Universidad de Santiago de Chile
Facultad de Ingeniería
Departamento de Ingeniería Informática

Estructura de datos y análisis de algoritmos

Laboratorio 3 – Salvación espacial II

Christofer Rodríguez

Profesor: Mario Inostroza

Ayudante: Estaban Silva

Santiago – Chile
2-2020

Tabla de contenidos

Índice de figuras

Índice de tablas

CAPÍTULO 1. INTRODUCCIÓN

Debido a una pandemia mundial ocurrida en la tierra la doctora Ximi junto con su tripulación desean llegar al planeta Pizza Planet. Es por esto por lo que es necesario crear un programa en el lenguaje de programación en C que calcule el camino más corto que permita llegar desde el planeta Tierra al planeta Pizza Planet, con el combustible disponible en la nave y entregue este camino mediante un archivo de texto. Por medio de archivos de entrada se generarán las conexiones entre cada planeta con su respectivo tiempo de viaje y en que planetas se encuentran las estaciones de recarga de combustible.

Los objetivos de este informe son explicar la implementación de dicho programa, analizar la complejidad tanto del programa en general como de las funciones que lo componen, para finalmente se analizar el alcance y limitaciones de la solución empleada.

La manera de llevar a cabo estos objetivos será, primeramente, explicar los conceptos fundamentales para comprender este documento en el marco teórico, luego se explicará y analizará paso a paso tanto las estructuras, como los algoritmos utilizados para la implementación del programa en la descripción de la solución, además se estudiará su complejidad algorítmica, lo que nos permitirá llegar a expresiones matemáticas que ayudarán a entender su comportamiento. Para finalizar, en el análisis de resultados se examinarán las falencias, limitaciones, ventajas, la eficiencia algorítmica y se comparara el tiempo teórico con el tiempo real de ejecución del programa creado.

CAPITULO 2. DESCRIPCIÓN DE LA SOLUCIÓN

2.1.1 Marco teórico

- **Algoritmo:** Procedimiento que mediante una serie de pasos transforma una entrada o un conjunto de estas en una salida.

- **Eficiencia algorítmica:** Análisis de la cantidad de recursos utilizados por un determinado algoritmo, estos recursos pueden ser, por ejemplo: tiempo o memoria.

- **Struct:** Declaración de estructura en la cual se especifica una secuencia de variables que pueden ser de distinto tipo, posteriormente sirve como plantilla para la creación de esta estructura.

- **Puntero:** Tipo de dato cuyo valor es una dirección de memoria.

- **Grafo:** Herramienta de modelamiento matemático representado gráficamente mediante puntos llamados vértices, unidos a través de aristas.

- **Matriz de adyacencia:** Representación de un grafo a través de una matriz bidimensional que asocia cada fila y columna a cada elemento del grafo.

- **Búsqueda en espacio estado: T**écnica de búsqueda en la que se evalúa cada posible situación en la que se pueda encontrar el problema.

- **Nodo:** Estructura que contiene una parte de datos, estos pueden ser más de uno y de diferentes tipos, mientras que la otra parte es un puntero que, en este caso, solamente apunta al siguiente elemento de la lista.

- **Arreglo:** Variables estructuradas, donde cada elemento se almacena de forma consecutiva dentro de la memoria, estos pueden ser de diferentes tipos de datos.

- **String:** Arreglo que contiene un carácter en cada una de sus posiciones, corresponde a la representación de una cadena de texto con largo definido.

2.1.2 Algoritmos y estructura de datos

Antes de explicar la solución implementada para este problema, es totalmente necesario indicar que el programa creado se encuentra incompleto, es por esto por lo que primeramente se procederá a explicar las funcionalidades con las que si cuenta este programa y luego se explicará el lineamiento de solución que se esperaba implementar dentro de esta solución.

Para solucionar el problema planteado se decidió utilizar una modificación de una matriz de adyacencia para la representación del grafo, esta matriz será tridimensional donde en la tercera dimensión se encontrará el tiempo empleado en el viaje y el gasto de combustible, luego mediante una búsqueda en espacio estado recorrer de todas las maneras posibles el grafo, una vez recorrido completamente se conservará la ruta más eficiente. Lo anterior se realizará con la ayuda de un arreglo de **strings** que contiene el nombre de los vértices y su posición dentro de este arreglo representa el índice que tiene este planeta dentro de la matriz de adyacencia.

A continuación, se enumerará de manera breve las etapas que sigue el programa para encontrar la solución del problema, cada una de estas serán explicada de manera más detallada a lo largo de esta sección:

1. Leer la información contenida en un archivo de entrada con la información de los nodos, aristas y costos.
2. Actualizar el contenido de la matriz de adyacencia con los costos en las posiciones que representen las respectivas aristas del mapa.
3. Leer la información de un archivo de entrada que contiene el combustible máximo de la nave y la ubicación de las estaciones de recarga.

*** Los puntos a continuación no se encuentran implementados dentro del programa.**

4. Realizar una búsqueda en espacio estado sobre el grafo para encontrar la mejor ruta de viaje.
5. Generar un archivo de salido con la ruta más eficiente para llegar al destino.

Cabe señalar que los puntos 1, 2 se realizarán a la vez, una vez que sean leídos los elementos del archivo de texto, estos serán incluidos dentro de su respectivo “contenedor”.

Para la solución de este problema, se utilizó una estructura de datos del tipo **struct** para representar un grafo.

La estructura **grafo** tiene como objetivo representar un grafo dentro del programa, esta estructura posee dos componentes. El primero de estos componentes es **n**, esta variable de tipo entero es la encargada de almacenar la cantidad de vértices presentes en el grafo; mientras que la segundo es un arreglo tridimensional de tipo entero en que representa la matriz de adyacencia del grafo.

```
struct grafo{
    int n;  //cantidad de nodos
    int ***A; //matriz de adyacencia
};
```

Ilustración 1: Estructura del grafo

Además, se establecieron dos variables constantes globales dentro del programa. La primera de estas constantes es la cantidad máxima de caracteres que puede contener un arreglo de tipo **char**, de esta manera podremos modificar fácilmente el tamaño permitido en el caso de que el nombre de los planetas sea muy largo; mientras que la segunda constante representa la cantidad máxima de nombres de nodos que pueden estar registrados dentro de una lista de nodos registrados. Si bien esta ultima constante limita la cantidad de nodos que puede tener el problema, es necesario definirla, ya debido al diseño de la solución es necesario saber la cantidad de nodos del grafo antes de leer el archivo de entrada que contiene esta información.

```
#define MAX_STRING 50
#define CANT_NODOS 10
```

Ilustración 2: Constantes globales

Para comenzar a solucionar este problema debemos leer el archivo de entrada que contiene los datos del grafo, esta tarea será realizada mediante el algoritmo llamado **crearGrafo**, este algoritmo leerá a información contenida dentro del archivo de texto a la vez que actualiza la información dentro de la estructura del grafo. Esta función recibe como argumentos el nombre de archivo de entrada que contiene la información, la dirección de memoria de la variable de tipo entero que almacenará la capacidad máxima de combustible de la nave y un arreglo de **strings** que contendrá el nombre de los vértices(planetas) del grafo.

En un inicio se creará una variable de tipo **FILE** para el archivo de entrada, se procede a abrir este archivo en modo lectura y antes de comenzar a trabajar con él, se comprueba de que este haya sido abierto de manera correcta, en el caso de que este no se haya abierto de manera correcta se retornará un **grafo** con una cantidad de nodos igual a cero para que este error sea identificado al momento de salir de este algoritmo.

```
FILE *pf;
struct grafo *g;
pf = fopen(nombreArchivo,"r");

//Si es que no se pudo abrir el
if(pf == NULL){
    g->n = 0;
    return g;
}
```

Ilustración 3:Apertura y comprobación del archivo de entrada

Si es que no existió ningún error al momento de abrir el archivo de entrada se crearán variables auxiliares del tipo entero para almacenar los datos contenidos en el archivo de entrada, se comienza leyendo el primer elemento del archivo de texto que corresponde a la cantidad de nodos del grafo, esta es almacenada en su respectiva variable dentro del **grafo**, luego se procede a asignar la memoria para la matriz de adyacencia tridimensional de tipo entero, una a medida que se crea la tercera de estas dimensiones, se inicializan sus dos posiciones en cero, de esta manera nos aseguramos de que no se mezclen valores indeseados provenientes de la asignación de memoria.

```
//Crear grafo, asignar la memoria correspondiente y
g = (struct grafo *)malloc(sizeof(struct grafo));
g->n = n;
g->A = (int ***)malloc(sizeof(int **)*n);
for(int i = 0; i < n; i++){
    g->A[i] = (int **)malloc(sizeof(int *)*n);
    for(int j = 0; j < n; j++){
        g->A[i][j] = (int *)malloc(sizeof(int)*2);
        g->A[i][j][0] = 0;
        g->A[i][j][1] = 0;
    }
}
```

Ilustración 4: Creación de la matriz de adyacencia

Luego de que se reserva memoria para la matriz, se crean dos variables auxiliares del tipo **string** para contener los nombres de los planetas que poseen una arista entre sí, ahora, mediante un ciclo **while** se lee cada posición del archivo de entrada hasta que no se detecte que se ha llegado al final de este. El primer elemento de una línea del archivo corresponde al vértice de “origen” de una arista, mientras que el segundo al vértice de “destino”; ya que no

podemos saber de antemano todos los vértices presentes en el grafo y cada uno de ellos tiene su propio nombre, registraremos el nombre de cada uno de estos planetas dentro de un arreglo de **strings**, además la posición que tengan estos planetas dentro del arreglo representará su posición dentro de la matriz de adyacencia del grafo.

Una vez leído el nombre del primer vértice dentro de una "relación de adyacencia" se procede a comprobar si este vértice se encuentra ya registrado dentro del arreglo de **strings**, esto se realiza con ayuda de la función **existeNodo**; este algoritmo recibe como argumentos la lista de nodos registrados, el nombre del planeta que se desea buscar y el largo (cantidad de elementos) de dicha lista.

Una vez dentro de la función mediante un ciclo **for** se recorre cada posición del arreglo hasta llegar al final de este (cantidad de elementos actuales), ha medida que se recorre se comprueba si el **string** almacenado en la actual posición es igual al nombre del nodo buscado, en caso de que corresponda, se retornará un 1 indicando que el planeta buscado ya está registrado, en caso de que se recorra completamente el arreglo y no se encuentre el nodo buscado, se retornará un 0 indicando que el nodo no se encuentra registrado.

```
int existeNodo(char listaNodos[][MAX_STRING], char nodo[MAX_STRING], int largo) {
    for(int i=0; i < largo; i++) {
        if(strcmp(listaNodos[i], nodo) == 0) {
            return 1;
        }
    }
    return 0;
}
```

Ilustración 5: Función existeNodo

Este algoritmo posee un $T(n) = 3n + C$, con un orden de complejidad $O(n)$.

Tanto en el caso para el planeta de "origen" como para el planeta de "destino" el retorno de la función **existeNodo** es igual a 0, indicando que el nodo no esta registrado, se procederá a registrar al final del arreglo de **string** que contiene el nombre de los planetas y se actualiza la cantidad de elementos que posee este arreglo; si el nodo ya se encontraba registrado, se seguirá leyendo el archivo de entrada.

Una vez leídos y registrados los planetas de la "relación de adyacencia", es necesario actualizar los datos de la arista dentro de la matriz de adyacencia, para esto es necesario conocer la posición donde se almacenará esta información, para esto obtenemos el índice de estos planetas mediante la función **indiceNodo**. Esta recibe como argumentos el arreglo de **string** que contiene el nombre de los planetas registrados, el nombre del planeta del cual se desea conocer su índice y la cantidad de elementos almacenados actualmente en el arreglo.

Al igual que el algoritmo **existeNodo** se recorre el arreglo mediante un ciclo **for**, donde se comprueba si el **string** almacenado en la posición actual es igual al nombre del planeta buscado, cuando se encuentra el planeta, se retorna el número de la iteración actual, este

número representa la posición que tiene el planeta dentro del arreglo y de la matriz. En caso de que no lo encuentre se retornará -1, este caso no debería ocurrir nunc, ya que esta función es llamada solamente con planetas previamente registrados.

```
int indiceNodo(char listaNodos[][MAX_STRING], char nodoBuscado[MAX_STRING], int largo) {
    for(int i = 0; i < largo; i++) {
        if(strcmp(listaNodos[i], nodoBuscado) == 0) {
            return i;
        }
    }
    return -1;
}
```

Ilustración 6: Función indiceNodo

El algoritmo **indiceNodo** posee un $T(n) = 4n + C$, y un $O(n)$.

Cuando ya poseemos el índice del planeta de origen y el índice del planeta de destino, se almacena el tiempo de viaje entre ambos planetas en la posición correspondiente a la unión de ambos planetas en la matriz y dentro de la tercera dimensión, este valor es almacenado dentro de la primera posición, luego se lee valor que representa el costo de combustible para viajar entre los planetas y es almacenado en el mismo lugar, pero dentro de la tercera dimensión es asignado en la segunda posición.

```
fscanf(pf, "%s", origen);
if(existeNodo(nodos, origen, n) == 0){
    strcpy(nodos[nodosRegistrados], origen);
    nodosRegistrados++;
}

fscanf(pf, "%s", destino);
if(existeNodo(nodos, destino, n) == 0){
    strcpy(nodos[nodosRegistrados], destino);
    nodosRegistrados++;
}

indiceOri = indiceNodo(nodos, origen, nodosRegistrados);
indiceDes = indiceNodo(nodos, destino, nodosRegistrados);

fscanf(pf, "%d", &g->A[indiceOri][indiceDes][0]);
fscanf(pf, "%d", &g->A[indiceOri][indiceDes][1]);
```

Ilustración 7: Lectura y asignación de datos

Cuando el archivo de entrada es leído totalmente, este es cerrado y se retorna el **grafo** creado con los datos contenidos en el primer archivo de texto.

La totalidad de la función **crearGrafo** posee un $T(n) = 17n^2 + 13Cn + 18C$, con un orden de complejidad $O(n^2)$.

Ya habiendo leído el primer archivo, se leer el segundo archivo de entrada que contiene la información de la capacidad del tanque de combustible de la nave y los planetas en los que se puede recargar combustible. Esto lo realizaremos con el algoritmo llamado **leerCombustible**, este recibe como argumentos el nombre del archivo de texto, la dirección de memoria donde se almacenará la capacidad del tanque de combustible, el arreglo con los nombres de los planetas registrados, la cantidad de elementos que posee este arreglo y la dirección de memoria de la variable de tipo entero que almacenará el largo del arreglo que almacenará el índice de los planetas donde se puede recargar combustible.

Al igual que la función **crearGrafo**, se abre el archivo de texto en modo lectura y se comprueba de que no existan errores al momento de abrir dicho archivo. Una vez se comprueba de que no existan errores, se lee el primer elemento del archivo, este corresponde a la capacidad máxima del tanque de combustible de la nave, este valor es almacenado en la variable correspondiente mediante la dirección de memoria ingresada a la función; luego se crea una variable auxiliar del tipo **string** para guardar el nombre del planeta que posee una estación de recarga y se reserva memoria para un arreglo de entero de un largo $n - 2$, esto ya que en el pero de los casos todos los planetas a excepción del origen y destino poseerán una estación de recarga.

Mediante un ciclo **while** se leer el archivo mientras no se detecte que se llegó al final de este, el nombre del planeta leído es almacenado en la variable auxiliar, luego se encuentra el índice de este planeta con la función **indiceNodo** y este índice es almacenado dentro del arreglo de enteros que contiene el índice de los planetas con estaciones de recarga. Cuando se termina de leer el archivo, este es cerrado y se retorna el arreglo de enteros con los índices de las estaciones de recarga.

```
while(feof(pf) == 0) {
    fscanf(pf, "%s", estacion);
    estacionesRecarga[*largoRecargas] = indiceNodo(listaNodos, estacion, largoNodos);
    *largoRecargas = *largoRecargas +1;
}

fclose(pf);
return estacionesRecarga;
```

Ilustración 8: Lectura del segundo archivo de entrada

El algoritmo **leerCombustible** posee un $T(n) = 4n^2 + 3Cn + 8C$ y un $O(n^2)$.

Ahora que ya poseemos todos los datos externos, encontramos mediante el algoritmo **indiceNodo** el índice del planeta de inicio (Tierra) y el índice del planeta de destino (Planet Pizza) para así comenzar a realizar la búsqueda de la ruta más eficiente.

Ya que la implementación de la solución al problema no fue terminada, el programa solamente mostrará por consola la matriz de adyacencia creada, los planetas registrados con sus respectivos índices; esto solamente con el objetivo de ilustrar la representación pensada para esta solución.

A pesar no implementar el resto de la solución, se explicará el lineamiento principal que se esperaba seguir para solucionar este problema.

Para la búsqueda de la ruta más eficiente, se tenía planeado realizar una búsqueda en espacio estados, se recorrerían todos los posibles caminos del grafo, siempre y cuando existiera un camino y se poseyera combustible suficiente para seguir viajando. El camino que debe ser almacenado a medida que se recorre y ser agregado a una “lista” de estados revisados, de esta manera se puede comprobar que no se repita un estado ya revisado, además se tendrá un fácil acceso al camino seguido al momento de generar el archivo con la ruta.

La primera vez que se recorra hasta el final (Planet Pizza) el grafo, se debe guardar esta ruta junto con su tiempo total como la solución más eficiente, luego cada vez que se logra una nueva ruta hasta el destino final, esta debe ser comparada a la solución más eficiente actual, si la nueva ruta no es mejor que la almacenada como eficiente, esta debe ser descartada, pero si la nueva es mejor que la anterior, esta debe ser almacenada como la mejor solución.

Cuando ya no queden estados abiertos por revisar, se debe generar un archivo de salida con el camino recorrido en la ruta obtenida en esta búsqueda.

2.2 Análisis de resultados

Claramente el programa creado no es capaz de resolver el problema de manera completa, solamente es capaz de leer los archivos de entrada y almacenarlos en las respectivas estructuras de datos creadas para la representación del problema.

Obviando la principal falencia de que el programa no soluciona el problema, a continuación, se presentarán las falencias que presentaría el programa si es que este hubiera sido implementado como se tenía planeado.

2.2.1 Falencias

Como ya se explicó anteriormente en la descripción de la solución, debido a la forma en la que se leer el primer archivo de entrada, solamente podemos retornar el grafo creado con los datos leído, es por esto por lo que el arreglo de **strings** que contiene el nombre de los planetas debe ser inicializado previamente a la lectura del archivo, esto genera que se deba asignar memoria desconociendo el largo necesario para este arreglo. Por lo anteriormente descrito, para que el programa funcionara, hubiera sido necesario asignar memoria para un largo innecesariamente elevado o el usuario del programa hubiera tenido que modificar el largo de este arreglo cuando el grafo poseyera una cantidad de vértices mayor a la ya establecida.

Otra falencia que surgiría de seguir el lineamiento de esta solución es que es necesario implementar cierta “restricción” para que la nave no entre a un bucle de movimientos, pero el implementar esta restricción podría llegar a provocar que la nave no sea capaz de volver al planeta en el que se encontraba anteriormente, cuando esto es totalmente posible y en ciertas ocasiones conveniente.

Debido a que es necesario tener registro de todas las posibles soluciones ya revisadas, se utilizaría una gran cantidad de memoria para almacenarla y esto sumado a la primera falencia descrita en esta sección, generaría una muy mala utilización de recursos.

2.2.2 Complejidad algorítmica

Ya que el programa no se encuentra completo, solamente se calculó el tiempo de ejecución de lo que se tiene implementado, luego de calcular la complejidad algorítmica da cada algoritmo utilizado en la ejecución del programa y bajo las condiciones descritas, se llegó a la siguiente expresión matemática:

$$T(n) = 21n^2 + 16Cn + 35C$$

Esta expresión representa la cantidad de instrucciones que tendrá que ejecutar el programa para poder leer y crear las estructuras de datos necesarias para comenzar a solucionar el problema.

Si bien de lo anteriormente obtenido podemos decir que el programa posee un orden de complejidad $O(n^2)$, este no representa el nivel de complejidad o el tiempo de ejecución de una solución completa para el problema; además tomando en cuenta que para llegar a la mejor ruta de viaje es necesario revisar todas las posibles soluciones, el tiempo de ejecución de este programa sería bastante elevado. Por lo anterior, no se adjuntó un gráfico ni se comparó los tiempos teóricos con los reales, ya que estos datos no serían para nada representativos.

2.3 Conclusión

No se logró resolver el problema planteado, si bien se tenía la idea general de como resolverlo, no fue posible implementarlo; cabe mencionar que el tiempo no fue un factor que contribuyó a esto, ya que el tiempo se dio. Al principal factor que se puede atribuir este suceso es a la falta de práctica, ya que claramente no basta con saber la teoría si no se es capaz de poner en práctica; pensando también en la solución que se tenía planeada, esta no era la mejor solución a este problema debido a su elevado tiempo de ejecución, algoritmos como una modificación del algoritmo de Dijkstra hubieran sido una mejor manera de abordar el problema. Como principal conclusión se puede decir que es totalmente necesario poner en practica cada uno de los conocimientos que aprendemos, ya que, si no los ponemos en práctica, solamente se quedaran en eso; a pesar de no lograr completar este laboratorio, se considera esta una importante experiencia para el futuro desarrollo como programador, debido a que no solo se aprende de los logros, sino también de los fracasos.

Referencias

REAL ACADEMIA ESPAÑOLA: *Diccionario de la lengua española*, 23.ª ed., [versión 23.3 en línea]. (Recuperado 19/01/2021). https://dle.rae.es

Universidad de Chile (2020). Espacios de estados. (Recuperado 19/01/2021) https://users.dcc.uchile.cl/~abassi/Cursos/IA/Apuntes/c5.html

Universidad Federico Santa María (2020). Punteros – Programación. (Recuperado 19/01/2021) http://progra.usm.cl/apunte/c/punteros.html