Estructuras de Datos y Algoritmos I

Trabajo practico final – Wirzt, Santiago – 27/6/2019

El propósito del trabajo es proporcionar un programa en C, capaz de resolver problemas del tipo TSP (Travelling salesman problem).

Estructura:

La estructura usada en mi programa es de la forma:

```
typedef struct _Ciudades {
        int cantidad;
        char** nombres;
        int* matrizCostos;
        int costoViaje;
        int* movimientos;
} * Ciudades;
```

Ciudades es siempre un puntero porque nunca hago uso de la estructura localmente, sino que siempre prefiero utilizarla en memoria.

Los miembros de la estructura son:

cantidad: Es un int que lmacena la cantidad de ciudades que tiene esa estructura.

nombres: Es un array de **char*** que almacena los nombres de las ciudades. El índice donde se almacena cada nombre representa el número que se le asigna a cada ciudad en el momento de la lectura.

matrizCostos: Es un array de ints que representa una matriz cuadrada simétrica, donde la posición (x,y) representa el costo de viaje entra las ciudades x e y. Por defecto, todas sus posiciones tienen valor 0. Debido a su forma de array unidimensional para acceder a una posición se debe calcular el índice de la siguiente forma:

(Fila * n) + Columna o **(Columna * n) + Fila**, donde **n = cantidad**.

costoViaje: Que representa el costo total del viaje, por defecto es -1.

movimientos: Array de ints, que almacena el orden de proceder de la solución, donde cada número representa una ciudad y debe estar conectada con las ciudades adyacentes.

Resolución:

La resolución del problema se hace a partir de una función recursiva, que a base de fuerza bruta, alcanza el camino con el menor costo total. La función recibe argumentos:

void brute_force(Ciudades c, int* actual, int costoActual, int nivel) {

La funcion procede a probar todos los caminos posibles, pero siempre arrancando de la ciudad 0. Esto es debido a que en este problema en particular, la solución es circular. Lo cuál implica que, no importa de donde arranque, la solución es siempre análoga a arrancar de 0. Por esta razón, para evitar probar los mismos caminos en distinto orden, se toma a 0 como inicio siempre.

Durante la solucion se utiliza el valor **costoViaje** almacenado en la estructura Ciudades, ya que si el camino que se evalúa actualmente es mayor al costo de un viaje obtenido anteriormente, este se descarta. (Esto ocurre si **costoViaje != -1**).

Una vez finalizada la recursión, se tiene por hecho que la solución encontrada es la mejor porque todas las opciones fueron comprobadas.

En casos donde la cantidad de Ciudades es mayor a 15, decidí utilizar un método de heurística (nearest neighbour) donde busco un camino posible avanzando de ciudad en ciudad por el camino más barato. Esto no devuelve la mejor solución pero permite poner una cota superior rápidamente para eliminar caminos incorrectos rapidamente en la función **brute_force**.

Condiciones de ejecución:

El programa espera que el archivo de datos ingresados este correctamente escrito para evitar alentizar el proceso de entrada revisando cada situación de error.

Debido a esto, junto con el programa **main**, se incluye un **check**, que recibe el archivo de entrada e informa al usuario los errores que contiene.

Se espera que el usuario ejecute **check** antes que **main** debido que una entrada errónea puede provocar que este último funcione de forma indefinida.

Formas de ejecución:

Junto con los archivos, tambien se provee un makefile, que simplifica los pasos a la hora de compilar.

- Para compilar los programas se debe tener instalado **make**.

Si esta instalado, solo se debe escribir **make** con la consola ubicada en la carpeta del trabajo.

En caso contrario, debe instalarse o ejecutarse las siguientes líneas de forma manual:

gcc -c colasenlazadas.c -pedantic -Wall -std=c99
gcc -o main main.c colasenlazadas.o -pedantic -Wall -std=c99
gcc -c slist.c -pedantic -Wall -std=c99
gcc -o check revisaentrada.c slist.o -pedantic -Wall -std=c99

Una vez compilado, se debe ejecutar el programa de control de entrada:

./check <archivoEntrada>

Este imprimirá en consola todos los problemas encontrados, una vez solucionados se recomienda volver a ejecutar para comprobar que no se olvidó nada.

Cuando el programa anterior no imprima nada se puede proceder a ejecutar el main.

./main <archivoEntrada> <archivoSalida>

Si el programa encuentra algun percance durante su ejecución, este se imprimirá en consola, de lo contrario terminara sin decir nada.

La solución será almacenada en **<archivoSalida>** con el costo total en la última línea.