MEMORIA EJERCICIOS GRAFOS

ALBERTO GÁLVEZ GÁLVEZ

DNI: 49448576W

USUARIO: B126

2º INGENIERÍA EN INFORMÁTICA

SUBGRUPO 2.3

UNIVERSIDAD DE MURCIA

ÍNDICE

Tabla resumen

Problema	402	403	404	407	409	411	413	418
Algoritmo	BPA	BPP	Dijkstra	Dijkstra	BPP*	Rel. eq.	Prim	BPP
Rep. Gr.	MAA	LAA	MAD	MAA	LAA	AD	AD	MA
ID envío	1489	1491	1492	1493	1494	1495	1496	1497

-BPA: Búsqueda Primero en Anchura

-BPP: Búsqueda Primero en Profundidad

-BPP*: Búsqueda Primero en Profundidad para encontrar los componentes fuertemente conexos de un grafo

-Rel. eq.: Relaciones de Equivalencia

-AD: Array en memoria Dinámica

-MA: Matriz en memoria Automática

-MAA: Matriz de Adyacencia en memoria Automática

-MAD: Matriz de Adyacencia en memoria Dinámica

-LAA: Array de Listas de Adyacencia en memoria Automática

Resolución de problemas

Consideraciones

Cabe destacar que, si bien es verdad que la resolución de los ejercicios se podría haber parametrizado mayormente, se ha hecho uso de algunas variables globales. Esto, con el fin de presentar un código más sencillo y legible (si fuese un proyecto con módulos se optaría por parametrizar al máximo) al evitar pasar numerosos parámetros a cada una de las funciones, en los casos de recursividad evitar realizar una implementación sustancialmente más compleja, etc.

Véase que, en los ficheros de código fuente, aparece explicado a modo de comentarios cada uno de los algoritmos usados, así como las variables, condiciones y bucles usados para la solución del ejercicio en cuestión.

Nótese también que, las decisiones de diseño de las soluciones a los ejercicios vienen motivadas por un compromiso entre, velocidad de ejecución, memoria usada, y simplicidad del código. Así, por ejemplo, siendo las soluciones en algunos ejercicios mejores en cuanto a velocidad y simplicidad del código, pero no tanto en cuanto a memoria, o viceversa.

402

Para la representación del grafo, se ha optado por una matriz de adyacencia en memoria automática. Esto, por una parte, debido a que el ejercicio se encuentra con una cota máxima muy pequeña que no nos produciría un desperdicio de memoria notable en casos extremos, y por otra, para conseguir una mayor velocidad respecto a la implementación con listas de adyacencia, al ser el acceso con arrays es mucho más rápido.

No encontré ningún tipo de dificultad para la implementación del ejercicio.

La eficiencia del algoritmo es $O(n^2)$ puesto que, la búsqueda primero en anchura con matrices de adyacencia no deja de ser recorrer una matriz de *n* filas y *n* columnas por fila.

403

Para la representación del grafo, se ha optado por un array de listas de adyacencia en memoria automática. Esto, ya que, en caso de representar el grafo con matrices de adyacencia, se produciría un ingente desperdicio de memoria e incluso de rendimiento como consecuencia de encontrarnos frente a una matriz extremadamente escasa, debido al hecho de que el grafo representa un laberinto (teniendo cada sitio escasas conexiones con algún otro).

La mayor dificultad encontrada a la hora de implementar este ejercicio fue el hecho de inicialmente no añadir a la cola (del registro del recorrido realizado para llegar al final del laberinto) de nuevo, el nodo al que se vuelve en caso de haber intentado llegar al final por un nodo suyo adyacente y no haberlo conseguido, para entonces probar con el nodo el nodo adyacente que se encuentra a la derecha del probado.

La eficiencia del algoritmo es $O(n+a)$ puesto que, la búsqueda primero en profundidad con listas de adyacencia no deja de ser recorrer *n* listas, accediendo en total *a* aristas.

404

Para la representación del grafo, se ha optado por una matriz de adyacencia creada dinámicamente. Por una parte, para ahorrar memoria en comparación a si se realizase con un array en memoria automática, cuyo número de filas y de columnas fuese establecido de acuerdo con el número de ciudades dado por el enunciado del problema como cota máxima, y por otra, para conseguir una mayor velocidad respecto a la implementación con listas de adyacencia, ya que el acceso con arrays es más rápido que el acceso con punteros usado en las listas. Cabe destacar que en el enunciado del problema se nos dice que se trata de un mapa muy detallado, así indicándonos que no nos encontramos frente a

casos que puedan dar lugar a matrices escasas, siendo la representación con listas de adyacencia ideal.

La mayor dificultad encontrada a la hora de implementar este ejercicio fue el hecho de designar la no conexión entre ciudades mediante el número *-1*, ya que producía la necesidad de realizar numerosas adaptaciones al algoritmo de Dijkstra visto en teoría (pues en él, se asumen aristas positivas).

La eficiencia del algoritmo es $O(n^2)$ puesto que, para la ejecución del algoritmo se debe hacer:

1. Inicializar adecuadamente las estructuras de datos usadas en el algoritmo de Dijkstra mediante un bucle: $O(n)$.
2. Realizar $n-1$ veces (puesto que en la inicialización se incluye en la solución el nodo con el que ejecutar el algoritmo) como máximo, puesto que si se encuentra el camino deseado el algoritmo se detiene.
    a. Buscar el nodo con mínimo *D[v]* y *S[v]* falso: $O(n)$.
    b. Actualizar los valores de los candidatos: $O(n)$.

Como se ve, nos encontramos: $O(n+(n-1)*2n) = O(n^2)$.

407

Para la representación del grafo, se ha optado por una matriz de adyacencia creada en memoria automática. Esto, por una parte, porque la cota máxima que nos da el ejercicio de número de ciudades no produciría un desperdicio de memoria notable en casos extremos, y por otra, para conseguir una mayor velocidad respecto a la implementación con listas de adyacencia, ya que el acceso con arrays es mucho más rápido que el acceso con punteros usado en las listas.

Una vez implementado el ejercicio 404, la realización de este ejercicio no supuso ninguna dificultad, puesto que consiste en usar *n* veces (con algunas modificaciones como obviar los casos en los que no haya conexión entre ciudades al ser un grafo conexo, comprobaciones para la excentricidad, etc) el algoritmo implementado en el ejercicio anterior.

La eficiencia del algoritmo es $O(n^3)$ puesto que, para la ejecución del algoritmo se debe hacer *n* veces:

1. Inicializar adecuadamente las estructuras de datos usadas en el algoritmo de Dijkstra mediante un bucle: $O(n)$.
2. Realizar $n-1$ veces (puesto que en la inicialización se incluye en la solución el nodo con el que ejecutar el algoritmo) como máximo, puesto que si se encuentra el camino deseado el algoritmo se detiene.
    a. Buscar el nodo con mínimo *D[v]* y *S[v]* falso: $O(n)$.
    b. Actualizar los valores de los candidatos: $O(n)$.
3. Recorrer el array de distancias calculado en la última ejecución de Dijkstra para obtener la mayor de ellas: $O(n)$.

4. Comprobar si la distancia obtenida es menor que la de la ciudad menos excéntrica hasta el momento.

Como se ve, nos encontramos: $O(n*(n+(n-1)*2n+n)) = O(n^3)$.

409

Para la representación del grafo, se ha optado por un array de listas de adyacencia en memoria automática, ya que el enunciado deja claro que se cumple: $a << n$.

La realización del ejercicio fue bastante tediosa. Esto, debido a que inicialmente el orden de los nodos que se inserta en el array *nPost* lo realizaba en el orden en el que se visitaban los nodos, en lugar de en el orden en el que se resuelven sus llamadas. También, el hecho de tener que mostrar ordenadamente la salida aumentaba considerablemente la dificultad del ejercicio.

El algoritmo para encontrar los *CFC* tiene una complejidad de $O(n+a)$, que es la complejidad de la *BPP* con listas de adyacencia. Sin embargo, en este ejercicio, puesto que para poder mostrar los datos ordenadamente se hace uso de la función *merge* en cada *BPP* realizada con la matriz traspuesta, la cual tiene una complejidad de $O(n'+n''-1)$ siendo n' el número de nodos del grafo leído y n'' el número de nodos del grafo traspuesto. Se puede deducir que la complejidad frente a la que nos encontramos es: $O((n+a)*(n-1)) = O(n^2+n*a)$.

411

Este ejercicio, en lugar de ser resuelto como un problema de grafos (con *BPPs*), ha sido resuelto mediante relaciones de equivalencia con compresión de caminos (tema 3) para una mayor eficiencia. Para ello, se ha hecho uso de un array de punteros al padre creado en memoria dinámica.

La mayor dificultad de este fue el no contemplar inicialmente la unión entre elementos de una misma clase, entrando en un bucle infinito pues.

La eficiencia del algoritmo es $O(n+a*log(n)+n)$, puesto que la ejecución se realiza en los siguientes pasos:

a. Inicialización del array de punteros al padre: $O(n)$.
b. Unión de clases de equivalencia a veces, siendo $O(log\ n)$ el coste de la operación *encuentra*, que debido a la compresión de caminos tenderá a ser menor realmente: $O(a*log(n))$.
c. Asignación de una isla a cada raíz recorriendo el array de punteros al padre: $O(n)$.

Como se ve, nos encontramos: $O(n+a*log(n)+n) = O(n+a*log(n))$.

413

Para la implementación de este ejercicio se ha optado por se ha optado por la implementación mediante un array de estructuras *Punto* en memoria dinámica, ya que todos los nodos se relacionan entre sí, calculándose las distancias entre ciudades mediante sus coordenadas.

Para la implementación de este ejercicio se ha optado por el algoritmo de Prim, ya que al estar relacionados todos los nodos entre sí (ya que las distancias entre ciudades se calculan mediante las coordenadas) es más eficiente que el algoritmo de Kruskal;

Este ejercicio no supuso ningún tipo de complicación, debido a la semejanza en de implementaciones entre el algoritmo de Prim y Dijkstra.

La eficiencia del algoritmo es $O(n^2)$ puesto que, para la ejecución del algoritmo se debe hacer:

1. Inicializar adecuadamente las estructuras de datos usadas en el algoritmo de Prim mediante un bucle: $O(n)$.
2. Realizar n-1 veces (puesto que en la inicialización se incluye en la solución el nodo con el que ejecutar el algoritmo).
    a. Buscar el nodo con mínimo *MenorCoste[v]*: $O(n)$.
    b. Actualizar los valores de los candidatos: $O(n)$.
3. Sumar todos los costes del array *MenorCoste*.

Como se ve, nos encontramos: $O(n+(n-1)*2n+n) = O(n^2)$.

418

Este ejercicio se ha implementado mediante una matriz que representa las alturas de cada una de las casillas de la celda del alien. Teniendo en cuenta que los nodos/casillas adyacentes son aquellos que se encuentran una posición arriba, a la derecha, abajo, o a la izquierda, como si de un tablero se tratase, la representación más adecuada resulta esta.

La mayor dificultad encontrada fue la eficiencia, encontrándome en la plataforma *Mooshak* con *Time Limited Exceded*, ya que, inicialmente, entre cada *BPP* aumentaba la altura del veneno en 1, en lugar de como describo más adelante.

En cuanto a la complejidad, se trata de realizar sucesivamente *BPPs* de la casilla inicial sucesivamente hasta que mediante la *BPP* no se consiga visitar el nodo/casilla que representa la salida de la celda. Entre cada *BPP*, ponemos el veneno a una altura más de la casilla más baja que se ha encontrado en el camino por el que ha escapado. Por ello, complejidad viene determinada por $O(n^2)$, ya que es realizar *BPPs* un número determinado de veces, que depende de los datos de entrada.

Conclusiones

En primer lugar, he de decir que los grafos me parecen interesantísimos, tanto por su implementación, como también por la cantidad de problemas de toda índole y gran interés que pueden modelizar y solucionar de forma óptima.

En segundo lugar, me he sentido abrumado y decepcionado por el plazo de entrega de la actividad en fecha 28/12/2020, pareciéndome injusta por el hecho de que a todos los grupos se nos exige la misma fecha, cuando algunos grupos han dado los contenidos considerablemente antes que otros. Por ejemplo, el grupo 3 recibió la clase en la que se explicaba el algoritmo de Prim (entre otros) en fecha 2/12/2020, mientras que el grupo 2 en fecha 14/12/2020. Esto, teniendo en cuenta la proximidad de la fecha de entrega, nos pone en gran desventaja a ciertos grupos respecto a otros (por lo menos a los alumnos de estos que no se conforman con la realización de los ejercicios más simples para simplemente aprobar), al tener unos aproximadamente un mes desde que se dan los contenidos hasta la entrega de la tarea, y otros tan solo dos semanas.

En tercer lugar, tal como señalé en la memoria del *Proyecto de C++ de los temas 2 y 3,* considero que habría sido de gran ayuda haber sido formados en herramientas *debugger*, aunque fuese mínimamente. Se habría agradecido bastante un tutorial básico de *debug en C++.*

En cuarto lugar, los ejercicios de grafos son curiosos, puesto que algunos ejercicios que aparentemente parecen sencillos, requieren de consideraciones que los hacen tener soluciones mucho más complejas que algunos que a priori parecen muy complicados. Un ejemplo de esto, son las modificaciones que hay que realizar en el algoritmo de Dijkstra en caso de que las no conexiones se señalicen con un número negativo en lugar de con un número positivo extremadamente grande.

En quinto lugar, he de decir que intenté realizar el ejercicio *412* sin éxito, al no salirme bien algunos casos del ejemplo extendido. Teniendo 19 puntos por la realización de los demás ejercicios, finalmente decidí no solucionarlo.

Como conclusión y tras haberle dedicado aproximadamente unas 30 horas a la realización de la actividad, puedo decir que, pese a algunos factores insatisfactorios, la realización de esta ha sido de gran interés y agrado por su interesante ámbito de estudio, asentando los conocimientos vistos en teoría de forma muy satisfactoria.