Documentación actividad tema 4

Jaime Sánchez Sánchez

48754213D

Subgrupo 2.3

27 de diciembre de 2020

|  |  |  |  |
| --- | --- | --- | --- |
| Número de ejercicio | Número de envío | Algoritmo | Representación |
| 402 | 1866 | Búsqueda primero en anchura | Matrices |
| 403 | 1867 | Búsqueda primero en profundidad | Listas |
| 404 | 1868 | Dijkstra | Matrices |
| 407 | 2533 | Floyd | Matrices |
| 411 | 2484 | Búsqueda en profundidad | Listas |
| 412 | 1872 | Dijkstra (modificado) | Matrices |
| 413 | 2534 | Prim | Matrices |
| 418 | 2535 | Búsqueda primero en profundidad | Matrices |

# TABLA DE ENVÍOS

Como podemos ver en la tabla la gran mayoría de ejercicios han sido resueltos mediante una representación usando matrices de adyacencia.

Dependiendo del ejercicio podemos encontrar unos motivos u otros, los principales son que la representación mediante listas es ligeramente más difícil de implementar que una representación mediante arrays y que haciendo uso de la memoria dinámica podemos paliar uno de los principales problemas de la representación mediante arrays que es el malgasto de memoria.

Por otra parte, en la mayor parte de los ejercicios encontramos un número de aristas suficientemente significativo como para decantarnos por los arrays.

En cuanto a los algortimos escogidos, podemos ver que se han usado tanto bpp como bpa. A la hora de encontrar un camino mínimo normalmente he escogido Dijkstra sobre Floyd por varias razones. Entre ellas que es más eficiente, para el 412 solo necesitaba hacerle una pequeña modificación y que, a mi parecer, Dijkstra es mucho más fácil de entender que el algoritmo de Floyd. En cuanto a los árboles de expansión he escogido Prim, al ser el algoritmo que mejor recordaba de la asignatura Aritmética y Matemática Discreta (AMD).

Las particularidades de cada ejercicio se detallarán más adelante.

# Problemas resueltos

A continuación, se detallarán los pasos a seguir en cada problema, además de un rápido análisis de la eficiencia de los programas y los principales problemas que me encontré en cada uno.

**402**

**-Análisis:**

A pesar de ser el ejercicio más fácil de todos también es el primero que se suele hacer de grafos por lo que cuesta más de lo que en principio debería.

El ejercicio trata sobre hacer un recorrido en profundidad en un grafo en donde los nodos son letras del abecedario. La complicación inicial es cómo hacer que la matriz (o la lista) representa a las letras y sus conexiones pero esto se soluciona con una simple operación aritmética en donde la letra A debe ir en primer lugar y la Z en el último.

Elegí matrices porque el número de nodos máximos tampoco era tan grande por lo que el malgasto de memoria no sería demasiado. Por otra parte, al ser el primer ejercicio me resultaba más cómodo realizarlo mediante arrays. A partir de aquí y, apoyándome en la información que tenemos en las transparencias, programé el algoritmo requerido.

En cuanto a la eficiencia, la inicialización del array dependía directamente del número de aristas y el coste de un recorrido en anchura es de O().

En general se trata de un buen ejercicio de grafos que sirve a modo de introducción.

# **403**

**-Análisis:**

Este ejercicio se basa en buscar la salida de un laberinto aplicando teoría de grafos. Aunque el ejercicio sigue siendo de los más fáciles se le da giro que obliga a pensar y a modificar el algoritmo para adecuarlo a lo que se pide.

La representación usada aquí fue con listas, esto es porque cada uno de los caminos del laberinto no conectarían, a priori, con demasiados caminos. En otras palabras, habría pocas aristas. A pesar de esto, siempre se puede introducir un caso de prueba en donde haya muchas aristas.

El algoritmo es una simple búsqueda en profundidad, pero tiene ciertos matices. El algoritmo debe imprimir el recorrido que ha hecho para salir del laberinto, así como apuntar por cuantos sitios del laberinto va pasando, esto incluye tanto los sitios que hace avanzando como aquellos que hacer volviendo hacia atrás.

El problema principal a la hora de hacer este ejercicio fue como detectar cuando el programa “volvía hacia atrás” cuando terminaba la búsqueda recursiva sobre un nodo. La solución fue colocar un contador que indica el número de veces que hemos pasado por eso sitio concreto. Por lo tanto, si al pasar por un nodo vemos que el contador es mayor que 0 y que el nodo que representa la salida aún no ha sido visitado, quiere decir que estamos volviendo hacia atrás en el laberinto.

Los nodos que vamos recorriendo se guardan en una estructura auxiliar, una lista. En la primera posición encontramos el primer nodo que visitamos (sin contar el nodo en donde empezamos) y en la última posición encontramos el último nodo que visitamos (sin contar el nodo que representa la salida). Para saber si estoy encerrado en el laberinto simplemente recorro la lista auxiliar de nodos recorridos y si el último nodo no es la salida entonces estoy encerrado.

En cuanto a la eficiencia hay que tener cuenta la inicialización de las matrices que se usan para realizar el algoritmo correctamente que siempre dependerá del número de nodos. Por otro lado, encontramos el algoritmo bpp en sí que al estar usando una representación mediante listas el coste sería O(n + a).

**404**

**Análisis:**

Este es el primer ejercicio difícil y con el primero que tuve problemas al intentar resolverlo. El ejercicio consiste en desarrollar un algoritmo que calcula la distancia mínima entre dos ciudades pasando por una tercera ciudad.

La representación usada para el grafo es una matriz, pues se especifica en el enunciado que el mapa es muy detallado y que, por tanto, habrá muchas aristas. Por otro lado, escogí usar Dijkstra sobre Floyd porque Dijkstra me pareció más sencillo de entender y, además, es más eficiente. Usé el valor -1 como marca de que no hay arista.

El principal problema lo encontré a la hora de calcular el nodo v que en cada iteración de Dijkstra pasa a estar dentro del conjunto de nodo solución. Este nodo no puede haber estado ya en la solución y debe ser el que menor coste tenga en ese momento. Tuve múltiples problemas como bucles infinitos o que simplemente no se estaba escogiendo el nodo correcto. Decidí usar dos bucles for. El primero escoge un nodo asegurando que no está en la solución ya, pero no asegura que sea el de menor coste. Esto último lo hace el segundo bucle for que se encarga de comparar el nodo escogido en el primer bucle con el resto, encontrando el mínimo. Otro problema que tuve fue como detectar si llegar a una ciudad era imposible ya que mi algoritmo terminaba en el momento en el que la ciudad destino ha sido visitada. Cuando no se puede llegar a esa ciudad el programa entra en un bucle infinito. Lo que hizo fue detectar si este caso se daba y salir de la función si así era. La forma de detectarlo es con un contador que aumenta en cada iteración, si este contador es mayor que el número de nodos entonces nunca llegará a la ciudad origen. No deja de ser como la condición del bucle for usado en el algoritmo de las transparencias de teoría.

En cuanto a la eficiencia, la inicialización de la las matrices bidimensionales es de orden cuadrático y la matrices unidimensionales tienen coste lineal. Por otro lado, la eficiencia del algoritmo en si es prácticamente la eficiencia del algoritmo de Dijkstra original, solo que quizás un poco más eficiente porque mi algoritmo para cuando ha llegado a la ciudad origen.

**407**

**Análisis:**

Este ejercicio consiste en comparar el máximo valor de todos los caminos mínimos de todos los nodos y quedarnos con el menor de ellos.

La representación escogida son arrays, en mi opinión suele ser la mejor representación en ejercicios donde los nodos son ciudades y las aristas carreteras.

Elegí el algoritmo de Floyd por el hecho de que, en este caso, había que calcular los caminos mínimos de todos con todos. Además, era más fácil de programar que Dijkstra.

No encontré demasiadas complicaciones a la hora de realizar el ejercicio, algo normal teniendo en cuenta que es de los más fáciles.

El principal problema de mi algoritmo es la eficiencia. Dejando a un lado las inicializaciones a la hora de leer el grafo, Floyd es un algoritmo ineficiente (O()) y cada vez que se ejecuta este algoritmo se realiza una inicialización de las estructuras usadas en él (O()). Por último, la función que calcula la solución final es de orden cuadrático.

**411**

**Análisis:**

Se trata del último ejercicio que realicé. Simplemente debemos identificar grafos, dentro de un grafo mayor. Estos grafos son denominados islas.

La representación usada son listas. Puesto que muchas componentes del grafo se encuentran totalmente desconectados, las listas es la representación idónea para este ejercicio.

No he encontrado prácticamente dificultad al realizar el ejercicio. Se trata de realizar una búsqueda en profundidad en la que debemos ir apuntando a que isla pertenece cada nodo. La manera de apuntar esto es mediante un array en donde guardamos para cada elemento la isla a la que pertenece. Las islas se asignan teniendo en cuenta la iteración del bucle no recursivo en la que nos encontramos.

La eficiencia es bastante buena, pues es un simple recorrido y además estamos usando listas. Quitando la lectura del grafo, la inicialización de las estructuras usadas en la bpp es de coste lineal y el coste del bpp usando listas es O(n + a).

**412**

**Análisis:**

Este es sin duda uno de los ejercicios que más me han gustado de todos. De nuevo, se trata de calcular el camino mínimo entre dos nodos pero, esta vez, el camino mínimo es la arista más larga de ese camino.

La representación usada es arrays. La razón principal es que, al especificarse que siempre habrá un camino entre dos nodos, supuse que podría haber muchas aristas. El algoritmo utilizado es de nuevo Dijkstra pero con ciertos matices, pues en la matriz de costes siempre debemos guardar el valor de la arista más larga del camino mínimo. Elegí de nuevo Dijkstra porque lo tenía ya implementado y además supuse que con algunas modificaciones, el algoritmo debería funcionar.

No encontré grandes complicaciones a la hora de realizar el ejercicio y como puede verse en el código siempre me quedo con la arista más larga del camino mínimo y la comparo con la arista del camino mínimo de la matriz de costes, quedándome de entre estas dos con la más pequeña.

En cuanto a la eficiencia del algoritmo, es similar al ejercicio 404.

**413**

**Análisis:**

Este se trata de otro ejercicio que me pareció muy interesante y que fue entretenido de hacer. Se trata de programar un árbol de expansión mínimo usando algunos de los dos algoritmos utilizados. Elegí Prim al ser el que mejor entendía y recordaba de los dos.

La representación escogida es una matriz. Esto se debe a que en la inicialización de la matriz de adyacencia hago que exista una arista entre todos los nodos y, por otro lado, también me parecía más fácil de implementar que usando listas.

El principal problema que me encontré fue como representar los nodos en sí, ya que solo nos daban unas coordenadas y tampoco tenía muy claro que aristas había que poner y cuáles no. La solución al primer problema fue usar una matriz de estructuras. Cada elemento representa a su nodo correspondiente. De esa forma ya tenía a cada ciudad representada, pero aún falta la unión entre ellas (las aristas). La solución a esto (que soluciona también mi segundo problema anteriormente mencionado) fue conectar inicialmente todas las ciudades entre ellas, el algoritmo de Prim ya se encargaría de construir el árbol correspondiente.

EL algoritmo en sí no supuso mayores problemas. Cabe destacar que en este punto ya había realizado más ejercicios de grafos por lo que comenzaba a acostumbrarme a los grafos y por tanto las soluciones las conseguía mucho más rápido.

En cuanto a la eficiencia, dejando de lado las inicializaciones de los arrays, inicializar las estructuras para realizar el algoritmo de Prim tiene un coste lineal, encontrar el nodo mínimo tiene un coste cuadrático y por último, el propio algoritmo en sí, tiene coste cuadrático.

**418**

**-Análisis:**

Este es el único ejercicio difícil que he conseguido resolver. A pesar de tratarse de uno difícil su premisa es sencilla, llegar de una esquina a otra de una habitación teniendo en cuenta unas restricciones. En un primer vistazo no parece un ejercicio de grafos, pero realmente mi algoritmo realiza una bpp sobre la matriz que representa la habitación. La premisa del ejercicio es que debemos calcular el veneno mínimo que debemos usar para que un alien no sea capaz de llegar a la esquina opuesta (empieza siempre en la [1,1]). El veneno ocupa las baldosas cuya altura sea menor que la altura del veneno, por tanto, el alien no puede pasar a través de esas baldosas. El alien tampoco puede moverse a una casilla cuya diferencia de altura con su baldosa actual sea de dos o más.

Sin duda es el ejercicio donde más problemas he tenido, llegando a tener tres versiones del programa. La idea de mi programa es sencilla, voy probando alturas del veneno hasta que el alien no consigue llegar a la otra esquina (que representa la salida). Hay un caso especial a considerar que si el alien está encerrado por baldosas cuya altura no le dejan moverse, en este caso, el veneno mínimo es 0. Cuando la altura del veneno es 0 y el alien no ha llegado a la otra esquina se puede intuir que el alien está encerrado.

Los problemas vinieron con el recorrido sobre la matriz. En la primera versión fallaba porque siempre me movía hacia la derecha y abajo, nunca arriba o a la izquierda. Por tanto, en la primera versión había múltiples caminos que no recorría. La segunda versión intenta solucionar el problema del recorrido incompleto con un extenso análisis de casos. Esta versión funcionaba incluso peor que la anterior, pues el Juez Online me daba un “runtime error”. Finalmente, simplifiqué la segunda versión en unas 15 líneas, que comparada con la segunda versión (50 líneas), mejoraba mucho el rendimiento y reducía la posibilidad de errores. El último problema que tuve con este ejercicio fue que no consideraba la casilla de la salida a la hora de comparar su altura con el veneno, no deja de ser un error de despiste.

En cuanto a la eficiencia, realmente se trata de un bpp por lo que su eficiencia es similar a este tipo de recorrido.

En cuanto a la representación, elegí una matriz porque a nivel conceptual es mucho más fácil de entender, ya que la matriz funciona como una especie de plano de la habitación.

**Conclusiones**

Este proyecto es posiblemente el más complicado de todos los realizados en la asignatura. Los problemas planteados son, en general, de una dificultad bastante alta, incluso aquellos que no son denominados como difíciles.

Los problemas que podemos ver resueltos en esta documentación no son todos los que he intentado hacer, pero por falta de tiempo no me ha sido posible realizar más. Por ejemplo, el problema 420 trata sobre buscar las articulaciones de un grafo. Sin embargo, a la hora de la implementarlo me he encontré con diversas complicaciones que hicieron que dejara de intentarlo al no tener mucho tiempo para realizar el resto de problemas.

En general, me ha llevado bastante tiempo realizar esta actividad. Problemas como el 402 y el 403 me tomaron cerca de 5 horas cada uno para su resolución. Otros como el 404 y 413 me tomaron alrededor de las 6 horas para resolverlos. A pesar de tener las ideas más claras, la implementación de estos 2 últimos problemas es más compleja que la de los dos primeros y por tanto pueden aparecer muchos más errores de programación. El problema 412 no me llevo tanto, pues era muy parecido al problema 404, fueron unas 4 horas. El ejercicio 407 tampoco necesité demasiado tiempo, tardé 2 horas. Sin duda, el ejercicio que menos tiempo me llevo fue el 411 que no llegó a una hora. Por último, el que más tiempo me llevo fue el 418, no en si por la complejidad del programa, sino por la complicación a la hora de saber si estaba recorriendo la matriz por todos los caminos posibles. Fueron unas 14 horas.

Me ha parecido una actividad muy interesante y que ayuda a profundizar en el tema de grafos. La actividad me ha parecido especialmente entretenido cuanado aplicamos los conocimientos teóricos a una situación que podría darse en la vida real.