Departamento de Ciencia y Tecnología
Tecnicatura Universitaria en Programación Informática

Implementación de la estructura h-grafo

Griselda Cardozo **Alejandro Merlo**

Director: Dr. Francisco Soulignac

Bernal, 10 de marzo de 2016

Implementación de la estructura h-grafo

Poner aca un resumen

Palabras clave: *h-grafo, grafos ralos, arboricidad, implementación.*

Para quien sea.
Alguna frase si queres
Griselda Cardozo.

Para quien sea.
Alguna frase si queres
Alejandro Merlo.

Agradecimientos

A quien corresponda, o se elimina

Fecha del dia del agradecimiento.

Índice general

1. Introducción **1**
1.1. Preliminares 3

2. El tipo abstracto Grafo **5**
2.1. `create_graph()` 5
2.1.1. `insert_vertex(G, info)` 6
2.1.2. `remove_vertex(G, v)` 6
2.1.3. `add_edge(G, v, w)` 7
2.1.4. `remove_edge(G, v, w)` 8
2.1.5. `add_vertex(G, info, N)` 8
2.1.6. `vertex_iterator(G)` 9
2.1.7. `neighbor_iterator(v)` 10
2.1.8. `H_neighbor_iterator(v)` 10
2.1.9. `degree(v)` 11

3. La estructura h-grafo **13**
3.0.1. `G.insert_vertex(N, info)` 14
3.0.2. `G.remove_vertex(v)` 14
3.0.3. `G.insert_edge(v, w)` 14
3.0.4. `G.remove_edge(v,w)` 17

4. Implementación en C ++ **19**
4.0.1. Interfaz de C++ 19
4.0.2. Interfaz del tipo Graph 23
4.0.3. Estructura en C++ 24

5. Resultados de la experimentación **31**

1 Introducción

Los grafos son estructuras abstractas que sirven para modelar distintas tipos de relaciones matemáticas. Formalmente, un *grafo* es un par $G = (V, E)$ donde V es un conjunto (finito) de *vértices* y E es un conjunto formado por pares de vértices. En términos matemáticos, los vértices son objetos cuya única propiedad es la de ser distinguibles; en la práctica, los vértices representan objetos de algún dominio de aplicación. Los pares (v, w) que pertenecen a E son llamados *aristas*, y su propósito es reflejan alguna relación entre v y w. Los grafos se usan para modelar relaciones de problemas reales en forma abstracta. Por ejemplo, una red social se modela con un grafo que tiene un vértice por cada persona y una arista entre dos vértices para indicar que las personas son amigas. La ventaja de utilizar grafos en lugar de los objetos que ellos modelan, es que tenemos acceso a una terminología común, de manera tal que podemos traducir los avances de una aplicación a otra. De esta forma, podemos aprovechar la teoría de grafos para resolver problemas de areas tan diversas como las matemáticas discretas, la infórmatica, las telecomunicaciones, la biología, la sociología, la filosofía, etc. [**?**]. La *teoría de grafos* es la disciplina que se encarga del estudio de grafos, mientras que la teoría *algorítmica* de grafos se encarga de los problemas computacionales asociados.

El objetivo del presente trabajo es implementar la estructura de datos h-grafo descrita por Lin et al. [**?**]. Como objetivos particulares de nuestro desarrollo, esperamos:

- Verificar la correctitud de la misma. Si bien el artículo describe la estructura con suficiente detalle, la misma no está implementada en un lenguaje de programación real. En consecuencia, podrían haber detalles de implementación no detectados a la hora de describir la estructura.

- Analizar la dificultad de implementar la estructura en un lenguaje eficiente de escala industrial. Como es común en los artículos de investigación teóricos, los investigadores suponen la existencia de ciertas tipos de datos (como las listas) con una interfaz apropiada a sus objetivos. Sin embargo, las implementaciones reales de las mismas muchas veces distan de dicha interfaz ideal, lo que obliga a re-implementar estructuras básicas (e.g., re-implementar listas para tener acceso a la representación interna) o a resolver las discrepancias usando las estructuras ya existentes (e.g., utilizar *iteradores* como si fueran punteros a los nodos de la representación). En este trabajo tomamos la segunda opción.

Como objetivo futuro, que excede el alcance del trabajo realizado, sería deseable también analizar la eficiencia de la estructura de datos, tanto de forma aislada como en comparación con otras estructuras.

La estructura h-grafo se usa para implementar el *tipo abstracto de datos* (TAD) Grafo, que es una representación computacional del correspondiente objeto matemático. De acuerdo a la interfaz que uno elija para operar con un grafo, es posible definir distintos tipos de TADs, con distintas características, que sean más o menos útiles para las distintas aplicaciones. En particular, la estructura h-grafo fue concebida para aplicaciones que requieren grafos dinámicos. Por *dinámico* nos referimos a que la interfaz debe proveer operaciones eficientes para insertar y borrar vértices y aristas, a la vez que provee operaciones eficientes de consulta. Es por este motivo que el TAD Grafo que nosotros presentamos es consistente con el dinamismo requerido por las aplicaciones.

La estructura h-grafo es una estructura general que se puede usar para representar a cualquier grafo. Sin embargo, la estructura es particularmente eficiente en grafos *ralos*, i.e., grafos con pocas aristas. Existen distintas definiciones de dan cuenta de qué es un grafo ralo. La definición más básica (y general) establece que un grafo es ralo cuando tiene $O(n)$ aristas, siendo n la cantidad de vértices del grafo. Si bien esta noción es útil, el resultado es que cualquier grafo no-ralo se puede convertir en un grafo ralo agregando suficiente vértices de grado bajo. En muchas aplicaciones, estos vértices se pueden preprocesar eficientemente, lo que nos deja con un grafo no-ralo. En particular, la estructura h-grafo requiere, para ser eficiente, que cualquier parte del grafo sea rala. En otras palabras, la estructura es eficiente cuando todo subgrafo con k vértices tiene $O(k)$ aristas. Esta es la noción que nosotros vamos a considerar cuando decimos que un grafo es ralo. En particular, Nash-Williams [**?**] observo que un grafo es ralo baja, razón por la cual el artículo de Lin et al. [**?**] utiliza el parámetro de la arboricidad para analizar la eficiencia de los algoritmos. Vale remarca que la arboricidad de un grafo con m aristas es a lo sumo $O(\sqrt{m})$, y que los grafos planares tienen arboricidad 3.

Los objetivos específicos de este este trabajo son:

1. describir una interfaz apropiada para el TAD Grafo, siguiendo los usos y costumbres de nuestro lenguaje de implementación (C++),
2. desarrollar el TAD grafo implementado con la estructura h-grafo, aprovechando la biblioteca estándar de nuestro lenguaje,
3. verificar que la eficiencia teórica coincide con la de nuestra implementación para algunos grafos, y
4. probar la eficiencia de la estructura en algunos grafos ralos grandes.

Como objetivo a futuro quedará el análisis de la estructura en aplicaciones reales de gran escala.

El presente documento esta organizado de la siguiente forma. En el Capítulo 2 describimos el TAD Grafo, desde un punto de vista agnóstico con respecto al lenguaje, haciendo un repaso de su interfaz. En el Capítulo 3 explicamos la estructura h-grafo, a la vez que describiremos los diferentes algoritmos que implementan las distintas operaciones del TAD Grafo. Este Capítulo también está presentado en forma independiente al lenguaje. Luego, en el Capítulo 4, nos centraremos en la implementación del TAD usando el lenguaje C++. Este capítulo está dividido en dos secciones. La primera, Sección 4.0.1, da cuenta de la interfaz del h-grafo siguiendo los lineamientos de C++. La segunda, Sección 4.0.3, explica cómo se implementa la estructura en C++ aprovechando la biblioteca estándar. El Capítulo 5 presenta brevemente algunos resultados en los que medimos la eficiencia de la estructura h-graph. Finalmente, el Capítulo **??** presenta un breve resumen de lo logrado junto con las posibilidades de desarrollo a futuro.

1.1. Preliminares

Como mencionamos en la sección anterior, el objetivo central de este trabajo es implementar un TAD Grafo que explote la estructura de datos h-grafo. El TAD Grafo es una representación computacional del objeto abstracto *grafo*, que es el que se estudia en teoría de grafos. En esta sección incluimos definiciones de la teoría de grafos que son necesarias para comprender nuestro trabajo (ver Figura 1.1 en donde se ejemplifican muchas de las definiciones de esta sección).

Un *grafo* es un par $G = (V, E)$ donde V es un conjunto finito de *vértices* y E es un conjunto de pares no ordenados, llamados *aristas*. Vamos a escribir $V(G)$ y $E(G)$ para denotar a V y E, mientras que al par no ordenado formado por v y w lo vamos a escribir como vw. Para cada $vw \in E$, decimos que v y w son *vecinos* o *adyacentes*. El grado $d(v)$ de un vértice v es la cantidad de vecinos que tiene v. El k-vecindario de v es el conjunto $N(v, k)$ de todos los vecinos de v que tienen grado k, para algun $k \in \mathbb{N}$. Obviamente, $N(v, k) = \emptyset$ para todo $k \geq n$. Al conjunto $N(v)$ de todos los vecinos de v lo llamamos, simplemente, su *vecindario*. Notar que $N(v)$ es la unión de todos los k-vecindarios de v, i.e., $N(v) = \bigcup_{i=0}^{n-1} N(v, k)$.

Un grafo H es un *subgrafo* de G cuando $V(H) \subseteq V(G)$ y $E(H) \subseteq E(G)$. Si $V(H) = V(G)$, entonces H es un subgrafo *generador* de G. Un *camino* de G es una secuencia de vértices distintos $v_1, \ldots, v_k$ tal que $v_i v_{i+1} \in E(G)$ para todo $1 \leq i < k$. Un ciclo es una secuencia $v_1, \ldots, v_k, v_1$ tal que $v_1, \ldots, v_k$ es un camino. Los *bosques* son aquellos grafos que no tienen ciclos. La *arboricidad* $\alpha(G)$ de un grafo G es la mínima cantidad de bosques generadores en las que se puede particionar $E(G)$. Los siguientes resultados relacionan la arboricidad de G con su cantidad de aristas.

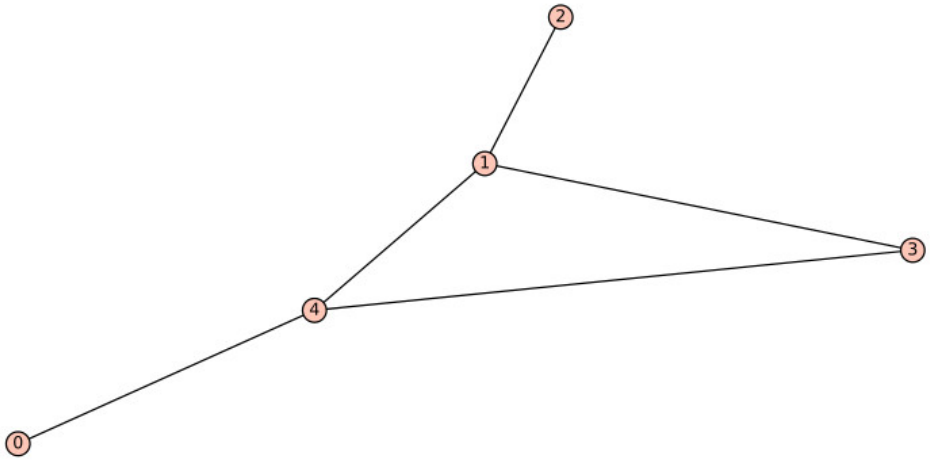

Figura 1.1: Grafo G con $V(G) = \{0, 1, 2, 3, 4\}$ y $E(G) = \{(0, 4), (1, 2), (1, 3), (1, 4), (3, 4)\}$. El vértice 1 tiene grado 3 y su vencindario es $\{2, 3, 4\}$. La secuencia $0, 4, 1, 2$ es un camino, mientras que $4, 1, 3, 4$ es un ciclo. El grafo H con $V(H) = V(G)$ y $E(H) = \{04, 13, 12\}\}$ es un bosque que es subgrafo generador de G, i.e., H es un bosque generador de G. Como $\{H, J\}$ es una partición en bosques de G, con $V(J) = V(G)$ y $E(J) = E(G) \setminus E(H)$, entonces $\alpha(G) \leq 2$. Más aún, $\alpha(G) = 2$ porque G no se puede particionar en un único bosque (ya que no es un bosque).

Teorema 1 ([?]). *Para todo grafo G ocurre que*

$$\alpha(G) = \text{máx}\left\{ \frac{|E(H)|}{|V(H)| - 1} \middle| H \text{ es subgrafo de } G \right\}$$

Teorema 2 ([?]). *Para todo grafo G ocurre que $\alpha(G) \leq 2\sqrt{m}$.*

2 El tipo abstracto Grafo

El propósito de esta sección es presentar la interfaz del tipo abstracto Grafo que implementaremos. Cada instancia de este TAD es *dinámica*, en el sentido que la instancia va mutando de acuerdo la inserción de vértices y aristas. Asimismo, el TAD ofrece distintas operaciones de consulta, que están pensadas para poder recorrer eficientemente el vecindario de las distintas aristas del grafo. Estas operaciones son muy útiles, por ejemplo, cuando uno quiere analizar la estructura "local" del grafo. Es decir, cómo se conectan los vecinos de un vértice v dado entre sí.

Para describir la interfaz del TAD Grafo, vamos a detallar cada una de las operaciones junto con sus propósitos y sus requerimientos (precondiciones). Esta descripción será de alto nivel, tal y como se desarrolla en [**?**]. La idea de esta sección es fijar los conceptos necesarios para la posterior descripción de su implementación en el lenguaje C++. Asimismo, mostraremos ejemplos de uso de las distintas operaciones y describiremos la complejidad temporal esperada. Obviamente, esta complejidad está en función de la implementación que se pospone al Capítulo 3.

Recordemos que si bien los vértices de un grafo son elementos abstractos, los mismos se usan para representar entidades de aplicaciones reales. Es común que estas entidades tengan alguna información propia que es requerida dentro de la aplicación. Es por este motivo que el grafo permite asociar cierta información a cada vértice, lo que convierte al TAD Grafo en un tipo paramétrico. Específicamente, el TAD Grafo almacena objetos de un tipo un tipo genérico `elem`.

2.1. `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 vértices y sus vecinos. El Pseudocódigo 2.1 muestra cómo crear un grafo G vacío.

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

Retorna: un puntero `G` al grafo G.

Complejidad temporal: O(1)

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

Pseudocódigo 2.1: Ejemplo de uso de `create_graph`

2.1.1. `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. A este vértice v se le puede asociar cierta información que es la reflejada por el parámetro `info`. 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 puntero `v` es un objeto conocido sólo por `G` y no es compartido con otras instancias del TAD Grafos. En particular, el uso de `v` con otra instancia que no sea `G` conduce a un comportamiento indefinido.

El Pseudocódigo 2.2 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: O(n)

```
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 2.2: 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 2.1.6 para más información de `vertex_iterator`.

2.1.2. `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 vértice v en G. Como resultado de esta operación, se invalidan aquellos iteradores que están relacionados con v (es decir:

iteradores de vértices que apuntan a v, iteradores de vecindarios de v, iteradores de vecindarios de cualquier vértice $w \in N(v)$ que circunstancialmente este recorriendo a v, e iteradores de vecindarios que circunstancialmente esten recorriendo algún $w \in N(v)$).

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

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

Complejidad temporal: O(n)

```
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 vertex_iterator(G):
6     print get_info(vt)
```

Pseudocódigo 2.3: Ejemplo de uso de remove_vertex. El 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 2.1.6 para más información de `vertex_iterator`.

2.1.3. `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 vértices v y w en G.

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

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

Complejidad temporal:

```
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 neighbor_iterator(v):
6     print get_info(n)
```

Pseudocódigo 2.4: Ejemplo de uso de add_edge. El 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 vértices 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 2.1.7 para más información de `neghbor_iterator`.

2.1.4. `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 vértices v y w en G. Como resultado de esta operación, se invalidan aquellos iteradores de vecindarios que están relacionados con v o w (es decir: iteradores de vecindarios de v o w, e iteradores de vecindarios de otros vértices que circunstancialmente estén recorriendo algún $z \in N(v) \cup N(w)$).

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

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

Complejidad temporal:

2.1.5. `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 vértice 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.

```
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 neighbor_iterator(v):
9     print get_info(n)
```

Pseudocódigo 2.5: 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 2.1.7 para más información de neghbor_iterator.

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: O(1)

```
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 neighbor_iterator(v):
6     print get_info(w)
```

Pseudocódigo 2.6: Ejemplo de uso de add_vertex. El 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".

2.1.6. vertex_iterator(G)

Provee de un puntero que permite desplazarse a través de los distintos vertices del objeto G que representa un grafo G y acceder a ellos, a fin de poder manipularlos de una forma rápida y simple.

Parámetros: `G` representa un grafo G.

Propósito: proveer al usuario de un mecanismo para iterar los vertices de G.

Retorna: un iterador `i` a los vértices de G.

Complejidad temporal:

```
1 G := create_graph()
2 x := insertVertex(G, 1);
3 y := insertVertex(G, 2);
4 for v in vertex_iterator(G):
5     print get_info(v)
```

Pseudocódigo 2.7: Ejemplo de uso de `vertex_iterator`. El código crea un grafo G con el vertice 1 y el vertice 2. Luego, mediante el iterador de G se recorre sus vertices y el ciclo imprime "12".

2.1.7. `neighbor_iterator(v)`

Provee de un puntero que permite desplazarse a través de los vecinos del vértice v representado por el objeto `v`, a fin de acceder y manipular cada vecino de forma rápida y simple. Si bien no está implementado en esta versión, dicho iterador tambien podría usarse para eliminar directamente la arista entre v y w para cada $w \in N(v)$ que se recorre.

Parámetros: `v` representa un vértice v.

Propósito: proveer al usuario de un mecanismo para iterar los vecinos de v.

Retorna: un iterador `i` a los vecinos de menor grado de v.

Complejidad temporal:

2.1.8. `H_neighbor_iterator(v)`

Provee de un puntero que permite desplazarse a través de los distintos vecinos de grado mayor o igual al vértice v representado por el objeto `v`, a fin de poder acceder y manipular cada vecino de forma rápida y simple. Si bien no está implementado en esta versión, dicho iterador tambien podría usarse para eliminar directamente la arista entre v y w para cada $w \in N(v)$ que se recorre.

```
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 neighbor_iterator(v):
6     print get_info(w)
```

Pseudocódigo 2.8: Ejemplo de uso de neighbor_iterator. El código crea un grafo G con el vértice 1, el vértice 2 y el vértice 3. Luego, mediante el iterador de v se recorre sus vecinos de menor grado y el ciclo imprime "12".

Parámetros: `v` representa un vértice v.

Propósito: proveer al usuario de un mecanismo para iterar los vecinos de grado mayor o igual al de v.

Retorna: un iterador `i` a los vecinos de grado mayor o igual al de v.

Complejidad temporal:

```
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 H_neighbor_iterator(x):
6     print get_info(w)
```

Pseudocódigo 2.9: Ejemplo de uso de `H_neighbor_iterator`. El código crea un grafo G con el vértice 1, el vértice 2 y el vértice 3. Luego, mediante el iterador de x se recorre sus vecinos de mayor o igual grado y el ciclo imprime "3".

2.1.9. `degree(v)`

Dado un objeto `v` que representa un vértice v devuelve la cantidad de vecinos de dicho vértice.

Parámetros: `v` representa un vértice v.

Retorna: un número que denota la cantidad de vecinos de v.

Complejidad temporal: O(1)

```
1 G := create_graph()
2 x := insertVertex(G, 1);
3 y := insertVertex(G, 2);
4 v := add_vertex(G, 3, {x, y})
5 print degree(v)
```

Pseudocódigo 2.10: Ejemplo de uso de degree. El código crea un grafo G con el vértice 1, el vértice 2 y el vértice 3. Luego se imprime el grado de v, es decir, "2".

3 La estructura h-grafo

En esta sección describimos la estructura h-grafo que usamos para implementar el TAD grafo descripto en la Seccion 2. Esta estructura es particularmente apropiada para grafos ralos, i.e., grafos cuya arboricidad es baja. Esto se ve reflejado en 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 explicar, a continuación, como esta compuesta la estructura interna del h-grafo y luego detallar cada uno de sus operaciones.

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 1
- $H(v)$ 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 2.1.6). 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 2). 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.

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

El proceso para insertar un nuevo vértice 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

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

3.0.2. `G.remove_vertex(v)`

El proceso de remover un vértice v en G consiste en primer lugar en remover las aristas en las que esta involucrado el vértice v. Para eso es necesario recorrer todo el vecindario de v y aplicar la función `G.remove_edge(v,w)` (ver: 3.0.4) para cada $w \in N(v)$.

Por último se elimina físicamente al objeto `v` que representa a v en G.

Pseudocódigo

3.0.3. `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.remove_vertex(v):
2     para cada w en N(v):
3         para cada z en N(v,d(z)).
4                 remove_edge(w,z)
5
6 borrar v de la lista de vertices de G.
```

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 vértices 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 w 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 fsica)
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 fina
```

```
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
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 v
8              Agregar una nueva lista N(z, d(v)+1) a continuacion de N(
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.

3.0.4. `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 3.0.3)

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(w,d(v))`. 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 $d(v) \geq d(z)$ 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-
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 3.1: Implementación de `remove_edge`, deshaciendo las tres fases de la `insert_edge`.

4 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 4.0.1 describe la interfaz pública, mientras que la Sección 4.0.3 describe la estructura interna y sus algoritmos.

Tal como vimos en la Sección 3, la estructura h-grafo guarda, para cada vértice v, una lista $N(v, d)$ que contiene todos los vecinos de v de grado d. Como veremos en la Sección 4.0.3, la idea es utilizar el tipo `std::list` de C++ para almacenar los vértices 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.

En la Sección 3.0.1 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.

4.0.1. Interfaz de C++

Tal como mencionamos anteriormente, en esta sección describiremos la interfaz pública del tipo **Graph** que está implementado con un h-grafo. La clase **Graph** de C++ es la que posee los métodos visibles para el usuario, que fueron descriptos en alto nivel en la Sección 2. Tal cual se observa en la Sección 2, el usuario no tiene acceso directo a los vértices del h-grafo, sino que accede y manipula los mismos a través de *punteros*[1].

[1] No confundir el concepto de puntero con una dirección de memoria. Conceptualmente los punteros son objetos que referencian a otros objetos, pero no necesariamente permiten modificar el objeto referenciado.

Siguiendo la política de la biblioteca estándar de C++, estos punteros se representan usando iteradores (ver Sección 2). Recordemos que la ventaja de los iteradores es que sirven para recorrer la estructura sin saber cómo fue implentada, a la vez que se pueden utilizar como punteros "seguros" que no exponen la estructura interna.

Debido a la importancia de los iteradores en la definición de la interfaz pública, en esta sección empezamos describiendo los tres tipos de iteradores que se proveen.

const_vertex_iterator

El tipo **const_vertex_iterator** es un iterador al conjunto de los vértices del grafo. Siguiendo la terminología de la Sección 2, este iterador es el que se obtiene, por ejemplo, al aplicar la función `vertex_iterator`. Conceptualmente, el tipo **const_vertex_iterator** representa un iterador bidireccional típico de C++.[2] En consecuencia provee todas las funciones requeridas de este tipo de iteradores (e.g., `operator++` para avanzar, `operator--` para retroceder, etc.).

Además de las funciones típicas de un iterador, el tipo **const_vertex_iterator** provee las siguientes funciones que permiten acceder al vecindario del vértice v apuntado por el iterador:

- `neighbor_iterator begin() const` y `neighbor_iterator end() const`: implementan la función `neighbor_iterator` que permite recorrer el vecindario de v. En particular, `begin` retorna un iterador al inicio de $N(v)$ (visto como una secuencia) y `end` retorna un iterador a la posición que le sigue al último elemento de $N(v)$. El tipo (`neighbor_iterator`) de este iterador se describe en la Sección 4.0.1.

- `deg_iterator H_begin() const` y `deg_iterator H_end() const`: implementan la función `H_neighbor_iterator` que permite recorrer los vecinos de grado al menos $d(v)$. En particular, `begin` retorna un iterador al inicio de $H(v)$ (visto como una secuencia) y `end` retorna un iterador a la posición que le sigue al último elemento de $H(v)$. El tipo (`deg_iterator`) de este iterador se describe en la Sección 4.0.1.

Como mencionamos en la Sección 2, las funciones que manipulan un vértice v del grafo requieren un punteros a v. En la implementación, este punteros es un **const_vertex_iterator** cuyo `operator*` denota a v. Hay dos formas para obtener dicho puntero: 1) cuando se inserta un nuevo vértice se obtiene un puntero al vértice agregado, 2) invocando directamente al método `begin` del grafo. El Codigo 4.2 muestra un ejemplo del uso.

[2] Los iteradores bidireccionales permiten recorrer la estructura hacia adelante y atrás. El iterador en todo momento apunta o bien a un vértice del grafo o bien a una posición indeterminada que representa el final de la secuencia iterada.

```
1  void ejemplo_const_vertex_iterator() {
2      Graph<int> G;
3      const_vertex_iterator v1 = G.insertVertex(1);
4      const_vertex_iterator v2 = G.insertVertex(2);
5      //notar el uso de iteradores para agregar arista
6      G.add_edge(v1,v2);
7
8      for(auto it = G.begin(); it != G.end(); ++it) {
9          cout << *it << endl;
10     }
11 }
```

Código 4.1: Ejemplo de uso de `const_vertex_iterator` para manipular el grafo. En el ejemplo, se crean dos vértices unidos por una arista y se imprime dicha arista.

deg_iterator

El tipo **deg_iterator** es un iterador al *high neighborhood* $H(v)$ de un vértice v dado. Siguiendo la terminología de la Sección **??**, **deg_iterator** es el tipo de iterador que se obtiene al aplicar la función `H_neighbor_iterator`[3]. Conceptualmente provee todas las funciones requeridas para ser considerado un iterador bidireccional de C++.

En principio, la única funcionalidad de este iterador es poder acceder a cada vértice de $H(v)$. Sin embargo, para evitar múltiples indirecciones, cada iterador `it` provee las siguientes métodos para acceder al vecindario del vértice $w \in H(v)$ apuntado por `*it`[4]:

- `neighbor_iterator begin() const` y `neighbor_iterator end() const`: aplican `it->begin()` e `it->end()`. Conceptualmente, implementan la función `neighbor_iterator($w$)` de la Sección 2.
- `deg_iterator H_begin() const` y `deg_iterator H_end() const`: aplican `it->H_begin()` e `it->H_end()`. Conceptualmente, implementan la función `H_neighbor_iterator($w$)` de la Sección 2.

Hay dos formas de obtener un **deg_iterator**: 1) invocando `H_begin` sobre un iterador que apunta a un vértice (independientemente del tipo de iterador) y 2) invocando el

[3] La razón por la que se llama **deg_iterator** en lugar de **h_iterator** es que la misma implementación se podria usar para recorrer $N(v, d)$ para cualquier grado d. Sin embargo, no implementamos dicha funcionalidad de forma pública.

[4] Si bien no es del todo cierto, consideramos acá que `*it` es de tipo `const_vertex_iterator`.

método `H_begin(v)` del tipo **Graph**, que toma un **const_vertex_iterator** v como parámetro. El Codigo **??** muestra un ejemplo del uso.

```
1  //Imprime H(v)
2  void print_H_vertice(const Graph<int>& G, const_vertex_iterator v) {
3      for(auto it = G.H_begin(v); it != G.H_end(v); ++it) {
4          //it es de tipo deg_iterator
5          cout << *it << ',';
6      }
7  }
8
9  //Para todo v, imprime los vertices de H(w) para w en H(v)
10 void print_Hs_H(const Graph<int>& G) {
11     for(auto v = G.begin(); v != G.end(); ++v) {
12     //v es de tipo const_vertex_iterator
13         for(auto w = v.begin(); w != v.end(); ++v) {
14             //w es de tipo deg_iterator
15             for(auto z = w.begin(); z != w.end(); ++z) {
16                 //z pertenece a H(w) y w pertenece a H(v)
17                 cout << *z << ',';
18             }
19         }
20     }
21 }
```

Código 4.2: Ejemplo de uso de `deg_iterator`. En el ejemplo, se accede a $H(v)$ para un vértice v invocando `H_begin` usando el grafo, usando directamente un **const_vertex_iterator** y usando un **deg_iterator**.

neighbor_iterator

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 **neighbor_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                       }
```

4.0.2. Interfaz del tipo Graph

Habiendo explicado las interfaces de los iteradores, estamos en condiciones de explicar la interfaz pública de la clase **Graph** que implementa las operaciones vistas en la Sección 2. Remarcamos nuevamente que aquellos métodos que en la Sección 2 requerían punteros toman iteradores como parámetro. El tipo **Graph** provee los siguientes métodos:

- `template<class iter> const_vertex_iterator add_vertex(const Elem& elem, iter begin,` implementa la función `G.insertVertex(N, info)` (Sección 2.1.5). En este caso, `this` se corresponde con `G`, `elem` con `info`, y el rango `[begin, end)` con `N`. Por otra parte, el puntero al nuevo vértice insertado es un **const_vertex_iterator**. Notar que se espera que el tipo `iter` respete el protocolo de un iterador unidireccional de C++.

insertVertex

`insertVertex(const Elem& elem)`: Su propósito es insertar un nuevo vértice al grafo. El tipo de retorno del método es un iterador `const_vertex_iterator` (4.0.1), recordemos que este iterador es el que va a permitir al usuario agregar y borrar aristas. El Codigo 4.3 muestra un ejemplo del uso.

remove_vertex

`remove_vertex(const_vertex_iterator v)`: Su propósito es eliminar un vértice `v` del grafo y dejar en forma consistente el vecindario de los demás vértices con los que el vértice `v` tenía relación. Esto significa eliminar las aristas que involucraban a `v`. El método no tiene ningún tipo de retorno. El Codigo 4.4 muestra un ejemplo del uso.

add_edge

`add_edge(const_vertex_iterator v, const_vertex_iterator w)`: Su propósito es crear la relación entre el vértice `v` y el vértice `w`. Deja en forma consistente el vecindario de `v` y el vecindario de `w`.

```
1
2 int main() {
3       Graph<int> G;
4       const_vertex_iterator v := G.insertVertex(1);
5       const_vertex_iterator w := G.insertVertex(2);
6       for(auto it = G.begin(); it != G.end(); ++it) {
7       cout << *it << endl;
8   }
9 }
```

Código 4.3: Ejemplo de uso de insertVertex. Crea un grafo G de tipo entero con dos vértices, uno con el entero 1 y el otro con el entero 2 como información. Luego, el ciclo imprime "1 2". Ver Sección 4.0.1 para más información de `vertex_iterator`.

remove_edge

`remove_edge(const_vertex_iterator v, const_vertex_iterator w)`: su propósito es eliminar la relación entre los vértices `v` y el vértice `w`. Deja en forma consistente los vecindarios de los dos vértices.

Toma como parámetro un iterador que apunta a (v) y otro que apunta a `w`.

El método no tiene ningún tipo de retorno. El Codigo 4.4 muestra un ejemplo del uso.

4.0.3. 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 3 archivos fuente:

contiene la estructura general del h-grafo.

contiene la descripción de lo que es un vértice.

contiene la estructura de un vecino.

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

```
1  int main() {
2        Graph<int> G;
3        const_vertex_iterator v := G.insertVertex(1);
4        const_vertex_iterator w := G.insertVertex(2);
5    remove_vertex(v);
6        for(auto it = G.begin(); it != G.end(); ++it) {
7        cout << *it << endl;
8    }
```

Código 4.4: Ejemplo de uso de remove_vertex. Reescribimos el pseudocódigo 2.3 a c++. El 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, remueve `v` del grafo y el ciclo imprime "2" como resultado. Ver Sección 4.0.1 para más información de `vertex_iterator`.

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 es una lista de `Vertex` (ver Seccion 4.0.3).

La clase `Graph` también define las estructuras centrales de representación para los vecindarios

Como veremos en la Sección 4.0.3, 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 (CRTP)* [], 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  int main() {
2        Graph<int> G;
3        const_vertex_iterator v := G.insertVertex(1);
4        const_vertex_iterator w := G.insertVertex(2);
5    const_vertex_iterator x := G.insertVertex(3);
6    G.add_edge(v, w);
7    G.add_edge(v, x);
8    G.remove_edge(v, w);
9    for(auto it = G.N_begin(v); it != G.N_end(v); ++it) {
10       cout << *it << ',';
11   }
12 }
```

Código 4.5: Ejemplo de uso de `remove_edge`. Reescribimos el pseudocódigo 2.5 a c++. 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 4.0.1 para más información de `neighbor_iterator`.

Vertex (Vertex.h)

De acuerdo a lo visto en la Sección 4.0.3, `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 vértice (`elem`), el grado del vértice (`degree`) y su vecindario (`neighborhood`). Como vimos en la Sección 3, 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`.

Recordemos que `Graph` concentra las definiciones de qué significa cada tipo, incluyendo qué estructura se usa para representar un vecindario $N(v)$ (`Neighborhood`), qué estructura se usa para representar cada lista $N(v,i)$ y $H(v)$ dentro del vecindario (`degNeighborhood`), y qué estructura se usa para representar un vecino que contiene los self y list pointers (`Neighbor`). Por simplicidad, renombramos estos tipos dentro de la estructura de `Vertex`.

```
1  template<class Elem> class Graph
2  {
3      using elem_type = Elem;
4  private:
5      friend class impl::Vertex<Graph>;
6      friend class impl::Neighbor<Graph>;
7
8      //CRTP: Curiously recurring template pattern
9      using Vertex = impl::Vertex<Graph>;
10     using Neighbor = impl::Neighbor<Graph>;
11
12     using Vertices = std::list<Vertex>;
13     using degNeighborhood = std::list<Neighbor>;
14     using Neighborhood = std::list<degNeighborhood>;
15     using neighbor_iterator = typename Vertex::neighbor_iterator;
16     using deg_iterator = typename Vertex::deg_iterator;
17 };
```

Código 4.6: Estructura del tipo Grafo en C++.

Neighbor (Neighbor.h)

Los objetos de esta clase se van a almacenar en listas que juntas representan el vecindario $N(v)$ de un vértice v.

Cada `Neighbor` representa un vecino w de v que mantiene toda la información necesaria para que sea eficiente eliminar v de $N(w)$ cuando se tiene acceso a w en $N(v)$, como se vio en la seccion 3.0.4

Esta estructura es simplemente una tripla de tres punteros:

`neighbor`: que es un iterador al vértice v en la lista de vértices de `Graph`, descripto en la sección 4.0.1 item (**??**)

`list_pointer`: es un puntero a la lista en que esta `v` en `w`

`self_pointer`: es un puntero directo a v en $N(v)$.

```
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      //Estructura donde se guardan los vertices
8      using Vertices = typename Graph::Vertices;
9      //Estructura de vecino.
10     using degNeighborhood = typename Graph::degNeighborhood;
11     //Estructura donde se guardan todos los vecindarios de todos los
12     using Neighborhood = typename Graph::Neighborhood;
13
14     elem_type elem;  //elemento
15     size_t degree;  //grado
16     Neighborhood neighborhood;  //vecindario
17 }
```

Código 4.7: Estructura del tipo Grafo en C++.

```
1    struct Neighbor
2       {
3            using NeighborPtr = typename Graph::Vertices::iterator;
4            using SelfPtr = typename Graph::degNeighborhood::iterator;
5            using ListPtr = typename Graph::Neighborhood::iterator;
6            using elem_type = typename Graph::Vertex::elem_type;
7
8            Neighbor() = default;
9            Neighbor(NeighborPtr neighbor) : neighbor(neighbor) {}
10
11           /**
12            * Es un puntero directo al vecino en la lista de vertices.
13            */
14           NeighborPtr neighbor;
15
16           /**
17            * Es un puntero directo a la posicion de v en la lista de N(w) qu
18            */
19           SelfPtr self_pointer;
20
21           /**
22            * Es un puntero directo a la lista de L(w) que contiene a v. (Sol
23            * cuando v esta en high_neighborhood de w, dejamos low_neighborho
24            */
25           ListPtr list_pointer;
26   }
```

Código 4.8: Neighbor.h

5 Resultados de la experimentación

"tiempo vs complejidad"

A continuación mostraremos los gráficas del tiempo de ejecución en crear el grafo y encontrar los triángulos en él en función a la cantidad de nodos. El eje y refleja el tiempo en milisegundos, el eje x representa la cantidad de nodos.

Gráfico del tiempo de ejecución de 100 grafos connected_watts_strogatz_graph

Figura 5.1: El grafo más pequeño es de 10 nodos y 20 aristas. El grafo más grande es de 505 nodos y 1010 aristas.

Gráfico del tiempo de ejecución de 100 grafos erdos_renyi_graph

Figura 5.2: El grafo más pequeño es de 10 nodos y 2 aristas. El grafo más grande es de 505 nodos y 4443 aristas. El tiempo máximo de ejecución es de 79088.2 ms, es decir, poco más de 13 minutos

Gráfico del tiempo de ejecución de 100 grafos gnp_random_graph

Figura 5.3: El grafo más pequeño es de 10 nodos y 1 aristas. El grafo más grande es de 505 nodos y 1258 aristas.

Gráfico del tiempo de ejecución de 100 newman_watts_strogatz_graph

Figura 5.4: El grafo más pequeño es de 10 nodos y 29 aristas. El grafo más grande es de 505 nodos y 1322 aristas.El tiempo máximo de ejecución es de 4603.91 ms, es decir, alrededor de 7 minutos