Algoritmos de
GRAFOS

Universidad de Murcia
2º Grado en Ingeniería Informática
Asignatura: Algoritmos y estructuras de datos I
Curso 2017/2018

Proyecto realizado por Nicolás Enrique Linares La Barba.

Índice

PROBLEMA 402

Análisis

El problema 402 consiste en realizar un **algoritmo de búsqueda en anchura** sobre los grafos especificados. Este recorrido en anchura aplicado a un grafo es similar a un recorrido por niveles en un árbol, donde en primer lugar se visita la raíz, a continuación, todos sus hijos, después los nietos, etc. Hay que destacar la diferencia entre árbol y grafo, la cual radica en que el primero comienza a partir de un nodo raíz, es decir, un nodo que tiene la función de iniciar los procedimientos, pero en el segundo cualquier nodo puede hacer de nodo raíz.

Descripción abstracta de la resolución

Mediante este algoritmo visitaremos todos los nodos del grafo tomando las aristas como elementos de unión entre estos. Las aristas podrán ser dirigidas o no, sin provocar ningún cambio en el algoritmo, y el grafo podrá ser fuertemente conexo (en el caso de grafos dirigidos), conexo (en grafos no dirigidos) o por el contrario no serlo, puesto que el algoritmo permite recorrer todos los nodos, estén unidos entre ellos o no. Además, los recorridos dependerán en gran medida del nodo por el que se empiece, por ello los recorridos no son únicos.

Implementación

El algoritmo se divide en dos procedimientos, uno principal y otro recursivo:

- El procedimiento principal se encargará de comenzar la ejecución del recorrido, para ello inicializa un array de tamaño N (nodos del grafo) a false, el cual servirá para tener constancia de qué nodos han sido visitados (true), posteriormente, será recorrido mediante un bucle *for* por cada nodo que no haya sido recorrido.

- El procedimiento recursivo, marcará como visitado el vértice que se le pasa como parámetro y pondrá a la cola a todos aquellos nodos adyacentes. A partir de aquí irá sacando vértices de la cola, marcándolos como visitados y añadiendo nuevos a la cola en caso de que este tenga adyacentes. Cada elemento que se extraiga de la cola será mostrado por pantalla.

El orden utilizado a la hora de seleccionar nodos será alfabéticamente, empezando el recorrido por el nodo A y siguiendo según los nodos que quedan por visitar, también en orden alfabético.

Eficiencia

El tiempo de ejecución del recorrido en anchura viene determinado por el tiempo que se tarda en visitar todos los nodos una vez, para una implementación con matriz de adyacencia es de Orden (n^2) puesto que se recorre toda la matriz, por otro lado, si utilizamos listas de adyacencias sería de Orden (n + a) por recorrer todas las listas. Utilizando la orden *time* en el terminal de Ubuntu obtenemos el siguiente tiempo de ejecución:

real 0m0.006s user 0m0.004s sys 0m0.000s

PROBLEMA 404

Análisis

El problema 404 es un tipo de problema de caminos mínimos empezando por un origen, donde las aristas del grafo tendrán asociado un peso o coste y, por tanto, el camino mínimo será aquel camino de menor coste, formado por la suma de las aristas recorridas desde un vértice inicial hasta otro final; se puede dar el caso de que el nodo al que queremos llegar no se encuentre conexo y sea imposible llegar hasta él.

Descripción abstracta de la resolución

El ejercicio ha sido resuelto mediante el algoritmo de ***Edsger Wybe Dijkstra,*** cuya idea general consiste en tener un nodo inicial, de partida, para el que calcularemos todos los caminos mínimos desde ese vértice hasta el resto de los vértices del grafo. Como el problema nos pide encontrar el camino mínimo entre dos vértices, pero pasando por uno intermedio, ha sido planteado de la siguiente manera: se calcula en primer lugar el camino mínimo entre el nodo inicial (A) y el intermedio (C) para el cual si es posible añade la distancia a una variable global llamada *resultadoCamino*, y el procedimiento retornará *true* confirmando que ha resultado con éxito y que puede continuar con la segunda llamada. En esta segunda parte se vuelve a llamar al algoritmo, pero pasándole como parámetros el nodo intermedio (C) y el de llegada (B), igual que en el caso anterior. Si también devuelve *true* el resultado estará en la variable global, pero si en uno de los casos anteriores el algoritmo retorna *false* querrá decir que es imposible llegar a uno de los vértices.

Implementación

En cuanto a la implementación, primero leemos el grafo de la entrada y lo guardamos en la matriz de costes los pesos de todos los pares, siendo 0 cuando no exista arista. Para *Dijkstra* se hará uso de dos arrays locales: *D* de tipo entero para los costes de caminos mínimos y *T* de tipo booleano para marcar a los visitados. *D* es inicializada con la fila de la matriz de costes del vértice inicial, donde por cada iteración del bucle principal se buscará primero el siguiente vértice a visitar, que será el adyacente con menor coste y que no haya sido visitado ya (comprobándolo en el array *T*) y, a continuación, se recalculan todos los caminos por si hubiese uno más corto. Como condición final, compruebo si el vértice origen es igual al de llegada, lo que supondrá un 0 en su posición del array D (la distancia a sí mismo), o que el vértice de llegada no tenga un 0 en su posición del array D, lo que significará que ha sido visitado. En caso contrario se devuelve falso porque ha sido imposible calcular la distancia (0 en su posición del array D).

Eficiencia

El problema ha sido resuelto con una matriz y los arrays *D* y *T,* explicados anteriormente, lo que supondrá un Orden (n^2) por recorrer todas las aristas del grafo. En caso de hacerlo con listas obtendríamos un Orden ($n + a + n^2$) = Orden (n^2). Utilizando la orden *time* obtenemos el siguiente tiempo de ejecución:

real 0m0.120s user 0m0.112s sys 0m0.004s

PROBLEMA 407

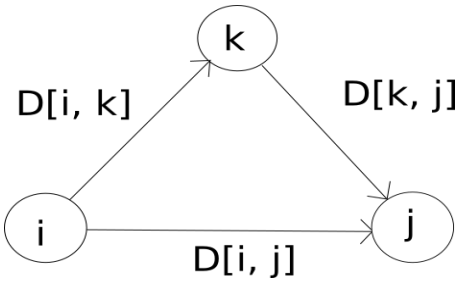
Análisis

El problema 407 es un tipo de problema de caminos mínimos entre todos los vértices que se puede resolver con el algoritmo de Floyd, que encuentra una solución más directa, al igual que con el algoritmo de Dijkstra, que para este caso deberemos repetirlo una vez por cada nodo del grafo. El tipo de grafo del problema es conexo, no dirigido y etiquetado.

Descripción abstracta de la resolución

Se ha utilizado el algoritmo de ***Robert W. Floyd****, que* nos permitirá encontrar los caminos más cortos entre todos los pares de vértices, así como localizar aquella distancia mínima (el centro del grafo) de entre todas las distancias máximas que hay entre todos los vértices (excentricidades). Se define la excentricidad en un vértice como la mayor distancia de entre todos los caminos posibles al resto de nodos del grafo. El centro del grafo será la mínima excentricidad.

La idea consiste en pivotar por cada nodo k del grafo encontrando la mínima distancia entre nodos y comprobando en cada caso si debemos pasar por el nodo k o no. Pasaremos por el nodo k si supone un menor coste que pasar desde i hasta j, es decir, si D[i,k] + D[k,j] < D[i,j] entonces pasaremos por k.



Implementación

Se hará uso de una matriz *D* de costes que se irá actualizando a medida que se recorren y se calculan todas las distancias. En el algoritmo de Floyd inicializamos primero la matriz D con un valor muy alto (simulando un infinito), y establecemos los costes iniciales en el primer bucle for. A continuación, comienza el pivotaje en el siguiente bucle, mencionado en el apartado anterior.

Para hacer el código más legible he implementado un procedimiento adicional llamado *calcularExcentricidad*, que se ocupará en el bucle interno de obtener la mayor distancia de cada fila (correspondiente a cada nodo) y en el bucle externo se quedará con la menor de ellas, realizando la comparación alfabética en caso de igualdad. Finalmente muestra el resultado por pantalla.

Eficiencia

El algoritmo supone un orden de complejidad de Orden(n^3) por el proceso de pivotaje para calcular la distancia mínima entre cada par de nodos, ya que el resto de casos solo es recorrer la matriz una vez, Orden(n^2). Utilizando la orden *time* obtenemos el siguiente tiempo de ejecución:

real 0m0.138s user 0m0.136s sys 0m0.000s

PROBLEMA 411

Análisis

El problema 411 se resuelve de manera trivial utilizando la búsqueda primero en profundidad o la búsqueda primero en anchura, ambas son útiles para este problema puesto que va recorriendo el grafo comenzando por un nodo y continuando por los adyacentes de manera recursiva.

Descripción abstracta de la resolución

La idea se ha basado en el algoritmo de búsqueda en profundidad, el cual tiene un procedimiento principal (*busquedaPP*) que hace de arranque para la llamada recursiva al segundo procedimiento (*bpp*) , que se encarga de recorrer los vértices adyacentes al primer vértice que se le pasa como parámetro. De esta manera, sabemos que cuando haya terminado la recursión ya no quedarán vértices adyacentes y, por tanto, o se ha acabado de recorrer el grafo y solo tiene una isla (grafo conexo) o, por el contrario, todavía quedan vértices que no han sido visitados y que deberemos recorrer.

Implementación

En cuanto a la implementación solo es necesario un contador de islas, donde cada vez que iniciemos la llamada recursiva se incrementará una unidad, y un array de enteros (*procedencia*) para saber a qué isla pertenece cada persona, el procedimiento recursivo irá marcando en la posición de la persona, que está siendo visitada, con el número de isla que se está recorriendo en ese instante.

Eficiencia

El tiempo de ejecución no cambia con la modificación del algoritmo, sigue siendo de Orden (n^2) utilizando matrices de adyacencia y de Orden (n + a) con listas de adyacencia. Con la orden *time* se obtienen los siguientes tiempos:

real 0m0.039s user 0m0.020s sys 0m0.016s

PROBLEMA 413

Análisis

El problema 413 se resuelve recorriendo todo el grafo para buscar aquellas aristas entre nodos con coste mínimo. Se hace uso del concepto de **árbol de expansión de coste mínimo**, debido a la semejanza que existe entre un árbol y un grafo no dirigido, conexo y sin ciclos. Este tipo de problemas se pueden resolver con el algoritmo de Prim o de Kruskal, ambos obtendrían la solución, pero utilizarían un recorrido diferente, lo que no afectaría en este caso.

Descripción abstracta de la resolución

Ha sido resuelto con el **algoritmo de R. C. Prim**, el cual recorre todo el grafo nodo a nodo apuntando aquellas aristas de menor coste y formando un árbol de expansión con todos los nodos donde el coste total de las aristas que componen el árbol es el mínimo posible. Hay que destacar que la ejecución con éxito de este algoritmo precisa que todos los vértices se encuentren unidos entre sí.

Implementación

En este problema nos proporcionan las coordenadas de las colonias en un plano, por ello he decidido utilizar una estructura *Colonia* para agrupar la coordenada x e y. Estas *Colonias* son guardadas inicialmente en la lectura de datos en un array dinámico para a continuación calcular, con la fórmula mencionada en el ejercicio, todas las distancias (aristas) entre un nodo con el resto de los nodos del grafo, porque necesitamos todas las aristas que pueda contener, y finalmente borramos con delete[] el array dinámico. De esta manera se rellena la matriz de costes que utilizaremos en el algoritmo de Prim.

El algoritmo comienza tomando un vértice *k*, en este caso siempre empieza por el primero, y compara todas las distancias hacia el resto de vértices para quedarse con el menor peso y con el vértice que une esa arista (primer bucle del bucle principal). Después se guarda ese coste y comienza otro bucle para obtener la menor arista, que no haya sido escogida ya, de los vértices visitados.

La solución del problema, es decir, la distancia total de las aristas mínimas va a ser guardada en una variable *solucion* que se irá incrementando según se calcule la distancia en cada iteración del bucle principal, ya que una vez escogida la arista no se modifica esa elección.

Eficiencia

El tiempo de ejecución del algoritmo requiere un orden total de Orden(n^2) ya sea utilizando matrices de adyacencia o listas, debido al recorrido del bucle principal por la matriz de costes. Con la orden *time* se obtienen los siguientes tiempos:

real 0m0.260s user 0m0.252s sys 0m0.004s

Conclusión

A mi parecer, en cuanto al uso de matrices de adyacencias o listas de adyacencias, he visto más simple el uso de matrices, aunque a veces es cierto que se desperdicia más memoria en tiempo de ejecución por tener espacios que no se usan, su utilización es más clara a la hora de programar y, sobre todo, a la hora de acceder a una posición directamente gracias a los índices. En cuanto a la eficiencia tanto las matrices como las listas ofrecen un mismo orden de ejecución, con muy poca diferencia en algunos casos.

La teoría de grafos ofrece interesantes técnicas de resolución de problemas mediante algoritmos que hemos podido estudiar a lo largo del tema y aplicar en estos ejercicios, siendo capaz de utilizar estas herramientas para transformar problemas reales en problemas que pueden ser abarcados con la implementación y el estudio de grafos.