Laboratorio - Tarea de implementación (2021-1)

23 de agosto de 2023

Índice

1. Objetivo

El objetivo de esta tarea es realizar una implementación de la resolución de un problema haciendo uso de modelado mediante grafos y de las técnicas algorítmicas tratadas en el curso para su exploración.

Se espera que además de cumplir con el orden de tiempo de ejecución solicitado, la implementación realizada también haga un buen uso de memoria.

2. Problema

El problema considerado es el planteado en el Ejercicio 3.10 de Kleinberg & Tardos, hallar la cantidad de caminos de largo mínimo entre 2 nodos de un grafo no dirigido G, con la particularidad de que para este laboratorio se pide encontrar la cantidad de caminos de largo mínimo entre un vértice distinguido s y cada uno de los restantes vértices del grafo.

Observar que el grafo no dirigido G considerado en el problema **puede no ser conexo**.

3. Materiales

El material se encuentra en el archivo `TareaP3-2021-1-GRAFOS.tar.gz` que está disponible en la sección **Laboratorio** del sitio EVA del curso. Se desempaqueta usando el comando `tar`:

```
$ tar zxvf TareaP3-2021-1-GRAFOS.tar.gz
```

Se obtiene un directorio `Tarea1(2021)` con lo siguiente:

- El subdirectorio `include` con los archivos de definición de los TADs Grafo, Lista, Cola, y Pila, cuya implementación se provee en los archivos del subdirectorio `src`. Los TADs pueden ser utilizados en la implementación realizada. El subdirectorio también contiene el archivo de definición del módulo `caminos`, así como un archivo `utils.h` (que no tiene .cpp correspondiente) donde se definen tipos utilizados en los TADs.
- El archivo `Makefile` con reglas para la compilación y testing (ver sección 6.2).
- El archivo `principal.cpp` (programa principal).
- El subdirectorio `test` que contiene los casos de prueba para `principal.cpp`.
- El archivo `LetraTarea1(2021).pdf` con la letra del laboratorio.

4. ¿Qué se pide?

Dar una implementación de un algoritmo que resuelve el problema, cuyo tiempo de ejecución sea $O(m + n)$. Recordar que en el problema, n es la cantidad de vértices y m es la cantidad de aristas del grafo no dirigido G.

Se debe implementar dicho algoritmo en el archivo `caminos.cpp` (**con este nombre y extensión**) que implementa la función declarada en `caminos.h`. Observar que en caso de hacer uso de los TADs proporcionados, sus implementaciones **no deben ser modificadas**.

5. Lenguaje y sistema operativo

Sistema operativo

El sistema operativo es la versión de `Linux` instalada en las máquinas de la Facultad.

Lenguaje

El lenguaje que se utiliza es C*, que es una extensión de C con las siguientes funcionalidades de C++:

- Funciones `new` y `delete`.
- Pasaje por referencia.
- Tipo `bool`.
- Definiciones de variables de tipo `struct` o `enum` al estilo C++.

Herramientas

Las herramientas que se deben tener instaladas para poder desempaquetar, compilar, ejecutar y testear el laboratorio son: `tar`, `gzip`, `g++`, `make`, `valgrind` y `diff`.

Compilador

El compilador es la versión de g++ instalada en las máquinas `Linux` de la Facultad.

6. Apéndice

6.1. Verificación

La implementación de código es solo una etapa del desarrollo de software. Tras la implementación de cada módulo, de cada función, se debe verificar que se está cumpliendo con los requerimientos establecidos para las funcionalidades implementadas.

Se incluyen como ejemplos algunos casos de prueba (que **NO** necesariamente serán los únicos utilizados para la corrección). En ellos cada archivo `.in` representa lo leído en la entrada estándar y el correspondiente archivo `.out` lo que se escribe en la salida estándar. Se pueden redireccionar la entrada y salida estándar con los operadores de redirección < y > respectivamente. Por lo tanto se puede ejecutar `principal` con cada archivo `.in` redirigiendo la salida hacia otro archivo (por ejemplo, con extensión `sal`) que luego se compara con el archivo `.out` correspondiente. La comparación se hace con la utilidad `diff`. Si los archivos comparados son iguales no se imprime nada en la salida, por lo que si la salida de la ejecución de `diff` se redirige hacia un archivo, este tendrá tamaño 0. Se muestra un ejemplo de ejecución y comparación exitosa:

```
$ ./principal < test/01.in > test/01.sal
$ diff test/01.out test/01.sal > test/01.diff
```

6.2. Makefile

Para automatizar el proceso de desarrollo se entrega el archivo `Makefile` que consiste en un conjunto de reglas para la utilidad `make`.

Cada regla consiste en un *objetivo*, las *acciones* para conseguir el objetivo y las *dependencias* del objetivo. Cuando el objetivo y las dependencias son archivos, las acciones se ejecutan cuando el objetivo no está actualizado respecto a las dependencias (o sea, es un archivo que no existe o su fecha de modificación es anterior a la de alguna de las dependencias). Por más información ver el manual de `make` y el Instructivo Makefile que está en la Sección **Laboratorio** del sitio `web` del curso.

En el `Makefile` entregado las reglas incluidas son:

- `principal`: para compilar y enlazar.
- `clean`, `clean_bin` y `clean_test`: para borrar archivos.
- `testing`: para hacer pruebas.

make principal La regla `principal` compila y luego genera el ejecutable `principal`. Esta regla es la predeterminada, o sea que es la que se invoca si no se especifica ninguna.

make testing Con la regla `testing` se ejecuta el programa con los casos de entrada (`in`) generando archivos con la extensión `sal` y estos se comparan con las salidas esperadas (`out`), obteniendo archivos con extensión `diff`.

El hecho de que en la salida no haya referencias al proceso de ejecución y comparación indica que las comparaciones fueron exitosas.

```
$ make testing
./principal < test/00.in > test/00.sal
./principal < test/01.in > test/01.sal
./principal < test/02.in > test/02.sal
./principal < test/03.in > test/03.sal
./principal < test/04.in > test/04.sal
./principal < test/05.in > test/05.sal
```

Una nueva ejecución no hará nada:

```
$ make testing
```

Si los archivos a comparar no son iguales se indica en la salida estándar. Y se puede comprobar que en el directorio `test` los archivos `diff` no tienen tamaño 0.

```
$ make testing
./principal < test/00.in > test/00.sal
---- ERROR en caso test/01.diff ----
./principal < test/02.in > test/02.sal
./principal < test/03.in > test/03.sal
---- ERROR en caso test/03.diff ----
./principal < test/04.in > test/04.sal
./principal < test/05.in > test/05.sal
-- CASOS CON ERRORES --
03
01
```

```
$ stat --print="%n  %s \n" test/*.diff
%test/00.diff  0
%test/01.diff  59
%test/02.diff  0
%test/03.diff  143
%test/04.diff  0
%test/05.diff  0
%
```

Observación: la ejecución anterior es una simplificación de la ejecución completa que se muestra a continuación.

```
$timeout 6 valgrind -q --leak-check=full ./principal < test/00.in > test/00.sal 2>&1
```

Donde `timeout 6` significa que la ejecución del caso se cortaría en caso de demorar más de 6 segundos y `valgrind -q -leak-check=full` es la aplicación que analiza la correcta utilización de la memoria.

make clean En ocasiones puede ser útil borrar los archivos generados. Esto se hace con `make clean`. Si sólo se desea borrar los archivos generados por la compilación se debe invocar `make clean_bin`. Para borrar sólo los archivos generados por la ejecución de `principal` se debe invocar `make clean_test`.

6.3. assert

Para realizar diagnósticos durante el proceso de desarrollo se puede usar la macro `assert`, que toma como parámetro una expresión booleana. Si el resultado de la expresión es `true` no hace nada, pero si el resultado es `false` termina la ejecución del programa indicando el error. Se puede usar para verificar el cumplimento de pre o post condiciones, de invariantes en un bucle, etc. Para usarla se debe incluir el archivo de encabezamientos correspondiente:

```
#include <assert.h>
```

Ejemplo: Mediante el código

```
int cant_scanf = scanf("%s", nom_com);
assert(cant_scanf == 1);
```

se controla que la lectura desde la entrada estándar asigne exactamente un valor a una variable.

El uso de `assert` debe limitarse a la etapa de desarrollo porque enlentece la ejecución del programa. Por ejemplo, si se quiere comprobar que se cumple una precondición de pertenencia a una lista se puede incluir:

```
assert(pertenece_a_lista(num, lst));
```

Pero esto implica una recorrida a una lista, que no es necesaria, ya que debe asumirse como precondición.
Para evitar tener que remover del código todas las instancias de `assert` se dispone de la directiva `DNDEBUG` como opción de compilación. Su uso se puede ver en la línea 56 del `Makefile`.

Precaución. La expresión pasada como parámetro a `assert` no debe tener efectos secundarios, ya que estos no se producirán cuando se compile con la directiva `DNDEBUG`. La siguiente invocación será removida del código al incluir esta directiva al compilar:

```
assert(scanf("%s", nom_com));
```

Por lo tanto se remueve la lectura y asignación del valor de la variable. La forma correcta de uso es la que está al inicio de esta sección.