TIP: Implementación de algoritmos y estructuras de datos eficientes para grafos

Alejandro Merlo Griselda Cardozo

(draft 18 de noviembre de 2015)

Índice

1. Resumen

2. Introducción

Decir qué es un grafo como objeto matemático (representa una relacion). Un grafo es un par $G = (V, E)$ donde V son vertices y E son pares de vértices. Los grafos se usan para modelar objetos

Figura de un grafo en el que mostramos las definciones

Figura 1: Grafo cuyo conjunto de vertices es $V =$ y cuyo conjunto de aristas es $E =$. En este grafo.

de aplicaciones reales en forma abstracta. Por ejemplo, en una red social blah. La ventaja de los grafos es que proveen una terminologia comun, blah. La teoría de grafos se encarga del estudio de grafos y, en particular, la teoria algoritmica de grafos se encarga de los problemas computacionales asociados.

Este trabajo surge de la necesidad de verificar la viabilidad y dificultad de implementar la estructura h-grafo descripta por Lin et al. en []"Arboricity, h-index, and dynamic algorithms". Esta estructura implementa el *tipo abstracto de datos* (TAD) Grafo. Esta estructura esta pensada para aplicaciones que utilizan grafos dinámicos. Por *dinámico* nos referimos a que permite la inserción y borrado de vértices y aristas, a la vez que provee funciones de consulta. La ventaja de esta estructura es que es muy eficiente en grafos ralos, i.e., grafos que tienen una baja *arboricidad*.

La estructura h-grafo es conceptualmente simple. La misma consiste de tres campos $(d(v), L(v), H(v))$ por cada vértice v de un grafo G, donde $d(v)$ es el grado de v, $L(v)$ son los vecinos de v con grado menor a $d(v)$ y $H(v)$ son los vecinos de v con grado al menos $d(v)$. Asimismo, la estructura mantiene una serie de punteros a fin de que las operaciones puedan resolverse eficientemente.

Los objetivos principales de este trabajo son:

- desarrollar el TAD grafo implementado con la estructura h-grafo,
- verificar que la eficiencia teórica coincide con la de nuestra implementación, y
- testear la eficiencia de la estructura en grafos ralos grandes.

El presente documento esta organizado de la siguiente forma. En la Sección **??** damos blah. En la Seccion **??** hacemos bleh.

3. Preliminares

Recordemos que uno de los objetivos de este trabajo es implementar un TAD grafo. El TAD grafo representa el objeto *grafo* que se estudia en matemática. En esta sección presentamos las definiciones centrales de un grafo que impactan en el desarrollo de nuestro TAD.

Un *grafo* es un par (V, E) donde V ...(ver Figura 1 en donde se ejemplifican muchas de las definiciones). Para cada $vw \in E$, decimos que v y w son *vecinos* o *adjacentes*. El grado de v es la cantidad de vecinos que tiene. El k-vecindario de v es el conjunto de todos los vecinos de v que tienen grado k, para algun k dado. Al conjunto de todos los vecinos de v lo llamamos, simplemente, su *vecindario*.

Camino: es cualquier subconjunto de aristas contiguas, es decir, donde se cumpla que para cada par de aristas, el último vértice de la primer arista es el mismo que el primer vértice de la segunda arista; salvando el caso de la primera y de la última arista que pueden cumplir o no este requisito. Circuito: es un subconjunto de las aristas del grafo que forman un camino donde el primer vertice es el último en ser accedido. Bosque: es aquel grafo que no tiene circuitos. Arboricidad: es la mínima cantidad de bosques con aristas disjuntas que se pueden formar dentro del grafo.

4. El Tipo Grafo

El propósito de esta sección es presentar la interfaz abstracta del objeto grafo que implementaremos. Este objeto es dinámico en el sentido .... Asimismo, ofrece distintas operaciones de consulta, que estan pensadas para poder recorrer eficientemente el vecindario de las distintas aristas del grafo. Estas operaciones son muy utiles, por ejemplo, cuando uno quiere analizar la estructura local del grafo. Es decir, cómo se conectan los vecinos de un vertice v dado.

Para describir la interfaz del TAD grafo, vamos a detallar cada una de las operaciones junto con sus propositos y sus requerimientos. Esta descripción será de alto nivel tal y como se desarrolla en []. La idea de esta descripcion es fijar los conceptos necesarios para la posterior descripcion de su implementación en el lenguaje C++. Asimismo, mostraremos un ejemplo de su uso potencial y describiremos la complejidad temporal esperada. Obviamente, esta complejidad está en función de la implementación que describimos en la Seccion 5.

Aclarar que el tipo grafo es parametrico, es decir, que almacena objetos de un tipo `elem`

4.1. `create_graph()`

Crea un objeto `G` que representa un grafo G de elementos `elem`. Retorna un puntero `G` al nuevo grafo creado. Este objeto guarda cierta información correspondiente a los vertices y sus vecinos. El Pseudocódigo 1 muestra cómo crear un grafo G vacío.

Propósito: crea un `G` sin vertices ni aristas.

Retorna: un puntero `G` al grafo G.

Complejidad temporal:

```
1 G := create_graph()
```

Pseudocódigo 1: Ejemplo de uso de create_graph

4.2. `insert_vertex(G, info)`

Modifica un objeto `G` que representa un grafo G, a fin de representar el grafo H que se obtiene de insertar un nuevo vértice v en G. Retorna un puntero `v` al nuevo vértice v agregado. Este puntero `v` debe usarse para interactuar con G a fin de modificar sus propiedades. Notar que el vértice v es un objeto conocido sólo por G y no es compartido con otros grafos. A este vertice se le puede asociar cierta información que es la reflejada por el parametro `info`. Blah.

El Pseudocódigo 3 muestra cómo usar `insert_vertex`.

Parámetros: `G` representa un grafo G e `info` es cualquier objeto.

Propósito: modifica `G` para representar un grafo $G + v$.

Retorna: un puntero `v` al vértice v.

Complejidad temporal:

4.3. `remove_vertex(G, v)`

Modifica un objeto `G` que representa un grafo G, a fin de representar el grafo H que se obtiene de remover el vértice existente `v` que representa al vertice v en G.

Parámetros: `G` representa un grafo G y `v` representa un vertice existente v de G.

Propósito: modifica `G` para representar un grafo $G - v$.

Complejidad temporal:

```
1 G := create_graph()
2 v := insert_vertex(G, 'a')
3 w := insert_vertex(G, 'a')
4     for v in vertex_iterator(G):
5         print get_info(v)
```

Pseudocódigo 2: Ejemplo de uso de insert_vertex. Crea un grafo G con dos vértices, ambos con la letra "a" como información. Luego, el ciclo imprime "aa". Ver Sección **??** para más información de `vertex_iterator`.

Ejemplo de uso El siguiente código crea un grafo G con dos vértices, el primero `v` representa al vértice v con la letra 'a' y el segundo `w` representa al vértice w con la letra 'b' como infomación. Luego, remueve `v` del grafo y el ciclo imprime "b" como resultado. Ver Sección **??** para más información de `vertex_iterator`.

```
1 G := create_graph()
2 v := insert_vertex(G, 'a')
3 w := insert_vertex(G, 'b')
4 remove_vertex(G, v)
5 for vt in vertices(G):
6   print get_info(vt)
```

4.4. `add_edge(G, v, w)`

Modifica un objeto `G` que representa un grafo G, a fin de representar el grafo H que se obtiene de agregar la arista vw conformada por `v` y `w` que representan los vertices v y w en G.

Parámetros: `G` representa un grafo G, `v` y `w` representan los vertices existentes v y w de G.

Propósito: modifica `G` para representar un grafo $G + vw$.

Complejidad temporal:

Ejemplo de uso El siguiente código crea un grafo G con dos vértices, el primero `v` representa al vértice v con el número 1 y el segundo `w` representa al vértice w con el número 2 como infomación. Luego, crea la arista vw entre estos dos vertices dentro del grafo G. El ciclo imprime cada uno de los valores de los vecinos de v, en este caso como w es su único vecino imprime "2". Ver Sección **??** para más información de `neghbor_iterator`.

```
1 G := create_graph()
2 v := insert_vertex(G, 1)
3 w := insert_vertex(G, 2)
4 add_edge(G, v, w)
5 for n in neighbors(v):
6   print get_info(n)
```

4.5. `remove_edge(G, v, w)`

Modifica un objeto `G` que representa un grafo G, a fin de representar el grafo H que se obtiene de remover la arista vw conformada por `v` y `w` que representan los vertices v y w en G.

Parámetros: `G` representa un grafo G, `v` y `w` representan vertices adyacentes v y w de G, respectivamente.

Propósito: modifica `G` para representar un grafo $G - \{vw\}$.

Complejidad temporal:

Ejemplo de uso El siguiente código crea un grafo G con dos vértices, el primero `v` representa al vértice v con el número 1, el segundo `w` representa al vértice w con el número 2 y el tercero `x` representa al vértice x con el número 3 como infomación. Luego, crea las aristas vw y vx entre estos tres vertices dentro del grafo G y consecuentemente se elimina la arista vw. El ciclo imprime cada uno de los valores de los vecinos de v, en este caso como x termina siendo su único vecino imprime "3". Ver Sección **??** para más información de `neghbor_iterator`.

```
1 G := create_graph()
2 v := insert_vertex(G, 1)
3 w := insert_vertex(G, 2)
4 x := insert_vertex(G, 3)
5 add_edge(G, v, w)
6 add_edge(G, v, x)
7 remove_edge(G, v, w)
8     for n in neighbors(v):
9         print get_info(n)
```

Pseudocódigo 3: Ejemplo de uso de `remove_edge`. En el código se crea un grafo G con vértices v, w y x, siendo v adyacente tanto a w como a x. Luego se utiliza `remove_edge` para eliminar la arista vw, con lo cual v queda adyacente únicamente a x. El ciclo final, pues, imprime el valor 3 asociado a x. Ver Sección **??** para más información de `neghbor_iterator`.

4.6. `add_vertex(G, info, N)`

Modifica un objeto `G` que representa un grafo G, a fin de representar el grafo H que se obtiene de agregar un nuevo vertice v cuya información está reflejada en el parámetro `info` y una lista de punteros `N` que representa al vecindario N que será asociado a v

Parámetros: `G` representa un grafo G, `info` es cualquier objeto y `N` representa el vecindario N.

Propósito: modifica `G` para representar un grafo $G + vn$ donde n es cada elemento de N.

Retorna: un puntero `v` al vértice v.

Complejidad temporal:

Ejemplo de uso El siguiente código crea un grafo G con tres vertices, teniendo el tercer vertice, llamado v, a los otros dos como vecinos. Luego, el ciclo imprime "12".

```
1   G := create_graph()
2   x := insertVertex(G, 1);
3   y := insertVertex(G, 2);
4   v := add_vertex(G, 3, {x, y})
5   for w in neighbors(v):
6       print get_info(w)
```

4.7. `vertices(G)`

4.8. `neighbors(G, v)`

4.9. `H_neighbors(G, v)`

4.10. `degree(G, v)`

5. La estructura h-grafo

En esta seccion describimos la estructura h-grafo que usamos para implementar el TAD grafo descripto en la Seccion 4. Esta estructura es particularmente apropiada para grafos ralos, i.e., grafos cuya arboricidad es baja. Esto se ve reflejado en que las complejidades temporales esperadas, dado que las mismas dependen de la cantidad de aristas y la arboricidad del grafo. La descripción la hacemos a alto nivel tal como en el paper. La idea es.

Conceptualmente, la estructura mantiene un objeto `v` para cada vértice v del grafo G. Este objeto *representa* a v y mantiene la tripla $(d(v), N(v), H(v))$. Recordemos que:

- $d(v)$ se refiere al grado del vértice
- $N(v)$ representa a los vecinos de v que tienen menor grado que v
- $H(v)$ representa a los vecinos de v que tienen al menos el grado de v.

Obviamente $d(v)$ se representa usando un número. Para representar $N(v)$, el h-grafo mantiene (ver Figura) una lista con objetos que representan vecinos que tienen igual grado entre sí. Cada uno de estos vecinos, llamémoslo w, guarda un puntero a la lista $H(w)$ y otro puntero al lugar exacto dónde esta v en $H(w)$.

Vale remarcar que los vértices son privados del grafo, en el sentido de que no se pueden manipular directamente. En su lugar, el usuario modifica v a través de un *puntero* que obtiene al insertar el vértice (ver). Remarcamos que este puntero es un objeto inteligente en el sentido que no muestra la implementacion al usuario.

En el resto de esta sección describimos cómo implementar, usando la estructura h-grafo, cada una de las operaciones que el TAD Grafo soporta (ver Seccion **??**). Para que quede claro que estamos usando la estructura h-grafo, vamos a escribir cada operacion `oper(G, ···)` usando la notación punto, i.e., `G.oper(···)` a fin de remarcar que tenemos acceso a la estructura de `G`.

En esta sección se describirá el algoritmo para insertar aristas y vértices y el algoritmo para removerlos. Cómo se dijo anteriormente, la estructura del h-graph consiste de tres elementos, $d(v)$ que es el grado del vértice, $N(v)$ es una lista de sublistas que representa al vecindario de v, cada sublista no vacía contiene a los vecinos de determinado grado. Las sublistas están ordenadas. $H(v)$ es una la lista que contiene a los vecinos que tienen al menos el grado de v.

h-graph mantiene un puntero directo a los objetos que representan a los vértices, aristas y listas.

5.1. `G.insert_vertex(N, info)`

El proceso para insertar un nuevo vertice v en G se divide en dos partes. Primero, la inserción de un nuevo objeto v que representa un vértice aislado. Segundo, la inserción de una arista entre v y cada uno de los vértices representados por N. Claramente, el grafo obtenido representa a $G + v$.

Pseudocódigo

5.2. `G.insert_edge(v, w)`

El algoritmo para insertar la nueva arista vw al grafo G, siendo v y w vértices de G, tiene tres fases bien delimitadas que describimos a continuación.

```
1  G.insert_vertex(N, info):
2      v := G.append((0, [], []))
3      for w in N:
4          G.insert_edge(v, w)
```

Primer fase. La primer fase consiste en actualizar los vecindarios de v y w. Recordemos que $H(v)$ contiene aquellos vecinos de v que tienen grado al menos $d(v)$. Como $d(v)$ se incrementa en 1 cuando agregamos la nueva arista vw, estamos obligados a sacar de $H(v)$ aquellos vértices cuyo grado es exactamente $d(v)$. Claramente, cada vértice $z \in H(v)$ que tenemos que sacar tiene grado $d(v)$. Por lo tanto, tenemos que colocar todos estos vertices en un nueva lista que represente a $N(v, d(v))$. Entonces, para actualizar el vecindario de v, agregamos a $\mathcal{N}(v)$ una nueva lista $N(v, d(v))$, y movemos cada $z \in H(v)$ de grado $d(v)$ hacia $N(v, d(v))$. Para mover el vértice z, tenemos que actualizar los punteros involucrados. Notemos que al mover el objeto `z` que representa a z de $H(v)$ a $N(v, d(v))$, tanto el self pointer de `z` (que indica dónde se encuentra representado v en $N(z)$) como su list pointer (que indica la lista que contiene a v en $N(z)$) son correcto. En cambio, tenemos que actualizar el list pointer y el self pointer del objeto `v` que representa a v en $N(z)$. Análogamente, para actualizar el vecindario de w, agregamos una lista $N(w, d(w))$ a $\mathcal{N}(w)$ y movemos cada $z \in H(w)$ de grado $d(w)$ hacia $N(w, d(w))$. Vale remarcar que $N(v, d(v))$ (resp. $N(w, d(w))$) se crea solo si existe algún vecino de v (resp. w) con grado $d(v)$ (resp. $d(w)$).

Segunda fase. La segunda fase consiste en actualizar el vecindario $N(z)$ de cada vértice z que sea vecino de v o w. Notemos que si $v \in H(z)$ antes de la inserción, entonces $v \in H(z)$ luego de inserción, y por lo tanto el objeto que representa a v no debe actualizarse. En cambio, si $v \notin H(z)$, entonces $d(v) < d(z)$ en G y $v \in N(z, d(v)) \in \mathcal{N}$. Por lo tanto, tenemos que mover v desde $N(z, d(v))$ hacia $N(z, d(v) + 1)$ para todo $z \in H(v)$ (por simplicidad, consideramos que $N(z, d(z)) = H(z)$). Para esto, recorremos cada $z \in H(v)$ que tenga grado a mayor a $d(v)$ y movemos el objeto `v` que representa a v en $N(z)$. Notemos que dicho objeto `v` es el apuntado por el self pointer del objeto `z` que representa a z en $H(v)$. Por lo tanto, podemos acceder eficientemente a `v` para moverlo, actualizando los self y list pointers de `z`. Una vez finalizado el procesamiento de $H(v)$, tenemos que proceder de manera análoga con $H(w)$.

Tercer fase. Por último, tenemos que agregar físicamente a vw en G. Para ello, creamos un objeto `w` que represente a v en $N(v)$ y otro `v` que represente a v en $N(w)$, y aumentamos el grado de v y w en 1.

A continuación presentamos el pseudocódigo del algoritmo con sus tres fases correspondientes.

```
1  G.insert_edge(v,w):
2    //Fase 1
3    G.update_neighborhood(v)
4    G.update_neighborhood(w)
5    //Fase 2
6    G.update_neighbors(v)
7    G.update_neighbors(w)
8    //Fase 3 (insercion fisica)
9    Insertar un nuevo objeto v al final de N(w, d(v))
10   Insertar un nuevo objeto w al final de N(v, d(w))
11   poner v->self_pointer := ultimo de N(v, d(w))
12   poner v->list_pointer := N(v, d(w))
13   poner w->self_pointer := ultimo de N(w, d(v))
14   poner w->list_pointer := N(w, d(v))
```

```
1  G.update_neighborhood(v):
2      crear una nueva lista N(v,d(v))
3      para cada z in H(v) si d(v) = d(z):
4          mover z de H(v) a N(v, d(v))
5          //Obs: z->self_pointer es el objeto que representa a v en N(z)
6          poner z->self_pointer->list_pointer := N(v,d(v))
7      si N(v,d(v)) tiene elementos, entonces insertar N(v,d(v)) al final de N(v
```

```
1  G.update_neighbors(v):
2      para cada z in H(v) si d(v) < d(z):
3          //Obs: z->self_pointer es el objeto que representa a v en N(z),
4          sea v' := z->self_pointer
5          //     y z->list_pointer es el objeto que representa N(z, d(v)) en N(
6          sea N(z,d(v)) := z->list_pointer
7          Si la lista que sigue a N(z, d(v)) en N(z) no corresponde a vertices
8              Agregar una nueva lista N(z, d(v)+1) a continuacion de N(z,d(v))
9          Mover v' de N(z, d(v)) a N(z, d(v)+1).
10         Si N(z, d(v)) queda vacia, entonces remover N(z,d(v)) de N(z)
```

Ejemplo de una ejecucion.

5.3. `G.remove_edge(v,w)`

El algoritmo para remover la arista vw del grafo G consiste en deshacer, en el orden inverso, las tres fases que usamos para su inserción (ver Seccion 5.2)

Deshacer tercer fase. Recordemos que la tercer fase del proceso de inserción de vw consiste en agregar físicamente la arista vw. Luego, para deshacer la tercer fase tenemos que remover físicamente a vw de G. Supongamos, invirtiendo v con w de ser necesario, que $d(v) \leq d(w)$, i.e., $w \in H(v)$. El primer paso es recorrer $H(v)$ hasta localizar al objeto `w` que representa a w en $H(v)$. En el segundo paso, usamos el list y self pointers de `w`, para acceder y borrar la encarnación `v` de v en la lista `N(d(v), w)`. Por último, eliminamos a `w` de $H(v)$ y decrementamos en 1 el grado de v y w.

Deshacer segunda fase. Al igual que para la inserción, la segunda fase consiste en actualizar el vecindario $N(z)$ de cada vértice z que sea vecino de v o w. Obviamente, en este caso, la actualización refleja la remoción de vw. Notemos que si $d(v) \geq z$ luego de deshacer la tercer fase, entonces $v \in H(z)$ tanto antes como luego del borrado. En consecuencia, no hace falta actualizar la encarnación de v en $N(z)$. En cambio, si $d(v) < d(z)$ luego de deshacer la tercer fase, entonces $z \in H(v)$, con lo cual tenemos que mover la encarnación `v` de v desde $N(z, d(v)+1)$ a $N(z, d(v))$. (Notar que, en caso en que $v \in H(z)$, `N(z, d(v)+1)` representa a $H(z)$.) Para ello, recorremos cada encarnación `z` de $z \in H(v)$ y, haciendo uso de sus list y self pointer, accedemos a la encarnación `v` de v en $N(z, d(v)+1)$ y la movemos físicamente a $N(z, d(v))$. Por último, actualizamos el list pointer de `z` para que apunte a $N(z, d(v))$ y el self pointer de `z` para que apunte a la encarnación de v en $N(z, d(v))$.

Deshacer primer fase. En esta fase actualizamos el vecindario de v y w, a fin de reflejar la remoción de vw. Recordemos que en la primer fase de inserción de vw movimos cada $z \in H(v)$ de grado $d(v)$ a una lista $N(v, d(v))$, debido a que $d(v)$ aumentaba en 1. Para deshacer este proceso, movemos cada vértice $z \in N(v, d(v))$ hacia $H(v)$. Recordemos que en el h-graph cada lista $N(d(v))$ es una lista no vacía, razón por la cual $N(v, d(v))$ es finalmente eliminada.

a continuación el pseudocódigo.

```
1  G.remove_edge(v,w) {
2      if grado(v) > grado(w): swap(v,w)
3      //Deshacer Fase 3
4      w := buscar w en H(v)
5      borrar a w->self_pointer de w->list_pointer
6      borrar w de H(v)
7      grado(v) := grado(v) - 1.
8      grado(w) := grado(w) - 1.
9      //Deshacer Fase 2
10     G.update_neighbors_delete(v)
11     G.update_neighbors_delete(w)
12     //Deshacer Fase 2
13     G.update_neighborhood_delete(v)
14     G.update_neighborhood_delete(w)
15 }
16
17 G.update_neighbors_delete(v) {
18     para cada z en H(v) {
19         Mover z->self_pointer de z->list_pointer al inicio de prev(z->list_po
20         Borrar z->list_pointer si queda vacia
21         z->list_pointer := prev(z->list_pointer)
22         z->self_pointer := primero de z->list_pointer
23     }
24 }
25
26 G.update_neighborhood_delete(v){
27     Si los vertices en prev(H(v)) no tienen grado d(v), retornar
28     para cada z en prev(H(v)) {
29         mover z al inicio de H(v)
30         //z->self_pointer es el objeto que representa a v en N(z).
31         z->self_pointer->list_pointer := H(v)
32         z->self_pointer->self_pointer := primero de H(v)
33     }
34     Borrar prev(H(v))
35 }
```

Pseudocódigo 4: Implementación de `remove_edge`, deshaciendo las tres fases de la `insert_edge`.

6. Implementación en C ++

En esta sección vamos a describir todos los detalles de la implementación de la estructura h-grafo en el lenguaje C++. La Sección **??** describe la interfaz pública, mientras que la Sección **??** describe la estructura interna y sus algoritmos.

Tal como vimos en la Sección **??**, la estructura h-grafo guarda, para cada vertice v, una lista $N(v, d)$ que contiene todos los vecinos de v de grado d. Como veremos en la Sección **??**, la idea es utilizar el tipo `std::list` de C++ para almacenar los vertices en $N(v, d)$. Pero, cada objeto `z` de $N(v, d)$ debe contener un puntero al objeto `v` que representa a v en $N(z, d(v))$. Obviamente, como los nodos de `std::list` son privados, no hay forma de obtener un puntero crudo a dicho nodo físico. Para resolver este problema, tenemos que usar un objeto que nos permita realizar las operaciones que requerimos de la lista, sin romper su estructura. En C++, los objetos que se comportan como estos *punteros restringidos* son los iteradores.

Que es un iterador de lista.

En la Sección **??** también mencionamos que `G.insert_vertex()` retorna un "puntero" al objeto `v`

que representa al nuevo vértice insertado. Estos punteros no permiten acceder indiscriminadamente a la estructura interna del grafo G. Más aún, el usuario de dicho puntero no debería conocer la estructura interna de G. Para modificar algún aspecto de v, el usuario debe invocar un método de G brindando `v` como parámetro. Por otra parte, la estructura h-grafo ofrece distintos iteradores para recorrer los vértices y las aristas de G. Siguiendo la filosofía de la biblioteca estándar, los punteros que se usan para manipular a G se corresponden con estos iteradores.

Los iteradores sirven para recorrer un contenedor, ocultando la estructura interna de éste. Para ello se valen de una interfaz común a todos los objetos iterables. Lo que permite usarlos como punteros seguros a la estructura interna sin exponerla. En c++ existen distintos tipos de iteradores

output e *input* son los más básicos, ellos pueden realizar operaciones de entrada o salida de manera secuencial. Soportan las operaciones ==, !=, *, ->

Forward esta diseñado para recorrer contenedores cuyos valores puedan ser añadidos y recuperados. Mantienen la restricción de que únicamente se pueden mover hacia delante

El *bidireccional* tiene las funcionalidades del *Forward* pero además pueden avanzar al elemento siguiente o retroceder al elemento anterior.

Random access son iteradores que pueden avanzar o retroceder más de una posición de una vez.

Todos los objetos iterables soportan el método **begin()** que devuelve un iterador al primer elemento a iterar. El método **end()** retorna un iterador al final de la estructura.

Para obtener el elemento al que apunta el iterador se usa el método de desreferencia *, tal como se hace con los punteros.

Ejemplo de uso de iteradores sobre una lista.

```
1  void show_list( const list<string) & sList){
2          list<string>:: const_iterator pos;
3
4      //obtenemos un iterador al primer elemento
5      pos = sList.begin();
6
7      while( pos != sList.end()){ // mientras pos sea distinto al final de la
           lista
8          cout<< *pos << endl;
9          pos++;  //avanza el iterador
10     }
11 }
```

6.1. Interfaz de C++

Existe una clase **Graph** que es la que posee los métodos visibles para el usuario. **Graph** es un template. Los templates permiten crear estructuras de datos que no dependen del tipo de dato que reciban. De esta manera conseguimos una generalización del grafo.

Respetando la naturaleza del h-graph el usuario no tiene acceso a los vértices. Por eso se diseñaron varios iteradores para recorrer la estructura. Recordemos que los iteradores sirven para recorrer una estructura sin tener en cuenta la forma en que fue implentada y a su vez se pueden utilizar como punteros a la estructura pero sin exponerla.

En la implementación del h-graph se provee al usuario tres tipos de iteradores:

1. **const_vertex_iterator** : es un iterador bidireccional que apunta a un vértice. Este iterador se obtiene al agregar un nuevo vértice al grafo. Este iterador le va servir al usuario para manipular el grafo, ya sea para agregar o borrar una arista, borrar un vértice o recorrer todos los vértices del grafo. Sin embargo, el iterador no permite al usuario modificar directamente información del vértice, como por ejemplo el grado.

   Para recorrer todos los vértices del grafo, la clase Graph posee los métodos *begin()* y *end()* que retornan el iterador const_vertex_iterator.

Ejemplo de uso del **const_vertex_iterator**

```
1  tip::Graph<int> G;
2  const_vertex_iterator v1 = G.insertVertex(1);
3  const_vertex_iterator v2 = G.insertVertex(2);
4
5  G.add_edge(v1,v2);
6
7  void print_vertices( Graph& G) {
8      for(auto it = G.begin(); it != G.end(); ++it) {
9          cout << *it << endl;
10     }
11 }
```

2. **deg_iterator**: es un iterador bidireccional que permite iterar los vecinos de determinado grado de un vértice. Este iterador es propio de la clase *Vertex*. El usuario para obtener este iterador, lo hace a través de la clase *Graph* con los métodos *H_ begin()*, *H_ end()*, lo cuales le permiten iterar el *HighNeighborhood* del vértice.

   Ejemplo de uso del **deg_iterator**

```
1  tip::Graph<int> G;
2  const_vertex_iterator v1 = G.insertVertex(1);
3  const_vertex_iterator v2 = G.insertVertex(2);
4
5  for(auto it = G.H_begin(v1); it != G.H_end(v1); ++it) {
6                  cout << *it << ',';
7              }
```

3. **neighbor_iterator**: es un iterador bidireccional que permite iterar todos los vecinos de un vértice. Es un iterador privado de la clase *Vertex*. El usuario tiene acceso a él a través de la clase *Graph* con los métodos *N_ begin(const_ vertex_ iterator v))* y *N_ end(const_ vertex_ iterator v))*

Ejemplo de uso del **deg_iterator**

```
1              tip::Graph<int> G;
2              const_vertex_iterator v1 = G.insertVertex(1);
3              const_vertex_iterator v2 = G.insertVertex(2);
4
5              for(auto it = G.N_begin(v1); it != G.N_end(v1); ++it) {
6                              cout << *it << ',';
7                          }
```

6.2. Estructura en C++

En esta sección describiremos la estructura interna de la estructura h-grafo, tal cual está implementada en C++. El código de la estructura se encuentra organizada en ¿4? archivos fuente:

contiene la estructura general del h-grafo.

contiene la descripción de lo que es un vértice.

A continuación describimos las clases principales del h-grafo, indicando el archivo en el que esta definida.

6.2.1. Graph (Graph.h)

Recordemos que el tipo Grafo que implementamos es un tipo paramétrico que permite asociar información de algún tipo a cada vértice. La representación de un grafo, usando la clase `Graph<Elem >` es sencilla en términos conceptuales, ya que simplemente una lista de `Vertex` (ver Seccion **??**). Como veremos en la Sección **??**, la clase `Vertex` mantiene toda la información de los vecinos de un vértice. En consecuencia, `Vertex` es una componente central de la estructura. Por este motivo, `Vertex` necesita tener acceso a algunas partes privadas del grafo, razón por la cual, `Graph<Elem>` es `friend` de `Vertex`. En particular, `Vertex` necesita saber de qué tipo es `Elem` para saber cómo almacenar la información de un vértice.

Un inconveniente central a la hora de definir el tipo `Graph<Elem>`, es que el mismo necesita conocer qué es un `Vertex`, mientras que, por lo mencionado anteriormente, `Vertex` necesita conocer el tipo `Elem`. Con lo cual, hay un inconveniente serio a la hora de decidir qué clase se crea primero. Una solución simple a este problema es usar el *Curiously recurring template pattern* [], que consiste escribir el tipo interno `Vertex` como un tipo paramétrico `Vertex<Graph>` que depende `Graph<Elem>` que lo utilice. Luego, `Graph<Elem>` puede utilizar una instancia de `Vertex<Graph>`. Más allá del patrón de implementación, la estructura no deja de ser una lista de vertices.

```
1  template<class Elem> class Graph
2  {
3      using elem_type = Elem;
4  private:
5      friend class impl::Vertex<Graph>;
6
7      //CRTP: Curiously recurring template pattern
8      using Vertex = impl::Vertex<Graph>;
9      using Vertices = std::list<Vertex>;
10     Vertices vertices;
11 };
```

Pseudocódigo 5: Estructura del tipo Grafo en C++.

6.2.2. Vertex (Vertex.h)

De acuerdo a lo visto en la Sección **??**, `Vertex` es un tipo paramétrico que representa una instancia de un vértice v del h-grafo. El parámetro esperado para el template `Vertex` es un clase de `Graph`. Vale remarcar que el tipo `Vertex` se usa únicamente dentro de la clase `Graph`, quien no exhibe su interfaz interna. En otras palabras, `Vertex` es invisible al usuario de `Graph`, más allá de que todos sus miembros sean públicos.

La clase `Vertex` posee la información asociada con el vertice (`elem`), el grado del vértice (`degree`) y su vecindario (`neighborhood`). Como vimos en la Sección **??**, el vecindario en el h-grafo se organiza como una lista de listas, cuyos elementos representan las listas no vacías de $N(v,0),\ldots,N(v,d(v)-1)$ y $H(v)$. Cada lista $N(v,i)$ guarda los vecinos de grado i y $H(v)$ guarda los vecinos de grado al menos $d(v)$. Para la representación en C++, por comodidad, almacenamos $H(v)$ como la última lista `neighborhood`.

```
1  template<class Graph> struct Vertex
2  {
3      //Tipo del elemento que se guarda en los vertice;
4      //notar que es el tipo que el usuario indica en la clase Graph
5      using elem_type = typename Graph::elem_type;
6
7      elem_type elem;  //elemento
8      size_t degree;  //grado
9      Neighborhood neighborhood;  //vecindario
10 }
```

Pseudocódigo 6: Estructura del tipo Grafo en C++.