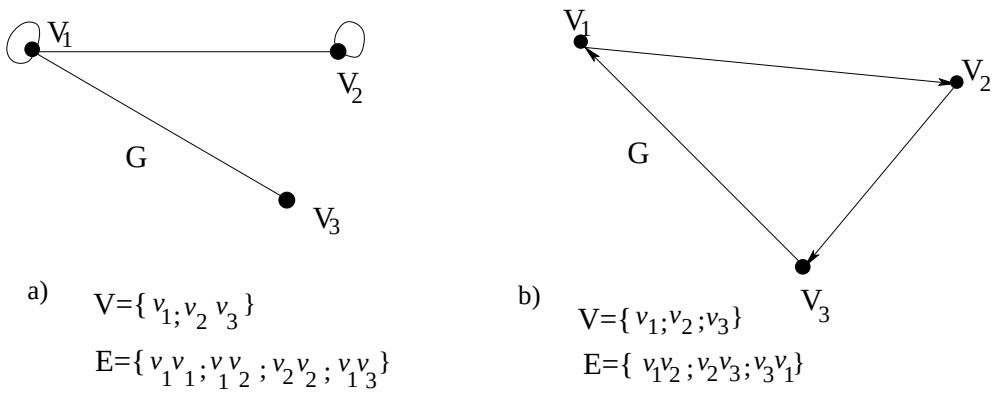


Figura 1.2: a)Pseudografo - b) Grafo Dirigido

Definición 1.6 *Un multigrafo dirigido $G = (V, E)$ consta de un conjunto V de vértices, un conjunto E de aristas y una función f de E en $\{\{u, v\}/u, v \in V\}$. Se dice que las aristas e_1, e_2 son aristas paralelas si $f(e_1) = f(e_2)$*

1.1.1. Adyacencia e Incidencia de vértices

Definición 1.7 *Dos vértices u,v de un grafo $G = (V, E)$ se dicen adyacentes si $\{u, v\} \in E$, es decir si son extremos de una arista.*

Asimismo dos aristas son adyacentes si tienen un mismo vértice como extremo.

Definición 1.8 *Si $e = \{u, v\}$ decimos que el lado e es incidente a los vértices u y v*

1.2. Representación de los grafos

Sea $G = (V, E)$ un grafo dirigido, su *matriz de adyacencia* será denotado por $A = [a_{ij}]$ que esta dada por:

$$a_{ij} = \begin{cases} 1 & \text{si} \quad (v_i, v_j) \in G \\ 0 & \text{si} \quad (v_i, v_j) \notin G \end{cases}$$

Un grafo dirigido G de n vértices y m aristas, la matriz de incidencia de G esta denotada por $B = [b_{ij}]$:

$$b_{ij} = \begin{cases} -1 & \text{si } v_i \text{ es el vértice inicial de la arista } e_j \\ 1 & \text{si } v_i \text{ es el vértice final de la arista } e_j \\ 0 & \text{en los otro casos} \end{cases}$$

El grafo G produce las siguientes matrices de incidencia y adyacencia.

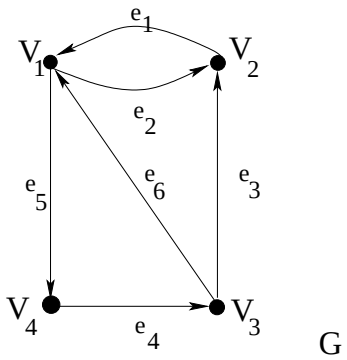


Figura 1.3: Grafo Dirigido

B(G)=

	e_1	e_2	e_3	e_4	e_5	e_6
v_1	−1	1	0	0	0	−1
v_2	1	−1	−1	0	0	0
v_3	0	0	1	−1	0	1
v_4	0	0	0	1	−1	0

A(G)=

	v_1	v_2	v_3	v_4
v_1	0	1	0	1
v_2	1	0	0	0
v_3	1	1	0	0
v_4	0	0	1	0

Esta representación es del orden $O(n^2)$ y es útil para grafos con número de vértices pequeños o grafos densos.

Otra forma de representación es mediante una *Lista de Adyacencia*. Dado un $G = (V, E)$ se construye un arreglo *Adj* de $\mid V \mid$ listas, para cada vértice en V. Por cada $u \in V$, la lista adyacente $Adj[u]$ contiene todos los vértices v tales que existe una arista $(u, v) \in E$. Es decir, que $Adj[u]$ consiste de todos los vértices adjacentes a u en G. Si G es una grafo dirigido, la suma de las longitudes de las listas de adyacencia es $\mid E \mid$. Si G es una grafo no dirigido, la suma de las longitudes de las listas de adyacencia es $2 \mid E \mid$, donde cada arista de la forma (u, v) es una arista no dirigida, entonces u aparece en la lista de adyacencia de v y viceversa.

Sea el grafo dirigido o no, la representación por lista de adyacencia es del orden $O(máx(V, E)) = O(V + E)$. Es una representación apropiada para grafos donde $\mid E \mid$ es menor que $\mid V \mid$. Una desventaja es que puede llevar un tiempo $O(n)$ determinar si existe una arista del vértice i al vértice j, ya que puede haber n vértices en la lista de adyacencia asociada al vértice i.

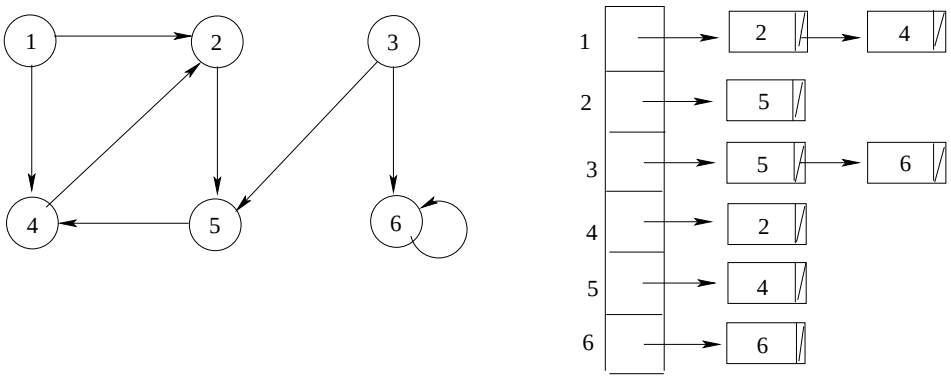


Figura 1.4: Representación mediante Lista de Adyacencia

1.3. Grado de un vértice

Definición 1.9 *El grado de un vértice u es el número de aristas que parten de el y se denota* $gr(u)$.

Todo lazo que parte de un vértice es considerado en el grado como 2 aristas.
Si hablamos de grafos orientados podemos definir también :

1. *Semigrado interior*: Es el número de aristas que llegan al vértice. Para determinadas situaciones, un vértice con semigrado interior cero puede ser un origen del grado.

2. *Semigrado exterior*: Es el número de aristas que parten del vértice. Un vértice con semigrado exterior cero puede representar, en determinadas situaciones, un destino del grafo.

En el gráfico siguiente se observa que el vértice α tiene semigrado interior cero y representa al origen del grafo. El vértices ω tiene semigrado exterior cero y representa el destino del grafo.

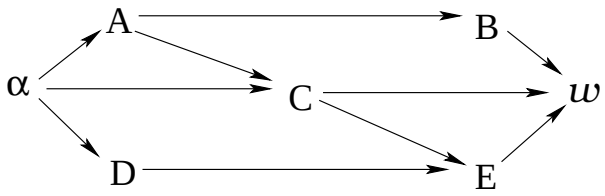


Figura 1.5: Grados de vértices

En una matriz adyacente de un digrafo G podremos obtener el grado exterior de un vértice v_i si sumamos todos los elementos de la fila v_i, del mismo modo si sumamos todos los elementos de la columna v_i obtendremos el grado interior del vértice v_i

Definición 1.10 *Llamaremos mínimo grado y máximo grado de un grafo a los números*

$$\delta(G) = min_{v \in V(G)}\{gr(v)\} \ y \ \triangle(G) = max_{v \in V(G)}\{gr(v)\}$$

Si los 2 números coinciden, por ejemplo en el valor k, entonces todos los vértices del grafo tendran grado k y será un grafo k-regular.

Teorema 1.1 *En un grafo la suma de los grados de los vértices es el doble del número de lados. Es decir, si $G = (V, E)$ es el grafo, entonces*

$$\sum_{v \in V} gr(v) = 2 \mid E \mid$$

Demostración : El resultado es evidente, pues cada arista involucra a dos vértices, veamos la matriz de incidencia:

	e_1	e_2	e_3	...	e_n
v_1	1	1	0	...	0
v_2	1	0	1	...	1
v_3	0	1	0	...	0
...	...	...	...	...	...
v_m	0	0	0	...	1

Las columnas estan etiquetadas con las aristas $\{e_1, ..., e_n\}$ y las filas con los vertices $\{v_1, ..., v_m\}$. En la posición (v_1, e_1) colocaremos un 1 si el vértices v_j es extremo de la arista e_i y un 0 en caso contrario. En la columna etiquetada por la arista e_1, aparecerán solo dos uno(esto porque son sus dos extremos); lo mismo ocurre con el resto de las columnas. Así que, sumando por columnas, obtenemos $2n = 2 \mid E \mid$. Al hacerlo por filas, observamos que la fila correspondiente al vértice v_j contendra tantos unos como vértices adyacentes tenga y la suma valdrá $gr(v)$. Ahora al efectuar la suma de todas las filas se obtiene en si la suma de todas las entradas de la matriz, por lo tanto $\sum_{v \in V} gr(v) = 2 \mid E \mid$

Teorema 1.2 *El número de vértices de grado impar es par*

Demostración:
Sea V_1 y V_2 el conjunto de vértices de grado impar y par en G, respectivamente, entonces:

$$\sum_{v \in V_1} gr(v) + \sum_{v \in V_2} gr(v) = \sum_{v \in V} gr(v)$$

Por el teorema anterior tenemos que es par. Luego como $\sum_{v \in V_2} gr(v)$ es también par, entonces tenemos que $\sum_{v \in V_1} gr(v)$ también es par, por lo tanto $\mid V(V_1) \mid$ es par

1.4. Cadenas y Ciclos - Caminos y Circuitos

Definición 1.11 *Una sucesión de vértices y lados $v_1, e_1, v_2, e_2, ..., e_k, v_{k+1}$ tal que $e_i = [v_i, v_{i+1}]$ se denomina cadena en un grafo y camino en un digrafo.*

Si no existen lados múltiples podemos denotar sin ambiguedad la cadena como una sucesión de vértices (vértice consecutivos adyacentes)

Definición 1.12 *Una cadena es cerrada si el vértice inicial y final es el mismo, es decir $v_0 = v_n$.*

Definición 1.13 *Se llamará cadena simple si no hay vértices repetidos en la sucesión*

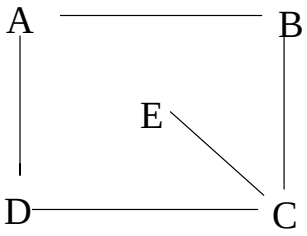


Figura 1.6: Grafo no dirigido

Definición 1.14 *Un ciclo es una cadena cerrada donde todos los vértices (excepto los extremos) son distintos.*

En la figura 1.6 se observa que el vértice A esta conectado a E a través de la cadena A-B-C-E, además se observa el ciclo A-B-C-D

En un multigrafo se considera ciclo a aquellas cadenas cerradas que no repiten aristas.

Definición 1.15 *En un grafo orientado se denominará circuito al camino cuyos vértices origen y destino coinciden.*

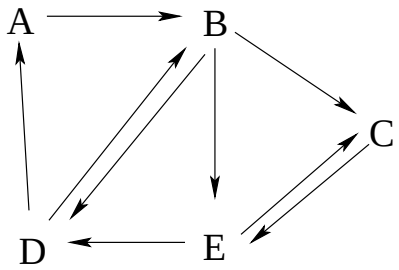


Figura 1.7: Grafo Orientado

Se observa en el grafo orientado que un camino es A, B y C ; además encontramos un circuito A,B,C,E y D

1.5. Grafos Etiquetados y Ponderados

Definición 1.16 *Un grafo G es un grafo etiquetado si sus aristas y vértices tienen asignado alguna identificación. En general G es un grafo ponderado si a cada arista e de G se le asigna un número no negativo $w(e)$ denominado peso o longitud de e*

Definición 1.17 *El peso (o longitud) de un camino en un grafo ponderado G será la suma de los pesos de las aristas del camino.*

Un problema importante es encontrar el camino más corto(liviano), esto es, el camino con el peso(longitud) mínimo entre dos vértices dados.

Definición 1.18 *La longitud de una cadena es el número de lados que hay en el.*

Definición 1.19 *La distancia entre 2 vértices distintos es igual a la longitud de la cadena más corta entre ellos, si no hay camino entre ellos la distancia no esta definida y es cero si los vértices son iguales.*

Definición 1.20 *El diámetro de un grafo es el máximo de las distancias entre cualesquiera par de vértices.*

1.6. Tipos de Grafos

Definición 1.21 *Un grafo $G = (V, E)$ se dice que es libre si $E = \emptyset$, es decir, si no tiene aristas.*

Definición 1.22 *Diremos que un grafo es un L_n, un grafo lineal con n vértices ($n \geq 2$) si tiene n vértices (dos de grado 1 y el resto, si los hay, de grado 2)*

Definición 1.23 *Un grafo simple $G = (V, E)$ se dice que es completo si cada vértice esta conectado a cualquier otro vértice en G. El grafo completo con n vértices se denota K_n*

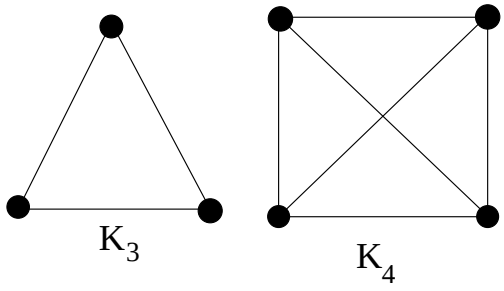


Figura 1.8: Grafo Completo

Definición 1.24 *Un grafo $G = (V, E)$ se dice que es regular de grado k-regular si cada vértice tiene un grado k, es decir, un grafo es regular si todos los vértices tienen el mismo grado.*

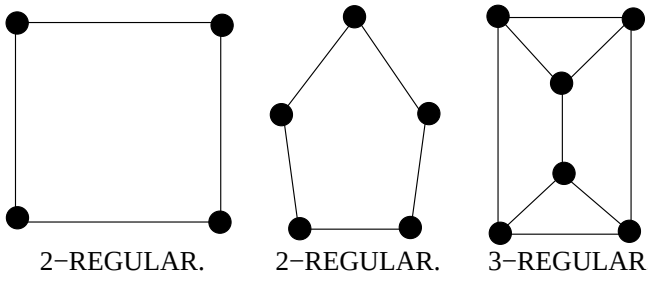


Figura 1.9: Grafos Regulares

1.7. Isomorfismo de Grafos

Definición 1.25 *Los grafos $G_1 = (V_1, E_1)$ y $G_2 = (V_2, E_2)$ son isomorfos si existe una función biyectiva f de V_1 en V_2 con la propiedad de que para cada par de vértices u,v son adyacentes en G_1 si y solo si $f(u)$ y $f(v)$ son adyacentes en G_2. Es decir $\{u, v\} \in E_1 \Leftrightarrow \{f(u), f(v)\} \in E_2$. Si G_1 y G_2 son isomorfos lo denotamos por $G_1 \cong G2$*

Si dos grafos G_1 y G_2 son isomorfos, tienen el mismo número de vértices, el mismo número de aristas, el mismo número de vértices de cualquier grado, el mismo número de ciclos de cualquier longitud, etc.

1.8. Subgrafos

Definición 1.26 *Sea $G = (V, E)$ un grafo, un subgrafo de G es cualquier grafo $H = (W, F)$, de modo que $W \subseteq V$ y $F \subseteq E$.*

Un subgrafo se obtiene eliminando algunas aristas y/o vértices. Si se suprime un vértice, se suprimen todas las aristas que tienen por origen o fin dicho vértice. Si F contiene todos los lados

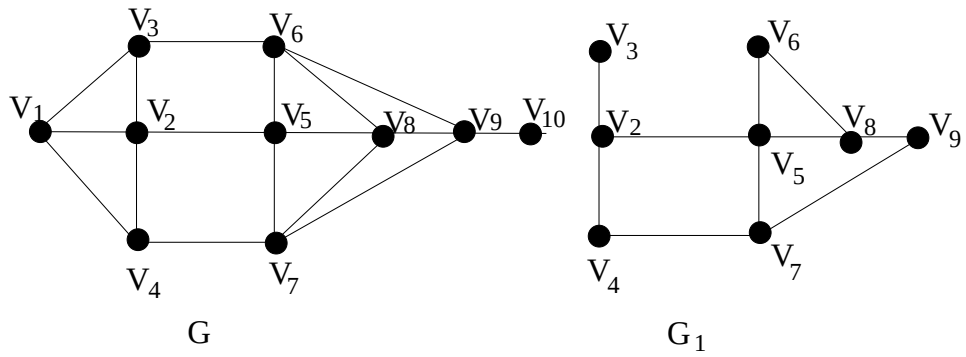


Figura 1.10: El grafo G_1 viene a ser un subgrafo de G

de E que unen a los puntos de W de G se dice que H es un subgrafo completo de G generado por W. Si $W = V$ decimos que H es un subgrafo extendido de G.

1.9. Grafo Bipartitos

Definición 1.27 *Se dice que un grafo simple $G = (V, E)$ es bipartito si el conjunto de vértices V se puede dividir en dos conjuntos V_1 y V_2 ($V_1 \cup V_2 = V$, $V_1 \cap V_2 = \emptyset$) de tal manera que toda arista $e \in E$ conecta a un vértice V_1 con un vértice V_2*

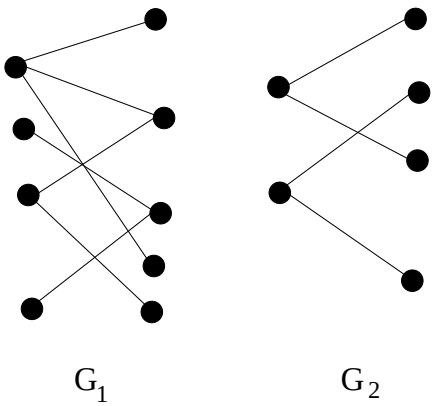


Figura 1.11: Grafos bipartitos

Esto significa que el subgrafo completo generado por V_1 es libre de lados; asimismo de V_1 con un vértice de V_2

Un subgrafo bipartito se dice completo si cada vértice de V_1 esta conectado a todos los vértices de V_2; si $v(V_1) = n$ y $v(V_2) = m$ este grafo se denota $K_{m,n}$

1.10. Conexidad

Definición 1.28 *Un grafo G es conexo si para cada par de vértices distintos v_i, $v_j \in V$ existe un camino que los une, en caso contrario diremos que es disconexo.*

Definición 1.29 *Un grafo orientado es fuertemente conexo si existe al menos un camino entre toda pareja de vértices. Todo grafo orientado fuertemente conexo será también conexo.*

Ahora que sucede cuando un grafo no es conexo, existirán vértices que no pueden ser conectados. Esto nos hará observar que el grafo estará formado por diversos 'bloques' de vértices, cada uno de los cuales será un grafo conexo, llegando a la siguiente definición.

Definición 1.30 *H es una componente conexa de G si H es un subgrafo conexo completo maximal. Es decir no existe un subgrafo completo de G que contenga propiamente a H y sea conexo.*

Sea G un grafo y $v \epsilon V(G)$ un vértice de G, se define $G\backslash\{v\} = G - v$ como el subgrafo de G que se obtiene al borrar el vértice v del grafo G y todos los lados incidentes a v

Sea G un grafo y $e \epsilon E(G)$ un lado de G, se define $G\backslash\{e\} = G - e$ como el subgrafo de G que se obtiene al borrar el lado e del grafo G. Asi $V(G) = V(G - e)$ y $E(G - e) = E(G)\backslash\{e\}$

Definición 1.31 *Una arista e de un grafo G se dice que es puente si $G\backslash\{e\}$ tiene más componentes conexas que G.*

Lema 1.1 *Si G es un grafo conexo y e es una arista puente de G, entonces $G \setminus \{e\}$ tiene exactamente dos componentes conexas.*

Demostración : Llamemos v y w a los vértices que son extremos de la arista e. Y dividamos los vértices de G en dos clases:

1. El conjunto de vértices V_1, formado por aquellos para los que existe un camino que los conecta con v sin usar la arista e (esto es, sin pasar por el vértice w). Entre estos está por supuesto, el propio vértice v

2. El conjunto V_2 de los vértices que necesariamente han de usar la arista e para conectarse a v. Entre ellos esta w ya que es un extremo de la arista.

Se observa en la partición de los vértices de G, $V_1 \cap V_2 = \emptyset$: si $\exists\ x \in V(G)\ /\ x \in V_1 \cap V_2$, entonces e no seria un arista puente, porque podriamos quitarla sin que se desconectara el grafo. Si ahora formamos el grafo $G \setminus \{e\}$, sus dos componentes son conexas y estas son V_1(con sus aristas) y los vértices V_2 (con sus aristas)

Proposicion 1.1 *Si $G = (V, E)$ es una grafo conexo, entonces $\mid E \mid \geq \mid V \mid -1$*

Demostración : La razón es que la conexión óptima (con menor número de aristas) se produce cuando tenemos exactamente una arista menos que vértices. Pero en general tendremos más aristas. Por inducción sobre $\mid E \mid$. Si tenemos que un grafo conexo $\mid E \mid = 1$ la única posibilidad es que el grafo L_2 que tiene 2 vértices.

Supongamos cierto que si tenemos un grafo conexo con k aristas, para cualquier $k \leq m$, entonces $\mid V \mid \leq k+1$. Consideremos entonces un grafo conexo G con $\mid E(G) \mid = m+1$. Sea e una arista cualquiera de G y construyamos un nuevo grafo H quitando esta arista e. El grafo H tiene los mismos vértices que G pero una arista menos (la arista e) que G. Entonces existen 2 posibilidades para este nuevo grafo:

1. Si H sigue siendo conexo(es decir, si a no era arista puente de G), por hipótesis de inducción (tiene m aristas) y teniendo en cuenta que $\mid E(H) \mid = \mid E(G) \mid -1$ y que $V(G) = V(H)$ tendremos que:

$$\mid E(H) \mid \geq \mid V(H) \mid -1 \Longrightarrow \mid E(G) \mid \geq \mid V(G) \mid$$

2. Pero si e era puente en G, H ya no es conexo, sino que tiene dos componentes conexas, por el lema anterior, que serán H_1 y H_2. Ambas son grafos conexos y tienen menos aristas que G (observemos en que estos subgrafos pueden constar de un único vértice). Teniendo en cuenta que:

$$\mid E(H_1) \mid + \mid E(H_2) \mid = \mid E(H) \mid = \mid E(G) \mid -1 \text{ y } \mid V(H_1) \mid + \mid V(H_2) \mid = \mid V(H) \mid$$

por la hipótesis de inducción:

$$\begin{array}{l} \mid E(H_1) \mid \geq \mid V(H_1) \mid -1 \\ \mid E(H_2) \mid \geq \mid V(H_2) \mid -1 \end{array} \Longrightarrow \mid E(G) \mid -1 \geq \mid V(G) \mid -2 \Rightarrow \mid E(G) \mid \geq \mid V(G) \mid -1$$

1.11. Árboles

Definición 1.32 *Un árbol T es un grafo no dirigido, acíclico y conexo.*

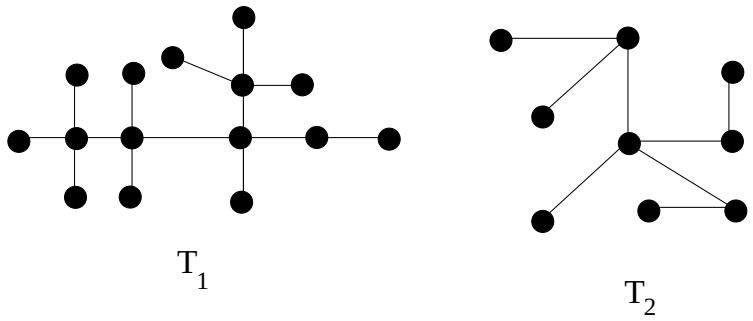


Figura 1.12: T_1 y T_2 árboles

Un árbol puede ser tanto orientado como no orientado, siendo un grafo conexo sin circuitos(grafos orientados) o ciclos (grafos no orientados)

Teorema 1.3 *(Propiedades de los árboles) Sea G=(V,E) un grafo no dirigido. Los siguientes fundamentos son equivalentes:*

1. *G es una árbol*

2. *Cualesquiera dos vértices en G son conectados por un único camino simple*

3. *G es conexo, pero si cualquier arista es removida de E, el grafo resultante es disconexo*

4. *G es conexo, y* $\mid E \mid = \mid V \mid -1$

5. *G es acíclico y* $\mid E \mid = \mid V \mid -1$

6. *G es acíclico, pero si cualquier arista es añadida a E, el grafo resultante contiene un ciclo.*

Demostración

(1)⇒ (2) Como el árbol es conexo, cualquiera de los dos vértices en G son conectados por lo menos con un camino simple.

Sea u y v vertices que son conectados por dos caminos simples distintos p_1 y p_2. Sea ω el vertice el cual los primeros caminos divergen, esto es, ω es el primer vértice sobre p_1 y p_2 cuyo sucesor sobre p_1 es x y cuyo sucesor sobre p_2 es y, donde $x \neq y$. Sea z el primer vértice en el cual los caminos reconvergen, esto es, z es el primer vértice siguiendo ω sobre p_1 que es también sobre p_2. Sea p' el subgrafo de p_1 desde ω a través de x hacia z. Los caminos p' y p'' no comparten vértices a excepción de sus puntos finales. Así, el camino obtenido por la concatenacion p' y el reverso de p'' es un ciclo. Siendo esto una contradicción. Asi, si G es un árbol, pueden existir mas de un camino entre dos vértices

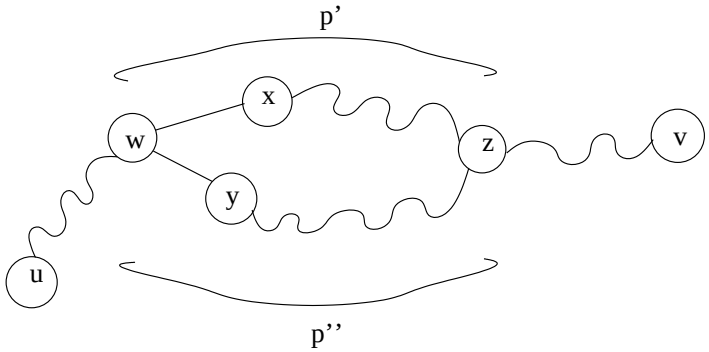


Figura 1.13: *Un paso de la demostración del Teorema 1.3: Si(1) G es un árbol, entonces(2) dos vértices cualquiera en G esta conectados por un único camino simple. Asumamos para el la contradicci'on que el vértice u y v son conectados por dos caminos simples distintos* p_1 *y* p_2*. Estos caminos primero divergen en el vértice w y luego convergen en el vértice z. El camino* p' *concatenado con el reverso del camino* p'' *forma un ciclo, lo cual es una contradicción*

(2)⇒ (3) Si cualquiera de dos vértices en G son conectados por un único camino simple, entonces G es conexo. Sea (u, v) cualquier arista en E. Esta arista es un camino desde u hacia v y también este debe ser el único camino desde u hacia v. Si nosotros removemos (u, v) desde G, no existe camino desde u hacia v, y por lo tanto su eliminación desconecta G.

(3)⇒ (4) Por hipótesis, el grafo G es conexo y por la propiedad 1.1 podemos afirmar que $\mid E \mid \geq \mid V \mid -1$. Debemos probar que $\mid E \mid \leq \mid V \mid -1$ por inducción. Un grafo conexo con $n = 1$ o $n = 2$ vértices tiene $n - 1$ aristas. Supongamos que G tiene $n \geq 3$ vértices y que todos los grafos satisfacen (3) con menos de n vértices también satisfaciendo $\mid E \mid \geq \mid V \mid -1$. Removamos

una arista arbitraria de G separando el grafo en $k \geq 2$ componentes conectados (actualmente k=2). Cada componente satisface (3), o sino G no cumple (3). Asi, por inducción, el número de aristas en toda la combinación de componentes es a lo mucho $\mid V \mid -k \geq \mid V \mid -2$. Añadiendo la arista removida tendremos $\mid E \mid \geq \mid V \mid -1$

(4)⇒ (5) Supongamos que G es conexo y que $\mid E \mid = \mid V \mid -1$. Debemos mostrar que G es acíclico. Supongamos que G tiene un ciclo conteniendo k vértices $v_1, v_2, ..., v_k$. Sea $G_k = (V_k, E_k)$ un subgrafo de G consistiendo de un ciclo. Notar que $\mid V_k \mid = \mid E_k \mid = k$. Si $k < \mid V \mid$, debe existir un vértice $v_{k+1} \in V - V_k$ que es adyacente a algún vértice $v_i \in V_k$ ya que G es conexo. Definamos $G_{k+1} = (V_{k+1}, E_{k+1})$ como un subgrafo G con $V_{k+1} = V_k \cup \{v_{k+1}\}$ y $E_{k+1} = E_k \cup \{(v_i, v_{k+1})\}$. Notar que $\mid V_{k+1} \mid = \mid E_{k+1} \mid = k + 1$. Si $k + 1 < n$ podemos continuar definiendo G_{k+2} de la misma manera, y así sucesivamente, hasta obtener $G_n = (V_n, E_n)$, donde $n = \mid V \mid$, $V_n = V$ y $\mid E_n \mid = \mid V_n \mid = \mid V \mid$. Como G_n es subgrafo de G, tenemos que $E_n \subseteq E$ y por lo tanto $\mid E \mid \geq \mid V \mid$, el cual contradice la hipótesis que $\mid E \mid = \mid V \mid -1$. Asi, G es cíclico.

(5)⇒(6) Supongamos que G es acíclico y que $\mid E \mid = \mid V \mid -1$. Sea k el número de componentes conexos de G. Cada componente conexa es un árbol libre por definición, y como (1) implica (5), la suma de todas las aristas en todas las componentes conexas de G es $\mid V \mid -k$. Consecuentemente, debemos tener $k = 1$ y G es en efecto un árbol. Como (1) implica (2), dos vértices cualquiera en G son conectados por un único camino simple. Así, añadiendo cualquier arista a G creamos un ciclo.

(6)⇒ (1) Supongamos que G es acíclico pero que si una arista cualquiera es añadida a E, un ciclo es creado. Debemos mostrar que G es conexo. Sea u y v vértices arbitrarios en G. Si u y v no son adyacente, añadiendo la arista (u, v) creamos un ciclo en el cual todas las aristas incluido (u, v) pertenecen a G. Esto indica que existe un camino desde u hacia v, y como u y v han sido escogidos arbitrariamente, G es conexo.

1.11.1. Arboles enraizados y ordenados

Un *árbol enraizado* es una árbol en el cual uno de los vértices es distinto de los otros, dicho vértice distinto es llamado *raiz* del árbol.

Consideremos un nodo x en un árbol enraizado T con una raiz r. Cualquier nodo y sobre el único camino desde r hacia x es llamado un *antecesor* de x. Si y es un antecesor de x, entonces x es el *descendiente* de y (todo nodo es ambos antecesor y descendiente de si mismo). Si y es antecesor de x y $x \neq y$, entonces y es un *antecesor propio* de x y x es una *descendiente propio* de y. El *subárbol enraizado* en x es un árbol inducido por descendientes de x, enraizados por x.

Si la última arista sobre el camino desde la raiz r del árbol T a un nodo x es (y, x), entonces y es un *padre* de x y x es el *hijo* de y. La raiz es el único nodo en T que no tiene padre. Un nodo que no tiene hijos se llamará *nodo externo*. Un nodo que tiene hijos será llamado *nodo interno*

El número de hijos de un nodo x en un árbol enraizado T es llamado el *grado* de x. La longitud del camino desde la raiz r hacia un nodo x es la *profundidad* de x en T. La profundidad más larga de cualquier nodo en T es el *peso* de T.

Un *árbol ordenado* es una árbol enraizado en el cual los hijos de cada nodo tienen orden. Esto es, que si un nodo tiene k hijos, entonces existe un primero hijo, un segundo hijo, ... , y un k-esimo hijo.

1.11.2. Árboles binarios

Los árboles binario son la mejor descripción recursiva. Un árbol binario T es una estructura definida sobre un conjunto finito de nodos que o bien:

- No contiene nodos
- Esta compuesto de 3 conjuntos disjuntos de nodos: una nodo raiz, un árbol binario llamado *subárbol izquierdo* y un árbol binario llamado *subárbol derecho*

El árbol binario que no contiene nodos es llamado árbol vacio o árbol nulo, algunas veces denotado como NIL.

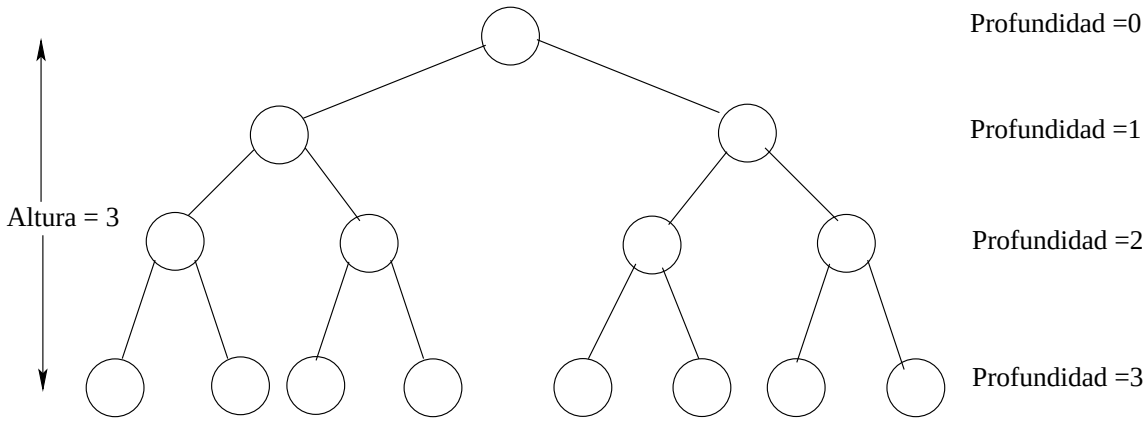


Figura 1.14: *Un árbol completo binario de altura 3 con 8 hojas y 7 nodos internos*

Un árbol completo *k-ario* es cuando todos sus hojas tienen la misma profundidad y todos los nodos internos tiene un grado k

Una raiz tiene k hijos de profundidad 1, cada uno de los cuales tiene k hijos de profundidad 2, etc. Así, el número de hojas de profundidad h es k^h.
En consecuencia:

$$n = k^h \Rightarrow logn = h \cdot logk \Rightarrow h = log_k n$$

la altura de un árbol completo con n hojas es $log_k n$.

El número de nodos internos de un arbol k-ario de altura h es:

$$1 + k + k^2 + \cdots + k^{h-1} = \sum_{i=0}^{h-1} k^i = \frac{k^h - 1}{k - 1}$$

Así, un árbol completo binario (k=2) tendrá $2^h - 1$ nodos internos.

Capítulo 2

Complejidad de Algoritmos

En un sentido amplio, dado un problema y un dispositivo donde resolverlo, es necesario proporcionar un método preciso que lo resuelva, adecuado al dispositivo. Este método será denominado algoritmo.

Para ciertos problemas es posible encontrar más de un algoritmo capaz de resolverlos, lo cual nos enfrenta al problema de escoger uno de ellos. La tarea de decidir cual de ellos es el mejor debe basarse en criterios acordes a nuestros intereses. En la mayoria de los casos la elección de un buen algoritmo esta orientada hacia la disminución del costo que implica la solución del problema; bajo este enfoque es posible dividir los criterios en dos clases:

1. Criterios orientados a minimizar el costo de desarrollo: claridad, sencillez y facilidad de implantación, depuracion y mantenimiento.

2. Criterios orientados a disminuir el costo de ejecución: tiempo de procesador y cantidad de memoria utilizados.

Los recursos que consume un algoritmo pueden estimarse mediante herramientas teóricas y constituyen, por lo tanto, una base confiable para la elección de un algoritmo. La actividad dedicada a determinar la cantidad de recursos que consumen los algoritmos se denomina análisis de algoritmo.

2.1. Conceptos Básicos

Definición 2.1 *Un algoritmo es un conjunto finito de instrucciones no ambiguas y efectivas que indican como resolver un problema, producen al menos una salida, reciben cero o más entradas y para ejecutarse necesitan una cantidad finita de recursos.*

Una instrucción es *no ambigua* cuando la acción a ejecutar esta perfectamente definida, por ejemplo, instrucción del tipo:

$$x \leftarrow log(0)$$

no pueden formar parte de un algoritmo.

Una instrucción es *efectiva* cuando se puede ejecutar en un intervalo finito de tiempo.

Si un conjunto de instrucciones tiene todas las caracteristicas de un algoritmo, excepto ser finito en tiempo se le denomina proceso computacional.

Se asume que un problema tiene solución algorítmica si además de que el algoritmo existe, su tiempo de ejecución es razonablemente corto.

Asumiremos también las siguientes propiedades de los algoritmos:

1. Todo algoritmo tiene un *estado inicial* el cual describe la situación previa a la ejecución del algoritmo
2. Todo algoritmo tiene un *estado final* el cual describe la situación posterior a la ejecución del algoritmo, o dicho de otra manera, describe el objetivo que se quiere lograr con el algoritmo.
3. Todo algoritmo puede hacer uso de *variables*, que son objetos que pueden almacenar información.
4. Todo algoritmo debe tener su ejecución eventualmente, para cualquier estado final.

Usualmente al estado inicial y final de un algoritmo se le denomina *entrada* y *salida* y a todas las posibles entradas del algoritmo las llamaremos *instancias*.

2.2. Eficencia y Complejidad

Una vez dispongamos de un algoritmo que funciona correctamente, es necesario definir criterios para medir su rendimiento y comportamiento. Estos criterios se centran principalmente en su simplicidad y en el uso eficiente de los recursos.

La memoria y el tiempo de procesador son los recursos sobre los cuales se concentra el interés de analizar un algoritmo, así pues distinguiremos dos clases

1. *Complejidad espacial*, mide la cantidad de memoria que necesitará un algoritmo para resolver un problema de tamaño
2. *Complejidad temporal*, indica la cantidad de tiempo que requiere un algoritmo para resolver un problema de tamaño n

La cantidad de memoria que utiliza un algoritmo depende de la implementación, no obstante, es posible obtener una medida del espacio necesario con la sola inspección del algoritmo. En general se requieren dos tipos de celdas de memoria:

1. *Celdas estáticas*: Son las que se utilizan en todo el tiempo que dura la ejecución del programa.
2. C*eldas dinámicas*: Se emplean solo durante un momento de la ejecución y por tanto pueden ser asignadas y devueltas conforme se ejecute el algoritmo.

2.3. Análisis de Algoritmo

El tiempo de ejecución de un algoritmo va a depender de diversos factores como son:

1. Los datos de entrada que le suministremos.

2. La calidad del código generado por el compilador.

3. La naturaleza y rapidez de las instrucciones de la máquina que hace las corridas.

4. La complejidad intrinseca del algoritmo.

Hay dos estudios posibles sobre el tiempo:

1. *Medida Teórica*(a priori): Consiste en obtener una función que acote(superior o inferior)el tiempo de ejecución del algoritmo para unos valores de entrada dados.

2. *Medida Real*(a posteriori): Consiste en medir el tiempo de ejecución del algoritmo para unos valores de entrada dados y en un ordenador concreto.

Ambas medidas son importantes puesto que, si bien la primera nos ofrece estimaciones del comportamiento de los algoritmos de forma independiente del ordenador en donde serán implementados y sin necesidad de ejecutarlos, la segunda representa las medidas reales del comportamiento del algoritmo. Estas medidas son funciones temporales de los datos de entrada.

Entendemos por tamaño de la entrada el número de componentes sobre los que se va ejecutar el algoritmo, como por ejemplo la dimensión del vector a ordenar.

Denotaremos por $T(n)$ el tiempo de ejecución de un algoritmo para una entrada de tamaño n. Teoricamente $T(n)$ debe indicar el número de instrucciones ejecutadas por un ordenor idealizado. Debemos buscar por tanto medidas simples y abstractas, independientes del ordenador a utilizar. Para ello es necesario acotar de algún forma la diferencia que se puede producir entre distintas implementaciones de un mismo algoritmo, ya sea del mismo código ejecutado por dos máquinas de distinta velocidad, como de dos códigos que implementen el mismo método. Esta diferencia es la que acota el siguiente principio:

Definición 2.2 : Principio de Invarianza *Dado un algoritmo y dos implementaciónes* I_1 *e* I_2*, que tardan* $T_1(n)$ *y* $T_2(n)$ *segundos respectivamente, el* Principio de Invarianza *afirma que:*

$$\exists\ c \in \mathbf{R}\ con\ c > 0\ y\ n_0 \in N\ /\ \forall\ n \geq n_0 \rightarrow T_1(n) \leq cT_2(n)$$

Es decir, el tiempo de ejecución de dos implementaciones distintas de un algoritmo dado no va a diferir más que en una constante multiplicativa.

Con esto podemos definir sin problemas que un algoritmo tarda un tiempo del orden de $T(n)$ si existe una constante real $c > 0$ y una implementación I del algoritmo que tarda menos que $cT(n)$, para todo tamaño n de la entrada.

También es importante hacer notar que el comportamiento de un algoritmo puede cambiar notablemente para diferentes entradas (por ejemplo, lo ordenado que se encuentren ya los datos a ordenar). De hecho para muchos programas el tiempo de ejecución es en realidad una función de la entrada específica y no solo del tamaño de esta.

El proceso de ejecución de un algoritmo puede ser dividido en etapas elementales denominados pasos. Cada paso consiste en la ejecución de un número fininto de operaciones básicas cuyo tiempo de ejecución son considerados constantes. Una operación de mayor frecuencia de ejecución en el algoritmo será denominado *Operación Básica*.

Sea $I(n) = \{I_1, I_2, ..., I_k\}$ el conjunto de todas las entradas posibles del problema cuyo tamaño es n. Sea $O(n) = \{O_1, O_2, ..., O_k\}$ el conjunto formado por el número de operaciones que un algoritmo realiza para resolver cada entrada. Entonces, O_j es el número de operaciones ejecutadas para resolver la entrada I_j, para $1 \leq j \leq k$. Se distinguen tres casos para un mismo algoritmo:

1. Complejidad de Peor Caso = $\underset{I_i \in I}{\text{máx}}\{O_i\}$

2. Complejidad de Mejor Caso = $\underset{I_i \in I}{\text{mín}}\{O_i\}$

3. Complejidad de Caso medio = $\sum_{i=1}^{k} O_i P_i$

En otras palabras, el mejor caso se presenta cuando para una entrada de tamaño n, el algoritmo ejecuta el mínimo número posible de operaciones, el peor caso cuando hace el máximo y en el caso medio se consideran todos los casos posibles para calcular el promedio de las operaciones que se hacen tomando en cuenta la probabilidad de que ocurra cada instancia.

Ejemplo: Consideremos el siguiente algoritmo

```
func b\'usquedalineal(Valor,A,n)
  comienza
      i=1
     mientras (i<n) y (A[i]<>Valor) hacer
      i= i+1
     sino
      b\'usquedalineal =i
  terminar
```

Para hacer el análisis de su comportamiento tomemos como operación básica las comparaciones con elementos del arreglo y como caso muestra A=[2,7,4,1,3] y n=5

Si Valor = 2, se hace una comparación
Si Valor = 4, se hacen tres comparaciones
Si Valor = 8, se hacen cinco comparaciones

El análisis temporal sería:

1. Mejor Caso; ocurre cuando el valor es el primer elemento del arreglo

   Mejor Caso = 1

2. Peor Caso; ocurre cuando el valor no se encuentra en el arreglo

   Peor Caso = n+1

3. Caso Medio = $1P(1) + 2P(2) + 3P(3) + ... + nP(n) + nP(n+1)$

   donde P(i) es la probabilidad de que el valor se encuentre en la posición i, $(1 \leq i \leq n)$ y P(n+1) es la probabilidad de que no este en el arreglo. Si se supone que todos los casos son igualmente probables $P(i) = \frac{1}{n+1}$

   $$\text{Caso Medio} = \frac{1}{n+1}\sum_{i=1}^{n+1} i = \frac{(n+1)(n+2)}{2(n+1)} = \frac{n+2}{2}$$

A la hora de medir el tiempo, siempre lo haremos en función del número de operaciones elementales que realiza dicho algoritmo, entendiendo por operaciones elementales (en adelante OE) aquellas que el ordenador realiza en tiempo acotado por una constante. Asi consideraremos OE las operaciones aritmeticas básicas, asignaciones a variables de tipo predefinido por el compilador, los saltos (llamadas funciones y procedimientos, retorno desde ellas, etc), las comparaciones lógicas y el acceso a estructuras indexadas básicas, como son los vectores y matrices. Cada una de ellas contabilizara como 1 OE.

2.3.1. Reglas generales para el cálculo del número de OE

La siguiente lista presenta un conjunto de reglas generales para el cálculo del número de OE, siempre considerando el peor caso. Estas reglas definen el número de OE de cada estructura básica del lenguaje, por lo que el número de OE de un algoritmo puede hacerse por inducción sobre ellas.

Vamos a considerar que el tiempo de una OE es, por definición, de orden 1. La constante c que menciona el *Princicipio de Invarianza* dependerá de la implementación particular, pero nosotros supondremos que vale 1

El tiempo de ejecución de una secuencia consecutiva de instrucciones se calcula sumando los tiempos de ejecución de cada una de las instrucciones.

El tiempo de ejecución de la sentencia 'CASE OF' es $T = T(C) + \text{máx}\{T(S_1), ..., T(S_n)\}$. Observese que $T(C)$ incluye el tiempo de comparación con $v_1, v_2, ..., v_n$

El tiempo de ejecución de la sentencia 'IF THEN ELSE', es $T = T(C) + \text{máx}\{T(S_1), T(S_2)\}$

El tiempo de ejecución de un bucle de sentencia 'WHILE DO' es $T = T(C)+$(número de iteraciones)$*(T(S) + T(C))$. Observerse que tanto $T(C)$ como $T(S)$ pueden variar en cada iteración y por tanto habrá que tenerlo en cuenta para su cálculo.

Para calcular el tiempo de ejecución del resto de sentencia iterativas (FOR, REPEAT, LOOP) basta expresarlas como un bucle WHILE.

El tiempo de ejecución de una llamada a un procedimiento o función $F(P_1, P_2, ..., P_n)$ es 1 (por llamada), más el tiempo de evaluación de los parámetros $P_1, P_2, ..., P_n$, más el tiempo que tarde en ejecutarse F, esto es, $T = 1 + T(P_1) + T(P_2) + ... + T(P_n) + T(F)$. No contabilizamos la copia de los argumentos a la pila de ejecución, salvo que se trate de estructuras complejas(registros o vectores) que se pasan por valor. En este caao contabilizaremos tantas OE como valores simples contenga la estructura. El paso de parámetros por referencia, por tratarse simplemente de punteros, no contabiliza tampoco.

El tiempo de ejecución de las llamadas a procedimientos recursivos va a dar lugar a ecuaciones en recurrencia. También es necesario tener en cuenta, cuando el compilador las incorpore, las optimizaciones del código y la forma de evaluación de las expresiones.

2.4. Cotas de Complejidad: Medidas Asintóticas

2.4.1. Dominio Asintótico

Cuando se implanta un algoritmo para resolver problemas pequeños, el costo adicional por el grado de ineficiencia del algoritmo elegido es poco significativo. Por el contrario cuando el tamaño del problema es grande, la cantidad de recursos que el algoritmo necesite puede crecer tanto que lo haga impráctico. Por esta razón, para elegir entre dos algoritmos es necesario saber como se comportan con problemas grandes. En atención a esta necesidad se estudiará la velocidad de crecimiento de la cantidad de recursos que un algoritmo requiere conforme el tamaño del problema se incremente, es decir, se estudiará el comportamiento asintótico de las funciones complejidad.

Definición 2.3 *Sean f y g funciones reales, definidas sobre* $\mathbf{N}$. *Se dice que f domina asintóticamente a g o que g es dominada asintóticamente por f,*

$$Si\ \exists\ k_0 \in \mathbf{N}\ y\ c \geq 0\ tales\ que \mid g(n) \mid \leq c \mid f(n) \mid, \forall\ n \geq k_0$$

El hecho de que una función domine asintóticamente a otra interpreta que apartir de un valor k_0 se tiene que $c \mid f(n) \mid$ es mayor que $\mid g(n) \mid$ y esto se conserva conforme n crece.

Ahora vamos a intentar clasificar estas funciones de forma que podamos comparalas.

2.4.2. Cota Superior: Notación O

Dada una función f, queremos estudiar aquellas funciones g que a lo sumo crecen tan deprisa como f. Al conjunto de tales funciones se le llama cota superior de f y los denominamos $O(f)$. Conociendo la cota superior de una algoritmo podemos asegurar que, en ningun caso, el tiempo empleado será de un orden superior al de la cota.

Definición 2.4 *Sea $f : \mathbf{N} \to [0, \infty >$. Se define el conjunto de funciones de orden O de f como:*
$O(f) = \{g : \mathbf{N} \to [0, \infty > \ / \ \exists\ c \in \mathbf{R},\ c > 0,\ \exists\ n_0 \in \mathbf{N}\ tal\ que\ g(n) \leq cf(n)\ \forall\ n \geq n_0\ \}$

Diremos que una función $t : \mathbf{N} \to [0, \infty >$ es de orden O de f si $t \in O(f)$

Intuitivamente $t \in O(f)$ indica que t esta acotada superiormente por algun múltiplo de f. Normalmente nos interesará la menor función f tal que $t \in O(f)$

Propiedades de O

Veamos las propiedades de la cota superior

1. Para cualquier función f se tiene que $f \in O(f)$
2. $f \in O(f) \Rightarrow O(f) \subset O(g)$
3. $O(f) = O(g) \Leftrightarrow f \in O(g)$ y $g \in O(f)$
4. Si $f \in O(g)$ y $g \in O(h) \Rightarrow f \in O(h)$
5. Si $f \in O(g)$ y $f \in O(h) \Rightarrow f \in O(min(g, h))$
6. Regla de la suma: Si $f_1 \in O(g)$ y $f_2 \in O(h) \Rightarrow f_1 + f_2 \in O(max(g, h))$

7. Regla del producto: Si $f_1 \in O(g)$ y $f_2 \in O(h) \Rightarrow f_1.f_2 \in O(g.h)$

8. Si existe $\lim_{n\to\infty} \frac{f(n)}{g(n)} = k$, dependiendo de los valores que tome k obtenemos:

   a) Si $k \neq 0$ y $k < \infty$ entonces $O(f) = O(g)$

   b) Si $k = 0$ entonces $f \in f \in O(g)$, es decir, $O(f) \subset O(g)$ pero sin embargo se verifica que $g \notin O(f)$

Observemos que es importante que el límite exista, pues si no existe no podria realizarse tal afirmación.

De las propiedades se deduce que la relación "$\equiv_O$"definida por $f \equiv_O g \Leftrightarrow O(f) = O(g)$ es una relacion de equivalencia.

2.4.3. Cota inferior: Notación Ω

Dada una función f, queremos estudiar aquellas funciones g que a lo sumo crecen tan lentamente como f. Al conjunto de tales funciones se le llama cota inferior de f y lo denominamos $\Omega(f)$. Conociendo la cota inferior de un algoritmo podemos asegurar que, en ningún caso, el tiempo empleado será de un orden inferior al de la cota.

Definición 2.5 *Sea* $f : \mathbf{N} \to [0, \infty >$. *Se define el conjunto de funciones de orden* Ω *de* f *como:*
$\Omega(f) = \{g : \mathbf{N} \to [0, \infty > \ / \ \exists\ c \in \mathbf{R},\ c > 0,\ \exists\ n_0 \in \mathbf{N}$ *tal que* $g(n) \geq cf(n)\ \forall n \geq n_0\ \}$

Diremos que una función $t : \mathbf{N} \to [0, \infty >$ es de orden Ω de f si $t \in \Omega(f)$

Intuitivamente, $t \in \Omega(f)$ indica que t esta acotada inferiormente por algún múltiplo de f. Normalmente estaremos interesados en la mayor función f tal que $t \in \Omega(f)$ a la que denominaremos su cota inferior.

Propiedades de Ω

Veamos las propiedades de la cota inferior

1. Para cualquier función f se tiene que $f \in \Omega(f)$
2. $f \in \Omega(f) \Rightarrow \Omega(f) \subset \Omega(g)$
3. $\Omega(f) = \Omega(g) \Leftrightarrow f \in \Omega(g)$ y $g \in \Omega(f)$
4. Si $f \in \Omega(g)$ y $g \in \Omega(h) \Rightarrow f \in \Omega(h)$
5. Si $f \in \Omega(g)$ y $f \in \Omega(h) \Rightarrow f \in \Omega(min(g, h))$
6. Regla de la suma: Si $f_1 \in \Omega(g)$ y $f_2 \in \Omega(h) \Rightarrow f_1 + f_2 \in \Omega(max(g, h))$
7. Regla del producto: Si $f_1 \in \Omega(g)$ y $f_2 \in \Omega(h) \Rightarrow f_1.f_2 \in \Omega(g.h)$
8. Si existe $\lim_{n\to\infty} \frac{f(n)}{g(n)} = k$, dependiendo de los valores que tome k obtenemos:

   a) Si $k \neq 0$ y $k < \infty$ entonces $\Omega(f) = \Omega(g)$

b) Si $k = 0$ entonces $f \in f \in \Omega(g)$, es decir, $\Omega(f) \subset \Omega(g)$ pero sin embargo se verifica que $g \notin \Omega(f)$

De las propiedades se deduce que la relación "$\equiv_\Omega$"definida por $f \equiv_\Omega g \Leftrightarrow \Omega(f) = \Omega(g)$ es una relación de equivalencia.

2.4.4. Orden Exacto: Notación Θ

Como la última cota asintótica, definiremos los conjuntos de funciones que crecen asintóticamente de la misma forma.

Definición 2.6 *Sea* $f : \mathbf{N} \to [0, \infty >$. *Se define el conjunto de funciones de orden* Θ *de f como:*

$$\Theta(f) = O(f) \cap \Omega(f)$$

o lo que es igual:

$$\Theta(f) = \{g : \mathbf{N} \to [0, \infty > \ / \ \exists\ c, d \in \mathbf{R},\ c, d > 0,\ \exists\ n_0 \in \mathbf{N} \text{ tal que } cf(n) \leq g(n) \leq df(n) \quad \forall\ n \geq n_0\}$$

Intuitivamente, $t \in \Theta(f)$ indica que t esta acotada tanto superior como inferiormente por múltiplos de f, es decir, que t y f crecen de la misma forma.

Propiedades de Θ

Veamos las propiedades de la cota inferior

1. Para cualquier función f se tiene que $f \in \Theta(f)$
2. $f \in \Theta(f) \Rightarrow \Theta(f) \subset \Theta(g)$
3. $\Theta(f) = \Theta(g) \Leftrightarrow f \in \Theta(g)$ y $g \in \Theta(f)$
4. Si $f \in \Theta(g)$ y $g \in \Theta(h) \Rightarrow f \in \Theta(h)$
5. Regla de la suma: Si $f_1 \in \Theta(g)$ y $f_2 \in \Theta(h) \Rightarrow f_1 + f_2 \in \Theta(max(g, h))$
6. Regla del producto: Si $f_1 \in \Theta(g)$ y $f_2 \in \Theta(h) \Rightarrow f_1.f_2 \in \Theta(g.h)$
7. Si existe $\lim_{n\to\infty} \dfrac{f(n)}{g(n)} = k$, dependiendo de los valores que tome k obtenemos:
   a) Si $k \neq 0$ y $k < \infty$ entonces $\Theta(f) = \Theta(g)$
   b) Si $k = 0$ los ordenes exactos de f y g son distintos

Gran parte de los algoritmos tienen complejidad que cae en uno de los siguientes casos:

- $O(1)$: Complejidad constante
- $O(logn)$: Complejidad logarítmica
- $O(n)$: Complejidad lineal
- $O(nlogn)$: Complejidad "n log n"

- $O(n^2)$: Complejida cuadrática
- $O(c^n)$, $c > 1$: Complejidad exponencial
- $O(n!)$: Complejidad factorial

Dentro de estas existe algunas relaciones entre los órdenes de complejidad dadas de la siguiente forma:

$O(1) \subset O(logn) \subset O(n) \subset O(nlogn) \subset O(n(logn)^2) \subset O(n^{1,011...}) \subset O(n^2) \subset O(n^3) \subset ... \subset O(2^n) \subset O(n!) \subset O(n^n)$

Algunas Observaciones sobre las cotas asintoticas

El orden de un polinomio $a_n x^n + ... + a_i x + a_0$ es $O(x^n)$

$$\sum_{i=1}^{n} 1 \in O(n); \sum_{i=1}^{n} i \in O(n^2); \sum_{i=1}^{n} i^m \in O(n^{m+1})$$

Si hacemos una operación para n, otra para n/2, n/4,..., aparecera un orden logarítmico $O(log_2 n)$

2.5. Lenguaje Formal

Para llegar a este concepto es necesario definir antes otras nociones preliminares.

2.5.1. Alfabeto, cadena de caracteres

La noción mas primitiva es la de símbolo, que es simplemente una representacion distinguible de cualquier información. Los símbolos pueden ser cualesquiera, como w, 9 ,", etc. Un símbolo es una entidad indivisible.

Un alfabeto es un conjunto no vacio de símbolos. En general se utiliza la notación Σ para representar un alfabeto

Con los símbolos de un alfabeto es posible formar secuencias o cadenas de caracteres, tales como mxzxzor, etfes, etc. Las cadenas de caracteres son llamadas palabras.

Un caso particular de cadena es la palabra vacia ε, la cual no tiene ninguna letra. La longitud de una palabra es la cantidad de letras que contiene, contando las repeticiones, se denota por $\mid w \mid$ para una palabra w.

Cuando escribimos varias palabras o caracteres uno a continuación de otro, se supone que forman una sola palabra (se concatenan). La concatenación de palabras es asociativa, esto es, $(xy)z = x(zy)$ pero no conmutativa en el caso general. La longitud de una concatenación cumple la propiedad : $\mid uv \mid = \mid u \mid + \mid v \mid$

Una palabra v es subcadena de otra w cuando existen cadenas x, y; posiblemente vacias; tales que $xvy = w$

El conjunto de todas las palabras que se pueden formar con un alfabeto $\sum$ es denotado convencionalmente por $\sum^*$.

Por ejemplo, si $\sum = \{a, b\}$, $\sum^* = \{\varepsilon, a, aa, aaa, aaaa, ..., b, bb, ..., ab, aba, abb, ...\}$. El conjunto Σ^* es infinito, pero enumerable.

2.5.2. Operaciones con lenguajes

Se pueden utilizar varias operaciones para producir nuevos lenguajes a partir de otros dados. Supongase que L_1 y L_2 son lenguajes sobre un alfabeto común. Entonces:

1. La concatenación L_1L_2 consiste de todas aquellas palabras de la forma vw donde v es una palabra de L_1 y w es una palabra de L_2
2. La intersección L_1 & L_2 consiste en todas aquellas palabras que estan contenidas tanto en L_1 como es L_2
3. La unión $L_1 \mid L_2$ consiste en todas aquellas palabras que estan contenidas ya sea en L_1 o en L_2
4. El complemento L_1 consiste en todas aquellas palabras producibles sobre el alfabeto de L_1 que no estan ya contenidas en L_1
5. El cociente L_1/L_1 consiste de todas aquellas palabras v para las cuales existe un palabra w en L_2 tales que vw se encuentran en L_1
6. L_1^* consiste de todas aquellas palabrs que pueden ser escritas de la forma $W_1W_2W_3, ..., W_n$ de todo W_i se encuentra en L_1 y $n \geq 0$
7. La intercalación $L_1^*L_2$ consiste de todas aquellas palabras que pueden ser escritas de la forma $v_1w_1v_2w_2, ..., vnw_n$ son palabras tales que la concatenación $v_1, ..., v_n$ está en L_1 y la concatenación $w_1, ..., w_n$ está en L_2

En conclusión

Definición 2.7 *Un lenguaje formal es un conjunto de palabras (cadenas de caracteres) de longitud finita en los casos más simples o expresiones válidas (formuladas por palabras) formadas a partir de un alfabeto (conjunto de caracteres) finito*

2.6. Problemas Matemáticos

Los problemas matemáticos se pueden dividir en primera instancia en dos grupos:

- *Problemas Indecidibles:* Aquellos que no se pueden resolver mediante un algoritmo
- *Problemas Decidibles:* Aquellos que cuentan al menos con un algoritmo para su computo.

Sin embargo, que un problema sea decidible no implica que se pueda encontrar su solución, pues muchos problemas que disponen de algoritmos para su resolución son inabordables para un computador por el elevado número de operaciones que hay que realizar para resolverlos. Nosotros analizaremos el caso de los problemas Decidibles.

2.6.1. Problemas de Decisión

En Computación, un problema es un conjunto de frases de longitud finita que tienen asociadas frases resultantes también de longitud finita

Un problema de decisión es una problema en donde las respuestas posibles son SI o NO. Un ejemplo típico de problema de decisión es la pregunta: ¿Es un número entero dado primo? y una entrada o instancia del problema sería: ¿ Es 17 primo?

Un problema de decisión también se puede formalizar como el problema de decidir si una cierta frase pertenece a un conjunto dado de frases también llamado lenguaje formal. El conjunto contiene exactamente las frases para las cuales la respuesta a la pregunta es positiva.

Si existe un algoritmo que pueda decidir para cada posible frase de entrada si esa frase pertenece al lenguaje, entonces se dice que el problema es tratable, de otra forma se dice que es un problema intratable. Cuando existe un algoritmo que puede responder positivamente cuando la frase esta en el lenguaje, pero que corre indefinitivamente cuando la frase no pertenece al lenguaje se dice que el problema es parcialmente tratable.

2.6.2. Problemas Tratables e Intratables

Los problemas computacionales los podemos dividir en:

- **Tratables:** Problemas para los cuales existe un algoritmo de complejidad polinomial. A esto problemas se les conoce también como problemas P
- **Intratables:** Problemas para los cuales no se conoce ningún algoritmo de complejidad polinomial. A estos problemas se les conoce también como problemas NP

2.6.3. Algoritmos Determinísticos y No Determinísticos

Definición 2.8 *Se llamará* Algoritmo Determinístico *a aquel que siga una misma secuencia de estados completamente predeterminados, conociendose las entradas y produciendo la misma salida.*

Esto nos indica que el resultado es único en el sentido de que cada instrucción tiene un resultado conocido dependiendo del contexto.

Para abordar el siguiente concepto primero definamos una función *ELEGIR* de la siguiente manera:

Elige(S). Es una función que regresa uno de los elementos de S. Por ejemplo S={Azul, Rojo, Amarillo} y P=ELIGE(S) entonces P=Rojo o P=Azul o P=Amarillo.

Exito: Notifica una ejecución exitosa
Fracaso: Notifica una ejecución no exitosa
Esta función ejecuta la elección a discresión suya.

Ejemplo: Búsqueda de un elemento x en un arreglo A.

```
Func BUSCA(A,x,indice);
  Comienza
     i=ELIGE(1,...,n)
       si A[i]=x entonces
                  indice = i
                  Exito;
             Otro
                  Fracaso
      Fin de si
  Termina
```

Este algoritmo nos estaria mostrando una búsqueda no deterministica.
Podemos interpretar la función *ELIGE* de las siguientes maneras:

1. Que *ELIGE* siempre devuelve el valor que necesitamos de una manera mágica.
2. Al llamar a *ELIGE*, el programa se multiplica, después del llamado hay tantas copias como valores pueden devolver el llamado. La ejecución de la instrucción Exito detiene todas las copias del programa. La ejecución de la instruccion Fracaso detiene la copia del programa que la ejecuta.

Las interpretaciones son equivalentes pero no es aconsejable usarlos para el diseño de algoritmos

Ejemplo: Ordenamiento de una arreglo A con n elementos

```
Proc ORDENA(A,n)
   Comienza
       Para i=1 a n hacer
          B[i] = ELIGE (A)
          A = A - B[i];
       fin de hacer
     Exito;
   Termina.
```

El algoritmo fue diseñado pensando en la primera interpretación de que la elección es justo la que nos dará el orden apropiado. Pero si utilizamos la segunda interpretación el algoritmo siempre regresará éxito lo cual es erróneo.

```
Proc ORDENA(A,n)
   Comienza
        Para i=1 a n hacer
           B[i]= ELIGE (A);
           A = A - B[i];
        Fin de hacer
       i = 1;
       mientras i<n o B[i] < = B[i+1] hacer
         i = i +1
          Si i=n entonces
              Exito;
           otro
            Fracaso;
   Termina.
```

El algoritmo hecho es independiente de la interpretación de *ELIGE* y regresa el resultado correcto utilizando multiprocesos.
Consideremos un ejemplo más antes de definir '*Algoritmo no Deteministico*'

Ejemplo: Considere el problema de factorización entera: Dado $n \in N$ compuesto, determinar $d \in N$, $1 < d < n$ y un divisor de n.

```
Proc INTENTA(n)
 Inicio
   d:=0
   r:=1
     Mientras (r<>0) hacer
          Elegir entre
            opc 1: d:=10*d+1
            opc 2: d:=10*d+2
            opc 3: d:=10*d+3
            opc 4: d:=10*d+4
            opc 5: d:=10*d+5
            opc 6: d:=10*d+6
            opc 7: d:=10*d+7
            opc 8: d:=10*d+8
            opc 9: d:=10*d+9
            opc 10: si d<>0 entonces
                      d:=10*d
                    Fin de si
          Fin de elegir\
        Si d>=n entonces
          d:=0
        Fin de si
        Si d>=2 entonces
         r:=n mod d
        Fin de si
     Fin de mientras
 Fin
```

Las elecciones posibles para el algoritmo anterior se ilustran de la siguiente forma, cada elección que se da es un paso y a la vez una posición de la cantidad de cifras que tiene el número n.

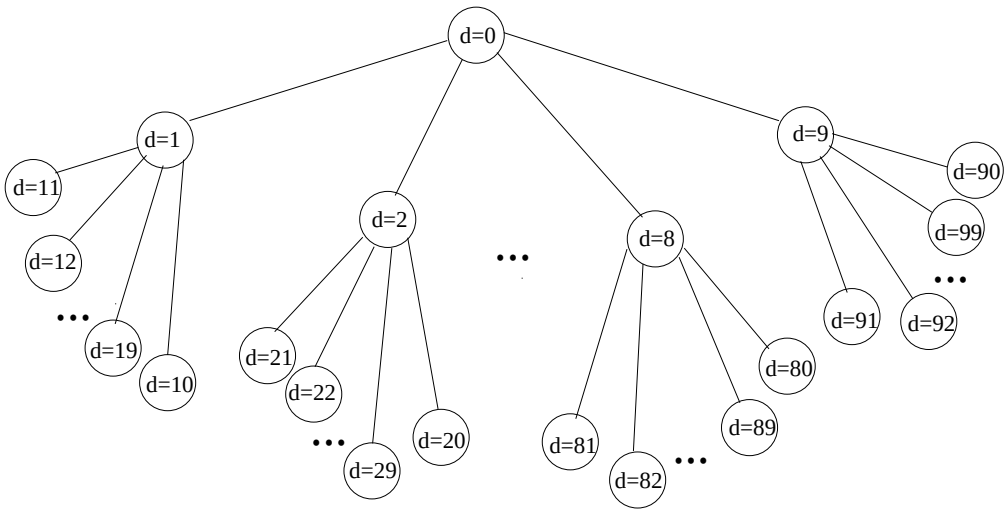


Figura 2.1: *Algoritmo INTENTA*

Observe que si n es compuesto, existen un conjunto de elecciones que el ejecutor puede tomar para que el algoritmo anterior con entrada n se detenga, y en este caso, el algoritmo devolverá precisamente un divisor no trivial de n.

Ahora consideramos la cantidad de pasos ejecutados por el algoritmo anterior, con alguna entrada n. Asumimos $n = \overline{a_s a_{s-1} ... a_1}_{10}$ de s cifras en su representación en base 10. Entonces un divisor no trivial de n tiene a lo más $s-1$ cifras. Luego el algoritmo anterior realizará por lo menos $O(s) = O(log(n))$ pasos. El problemas surge cuando se quiere intentar calcular la cantidad máxima de pasos ejecutados. Observe que dependiendo de las elecciones a ejecutar, el algoritmo anterior puede demorarse un tiempo arbitrario en terminar. Esto motiva la siguiente definición de algoritmo no determinístico.

Definición 2.9 *Un conjunto finito de instrucciones, entre las cuales pueda estar la estructura elige, se dice Algoritmo No Determinístico si el ejecutor para cada ELEGIR realiza elecciones de tal manera que minimizen la cantidad de instrucciones ejecutadas por el algoritmo*

Podemos decir además que es un algoritmo que con la misma entrada ofrece muchos posibles resultados. No se puede saber de antemano cual será el resultado de la ejecución de un algoritmo no determinístico.

De esto podemos concluir que todo algoritmo determinístico es no determinístico, además el algoritmo *INTENTA* se puede considerar como no determinístico y en cierto modo dado un n, el ejecutor hallar la solución de un modo que en realidad desconocemos. Por lo tanto no cualquiera será capaz de ejecutar un algoritmo no determinístico. Ahora bien para ejecutar la estructura *ELEGIR* en un algoritmo no determínistico podemos hacerlo de la siguiente forma:

Aleatorización de un algoritmo no deteminístico: En este caso solo consideraremos las instrucciones dadas en un algoritmo no determinístico como un algoritmo aleatorio, de tal manera que elegir sea realizado con una elección al azar. Ahora para aparentar esa sucesión de números elegida al azar será necesario un algoritmo al que llamaremos, generador de números aleatorios.

La desventaja de este método es que pese a que en el mejor de los casos el algoritmo puede terminar rapidamente, en el peor de los casos tomaria un tiempo arbitrario en terminar.

Conversión a determínistico: En este caso se busca eliminar el uso de la estructura ELEGIR. Para esto, modificamos el algoritmo para que en cierto orden este realize todas las posibles elecciones dadas en *ELEGIR*.

Ejemplo: Como mencionamos anteriormente al ejecutar el algoritmo INTENTA, el ejecutor puede construir en la variable d, cualquier número natural. Luego, podemos modificar el algoritmo para obtener lo siguiente:

```
Algoritmo Factoriza(n)
   Inicio
       d:=0
       r:=0
     Mientras (r<>0) hacer
       d:=d+1
       r:= n mod d
     Fin de Mientras
   Fin
```

Observe que para n compuesto este algoritmo se detendrá eventualmente. Además que es el algoritmo de factorización posee un complejidad $O(n) = O(e^{ln(n)})$

En general, convertir un algoritmo no-determinístico en uno determinístico implica usualmente un incremento de la complejidad del algoritmo. En el ejemplo anterior, este aumentó de $O(log(n))$ a $O(n) = O(e^{ln(n)})$, es decir aumento de complejidad polinomial (respecto al tamaño de n) a complejidad exponencial.

2.6.4. Reductibilidad o Transformación Polinómica

Una transformación polinomial es una forma de reducir un problema de decisión en otro de forma que cualquier algoritmo que resuelva el primer problema produzca inmediatamente una solución al segundo, por un coste polinómico.

Esto es: Sean L y M lenguajes formales sobre los alfabetos Σ, Γ, respectivamente. Una transformación polinómica de L en M, es una función $f : \Sigma^* \to \Gamma^*$ que puede ser calculada en tiempo polinomial tal que $w \in L \Leftrightarrow f(w) \in M \; \forall \; w \in \Sigma^*$ nosotros lo denotaremos por: $L \leq_p M$

Cuando esta función f existe, tambien se puede decir que L es polinomicamente transformanle en M

2.7. Clases de Complejidad

Luego de varios análisis de problemas se ha llegado a constatar de que existen problemas muy difíciles, problemas que desafian la utilización de los ordenadores para resolverlos. En lo que sigue esbozaremos las clases de problemas que hoy por hoy se escapan a un tratamiento informático

2.7.1. Complejidad de Clase P

Contiene aquellos problemas de decisión que pueden ser resueltos en tiempo polinómico por un algoritmo determinístico. Los problemas de complejidad polinómica son tratables ya que se pueden resolver en la práctica en tu tiempo razonable. Aquellos problemas para los que la mejor solución que se conoce es de complejidad superior a la polinómica, se dice que son problemas intratables.

Propiedad

Los algoritmos de tiempo polinomial son cerrados bajo composición. Intuitivamente, decimos que si uno escribe una función la cual es de tiempo polinomial, asumimos que la funciones de llamada son de tiempo constante, y si estas funciones llamadas así mismo requiere un tiempo polinomial, entonces el algoritmo entero toma tiempo polinomial.

Algoritmos de Verificación

Consideremos el problema de la suma de subconjuntos: Dado un conjunto de numeros enteros $\{-7, -3, -2, 5, 8\}$, deseamos saber si existe un subconjunto de esos enteros tal que su suma sea cero, la respuesta es que si existe $\{-3, -2, 5\}$.

Cuando el número de enteros que tenga que evaluar el algoritmo sea muy grande, el tiempo que necesitará para calcular la respuesta crecerá exponencialmente y esto hace que el problema se convierta en un NP-completo. Sin embargo, notamos que si nosotros damos un subconjunto en particular(o también certificado), nosotros podemos facilmente verificar que la suma de dicho subconjunto da cero. Así ese subconjunto en particular es una prueba que existe una respuesta afirmativa. Un algoritmo que verifica la existencia de alguna respuesta afirmativa se llamará algoritmo de verificación los cuales son de tiempo polinomial.

2.7.2. Complejidad de Clase NP

NP(non-deterministic Polinomial Time) es una de las clases más fundamentales, contiene los problemas de decisión que son resueltos por un algoritmo no determinístico en tiempo polinómico. Algunos de estos problemas intratables pueden caracterizarse por el curioso hecho de que puede aplicarse un algoritmo de verificación de tiempo polinómico para comprobar si una posible solución es valida o no. Esta característica lleva a un método de resolución no determinística consistente en aplicar heurísticos para obtener soluciones hipotéticas que se van desestimando a ritmo polinómico.

La clase P se encuentra contenida en NP, pero NP contiene muchos problemas importantes, llamados también NP-Completos, para el cual no se conoce algoritmos de tiempo polinomial.

Muchos problemas naturales de la informática son cubiertos por la clase NP, en particular, los problemas de decisión y los problemas de optimización estan contenido en NP

Los problemas que presentan una respuesta negativa al momento de aplicar un algoritmo de verificación serán llamados $co - NP$(Complemento de NP), aún quedado sin resolver si todos los problemas que tienen respuesta negativa también tienen un certificado.

2.7.3. Complejidad de Clase NP-COMPLETOS

Se conoce una amplia variedad de problemas de tipo NP, de los cuales destacan algunos de ellos de extrema complejidad. Son problemas NP y son los peores problemas posibles de clase NP y es probable que no formen parte de la clase de complejidad P. NP-completo es el conjunto de los problemas de decisión en NP tal que todo problema en NP se puede reducir en cada uno de los problemas de NP-completo.

El problema de suma de subconjuntos puede ser un buen ejemplo ya que la verificación de alguna respuesta es fácil, pero no se conoce mejor solución que explorar todos los $2^n - 1$ subconjuntos posibles hasta encontrar uno que cumpla con la condición.

Definición 2.10 *Un problema de decisión C es NP-completo si :*

1. $C \in NP$
2. *Todo problema de NP se puede transformar polinomialmente en C.*

Un problema que satisface la segunda condición pero no la primera se le denomina NP-duro

Entre los problemas más conocidos de NP-completos podemos mencionar el problema de isomorfismo de grafos. Dos grafos son isomorfos si se puede transformar uno en el otro simplemente renombrando los vértices. Aún se sospecha que el problema no esta ni en P ni en

NP-completo aunque si en NP. Se trata de un problema difícil, pero no tanto como para estar en NP-Completo.

Para probar que un nuevo problema es NP-completo es primero demostrar que está en NP y luego transformar polinomicamente un problema que ya este en NP-completo a este, para ello es útil conocer los problemas que han sido demostrado son NP-completo:

- Problema de satisfacibilidad booleana(SAT)
- Problema de la mochila
- Problema del ciclo hamiltoniano
- Problema del vendedor viajero
- Problema de la clique
- N-Rompecabeza
- Problema de suma de subconjunto

Por mencionar algunos de ellos. Actualmente, todos los algoritmos conocidos para problemas NP-completos utilizan tiempo exponencial con respecto al tamaño de la entrada. Se desconoce si hay mejores algoritmos, por lo cual, para resolver un problema NP-completo de tamaño arbitrario, se utiliza: Aproximación, Probabilidades, Heurística, Parametrización y Casos Particulares.

2.7.4. Complejidad de Clase NP-Duro

La Clase NP-Duro es el conjunto de los problemas de decisión que contiene los problemas M tales que todo problema L en NP puede ser transformado polinomicamente en M. Esta clase puede ser descrita como los problemas de decisión que son al menos tan difíciles como un problema de NP.

Asumiendo que M es NP-completo es decir:

1. L esta en NP
2. $\forall$ L' en NP, $L' \leq L$

En el conjunto NP-Duro se asume que L satisface la propiedad 2, pero no la propiedad 1

Capítulo 3

Estructura de Datos

3.1. Diseñando algoritmos

Existen muchas maneras de diseñar algoritmos. Nosotros en particular analizaremos la alternativa de desarrollo, conocida como *'Divide y Conquista'*. Usaremos divide y conquista para diseñar un algoritmo de clasificación cuyo peor caso de tiempo de ejecución es mucho menor que la de inserción.

3.1.1. Desarrollando 'Divide y Conquista'

Muchos de los algoritmos que usamos son de estructura recursiva: Para resolver un problema dado ellos se llaman así mismo recursivamente uno o más veces para hacer frente estrechamente a un subproblema relacionado. Este algoritmo típicamente llamado *Divide y Conquista* se enfoca en dividir un problema en muchos subproblemas que son similares al original pero de un tamaño más pequeño, resuelve estos subproblemas recursivamente y luego combina esa soluciones para crear la solución al problema general.

El desarrollo de *Divide y Conquista* envuelve 3 pasos para cada nivel de recursión:

- *Dividir*; el problema en un número de subproblemas
- *Conquista*; los subproblemas resolviendolos recursivamente. Esto si el tamaño del subproblema es lo suficientemente pequeño, sin embargo, solo resuelve el subproblema de una manera sencilla
- *Combinar*; las soluciones de los subproblemas dentro de la solución para el problema original

3.1.2. Analizando los algoritmos Divide y Conquista

Cuando un algoritmo contiene un llamado recursivo de si mismo, su tiempo de ejecución puede ser a menudo descrito por una ecuación de recurrencia, la cual decribe el tiempo total de ejecución en un problema de tamaño n en terminos de tiempo de ejecución para entradas más pequeñas.

Una recurrencia para un tiempo de ejecución de divide y conquista esta basado en los 3 pasos anteriormente mencionados. Sea $T(n)$ el tiempo de ejecución de un problema de tamaño n. Si el tamaño del problema es lo suficientemente pequeño, diremos que $n \leq c$ para cualquier

constante c, la sencilla solución tomará un tiempo constante, el cual será $\Theta(1)$. Supongamos que dividimos el problema en a subproblemas, el cual cada uno es $1/b$ del tamaño original. Si nosotros tomamos $D(n)$ como el tiempo para dividir el problema en subproblemas y $C(n)$ como el tiempo de combinar los subproblemas en la solución del problema original, obtendremos:

$$T(n) = \begin{cases} \Theta(1) & \text{si}, n \leq c \\ aT(n/b) + D(n) + C(n) & \text{en otros casos} \end{cases}$$

3.2. Recurrencias

Cuando algunos algoritmos contienen un llamado recursivo de si mismo, su tiempo de ejecución puede a menudo describirse por una recurrencia. Una recurrencia es una ecuación o desigualdad que describe una función en términos de sus valores de entradas pequeñas. En esta sección revisaremos rapidamente 3 métodos para resolver recurrencias. Por ejemplo, describimos la siguiente recurrencia

$$T(n) = \begin{cases} \Theta(1) & \text{si}, n = 1 \\ 2T(n/2) + \Theta(n) & \text{si}, n > 1 \end{cases}$$

cuya solución es $T(n) = \Theta(n.lg(n))$

3.2.1. Método de sustitución

El método de sustitución para resolver recurrencias implica .ªdivinar"la forma de la solución y entonces usar la inducción matemática para encontrar la constante que mostrará que la solución funciona. El nombre viene de la sustitución de la posible respuesta por la función cuando la hipótesis inductiva es aplicada para valores pequeños. Este método es muy fuerte pero obviamente solo puede ser aplicado en casos cuando es fácil .ªdivinar"la forma de la respuesta.

El método de sustitución puede ser usado para establecer cualquier cota super o inferior sobre una recurrencia. Por ejemplo, determinemos una cota superior sobre la siguiente recurrencia

$$T(n) = 2T(\lfloor n/2 \rfloor) + n$$

Suponemos que la posible solución es $T(n) = O(nlg(n))$. El método buscará probar que $T(n) \leq c{\cdot}nlg(n)$ para cualquier elección apropiada de la constante $c > 0$. Empezamos asumiendo que esta acotación se mantiene para $\lfloor n/2 \rfloor$, esto es, $T(\lfloor n/2 \rfloor) \leq c\lfloor n/2 \rfloor lg(\lfloor n/2 \rfloor))$. Sustituyendo en la recurrencia tenemos:

$$\begin{array}{rcl} T(n) & \leq & 2(c\lfloor n/2 \rfloor lg(\lfloor n/2 \rfloor)) + n \\ & \leq & c.n.lg(n/2) + n \\ & = & c.n.lg(n) - c.n.lg2 + n \\ & = & c.n.lg(n) - c.n + n \\ & \leq & c.n.lg(n) \end{array}$$

donde el último paso se mantiene siempre y cuando $c \geq 1$

La inducción matemática ahora requiere mostrar que nuestra solución se mantiene para cualquier condición de acotación. Esto es, debe mostrarse que puede escogerse una constante c, lo suficientemente grande tal que la cota $T(n) \leq n.lg(n)$ trabaja en las condiciones de frontera. Asumimos, por el motivo del argumento, que $T(1) = 1$ es la única condición de límite de la recurrencia. Entonces, desafortunadamente, no podemos escoger un c suficientemtene grande,

ya que $T(1) \leq 1.c.lg1 = 0$. Esta dificultad de demostrar una hipótesis de inducción para una determina condición de frontera puede ser fácilmente superado. Tomaremos ventaja del hecho que la notacion asintótica solo requiere probar que $T(n) \leq c.n.lg(n)$ para $n \geq n_0$, donde n_0 es una constante, la idea es eliminar la difícil condición de frontera $T(1) = 1$ desde la consideración en la prueba inductiva e incluir $n = 2$ y $n = 3$ como parte de las condiciones de frontera para la prueba inductiva porque para $n > 3$, la recurrencia no depende directamente en $T(1)$. Desde la recurrencia, llegamos $T(2) = 4$ y $T(3) = 5$. La prueba inductiva que $T(n) \leq c.n.lg(n)$ para alguna constante $c \geq 2$ puede ahora ser completada eligiendo un c suficientemente grande, tal que $T(2) \leq 2.c.lg2$ y $T(3) \leq 3.c.lg3$. Como resultado, cualquier elección $c \geq 2$ es suficiente.

3.2.2. Método de la iteracción

Este método no requiere de una buena elección para llegar a la respuesta, pero requiere de mucho algebra. La idea es expandir (iterar) la recurrencia y expresar esta como una sumatoria de términos que solo dependan de n y las condiciones iniciales. Veamos el siguiente ejemplo:

$$T(n) = 3T(\lfloor n/4 \rfloor) + n$$

Iterando obtenemos:

$$\begin{aligned} T(n) &= n + 3T(\lfloor n/4 \rfloor) \\ &= n + 3(\lfloor n/4 \rfloor + 3T(\lfloor n/16 \rfloor)) \\ &= n + 3(\lfloor n/4 \rfloor + 3(\lfloor n/16 \rfloor + 3T(\lfloor n/64 \rfloor))) \\ &= n + 3\lfloor n/4 \rfloor + 9\lfloor n/16 \rfloor + 27(\lfloor n/64 \rfloor) \end{aligned}$$

Donde $\lfloor \lfloor n/4 \rfloor /4 \rfloor = \lfloor n/16 \rfloor$ y $\lfloor \lfloor n/16 \rfloor /4 \rfloor = \lfloor n/64 \rfloor$

Si observamos la recurrencia, el termino i-esimo es $3^i \lfloor n/4^i \rfloor$. En la iteración $n = 1$ cuando $\lfloor n/4 \rfloor = 1$ o equivalentemente , cuando i excede $log_4 n$. De continuar la iteracción hasta el punto y usando la acotación $\lfloor n/4^i \rfloor \leq n/4^i$, descubrimos que la sumatoria contiene una serie geométrica decreciente :

$$\begin{aligned} T(n) &\leq n + 3n/4 + 9n/16 + 27n/64 + ... + 3^{log_4 n}\Theta(1) \\ &\leq \sum_{i=0}^{\infty}(3/4)^i + \Theta(n^{log_4 3}) \qquad \text{(por identidad } 3^{log_4^n} = n^{log_4 3}) \end{aligned}$$

Como $log_4 3 < 1$ podemos decir entonces que $0 \leq n^{log_4 3} < n$

$= 3\text{n} + \Theta(n^{log_4 3})$
$= O(n)$

Recursión en árboles

Una recursión en un árbol es un conveniente camino para visualizar que sucede cuando la recurrencia es iterada, y este puede ayudar a organizar el libro de mantenimiento algebraico necesario para resolver la recurrencia. Este es especialmente usado cuando la recurrencia describe un algoritmo de divide y conquista. Por ejemplo analizando la derivación de la recursion del árbol por:

$$T(n) = 2T(n/2) + n^2$$

Evaluando la recurrencia mediante la adicion de los valores a través de cada nivel del árbol. El nivel superior tiene un valor total de n^2, el segundo nivel tiene un valor $(n/2)^2+(n/2)^2 = (n)^2/2$, el tercer nivel tiene un valor de $(n/4)^2 + (n/4)^2 + (n/4)^2 + (n/4)^2 = n^2/4$ y asi sucesivamente. Luego los valores decrecen geometricamente, el total es a lo mucho un factor constante más que el término más grande(primero) y por lo tanto la solución es $\Theta(n^2)$.

Como otro ejemplo más intricado, en la figura mostraremos la recursión del árbol por :

$$T(n) = T(n/3) + T(2n/3) + n$$

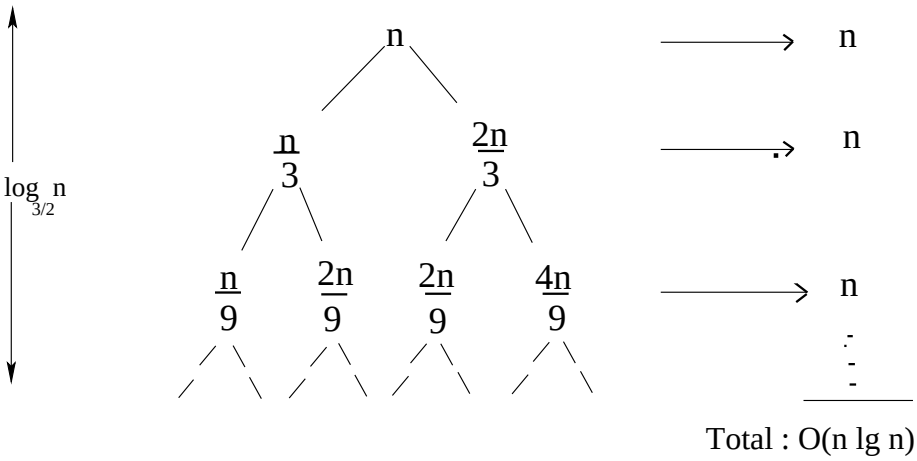


Figura 3.1: *La recursión de este árbol esta dada por la recurrencia* $T(n) = T(n/3)+T(2n/3)+n$

Cuando añadimos los valores a través de los niveles en la recursión del árbol, conseguimos un valor de n para cada nivel. El camino más largo desde la raiz hasta una hoja es $n \to (2/3)n \to (2/3)^2 n \to ... \to 1$. Como $(2/3)^k n = 1$ cuando $k = log_{3/2} n$, la altura del arbol es $log_{3/2} n$. Así, la solución de la recurrencia es a lo mucho $n.log_{3/2} n = O(n.lg(n))$

3.2.3. El Método Maestro

El método maestro permite resolver recurrencias de la forma:

$$T(n) = aT(n/b) + f(n)$$

donde $a \geq 1$ y $b > 1$ son constantes y $f(n)$ es una función asintóticamente positiva. El método maestro requiere la memorización de 3 casos, pero entonces la solución de muchas recurrencias pueden ser determinadas facilmente rápido.

La recurrencia anterior describe el tiempo de ejecución de un algoritmo que divide un problema de tamaño n en a subproblemas, cada uno de tamaño $1/b$, donde a y b son constantes positivas. Los subproblemas a son resueltos recursivamente, cada uno en un tiempo $T(n/b)$. El costo de dividir el problema y combinar los resultados de los subproblemas es descrita por la funcion $f(n)$.

Como cuestión de corrección técnica, la recurrencia no es actualmente bien definida porque n/b no podria ser entero. Reemplazando cada uno de los a terminos $T(n/b)$ con cualquier $T(\lfloor n/b \rfloor)$ o $T(\lceil n/b \rceil)$ no afecta el comportamiento asintótico de la recurrencia. Normalmente encontramos este incoveniente, por lo tanto, omitiremos las funciones máximo o mínimo entero al escribir la recurrencia de este forma.

Teorema 3.1 *(Teorema Maestro) Sea $a \geq 1$ y $b > 1$ constantes, sea $f(n)$ una función y sea $T(n)$ definido sobre enteros no negativos por la recurrencia*

$$T(n) = aT(n/b) + f(n)$$

donde interpretamos n/b como algún $\lfloor n/b \rfloor$ o $\lceil n/b \rceil$. Entonces $T(n)$ puede ser acotado asintoticamente como sigue:

1. *Si $f(n) = O(n^{log_b a - \epsilon})$ para alguna constante $\epsilon > 0$, entonces $T(n) = \Theta(n^{log_b a})$*
2. *Si $f(n) = \Theta(n^{log_b a})$ entonces $T(n) = \Theta(n^{log_b a} lgn)$*
3. *Si $f(n) = \Omega(n^{log_b a + \epsilon})$ para alguna constante $\epsilon > 0$ y si $af(n/b) \leq cf(n)$ para alguna constante $c < 1$ y todo n suficiemente grande, entonces $T(n) = \Theta(f(n))$*

Veamos una ejemplo de como usar el teorema. Consideremos:

$$T(n) = 9T(n/3) + n$$

Por la recurrencia, tenemos $a = 9$ y $b = 3$ y $f(n) = n$ y además $n^{log_b a} = n^{log_3 9} = \Theta(n^2)$. Como, $f(n) = O(n^{log_3 9 - \epsilon})$, donde $\epsilon = 1$, aplicamos el caso 1 del teorema maestro y concluimos que la solucion $T(n) = \Theta(n^2)$.

3.3. Ordenamiento por montículos

Este tipo de ordenamiento introduce una técnica que usa una estructura de datos, al cual llamaremos *heap(montículo)* para manejar información durante la ejecución del algoritmo. No solo es el heap una estructura usada para el ordenamiento, sino también se usa en la eficiencia de la cola de prioridades.

3.3.1. Heap

La estructura de datos del Heap(binario) es un arreglo que puede ser visto como un árbol binario completo. Cada nodo de este árbol corresponde a un elemento del arreglo que almacena el valor del nodo. El árbol es completamente llenado en todos los niveles excepto posiblemente en los más bajos, el cual es llenado de izquierda a derecha. Un arreglo A que representa un heap es un objeto con dos atributos : Longitud[A], el cual es el número de elementos del arreglo y el heap-size[A], que es el número de elementos en el heap almacenados dentro del arreglo A. La raiz del árbol es $A[1]$ y dado el indice i de un nodo, los indices de su padre (i), hijo izquierdo(i) e hijo derecho(i) serán calculados simplemente:

Padre (i) $\rightarrow$ retornará $\lfloor i/2 \rfloor$

Izquierda (i) $\rightarrow$ retornará $2i$

Derecha (i) $\rightarrow$ retornará $2i + 1$

Los Heaps también satisfacen la propiedad que para todo nodo i distinto de la raiz:

$$A[Padre(i)] \geq A[i]$$

Esto es que el valor del nodo es a lo mucho el valor de su padre. Así el mayor elemento del heap es almacenado en la raiz y el subárbol enraizado a un nodo contiene valores más pequeño que el propio nodo.

Definimos la altura de un nodo en un árbol como el número de aristas a lo largo de un camino de descenso simple, desde el nodo hasta una hoja y luego definimos la altura del árbol como la altura de su raiz. Como un heap de n elementos es basado en un árbol binario completo, su altura es $\Theta(lg(n))$.

Debemos ver que las operaciones básicas sobre un heap se ejecutan en un tiempo proporcional a la altura del árbol y es $O(lg(n))$

3.3.2. Manteniendo las propiedades del Heap

El procedimiento *HEAPIFY* es una subrutina para manejar los heaps. Sus entradas son un arreglo A y el índice i del arreglo. Cuando HEAPIFY es llamado, esto es asumiendo que los árboles binarios enraizados a la izquierda(LEFT(i)) y derecha (RIGHT(i)) son heaps, pero que $A[i]$ quizas es más pequeño que su hijo, se tomará en cuenta la contradicción de la propiedad del heap $A[Padre(i)] \geq A[i]$

```
HEAPIFY(A,i)
  1. l = Left(i)
  2. r = Right(i)
  3.   Si l<= heap-size[A] y A[l]>A[i]
  4.      entonces mayor =l
  5.         sino mayor =i
  6.   Si r <= heap-size[A] y A[r]> A[mayor]
  7.      entonces mayor = r
  8.   Si mayor <> i
  8.      entonces cambiar A[i] por A[mayor]
  9.        HEAPIFY(A,mayor)
```

En cada paso, el mayor elemento A[i], A[Left(i)] y A[Right(i)] es determinado y su índice es almanecenado en *mayor*. Si A[i] es el mayor, entonces el subárbol enraizado al nodo i es un heap y el procedimiento termina. De otra modo, uno de los dos hijos tiene un elemento mayor y A[i] es cambiado con A[mayor], el cual produce que el nodo i y sus hijos satisfagan la propiedad del heap. El nodo mayor sin embargo, ahora tiene un valor original A[i] y así el subárbol enraizado al mayor quizás viole la propiedad del heap. Consecuentemente, HEAPIFY debe ser llamado recursivamente sobre este subárbol.

El tiempo de ejecución del HEAPIFY sobre un árbol de tamaño n enraizado al nodo dado i es del tiempo $\Theta(1)$ al fijar las relaciones entre los elementos A[i], A[Left(i)] y A[Right(i)], más el tiempo de ejecución de HEAPIFY sobre un subárbol enraizado a uno de los hijos del nodo i. Dependiendo de la altura del árbol se tendrá que repetir el algoritmo de manera recursiva convirtiendo la altura del árbol en el factor para analizar su complejidad. Los subárboles de sus hijos cada uno tiene un tamaño a lo mucho de $2n/3$, el peor de los casos ocurre cuando la última fila del árbol esta exactamente media llena, y el tiempo de ejecución de HEAPIFY puede ser descrito por la recurrencia.

$$T(n) \leq T(2n/3) + \Theta(1)$$

La solución de recurrencia, por el caso 2 del teorema maestro es $T(n) = O(lg(n))$. Alternativamente podemos también caracterizar el tiempo de ejecución de HEAPIFY sobre un nodo de altura h como $O(h)$

3.3.3. Construyendo un Heap

El procedimiento CONSTRUIR-HEAP se va realizando en el resto de nodos del árbol y ejecuta HEAPIFY sobre cada uno. El orden en el cual los nodos son procesados garantizando que los subárboles enraizado al hijo de un nodo i son heaps antes que HEAPIFY es ejecutado a ese nodo.

```
CONSTRUIR-HEAP(A)
1.   Heap-size[A]= longitud[A]
2.   Para i= max-entero(longitud[A]/2) hasta 1
3.      Hacer HEAPIFY(A,i)
```

Nosotros podemos calcular una cota superior simple al tiempo de ejecución de CONSTRUIR-HEAP. Cada llamado de HEAPIFY cuesta un tiempo de $O(lg(n))$, y existe $O(n)$ llamadas. Así, el tiempo de ejecución será a lo mucho $O(n.lg(n))$. Esta cota superior, aunque correcta, no es asintóticamente estricta.

Podemos derivar una acotación estricta al observa que el tiempo para ejecutar HEAPIFY a un nodo varia con la altura del nodo en el árbol y las alturas de la mayoria de nodos son pequeños. Nuestro análisis nos lleva a que en un n-elemento del heap existe a lo mucho $\lceil n/2^{h+1} \rceil$ nodos de altura h.

El tiempo que requiere un HEAPIFY cuando es llamado sobre un nodo de altura h es $O(h)$, entonces podemos expresar el tiempo total de CONSTRUIR-HEAP como:

$$\sum_{h=0}^{\lfloor lg(n) \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor lgn \rfloor} \frac{h}{2^n} \right)$$

La última sumatoria puede ser evaluada sustituyendo x=1/2 , obtendremos:

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2}$$

$$= 2$$

Así, el tiempo de ejecución de CONSTRUIR-HEAP puede ser acotado por:

$$O\left(\sum_{h=0}^{\lfloor lgn \rfloor} \frac{h}{2^h} \right) = O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right)$$

$$= O(n)$$

Por lo tanto, podemos contruir un heap desde un arreglo desordenado en tiempo lineal.

3.3.4. El algoritmo de Heapsort(ordenamiento por montículos)

El algoritmo de heapsort empieza usando CONSTRUIR-HEAP para construir el heap de la entrada de un arreglo $A[1..n]$, donde $n = tamaño[A]$. Como el máximo elemento del arreglo esta almacenado en la raiz $A[1]$, este puede ser puesto dentro de su correcta posición al cambiarlo con $A[n]$. Si ahora descartamos el nodo n del heap (decreciendo el heap-size[A]), observamos que $A[1..(n-1)]$ facilmente se convierte en un heap. Los hijos de una raiz permanecen en heaps, pero el nuevo elemento raiz viola la propiedad del heap, por eso para restaurar dicha propiedad se ejecuta el llamado HEAPIFY(A,1). El algoritmo heapsort repite el proceso para el heap desde n-1 hasta 2.

```
HEAPSORT
1. CONSTRUIR-HEAP(A)
2. Para i=longitud[A] hasta 2
3.   Hacer cambiar A[1] por A[i]
4.      heap-size [A]= heap-size A[i]-1
5.      HEAPIFY(A,1)
```

El Heapsort toma un tiempo de ejecución $O(n.lg(n))$, ya que el llamado CONSTRUIR-HEAP toma un tiempo $O(n)$ y cada uno de los n-1 llamados a HEAPIFY toma $O(lg(n))$

3.3.5. Cola de prioridades

Una cola de prioridades es una estructura de datos que almacena un conjunto S de elementos, cada uno con un valor asociado llamado *clave*. Una cola de prioridades soporta las siguientes operaciones.

- **Insertar**(S, x); inserta el elemento x en el conjunto S. Esta operación podria escribirse como $S \leftarrow S \cup \{x\}$
- **Máximo**(S); retorna el elemento de S con la mayor clave.
- **Extraer-Max**(S); remueve y retorna el elemento de S con la mayor clave.

Podemos también utilizar el heap para implementar una cola de prioridades. La operación HEAP-MAXIMO retorna el máximo elemento del heap en un tiempo $\Theta(1)$ por simplemente retornar un valor $A[1]$ en el heap. El procedimiento HEAP-EXTRAER-MAX es similiar para el ciclo PARA (lineas 3-5) del procedimiento Heapsort.

```
HEAP-EXTRAER-MAX(A)
1.   Si heap-size[A]<1
2.     entonces error‘heap underflow’
3.   max = A[1]
4.   A[1]= A[heap-size[A]]
5.   heap-size[A]= heap-size[A]-1
6.   HEAPIFY(A,1)
7.   retornar max
```

El tiempo de ejecución de HEAP-EXTRAER-MAX es $O(lg(n))$, ya que solo realiza una cantidad constante de operaciones antes del tiempo $O(lg(n))$ del HEAPIFY.

El procedimiento HEAP-INSERTAR inserta un nodo en el heap A. Produciendo así esta primera expansión del heap al añadir una nueva hoja en el árbol. Entonces, este atraviesa un camino desde esta hoja hacia la raiz para encontrar el lugar apropiado para este nuevo elemento.

```
HEAP-INSERTAR(A,clave)
1.   heap-size[A]= heap-size[A]+1
2.   i = heap-size[A]
3.   Mientras i>1 y A[Padre(i)]< clave
4.      Hacer A[i] = A[Padre(i)]< clave
5.          i = Padre(i)
6.   A[i]= clave
```

El tiempo de ejecución del HEAP-INSERTAR para un n-elemento del heap es $O(lg(n))$, ya que el camino trazado desde una nueva hoja hasta la raiz tiene un longitud $O(lg(n))$.

En conclusión, un heap puede soportar cualquier operación de cola de prioridades sobre un conjunto de tamaño n en el tiempo $O(lg(n))$

Se pueden tomar en cuenta además diferentes tipos de heaps que sirven para implementarse en distintos algoritmos.

Implementación d-Heap:

Dado un parámetro $d \geq 2$, la estructura de un d-Heap, requiere un tiempo del orden $O(log_d n)$ para llevar a cabo las operaciones de insertar o cambiar-clave, requiere un tiempo del orden $O(d.log_d n)$ para eliminar-min y requiere de un tiempo del orden $O(1)$ para otras operaciones.

Implementación de Heap de Fibonacci:

Esta estructura permite todas las operaciones en un tiempo de amortizacion del orden $O(1)$ excepto eliminar-min, el cual requiere un tiempo del orden $O(log(n))$

3.4. Algoritmos Elementales en Grafos

Los grafos son una estructura de datos en ciencias de la computación y los algoritmos son las herramientas fundamentales para trabajar en este campo con ellos. Se sabe además, que el tiempo de ejecución de un algoritmo depende de el número de entradas, en este caso el número de vértices $\mid V \mid$ y el número de aristas $\mid E \mid$. Se analizó en secciones anteriores que un grafo puede ser almacenado mediante una representación, una matriz de adyacencia, la cual nos indica la organización que tiene el grafo dado. En esta sección analizaremos los algoritmos de búsqueda en un grafo.

3.4.1. Búsqueda en Anchura

Es uno de los algoritmos de búsqueda sencillos y que sirve como base para algoritmos importantes, como es el caso del algoritmo de Dijkstra de caminos mínimos, y el algoritmo de Prim de árbol de expansión mínima.

Dado un grafo $G = (V, E)$ y una fuente de vértices distintos a s, la búsqueda en anchura explora las aristas de G hasta descubrir todos los vértices que sean accesibles desde s. Este calcula

la distancia (el menor número de aristas) desde s a todos los vértices accesibles. Produciendo esto un *árbol de anchura* con raiz s que contiene todos los vértices accesibles. Para un vértice v accesible desde s, el camino en el árbol en anchura desde s a v corresponde al *camino más corto* desde s hacia v en G, es decir, un camino que contenga el menor número de aristas. El algoritmo se puede aplicar en grafos dirigidos como no dirigidos.

Para entender el proceso que efectua el algoritmo de búsqueda en anchura, tomaremos en cuenta que cada vértice ira tomando distintos colores: blanco, gris o negro, dependiendo del estado en que se encuentre. Todos los vértices empezarán de color blanco, cada vértice que haya sido descubierto pasará a ser negro o gris. Al ingresar el vértice inicio y encontrarlo en el grafo G, este se tornará de color gris y será almacenado en la cola, al tomarlo se volverá de color negro y todos aquellos vértices que sean adyacentes a él se irán a almacenando en la cola y estarán de color gris, además de ir calculando la distancia para cada vértice alcanzado y sale de la cola el vértice que fue tomado. Siendo todos los vértices de color gris la frontera entre los encontrados y no encontrados. Luego se vuelve a repetir el proceso con los vértices que fueron ingresados en la cola repitiendo el proceso al buscar sus respectivos vértices adyacentes a cada uno de ellos.

La búsqueda en anchura construye un árbol en anchura ya que cada vértices u que es descubierto en la lista de adyacencia que es accesible desde s forma una arista (s, u) del árbol. Sea la arista (u, v), diremos que u es predecesor de v en el árbol en anchura.

Veamos el siguiente algoritmo *BFS* donde asumiremos el grafo $G = (V, E)$ es representado usando una lista de adyacencia (Adj[u]). El color de cada vértice $u \in V$ ser almacenará en color[u] y el predecesor de u se almacenará en la variable $\pi[u]$. Si u no tiene predecesor (por ejemplo si $u = s$ o u no ha sido descubierto), entonces $\pi[u]$=NIL. La distancia de la fuente s al vértices u calculada por el algoritmo es almacenada en $d[u]$. El algoritmo utiliza una cola que posea un inicio y un final para poder manejar los vértices grises que se almacenan.

```
Algoritmo BFS (G,s)
1    Para cada vrtice u en V[G]- {s}
2        Hacer color[u]= blanco
3          d[u]= infinito
4          pi[u]=NIL
5    color [s]= gris
6    d[u]= 0
7    pi[s]= NIL
8    Q recibe {s}
9    Mientras Q no este vacio
10       Hacer u = head[Q]
11          Para cada v en Adj[u]
12             Hacer  si color [v]=blanco
13                      entonces color [v]=Gris
14                               d[v]= d[v]+1
15                               pi[v]= u
16                               Encola (Q,v)
17       Desencola(Q)
18       color[u]= negro
19   Fin
```

El gráfico 3.2 nos muestra el proceso del algoritmo BFS.

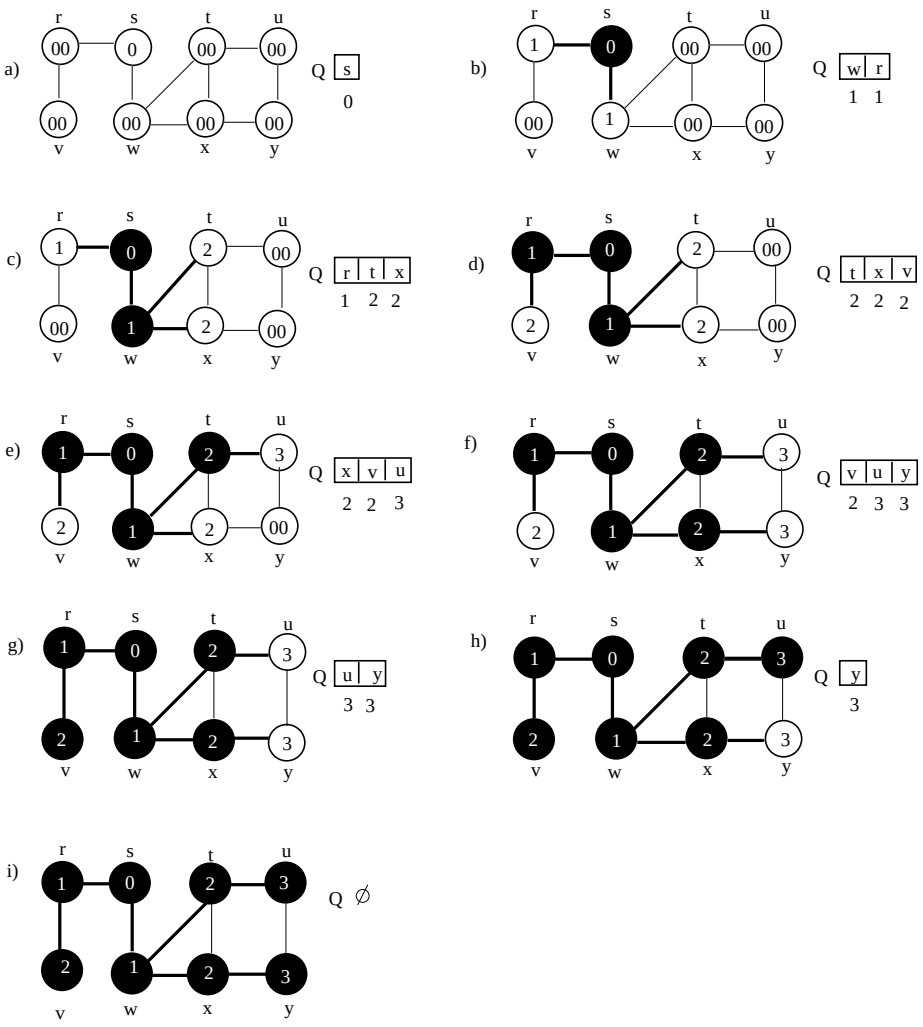


Figura 3.2: *Las operaciones de BFS estan sobre un grafo no dirigido. Las aristas de los árboles se muestran más oscuras cuando estas han realizado un paso de BFS. Dentro de cada vértice u se muestra d[u]. La cola Q es mostrada en cada iteracción del ciclo while de las lineas 9-18. Las distancias de vértice son mostrados debajo de cada vértice en la cola*

El Proceso de BFS es el siguiente: En las lineas 1-4 todos los vértices son blanqueados y el valor $d[u]$ es infinito para todos los vértices u, y el conjunto de los padres de todos los vértices son NIL. En la linea 5 da color gris al vértice s, al ser descubierto. La linea 6 inicializa $d[s]$ como 0 y la linea 7 da su predecesor como NIL. En la linea 8 inicializa Q para continuar la cola después del vértice s. Q siempre contendrá los vértices de gris.

El ciclo en las lineas 9-18 iteractua a lo largo de todos los vértices grises, los cuales son vértices descubiertos al examinar por completo sus vértices adyacentes. En la linea 10 determina el vértice gris u y lo coloca al inicio de la cola Q. El ciclo FOR de las lineas 11-16 consideran cada vértice v en la lista de adyacencia de u. Si v es blanco, entonces aun no ha sido descubierto y el algoritmo lo descubre ejecutando las lineas 13-16. Así se vuelve gris y su distancia $d[v]$ es $d[u] + 1$. Entonces, u es descubierto como su padre. Finalmente, se coloca al final de la cola Q. Cuando todos los vértices adyacentes de u has sido examinados, u es removido de la cola Q y

se vuelve negro en las lineas 17-18.

Análisis de Complejidad

Analizaremos ahora el tiempo de ejecución del algoritmo sobre un grafo $G = (V, E)$. Después de la inicialización, no hay vértices que es aún blanqueado, además en la linea 12 se asegura que cada vértice es puesto en cola a lo mucho una vez y luego sacado de la cola a lo mucho una vez. Las operaciones de encolar y desencolar toman un tiempo de $O(1)$ veces, el tiempo total para las operaciones de encolar sera $O(V)$, porque la lista de adyacencia de cada vértice es recorrida solo cuando el vértice es desencolado, la lista de adyacencia de cada vértice es recorrida a lo mucho una vez. Luego la suma de las longitudes de toda la lista de adyacencia es $\Theta(E)$, a lo mucho el tiempo $O(E)$ se gasta en el recorrido total de la lista de adyacencia. Al inicio para la inicialización es $O(V)$, entonces el tiempo de ejecución de BFS es $O(V + E)$

3.4.2. Caminos mínimos

Hemos observado que en la búsqueda en anchura encontramos la distancia de cualquier vértices accesible en un grafo $G = (V, E)$ desde un vértice inicial $s \in V$. Definimos la *distancia de camino mínimo* $\delta(s, v)$ de s a v como el mínimo número de aristas en cualquier camino desde el vértice s al vértice v, o es infinito si no existe camino de s a v. Antes de mostrar que la búsqueda en anchura actualmente calcula la distancias de caminos mínimos, analizaremos algunas propiedades.

Lema 3.1 *Sea $G = (V, E)$ un grafo dirigido o no dirigido , y dado $s \in V$ un vértice arbitrario. Entonces, para cualquier arista $(u, v) \in E$,*

$$\delta(s, v) \leq \delta(s, u) + 1$$

Demostración: Si u es accesible desde s, entonces también lo es v. En este caso, el camino corto de s hacia v no puede ser más largo que el camino corto de s hacia u siguiendo por la arista (u, v), y además la desigualdad se mantiene. Si u no es accesible desde s, entonces $\delta(s, u) = \infty$ y la desigualdad se mantiene.

Nosotros queremos mostrar que BFS, propiamente calcula $d[v] = \delta(s, v)$ para cada vértice $v \in V$. Nosotros mostraremos que $d[v]$ acota $\delta(s, v)$ superiormente

Lema 3.2 *Sea $G = (V, E)$ un grafo directo o indirecto, y supongamos que BFS es ejecutado sobre G dando un vértice inicial $s \in V$. Entonces se debe cumplir, para cada vértice $v \in V$, el valor de d[v] calculado por BFS, satisface $d[v] \geq \delta(s, v)$*

Demostración: Nosotros usaremos la inducción sobre el número de veces que un vértice se situa en la cola. Nuestra hipótesis de inducción es que $d[v] \geq \delta(s, v)$ para todo $v \in V$. La base de la induccción se situa inmediatamente luego que s ocupa el lugar en la cola Q. La hipótesis de inducción se mantiene alli, porque $d[s] = 0 = \delta(s, s)$ y $d[v] = \infty$ para todo $v \in V - \{s\}$.

Por inducción, consideremos el vértice blanco v que es descubierto durante la búsqueda desde el vertice u. La hipótesis de inducción implica que $d[u] \geq \delta(s, u)$. Ahora por lo asignado en el algoritmo BFS tenemos que:

$$d[v] = d[u] + 1 \geq \delta(s, u) + 1 \geq \delta(s, v)$$

El vértice v es entonces insertado en la cola y no vuelve a ser insertado porque ahora es de color gris como se indica en el algoritmo. El encolamiento es solo cuando son vértices blancos, además el valor de $d[v]$ nunca cambia nuevamente y la hipótesis de inducción se mantiene.

Para probar que $d[u] = \delta(s, v)$, debemos ver como el encolamiento Q opera durante el proceso de BFS. El proximo lema muestra que en todo momento, existe a lo mucho dos distintos valores de d en la cola.

Lema 3.3 *Supongamos que durante la ejecución de BFS sobre un grafo G=(V,E), la cola Q contiene los vértices* $\langle v_1, v_2, ..., v_r \rangle$*, donde* v_1 *es el head(inicio) de Q y* v_r *el final de la cola. Entonces,* $d[v_r] \leq d[v_1 + 1]$ *y* $d[v_i] \leq d[v_{i+1}]$ *para i= 1,2,..,r-1*

Demostración: La prueba es por inducción sobre el número de operaciones de encolación. Inicialmente, cuando la cola contiene solo s, el lema se mantiene. Por inducción, nosotros debemos probar que el lema se mantiene en ambos encolando y desencolando un vértice. Si el head(inicio) v_1 de una cola es desencolado, la nueva head(inicio) será v_2 (Si la cola llega a estar vacia, entonces el lema se mantiene). Pero entonces tenemos que $d[v_r] \leq d[v_1] + 1 \leq d[v_2] + 1$ y las desigualdades restantes no son afectadas. Además, el lema sigue con v_2 como el head(inicio).

En la linea 16 de BFS, cuando el vértice v es encolado, además de volverse el v_{r+1}, el head(inicio) v_1 de Q es en efecto el vértice u el cual su lista de adyacencia actualmente se viene analizando. Nosotros también tenemos que $d[v_r] \leq d[v_1] + 1 = d[u] + 1 = d[v] = d[v_{r+1}]$ y las desigualdades no son afectadas. Además el lema continua cuando v es encolado.

Teorema 3.2 *(Búsqueda en anchura mejorada)*

Sea $G = (V, E)$ *una grafo directo o indirecto y supongamos que la BFS es ejecutada sobre G desde un vértice inicial* $s \in V$*. Entonces, durante su ejecución BFS descrubre todos los vértices* $v \in V$ *que son accesibles desde el vértice inicial s, con la restriccción de* $d[v] = \delta(s, v)$ *para todo* $v \in V$*. Además, para cualquier vertice* $v \neq s$ *que es accesado desde s, uno de los caminos cortos desde s hacia v es el camino corto de s hacia* $\pi[v]$ *siguiendo por la arista* $(\pi[v], v)$

Demostración : Empezaremos con el caso en el cual v es no accesible desde s. Luego por el lema 3.2, dado $d[v] \geq \delta(s, v) = \infty$, el vértice v no tiene valores finitos para d[v] en la linea 14. Por inducción, no existe un primer vértice cuyo valor d llegue al infinito en la linea 14. La linea 14 es solo ejecutada para vértices con valores d finitos. Entonces, si v es no accesible, este no es descubierto.

La siguiente parte de la prueba será para vértices accesibles desde s. Sea V_k el que denotará a el conjunto de los vértices de distancia k desde s, esto es, $V_k = \{v \in V : \delta(s, v) = k\}$. La prueba será por inducción sobre k. Como hipótesis inductiva, nosotros asumiremos que para vértice $v \in V_k$, existe exactamente un punto durante la ejecución de BFS el cual:

- v es gris
- d[v] es conjunto para k
- Si $v \neq s$, entonces $\pi[v]$ es par de u para algun $u \in V_{k-1}$
- v e insertado dentro de la cola Q

Nosotros notaremos antes, que existe ciertamente a lo mucho un tal punto.

La base es para $k = 0$. Nosotros tenemos $V_0 = \{s\}$, luego el vértice inicial s es el único vértice de distancia 0 a s. Durante la inicialización, s es gris, $d[s]$ es de valor 0 y s es colocado en Q, la hipótesis de inducción se mantiene.

Por inducción, nosotros empezaremos por notar que la cola Q nunca está vacia hasta que termina el algoritmo y además, un vértice u es insertado en la cola,ni tampoco $d[u]$ y $\pi[u]$ aun cambian.. Por el Lema 2.3, además, si los vértices son insertados en la cola durante el trascurso del algoritmo en el orden v_1, v_2,... v_r, entonces la secuencia de distancia es monotonamente creciente: $d[v_i] \leq d[v_{i+1}]$ para $i = 1, 2, ..., r-1$.

Ahora consideremos un vértice arbitrario $v \in V_k$, donde $k \geq 1$. La propiedad de monótona, combinada con $d[v] \geq k$ (del lema 2.2) y la hipótesis inductiva, implica que v debe ser descubierto después de que todos los vértices en V_{k-1} son encolados, si se descubre a todos.

Luego $\delta(s, v) = k$, existe un camino de k aristas desde s hacia v y además existe un vértice $u \in V_{k-1}$ tal que $(u, v) \in E$. Sin perdida de generalidades, sea u el primero de esos vértices grises, lo que debe suceder, por inducción, es que todos los vértices V_{k-1} son grises. El algoritmo BFS encola todos los vértices grises, por lo tanto en última instancia u debe aparecer como el head(inicio) de la cola en la linea 10. Cuando u aparece como el head(inicio), su lista de adyacencia es recorrida y v es descubierto. (El vértice v no podria haber sido descubierto antes, ya que este no es adyacente a cualquier vértice en V_j para $j < k-1$, de otro modo, v no podría pertenecer a V_k y por supuesto, u es el primer vértice descubierto en V_{k-1} el cual v es adyacente). La linea 13 pone gris los vértices v, la linea 14 establece $d[v] = d[u] + 1 = k$, la linea 15 le asigna a $\pi[v]$ u y en la linea 16 inserta v en la cola. Como v es una vértice arbitrario en V_k, la hipótesis inductiva fue probada.

Concluida la prueba del lema, observamos que si $v \in V_k$, entonces vemos que $\pi[v] \in V_{k-1}$. Además, obtenemos un camino corto de s hacia v por caminos cortos de s hacia $\pi[v]$ y entonces pasar por la arista $(\pi[v], v)$

3.4.3. Búsqueda en Profundidad

La estrategia que seguiremos para la búsqueda en profundida, como su mismo nombre lo dice será profundizar la búsqueda en el grafo lo más que se pueda. El algoritmo de recorrido en profundidad DFS (Depht-first search), explora sistemáticamente las aristas del grafo de manera que primero se visitan los vértices adyacentes del visitado más recientemente. De esta forma se va profundizando en el grafo, es decir alejandose del vértice inicial. Cuando la exploración de estas aristas no encuentra algun vértice más, se retrocederá al vértice predecesor de este repitiendo el mismo proceso de visita.

En la búsqueda en anchura se reconocia los vértices adyacentes del vértice descubierto por el recorrido de la lista de adyacencia, en la búsqueda en profundidad se buscará almacenar dichos datos como el predecesor del vértice que se explora, llegando a formar un conjunto de predecesores de un determinado vértice. Este subgrafo predecesor formará un árbol donde quizas este compuesto de muchos árboles más, llamados árboles en profundidad.

Tomaremos también que cada vértices incialmente será inicializado con el color blanco, será gris cuando es descubierto y será negro cuando ha finalizado. Esto nos garantizará que todos los vértices finalizarón exactamente en un árbol en profundidad, así sean árboles disjuntos.

Tomaremos en cuenta que cada vértices llevará una marca, d[v] que será cuando el vértice v fue encontrado y llevará color gris, y la segunda marca f[v] cuando se finalizó de examinar la lista de adyacencia del vertice v y llevará color negro. Almacenará además en la variable d[u] cuando es descubierto el vértice u y en la variables f[u] cuando finaliza con el vértice u

Veamos ahora el siguiente algoritmo que nos indica el proceso de DFS con la entrada de un grafo G, ya sea dirigido o no dirigido.

```
DFS (G)
1   Para cada vertice u en V[G]
2      Hacer color [u]= blanco
3           pi[u]= nil
4   veces = 0
5   Para cada vertice u en V[G]
6      Hacer Si color [u]= blanco
7               Entonces DFS-Visitar(u)
8   Fin


DFS-Visitar (u)
1    color [u]= gris                 El vertice u ha sido descubierto
2    d[u]= tiempo= tiempo + 1
3    Para cada v en Adj[u]           Explora la arista (u,v)
4        Hacer si color[v]= blanco
5           Entonces pi[v]= u
6             DFS-Visitar(v)
7    color[u]= negro                 Vertice u a color negro, esta finalizando
8    f[u]=tiempo= tiempo +1
```

Análisis de Complejidad:

Observamos en el algoritmo DFS que el tiempo de llamada en las lineas 1-2 y las lineas 5-7 son de tiempo $\Theta(V)$. El procedimiento DFS-Visitar es llamado exactamente una vez para cada vértice $v \in V$, luego DFS-visitar es invocado solo en vértices blancos y lo primero que hará será pintarlos de gris. Durante esta ejecución de DFS-visitar(v),la secuencia de lineas 3-6 es ejecutada en $\mid Adj[v] \mid$ veces. Luego tendremos que:

$$\sum_{v \in V} \mid Adj[v] \mid = \Theta(E)$$

el costo total de las lineas 2-5 de DFS-visitar es $\Theta(E)$. El tiempo de ejecución de DFS, será de $\Theta(V + E)$

3.4.4. Propiedades de la búsqueda en profundidad

La búsqueda en profundidad nos puede facilitar mucha información de la estructura de un grafo. La propiedad más importante nos dice que el subgrafo predecesor G_π forma de hecho un bosque de arboles, luego la estructura de los árboles en profundidad son exactamente espejo de la estructura recursiva del llamado DFS-Visitar(u). Esto es, $u = \pi[v]$ si y solo si DFS-Visitar(v) fue llamado durante la búsqueda de la lista de adyacencia de u.

Otra propiedad importante es que descubrimiento y los tiempos de finalización tiene una *estructura de paréntesis*. Si representamos el descubrimiento de un vértice u con un paréntesis a la izquierda '*(u*' y luego representamos su tiempo de finalización por un parentesis a la derecha '*u)*', entonces su descubrimiento y finalización será una expresión que estarán bien definida en ese sentido los paréntesis son jerazquizados correctamente.

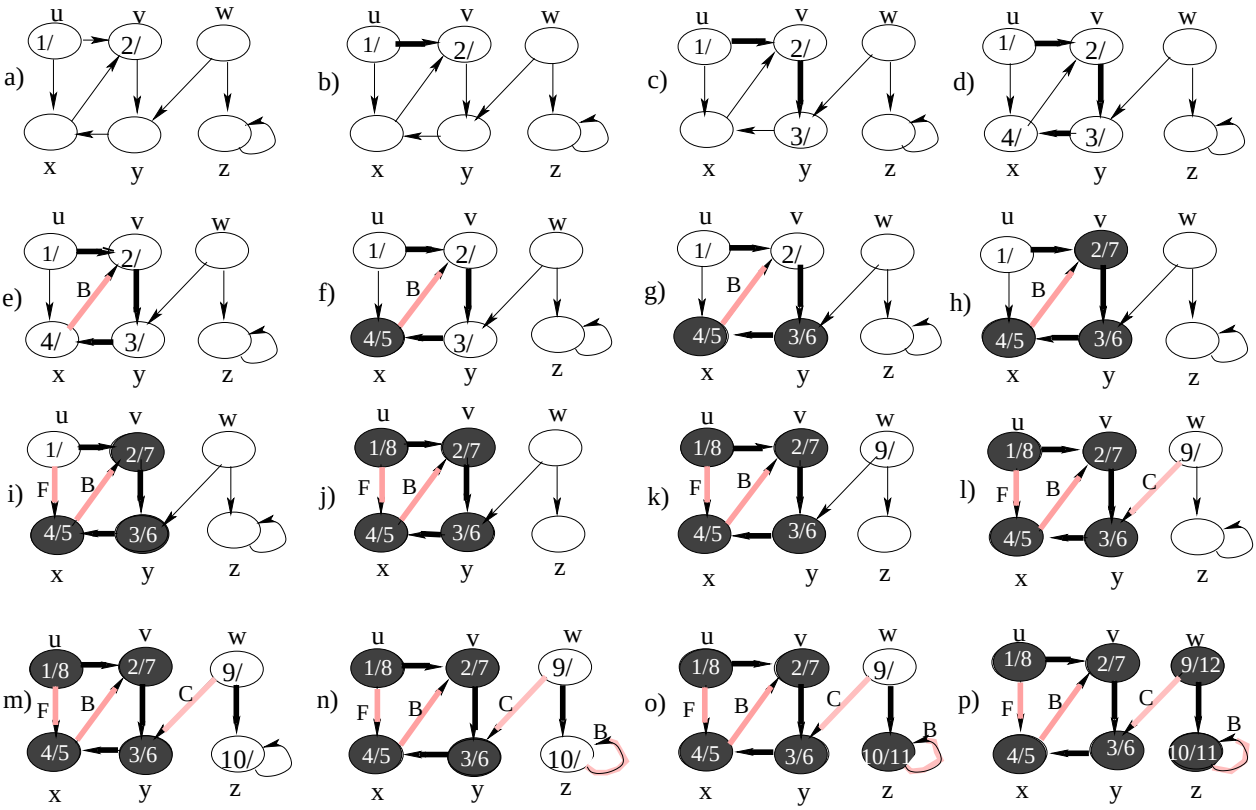


Figura 3.3: *Se muestra el proceso de búsqueda en Profundidad de un grafo dirigido. Las aristas que son exploradas se muestran de color oscuro o claro. Las arista con las marcas B, C o F indican si es una arista trasera(B), arista delantera(F) o arista cruzada(C) El tiempo de descubrimiento y finalización se muestra en las aristas*

Teorema 3.3 *(Teorema del paréntesis) En cualquier búsqueda en profundidad de un grafo directo o indirecto $G = (V, E)$, para cualquiera 2 vértices u y v, exactamente una de las siguientes 3 condiciones se cumple:*

- *El intervalo $[d[u], f[u]]$ y $[d[v], f[v]]$ son completamente disjuntos*
- *El intervalo $[d[u], f[u]]$ es contenido enteramente dentro del intervalo $[d[v], f[v]]$ y u es un descendiente de v en el árbol en profundiad.*
- *El intervalo $[d[v], f[v]]$ es contenido enteramente dentro del intervalo $[d[u], f[u]]$ y v es descendiente de u en el árbol en profundidad*

Demostración : Empezemos con el caso en el cual $[d[u] < d[v]]$. Existen 2 subcasos por considerar, dependiendo de que $[d[v] < f[u]]$ o no. En el primer caso $[d[v] < f[u]]$, asi v fue descubierto mientras u era aun gris. Esto implica que v es descendiente de u. Sin embargo, luego v fue descubierto más recientemente que u,todas sus aristas salientes son exploradas y v es finalizado,antes que la búsqueda retorne y finalice u. En este caso, por lo tanto, el intervalo $[d[v], f[v]]$ esta enteramente contenido dentro del intervalo $[d[u], f[u]]$. En el otro subcaso,$[d[v] > f[u]]$, por la desigualdad hallada en DFS, que nos indica que $[d[v] < f[u]]$ para todo vértice u, implica que los intervalos $[d[u] < f[u]]$ y $[d[v] < f[v]]$ son disjuntos.

Corolario 3.1 *(Jerarquización de intervalos descendentes) El vértice v es un descendente apropiado de el vértice u en el bosque en profundidad para un grafo directo o indirecto G si y solo si* $d[u] < d[v] < f[v] < f[u]$

Demostración: Por el teorema 3.3 queda comprobado

Teorema 3.4 *(El Teorema del camino blanco) En un bosque en profundidad de un grafo directo o indirecto G=(V,E), el vértice v es un descendiente del vértice u si y solo si en el instante d[u] que se descubre u, el vértice v puede ser alcanzado desde u a lo largo del camino enteramente formado por vértices blancos.*

Demostración :($\Rightarrow$) Asumamos que v es descendiente de u. Sea ω cualquier vértice sobre el camino entre u y v en el arbol en profundidad, asi que ω es un descendiente de u. Por el corolario 3.1, $[d[u] < d[\omega]]$ y así ω es blanco en el instante $d[u]$.

($\Leftarrow$) Supongamos que el vértice v es accesible desde u a lo largo de un camino de vértices blancos en el instante $d[u]$, pero v no llega a ser descendiente de u en el árbol en profundidad. Sin perdida de generalidades, asumamos que todos los otros vértices a lo largo del camino llegan a ser descendiente de u (de otro manera, v sería el más cercano vértice a u a lo largo del camino que no llega a ser descediente de u). Sea ω el predecesor de v en el camino, asi que ω es el descendiente de u (ω y u pueden ser algún vértice) y por el corolario 3.1, $[f[\omega] \leq f[u]]$. Notamos que v debe ser descubierto luego que u es descubierto, pero antes que ω finalice. Por lo tanto, $d[u] < d[v] < f[\omega] \leq f[u]$. El teorema 3.3 entonces implica que el intervalo $[d[v], f[v]]$ es contenido enteramente dentro de el intervalo $[d[u], f[u]]$. Por el corolario 3.1, v debe ser después de todo un descendiente de u

3.4.5. Clasificación de Aristas

Otra propiedad importante consiste en la clasificacion de aristas en la entrada de un grafo $G = (V, E)$. Nosotros podemos definir 4 tipos de aristas en terminos de bosques en profundidad G_π producido por una búsqueda en profundidad sobre G.

1. **Arista de árbol**, son arista en un bosque en profundidad G_π. La arista (u, v) es una arista del árbol si v fue el primero en descubrirse al explorar la arista (u, v)

2. **Arista Trasera**, son aquellas aristas (u, v) que conectan un vértice a un antecesor v en un árbol en profundidad. Un lazo de si mismo es considerado una arista de regreso.

3. **Arista Delantera**, son aquellas aristas de ningún árbol (u, v) conectadas a un vértice u a un descendiente v es un árbol de profundidad

4. **Aristas Cruzadas**, son todas las otras aristas. Estas pueden ir entre vértices en el mismo árbol en profundidad, mientras un vértice no sea antecesor de la otra, o puede ser entre vértices en direferente árboles en profundidad.

3.4.6. Clase topológica

Veremos ahora como una búsqueda en profundidad puede ser usada para una clase topológica de un grafo dirigido acícico o conocido como DAG(Directed Acyclic Graph). Una clase topológica de un DAG $G = (V, E)$ es un ordenamiento lineal de todos sus vértices tal que si G contiene un vértice (u, v), entonces u aparece antes que v en el ordenamiento (Si el grafo es no acíclico, entonces el ordenamiento no lineal es posible). Una clase topológica de un grafo puede ser vista

como un ordenamiento de sus vértices a lo largo de una linea horizontal, tal que todas sus aristas dirigidas van de izquierda a derecha.

Los grafos dirigidos acíclicos son usado en muchas aplicaciones para indicar el precedentes entre eventos. En la figura 3.4 da un ejemplo al momento de vestirse en la mañana. El personaje debe ponerse una prenda antes que otra (es decir, medias antes que zapatos). Otro punto señala que puede ser en cualquier orden(es decir medias y luego pantalones). Una arista dirigida (u, v) en el DAG de la figura 3.4 indica que la prenda u debe ser puesta antes que la prenda v. Una clase topológica de este DAG por lo tanto da un orden para conseguir vestirse.

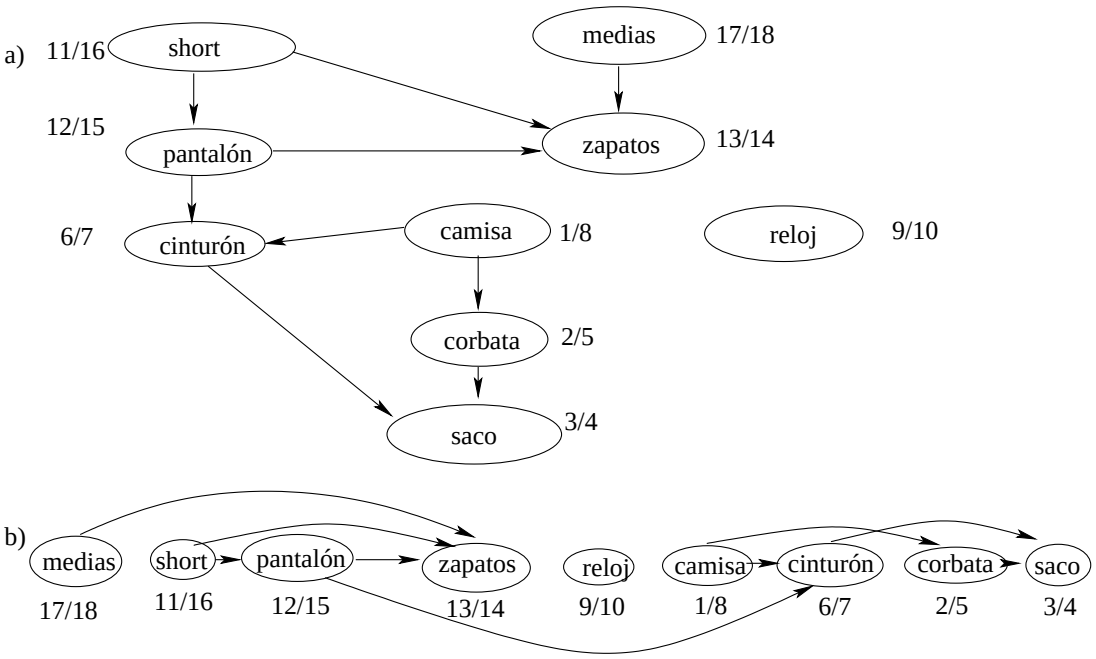


Figura 3.4: *a) Muestra la clasificación topológica de como vestirse. Cada arista dirigida* (u, v) *significa que prenda* u *se pondrá antes que la otra* v*. El tiempo o paso de descubrimiento y finalización de la búsqueda en profundidad se muestra al lado del vértice. b) El mismo grafo topologicamente ordenado. Sus vértices son organizados de izquierda a derecha en orden decreciente al tiempo de finalización. Observar que todas las aristas dirigidas van de izquierda a derecha*

El siguiente algoritmo simple realiza la búsqueda topológica en un DAG

```
CLASE-TOPOLOGICA (G)

1   Llamar B\'USQUEDA EN PROFUNDIDAD (G) para calcular el tiempo de
    finalizacion f[v] para cada vertice v
2   Como cada vertice es finalizado, insertar este al inicio de una
    lista encadenada
3   Retornar la lista encadenada de los vertices.
```

En la figura 3.4 muestra el DAG clasificado topologicamente como un ordenamiento de vértices a lo largo de una linea horizontal tal que todas las aristas dirigidas van de izquierda a

derecha. Se observa que aparecen en orden decreciente a su tiempo de finalización

El tiempo de ejecución de la clase topológica es $\Theta(V+E)$, ya que la búsqueda en profundidad toma un tiempo $\Theta(V+E)$ y el insertar cada arista $\mid V \mid$ al inicio de la lista encadenada es de $O(1)$.

Lema 3.4 *Un grafo dirigido G es acíclico si y solo si una búsqueda en profundidad de G no produce aristas trasera.*

Demostración :($\Rightarrow$) Supongamos que existe una arista trasera(u,v). Entonces, el vértice v es una antecesor del vértice u en la bosque de búsqueda en profundidad. Existe además un camino desde v hacia u en G y la arista trasera (u, v) completa el ciclo.

($\Leftarrow$)Supongamos que G contiene una ciclo c. Mostraremos que una búsqueda en profundidad de G produce una arista trasera. Sea v el primer vértice v en ser descubierto en c, y sea (u, v) la arista precedente en c. En el tiempo $d[v]$, existe un camino de vértices blancos desde v hacia u. Por el teorema de camino-blanco, el vértice u llega a ser descendiente de v en el bosque de búsqueda en profundidad. Por lo tanto (u, v) es una arista trasera.

Teorema 3.5 *El algoritmo CLASE-TOPOLÓGICA(G) produce una clase topológica de un grafo acíclico dirigido G*

Demostración: Supongamos que DFS se ejecuta dado un DAG $G = (V, E)$ para determinar el tiempo de finalización de sus vértices. Es suficiente mostrar que para cualquier par de vértices distintos u,$v \in V$, si existe una arista en G desde u hacia v, entonces $f[v] < f[u]$. Consideremos cualquier arista (u, v),explorado por DFS(G). Cuando esta arista es explorada, v no puede ser gris, entonces v puede ser antecesor de u y (u, v) puede ser una arista tracera, contradiciento el lema 3.4. Por lo tanto, v debe ser blanco o negro. Si v es blanco, llega a ser un descendiente de u y asi $f[v] < f[u]$. Si v es negro, entonces $f[v] < f[u]$ esta bien. Así, para cualquier arista (u, v) en el DAG, tenemos que $f[v] < f[u]$ probando el teorema.

Capítulo 4

Modelación mediante Grafos: Algoritmos para Caminos Mínimos

Mediante la teoría de grafos pueden representarse gran número de situaciones que supongan relaciones entre diversos elementos.

4.1. Posibilidades de Comunicación

De manera natural, un grafo puede representar las posibilidades de comunicación existentes entre diferentes puntos. Lo más frecuente es que los puntos esten representados en el grafo mediante vértices y las posibilidades de comunicación mediante arco (o en ocasiones aristas, si la comunicación entre dos nodos es siempre igual entre los dos sentidos). La representación de las posibilidades de comunicación se completa asociando a cada arista una magnitud relevante para la representación (distancia, tiempo, etc).

En el grafo se representa las diferentes posibilidades de comunicación entre cinco ciudades (A,B,C,D y E). Los valores sobre los arcos representan los tiempos de traslado de un punto a otro.

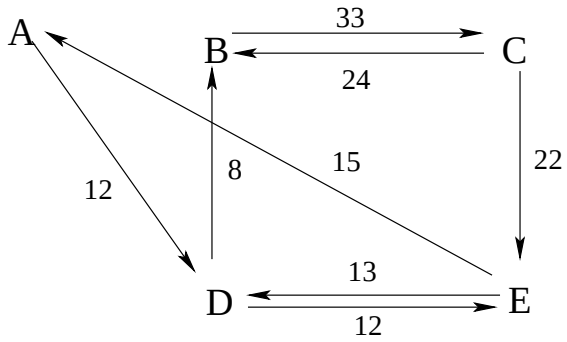


Figura 4.1: Posibilidad de Comunicación

Mediante grafos de este tipo podemos resolver problemas de conectividad (árbol parcial mínimo) o problemas de caminos, en que lo que se trata de encontrar el camino de mínima distancia entre 2 vértices del grafo.

4.2. Problemas de Caminos

En los problemas de caminos, se trabaja con grafos cuyos arcos o aristas tienen asociado un determinado valor que puede corresponder, aunque no necesariamente, a la distancia entre los vértices unidos por el arco o arista). Dichos problemas suelen reducirse a encontrar:

1. El conjunto de aristas que conecta todos los nodos del grafo, tal que la suma de los valores asociados a las aristas sea mínimo. Dado que dicho conjunto ha de ser necesariamente un árbol, suele llamarse a este problema de árbol parcial mínimo.

2. El camino más corto desde un vértice origen hasta otro extremo. Se denominará camino más corto a aquel cuya suma de los valores asociados de los arcos que lo componen sea mínimo.

4.3. Grafos Dirigidos: Caminos mínimos

Un motorista desea encontrar la ruta más corta posible entre Lima y Arequipa. Dado un mapa de carreteras de Perú debemos resolver como calculamos la ruta de menor distancia.

Un posible forma sería enumerando todas las rutas de Lima a Arequipa, sumando las distancias de cada ruta y seleccionar la más corta. Eso es fácil de ver, sin embargo, aún si rechazamos rutas que son cíclicas, se presentan muchas posibilidades.

En el problema de caminos corto, nosotros damos los pesos, un grafo dirigido $G = (V, E)$, con una función de peso $w : E \to R$ aplicada a las aristas con pesos de valores reales. El peso del camino $p =< v_0, v_1, v_2, ..., v_k >$ es la suma de los pesos que sus aristas contituyen:

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

Nosotros definimos el camino de menor peso de u hacia v por:

$$\delta(u, v) = \begin{cases} min\{w(p) : u \overset{p}{\rightsquigarrow} v\} & \text{si existe un camino desde } u \text{ hacia } v \\ \infty & \text{sino existe} \end{cases}$$

Un camino corto desde el vértice u hacia el vértice v es entonces definido como cualquier camino p con peso $w(p) = \delta(u, v)$.

Los pesos que llevan las aristas pueden ser interpretados como un tipo de medida sobre las distancias. Estos puede ser a menudo tiempo, costo, perdidas o cualquier otra cantidad que se acumula a lo largo de un camino y que uno desea minimizar.

El algoritmo de búsqueda en anchura que se analizó con anterioridad es un algoritmo de camino mínimo que trabaja sin pesos sobre el grafo, esto es, que cada arista del grafo lleva como valor de peso la unidad.

En esta sección veremos las distintas variantes que puede existir teniendo una fuente simple (punto dado inicial) para el problema de caminos mínimos. Dado un grafo $G = (V, E)$ nosotros queremos encontrar el camino mínimo desde una fuente (o punto inicial) el cual ser´ía un vértice $s \in V$ con todos los vértices $v \in V$. Muchos otros problemas pueden ser resueltos por el algoritmo para el caso que sea desde una fuente simple, incluyendo las siguientes variantes:

- ***Problema de Caminos mínimos para destino simple***: Encontrar un camino mínimo para un vértice destino t dado, desde todos los vértices v. Si invertimos la dirección de cada arista en el grafo, nosotros podemos reducir este problema a el problema de fuente simple

- ***Problema de camino mínimo para una pareja simple***: Encontrar el camino mínimo desde u hacia v para un par u y v dados. Si nosotros resolvemos el problema de fuente simple con un vértice inicial u, nosotros podemos resolver este problema también. Sin embargo, no hay algoritmos para este problema donde su tiempo de ejecución es asintóticamente más rápido que los mejores algoritmos de fuente simple en el peor de los casos.

- ***Problema de camino mínimo para todas las parejas***: Encontrar un camino mínimo desde u hacia v para toda pareja de vértices u y v. Este problema puede ser resuelto ejecutando un algoritmo de fuente simple a cada uno de los vértices, pero esto puede usualmente resolverse rápido, y su estructura es de gran interés.

4.3.1. Pesos negativos

En algunas instancias del problema de caminos mínimos para un fuente simple, existirán aristas las cuales su peso será negativo.

Si el grafo $G = (V, E)$ no contiene ciclos de pesos negativos accesibles desde una fuente s, entonces para todo $v \in V$, el camino de peso mínimo $\delta(s, v)$ estará bien definido, aún si estos tienen valores negativos.

Si existe un ciclo de pesos negativos accesibles desde s, el camino de pesos mínimo no estará bien definido. No hay un camino desde s hacia un vértice sobre un ciclo que pueda ser el camino más corto; un camino de menor peso puede ser siempre encontrado siguiendo el procedimiento de caminos 'mínimos' y entonces atravesar los ciclos de pesos negativos.

Si existe un ciclo de pesos negativos sobre alguno camino desde s hacia v, nosotros lo definiremos como $\delta(s, v) = -\infty$

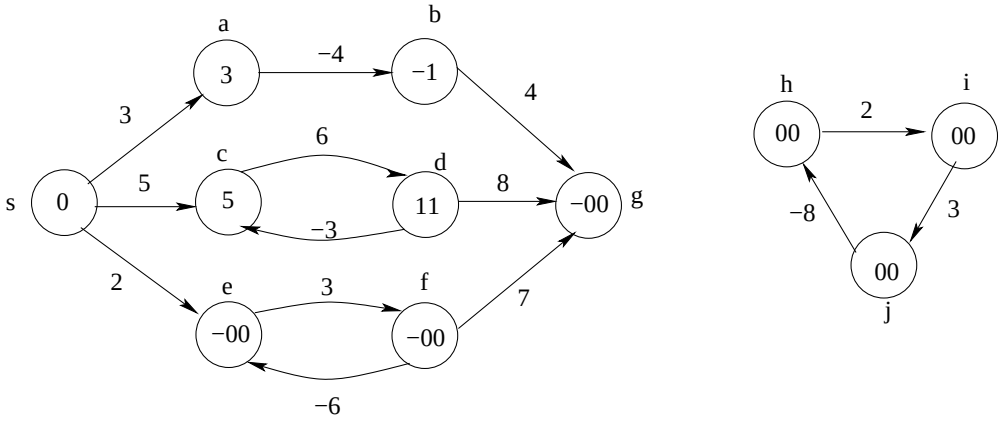


Figura 4.2: *Se muestra un grafo dirigido de aristas de peso negativos, y dentro de cada vértice su valor de camino mínimo desde la fuente s. Como los vértices e y f forma un ciclo de pesos negativos accesible desde s, ellos tienen un camino de pesos mínimo de valor $-\infty$. Como el vértice g es accesible desde el vértice el cual tiene camino mínimo $-\infty$ este también tiene el mismo camino mínimo $-\infty$. Los vértices como h, i y j no son accesibles desde s, y esto tienen un camino de pesos mínimo de valor ∞, aún siendo un ciclo de pesos negativos*

Si observamos se ilustración en la figura, el efecto negativo de los pesos sobre los caminos mínimos. Porque existe solo un camino desde s hacia a (camino (s, a)), $\delta(s, a) = \omega(s, a) = 3$. Similarmente, existe un único camino de s hacia b, y es $\delta(s, b) = \omega(s, a)+\omega(a, b) = 3+(-4) = -1$. Existe infinitos caminos desde s hacia c: (s, c); (s, c, d, c); (s, c, d, c, d, c) y sobre este. Como el ciclo (c, d, c) tiene un peso de $6+(-3) = 3 > 0$, el camino mínimo desde s hacia c es (s, c), con peso $\delta(s, c) = 5$. Similarmente, el camino de s hacia d es (s, c, d) con peso $\delta(s, d) = \omega(s, a) + \omega(c, d) = 11$. Analogamente, existe infinitos caminos desde s hacia e: (s, e); (s, e, f, e); (s, e, f, e, f, e) y sobre este. Luego el ciclo (e, f, e) tiene un peso de $3 + (-6) = -3 < 0$, sin embargo no existen caminos mínimos de s hacia e. Atravezando los ciclos de pesos negativos (e, f, e) muchas veces, podremos encontrar caminos desde s hacia e con grandes pesos negativos y además $\delta(s, e) = -\infty$. Similarmente, $\delta(s, f) = -\infty$ ya que g es accesible desde f podremos encontrar caminos con grandes pesos negativos desde s hacia g y $\delta(s, g) = -\infty$. Los vértices h,i y j forman un ciclo de pesos negativos. Estos no son accesibles desde s sin embargo $\delta(s, h) = \delta(s, i) = \delta(s, j) = -\infty$

Algunos algoritmos como el de Dijkstra asumen que todos los pesos son no negativos. En cambio el algoritmo de Bellman-Ford permite el uso de pesos negativos.

4.3.2. Representación de caminos mínimos

Dado un grafo $G = (V, E)$, nosotros mantenemos para cada vértices $v \in V$ un predecesor $\pi[v]$ que es cualquier otro vértice o Nulo. Los algoritmos de caminos mínimos presentan un conjunto de valores π que forman una cadena de predecesores originado por un vértice v, que recorre un camino mínimo desde s hasta v.

Durante la ejecución del algoritmo de caminos mínimos, sin embargo, los valores π no necesitan indicar los caminos mínimos. Como en la búsqueda en anchura, debemos estar interesados en el subgrafo predecesor $G_\pi = (V_\pi, E_\pi)$ inducido por el valor π. Asi definiremos el conjutno de vértices V_π como el conjunto de vertices de G con predecesores no nulos, más la fuente s:

$$V_\pi = \{v \in V : \pi[v] \neq NIL\} \cup \{s\}$$

Las aristas dirigidas es el conjunto E_π que es inducido por los valores π por los vértices en V_π:

$$E_\pi = \{(\pi[v], v) \in E : v \in V_\pi - \{s\}$$

Nosotros debemos probar que los valores π producidos por el algoritmo tienen la propiedad que al finalizar G_π es un árbol de caminos mínimos, informalmente un árbol enraizado contiene un camino mínimo desde una fuente s hacia todo vértice que es accesible desde s. Un árbol de caminos mínimos es como el árbol de búsqueda en anchura pero contiene caminos mínimos desde la fuente definida en función a los pesos de sus aristas en lugar del número de aristas. Es decir, sea $G = (V, E)$ un grafo dirigido, con una función de pesos $\omega : E \to R$ y asumimos que G contiene ciclos de pesos no negativos accesibles desde una vértice inicio $s \in V$, tal que los caminos mínimos estan bien definidos. Un árbol enraizado de caminos mínimos en s es un subgrafo dirigido $G' = (V', E')$ donde $V' \subseteq V$ y $E' \subseteq E$ tal que:

1. V' es un conjunto de vértices accesibles desde s en G
2. G' forma un árbol enraizado con una raiz s
3. Para todo $v \in V'$, el único camino simple desde s hacia v en G' es un camino mínimo desde s hacia v en G.

Los caminos mínimos no son necesariamente únicos, y algunos no son árboles de caminos mínimos.

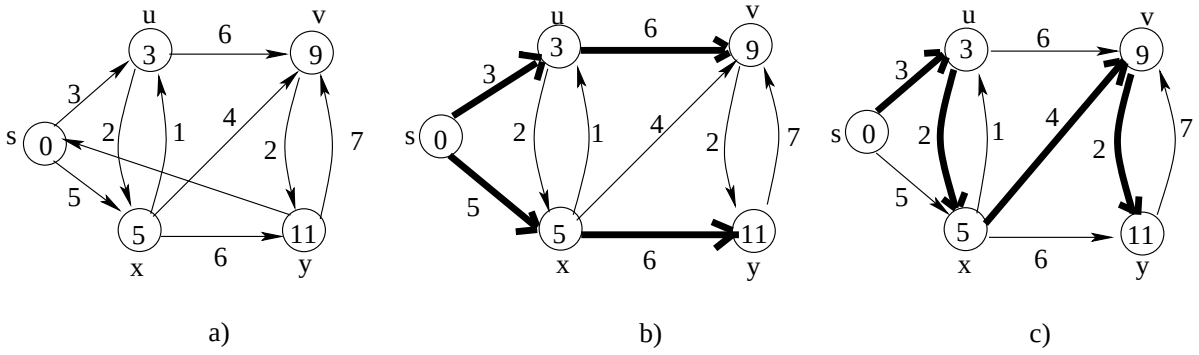


Figura 4.3: *a) Un grafo dirigido con un camino de pesos mínimo y fuente s. b) Las aristas oscuras forman un árbol de camino mínimo enraizado a la fuente s. c) Otro árbol de camino mínimos con la misma raiz*

4.4. Caminos mínimos y Relajación

Para entender mejor los algoritmos de caminos mínimos con destino simple, utilizaremos algunas técnicas y características que poseen estos caminos mínimos al ser explorados. La técnica que usaremos se denomina relajación, un método que va disminuyendo en varias ocasiones una cota superior encontrada sobre el actual camino de pesos mínimo de cada vértice, hasta que la cota superior sea igual al camino de pesos mínimo. Veremos ahora como trabaja la relajación y formalmente probaremos muchas de sus caracteristicas.

4.4.1. Estructura óptima de un camino mínimo

Los algoritmos de caminos mínimos tipicamente exploran la característica que un camino mínimo entre 2 vértices contiene otro camino mínimo en este. El siguiente lema y su corolario establecen la característica de subestructura óptima de caminos mínimos de manera más precisamente.

Lema 4.1 *(Los subcaminos de caminos mínimos son caminos mínimos) Dado un peso, un grafo dirigido $G = (V, E)$ con una función de peso $w : E \to R$, sea $p = (v_1, v_2, ..., v_k)$ un camino mínimo desde el vértice v_1 hacia el vértice v_k y para cualquier i y j tal que $1 \leq j \leq k$, sea $p_{ij} = (v_i, v_{i+1}, ..., v_j)$ sea un subcamino de p desde el vértice v_i hasta el vértice v_j. Entonces, $p_{i,j}$ es un camino mínimo desde v_i hasta v_j.*

Demostración: Si descomponemos el camino p en $v_1 \overset{p_{1j}}{\rightsquigarrow} v_i \overset{p_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k$ entonces $w(p) = w(p_{1i}) + w(p_{ij}) + w(p_{jk})$. Ahora asumiremos, que existe un camino p'_{ij} desde v_i hasta v_j con peso $w(p'_{ij}) < w(p_{ij})$. Entonces, $v_1 \overset{p_{1j}}{\rightsquigarrow} v_i \overset{p'_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k$ es un camino desde v_1 hasta v_k el cual su peso $w(p) = w(p_{1i}) + w(p'_{ij}) + w(p_{jk})$ es menor que $w(p)$, lo cual contradice la premisa de que p es un camino mínimo desde v_1 hasta v_k.

Cuando se analizó en el capítulo anterior la búsqueda en anchura, se probo el lema 3.1 una característica simple de camino de distancia mínima en grafos sin pesos. Ahora en el siguiente corolario generalizaremos esta características para grafos con pesos

Corolario 4.1 *Sea $G = (V, E)$ un grafo dirigido con pesos y la función $\omega : E \to R$. Supongamos que existe un camino mínimo p desde una fuente s hacia un vértice v puede ser descompuesto en $s \overset{p'}{\rightsquigarrow} u \to v$ para algún vértice u y camino p'. Entonces, el peso de un camino mínimo desde s hacia v es $\delta(s, v) = \delta(s, u) + \omega(u, v)$*

Demostración: Por el lema 4.1, el subcamino p' es un camino mínimo desde la fuente s hacia el vértice u. Además:

$$\begin{aligned} \delta(s,u) &= \omega(p) \\ &= \omega(p') + \omega(u,v) \\ &= \delta(s,u) + \omega(u,v) \end{aligned}$$

Lema 4.2 *Sea $G = (V, E)$ un grafo dirigido con una función de peso $\omega : E \to R$ y un vértice fuente s. Entonces, para todas las aristas $(u, v) \in E$, tenemos que $\delta(s, v) \leq \delta(s, u) + \omega(u, v)$*

Demostración : Un camino mínimo p desde una fuente s hacia un vértice v no tiene más peso que cualquier otro camino desde s hacia v. Especificamente, un camino p no tiene más peso que un camino en particular que toma un camino mínimo desde una fuente s hacia un vértice u y entonces toma la arista (u, v)

4.4.2. Relajación

Algunos de los algoritmos que utilizaremos utilizan la técnica de relajación. Para cada vértice $v \in V$, nosotros mantendremos un atributo $d[v]$, el cual es una cota superior sobre el peso de un camino mínimo desde una fuente s hacia v. Nosotros llamaremos $d[v]$ un camino mínimo estimado. Mediante el siguiente procedimiento inicializaremos el camino mínimo estimado.

```
INICIALIZAR-FUENTE-SIMPLE (G,s)
1.   Para cada vertice v en V[G]
2.       Hacer d[v]= infinito
3.         pi[v]=NIL
4.   d[s]=0
```

Se inicializa, $\pi[v] = NIL$ para todo $v \in V$, $d[v] = 0$ para $v = s$ y $d[v] = \infty$ para $v \in V - \{s\}$

El proceso de relajar una arista (u, v) consiste en ir probando si podemos mejorar el camino corto a v encontrado cuan lejos puede ir hacia u, he ir actualizando $d[v]$ y $\pi[v]$. Un paso de relajación quizas decrezca el valor de un camino mínimo estimado $d[v]$ y actualiza el campo predecesor de v que es $\pi[v]$. El siguiente proceso nos indica un paso de relajación sobre la arista (u, v).

```
RELAJAR(u,v,w)
1.   Si d[v]> d[u] + w(u,v)
2.    Entonces d[v]= d[u]+ w (u,v)
3.      pi[v]= u
```

Cada algoritmo hará un llamado de inicialización y luego repetidamente relajará aristas. Sin embargo, la relajación es la única forma por la cual el camino mínimo estimado y el predecesor cambia. En el algoritmo de Dijkstra y el algoritmo de caminos mínimos para grafos acíclicos y dirigidos, cada arista es relajada solo una vez, en el algoritmo de Bellman-Ford, la arista es relajada muchas veces.

4.4.3. Propiedades de la Rejalación

Las correcciones que puedan existir en este capítulo depende de la importancia de las propiedades de relajación que se resumen en los siguientes lemas. La mayoria de estos lemas describe la salida de la ejecución de una secuencia de pasos de relajación sobre una arista con su determinado peso en un grafo dirigido que ha sido inicializado. Ha excepción del último lema, estos se aplican a cualquier secuencia de pasos de relajación, no solo a los que producen valores de camino mínimo.

Lema 4.3 *Sea $G = (V, E)$ un grafo dirigido con una función de peso $\omega : E \to R$ y sea $(u, v) \in E$. Entonces, inmediatamente después de relajar la arista (u, v) al ejecutar RELAJAR(u,v,w), tendremos que $d[v] \leq d[u] + \omega(u, v)$*

Demostración: Si, antes de la relajar la arista (u, v), tenemos $d[v] > d[u] + \omega(u, v)$ entonces $d[v] = d[u] + \omega(u, v)$. Si, en lugar tenemos, $d[v] \leq d[u] + \omega(u, v)$ antes de la relajación, entonces ni $d[u]$ ni $d[v]$ cambian y también $d[v] \leq d[u] + \omega(u, v)$.

Lema 4.4 *Sea $G = (V, E)$ un grafo dirigido con una función de peso $\omega : E \to R$. Sea $s \in V$ el vértice inicial, y sea el grafo inicializado por INICIALIZAR-FUENTE-SIMPLE(G,s). Entonces, para todo $v \in V$ se tiene que $d[v] \geq \delta(s, v)$ y esta invariante es mantenida sobre la secuencia de pasos de relajación sobre las aristas de G. Además, una vez que $d[v]$ alcanza su cota inferior $\delta(s, v)$, esta nunca cambia.*

Demostración: La invariante $d[v] \geq \delta(s, v)$ es ciertamente verdadera luego de la inicialización ya que $d[s] = 0 \geq \delta(s, s)$ (recordar que $\delta(s, s)$ es $-\infty$ si s esta sobre un ciclo de pesos negativo y 0 en otro caso) y $d[v] = \infty$ implica que $d[v] \geq \delta(s, v)$ para todo $v \in V - \{s\}$. Nosotros debemos probar por contradicción que la invariante se mantiene sobre cualquier secuencia de pasos de relajación. Sea v el primer vértice para el cual un paso de relajación de la arista (u, v) produce $d[v] < \delta(s, v)$. Entonces, solo después de relajar la arista (u, v), tendremos:

$$\begin{aligned} d[u] + \omega(u, v) &= d[v] \\ &< \delta(s, v) \\ &\leq \delta(s, u) + \omega(u, v) \quad \text{(por el lema 4.2)} \end{aligned}$$

el cual implica que $d[u] < \delta(s, u)$. Pero como relajamos la arista (u, v) no produce cambios en $d[u]$, esta desigualdad debe ser verdadera antes de relajar la arista, lo cual contradice el cambio de v como el primer vértice por el cual $d[v] < \delta(s, v)$. Concluimos que la invariante $d[v] \geq \delta(s, v)$ se mantiene para todo $v \in V$.

Vemos que el valor de $d[v]$ nunca cambia una vez que $d[v] = \delta(s, v)$, notamos que alcanzó su cota inferior, $d[v]$ no puede decrecer porque hemos mostrado que $d[v] \geq \delta(s, v)$ y este no puede incrementarse porque los pasos de relajación no incrementan los valores de d

Corolario 4.2 *Supongamos que en un grafo dirigido $G = (V, E)$ con una función de peso $\omega : E \to R$, no existe camino que conecte un vértice inicial(fuente) $s \in V$ a un vértice dado $v \in V$. Entonces, después de inicializar el grafo por INICIALIZAR-FUENTE-SIMPLE(G,s), tenemos $d[v] = \delta(s, v)$ y esta igualdad se mantiene como una invariante sobre cualquier secuencia de pasos de relajación sobre las aristas de G.*

Demostración Por el lema 4.4, tenemos que $\infty = \delta(s, v) \leq d[v]$, asi $d[v] = \infty = \delta(s, v)$

El siguiente Lema nos dará las condiciones suficientes para la relajación y poder hacer una estimación de un camino mínimo que converga hacia un camino de menor peso.

Lema 4.5 *Sea $G = (V, E)$ un grafo con una función de peso $\omega : E \to R$, sea $s \in V$ un vértice fuente, y sea $s \rightsquigarrow u \to v$ un camino mínimo en G para algún vértice $u, v \in V$. Supongamos que G es inicializado por INICIALIZAR-FUENTE-SIMPLE(G,s) y entonces la secuencia de pasos de relajación que incluye el* llamado *RELAJAR(u,v,w) es ejecutado sobre las aristas de G. Si $d[u] = \delta(s, u)$ para cualquier momento antes del* llamado*, entonces $d[u] = \delta(s, u)$ para todo momento después del* llamado

Demostración Por el lema 4.4, si $d[u] = \delta(s, u)$ para algún punto antes de relajar la arista (u, v), entonces la igualdad se mantiene después de eso. En particular, después de relajar la arista (u, v), tenemos

$$\begin{array}{rcll} d[v] & \leq & d[u] + \omega(u, v) & \text{(por el Lema 4.3)} \\ & = & \delta(s, u) + \omega(u, v) & \\ & = & \delta(s, v) & \text{(por corolario 4.1)} \end{array}$$

entonces: $d[v] \leq \delta(s, v)$

Por el lema 4.4, tenemos que $d[v] \geq \delta(s, v)$, por lo tanto concluimos que $d[v] = \delta(s, v)$ y esta igualdad se mantiene después de eso.

4.4.4. Arboles de caminos mínimos

Nosotros mostraremos que la relajación produce el camino mínimo estimado para descender monótonamente hacia el actual camino de pesos mínimo. Debemos también demostrar, que una vez que una secuencia de relajaciones ha calculado el actual camino de pesos mínimos, el subgrafo predecesor G_π inducido por el resultado de los valores de π, es un árbol de caminos mínimos para G. Empezaremos mostrando que el subgrafo predecesor siempre forma un árbol enraizado, el cual tiene como raiz el vértice inicial o fuente.

Lema 4.6 *Sea $G = (V, E)$ un grafo dirigido con una función de peso $\omega : E \to R$ y una vértice inicial(fuente) $s \in V$ y asumimos que G contiene ciclos no negativos que son accesibles desde s. Entonces, después de inicializar el grafo por INICIALIZAR-FUENTE-SIMPLE(G,s), el subgrafo predecesor G_π forma un árbol enraizado con una raiz s y cualquier secuencia de pasos de relajación sobre las aristas de G mantienen esta característica como una invariante.*

Demostración: Inicialmente, el único vértice en G_π es el vértice inicial(fuente) y el lema se cumple trivialmente. Consideremos el subgrafo predecesor G_π que se presenta después de una secuencia de pasos de relajación. Nosotros debemos probar que G_π es acíclico. Por contradicción, supongamos que algún paso de relajación crea un ciclo en el grafo G_π. Sea el ciclo $c = (v_0, v_1, ..., v_k)$ donde $v_k = v_0$. Entonces $\pi[v_i] = v_{i-1}$ para $i = 1, 2, ..., k$ y sin perdida de generalidad, podemos asumir que esta fue la relajación de la arista (v_{k-1}, v_k) que crea el ciclo en G_π

Afirmamos que todos los vértices sobre el ciclo c son accesible desde la fuente s, esto debido a que cada vértice sobre c tiene solo un predecesor no nulo y tambien cada vértice sobre c fue asignado una estimación de camino mínimo finito cuando a este se le asigno su valor de π no nulo. Por el lema 4.4, cada vértice sobre el ciclo c tiene un camino de pesos mínimo finito, el cual implica que este es accesible desde s

Examinaremos las estimaciones del camino mínimo sobre c solo antes del llamado del procedimiento RELAJAR(v_{k-1}, v_k, ω) y mostraremos que c es un ciclo de pesos negativos, lo cual será una contradicción con lo que se asumimos que G contiene ciclo de pesos no negativos

que son accesibles desde la fuente. Después de la ejecución de RELAJAR(v_{k-1}, v_k, ω), tenemos $\pi[v_i] = v_{i-1}$ para $i = 1, 2, ..., k-1$. Además, para $i = 1, 2, ..., k-1$, la última actualización de $d[v_i]$ fue por la asignación $d[v_i] \leftarrow d[v_{i-1}] + \omega(v_i, v_{i-1})$. Si $d[v_{i-1}]$ cambia luego, entonces este decrece. Por lo tanto, solo después de la ejecución RELAJAR(v_{k-1}, v_k, ω), tenemos

$$d[v_i] \geq d[v_{i-1}] + \omega(v_{i-1}, v_i) \qquad \text{para todo } i = 1, 2, ..., k-1 \tag{4.1}$$

Porque $\pi[v_k]$ esta cambiando por el llamado, inmediatamente de antemano también tenemos una desigualdad estricta

$$d[v_k] > d[v_{k-1}] + \omega(v_{k-1}, v_k)$$

Sumando esta desigualdad estricta con las k-1 desigualdades (4.1), obtenemos que la suma de las estimaciones de camino mínimo alrededor del ciclo c produce:

$$\begin{aligned} \sum_{i=1}^{k} d[v_i] &> \sum_{i=1}^{k} (d[v_{i-1}] + \omega(v_{i-1}, v_i)) \\ &= \sum_{i=1}^{k} (d[v_{i-1}] + \sum_{i=1}^{k} \omega(v_{i-1}, v_i)) \end{aligned}$$

Pero

$$\sum_{i=1}^{k} d[v_i] = \sum_{i=1}^{k} d[v_{i-1}]$$

luego cada vértice en el ciclo c aparece exactamente una vez en cada sumatoria. Esto implica que

$$0 > \sum_{i=1}^{k} \omega(v_{i-1}, v_i)$$

Además, la suma de pesos alrededor de el ciclo c es negativa, por lo tanto se produce una contradicción.

Nosotros hemos probado que G_π es un grafo acíclico dirigido. Para mostrar que este forma un árbol enraizado con raiz s, es suficiente probar que para cada vértice $v \in V_\pi$, existe un único camino desde s hacia v en G_π.

Primero mostraremos que un camino desde s existe para cada vértice en V_π. Los vértices en V_π son aquellos con valores π no nulos, además de s.

Por inducción, si fuera solo un vértice sería la fuente s, caso trivial. Para n vértices por hipótesis de inducción, existe un V_π tal que existe un camino desde s a todos los vértices $v \in V_\pi$. Ahora sea $u \in V$, donde $(v, u) \in E$, para cualquier $v \in V_\pi$, entonces $\pi(u) = v$. Por la hipótesis de inducción, $\forall\ v \in V_\pi$ existe un camino desde s que lo acceda, por lo tanto al añadir la arista $(\pi(u), u)$ también formará parte de E_π, luego con el paso de relajación se debe cumplir que $d[v] + \omega(v, u) = d[u]$ y mantenerse, mostrando que existe un camino desde s a todos los vértices en V_π.

Para completar la prueba del lema, probaremos ahora que para cualquier vértice $v \in V_\pi$, existe a lo mucho un camino desde s hacia v en el grafo G_π. Supongamos de otra manera, que existen 2 caminos simples desde s hacia algún vértice v: p_1, el cual puede ser descompuesto en $s \rightsquigarrow u \rightsquigarrow x \rightarrow z \rightsquigarrow v$ y p_2, el cual puede ser descompuesto en $s \rightsquigarrow u \rightsquigarrow y \rightarrow z \rightsquigarrow v$, donde $x \neq y$

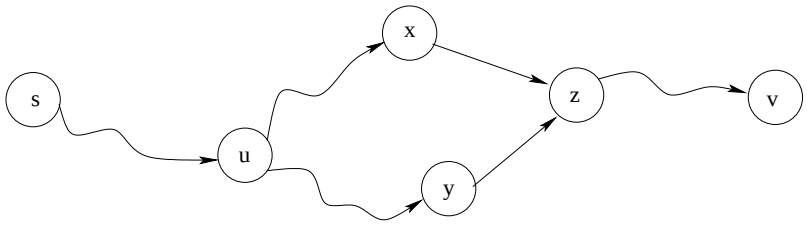


Figura 4.4: *Mostramos que el camino en G_π desde la fuente s hasta el vértice v es único. Si existen dos caminos $p_1(s \rightsquigarrow u \rightsquigarrow x \rightarrow z \rightsquigarrow v)$ y $p_2(s \rightsquigarrow u \rightsquigarrow y \rightarrow z \rightsquigarrow v)$, donde $x \neq y$, entonces $\pi[z] = x$ y $\pi[z] = y$, contradicción*

(ver figura 4.4). Pero entonces, $\pi[z] = x$ y $\pi[z] = y$ y $\pi[z] = y$, el cual implica la contradicción que $x = y$. Concluimos que existe un único camino simple en G_π desde s hacia v y además G_π forma un árbol enraizado con raiz s

Nosotros ahora mostraremos que luego de realizar una secuencia de pasos de relajación, todos los vértices han sido asignados a sus verdaderos caminos de pesos mínimo, entonces el subgrafo predecesor G_π es una árbol de camino mínimo.

Lema 4.7 *Sea $G = (V, E)$ un grafo dirigido con una función de pesos $\omega : E \rightarrow R$ y un vértice inicial(fuente) $s \in V$ y asumimos que G contiene ciclos de peso no negativos que sean accesibles desde s. Si ejecutamos además INICIALIZAR-FUENTE-SIMPLE(G,s) y se ejecuta cualquier secuencia de pasos de relajación sobre las aristas de G produciendo $d[v] = \delta(s, v)$ para todo $v \in V$. Entonces, el subgrafo predecesor G_π es un árbol de caminos mínimos de raiz s*

Demostración: Debemos probar que las 3 caracteristicas de árboles de caminos minimos se mantienen para G_π. Para mostrar la primera característica, debemos probar que V_π es un conjunto de vértices accesibles desde s. Por definición, un camino de pesos mínimo $\delta(s, v)$ es finito si y solo si v es accesible desde s y además los vértices que son accesibles desde s son exactamente aquellos con valores d finitos. Pero un vértice $v \in V - \{s\}$ se le ha asignado un valor finito por $d[v]$ si y solamente si $\pi[v] \neq nulo$. Así, los vértices en V_π son exactamente esos accesibles desde s

La segunda característica se prueba directamente del lema 4.6.

Para probar la última característica de árboles de caminos mínimos: Para todo $v \in V_\pi$, el único camino simple $s \overset{p}{\rightsquigarrow} v$ en G_π es un camino mínimo desde s hacia v en G. Sea $p = (v_0, v_1, ..., v_k)$, donde $v_0 = s$ y $v_k = v$. Para $i = 1, 2, ..., k$, nosotros tenemos que $d[v_i] = \delta(s, v_i)$ y $d[v_i] \geq d[v_{i-1}] + \omega(v_{i-1}, v_i)$, concluyendo entonces que $\omega(v_{i-1}, v_i) \leq \delta(s, v_i) - \delta(s, v_{i-1})$. Sumando todos los pesos a lo largo del camino p obtenemos

$$\begin{aligned} \omega(p) &= \sum_{i=1}^{k} w(v_{i-1}, v_i) \\ &\leq \sum_{i=1}^{k} (\delta(s, v_i) - \delta(s, v_{i-1})) \\ &= \delta(s, v_i) - \delta(s, v_0) \\ &= \delta(s, v_k) \end{aligned}$$

En la tercera linea aplicamos la suma telescópica sobre la segunda linea y en la última tenemos que $\delta(s, v_0) = \delta(s, s) = 0$. Asi, $\omega(p) \leq \delta(s, v_k)$. Luego $\delta(s, v_k)$ es la cota inferior sobre el

peso de cualquier camino desde s hacia v_k. Concluimos que $\omega(p) = \delta(s, v_k)$ y asi p es un camino mínimo desde s hacia $v = v_k$

4.5. Algoritmo de Dijkstra

El algoritmo de Dijskstra resuelve los problemas de caminos mínimos con pesos de destino simple en un grafo dirigido $G = (V, E)$ para el caso en el cual todas las aristas tienen pesos no negativos. Además se deberá asumir que $\omega(u, v) \geq 0$ para cada arista $(u, v) \in E$.

El algoritmo de Disjkstra mantiene un conjunto de vértices cuyo camino de pesos mínimos final desde el inicio(fuente) s se ha determinado. Esto es, para todos los vértices $v \in S$, donde $d[v] = \delta(s, v)$. El algoritmo selecciona repetidamente el vértice $u \in V - S$ con la menor estimación de camino mínimo, inserta u en S y relaja todas las aristas salientes de u. En la siguiente implementación , mantendremos la cola de prioridades Q que contiene todos los vértices en $V - S$, ingresado por sus valores d. La implementación asume que el grafo G es representado por una lista de adyacencia.

```
DIJKSTRA(G,w,s)\\
1.   INICIALIZAR-FUENTE-SIMPLE(G,s)
2.   S = vacio
3.   Q = V[G]
4.   Mientras Q <> vacio
5.        Hacer u = Extraer-min(Q)
6.           S= S U {u}
7.           Para cada vertice v en Adj[u]
8.              Hacer RELAJAR(u,v,w)
```

El algoritmo de Dijkstra relaja todas las aristas. En la linea 1, se realiza la inicialización respectiva de d y los valores π, en la linea 2 inicializa el conjunto S como el conjunto vacio. En la linea 3 luego de inicializar la cola de prioridad Q que contiene todos los vértices en $V - S = V - \emptyset = V$. En cada instante que se produzca el ciclo MIENTRAS desde la linea 4-8 un vértice u es extraido de $Q = V - S$ e insertado en el conjunto S (Al inicio del ciclo $u = s$). El vértice u, por lo tanto, tiene la más pequeña estimación de camino mínimo de cualquier vértice en $V - S$. Entonces, en las lineas 7-8 se relaja cada arista (u, v) que salen de u, así actualizando la estimación $d[v]$ y el predecesor $\pi[v]$ si el camino mínimo hacia v puede ser mejorado al ir através de u. Observemos que los vértices nunca son insertados en Q después de la linea 3 y que cada vértice es extraido de Q e insertado en S exactamente una vez. así que el ciclo MIENTRAS de las lineas 4-8 iteractua exactamente $\mid V \mid$ veces.

Con el siguiente teorema y su respectivo corolario mostraremos que el algoritmo de Dijkstra calcula caminos mínimos. La clave esta en mostrar que cada vértice u es insertado en el conjunto S y tener $d[u] = \delta(s, u)$

Teorema 4.1 *(Correción del algoritmo de Dijkstra) Si ejecutamos el algoritmo de Dijkstra sobre un grafo dirigido $G = (V, E)$ con una función de pesos no negativos ω y una fuente s, entonces al término, $d[u] = \delta(s, u)$ para todo vértice $u \in V$*

Demostración: Debemos probar que para cada vértice $u \in$ V, tenemos $d[u] = \delta(s, u)$ en el instante cuando u es insertado en el conjunto S y esta igualdad se mantiene después de todo.

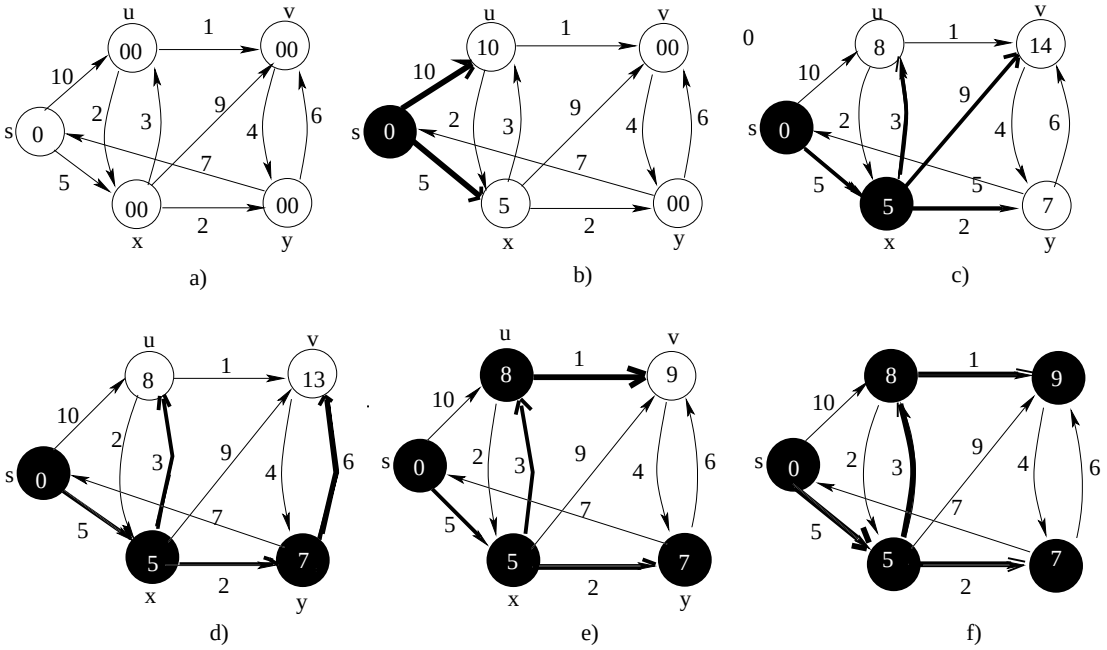


Figura 4.5: *Ejecución del algoritmo de Dijkstra. La fuente es el vértice s. El camino mínimo estimado se muestra dentro de los vértices y las aristas oscuras indican el valor del predecesor: Si* (u, v) *es oscura, entonces* $\pi[v] = u$*. Los vértices negros estan en el conjunto* S*, y los vertices blancos en la cola de prioridad* $Q = V - S$*. a) Muestra la situación antes de la primera iteracción del ciclo WHILE de 4-8. Los vertices oscuros tienen un valor mínimo de d y es escogido en la linea 5. b)-f) muestra la situación luego de cada iteracción recursiva del ciclo WHILE. La arista oscura en cada parte es escogido como el vértice u en la linea 5 de la proxima iteracción. Los valores de d y* π *muestran en f) los valores finales.*

Por contradicción, sea u el primer vértice para el cual $d[u] \neq \delta(s, u)$ cuando este es insertado en el conjunto S. Debemos enfocarnos en la situación del inicio de la iteracción del ciclo MIENTRAS, en el cual u es insertado en S y deriva la contradicción que $d[u] = \delta(s, u)$, en el momento de examinar un camino mínimo desde s hacia u. Nosotros debemos tener $u \neq s$ porque s es el primer vértice insertado en el conjunto S y $d[s] = \delta(s, s) = 0$ en este momento.

Como $u \neq s$, tenemos que $S \neq \emptyset$ solo antes que u sea insertado en S. Debe haber algún camino desde s hacia u, por otro lado $d[u] = \delta(s, u) = \infty$ por el corolario 4.2, el cual iría en contra de lo que hemos asumido que $d[u] \neq \delta(s, u)$. Como existe por lo menos un camino, existe un camino mínimo p desde s hacia u. El camino p conecta un vértice en S, llamemoslo s, con un vértice en $V - S$, llamemolo u. Consideremos el primer vértice y a lo largo de p tal que $y \in V - S$, y sea $x \in V$ el predecesor de y. Así, el camino p puede ser descompuesto por $s \overset{p_1}{\rightsquigarrow} x \rightarrow y \overset{p_2}{\rightsquigarrow} u$.

Afirmamos que $d[y] = \delta(s, y)$ cuando u es insertado en S. Para probar esta afirmación, observamos que $x \in$ S. Entonces, como u es escogido como el primer vértice para el cual $d[u] \neq \delta(s, u)$ cuando este es insertado en S, tenemos $d[x] = \delta(s, x)$ cuando x es insertado en S. La arista (x, y) fue relajada en este momento y se cumple la afirmación por el lema 4.5.

Podemos ahora obtener un contradicción al probar el teorema. Como y ocurre antes que u sobre un camino mínimo desde s hacia u y toda arista tiene peso no negativo (notables en la trayectoria p_2), tenemos $\delta(s, y) \leq \delta(s, u)$ y así:

$$\begin{aligned} d[y] &= \delta(s,y) \\ &\leq \delta(s,u) \\ &\leq d[u] \qquad \text{(Por el lema 4.4)} \end{aligned}$$

Pero como ambos vértices u e y estan en $V - S$ cuando u fue escogido en la linea 5, tenemos $d[u] \leq d[y]$. Así, las 2 desigualdades son en efecto iguales, dando:

$$d[y] = \delta(s,y) = \delta(s,u) = d[u]$$

Consecuentemente, $d[u] = \delta(s,u)$, el cual es una contradicción a nuestra elección de u. Concluimos que en el momento que cada vértice $u \in V$ es insertado en el conjunto S, tenemos $d[u] = \delta(s,u)$ y por el lema 4.4, esta igualdad se mantiene después de eso.

Corolario 4.3 *Si ejecutamos el algoritmo de Dijkstra sobre un grafo $G = (V,E)$ con una función ω de pesos no negativos y una fuente s, entonces al término, el subgrafo predecesor G_π es un árbol de caminos mínimos de raiz s*

Demostración: Por el teorema 4.1 y el lema 4.7 se cumple.

4.5.1. Análisis de la complejidad del Algoritmo de Dijkstra

Considerando el primer caso en el cual mantenemos la cola de prioridad $Q = V - S$ como un arreglo lineal. Para cada implementación, cada operación EXTRAER-MIN tomaría un tiempo del orden $O(V)$ y son $\mid V \mid$ operaciones , para un total del tiempo de EXTRAER-MIN de $O(V^2)$. Cada vertice $v \in V$ es insertado en el conjunto S exactamente una vez, así cada arista en la lista de adyacencia Adj[u] es examinada por el ciclo PARA en las lineas 4-8 exactamente una vez durante el curso del algoritmo. Luego el total del número de aristas en toda la lista de adyacencia es $\mid E \mid$, existe un total de $\mid E \mid$ iteraciones del ciclo PARA y en cada iteracción se toma un tiempo $O(1)$. El tiempo de ejecución del algoritmo completo es $O(V^2 + E) = O(V^2)$.

A continuación presentaremos otras formas de implementación.

Implementación con Heap Binario

Si el grafo es escaso, es práctico para implementar la cola de prioridades Q con un *heap binario*. El algoritmo que resulta es algunas veces llamado **algoritmo de Dijkstra modificado**. Cada operación de EXTRAER-MIN entonces toma un tiempo de $O(lgV)$. Como antes, existe $\mid V \mid$ operaciones. El tiempo para construir el heap binario es $O(V)$. La asignación $d[v] \leftarrow d[u] + \omega(u,v)$ en RELAJAR es realizado por el llamado DECRECER-CLAVE $(Q, v, d[u] + \omega(u,v))$, el cual toma un tiempo de $O(lgV)$ y existen aun a lo mucho $\mid E \mid$ operaciones. El tiempo total de ejecución es por lo tanto $O((V+E)lgV)$, el cual es $O(E.lgV)$ si todos los vértices son accesibles desde la fuente.

Implementación con Heap de Fibonacci

Por implementación de la cola de prioridad Q con un heap de Fibonacci, nosotros alcanzamos un tiempo de ejecución de $O(VlgV + E)$. El costo de amortización de cada una de las $\mid V \mid$ operaciones es $O(lgV)$ y cada de los $\mid E \mid$ llamados en DECRECER-CLAVE toma solo $O(1)$ el tiempo de amortización.

Implementación con d-Heap

De acuerdo a lo ya mencionado en la sección 3.3, en la versión del algoritmo de Dijkstra, consecuentemente el tiempo de ejecución es $O(m.log_d n + n.d.log_d n)$. Para obtener un elección óptima de d se comparan dos términos, tal que $d = max\{2, [m/n]\}$. El resultado del tiempo de ejecución es $O(m.log_d n)$. Para el caso de redes poco densas (es decir, $O(n)$), el tiempo de ejecución del algoritmo será del orden $O(m.log(n))$. Para el caso de redes densas (es decir, $m = \Omega(n^{1+\epsilon})$, para algún $\epsilon > 0$), el tiempo de ejecución del la implementación d-Heap es:

$$O(mlog_d n) = O\left(\frac{mlog(n)}{log(d)}\right) = O\left(\frac{mlog(n)}{log(n^\epsilon)}\right) = O\left(\frac{mlog(n)}{\epsilon log(n)}\right) = O\left(\frac{m}{\epsilon}\right) = O(m)$$

La última igualdad es verdadera cuando ϵ es una constante. Así el tiempo de ejecución es $O(m)$, el cual es óptimo

Implementación Circular

Este algoritmo tiene como propiedad principal el hecho de que los valores de las distancias que el algoritmo de Dijkstra va designando como permanente no decrecen. El algoritmo permanentemente da valores a un nodo i con el menor valor temporal $d(i)$ mientras recorre las aristas en $A(i)$ durante las operaciones de actualización y nunca decrece el valor de la distancia de cualquier nodo de valor temporal por debajo de $d(i)$, ya que la longitud de las aristas son no negativas. El algoritmo Circular almacena nodos con una valor temporal finito de manera ordenada. Esto mantiene $nC + 1$ conjuntos, llamados *arreglos*, numerados desde 0, 1, 2, ..., nC. Un arreglo k almacena todos los nodos con valor de distancia temporal igual a k. Recordar que C representa la arista de mayor longitud en la red y por lo tanto nC es una cota superior sobre los valores de distancia de cualquier nodo de valor finito. Esta estructura nos permite realizar las operaciones como revisar si el arreglo esta vacio o no vacio y eliminar o añadir un elemento del arreglo con un tiempo del orden $O(1)$. Así la estructura toma un tiempo del orden $O(1)$ para cada actualización de la distancia y un tiempo total del orden $O(m)$ para todas las actualizaciones. El operación más extensa en esta implementación es revisar $nC + 1$ arreglos durante las selecciones del nodo. En consecuencia, el tiempo de ejecución del Algoritmo Circular es del orden $O(m + nC)$

Implementación con Heap de Base

La implementación con heap de Base es un híbrido de la implementación original del algoritmo de Dijkstra con la implementación circular(ya que usa $nC + 1$ arreglos). La implementación original considera todos los valores temporales de los nodos juntos(en un solo arreglo) y la búsqueda del nodo con su menor valor. El algoritmo Circular usa un número de arreglos grandes y separa los nodos almacenando dos de ellos cualquiera con diferentes valores en arreglos diferentes. La implementación con heap de Base, mejora estos dos métodos adoptando un enfoque intermedio: Almacenando muchos, pero no todos, los valores en los arreglos. Para esta versión uno debe tener presente dos condiciones:

1. El tamaño de los arreglos son 1, 1, 2, 4, 8, 16, ..., tal que son los números de arreglos necesarios es solo del orden $O(log(nC))$

2. Dinámicamente modifica los rangos de los arreglos y reubica los nodos con valores de distancia temporal de tal manera, que almacena el valor temporal mínimo en una arreglo el cual posee un ancho 1.

La propiedad 1 nos permite mantener solo arreglos del orden $O(log(nC))$ y por lo tanto supera el inconveniente de la implementación circular que usa demasiados arreglos. La propiedad número 2 nos permite, como en el algoritmo circular, evitar la necesidad de buscar en todo el arreglo un nodo con el menor valor de distancia. Cuando se implementa de esta manera, el algoritmo de heap de base tiene un tiempo de ejecución de $O(m + nlog(nC))$

4.6. Algoritmo de Bellman-Ford

El algoritmo de Bellman-Ford resuelve problemas de caminos mínimos con una fuente simple y en un caso general con arista que pueden tener pesos negativos. Dado un grafo $G = (V, E)$ con una fuente s y una función de peso $\omega : E \rightarrow R$, el algoritmo de Bellman-Ford retorna valores booleanos indicando si existe o no un ciclo de pesos negativos que es accesible desde una fuente. Si existe tal ciclo, el algoritmo indica que no existe solución. Si no existiera el ciclo, el algoritmo produce caminos mínimos y sus pesos.

Como el algoritmo de Dijkstra, el algoritmo de Bellman-Ford usa la técnica de relajación, decreciendo progresivamente al estimar $d[v]$ sobre un peso de un camino mínimo desde una fuente s a cada vértice $v \in V$ hasta el actual camino mínimo de peso $\delta(s, v)$. El algoritmo retorna VERDADERO si y solo si el grafo contiene un ciclo de pesos no negativos que son accesible desde una fuente.

```
BELLMAN-FORD(G,w,s)
1.   INICIALIZAR-FUENTE-SIMPLE(G,s)
2.   Para i=1 hasta |V[G]|-1
3.       Hacer para cada arista (u,v)en E[G]
4.            Hacer RELAJAR(u,v,w)
5.   Para cada arista (u,v)en E[G]
6.        Hacer si d[v]>d[u]+w(u,v)
7.          Entonces retornar FALSO
8.   retornar VERDADERO
```

Observando la ejecución del algoritmo de Bellman-Ford. Luego de la inicialización, el algoritmo ha realizado $\mid V \mid -1$ pasadas sobre las aristas del grafo. Cada pasada es una iteracción del ciclo PARA de las lineas 2-4 y consiste en relajar cada arista del grafo. Luego de hacer $\mid V \mid -1$ pasadas, en las lineas 5-8 revisamos para cada ciclo de pesos negativos y retornamos el apropiado resultado booleano.

El algoritmo de Bellman-Ford se ejecuta en un tiempo $O(V.E)$, luego la inicialización de la linea 1 toma un tiempo $\Theta(V)$, cada uno de los $\mid V \mid -1$ pasos sobre las aristas en las lineas 2-4 toma un tiempo de $O(E)$ y para el ciclo PARA de las lineas 5-7 toma un tiempo de $O(E)$.

Para probar la corrección del algoritmo de Bellman-Ford, empezaremos por mostrar que si existe un ciclo de pesos no negativos, el algoritmo calcula un camino de pesos mínimo correcto para todos los vértices accesibles desde la fuente.

Lema 4.8 *Sea $G = (V, E)$ un grafo dirigido con fuente s y una función de peso $\omega : E \rightarrow R$ y asumimos que G contiene ciclos de pesos no negativos que son accesibles desde s. Entonces, al terminar el algoritmo de BELLMAN-FORD, tendremos que $d[v] = \delta(s, v)$ para todos los vértices v que son accesibles desde s*

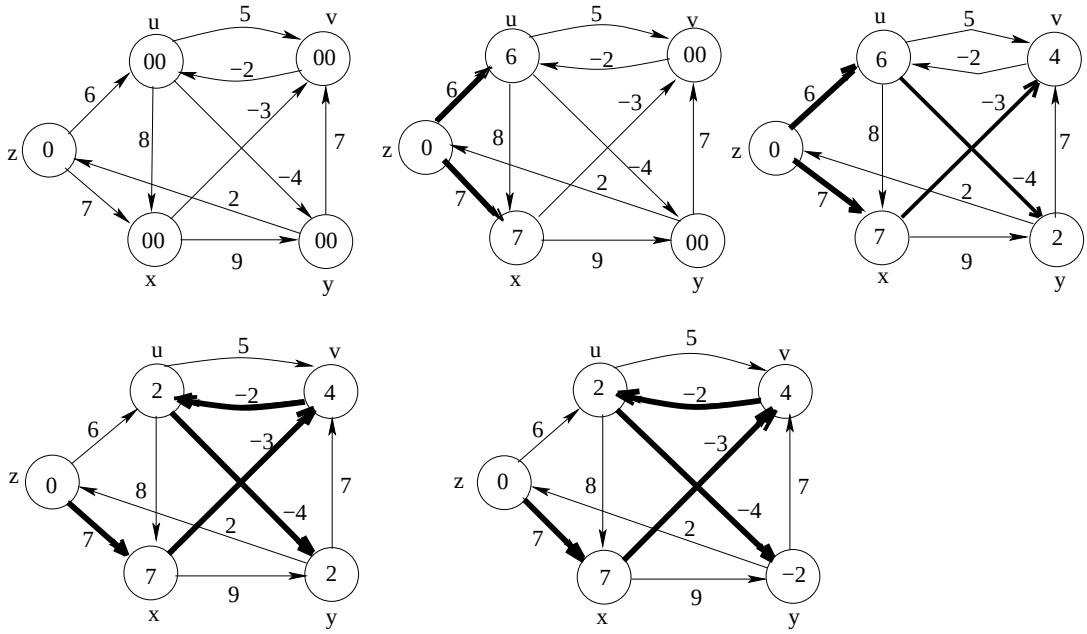


Figura 4.6: *Ejecución del algoritmo de Bellman-Ford. La fuente es el vértice z. Los valores d se muestran en el interior del vértice y las aristas oscuras indican los valores de π. En este ejemplo en particular, cada paso de relajación de las arista el orden alfabético es:*$(u, v), (u, x), (u, y), (v, u), (x, v), (x, y), (y, v), (y, z), (z, u), (z, x)$ *a) Muestra la situación antes del primer paso de relajación. b)-e) Muestra la situación luego de cada paso de relajación sobre las aristas. e) Se muestran los valores de d y π. En este caso el algoritmo retorna VERDADERO*

Demostración: Sea v un vértices accesibles desde s y sea $p = (v_0, v_1, ..., v_k)$ un camino mínimo desde s hacia v donde $v_0 = s$. El camino p es simple y tambien $k \leq \mid V \mid -1$. Queremos probar por inducción que para $i = 0, 1, ...k$, tendremos $d[v_i] = \delta(s, v_i)$, luego del paso i-esimo sobre las aristas de G y la igualdad se mantendrá. Como existe $\mid V \mid -1$ pasos, esto es una afirmación suficiente para probar el lema.

De lo básico, tenemos $d[v_0] = \delta(s, v_0) = 0$, luego de la inicialización y por el lema 4.4 esta igualdad se mantiene.

Por el paso inductivo, asumimos que $d[v_{i-1}] = \delta(s, v_{i-1})$, luego (i-1)-esimo paso. La arista (v_{i-1}, v_i) es relajada en el i-esimo paso por el lema 4.5. Concluimos entonces que $d[v_i] = \delta(s, v_i)$ luego del i-esimo paso y toda la subsecuencia de veces.

Corolario 4.4 *Sea $G = (V, E)$ un grafo dirigido con un vértice s fuente y una función de peso $\omega : E \to R$. Entonces para cada vértice $v \in V$, existe un camino desde s hacia v si y solo si BELLMAN-FORD termina con $d[v] < \infty$ cuando este se ejecuta sobre G.*

Demostración: ($\Rightarrow$) Sea un camino p de s hacia v de la forma $p =< v_0, v_1, ..., v_k >$, donde $v_0 = s$ y $v_k = v$ y además $k \leq \mid V \mid -1$. Por inducción buscamos probar que $d[v_i] = \delta(s, v_i) < \infty$. En el caso trivial, cuando $d[v_0] = d[s] = \delta(s, s) = 0$ esto es producido luego de inicialización.

Ahora por inducción $d[v_{i-1}] = \delta(s, v_{i-1}) < \infty$ ya que existe un camino accesible de s hacia v, si deseamos relajar la siguiente arista (v_{i-1}, v_i) por el lema 4.5 se relaja y mantiene invariante. Por lo tanto $d[v_i] = \delta(s, v_i) = d[v] < \infty$.

($\Leftarrow$) Si $d[v] < \infty$, entonces $d[v] = \delta(s, v) < \infty$ al finalizar el algoritmo de Bellman Ford tendremos entonces que $d[v] = \delta(s, v)$ para todos los vértices v que son accesibles desde s

Teorema 4.2 *(***Corrección del algoritmo de Bellman-Ford***) Sea el algoritmo de BELLMAN-FORD ejecutado sobre un grafo dirigido $G = (V, E)$ con una fuente s y una función de peso $w : E \to R$. Si G contiene un ciclo de pesos no negativos que son accesibles desde s, entonces el algoritmo retorna VERDADERO, tendremos $d[v] = \delta(s, v)$ para todo vértice $v \in V$ y el subgrafo predecesor G_π es un árbol de caminos mínimos enraizados a s. Si G contiene un ciclo de pesos negativos accesibles desde s, entonces el algoritmo retornará FALSO.*

Demostración Supongamos que el grafo G contiene ciclos de pesos no negativos que son accesibles desde una fuente s. Primero probaremos la afirmación que al terminar $d[v] = \delta(s, v)$ para todos los vértices $v \in V$. Si el vértice v es accesible desde s, entonces el lema 4.8 prueba esta afirmación. Si v no es accesible desde s, entonces la afirmación se desprende del corolario 4.2. Así probamos la afirmación. El lema 4.7, utilizada en la afirmación, implica que G_π es un árbol de caminos mínimo. Ahora usamos la afirmación para mostrar que BELLMAN-FORD retorna VERDADERO. Al terminar, tenemos que todas las aristas $(u, v) \in E$.

$$\begin{aligned} d[v] &= \delta(s, v) \\ &\leq \delta(s, u) + \omega(u, v) \quad \text{(por lema 4.2)} \\ &= d[u] + \omega(u, v) \end{aligned}$$

y así para ninguna se realizará la linea 6 porque BELLMAN-FORD retornará FALSO.
Esto por lo tanto retornará VERDADERO

Inversamente, supongamos que el grafo G contiene un ciclo de peso negativo $c = (v_0, v_1, ..., v_k)$ donde $v_0 = v_k$ que es accesible desde una fuente s. Entonces:

$$\sum_{i=1}^{k} \omega(v_{i-1}, v_i) < 0$$

Asumamos para la contradicción que el algoritmo de Bellman-Ford retorna VERDADERO. Además , $d[v_i] \leq d[v_{i-1}] + \omega(v_{i-1}, v_i)$ para $i = 1, 2, ..., k$. Sumando las desigualdades alrededor del ciclo c nos da:

$$\sum_{i=1}^{k} d[v_i] \leq \sum_{i=1}^{k} d[v_{i-1}] + \sum_{i=1}^{k} \omega(v_{i-1}, v_i)$$

Como en la prueba del lema 4.6, cada vértice en c aparece exactamente una vez en cada una de las primeras dos adiciones. Así,

$$\sum_{i=1}^{k} d[v_i] = \sum_{i=1}^{k} d[v_{i-1}]$$

Por otro lado, por el corolario 4.4, $d[v_i]$ es finito para $i = 1, 2, ..., k$. Así,

$$0 \leq \sum_{i=1}^{k} \omega(v_{i-1}, v_i)$$

con lo cual contradice la desigualdad inicial. Concluimos que el algoritmo de Bellman-Ford retornará VERDADERO si el grafo G contiene ciclos de pesos no negativos accesibles desde una fuente, y FALSO si sucede lo contrario.

4.7. Caminos mínimos desde una fuente simple en grafos dirigidos acíclicos

Por la relajación las aristas de un Grafo dirigido acíclico (DAG) $G = (V, E)$ según la clase topológica de sus vértices, podemos calcular los caminos mínimos desde una fuente simple en el tiempo $\Theta(V + E)$. Los caminos mínimos estan siempre bien definidos en un DAG, aun si existen aristas de pesos negativos, pudiendo existir ciclos de pesos no negativos.

El algoritmo empieza aplicando la clase topológica al DAG, imponiendo un orden lineal sobre los vértices. Si existe un camino desde el vértice u a el vértice v, entonces u precede a v en la clase topológica. Nosotros solo pasaremos una vez sobre el vértice en el orden de la clase topológica. Como el vertice es procesado, todos las aristas que salen del vértice son relajados.

```
DAG-CAMINOS MINIMOS (G,w,s)
1.    Clasificar topologicamente los vertices de G
2.    INICIALIZAR-FUENTE-SIMPLE (G,s)
3.    Para cada vertice u tomar en el orden de la clasificacion topologica
4.        HACER PARA cada vertice v en Adj[u]
5.           HACER RELAJAR (u,v,w)
```

El tiempo de ejecución del algoritmo es determinado en la linea1 por el ciclo PARA de las lineas 3-5. La clase topológica puede ser realizado en el tiempo $\Theta(V+E)$. En el ciclo PARA de las lineas 3-5,como en el algoritmo de Dijkstra, existe una iteracción por vértice. Para cada vértice, las aristas que salen del vértice son examinadas exactamente una vez. A diferencia del algoritmo del Dijkstra, nosotros usamos solo un tiempo de $O(1)$ por arista. El tiempo de ejecución es asi $\Theta(V + E)$, el cual es lineal en el tamaño de la representación en una lista de adyacencia de un grafo.

El siguiente Teorema muestra que los caminos mínimos de un DAG producen correctamente cálculos de caminos mínimos

Teorema 4.3 *Si un grafo dirigido $G = (V, E)$ tiene un vértice fuente s y es no cíclico, entonces en la finalización de los* Caminos mínimos de un DAG *produce, $d[v] = \delta(s, v)$ para todo vértice $v \in V$ y el subgrafo predecesor G_π es un árbol de caminos mínimos.*

Demostración Primero mostraremos que $d[v] = \delta(s, v)$ para todos los vértices $v \in V$ al finalizar. Si v no es accesible desde s entonces $d[v] = \delta(s, v) = \infty$ por el corolario 4.2. Ahora, supongamos que v es accesible desde s, asi que existe un camino mínimo $p = (v_0, v_1, ..., v_k)$, donde $v_0 = s$ y $v_k = v$. Como nosotros procesamos los vértices en el orden de clasificacion topologica, las aristas sobre p son relajadas en el orden $(v_0, v_1), (v_1, v_2), ..., (v_{k-1}, v_k)$. Una inducción simple y usando el lema 4.5 muestra que $d[v_i] = \delta(s, v_i)$ en la finalización para $i = 0, 1, ..., k$. Finalmente por el lema 4.7, G_π es una árbol de caminos mínimos.

Capítulo 5

Aplicaciones

Los problemas de caminos mínimos han surgido en una amplia variedad de problemas de ajustes prácticos, ya sea como problemas autónomos o como subproblemas de problemas más complejos. Sea por ejemplo: en el caso de las telecomunicaciones, de industrias de transporte, de cuando se desea enviar algún mensaje o en el caso de que un vehículo se encuentre entre dos localizaciones geográficas y busque hallar cual será el recorrido más corto y más barato posible. Existen además modelos de planificación urbana que se usan para calcular los patrones del flujo del tráfico, que son problemas de optimización no lineal complejos o modelos de equilibrio complejos; estos se construyen en base a asumir un comportamiento de los usuarios en el sistema de transporte, con respecto a la congestión vehicular a lo largo de un camino, desde su origen hasta su destino. Veamos a continuación algunas aplicaciones que incluyen aplicaciones genéricas de matemática como funciones de aproximación o resolución de ciertos tipos de ecuaciones diferenciales.

5.1. Aplicaciónes del Algoritmo de Bellman-Ford

5.1.1. El Problema del Barco Mercante

Un barco mercante viaja de puerto en puerto llevando carga y pasajeros. En un viaje de un barco mercante desde un puerto i hasta un puerto j se obtienen p_{ij} unidades en beneficios y requieren τ_{ij} unidades de tiempo. El capitan del barco mercante podria conocer cual es el recorrido W del barco (es decir el ciclo dirigido) que alcanze el mayor beneficio promedio diario posible. Nosotros definimos el beneficio diario de cualquier recorrido W por la siguiente expresión:

$$\mu(W) = \frac{\sum_{(i,j)\in W} p_{ij}}{\sum_{(i,j)\in W} \tau_{ij}}$$

Asumimos que $\tau_{ij} \geq 0$ para cada arista $(i,j) \in A$, y que $\sum_{(i,j)\in W} \tau_{ij} \geq 0$ para todo ciclo dirigido W en la red.

Aunque esta aplicación necesita de un análisis más profundo en el estudio de ciclos negativos, la cual no se esta desarrollando en el presente trabajo, podemos examinar las restricciones que este problema presenta. El capitan del barco desea conocer si algun recorrido W sera capaz de alcanzar un promedio de beneficios diarios superior al de una determinada cota μ_0. Nosotros mostraremos como formular este problema como un problema de detección de ciclos negativos.

En esta versión restringuida del problema del barco a vapor, deseamos determinar la red fundamental G que contenga un ciclo dirigido W que satifaga la siguiente condicion

$$\frac{\sum_{(i,j)\in W} p_{ij}}{\sum_{(i,j)\in W} \tau_{ij}} > \mu_0$$

Escribiendo esta desigualdad como $\sum_{(i,j)\in W}(\mu_0\tau_{ij} - pij) < 0$. Vemos que G contiene un ciclo dirigido W en G cuyo beneficio promedio excede $\mu - 0$ si y solo si la red contiene un ciclo negativo cuando el costo de la arista (i, j) es $(\mu_0\tau_{ij} - pij)$.

5.1.2. Diferencias Restringuidas y Caminos Mínimos: Programación Lineal

En general en los problemas de programación lineal, nosotros deseamos optimizar una funcion tema para un conjunto de inecuaciones lineales. En este sección nosotros investigaremos un caso especial de programación lineal que puede ser reducido para encontrar caminos mínimos desde una fuente simple. El problema de caminos mínimos con fuente simple que resulta puede entonces resolverse usando el algoritmo de Bellman-Ford, así también resuelve el problema de programación lineal.

En general en el problema de programación lineal, nosotros damos una matriz A de $m \times n$ y un *m-vector* b, y un *n-vector* c. Se desea encontrar un vector x de n elementos que maximize un función objetivo $\sum_{i=1}^{n} c_i x_i$ tema para las m restricciones dadas por $Ax \leq b$

Como no profundizaremos en los algoritmos de programación lineal debemos tomar en cuenta; primero que dado un problema puede ser moldeado como un problema de programación lineal de tamaño polinomial lo cual significa que este algoritmo es de tiempo polinomial. Segundo, existen muchos casos especiales de programación lineal para el cual existen algoritmos más rápidos. Por ejemplo, el problema de caminos mínimos con fuente simple es un caso especial de la programación lineal.

Algunas veces nosotros no tenemos real cuidado acerca de la función objetivo; ya que solo deseamos encontrar una *solución factible*, esto es, cualquier vector x que satisface $Ax \leq b$ o determinar que no existe una solución factible. Nosotros nos enfocaremos en uno tal que sea un *problema de factibilidad*

Sistemas de diferencias restringuidas

En los sistemas de diferencias restringuida, cada fila de la matriz A en la programación lineal contiene 1 o -1, y todos los demás serán 0. Asi, las restricciones dadas por $Ax \leq b$ son un conjunto de m *diferencias restringuidas* que envuelven n incógnitas, en las cuales cada restricción es una simple desigualdad de la forma:

$$x_j - x_i \leq b_k \qquad \text{donde } 1 \leq i,\, j \leq n \text{ y } 1 \leq k \leq m.$$

Por ejemplo, consideremos el problema de encontrar un 5-vector $x = (x_i)$ que satisface:

$$\begin{pmatrix} 1 & -1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 & -1 \\ -1 & 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & -1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} \leq \begin{pmatrix} 0 \\ -1 \\ 1 \\ 5 \\ 4 \\ -1 \\ -3 \\ -3 \end{pmatrix}$$

Este problema es equivalente a encontrar las x_i incognitas, para i=1,2,...,5 tal que las siguientes 8 diferencias restringuidas son satisfechas:

$$\begin{aligned} x_1 - x_2 &\leq 0 \\ x_1 - x_5 &\leq -1 \\ x_2 - x_5 &\leq 1 \\ x_3 - x_1 &\leq 5 \\ x_4 - x_1 &\leq 4 \\ x_4 - x_3 &\leq -1 \\ x_5 - x_3 &\leq -3 \\ x_5 - x_4 &\leq -3 \end{aligned}$$

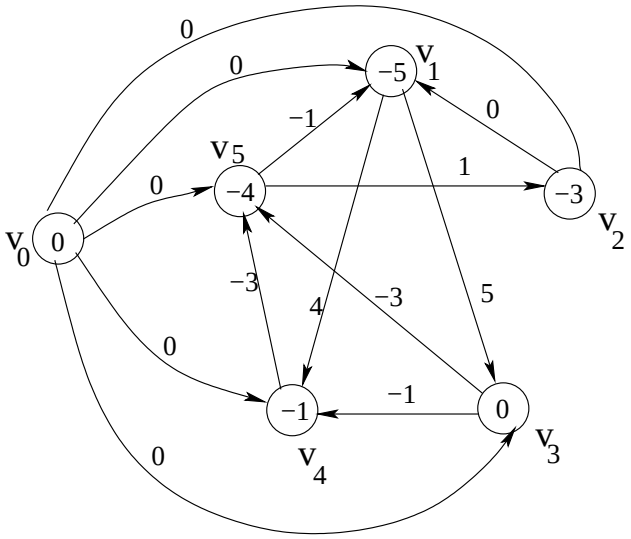


Figura 5.1: *Se muestras el grafo con las restricciones correspondientes al sistema de diferencias restringuidas dado. Los valores de $\delta(v_0, v_i)$ es mostrado en cada vértices v_i. Dando una solución factible $x = (-5, -3, 0, -1, -4)$*

Una solución de este problema es $x = (-5, -3, 0, -1, -4)$ la cual se puede verificar en cada desigualdad. En efecto, existe más de una solución para este problema. Otro es $x' = (0, 2, 5, 4, 1)$. Existen 2 soluciones en realidad: cada componente de x' es 5 veces más grande que el correspondiente de x y esto no es una simple coincidencia.

Lema 5.1 *Sea $x = (x_1, x_2, ..., x_n)$ sea una solución para el sistema $Ax \leq b$ de diferencias restringuidas y sea d cualquier constante. Entonces $x + d = (x_1 + d, x_2 + d, ..., x_n + d)$ es una solución bien definida para $Ax \leq b$*

Demostración Para cada x_i y x_j, nosotros tenemos que $(x_j + d) - (x_i + d) = x_j - x_i$. Así, si x satisface $Ax \leq b$, también $x + d$

En los sistemas de diferencias restringuidas ocurren muchas aplicacioens diferentes. Por ejemplo, las incógnitas x_i pueden ser el tiempo en el cual los eventos ocurren. Cada restricción puede ser vista como el inicio de un evento que no puede ocurrir antes que otro evento. Quizas los eventos son trabajos a realizarse durante la construcción de una casa. Si la excavación de la foza empieza en el tiempo x_i y toma 3 dias y la preparación de concreto para la foza empieza en el tiempo x_2, podria decirse que $x_2 \geq x_1 + 3$ o equivalente a $x_1 - x_2 \leq 3$. Así, la restricción del tiempo relativo puede ser expresado como diferencias restringuidas.

Grafos Restringuidos

Veamos como podemos interpretar un sistema de diferencias restringuidas desde el punto de vista de Teoría de Grafos. La idea es que un sistema $Ax \leq b$ de diferencias restringuidas, la matriz $n \times m$ de programación lineal puede ser vista como la matriz de incidencia para un grafo de n vértices y m aristas. Cada vértices v_i en el grafo para $i = 1, 2, ..., n$ corresponde a cada una de las n variables desconocidas x_i. Cada arista dirigida en el grafo corresponde a una de las m desigualdades envuelviendo 2 incógnitas.

Formalmente, Dado un sistema $Ax \leq b$ de diferencias restringuidas, el correspondiente grafo restringuido es un grafo dirigido con pesos $G = (V, E)$ donde:

$V = \{v_0, v_1, v_n\}$

y

$E = \{(v_i, v_j) : x_j - x_i \leq b_k \text{ es una restricción } \} \cup \{(v_0, v_1), (v_0, v_2)(v_0, v_3), ..., (v_0, v_n)\}$

El vértice adicional v_0 es incorporado, como veremos pronto , para garantizar que todos los vértices sean accesibles desde este. Asi, el conjunto de vértices V consiste de un vértice v_i para cada incógnita x_i, más un vértice adicional v_0. El conjunto de aristas E contiene una arista para cada diferencia restringuida, mas una arista $(v_0.v_i)$ para cada incógnita x_i. Si $x_j - x_i \leq b_k$ es una diferencia restringuida, entonces el peso de la arista (v_i, v_1) es $\omega(v_i, v_j) = b_k$. El peso de cada arista que sale de v_0 es 0.

En el siguiente teorema mostraremos que la solución del sistema de diferencias restringuidas puede ser obtenido al encontrar un camino de pesos mínimos en el correspondiente grafo restringuido

Teorema 5.1 *Dado un sistema $Ax \leq b$ de diferencias restringuidas, sea $G = (V, E)$ su correspondiente grafo. Si G contiene ciclos de pesos no negativos, entonces*

$$x = (\delta(v_0, v_1), \delta(v_0, v_2), \delta(v_0, v_3), ..., \delta(v_0, v_n)) \tag{5.1}$$

es una solución factible para el sistema. Si G contiene ciclos de pesos negativos entonces no existe una solución factibles para el sistema.

Demostración Primero mostraremos que si el grafo restringuido contiene ciclos de pesos no negativo, entonces la ecuación (5.1) da soluciones factibles. Consideremos la arista $(v_i, v_j) \in E$. Por el lema 3.2 ,$\delta(v_0, v_j) \leq \delta(v_0, v_i + \omega(v_i, v_j)$ or equivalentemente $\delta(v_0, v_j) - \delta(v_0, v_i \leq \omega(v_i, v_j)$. Así, siendo $x_i = \delta(v_0, v_i)$ y $x_j = \delta(v_0, v_j)$ satisface la diferencia restringuida $x_j - x_i \leq \omega(v_i, v_j)$ que corresponde a la arista (v_i, v_j)

Ahora mostraremos que si el grafo restringuido contiene un ciclo de pesos negativos, entonces el sistema de diferencias restringuidas no tiene solución factible. Sin perdida de generalidades, sea un ciclo de peso negativo $c = (v_1, v_2, ..., v_n)$ donde $v_1 = v_k$ (El vértices v_0 no puede estar sobre el ciclo c, porque este no tiene aristas enteras). El ciclo c corresponde a las siguientes diferencias restringuidas:

$$\begin{array}{rcl} x_2 - x_1 & \leq & \omega(v_1, v_2) \\ x_3 - x_2 & \leq & \omega(v_2, v_3) \\ & \vdots & \\ x_k - x_{k-1} & \leq & \omega(v_{k-1}, v_k) \\ x_1 - x_k & \leq & \omega(v_k, v_1) \end{array}$$

Como cualquier solución para x debe satisfacer cada una de esas k desigualdades, cualquier solución debe tambien satisfacer la desigualdad que resulta cuando sumamos todos juntos. Si sumamos el lado izquierdo, cada incognita x_i es añadida una vez y sustraida fuera una vez, asi que el lado izquierdo de la suma nos dara 0. El lado derecho de la suma da $\omega(c)$ y asi obtenemos $0 \leq \omega(c)$. Pero como c es un ciclo de peso negativo, $\omega(c) < 0$ y por lo tanto cualquier solución para x debe satisfacer $0 \leq \omega(c) < 0$, lo cual es imposible.

Solución de sistemas de diferencias restringuidas

En el teorema 5.1 nosotros usamos el algoritmo de Bellman-Ford para resolver el sistema de diferencias restringuidas. Por que existe aristas desde el vértice fuente v_0 a todos los otros vértices en el grafo restringuido, cualquier ciclo de peso negativo en el grafo restringuido es accesible desde v_0. Si el algoritmo de Bellman-Ford retorna VERDAD, entonces el camino de pesos mínimo dara una solución factible para el sistema. Si retorna FALSO, existirá una solución no factible para el sistema de diferencias restringuidas.

Un sistema de diferencias restringuidas con m restricciones y n incógnitas produce un grafo con $n + 1$ vértices y $n + m$ aristas. Así, usando el algoritmo de Bellman-Ford, resolvemos el sistema en el tiempo de $O((n + 1)(n + m)) = O(n^2 + nm)$.

5.1.3. El Horario del Operador Telefónico

Como una aplicación de las diferencias restringidas, se puede considerar el problema del operador telefónico. Una compañia telefónica necesita el horario de los operadores durante el día. Sea $b(i)$ para i= 0, 1, 2, ..., 23, denotando el número mínimo de operadores necesarios desde la i-ésima hora del día (aquí $b(0)$ denota el número de operadores requeridos entre la media noche y la 1 a.m.). Cada operador telefónico trabaja en un turno de 8 horas consecutivas y el turno se inicia en cualquier hora del día. La compañía telefónica quiere determinar un çiclo de horarios"que repetir diariamente(es decir, el número de operadores asignados para iniciar el turno desde las 6 a.m. hasta finalizar a las 2 p.m. sea el mismo para cada día). El problema de optimización requiere identificar el menor número de operadores necesarios para satisfacer

el requerimiento mínimo de estos para cada hora del día. Sea y_i el número de trabajadores los cuales inician turno en la i-ésima hora, nosotros podemos formular el problema del horario del operador telefónico como el siguiente modelo de optimización

$$\text{Minimizar} \quad \sum_{i=0}^{23} y_i \tag{5.2}$$

sujeto a

$$\begin{aligned} y_{i-7} + y_{i-7} + y_{i-7} ... + y_{i-7} &\geq b(i) \quad \text{para todo i =8 al 23} && (5.3)\\ y_{17+i} + ... + y_{23} + y_0 + ... + y_i &\geq b(i) \quad \text{para todo i =0 al 7} && (5.4)\\ y_i &\geq 0 \quad \text{para todo i=0 al 23} && (5.5) \end{aligned}$$

Notamos que la programación lineal tiene una estructura especial porque la matriz asociada restringida contiene solo 0 y 1 como elementos y el 1 de cada fila aparece consecutivamente. En este aplicación estudiaremos la versión restringida del problema de horario del operador telefónico: Deseamos determinar si algún posible horario usa p o menos operadores. Convertimos este problema de restricción en un sistema de diferencias restringidas redefiniendo las varibles. Sea $x(0) = y_0$, $x(1) = y_0 + y_1$, $x(2) = y_0 + y_1 + y_2$, ..., y $x(23) = y_0 + y_2 + ... + y_{23} = p$. Notamos que reescribiendo cada restricción en (5.3) como

$$x(i) - x(i-8) \geq b(i) \quad \text{para todo i=8 al 23} \tag{5.6}$$

y cada restricción (5.4) como

$$x(23) - x(16+i) + x(i) = p - x(16+i) + x(i) \geq b(i) \quad \text{para todo i=0 al 7} \tag{5.7}$$

Finalmente, la restricción no negatica (5.5) seria

$$x(i) - x(i-1) \geq 0 \tag{5.8}$$

Con esto hemos reducido la versión restringida de el problema de horario del operador telefonico en un problema de encontrar una solución factible en un sistema de diferencias restringidas.

5.1.4. Simulación y Resultados Numéricos aplicando el algoritmo de Bellman-Ford

A continuación presentaremos la simulación del algoritmo de Bellman-Ford sobre un sistema de diferencias restringuidas para poder hallar si existe una solución factible o no. Se mostrará las iteraciones de los pasos dados hasta llegar a encontrar el resultado deseado que deseamos.

Dado el sistema de diferencias restringuidas:

$$
\begin{aligned}
x_1 - x_2 &\leq 1 \\
x_1 - x_4 &\leq -4 \\
x_2 - x_3 &\leq 2 \\
x_2 - x_5 &\leq 7 \\
x_2 - x_6 &\leq 5 \\
x_3 - x_6 &\leq 10 \\
x_4 - x_2 &\leq 2 \\
x_5 - x_1 &\leq -1 \\
x_5 - x_4 &\leq 3 \\
x_6 - x_3 &\leq -8
\end{aligned}
$$

Obteniendo luego:

$$
\begin{pmatrix}
1 & -1 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & -1 & 0 & 0 \\
0 & 1 & -1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & -1 & 0 \\
0 & 1 & 0 & 0 & 0 & -1 \\
0 & 0 & 1 & 0 & 0 & -1 \\
0 & -1 & 0 & 1 & 0 & 0 \\
-1 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & -1 & 1 & 0 \\
0 & 0 & -1 & 0 & 0 & 1
\end{pmatrix}
\begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{pmatrix}
\leq
\begin{pmatrix} 1 \\ -4 \\ 2 \\ 7 \\ 5 \\ 10 \\ 2 \\ -1 \\ 3 \\ -8 \end{pmatrix}
$$

Produciendo el siguiente grafo:

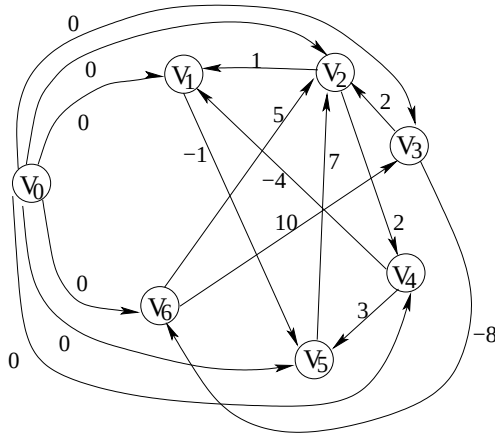


Figura 5.2: *Grafo con las restricciones correspondientes al sistema de diferencias restringuidas*

Produciendo la siguiente matriz de pesos:

$$
\begin{pmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & -1 & 0 \\
0 & 1 & 0 & 0 & 2 & 0 & 0 \\
0 & 0 & 2 & 0 & 0 & 0 & -8 \\
0 & -4 & 0 & 0 & 0 & 3 & 0 \\
0 & 0 & 7 & 0 & 0 & 0 & 0 \\
0 & 0 & 5 & 10 & 0 & 0 & 0
\end{pmatrix}
$$

A continuación mostraremos los momentos donde se dan los cambios en las iteraciones:

Inicializa:

Nodo	1	2	3	4	5	6	7
Padre	0	0	0	0	0	0	0
Dist	9999	9999	9999	9999	9999	9999	9999

Primer recorrido

Como están enlazados al nodo 1 de peso 0 se obtiene:

Nodo	1	2	3	4	5	6	7
Padre	0	1	1	1	1	1	1
Dist	0	0	0	0	0	0	0

Del nodo 2 al nodo 6:

Nodo	1	2	3	4	5	6	7
Padre	0	1	1	1	1	2	1
Dist	0	0	0	0	0	-1	0

Del nodo 4 al nodo 7:

Nodo	1	2	3	4	5	6	7
Padre	0	1	1	1	1	2	4
Dist	0	0	0	0	0	-1	-8

Del nodo 5 al nodo 2:

Nodo	1	2	3	4	5	6	7
Padre	0	5	1	1	1	2	4
Dist	0	-4	0	0	0	-1	-8

Del nodo 7 al nodo 3:

Nodo	1	2	3	4	5	6	7
Padre	0	5	7	1	1	2	4
Dist	0	-4	-3	0	0	-1	-8

Segundo Recorrido

Del nodo 2 al nodo 6:

Nodo	1	2	3	4	5	6	7
Padre	0	5	7	1	1	2	4
Dist	0	-4	-3	0	0	-5	-8

Del nodo 3 al nodo 5:

Nodo	1	2	3	4	5	6	7
Padre	0	5	7	1	3	2	4
Dist	0	-4	-3	0	-1	-5	-8

Del nodo 5 al nodo 2:

Nodo	1	2	3	4	5	6	7
Padre	0	5	7	1	3	2	4
Dist	0	-5	-3	0	-1	-5	-8

Tercer Recorrido

Del nodo 2 al nodo 6:

Nodo	1	2	3	4	5	6	7
Padre	0	5	7	1	3	2	4
Dist	0	-5	-3	0	-1	-6	-8

De aqui en adelante no existen más cambios. Al final se obtiene como resultado el vector x=(-5,-3,0,-1,-6,-8)

5.2. Aplicaciónes del Algoritmo de Dijkstra

Las aplicaciones del algoritmo de Dijkstra son diversas y de gran importancia en distintas áreas. Vamos a presentar algunas de ellas:

5.2.1. Aproximando funciones lineales por partes.

Existen actualmente numerosas aplicaciones en diferentes campos de la ciencia de funciones lineales por partes. En muchas ocasiones, estas funciones contienen un gran número de puntos de quiebre, los cuales son difíciles de almacenar y de manipular(aún el evaluar). En estas situaciones sería conveniente el reemplazar estas funciones lineales por partes por otra función de aproximación usando menos puntos de quiebre. Al realizar esta aproximación, podriamos ser capaces de ahorrar en espacio de almacenamiento y en el costo de utilización de la función, sin embargo, incurrimos en el costo debido a la inexactitud de la función de aproximación. Dicha aproximación busca conseguir el mejor equilibrio posible entre los conflictos de costos y beneficios.

Sea $f_1(x)$ una función lineal por partes de un escalar x. Representaremos la función en un plano de 2 dimensiones: De tal manera que pase através de n puntos $a_1 = (x_1, y_1)$, $a_2 = (x_2, y_2)$, ..., $a_n = (x_n, y_n)$. Supongamos que estos puntos posean un orden tal que $x_1 \leq x_2 \leq ... \leq x_n$. Asumimos que la función varia linealmente para todo dos puntos consecutivos x_i y x_{i+1}. Consideremos la situación en el cual n es muy grande y por razones prácticas deseamos aproximar la función $f_1(x)$ por otra función $f_2(x)$ que pase através solo de un subcobnjunto de puntos a_1, a_2, ..., a_n (incluyendo a_1 y a_n). Como ejemplo observamos, la figura 5.3: Aproximaremos la función $f_1(x)$ que pasa por 10 puntos por una función $f_2(x)$ que pasa por 5 puntos.

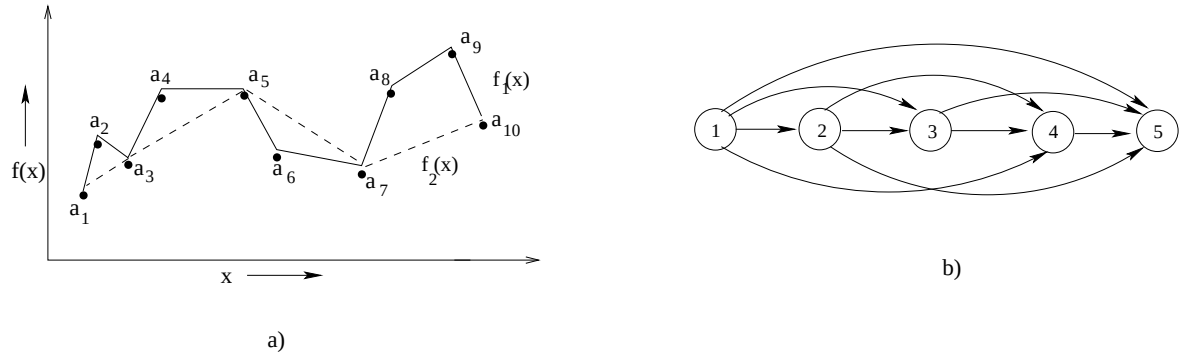


Figura 5.3: *(a)Aproximando la función $f_1(x)$ que pasa através de 10 puntos por la función $f_2(x)$; (b) Correspondiente al problema de caminos mínimos*

Esta aproximación resulta un ahorro de espacio de almacenamiento y en la utilización de la función. Para propósitos ilustrativos, asumiremos que nosotros podemos medir dicho costo por un unidad de costo α asociada a cualquier intervalo simple usa en la aproximación (la cual es definida por 2 puntos a_i, a_j). Notamos que la aproximación también introduce errores que viene asociado con consecuencias. Asumiremos que el error de una aproximación es proporcional a la suma de los errores al cuadrado entre los puntos actuales y los puntos estimados (es decir, la consecuencia

es $\beta \sum_{i=1}^{n} [f_1(x_i) - f_2(x_i)]^2$ para alguna constante β) Nuestro problema de decisión consiste en indentificar un subconjunto de puntos que sean usados para definir la función aproximación $f_2(x)$ tal que incurramos en un costo total mínimo, medido por la suma de gastos de almacenamiento y la utilización de la función aproximación, además del costo de los errores impuestos por la aproximación.

Formulamos este problema como un *problema de caminos mínimos* sobre una red G con n nodos, numeramos desde 1 hasta n consecutivamente. La red contiene una arista (i, j) para cada par de nodos i y j tal que $i < j$. En la figura damos un ejemplo de esta red con $n = 5$ nodos. La arista (i, j) en esta red significa que aproximaremos un segmentos lineales de la función $f_1(x)$ entre los puntos a_i, a_{i+1}, ..., a_j por un segmento lineal uniendo los puntos a_i y a_j. El costo c_{ij} de las aristas (i, j) tiene 2 componentes: El costo de almacenamiento α y la consecuencia asociada por la aproximación de todos los puntos entre a_i y a_j por los correspondientes puntos situados en la linea que unen a_i y aj. En el intervalo $[x_i, x_j]$, la función aproximación es $f_2(x) = f_1(x) + (x - x_i)[f_1(x_j - f_1(x_i)]/(x_j - x_i)$, tal que el costo total en los intervalos es:

$$c_{ij} = \alpha + \beta \left[\sum_{k=i}^{j} (f_1(x_k) - f_2(x_k))^2 \right]$$

Cada camino dirigido desde el nodo 1 al nodo n en G corresponde a la función $f_2(x)$ y el costo de este camino es igual al costo total por almacenar esta función y por el uso de esta aproximación de la función original. Por ejemplo, el camino 1-3-5 corresponde a la función $f_2(x)$ pasando através de los puntos a_1, a_3 y a_5. Como consecuencia de estas observaciones, vemos que el camino mínimo desde el nodo 1 hasta el nodo n especifica el conjunto de puntos óptimos necesarios para definir la función aproximación $f_2(x)$

5.2.2. Asignación de Actividades de Inspección en una línea de Producción

Una línea de producción consiste en una secuencia ordena de n fases de producción y cada fase tiene una operación de fabricación seguida de una inspección potencial. El producto entra en la fase 1 de la línea de producción en lotes de tamaño $B \geq 1$. Como los articulos dentro de un lote se mueve através de la fase de fabricación, las operaciones podrian introducir defectos. La probabilidad de producir un defecto en un fase 1 es α_i. Asumimos que todos los defectos son no reparables, tal que debemos descartar cualquier artículo defectuoso. Despues cada fase, puede inspeccionar todos los artículos o ninguno de ellos (no mostrando los artículos); asumimos que la inspección identifica todos los artículos defectuosos. La línea de producción debe finalizar con un estado de inspección tal que no se envie algun unidad defectuosa. Nuestro problema de decisión es encontrar un plan de inspección óptima que especifique cual es el momento en que debemos inspeccionar los artículos tal que se minimize el costo total de producción e inspección.

Con menos estados de inspección podriamos decrecer los costos de inspección, pero incrementaría el costo de producción debido a que se realizaría operaciones de fabricación innecesarias en algunas unidades que estarían defectuosas. El número óptimo de fases de inspección sería el que logre el equilibrio apropiado entre estos dos costos.

Supongamos que los siguientes datos de costo estan disponibles:

1. p_i, el costo de fabricación por unidad en una fase i.

2. f_{ij}, los costos fijos de inspeccionar un lote despues de la fase j, dado que la última vez inspeccionaron el lote despues de la fase i.

3. g_{ij}, la variable de costo por unidad por inspeccionar un artículo después de una fase j, dado que la última vez inspeccionarón el lote después de la fase i.

Los costos de inspección en el estado j dependerá de si el lote fue inspeccionado con anterioridad, es decir en el estado i, ya que el inspector necesita observar el caso de los defectos ocasionados en cualquiera de las fases intermedias $i+1$, $i+2$, ...,j.

Nosotros formulamos este problema de inspección como de caminos mínimos sobre una red de $(n+1)$ nodos, numerados de 1 hasta n. La red contiene una arista (i, j) para cada par de nodos i y j para lo cual $i < j$. La figura 5.4 muestra una red para un problema de inspección con 4 estados.

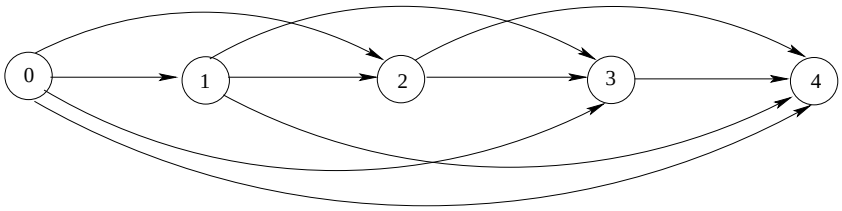


Figura 5.4: *Red de Caminos Mínimos asociado con un problema de inspección*

Cada camino en la red desde el nodo 0 al nodo 4 define un plan de inspección. Por ejemplo, el camino 0-2-4 implica que inspeccionamos los lotes después en la segunda y cuarta fase. Siendo $B(i) = B\prod_{k=1}^{i}(1-\alpha_k)$ denotamos el número de unidades no defectuosas en espera al final de la fase i, asociando el siguiente costo c_{ij} con cualquier arista (i, j) en la red:

$$c_{ij} = f_{ij} + B(i)g_{ij} + B(i)\sum_{k=i+1}^{j} p_k$$

Es fácil ver que c_{ij} denota el costo total incurrido en las fase $i+1, i+2, ..., j$; los primeros dos términos de la fórmula de costo son los costos de inspección fijos y variables, y el tercer término es el costo de producción incurrido en estas fases. Esta formulación de camino mínimo nos permite resolver la aplicación de inspección como un problema de flujo de red.

5.2.3. El Problema de la Mochila

Veamos la siguiente motivación: Un caminante debe decidir que mercancias incluirá en su mochila para su viaje. Este debe escoger entre p objetos: Objeto i de peso w_i y una utilidad u_i para el caminante. El ojetivo es maximizar la utilidad en el viaje del caminante sujeto a las limitaciones del peso que este pueda llevar siendo la capacidad de W kilos. Formularemos entonces el problema de la mochila como un problema del camino más largo sobre una red acíclica y mostraremos el como transformar un problema de camino más largo a un problema de camino mínimo. Para hacer una apropiada identificación entre fase y estado de cualquier programa dinámico y los nodos de la red, formularemos escencialmente problemas de programación dinámica determinística como equivalente a problemas de caminos mínimos. Ilustraremos nuestra formulación usando el problema de la mochila con 4 artículos que tienen los pesos y utilidades como se indica en la tabla.

j	1	2	3	4
u_j	40	15	20	10
w_j	4	2	3	1

En la figura 5.5 se muestra la formulación de los caminos más largos para este ejemplo del problema de la mochila, asumiendo que la mochila tiene una capacidad de $W = 6$. La red formulada tiene varias capas de nodos: Esta tiene una capa correspondiente para cada artículo y una capa correspondiente a un nodo inicial s y otro correspondiente a un nodo inferior t. Las correspondientes capas en un artículo i tiene $W + 1$ nodos, i^0, i^1, ..., i^W.

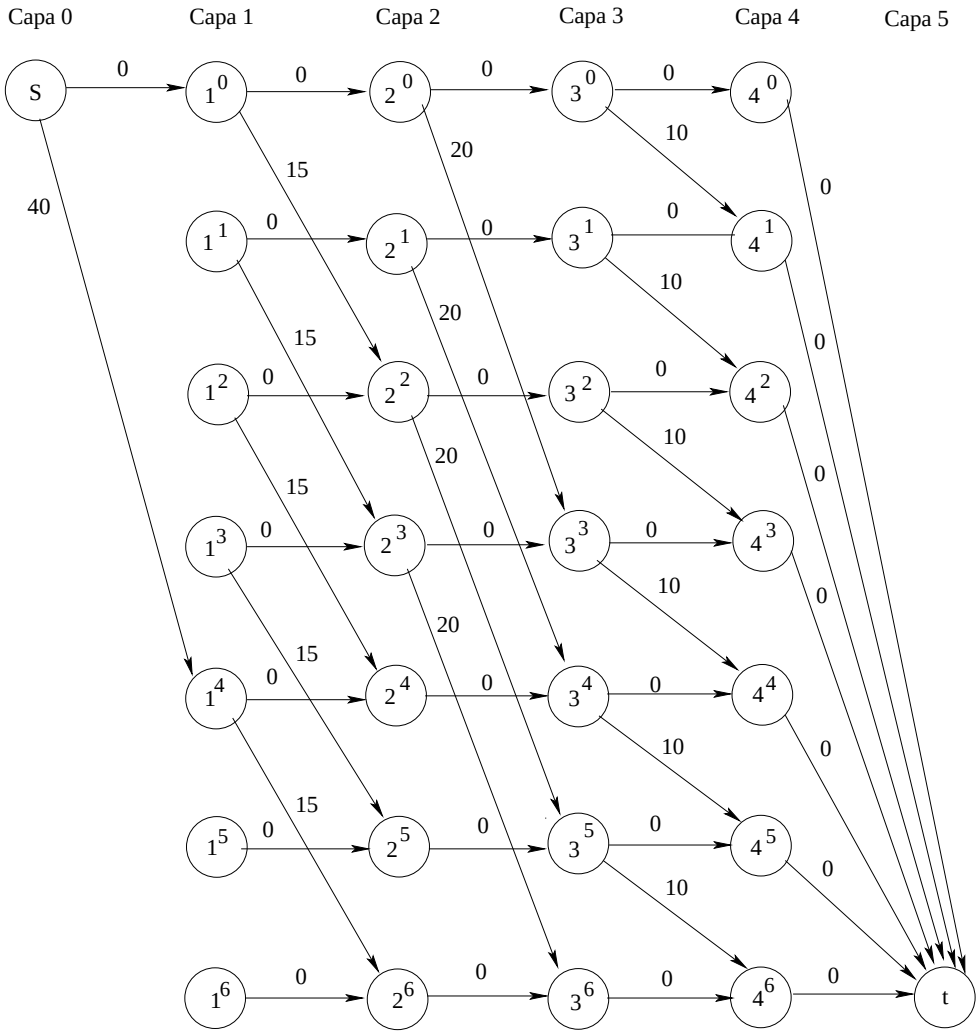


Figura 5.5: *Formulación del Camino más largo del Problema de la Mochila*

El nodo i^k en la red significa que los artículos 1, 2, ...,i han consumido k unidades de la capacidad de la mochila. El nodo i^k tiene a lo mucho dos arcos salientes, correspondiendo a dos decisiones:

1. No incluir el artículo $(i + 1)$ en la mochila
2. Incluir el artículo $i + 1$ en la mochila

Observar que nosotros escogemos la segunda alternativa solo cuando la mochila tiene la suficiente capacidad disponible para acomodar $(i+1)$ artículos, es decir $k+w_{i+1} \leq W$. La arista correspondiente a la primera decisión es $(i^k, (i+1)^k)$ con utilidad cero y la arista correspondiente a la segunda decision (proveniente de que $k+w_{i+1} \leq W$) es $(i^k, (i+1)^{k+w_i+1})$ con utilidad u_{i+1}.

El nodo inicial tiene 2 arista incidentes $(s, 1^0)$ y $(s, 1^{w-1})$ correspondiendo a las opciones de incluir si o no a el articulo 1 en la mochila vacia. Finalmente, se conecta todos los nodos en la capa correspondiente del ultimo articulo del nodo inferior t con las aristas de utilidad cero .

Cada posible solución del problema de la mochila define un camino dirigido desde el nodo s hasta el nodo t; ambas soluciones posibles y el camino tiene la misma utilidad. Visceversa, todo camino desde el nodo n hasta el nodo t define una posible solución del problema de la mochila con alguna utilidad. Por ejemplo el camino $s - 1^0 - 2^2 - 3^5 - 4^5 - t$ implican una solución la cual incluye los artículos 2 y 3 en la mochila y excluyen los artículos 1 y 4. Esto corresponde a mostrar que podemos encontrar la máxima utilidad seleccionando los artículos para encontrar el camino de utilidad máxima, esto es, el camino más largo en la red.

El problema del camino más largo y el problema de caminos mínimos están estrechamente relacionados. Podemos transformar un problema de camino más largo a un problema de caminos mínimos definiendo una arista de costo igual al negativo de la arista de utilidades. Si el problema de camino más largo contiene algún ciclo dirigido de longitud positiva, el resultado del problema de caminos mínimos contendrá un ciclo negativo y no podra ser resuelto con facilidad. Sin embargo, si todos los ciclos dirigidos en el problema del camino más largo tienen longitud no positiva, entonces en el correspondiente problema de caminos mínimos todos los ciclos dirigidos tendría longitud no negativa y este problema podría ser resuelto eficientemente. Observamos que en la formulación del camino más largo en el problema de la mochila, la red no es acíclica, asi el resultado del problema de caminos mínimos es eficientemente resoluble.

5.2.4. Extracción de características curvilineas de imágenes remotas usando técnicas de minimización de caminos

La extracción de características curvilineas de imágenes es una tarea importante en muchas aplicaciones del análisis de imágenes.

El presente trabajo es un proyecto para extraer representaciones vectoriales de caminos, carreteras y redes de rios de una imagen remotamente detectada para usarse en el ambiente de sistemas de información.

La imagen se representa como una matriz de puntos, cada uno con una especial intensidad. Cada nodo corresponde a un punto(píxel)de la imagen y tiene hasta ocho nodos adyacentes. El peso de los arcos viene dado en este caso por la diferencia de intensidad. Esta técnica presenta un gran ahorro de costo frente a las herramientas existentes que usan métodos de vectorización automático.

Funciones de Costo mínimo

Existe una amplia gama de aproximaciones a este problema de extraer características curvilineas dependiendo del tipo de imágenes (borrosas, claras, artificiales o naturales) y de la clase de características que involucra (amplio, estrecho, derecho o con torción).

Podemos plantear este problema como el de encontrar una trayectoria de costo mínimo a traves de una red. Por ejemplo, el problema de encontrar una estrecha linea oscura en una imagen clara, dado los puntos finales de una linea, involucra encontrar un camino entre 2 puntos el cual minimize.

$$\sum_{x \in S} I_x$$

donde I_x es la intensidad de la imagen en la posición del pixel x y S es el sistema de pixeles a lo largo de la trayectoria

Si relativamente las características directas son extraidas, la curvatura a lo largo del camino puede ser minimizada simultaneamente añadiendo el término de la forma

$$k \cdot \sum_{x \in S} c_x$$

donde c_x es un estimación de la curvatura de la trayectoria en la posición del pixel x y k es una constante la cual determina la fuerte relatividad del término.

Si los bordes son extraidos como lineas, por ejemplo a lo largo de los límites de una curva donde su característica es su amplitud, tal como la desenbocadura del rio que se ve en la figura 5.3, el camino de costo mínimo aproximado puede ser aplicado a la imagen modificada en lugar de la imagen pura.

Si una camino con intensidad marcada I_t es requerido, una apropiada función de costo puede ser dada por

$$k \cdot \sum_{x \in S} \mid I_t - I_x \mid$$

El uso de la búsqueda de caminos aproximados para lineas y bordes consecutivoss en imágenes fue propuesto hace mucho tiempo. Una diferencia se puede encontrar entre un óptimo y exhaustivo algoritmo de búsqueda, el cual encuentra un camino de costo mínimo global y un algoritmo de búsqueda heurística, el cual usa heurística para reducidir los espacios buscados. Un ejemplo es el caso de Montanari, quien introdujo la programación dinámica para detectar curvas óptimas con respecto a una figura en particular y Martelli que introdujo una búsqueda heurística para detectar bordes y contornos.

Luego de esos estudios, muchos autores han querido aplicar técnicas de búsqueda heurística para problemas de límites y detección de lineas, por ejemplo, los tiempos de búsqueda óptima han sido significativamente menos atractivos que una búsqueda heurística; cuando la heurística este disponible; y su naturaleza de computo intensivo ha dado como resultado la carencia de popularidad. Recientemente, la aproximación de la minimización de costos se esta trabajando bajo la idea de encontrar una solución local para un problema de camino de costo mínimo dado, donde el usuario da un camino inicial(idea de las serpientes). Si el mínimo local no es correcto, los términos adicionales pueden ser añadidos a la función de costo iteractuando para presionar a la serpiente(el camino) hacia la posición correcta.

Encontrando el camino óptimo

La idea de una búsqueda para una solución global óptima es lo importante en el análisis de este problema, pero un algoritmo de búsqueda relativamente rápido es adoptado y el área de búsqueda puede ser restriguido al seleccionar secciones de la imagen la cual incluyan las características requeridas

El problema de encontrar el camino de costo mínimo entre dos puntos es uno de los muchos problemas relacionados con grafos. Aunque ellos no han tenido uso extensivo en el análisis de

imágenes han sido ampliamente explorados y se han propuesto algoritmos eficientes. Estos algoritmos están expresados en términos de funciones de costo mínimo el cual usualmente calcula el incremento del costo del movimiento entre dos nodos adyacentes.

El algoritmo de Dijskstra originalmente es un algoritmo que garantiza el encontrar un camino de costo mínimo entre dos puntos. Este divide la red de nodos en dos conjuntos, aquellos para los cuales el costo mínimo del nodo inicial se conoce y para los que no se conoce. El algoritmo procede trasladando nodos desde el segundo conjunto al primero, examinando los vecinos de cada nodo en el primero conjunto. Cuando el costo es conseguido el nodo destino es conocido y el algoritmo finaliza. Muchas mejoras han sido dadas para hacer óptima la eficencia del algoritmo.

El algoritmo de Moore es similar al de Dijkstra, pero en vez de considerar dos conjuntos de nodos, este usa una cola de nodos, cuyos costos aún están por determinarse. Los miembros de la cola son removidos por el frente de la cola y sus nodos vecinos son examinados y añadidos al final de la cola para tener sus costos determinados. Cuando la cola esta eventualmente vacia, el algoritmo termina y el camino de costo mínimo desde el nodo inicial a cualquier otro nodo ha sido calculado. El costo del nodo destino esta entre estos.

Una mejora del algoritmo de Moore, atribuida a d'Esopo y desarrollado por Pape, utiliza una estrategia más complicada para agregar los nodos a cualquier parte de la cola, sea al frente o al final, dependiendo si han sido procesados antes. Esto mejora el funcionamiento del algoritmo, permitiendo que las soluciones sean calculadas en pocas iteraciones. Se conoce además que la complejidad del algoritmo de d'Esopo es $O(n)$.

Implementación

Nosotros hemos ampliado el algoritmo básico de d'Esopo para permitir que las funciones de costo en un punto dependan de más nodos que de apenas uno solo y sus vecinos inmediatos . La información hace disponible una función de costo, que incluye detalles fijos acerca de todo el camino, tales como las localización los puntos inicio y final y cualquier punto inmediato que ha sido indicado por el usuario.

Existe también información dinámica acerca del nodo que es considerado actualmente y nodos previos a lo largo del mejor camino encontrado hasta ahora. Esto facilita la inclusión de funciones de costo más complejas dependiendo, por ejemplo la curva del camino.

```
ALGORITMO EXTENDIDO DE D'ESOPO

  Para cada nodo
    nodo.costo = infinito
    nodo.direccion= indefinido

   Nodo_inicio.costo=0
   Inicializar(Info_camino)
   Ingresar a la cola(nodo_inicial)

 Mientras cola <> vacio
   nodo = remover_inicio_de_cola
```

```
Para cada vecino del nodo
 actualizar (info_camino, vecino)
 nuevo_costo = nodo.costo + costo_de(vecino,nodo,info_camino)

 Si nuevo_costo < costo.vecino
   vecino.costo = nuevo_costo
   vecino.direccion = direc_de(nodo)
 Si vecino nunca estuvo en cola
     Ingresar_a_cola(vecino)
   Si vecino no esta actualmente en cola
     Ingresar_a_cola(vecino)
```

Para el problema específico de encontrar un camino de costo mínimo en imágenes, algunas decisiones se toman acerca de la representación de la red. La imagen es representado por un mallado de puntos, la cual tiene una intensidad en particular. Cada nodo de la red corresponde a un punto(píxel) en la imagen y tiene ocho puntos vecinos los cuales corresponden a los puntos adyacente en la imagen.

La red, así mismo, es también representado como un mallado. Estos enlace entre los nodos no son explicitamente almacenados pero están implícitos en las locaciones de los nodos dentro de la malla. Cada nodo mantiene un registro del costo total del mejor camino hacia el nodo, la longitud del mejor camino y la dirección al punto previo en el camino. Además, las marcas indican si el nodo ha sido puesto en cola. Si este esta actualmente en cola o si este es un punto intermedio del camino seleccionado por el usuario.

El costo de moverse entre los nodos es calculado al ir procediendo el algoritmo y bajo algunas funciones de intensidad de puntos y localización dentro de la imagen. La función de costo es implementada de forma separada del algoritmo principal ya que pueden usarse diferentes funciones de costo. El usuario puede también aplicar información arbitraria a la función de costo.

Este diseño permite que una gran variedad de funciones de costo puedan ser implementadas, incluyendo costos que dependan de las intensidades de los puntos inicial y final y costos que varian con la curva local del camino.

Resultados

A continuación veamos la siguiente imagen:

En la figura se muestra las características a lo largo de un rio. La superficie del rio esta de color gris claro y las zonas urbanas y bancos de rio son de gris oscuro.

Figura 5.6: Imagen del Rio

En la figura muestra el resultado de usar herramientas digitales que ven la rugosidad de los límites del rio. La extracción de caminos son sobrepuestos en blanco sobre la imagen original.

Figura 5.7: Extracción de las bordes del Rio

En este ejemplo del los límites del rio es identificado por el cambio de intensidad entre las superficies del rio y el banco de rios. Para extraer estas características el detector de bordes Sobel fue aplicado en la imagen para iluminar los límites y la herramienta para digitalizar la rugosidad fue usado para continuar hallando las características de los bordes.

Otro caso de análisis podria ser la siguiente imagen:

En la figura, se muestra el curso del rio en Nepal. La imagen es infraroja, así el rio se muestra oscura a comparación con el calor de la tierra.

Figura 5.8: Rio de Nepal

En este ejemplo, el usuario ha solo selecionado dos puntos del rio, próximos a la parte superior e inferior de la imagen. El camino encontrado por el algoritmo es mostrado sobreponiendo una linea blanca sobre la imagen como se ve en la figura.

Figura 5.9: Extracción de la trayectoria del Rio

La peso relativo entre los términos que se utilizan para calcular el costo pueden ser ajustados y la trayectoria de costo mínimo resultante se demuestra destacada y sobrepuesta en la imagen original.

En este caso, para encontrar el camino oscuro, la función de costo minimiza la suma de las intensidades a lo largo del camino. Este puede ser visto, en el curso del rio que ha sido satisfactoriamente localizado a lo largo de su longitud.

5.2.5. Encaminamiento de paquetes

Consideremos una red telefónica en un momento dado, un mensaje puede tardar una cierta cantidad de tiempo en atravesar cada linea(debido a los efectos de congestión, retrazos en las conexiones, etc). En este caso tenemos una red con costes en los arcos y dos nodos especiales: el nodo comienzo y el nodo finalización, el objetivo aquí es encontrar un camino entre estos dos nodos cuyo coste total sea el mínimo.

Cuando deseamos enviar alguna información a través de la red, serán necesarias muchas conexiones a una ruta y esto se realiza a través de más de un conmutador, las cuales deben tener:

- Eficiencia: Usar el mínimo de equipamento posible
- Flexibilidad: Debe proporcionar tráfico razonable aún en condiciones adversas

Los conmutadores de la red telefónica pública tienen una estructura en árbol, en el caso del enrutamiento estático siempre se utiliza el mismo esquema a comparación del enrutamiento dinámico que permite cambios de rutas o topología (rutas alternativas) dependiendo del tráfico.

El encaminamiento en redes de paquetes conmutadas será un proceso complejo de vital importancia que debe presentar las siguiente características:

- Correctitud
- Simplicidad
- Robustez
- Estabilidad
- Imparcialidad
- Optimización
- Eficencia

Para lo cual se tomarán en cuenta criterios de rendimiento que buscarán seleccionar la mejor ruta a utilizar y obtenga el costo mínimo, evite retardos y sea eficiente. La red estará formada por nodos y unidos por aristas que llevarán los costos asociados a sus enlaces. Las decisiones para encontrar el mejor encaminamientos pueden ser basadas en el conocimiento de la red (no siempre es así):

- **Enrutamiento Distribuido:** Donde los nodos usan conocimiento local y a su vez pueden recolectar información de nodos adyacentes y de cualquier nodo en una ruta potencial
- **Enrutamiento Centralizado:** Donde se puede recolectar información de cualquier nodo
- **Tiempo de actualización:** Es cuando la información de la red es actualizada en los nodos. Un esquema fijo no se actualiza pero en el caso adaptivo se busca analizar la continuidad, el periodo, la importancia del cambio de carga y en la topología (nueva ruta alterna)

Existen 4 tipos de enrutamiento: Estático, Inundaciones, Aleatorio y Adaptable

Encaminamiento Estático

Es aquel donde se configura una única ruta permanente para cada par de estaciones origen-destino de la red. Cada ruta tiene un coste asociado de acuerdo a una determinada métrica, por ejemplo el número de nodos atravesados o el retraso de propagación en cada enlace de la ruta. Para esto se utiliza un algoritmo de costo mínimo como puede ser el algoritmo de Dijkstra o Bellman-Ford que nos ayuda a calcular la ruta ideonea. La información obtenida por el algoritmo se traslada a una *Matriz de Encaminamiento Central* (MEC) que reside normalmente en un nodo de conmutación, que hace las funciones de coordinación central (nodo central), permaneciendo invariable a menos que cambie la topología de la red. A partir de la MEC se crean *Tablas de Encaminamiento* asociadas (TE) que se almacenan en cada nodo de la red Una TE específica, para cada destino posible, el nodo siguiente al que han de ser reenviados los datos para poder llegar a su destino

Encaminamiento por Inundaciones(Flooding)

Aqui se utilizan todas las rutas existentes entre el par de nodos que desean comunicarse, el nodo origen envia copias del mismo paquete, con su dirección de destino, por todos los enlaces de salida. Los nodos intermedios de la red reenvian cada paquete que les llega por todos los enlaces conectados a dicho nodo, excepto por el enlace por el que llega el paquete. Tras un tiempo determinado, varias copias del mismo paquete llegarán a su destino para lo cual los paquetes duplicados se descartan en el destino. Para evitar aumentar la carga de la red innecesariamente con los reenvios de paquetes se puede tomar en cuenta los siguiente:

- Los nodos recuerdan la identidad de los paquetes que ya han sido transmitidos
- Cada paquete incluye un campo de cuenta de saltos(hops) que decrece cada vez que es reenviado

Se intentarán todas las rutas posibles entre origen-destino y por lo tanto al menos un paquete tomará la ruta con el menor número de saltos posibles por lo cual todos los nodos se visitan, ayudando a proporcionar información sobre ellos. La desventaja radica en que varias copias del mismo paquete viajan por la red, lo que implica una sobrecarga.

Encaminamiento Aleatorio

La elección de la ruta se puede realizar de manera probal´ística o por turnos. Cada nodo reenvia por un único enlace de salida. La selección del enlace se realiza con igual probalidad de entre aquellos enlaces que pueden conducir al destino. No es necesario información de la red por lo cual la ruta elegida no será normalmente la óptima

Encaminamiento Adaptivo

Es el método más usado en redes de conmutación de paquetes. La elección de la ruta se realiza de acuerdo al estado de la red (fallos o congestión). El encaminamiento de cada nodo cambia cuando las condiciones de la red no son favorables, por lo cual los algoritmos que se utilizan necesitan saber el estado de la red periodicamente. Cuanto más información se recoga de la red mejor sera la decisión de encaminamiento adoptada aunque esto produzco un tráfico adicional .

5.2.6. Simulación y Resultados Numéricos aplicando el Algoritmo de Dijkstra

En este caso de aplicación consideremos una red de telefonia. En un determinado instante se envia un mensaje que demorará un cierto tiempo para recorrer cada linea. La cantidad de tiempo estimado puede producir a la empresa de telecomunicaciones un gasto en tiempo y dinero en el seguimiento de los retrazos. Se necesita entonces un enrutamiento con el fin de reducir al mínimo los retrazos.

Observamos la siguiente figura y vemos que posee seis nodos y los respectivos pesos en las aristas(costos). Dado un nodo inicial se buscara hallar el camino mínimo al nodo final (nodo 6) donde se desea enviar el mensaje.

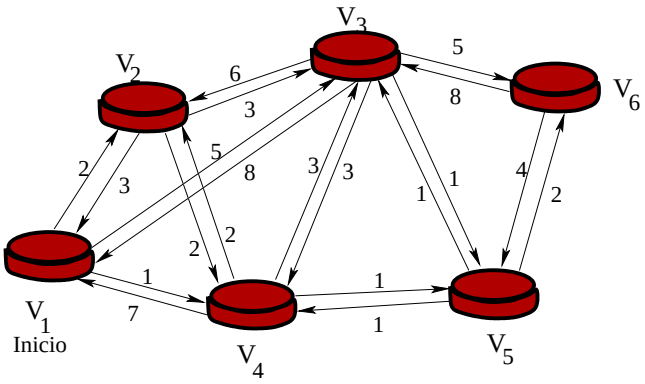


Figura 5.10:

Primero se *inicializa* las distancias colocando 9999 cuando sea infinita la distancia del nodo inicial a los demás y el arreglo D-temp donde se alojarán las distancias temporales.

Nodo	1	2	3	4	5	6
Dist	9999	9999	9999	9999	9999	9999
Padre	0	0	0	0	0	0
D-temp	9999	9999	9999	9999	9999	9999

Al ser el nodo 1 el inicial es el primero en ser alcanzado su distancia será 0 y será el primer mínimo a colocarse en la distancia y se colocar 0 en la cola indicando que ese nodo ya fue utilizado

Nodo	1	2	3	4	5	6
Dist	0	9999	9999	9999	9999	9999
Padre	0	0	0	0	0	0
D-temp	-	9999	9999	9999	9999	9999

PRIMERA ITERACIÓN

A continuación ira relajando para los nodos que son adyacente al nodo 1

Del nodo 1 al nodo 2

Nodo	1	2	3	4	5	6
Dist	0	9999	9999	9999	9999	9999
Padre	0	1	0	0	0	0
D-temp	-	2	9999	9999	9999	9999

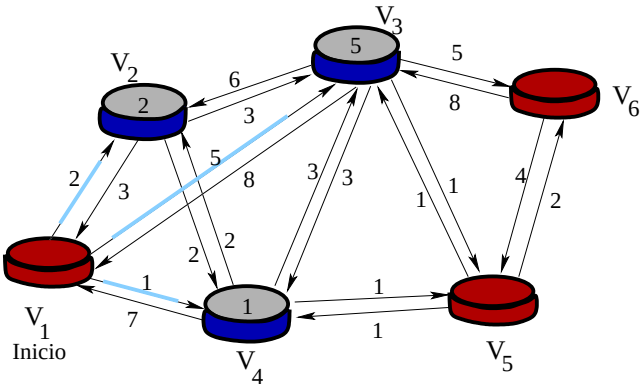


Figura 5.11:

Del nodo 1 al nodo 3

Nodo	1	2	3	4	5	6
Dist	0	9999	9999	9999	9999	9999
Padre	0	1	1	0	0	0
D-temp	-	2	5	9999	9999	9999

Del nodo 1 al nodo 4

Nodo	1	2	3	4	5	6
Dist	0	9999	9999	9999	9999	9999
Padre	0	1	1	1	0	0
D-temp	-	2	5	1	9999	9999

SEGUNDA ITERACIÓN

Extraerá el mínimo valor que se encuentre en D-temp, el cual es 1 con el nodo 4 y buscará ahora todos los adyacentes a dicho nodo, obteniendo la siguiente tabla.

Nodo	1	2	3	4	5	6
Dist	0	9999	9999	1	9999	9999
Padre	0	1	1	1	0	0
D-temp	-	2	5	-	9999	9999

Volverá a relajar ahora para todos los adyacentes al nodo 4

Del nodo 4 al nodo 1 ya no habria iteracción ya que el nodo 1 ya fue descubierto

Del nodo 4 al nodo 2

Nodo	1	2	3	4	5	6
Dist	0	9999	9999	1	9999	9999
Padre	0	1	1	1	0	0
D-temp	-	2	5	-	9999	9999

En el cuadro anterior se observa que no existen cambios del nodo 4 al nodo 2 ya que la distancia estimada previa al nodo 2 es menor que la que se obtiene relajando

Del nodo 4 al nodo 3

Nodo	1	2	3	4	5	6
Dist	0	9999	9999	1	9999	9999
Padre	0	1	4	1	0	0
D-temp	-	2	4	-	9999	9999

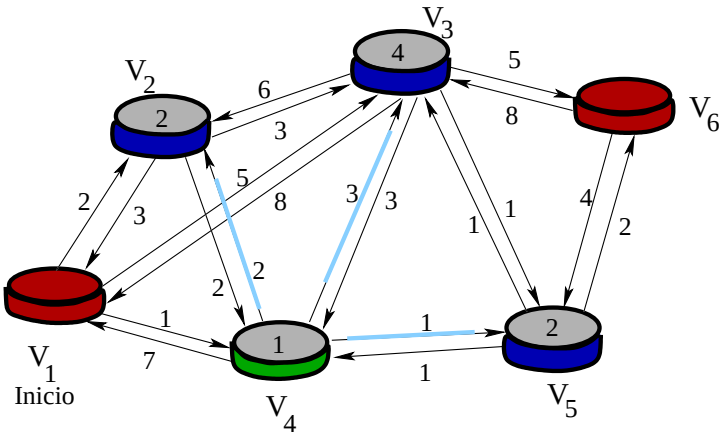


Figura 5.12:

Del nodo 4 al nodo 5

Nodo	1	2	3	4	5	6
Dist	0	9999	9999	1	9999	9999
Padre	0	1	4	1	4	0
D-temp	-	2	4	-	2	9999

TERCERA ITERACIÓN

Nuevamente extraerá el valor mínimo que se encuentra en D-temp el cual es 2 con el nodo 2.

Nodo	1	2	3	4	5	6
Dist	0	2	9999	1	9999	9999
Padre	0	1	4	1	4	0
D-temp	-	-	4	-	2	9999

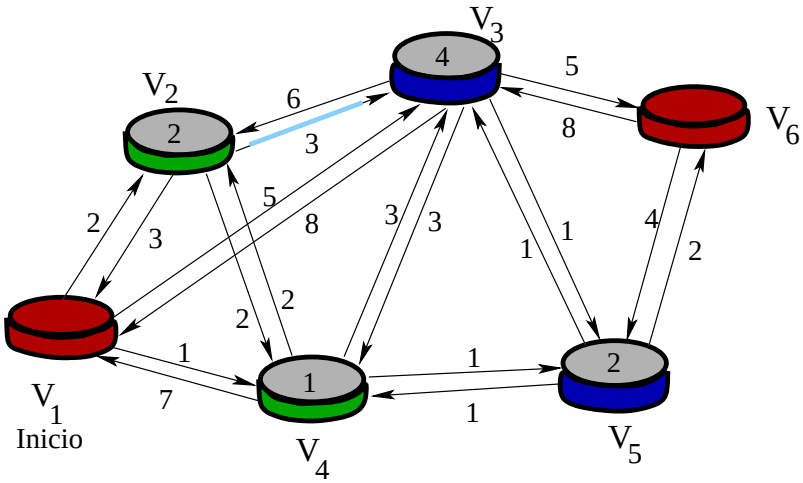


Figura 5.13:

Relajará nuevamente a los nodos adyacentes al nodo 2 que aún estén en la cola. En el caso de los nodos 1 y 4 no existiría iteración ya que ya fueron descubiertos

Del nodo 2 al nodo 3

Nodo	1	2	3	4	5	6
Dist	0	2	9999	1	9999	9999
Padre	0	1	4	1	4	0
D-temp	-	-	4	-	2	9999

CUARTA ITERACIÓN

Extrae el valor mínimo que se encuentre en D-temp el cual es 2 con el nodo 5

Nodo	1	2	3	4	5	6
Dist	0	2	9999	1	2	9999
Padre	0	1	4	1	4	0
D-temp	-	-	4	-	-	9999

Relajará ahora para todos los nodos adyacentes al nodo 5 que aún se encuentren en la cola. El nodo 4 ya fue descubierto

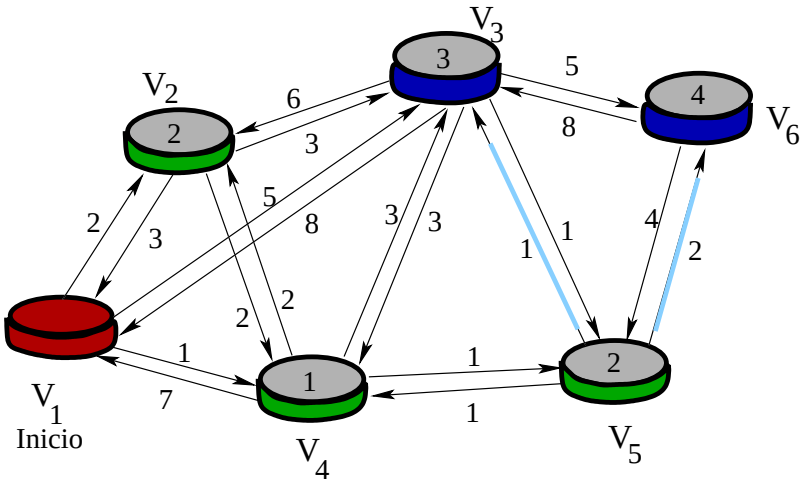


Figura 5.14:

Del nodo 5 al nodo 3

Nodo	1	2	3	4	5	6
Dist	0	2	9999	1	2	9999
Padre	0	1	5	1	4	0
D-temp	-	-	3	-	-	9999

Del nodo 5 al nodo 6

Nodo	1	2	3	4	5	6
Dist	0	2	9999	1	2	9999
Padre	0	1	5	1	4	5
D-temp	-	-	3	-	-	4

QUINTA ITERACIÓN

Extraer el valor mínimo de D-temp el cual es 3 con el nodo 3

Nodo	1	2	3	4	5	6
Dist	0	2	3	1	2	9999
Padre	0	1	5	1	4	5
D-temp	-	-	-	-	-	4

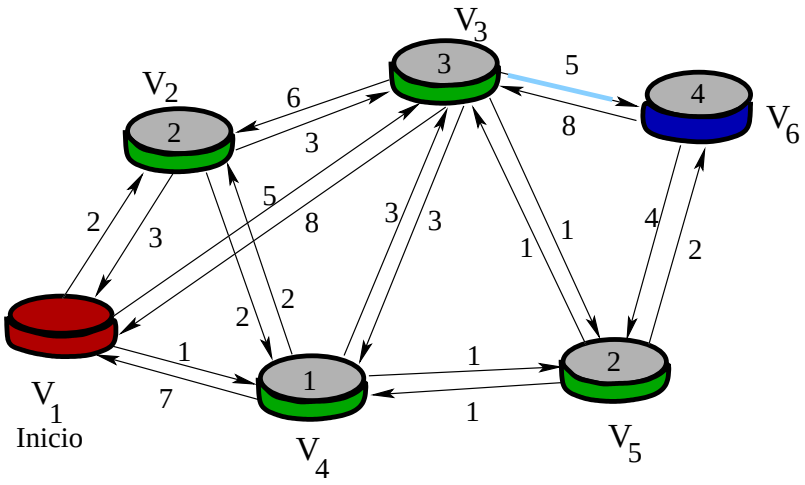


Figura 5.15:

En este momento no se da otra relajación. Se extraerá el último valor mínimo de D-temp

Nodo	1	2	3	4	5	6
Dist	0	2	3	1	2	4
Padre	0	1	5	1	4	5
D-Temp	-	-	-	-	-	-

Obteniendose como resultado los caminos mínimos desde la fuente dada a todos los demás nodos, indicando la ruta correspondiente a seguir:

CAMINOS MÍNIMOS DESDE EL VÉRTICE INICIAL 1
Distancia del vértice 1 al vértice 1 es: 0 La Ruta es: 1
Distancia del vértice 1 al vértice 2 es: 2 La Ruta es: 1-2
Distancia del vértice 1 al vértice 3 es: 3 La Ruta es: 1-4-5-3
Distancia del vértice 1 al vértice 4 es: 1 La Ruta es: 1-4
Distancia del vértice 1 al vértice 5 es: 2 La Ruta es: 1-4-5
Distancia del vértice 1 al vértice 6 es: 4 La Ruta es: 1-4-5-6

Capítulo 6

Conclusiones

Presentaremos a continuación a modo de resumen una tabla comparativa con los tipos de implementaciones que se puedan dar para el problema de caminos mínimos en caso particular para el Algoritmo de Dijkstra. Tomar encuenta que estas implementaciones también podrian ser utilizadas para el Algoritmo de Bellman-Ford, ya que tiene como base el Algoritmo de Dijkstra, previo análisis de los ciclos negativos que se puedan presentar. Los primeros datos que recibimos para el análisis del problema son los nodos y las aristas del respectivo grafo modelado, la implementación adecuada nos facilitará el acceso a estos.

Implementación por Matriz de Adyacencia:

- El costo de las operaciones de ver si el grafo esta vacio o no es constante
- El costo de las operaciones añadir-nodo, eliminar-arista, encontrar-nodo y añadir-arista es lineal respecto al número de nodos. El costo de adyacentes es lineal respecto al número de nodos, dado que hay que comprobar para todos los nodos si son distintos del nodo dado. El costo de la operación eliminar-nodo es cuadrático respecto al número de nodos.
- Si el grafo es no dirigido la matriz es simétrica
- Es la representación adecuada para grafos densos.
- El costo por espacio es cuadrático respecto al número de nodos.

Implementación por Lista de Adyacencia :

- El costo por analizar si el grafo esta vacio o no es constante
- El costo de las operaciones encontrar-nodo, añadir-arista, eliminar-arista, encontrar-arista, añadir-nodo es lineal respecto al número de nodos. El costo de la operación eliminar-nodo, es el producto del número de nodos por el máximo de los grados de los nodos.
- El costo por espacio es del orden de el número de nodos más el número de aristas.
- Esta representación es adecuada para grafos dispersos.

La implementación original del algoritmo de Dijkstra cuya complejidad es del orden $O(n^2)$, es dada a través de arreglos, la cual es estudiada en el capítulo 4 y en una aplicación en el Capítulo 5. Las diferentes formas de implementación fueron presentados para dar un alcanze de las distintas estrategias que existen para mejorar el tiempo de ejecución.

ALGORITMO	COMPLEJIDAD	CARACTERÍSTICA
Implementación Original	$O(n^2)$	1. Selecciona un nodo con un valor temporal de distancia mínima, designando este como permanente y examina las aristas incidentes a este nodo para modificar los otros valores de distancia 2. Es fácil de implementar 3. Logra el mejor tiempo de ejecución disponible para redes densas
Implementación Circular	$O(m + nC)$	1. Almacena los valores temporales de los nodos en una clasificacion ordenada en unidades de espacio cúbico e identifica el valor de la distancia mínima temporal examinando secuencialmente los arreglos 2. Es fácil de implementar y tiene un comportamiento empírico excelente 3. El tiempo de ejecución del algoritmo es pseudopolinomial y por lo tanto es teoricamente poco atractivo
Implementación con Heap Binario	$O(m.log(n))$	1. Esta estructura requiere un tiempo del orden O(log (n)) para insertar, extraer, decrecer-clave y requiere un tiempo del orden O(1) para otras operaciones del heap. 2. Es una implementación mas lenta que la implementación original en el caso de redes densas (es decir, $m = \Omega(n^2)$), pero es rápida cuando $m = O(n^2/logn)$
Implementación con d-Heap	$O(m.log_d n)$ donde d = m/n	1. Usa una estructura de datos d-heap para mantener los valores temporales de los nodos 2. El tiempo de ejecución lineal es siempre $m = \Omega(n^{1+\epsilon})$ para cualquier valor positivo $\epsilon > 0$
Implementación con Heap de Fibonacci	$O(m + n.log(n))$	1. Usa la estructura de datos de un heap de Fibonacci para mantener los valores temporales de los nodos 2. Logra el mejor tiempo polinómico de ejecución para resolver problemas de caminos mínimos 3. Intrincado y difícil de implementar
Implementación con Heap de base	$O(m + n.log(nC))$	1. Usa un heap de base para implementar el algoritmo de Dijkstra 2. Mejora el algoritmo Dial al almacenar los valores de los nodos temporalmente en arreglos con anchos variables 3. Logra un excelente tiempo de ejecución para problema que satisfacen similares hipótesis

Cuadro 6.1: Resumen de las Diferentes Implementaciones del Algoritmo de Dijkstra

Apéndice A

Implementación del Algoritmo de Dijkstra

```
unit Uinicio;
interface
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, Borland.Vcl.StdCtrls, Borland.Vcl.Buttons, System.ComponentModel,
  Borland.Vcl.Grids, Borland.Vcl.ExtCtrls;
type
  TForm1 = class(TForm)
    GroupBox1: TGroupBox;
    Edit2: TEdit;
    Label1: TLabel;
    BitBtn2: TBitBtn;
    GroupBox3: TGroupBox;
    Label7: TLabel;
    Label3: TLabel;
    Label2: TLabel;
    Edit3: TEdit;
    Edit4: TEdit;
    Edit1: TEdit;
    BitBtn1: TBitBtn;
    StringGrid1: TStringGrid;
    Label5: TLabel;
    GroupBox2: TGroupBox;
    Label6: TLabel;
    Edit5: TEdit;
    BitBtn3: TBitBtn;
    BitBtn4: TBitBtn;
    GroupBox4: TGroupBox;
    Label4: TLabel;
    Image1: TImage;
    procedure Edit1KeyPress(Sender: TObject; var Key: Char);
    procedure Edit2KeyPress(Sender: TObject; var Key: Char);
```

```
    procedure Edit3KeyPress(Sender: TObject; var Key: Char);
    procedure Edit4KeyPress(Sender: TObject; var Key: Char);
    procedure BitBtn1Click(Sender: TObject);
    procedure Edit5KeyPress(Sender: TObject; var Key: Char);
    procedure BitBtn3Click(Sender: TObject);
    procedure BitBtn4Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  Form1: TForm1;
implementation
{$R *.nfm}
type
 adyacente=array [1..100,1..100] of integer;
 matriz=array [1..100,1..100]of integer;
 Dist=array [1..100] of integer;
 Cola=array [1..100] of integer;
 Padre=array [1..50] of integer;
 Camino=array [1..50] of integer;
 lineas=array [1..100]of integer;

procedure inicializa(Rejilla:TstringGrid; n:integer);
var
 w,k:integer ;
 begin
  for w:=1 to n do
   for k:=1 to n do
    Rejilla.Cells[w,k]:=IntToStr(0)
 end;

procedure crear( var A:matriz; var B:adyacente; Rejilla:TstringGrid; n:integer);
 var
  i,j:integer;
 begin
  for i:=1 to n do
   for j:=1 to n do
     begin
      A[i,j]:=StrToint(Rejilla.cells[j,i]);
       if A[i,j]<>0 then
        B[i,j]:=1;
     end;
 end;

procedure LEER(var A:matriz; var n:integer);
var
```

```
i,j:integer;
Mat:string;
begin
 for i:=1 to n do
   begin
   for j:=1 to n do
      Mat:=Mat+IntToStr(A[i,j])+#9;
    Mat:=Mat+#13;
    end;
   ShowMessage('LA MATRIZ DE PESOS ES:'+#13+Mat);
end;

procedure inicializarQ( var D:Dist; var Q:Cola; n,ini:integer);
const
max=9999;
var
i:integer;
begin
 for i:=1 to n do
   begin
    Q[i]:=max;
    D[i]:=max;
   end;
 Q[ini]:=0;
 D[ini]:=0;
end;

procedure extraermin( var Q:Cola; var f:integer; var D:Dist; ini,n:integer);
const
max=9999;
var
temp,i:integer;
begin
temp:=max;
f:=ini;
for i:=1 to n do
  begin
   if (temp>Q[i]) and (Q[i]>0)then
       begin
        temp:=Q[i];
        f:=i
        end;
   end;
Q[f]:=0;
if temp<D[f] then
  D[f]:=temp;
end;
```

```
procedure relajar(var A:matriz;var Q:Cola; var D:Dist; var P:Padre; f,k:integer);
begin
   if Q[k] > D[f]+ A[f,k]  then
     begin
      Q[k]:= D[f]+ A[f,k];
      if Q[k]<D[k] then
        P[k]:=f;
    end;
end;

function ruta(var P:Padre; var R:Camino;  i:integer ;var L:lineas ):string;
var
j,n,cont:integer;
cam:string;
begin
    j:=1;
    cont:=i;
    while P[i]>0 do
       begin
         R[j]:=i;
         i:=P[i];
         j:=j+1;
       end;
     cont:=cont*j; {total de lineas a recorrer}
   R[j]:=i;
   for n:=j downto 1 do
    begin
     Cam:=Cam + IntToStr(R[n])+' ';
     if n>1 then
        begin
          L[cont]:=R[n]*10+R[n-1];
           cont:= cont-1;
         end;
     end;
    Ruta:=Cam;
 end;

function escribir(var D:Dist ;var P:Padre; var R:Camino; ini,n:integer;
                  var L:Lineas):string;
var
i:integer;
RESP:STRING;
begin
for i:=1 to n do
 begin
  if D[i]<9999 then begin
    RESP:=RESP+'La distancia del vertice '+IntToStr(ini)+' al vertice '+InTtoStr(i)
          +' es: '+InTtoStr(D[i]);
```

```
    RESP:=RESP + #13;
    RESP:=RESP+'La Ruta es: '+ ruta(P,R,i,L)+#13;
   end
  else
   begin
    RESP:=RESP+'La distancia del vertice '+IntToStr(ini)+' al vertice '+IntToStr(i)
          +' es infinita';
    RESP:=RESP + #13;
    RESP:=RESP+'No existe Ruta '+#13;
   end;
 end;
 escribir:='CAMINOS MINIMOS DESDE EL NODO INICIO '+IntToStr(ini)+':'+#13+RESP;
end;

procedure TForm1.Edit2KeyPress(Sender: TObject; var Key: Char);
Var
 n:integer;
begin
 If key=#13 then
  begin
  n:=strtoint(Edit2.text);
  Edit3.SetFocus;
  inicializa(StringGrid1,n);
  end;
   if not (key in ['0'..'9',#8])then key:=#0
end;

procedure TForm1.Edit3KeyPress(Sender: TObject; var Key: Char);
begin
 bitbtn3.enabled:=false;
 If key=#13 then Edit4.SetFocus;
  if not (key in ['0'..'9',#8])then key:=#0
end;

procedure TForm1.Edit4KeyPress(Sender: TObject; var Key: Char);
begin
If key=#13 then Edit1.SetFocus;
 if not (key in ['0'..'9',#8])then key:=#0
end;

procedure TForm1.Edit1KeyPress(Sender: TObject; var Key: Char);
begin
 If key=#13 then
 begin
  bitbtn1.Enabled:=true;
  bitbtn1.SetFocus
  end;
 if not (key in ['0'..'9',#8])then key:=#0
```

```
end;

procedure TForm1.Edit5KeyPress(Sender: TObject; var Key: Char);
var
  ini:integer;
begin
  If key=#13 then bitbtn4.enabled:=true;
    if not (key in ['0'..'9',#8])then key:=#0
end;

procedure TForm1.BitBtn1Click(Sender: TObject);
var
  i,j,cont,peso,n:integer;
begin
  try
   n:=strtoint(Edit2.text);
   i:=StrToinT(Edit3.text);
   j:=StrToinT(Edit4.text);
   StringGrid1.cells[j,i]:=edit1.text;
   Edit3.clear;
   Edit4.clear;
   Edit1.Clear;
   Edit3.SetFocus;
   bitbtn1.enabled:=false;
   bitbtn3.Enabled:=true;
  except
       on EconvertError do showmessage('error')
     end;
  end;

procedure TForm1.BitBtn4Click(Sender: TObject);
var
A:matriz;
B:adyacente;
P:Padre;
D:Dist;
Q:Cola;
R:Camino;
L:Lineas;
ini,n,i,f,k,j,l1:integer;
begin
 n:=strtoint(Edit2.text);
 ini:=strtoint(Edit5.text);
 inicializarQ(D,Q,n,ini);
 crear(A,B,stringGrid1,n);
 Leer(A,n);
for i:=1 to n do
  begin
```

```
    extraermin(Q,f,D,ini,n);
    for k:=1 to n do
      begin
         if B[f,k]=1 then
          relajar(A,Q,D,P,f,k);
      end;
  end;

label4.caption:=escribir(D,P,R,ini,n,L);

end;
end.
```

Apéndice B

Implementación del Algoritmo de Bellman-Ford

```
unit Uinicio;
interface
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, Borland.Vcl.StdCtrls, Borland.Vcl.Buttons, System.ComponentModel,
  Borland.Vcl.Grids;
type
  TForm1 = class(TForm)
    GroupBox1: TGroupBox;
    Edit2: TEdit;
    Label1: TLabel;
    BitBtn2: TBitBtn;
    GroupBox3: TGroupBox;
    Label7: TLabel;
    Label3: TLabel;
    Label2: TLabel;
    Edit3: TEdit;
    Edit4: TEdit;
    Edit1: TEdit;
    BitBtn1: TBitBtn;
    StringGrid1: TStringGrid;
    Label5: TLabel;
    GroupBox2: TGroupBox;
    Label6: TLabel;
    Edit5: TEdit;
    BitBtn3: TBitBtn;
    BitBtn4: TBitBtn;
    GroupBox4: TGroupBox;
    Label4: TLabel;
    procedure Edit1KeyPress(Sender: TObject; var Key: Char);
    procedure Edit2KeyPress(Sender: TObject; var Key: Char);
    procedure Edit3KeyPress(Sender: TObject; var Key: Char);
```

```
    procedure Edit4KeyPress(Sender: TObject; var Key: Char);
    procedure BitBtn1Click(Sender: TObject);
    procedure Edit5KeyPress(Sender: TObject; var Key: Char);
    procedure BitBtn3Click(Sender: TObject);
    procedure BitBtn4Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
implementation
{$R *.nfm}
type
 adyacente=array [1..100,1..100] of integer;
 matriz=array [1..100,1..100]of integer;
 Dist=array [1..100] of integer;
 Padre=array [1..50] of integer;
 Camino=array [1..50] of integer;

procedure inicializa(Rejilla:TstringGrid; n:integer);
var
 w,k:integer ;
 begin
  for w:=1 to n do
   for k:=1 to n do
    Rejilla.Cells[w,k]:=IntToStr(0)
 end;

procedure crear( var A:matriz; var B:adyacente; Rejilla:TstringGrid; n:integer);
 var
  i,j:integer;
 begin
  for i:=1 to n do
   for j:=1 to n do
     begin
      A[i,j]:=StrToint(Rejilla.cells[j,i]);
             if (i=1) and (j>1) then
                B[i,j]:=1;
       if A[i,j]<>0 then
        B[i,j]:=1;
     end;
 end;

procedure LEER(var A:matriz; var n:integer);
var
```

```
i,j:integer;
Mat:string;
begin
 for i:=1 to n do
   begin
   for j:=1 to n do
      Mat:=Mat+IntToStr(A[i,j])+#9;
    Mat:=Mat+#13;
    end;
   ShowMessage('LA MATRIZ DE PESOS ES:'+#13+Mat);
end;

procedure inicializarD( var D:Dist; n,ini:integer);
const
max=9999;
var
i:integer;
begin
 for i:=1 to n do
    D[i]:=max;
D[ini]:=0;
end;

procedure relajar(var A:matriz; var D:Dist; var P:Padre; i,j:integer);
begin
   if D[j] > D[i]+ A[i,j] then
     begin
      D[j]:= D[i]+ A[i,j];
      P[j]:=i;
    end;
end;

function ruta(var P:Padre;  R:Camino; i:integer):string;
var
j,n:integer;
cam:string;
begin
    j:=1;
    while P[i]>0 do
       begin
         R[j]:=i;
         i:=P[i];
         j:=j+1;
       end;
   R[j]:=i;
   for n:=j downto 1 do
    Cam:=Cam + IntToStr(R[n])+' ';
    Ruta:=Cam;
```

```
 end;

function escribir( var D:Dist ;var P:Padre; var R:Camino; ini,n:integer):string;
var
i:integer;
RESP:STRING;
begin
for i:=1 to n do
 begin
  if D[i]<9999 then
   RESP:=RESP+'La distancia del vertice '+IntToStr(ini)+' al vertice '
   +InTtoStr(i)+' es: '+InTtoStr(D[i])
  else
   RESP:=RESP+'La distancia del vertice '+IntToStr(ini)+' al vertice '
   +IntToStr(i)+' es infinita';
 RESP:=RESP + #13;
 RESP:=RESP+'La Ruta es: '+ ruta(P,R,i)+#13;
 end;
 escribir:='CAMINOS MINIMOS DESDE EL NODO INICIO '+IntToStr(ini)+':'+#13+RESP;
 end;

procedure TForm1.Edit2KeyPress(Sender: TObject; var Key: Char);
Var
 n:integer;
begin
 If key=#13 then
  begin
  n:=strtoint(Edit2.text);
  Edit3.SetFocus;
  inicializa(StringGrid1,n);
  end;
   if not (key in ['0'..'9',#8])then key:=#0
end;

procedure TForm1.Edit3KeyPress(Sender: TObject; var Key: Char);
begin
 bitbtn3.enabled:=false;
 If key=#13 then Edit4.SetFocus;
  if not (key in ['0'..'9',#8])then key:=#0
end;

procedure TForm1.Edit4KeyPress(Sender: TObject; var Key: Char);
begin
If key=#13 then Edit1.SetFocus;
 if not (key in ['0'..'9',#8])then key:=#0
end;

procedure TForm1.Edit1KeyPress(Sender: TObject; var Key: Char);
```

```
begin
 If key=#13 then
 begin
  bitbtn1.Enabled:=true;
  bitbtn1.SetFocus
  end;
 if not (key in ['0'..'9','-',#8])then key:=#0
end;

procedure TForm1.Edit5KeyPress(Sender: TObject; var Key: Char);
var
  ini:integer;
begin
  If key=#13 then bitbtn4.enabled:=true;
    if not (key in ['0'..'9',#8])then key:=#0
end;

procedure TForm1.BitBtn1Click(Sender: TObject);
var
  i,j,cont,peso,n:integer;
begin
  try
   n:=strtoint(Edit2.text);
   i:=StrToinT(Edit3.text);
   j:=StrToinT(Edit4.text);
   StringGrid1.cells[j,i]:=edit1.text;
   Edit3.clear;
   Edit4.clear;
   Edit1.Clear;
   Edit3.SetFocus;
   bitbtn1.enabled:=false;
   bitbtn3.Enabled:=true;
  except
       on EconvertError do showmessage('error')
     end;
  end;

procedure TForm1.BitBtn3Click(Sender: TObject);
var
A:matriz;
B:adyacente;
n:integer;
begin
Edit5.SetFocus;
bitbtn3.enabled:=false;
end;

procedure TForm1.BitBtn4Click(Sender: TObject);
```

```
var
A:matriz;
B:adyacente;
P:Padre;
D:Dist;
R:Camino;
ini,n,i,boolean,j,k:integer;
begin
 n:=strtoint(Edit2.text);
 ini:=strtoint(Edit5.text);
 crear(A,B,stringGrid1,n);
 Leer(A,n);
 inicializarD(D,n,ini);
 for k:=1 to n-1 do
  begin
     for i:=1 to n do
      begin
         for j:=1 to n do
         begin
           if B[i,j]=1 then
           relajar(A,D,P,i,j)
         end;
      end;
   end;
boolean:=0;
for i:=1 to n do
 begin
   for j:=1 to n do
      begin
       if A[i,j]<>0 then
          if d[j]>d[i]+A[i,j] then
               boolean :=boolean + 1
       end;
   end;
if boolean=1 then
  ShowMessage('NO EXISTE SOLUCION por ciclo de pesos negativos')
else
   begin
    showmessage('EXISTE SOLUCION');
    label4.caption:= escribir(D,P,R,ini,n);
   end;
  end;
end.
```

Bibliografía

[1] Thomas H.Cormen, Charles E. Leiserson, Ronald L.Rivest: INTRODUCTION TO ALGORITHMS. Editorial Mc.Graw-Hill

[2] José Luis Chacón: MATEMÁTICA DISCRETA. Grupo editorial Iberoamericana.

[3] Elizabeth Pérez, Rene Kinney: ANÁLISIS DE ALGORITMOS. Editorial Universidad Autónoma Metropolitana

[4] Harry R.Lewis, Christos H. Papadimitriou : ELEMENTS OF THE THEORY OF COMPUTATION. Editorial Pretence-Hall

[5] J.A. Boundy, U.S.R. Murty : GRAPH THEORY WITH APPLICATIONS. Editorial North Holland

[6] Andreas Polyméris : LA CLASE DE LOS PROBLEMAS NP. DIICC. Universidad de California. 2004

[7] Cormen, Leisernon y Rivest : INTRODUCTION TO THE ALGORITHMS. 2da Edición 2001

[8] Gloria Sánchez Torrubia, Victor Lozano Terrazas: ALGORITMO DE DIJKSTRA. Madrid. Departamento de informática. Universidad Politécnica de Madrid

[9] E. W. Dijkstra: A NOTE ON TWO PROBLEMS IN CONECTION WITH GRAPHS. Numerische Mathematik, vol. 1.

[10] Michael Sipser: INTRODUCTION TO THE THEORY OF COMPUTATION. Editorial PWS Publishing company. 1997.

[11] Reinhard Diestel: GRAPH THEORY. Editorial Springer-Verlag.

[12] Lane A. Hemaspaandra : SIGACT NEWS COMPLEXITY THEORY. Departamento de Ciencias de la Computación. Universidad de Rochester.

[13] Steven S. Skiena, Miguel A. Revilla: PROGRAMMING CHALLENGES. Editorial Springer.

[14] Jayme Luiz Szwarcfiter, Lilian Markenson: ESTRUCTURA DE DADOS E SEUS ALGORITMOS. Editorial L.T.C - Libros Técnicos y Científicos.

[15] Claudio l. Lucchesi, Imre Simon, Istvan Simon, Janos Simon, Tomasz Kowasltowski: ASPECTOS TEÓRICOS DE LA COMPUTACIÓN. Editorial L.T.C - Libros Técnicos y Científicos.

[16] Richard Johnsonbaugh: MATEMÁTICA DISCRETA. Editorial Iberoamericana.

[17] Ingo Wegener: COMPLEXITY THEORY - EXPLORING THE LIMITS OF EFFICIENT ALGORITHMS. Editorial Springer.

[18] Dexter C. Kozen: THEORY OF COMPUTATION. Editorial Springer

[19] Ravindra K. Ahuja, Thomas L. Magnanti, James B. Orlin: NETWORK FLOWS. Editorial Prentice Hall