Córdoba 28 de Julio 2023

Examen Final Laboratorio

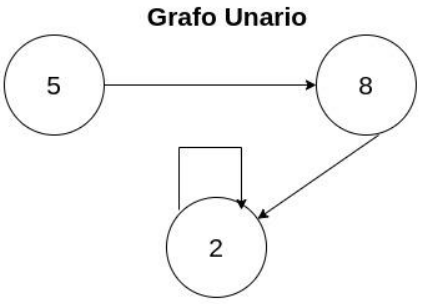
Algoritmos y Estructura de Datos II

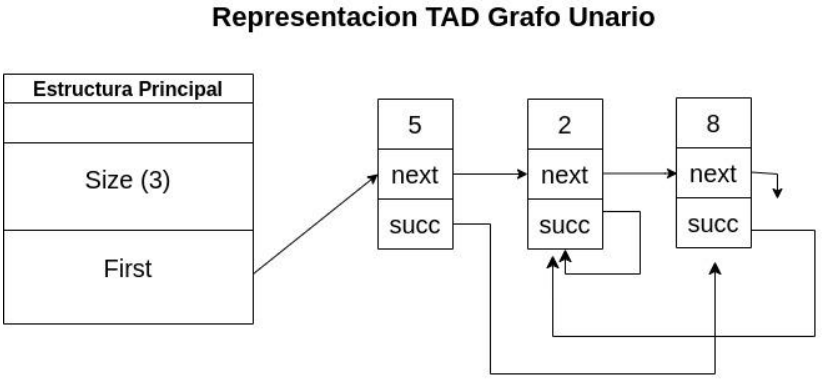
TAD Grafo Unario

Implementar el TAD ***grafo unario de enteros*** que representa un grafo dirigido cuyos vértices son valores enteros y donde cada vértice tiene a lo sumo una única arista dirigida a otro vértice del grafo. Es decir, que un mismo vértice “v” puede ser apuntado por varios vértices del grafo pero “v” solo puede apuntar a un único vértice del grafo, llamado sucesor de “v”, o a ninguno. Formalmente, recordar que un grafo dirigido G es un par G = (V,E) donde V es un conjunto de vértices (no hay vértices repetidos, ni noción de orden) y E es un conjunto de pares de la forma (v1,v2) que representa que existe una arista dirigida que parte del vértice v1 y llega al vértice v2.

Se debe representar el conjunto de vértices del grafo mediante nodos simplemente enlazados, usando una estructura principal que tenga un puntero a un primer vértice y que permita obtener la cantidad de vértices del grafo (size) en orden 1. Por otro lado, se debe representar la existencia de una arista del grafo con un segundo puntero del vértice de partida al vértice de llegada.

La implementación debe preservar la unicidad de vértices en el grafo (i.e., no hay vértices repetidos). A continuación, se muestra un ejemplo particular de grafo unario con tres vértices G = ({5,8,2} , {(5,8), (8,2), (2,2)}) y más abajo su correspondiente representación en el TAD grafo unario.





Las operaciones del TAD Grafo Unario se listan a continuación (algunas funciones ya se encuentran total y/o parcialmente implementadas para ayudarlos pero no se garantiza del todo que estén libres de bugs, así que deben revisar su implementación):

Función	Descripción
ugraph ugraph_empty(**void**)	Crea un grafo unario vacío (sin vértices y sin aristas).
ugraph ugraph_add_vertex(**ugraph** g, **vertex** v)	Agrega el vértice **v** al grafo unario g. Si v ya estaba presente, el grafo unario queda intacto.
ugraph ugraph_add_edge(**ugraph** g, **vertex** v1, vertex v2)	Agrega una arista de v1 a v2 siempre que ambos vértices existan en el grafo unario). Si v1 ya tenía una arista de salida entonces la reemplaza/actualiza por esta última.
bool ugraph_is_empty**(ugraph** g**)**	Indica si no hay vértices en el grafo unario g.
unsigned int ugraph_size(**ugraph** g)	Devuelve la cantidad de vértices que hay en el grafo unario en tiempo constante.
bool ugraph_member(**ugraph** g, **vertex**)	Indica si el vértice **v** se encuentra en el grafo unario g.
vertex ugraph_succ**(ugraph g, vertex v)**	Devuelve el único vértice sucesor de v si existe, i.e., el vértice v' tal v → v'.
ugraph ugraph_delete_incoming_edge**(ugraph g, vertex v**)	Elimina todas las aristas del grafo que apuntan al vértice v. Si v no está en el grafo entonces el grafo queda intacto.
ugraph ugraph_delete_outcoming_edge(**ugraph g, vertex v**	Elimina la única arista del grafo que parte del vértice v. Si v no estaba en el grafo entonces el grafo queda intacto.
ugraph ugraph_delete(**ugraph** g, **vertex** v)	Elimina el vértice v del grafo unario g (eliminado también todas las flechas que apunta a él si lo hubiere). Si v no estaba en el grafo unario entonces el grafo queda intacto.
void ugraph_dump**(g)**	Imprime por pantalla el grafo unario g de modo legible.
ugraph ugraph_destroy(**ugraph** g)	Destruye el grafo unario y libera toda la memoria usada.
bool ugraph_path**(ugraph** g, vertex v1,vertex v2**)**	Indica si existe un camino en el grafo g que parte del vértice v1 y termina en el vértice v2, i.e, si existen w1, …, wk vértices tales que v1->w1-> …. ->wk->v2.
bool ugraph_from_file**(const char *filepath)**	Construye una instancia del TAD a partir de un archivo de datos.

El programa resultante no debe dejar *memory leaks*.
Para verificar que la implementación del TAD funciona correctamente, se provee el programa (**main.c**) mediante el cual se pueden testear las funciones del TAD, cargando un grafo unario desde los archivos que se encuentran en la carpeta de inputs. Una vez compilado el programa puede probarse ejecutando:

```
$ ./ugraph_main input/example--005.in
```

Obteniendo como resultado:

```
Vertexes: 8 1 4 2 10
Edges:
      8 --> 10
      1 --> 8
      4 --> 8
      2 --> 4
      10 --> 2

El grafo unario SI tiene un camino del vertice 2 al vertice 10.
Si eliminamos el vertice 8, el grafo unario NO tiene un camino del
vertice 2 al vertice 10.
```

Consideraciones:

- Se provee el archivo **`Makefile`** para facilitar la compilación.
- Se recomienda usar las herramientas **`valgrind`** y **`gdb`**.
- Si el programa no compila, no se aprueba el final.
- Los *memory leaks* bajan puntos
- Entregar un código muy improlijo puede restar puntos.
- Si **`ugraph_size(ugraph g)`** no es de orden constante *O(1)* baja muchísimos puntos
- Para aprobar **se debe** hacer una invariante que cheque las tres propiedades fundamentales de la representación del TAD.
- **Solo se entregará ugraph.c**

Ejercicio extra solo para alumnos LIBRES

Se debe implementar la función que chequea si el grafo unario tiene un ciclo. *Ayuda: se puede definir esta función muy sencillamente utilizando alguna otra función del TAD (es decir, modularizando esta funcionalidad).*

`bool` `ugraph_cycle(`**`ugraph`** `g)`	Indica si el grafo g tiene un ciclo, i.e., si existen v, w1, …, wk vértices tales que v->w1-> …. ->wk->v.

Luego, descomentar en el archivo “main.c” el bloque de código “Testing Extra para alumnos Libres” y corremos nuevamente el programa:

```
$ ./ugraph_main input/example--005.in
Vertexes: 8 1 4 2 10
Edges:
      8 --> 10
      1 --> 8
      4 --> 8
      2 --> 4
      10 --> 2
El grafo unario SI tiene un camino del vertice 2 al vertice 10.
Si eliminamos el vertice 8, el grafo unario NO tiene un camino del
vertice 2 al vertice 10.
El grafo unario SI tiene ciclo.
Si eliminamos el vertice 8, el grafo unario NO tiene ciclo.
```