Algoritmos y Estructuras de Datos.
TPL2. Trabajo Práctico de Laboratorio. [2015-10-03]

PASSWD PARA EL ZIP: K69X H8G3 LP9X

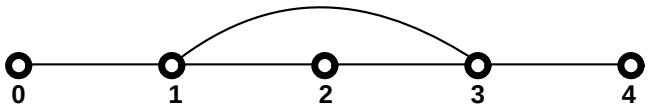
Ejercicios

[Ej. 1] **[getpath (25pt)]** Escribir una función
`void getpath(map<int,list<int> > &M, int vrtx, list<int> &path);` que retorna un camino `path` en el grafo simple definido por `M` que comienza en `vrtx` y no repite ningún vértice. El algoritmo debe utilizar una estrategia ávida (no recursiva) de la siguiente manera:

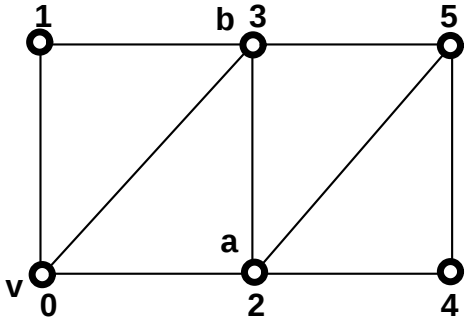
- Insertar `vrtx` en el `path`.
- Buscar entre los vecino del último nodo insertado (y que a su vez es el último en `path`), el primero que aún no se encuentre en `path`, e insertarlo.
- Repetir el paso previos hasta que no se encuentre ningún vecino que aún no haya sido insertado.

Ayuda: Es conveniente gestionar un `vector<bool> visited` que indique si el vértice `i` ya fue incluído en `path` o no (`visited[i]` es `true` si ya fue visitado o viceversa).
Por ejemplo, para el grafo de la figura, representado de la siguiente forma
`M={0->(1),1->(3,0,2),2->(1,3),3->(1,4,2),4->(3)}`
entonces si `vrtx=3` debe retornar `path=(3,1,0)`, y (para el mismo grafo)

- `vrtx=0, => path=(0 1 3 4)`
- `vrtx=1, => path=(1 3 4)`
- `vrtx=2, => path=(2 1 3 4)`
- `vrtx=4, => path=(4 3 1 0)`



[Ej. 2] **[tricount (35pt)]** Escribir una función
`int tricount(map<int,list<int> > &M);` que retorna la cantidad de triángulos en un grafo simple `M`. Es decir los vértices del grafo representan puntos en una hoja de papel, y las aristas segmentos que unen pares de puntos. Escribir una función para determinar cuantos triángulos hay dibujados. Por ejemplo en el grafo de la figura debe retornar 4.
`M={0->(3,1,2),1->(3,0),2->(5,0,4,3),3->(2,1,0,5),4->(2,5),5->(2,3,4)}`



Ayuda: Para cada vértice `v` del grafo tomar la lista de vecinos `v` y

- Inicializar un contador a `0`.

- Con dos iteradores recorrer todos los pares posibles **{a,b}** de la lista **v**.
- Para cada par **{a,b}**, verificar si **a** y **b** son vecinos entre sí. En caso afirmativo, incrementar el contador.

Al terminar de recorrer todos los vértices el número final debe ser 6 veces el número de triángulos, ya que cada triángulo tiene 3 aristas y cada arista es contada dos veces.
Para chequear si el vértice **b** es vecino de **a** se sugiere se sugiere escribir una función auxiliar **`bool contiene(list<int> &ngbrs,int b);`** que determina si el vértice **vrtx** está en la lista de vecinos **ngbrs**.
Nota sobre el recorrido de los pares: Para recorrer todos los pares de enteros **j,k** con valores entre **[0,M)** lo podemos hacer de dos formas.

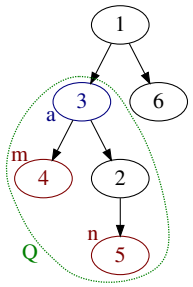
```
for (int j=0; j<M; j++) { // VERSION 1
  for (int k=0; k<M; k++) {
    // Inspecciona el par j,k...
  }
}
```

```
for (int j=0; j<M-1; j++) {  // VERSION 2
  for (int k=j+1; k<M; k++) {
    // Inspecciona el par j,k...
  }
}
```

La versión 2 garantiza que **j!=k** y que se pasa una sola vez por el mismo par, por ejemplo se pasa por **(2,5)** pero no por **(5,2)**. Tener en cuenta que si se recorre de la forma 2 se cuenta 1 sola vez cada arista, de manera que el contador final será **3 veces el número de triángulos y no 6**.

[Ej. 3] **[min-com-subtree (35pt)]** Dados dos nodos **m,n** de un arbol **T**, extraer el mínimo subárbol **Q** que contiene a **m**, y a **n**, es decir el árbol del primer antecesor común de ambos **a**. Por ejemplo en el caso que **T=(1 (3 4 (2 5)) 6)**, y ***m=4,*n=5** debe retornar el subárbol de 3, ya que es el primer antecesor común de ambos. Es decir debe retornar **Q=(3 4 (2 5))**.
Nota 1: Asumimos **`typedef tree<int> tree_t; typedef tree<int>::iterator node_t;`**
Nota 2: Se garantiza que **m** y **n** **no tienen relación de antecesor/descendiente** entre ellos, es decir **a!=m** y **a!=n**.
Consigna: Escribir
`void min_com_subtree(tree_t &T,node_t m,node_t n, tree_t &Q);`
Ayuda:

- Escribir una función auxiliar recursiva **`list<node_t> nodepath(tree_t &T,node_t n,node_t q);`** que retorna una lista de nodos que corresponde al camino que va desde **n** hasta el nodo **q** si **q** está en el subárbol de **n**. Si el nodo **q** no está en el subárbol entonces debe retornar la lista vacía. Hacerlo en forma recursiva.
  - Si **n** es Λ entonces debe retornar la lista vacía.
  - Si **n==q** debe retornar simplemente una lista conteniendo al nodo **n** (cortar recursión).
  - Si no es, llamar recursivamente a **nodepath(T,c,q)** para todos los hijos **c** de **n**. Si alguno de ellos retorna un camino no vacío entonces debe retornar esa lista, prependizando el nodo **n**.
- Usar **nodepath()** para hallar los caminos **mpath** y **npath** a **m,n**.
- Buscar el primer antecesor **a** común de **m,n** (en el ejemplo ***a=3**). Para esto ir recorriendo ambas listas con iteradores **am,an** hasta encontrar el primer par que es diferente ***am=*an**. El anterior es **a**. Como **m** y **n** no tienen relación de antecesor/descendiente está garantizado que se encontrará este antecesor común **a**, y que no es ni **a=m**, ni **a=n**.

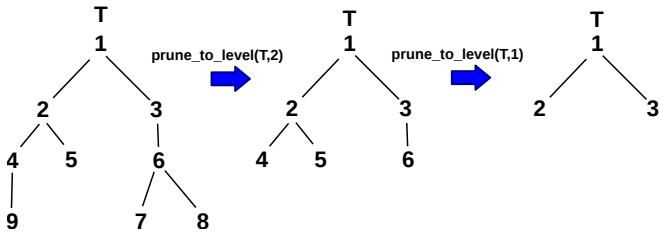


- Mover a **Q** el subárbol de **a** usando **`tree::splice()`**.

Notas:

- El algorimo descripto en la ayuda modifica `T` (es **destructivo**). (Notar que está OK, ya que la consigna no indica lo contrario).
- Notar que los caminos son **listas de nodos** en el árbol, por lo tanto si `am` es un iterator en `mpath`, `*am` es un nodo en el árbol `T` y `**am` el valor contenido en el árbol. Para la consigna pedida no hace falta inspeccionar los valores contenidos en los nodos. Por ejemplo al buscar el antecesor común sólo se deben comparar los nodos (`*am,*an`, no sus contenidos (`**am,**an`).

[Ej. 4] **[prune-to-level (25pt)]** Escribir una función `void prune_to_level(T,lev);` que poda todas los nodos de un árbol por debajo del nivel `lev` *in-place*. Por ejemplo si al árbol `T` de la izquierda le aplicamos `prune_to_level(T,2)` debe quedar el del centro y si le aplicamos `prune_to_level(T,1)` debe quedar el de la derecha.



Más ejemplos:

- `T=(4 (4 1 5) (0 (2 4) 2)), lev=4, => T=(4 (4 1 5) (0 (2 4) 2))`
- `T=(4 (4 1 5) (0 (2 4) 2)), lev=3, => T=(4 (4 1 5) (0 (2 4) 2))`
- `T=(4 (4 1 5) (0 (2 4) 2)), lev=2, => T=(4 (4 1 5) (0 2 2))`
- `T=(4 (4 1 5) (0 (2 4) 2)), lev=1, => T=(4 4 0)`
- `T=(4 (4 1 5) (0 (2 4) 2)), lev=0, => T=(4)`

Ayuda:

- Escribir una función auxiliar recursiva `void prune_to_level(T,n,lev);`
- Si `n` es Λ no debe hacer nada.
- Si `lev>0` debe aplicar recursivamente la función a cada uno de los hijos con `lev-1`
- Si `lev==0` debe eliminar todos los hijos.

Nota: La función debe operar correctamente para cualquier `lev`. Si es negativo debe dejar el árbol vacío, si es `lev==0` debe dejar sólo la raíz, y si es mayor que la profundidad del árbol no debe hacer nada.

Instrucciones generales

- El examen consiste en que escriban las funciones descriptas más abajo; 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. El paquete ya incluye el header `tree.h`.
- Para cada ejercicio hay dos funciones de evaluación, por ejemplo si `f` es la función a evaluar tenemos

```
ev.evalj(f,vrbs);
hj = ev.evalrj(f,seed); // para SEED=123 debe dar Hj=170
```

j es el número de ejercicio, por ejemplo para el ejercicio 1 tenemos las funciones (**eval1** y **evalr1**). La primera **ev.evalj(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
```

- La segunda función **evalrj** 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 **evalrj()** 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="")**
  - **void Eval::dump(list< list<int> > &LL,string s="")**.
  - **tree<int>::lisp_print()**: Lisp print de un árbol ordenado orientado (AOO). Nota: esta pertenece a la clase **tree**. Uso: **tree<int> T; T.lisp_print();**
  - Idem para AB: **btree<int>::lisp_print()**: Lisp print de un árbol binario (AB). Nota: esta pertenece a la clase **btree**. Uso: **btree<int> T; T.lisp_print();**
- Después del parcial deben entregar el programa fuente (sólo el **program.cpp**) renombrado con su apellido y nombre (por ejemplo **messilionel.pdf**). Primero el apellido.
- **Puntos:** Notar que la suma de los puntos es 120. La nota del parcial **min(sum(Zj),100)** es decir que si haciendo los tres primeros ejercicios se obtienen 95/100 pts. Para aprobar basta hacer 50/100 pts de manera que basta con cualquiera dos de los cuatro.
- **usercase**: Ahora las funciones **eval()** tienen dos parámetros adicionales:
  Eval::eval(func_t func,int vrbx,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.
- **Torneo de programación:**
  - Los ejercicios 2 (**tricount**) y 3 (**min_com_subtree**) participan del **Torneo**. La tabla de posiciones para el torneo se determina a partir del tiempo que tarda en correr el programa en **50 casos muy grandes**, (por ejemplo 50 grafos aleatorios de 250.000 vértices en el caso de **tricount**).
  - Para ello la función de evaluación por **seed/hash evalrj()** tiene un parámetro adicional **int hardness** (dificultad) que puede tomar valores **0<=hardnes<=5**, **Eval::evalr2(tricount,seed,vrbs,hardness);**.

- **`hardness=0`** (valor por defecto) corresponde a lo que se toma en el TPL para determinar el funcionamiento correcto, y por lo tanto los contenedores son pequeños.
- Para **`hardness>0`** el tamaño va creciendo geométricamente y la función reporta el tiempo que tardó en realizar la tarea. Para **`hardness=5`** corre el tamaño máximo de los contenedores. Por ejemplo la corrida para los 5 valores de **`hardness`** reportan

  ```
  EJ2. Hardness 1, elapsed 0.00811529[s]
  EJ2. Hardness 2, elapsed 0.0350215[s]
  EJ2. Hardness 3, elapsed 0.239452[s]
  EJ2. Hardness 4, elapsed 1.57176[s]
  EJ2. Hardness 5, elapsed 10.8449[s] (mips 1.08925, relative 9.95625[s])
  ```

  `elapsed` es el tiempo transcurrido. De esta forma Uds pueden probar diferentes alternativas de programación y ver si reducen o no el tiempo.
- Para definir los resultados del torneo se correrán todos los programas de los participantes en un **servidor dedicado**, por lo tanto no habrá tiempos de diferencia debido al procesador.
- Para **`hardness=5`** se reportan también los **`mips`** de la máquina (una medida de su eficiencia) y el tiempo relativo (o sea el tiempo real dividido los mips). De esta forma se tiene una medida de la eficiencia del algoritmo independiente de la rapidez del procesador.
- Como referencia, Para los ejercicios 2 y 3 de este TPL los valores obtenidos con un código implementado en base a la ayuda que hemos dado da los siguientes tiempos.

  ```
  EJ2. Hardness 5, elapsed 11.1808[s] (mips 1.06821, relative 10.4668[s])
  EJ3. Hardness 5, elapsed 5.57989[s] (mips 1.08289, relative 5.15276[s])
  ```

`mstorti@galileo/parc1-2015-76-g3da893d/Fri Oct 9 09:15:11 2015 -0300`