Unidad 7

Grafos Dirigidos

Un grafo dirigido G consiste en un conjunto de vértices (nodos o puntos) V y un conjunto de arcos A (arcos dirigidos o líneas dirigidas).

Un arco es un par ordenado de vértices (v, w) donde v es cola y w cabeza del arco.

W

V

El arco v --> w va de v a w y w es adyacente a v.

Los vértices de un grafo dirigido pueden usarse para representar objetos, y los arcos, relaciones entre objetos. Por ejemplo, los vértices pueden representar ciudades, y los arcos, vuelos aéreos de una ciudad a otra. En otro ejemplo, como el que se presentó en la sección 4.2, un grafo dirigido puede emplearse para representar el flujo de control en un programa de computador. Los vértices representar bloques básicos, y los arcos, posibles transferencias del flujo de control.

Un camino en un grafo dirigido es una secuencia de vértices v1, v2, …, vn tal que v1 -> v2, v2 -> v3, …, vn-1 -> vn son arcos. Este camino va del vértice v1 al vértice vn, pasa por los vértices v2, v3, …, vn-1 y termina en el vértice vn. La longitud de un camino es el número de arcos en ese camino, en este caso n-1. Un vértice solo tiene longitud de camino 0.

**Camino simple** -> si todos sus vértices, menos el primero y el último son distintos.

**Ciclo simple** -> camino simple de longitud por lo menos 1, que empieza y termina en el mismo vértice.

**Grafo dirigido etiquetado** -> cada arco, vértice o ambos tienen una etiqueta asociada. Una etiqueta puede ser un nombre, un costo o un valor de cualquier tipo de datos dado. En un grafo dirigido etiquetado, un vértice puede tener a la vez un nombre y una etiqueta. A menudo se empleará la etiqueta del vértice como si fuera el nombre.

**Representación común para un grafo dirigido:** matriz de adyacencia. Suponiendo que V = { 1, 2, …, n }, la matriz de adyacencia para G es una matriz A de dimensión n x n de booleanos, donde A[i, j] es verdadero si existe un arco que vaya de i a j. El tiempo de acceso requerido a un elemento es independiente del tamaño de V y A. Así, la representación con matriz de adyacencia es útil en los algoritmos para grafos, en los cuales suele ser necesario saber si un arco dado esta presente.

**Matriz de adyacencia etiquetada:** A[i, j] es la etiqueta del arco que va del vértice **i** al vértice **j**. Si no hay arco de i a j se debe emplear como entrada un valor que no pueda ser una etiqueta válida. La principal desventaja de una matriz de adyacencia es que requiere un espacio Ω(n2) aún si el grafo tiene menos de n2 arcos. Leer o examinar lleva tiempo O(n2) lo cual invalida los algoritmos O(n) de manipulación del grafo.

Para evitar esta desventaja, se puede utilizar otra representación común para un grafo dirigido llamada representación con lista de adyacencia.

**Representación con lista de adyacencia:** La lista de adyacencia de un vértice i, es una lista (sin orden) con todos los vértices adyacentes a i. Se puede representar G por medio de un arreglo CABEZA donde CABEZA[i] es un apuntador a la lista de adyacencia del vértice i. Requiere un espacio proporcional a la suma de los vértices más la suma de los arcos. Se usa bastante cuando el número de arcos es menor a n2. Una desventaja es que puede llevar un tiempo O(n) determinar si existe un arco de i a j.

**Problema de los caminos más cortos de un solo origen (Algoritmo de Dijkstra).**

Se considera un problema común de búsqueda de caminos en grafos dirigidos. Supóngase un grafo dirigido G = (V, A) en el cual cada arco tiene una etiqueta no negativa, y donde un vértice se especifica como origen. El problema es determinar el costo del camino más corto del origen a todos los demás vértices de V, donde la longitud de un camino es la suma de los costos de los arcos del camino.

Sea G un mapa de vuelos en el cual cada vértice representa una ciudad, y cada arco v -> w, una ruta aérea de la ciudad v a la ciudad w. La etiqueta del arco v -> w es el tiempo que se requiere para volar de v a w. La solución del problema de los caminos más cortos con un solo origen para este grafo dirigido determinaría el tiempo de viaje mínimo para ir de cierta ciudad a todas las demás del mapa.

Para resolver este problema, se manejará una técnica <<ávida>> conocida como algoritmo de Dijkstra, que opera a partir de un conjunto S de vértices cuya distancia más corta desde el origen ya es conocida. En principio, S contiene sólo el vértice de origen. En cada paso, se agrega algún vértice restante v a S, cuya distancia desde el origen es la más corta posible. Suponiendo que todos los arcos tienen costo no negativo, siempre es posible encontrar un camino más corto entre el origen y v que pasa sólo a través de los vértices de S, y que se llama especial. En cada paso del algoritmo, se utiliza un arreglo D para registrar la longitud del camino especial más corto a cada vértice. Una vez que S incluye todos los vértices, todos los caminos son <<especiales>>, así que D contendrá la distancia más corta del origen a cada vértice.

El algoritmo se da en la figura 6.8, donde se supone que existe un grafo dirigido G = (V, A) en el que V = { 1, 2, …, n } y el vértice 1 es el origen. C es un arreglo bidimensional de costos, donde C[i, j] es el costo de ir de vértice i al vértice j por el arco i -> j. Si no existe el arco i -> j, se supone que C[i, j] es ∞, un valor mucho mayor que cualquier costo real. En cada paso, D[i] contiene la longitud del camino especial más corto actual para el vértice i.

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

Imagen que contiene objeto, reloj

Descripción generada automáticamente

Tabla

Descripción generada automáticamente

**Por qué funciona el algoritmo de Dijkstra**

El algoritmo de Dijkstra es un ejemplo donde la <<avidez>> funciona, en el sentido de que lo que aparece localmente como lo mejor, se convierte en lo mejor de todo. En este caso, lo <<mejor>> localmente es encontrar la distancia al vértice w que está fuera de S, pero tiene el camino especial más corto.

**Tiempo de ejecución del algoritmo de Dijkstra**

El tiempo del algoritmo es O(n2), si a es mucho menor que n2 entonces el tiempo es O(alogn).

**Problema del camino más corto entre todos los pares (RW. Floyd)**

Supóngase que se tiene un grafo dirigido etiquetado que da el tiempo de vuelo para ciertas rutas entre ciudades, y se desea construir una tabla que brinde el menor tiempo requerido para volar entre dos ciudades cualesquiera. Este es un ejemplo del problema de los caminos más cortos entre todos los pares (CMCP). Para plantear el problema con precisión, se emplea un grafo dirigido G = (V, A) en el cual cada arco v -> w tiene un costo no negativo C[v, w]. El problema CMCP es encontrar el camino de longitud más corta entre v y w para cada par ordenado de vértices (v, w). (Podría resolverse por Dijkstra, tomando por turno cada vértice).

Una forma más directa de solución es mediante el algoritmo creado por R. W. Floyd. Por conveniencia, se supone otra vez que los vértices están numerados 1, 2, …, n. El algoritmo de Floyd usa una matriz de A de n x n en la que se calculan las longitudes de los caminos más cortos. Inicialmente se hace A[i, j] = C[i, j] para toda i != j. Si no existe un arco que vaya de i a j, se supone que C[i, j] = ∞. Cada elemento de la diagonal se hace 0.

Después, se hacen n iteraciones en la matriz A. Al final de la k-ésima iteración, A[i, j] tendrá por valor la longitud más pequeña de cualquier camino que vaya desde el vértice i hasta el vértice j y que no pase por un vértice con número mayor que k. Esto es, i y j, los vértices extremos del camino, pueden ser cualquier vértice, pero todo vértice intermedio debe ser menor o igual que k.

En la k-ésima iteración se aplica la siguiente fórmula para calcular A.

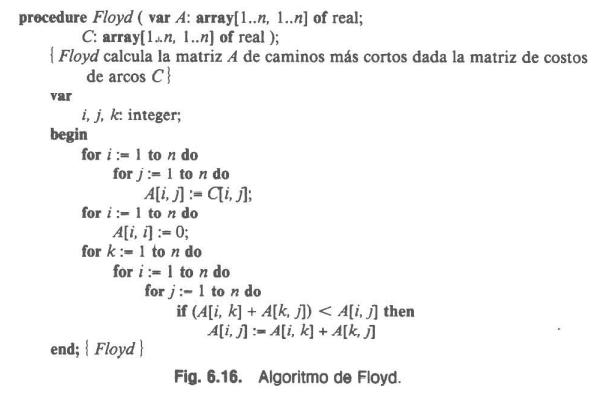
Ak[i, j] = min { Ak – 1[i, j]

Ak – 1[i, k] + Ak – 1[k, j] }

El subíndice k denota el valor de la matriz A después de la k-ésima iteración; no indica la existencia de n matrices distintas. Pronto, se eliminarán esos subíndices.

Para obtener Ak[i, j], se compara Ak-1[i, j], el costo de ir de i a j sin pasar por k o cualquier otro vértice con numeración mayor, con Ak-1[i, k] + Ak-1[k, j], el costo de ir primero de i a k y después de k a j, sin pasar a través de un vértice con número mayor que k. Si el paso por el vértice k produce un camino más económico que el de Ak-1[i, j], se elige ese costo para Ak[i, j].

El tiempo de ejecución del algoritmo es O(n3).



**Comparación entre los algoritmos de Floyd y Dijkstra**

Dado que la versión de Dijkstra con matriz de adyacencia puede encontrar los caminos más cortos desde un vértice en un tiempo O(n2), como el algoritmo de Floyd, también puede encontrar todos los caminos más cortos en un tiempo O(n3). El compilador, la máquina y los detalles de realización determinarán las contantes de proporcionalidad. La experimentación y medición son la forma más fácil de descubrir el mejor algoritmo para la aplicación en cuestión.

Si a, el número de aristas, es mucho menor que n2, aún con el factor constante relativamente bajo en el tiempo de ejecución O(n3) de Floyd, cabe esperar que la versión de Dijkstra con lista de adyacencia, tomando un tiempo O(na logn) para resolver el CMCP, sea superior, al menos para grafos grandes y poco densos.

**Recuperación de los caminos**

En muchos casos se desea imprimir el camino más económico entre dos vértices. Un modo de lograrlo es usando otra matriz P, donde P[i, j] tiene el vértice k que permitió a Floyd encontrar el valor más pequeño de A[i, j]. Si P[i, j] = 0, el camino más corto de i a j es directo, siguiendo el arco entre ambos. La versión modificada de Floyd de la figura 6.17 almacena los vértices intermedios apropiados en P.

Texto, Carta

Descripción generada automáticamente

Para imprimir los vértices intermedios del camino más corto del vértice i hasta el vértice j, se invoca el procedimiento camino(i, j) dado en la figura 6.18.

Texto

Descripción generada automáticamente

**Cerradura transitiva (Algoritmo Warshall)**

En algunos problemas podría ser interesante saber si existe un camino de longitud igual o mayor que uno que vaya de i a j.

Suponga que la matriz C es una matriz de adyacencia. Esto es C[i, j] = 1 si hay un arco de i a j, y 0 si no lo hay. Se desea obtener la matriz A tal que A[i, j] = 1 si hay un camino de longitud igual o mayor que uno de i a j y 0 en otro caso. A se conoce a menudo como cerradura transitiva de la matriz de adyacencia.

Se conoce como cerradura transitiva de la matriz de adyacencia. La cerradura transitiva puede obtenerse con un procedimiento similar a Floyd aplicando la siguiente fórmula en el k-ésimo paso en la matriz booleana A.

Ak[i, j] = Ak-1[i, j] o (Ak-1[i, k] y Ak-1[k, j])

Esta fórmula establece que hay un camino de i a j que no pasa por un vértice con número mayor que k si:

* Ya existe un camino de i a j que no pasa por un vértice con número mayor que k – 1, o si
* Hay un camino de i a k que no pasa por un vértice con número mayor que k – 1, hay un camino de k a j que no pasa por un vértice con número mayor que k – 1.

Texto, Carta

Descripción generada automáticamente

El centro de un grafo es un vértice con mínima excentricidad (vértice más cercano al vértice más distante).

**Búsqueda en profundidad**

Se selecciona un vértice de partida (se marca como visitado), después se recorre cada vértice adyacente de forma recursiva. Una vez visitados todos los vértices la búsqueda queda hecha. Va buscando en dirección hacia adelante (más profunda) mientras sea posible.

Texto

Descripción generada automáticamente

Todas las llamadas a bpf en la búsqueda en profundidad de un grafo con a arcos y n <= a vértices llevan un tiempo O(a):

* No se llama bpf en ningún vértice más de una vez.
* El tiempo consumido en las líneas 2 y 3 es proporcional a la suma de las longitudes de las listas, O(a)

Entonces, el tiempo total de la bpf de un grafo completo es O(a), o sea, el tiempo necesario para recorrer cada arco.

Los arcos que llevan a vértices sin visitar se conocen como **arcos de árbol** y forman un bosque abarcador en profundidad.

**Arco de retroceso ->** arco que lleva a un vértice antecesor.

**Arco de avance ->** va de un vértice a un descendiente propio.

**Arcos cruzados ->** van de un vértice a otro que no es antecesor ni descendiente.

**Identificación de los tipos de arco:**

Numerar en profundidad.

Si el arco es de **avance**, va de un vértice de baja numeración a uno de alta numeración (que ya fue visitado).

Si es de retroceso, a la inversa (y el destino es un ancestro).

Los arcos cruzados van de alta numeración a baja numeración (pero el destino no es un ancestro).

W es un descendiente de v si y sólo si,

Num(v) < Num(w) <= Num(v) + Cantidad de descendientes de v

**Obtención de un camino**

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

**Formas de recuperar caminos:**

El algoritmo presentado es una propuesta que admite variantes, por ejemplo en qué lugar verificar si se llega al Destino. Continua realizando una “bpf” luego de haber encontrado un camino.

“El camino” que se define, en realidad es una colección de vértices que se maneja con disciplina LIFO, y que contiene todos los vértices que todavía están pendientes en la recursión.

**Grafo dirigido acíclico** -> (es un grafo dirigido sin ciclos) -> Son más generales que los árboles pero menos que los grafos dirigidos arbitrarios.

Útiles para representar expresiones aritméticas con subexpresiones comunes.

También son apropiados para representar órdenes parciales.

Se realiza la búsqueda en profundidad y si se encuentra un arco de retroceso, el grafo tiene un ciclo.

Si un grafo dirigido tiene un ciclo, siempre habrá un arco de retroceso en la búsqueda en profundidad.