Algoritmos y Estructuras de Datos.
TPLSR16. Recuperatorio de Trabajos Prácticos de Laboratorio. [2017-02-09]

PASSWD PARA EL ZIP: X5NQ HJ6Q F6KD

Ejercicios

ATENCION: Deben necesariamente usar la opción `-std=gnu++11` al compilador, **si no no va a compilar**.

[Ej. 1] **[filter-sets]** Dada una lista de conjuntos `LS_IN` de enteros, y una función `int f(int)`, copiar en otra lista de conjuntos `LS_OUT` solo aquellos conjuntos para los cuales la función `f` es inyectiva (esto es, que genera imágenes diferentes para todos los elementos del conjunto).
Ejemplo: Si `LS_IN=[ (1,2,3), (2,-2,10), (0,5) ]` y `f` es `f(x)=x^2` entonces `LS_OUT` debería quedar `[ (1,2,3), (0,5) ]`, ya que para el segundo conjunto de la lista de entrada hay dos elementos (`2` y `-2`) que generan la misma imagen (`4`), razón por la cual ese conjunto no se omite.
Ayuda:

- Por cada conjunto de `S` de `LS_IN`
  - Generar un conjunto auxiliar con las imágenes de la función aplicada a sus elementos
  - Comparar los tamaños de ambos conjuntos para determinar si es debe insertar `S` en `LS_OUT`.

[Ej. 2] **[distancia]** Dado un grafo no ponderado `G`, un vértice de partida `x` y un vértice de llegada `y`, implementar el método `int distancia(map<int,list<int>>& G, int x, int y)` que retorna la distancia entre los vértices pasados como parámetros. Recordar que la distancia entre dos vértices de un grafo no ponderado se puede definir como el número de aristas del camino más corto que los une. Se asegura que los vértices pasados como argumentos pertenecen al grafo.
Ayuda:

- Chequear si el vértice actual es el vértice de llegada
- En caso positivo retornar la distancia acumulada hasta el momento
- En caso negativo:
  - Marcar el vértice como visitado
  - Lanzar la recursion a cada uno de los vértices vecinos aún no visitados incrementando la distancia actual en 1. Retornar la menor distancia hallada.
  - Si todos los vecinos ya fueron visitados retornar un valor de distancia muy grande (1000) que indique que no se pudo hallar el vértice de llegada por este camino.

Por ejemplo: Si `G[1] -> {2 ,3}, G[2] -> {1 ,3}, G[3] -> {1 ,4}, G[4] -> {3}` debe dar,

```
distancia(G,1,1) -> 0
distancia(G,2,4) -> 2
distancia(G,1,4) -> 2
distancia(G,4,3) -> 1
```

[Ej. 3] **[suma-en-la-hoja]** Implemente el método `void suma_en_la_hoja(tree<int>& T)` que agregue un hijo a cada hoja del árbol ordenado orientado `T` cuya etiqueta sea la suma de las etiquetas de los vértices que componen el camino desde la raíz hasta dicha hoja. Un árbol vacío retorna un árbol vacío.

Ejemplo: Si `T = (5 (2 1 3) 7)` , luego de `suma_en_la_hoja(T)` el árbol debe quedar:
`(5 (2 (1 8) (3 10)) (7 12))`
Ayuda:

- Implementar una función recursiva que reciba el árbol `T`, un iterador `it`, y un entero `s` que deberá contener la suma de las etiquetas del camino desde la raíz hasta el padre del nodo apuntado por `it`.
  - Si `it` apunta a un nodo válido, la función debe sumar a `s` la etiqueta de `it` e invocarse recursivamente para cada hijo.
- Si `it` apunta a un nodo lambda, la función debe insertar la suma en dicha posición. La recursión comienza con `it` apuntando a la raíz y `s=0`.

Instrucciones generales

- El examen consiste en que escriban las funciones descriptas más arriba; impleméntandolas en C++ de tal forma que el código que escriban **compile y corra correctamente**, es decir, no se aceptará un código que de algún error de compilación o que tire alguna excepción/señal de interrupción en runtime. Básicamente se hace una evaluación de caja negra, aunque le daremos un rápido vistazo al código.
- Salvo indicación contraria pueden utilizar todas las funciones y utilidades del estándar de C++ que por supuesto contiene a la librería STL.
- Se incluye un template llamado `program.cpp`. En principio sólo tienen que escribir el cuerpo de las funciones pedidas.
- Para cada ejercicio hay dos funciones de evaluación, por ejemplo si `f` es la función a evaluar tenemos

```
ev.eval<j>(f,vrbs);
hj = ev.evalr<j>(f,seed); // para SEED=123 debe dar Hj=170
```

  `j` es el número de ejercicio, por ejemplo para el ejercicio 1 tenemos las funciones (`eval<1>` y `evalr<1>`). La primera `ev.eval<j>(f,vrbs);` toma una serie de casos de prueba de entrada, le aplica la función del usuario `f` y compara la salida del usuario (`user`) con respecto a la esperada (`ref`). Si la verbosidad (el argumento `vrbs`) se pone en uno, entonces la función evaluadora reporta por consola los datos de entrada, la salida de la función de usuario y la salida esperada

```
m: 10, k: 3
T(ref): (10 (7 (4 1) 1) (4 1) 1)
T(user): (10 (7 (4 1) 1) (4 1) 1)
EJ1|Caso0. Estado: OK
```

- `ucase`: Además las funciones `eval()` tienen dos parámetros adicionales:
  `Eval::eval(func_t func,int vrbs,int ucase);`
  El tercer argumento 'ucase' (caso pedido por el usuario), permite que el usuario seleccione uno solo de todos los ejercicios para chequear. Por defecto está en `ucase=-1` que quiere "hacer todos". Por ejemplo `ev.eval4(prune_to_level,1,51);` corre sólo el caso 51.
- **Archivo con casos tests JSON:** Los casos test que corre la función `eval<j>` están almacenados en un archivo `test1.json` o similar. Es un archivo con un formato bastante legible. Abajo hay un ejemplo. `datain` son los datos pasados a la función y `output` la salida producida por la función de usuario. `ucase` es el número de caso.

```
{ "datain": {
  "T1": "( 0 (1 2) (3 4 5 6) )",
  "T2": "( 0 (2 4) (6 8 10 12) )",
  "func": "doble" },
```

```
"output": { "retval": true },
"ucase": 0 },
```

- La segunda función **evalr<j>** es el chequeo que llamamos **SEED/HASH**. La clase evaluadora genera una serie de contenedores a partir de la semilla **seed**, se los pasa a la función del usuario **f()**. Las respuestas de la **f()** van siendo procesadas por la función interna de hash que genera un **checksum H** de las respuestas. Por ejemplo para el primer ejercicio si **seed=123** entonces el checksum es **H=523**. Una vez que el alumno termina su tarea se le pedirá que corra la función **evalr<j>()** de la clase evaluadora con un valor determinado de la semilla **seed** y se comprobará que genere el valor correcto del checksum **H**.
  Desde el punto de vista del alumno esto no trae ninguna complicación adicional, simplemente debe llenar el parámetro **seed** con el valor indicado por la cátedra, recompilar el programa y correrlo. La cátedra verificará el valor de salida de **H**.
- En la clase evaluadora cuentan con funciones utilitarias como por ejemplo:
  void Eval::dump(list <int> &L,string s=""): Imprime una lista de enteros por **stdout**. Nota: Es un método de la clase **Eval** es decir que hay que hacer **Eval::dump(VX);**. El string **s** es un label opcional.
  - **void Eval::dump(list <int> &L,string s="")**
- Después del parcial deben entregar el programa fuente (sólo el **program.cpp**) renombrado con su apellido y nombre (por ejemplo **messilionel.cpp**). Primero el apellido.