Guion de prácticas
El problema del viajante de comercio
Parte 4

Metodología de la Programación
Grado en Ingeniería Informática
Curso: 2016-2017

1. El Problema del Viajante de Comercio

El objetivo de esta práctica es agregar la funcionalidad básica para lectura y escritura de ficheros de texto. Utilizando la heurística del vecino más cercano, se resolverán varias instancias del problema de viajante de comercio y se guardarán dos soluciones por cada una de ellas.

Parte 4 y última

La lectura y la escritura de ficheros son tareas fundamentales en la mayoría de programas que utilizamos diariamente. Basta pensar en un editor de texto, o un programa de manipulación de imágenes. Para poder leer o escribir un fichero es imprescindible conocer su formato o estructura y su codificación.

En esta parte final de la práctica se trabajará con ficheros codificados como texto con una estructura simple. En partes anteriores ya se realizó la lectura de datos de una instancia del problema desde un fichero usando un constructor. Ahora también crearemos un método que lea el problema desde un fichero.

También se extenderá nuestra implementación para permitir la escritura en fichero de las coordenadas de las ciudades en el orden indicado por una solución. En el contexto de nuestro problema es necesario considerar quien debe encargarse de guardar la solución en un fichero. ¿Debería ser la clase Problema, la clase Solucion o una función externa?. En esta práctica optaremos por la última opción. Para ello, implemente una función externa cuya cabecera será

```
bool guardarSolucion(const char * nombre, const char * cxx, const Problema & p, const
Solucion & s)
```

La función guardará en el fichero `nombre`, la solución `s` al problema `p` correspondiente a la instancia leída desde el fichero `cxx`. Devolverá `true` si se ha guardado con éxito o `false` si ha habido algún error. El formato del fichero de salida será el siguiente:

```
NOMBRE DE LA INSTANCIA: cxx. Por ejemplo c5
TAMANIO: xx. Por ejemplo: 5
LONGITUD: (la que se haya calculado). Por ejemplo: 23
ORDEN
.....
aquí vendrá la lista de índices de ciudades de la solución
separadas por comas y espacio. Por ejemplo, si hay 5 ciudades,
3, 2, 5, 4, 1
.....
```

```
COORDENADAS
.....
aquí vendrán las coordenadas de las ciudades en el orden
correspondiente, cada pareja de coordenadas separadas por
coma y cada ciudad en una línea. Por ejemplo, si hay 5
ciudades una salida sería:
3, 2
2.5, 3.1
7, 2.3
3.2, 2.3
1, 1
1.2, 2.1
.....
```

El resto tareas a desarrollar se describen a continuación.

Clase Problema

Añadir un constructor por defecto (que crea un problema de tamaño 0).

Añadir un método `leer` que recibe el nombre de un fichero y lee una instancia de problema desde el disco. El método devolverá un booleano indicando si la lectura fue correcta. Note que si, en el momento de leer, el objeto actual tiene tamaño del problema distinto de cero, significa que el objeto ya se había utilizado. En este caso, se debe liberar la memoria antes de proceder a la lectura y construcción de la nueva matriz de distancias.

Programa Principal

Implementar un programa `tspFinal.cpp` que se pueda llamar de 2 maneras diferentes:

Con un parámetro

Se llamará como `tspFinal listado.tsp`, donde `listado.tsp` es un fichero de texto con N lineas donde cada línea siguiente indica el nombre de un fichero de instancia del problema. Un ejemplo de fichero es:

```
c10.tsp
c100.tsp
c52.tsp
```

El programa debe funcionar de la siguiente manera.

1. supongamos que la primera instancia se llama `cxx.tsp`. Crear un objeto de la clase Problema a partir del fichero `cxx.tsp`. Si el fichero no existe, se ignora (saltar al paso 4).
2. Obtener la mejor solución posible utilizando la heurística del vecino más cercano para ese problema.
3. Guardar dicha solución en el fichero `cxx.sol`.
4. Volver al paso 1 y repetir para el resto de las instancias indicadas en el fichero de entrada.

Con dos parámetros

La llamada con `tspFinal cxx.tsp mejor.sol`, donde `cxx.tsp` es una instancia del problema y `mejor.sol` es el nombre del fichero donde se debe guardar (codificada en texto según el formato antes indicado) la mejor solución encontrada para la instancia. En este caso, el programa funciona de la siguiente manera:

1. Crear un objeto de la clase Problema a partir del fichero `cxx.tsp`. Si el fichero no existe, se muestra un mensaje de error y se finaliza la ejecución.
2. Obtener la mejor solución posible utilizando la heurística del vecino más cercano
3. Guardar dicha solución en un fichero cuyo nombre será el indicado en el segundo argumento.

Consideraciones Generales

Tenga en cuenta que:

- Si la llamada al programa no coincide con ninguno de los dos casos descritos, se debe mostrar un mensaje que indique la sintaxis válida.
- Si la llamada es válida, se debe comprobar que los ficheros de indicados como parámetros existan y dar el correspondiente error si no es así. Si los ficheros de salida existen se sobrescribirán.

2. Material a Entregar

Para la entrega final, recuerde que las clases Punto, Problema y Solucion deben estar completas. Esto significa que tienen que contar con los métodos básicos de consulta y asignación (set/get), los constructores por defecto, constructores de copia y sobrecarga del operador de asignación.

Utilice los conceptos vistos en teoría y en las sesiones de prácticas: interfaz pública y privada de cada módulo, compilación separada, fichero makefile, etc, organizando todo lo necesario en un directorio que se llamará `tspFinal` con subdirectorios `src, include, obj, bin y doc`. Se recomienda guardar las clases Punto, Problema y Solución en una biblioteca llamada `libtsp.a`

Cuando esté todo listo y probado, deberá empaquetar la estructura de directorios en un archivo con el nombre `tspFinal.zip` y lo entregará en la plataforma decsai en el plazo indicado. No deben entregarse archivos objeto (.o) ni ejecutables (conviene ejecutar `make clean` antes de proceder al empaquetado).

Debe incluirse además, en el directorio `tspFinal/doc`, un documento en formato pdf (llamado `valoracion.pdf`) con una valoración, honesta y realista del trabajo realizado, indicando las dificultades o problemas encontrados durante la realización de esta práctica final.

El alumno debe asegurarse de que ejecutando las siguientes órdenes se compila y ejecuta correctamente su proyecto:

```
unzip tspFinal.zip
make
bin/tspFinal lista.tsp  // caso 1
bin/tspFinal c10.tsp mejor.sol  // caso 2
```

La ejecución de `valgrind --leak-check=full --track-origins=yes xxx` donde `xxx` es cada uno de los casos, debe finalizar con 0 errores.

3. Evaluación

La evaluación constará de los siguientes apartados

- Compilación y comprobación del funcionamiento del programa para cada uno de los casos posibles.
- Comprobación del uso de memoria mediante valgrind.
- Revisión del código.
- Defensa de la práctica.