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, 12 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

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 4.1, da cuenta de la interfaz del h-grafo siguiendo los lineamientos de C++. La segunda, Sección 4.2, 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 vw *incide* tanto en v como en w y que, por lo tanto, 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*, mientras que $H(v)$ contiene a todos los vecinos de grado al menos $d(v)$. Notar que $N(v)$ (resp. $H(v)$)es la unión de todos los k-vecindarios de v (resp. con grado al menos $d(v)$, i.e., $N(v) = \bigcup_{i=0}^{n-1} N(v, k)$ y $H(v) = \bigcup_{i=d(v)}^{n-1} N(v, k)$.

Un grafo G' es un *subgrafo* de G cuando $V(G') \subseteq V(G)$ y $E(G') \subseteq E(G)$. Si $V(G') = V(G)$, entonces G' es un subgrafo *generador* de G, mientras que si $E(G') = \{vw \in E(G) \mid v, w \in V(G')\}$, entonces G' es un subgrafo *inducido* de G. Sean $V \subseteq V(G)$, $E \subseteq E(G)$, $v \in V(G)$ y $vw \in E(G)$. Para simplificar la notación de los algoritmos, vamos a escribir:

- $G \setminus E$ para referirnos al subgrafo de G generado por $E(G) \setminus E$,

- $G \setminus V$ para referirnos al subgrafo de G inducido por $V(G) \setminus V$,
- $G - vw$ para referirnos a $G \setminus \{vw\}$, y
- $G - v$ para referirnos a $G \setminus \{v\}$.

En otras palabras, $G \setminus E$ (resp. $G - \{vw\}$) se obtiene eliminando las aristas de E (resp. la arista vw), mientras que $G \setminus V$ (resp. $G - v$) se obtiene eliminando los vértices de V (resp. el vértice v) y las aristas que inciden en algún vértice de V (resp. en v). Análogamente, escribimos $H + vw$ para indicar que $G = H - vw$ y, únicamente en caso en que $N(v) = \emptyset$, $H = G + v$ para indicar que $G = H - v$.

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.

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

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

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

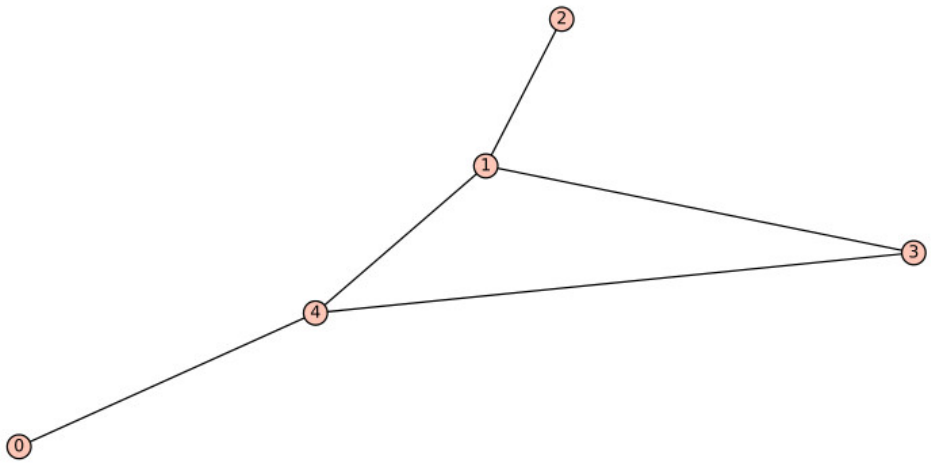

Figura de un grafo en el que mostramos las definiciones

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).

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`

2.2. `insert_vertex(G, info)`

Modifica un objeto `G` que representa un grafo G a fin de representar el grafo $G + v$. Al vértice v se le puede asociar cierta información que está reflejada por el parámetro `info`. Retorna un `vertex_iterador v` que está posicionado en el vértice v agregado. Decimos que `v` *representa* o *apunta* a v. Este iterador `v` debe usarse para interactuar con G a fin de modificar sus propiedades. Notar que el iterador `v` es un objeto conocido sólo por `G` y no es compartido con otras instancias del TAD GRAFO. 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 al grafo $G + v$.

Retorna: un `vertex_iterator v` al vértice v.

Complejidad temporal: O(n) Esta complejidad esta mal. Revisar.

```
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 2.7 para más información de `vertex_iterator`.

2.3. remove_vertex(G, v)

Modifica un objeto `G` que representa un grafo G a fin de representar al grafo $G - v$. Para ello, `v` debe ser un `vertex_iterator` que represente a v; el resultado de la operación está indefinido cuando `v` no representa a un vértice de G. Como resultado de esta operación, se invalidan aquellos iteradores que están relacionados con v. Esto incluye a cualquier:

- `vertex_iterator` que apunte a v,
- `neighbor_iterator` o `H_neighbor_iterator` asociado a v, y
- `neighbor_iterator` o `H_neighbor_iterator` asociado a $w \in N(v)$ que apunte a v.

El Pseudocódigo 2.3 muestra cómo usar `remove_vertex`.

Parámetros: `G` representa un grafo G y `v` representa a $v \in V(G)$.

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

Complejidad temporal: O(n) Esta complejidad esta mal. Revisar.

```
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 (letra 'a') y el segundo `w` representa al vértice w (letra 'b'). Luego, remueve `v` del grafo y, en consecuencia, el ciclo imprime 'b' como resultado. Ver Sección 2.7 para más información de `vertex_iterator`.

2.4. add_edge(G, v, w)

Modifica un objeto `G` que representa un grafo G a fin de representar el grafo $G + vw$. Para ello, `v` y `w` deben ser `vertex_iterator` que representen a v y w, respectivamente; el resultado de la operación está indefinido cuando `v` o `w` no representan a vértices de G.

El Pseudocódigo 2.4 muestra cómo usar `add_edge`.

Parámetros: `G` representa un grafo G, `v` y `w` representan a $v, w \in V(G)$, respectivamente.

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

Complejidad temporal: Falta

```
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 vértices v (número 1) y w (número 2) y agrega la arista vw. El ciclo imprime la información de los vecinos de v; en este caso el número 2. Ver Sección 2.8 para más información de `neghbor_iterator`.

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

Modifica un objeto `G` que representa un grafo G a fin de representar el grafo $G - vw$. Para ello, `v` y `w` deben ser `vertex_iterator` que representen a v y w, respectivamente; el resultado de la operación está indefinido cuando `v` o `w` no representan a vértices de G. Como resultado de esta operación, se invalidan todos los `neighbor_iterator` o `H_neighbor_iterator` asociados a v o w que apunten a v o w.

El Pseudocódigo 2.5 muestra cómo usar `remove_edge`.

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

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

Complejidad temporal:

```
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 2.8 para más información de `neghbor_iterator`.

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

Modifica un objeto `G` que representa un grafo G a fin de representar un grafo H tal que $G = H - v$, donde la información de v está reflejada en el parámetro `info`. Para construir H, `N` debe ser una lista de `vertex_iterator` de G, de forma tal que $N(v)$ en H coincide con los vértices representados por `N`. Intuitivamente, `add_vertex(G, info, N)` representa la grafo que se obtiene de agregar un nuevo vértice v cuyo vecindario esta representado por `N`. Retorna un `vertex_iterador v` que representa al vértice v agregado.

El Pseudocódigo 2.6 muestra cómo usar `add_vertex`.

Parámetros: `G` representa un grafo G, `info` es cualquier objeto y `N` es una lista de `vertex_iterator` de `G` que representa al vecindario N.

Propósito: modifica `G` para representar un grafo $G + \{vw \mid w \in N\}$.

Retorna: un `vertex_iterator v` al vértice v.

Complejidad temporal: O(1) Esta mal, revisar

2.7. `vertex_iterator(G)`

Provee de un `vertex_iterador` que permite desplazarse a través de los vértices del objeto `G` que representa un grafo G. Además del recorrido, cada `vertex_iterator` puede usarse

```
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 vertices x, y y v, donde v es vecino de x y z. Luego, el ciclo imprime "12".

para manipularlos al vértices que apunta de forma eficiente. Tener en cuenta que el iterador se invalida si se elimina el vértice apuntado por él. Las inserciones o remociones de otros vértices y aristas no tienen efectos en el iterador.

El Pseudocódigo 2.7 muestra cómo usar `vertex_iterator`.

Parámetros: `G` representa un grafo G.

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

Retorna: un `vertex_iterador i` que apunta al "primer" vértice 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 imprime alguna permutación de la secuencia 1,2.

2.8. `neighbor_iterator(v)`

Provee de un iterador, llamado `neighbor_iterator`, que permite recorrer los vecinos del vértice v representado por el `vertex_iterator v`. De esta forma se puede acceder eficientemente a cada vecino de v. Decimos que el iterador está *asociado* a v, mientras que apunta a un vértice $w \in N(v)$ correspondiente a la posición actual del iterador.

Si bien no está implementado en esta versión, un `neighbor_iterator` también podría usarse para eliminar directamente la arista entre v y w, donde w es el vértice apuntado por el iterador. Tener en cuenta que el `neighbor_iterator` asociado a v que apunta a w se invalida cuando se elimina vw. Más aún, las operaciones que alteren a $N(v) \cup N(w)$ (i.e., inserciones y remociones de vecinos de v o w) pueden alterar el orden en el que se recorren los vértices.

El Pseudocódigo 2.8 muestra cómo usar `neighbor_iterator`.

Parámetros: `v` representa un vértice v.

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

Retorna: un `neighbor_iterator w` para recorrer $N(v)$.

Complejidad temporal: Falta

```
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 imprime un permutación de 1,2.

2.9. `H_neighbor_iterator(v)`

Provee de un iterador, llamado `H_neighbor_iterator`, que permite recorrer $H(v)$ para el vértice v representado por el `vertex_iterator v`. De esta forma se puede acceder eficientemente a cada vecino "grande" de v. Decimos que el iterador está *asociado* a v, mientras que *apunta* a un vértice $w \in H(v)$ correspondiente a la posición actual del iterador. Si bien no está implementado en esta versión, un `H_neighbor_iterator` también podría usarse para eliminar directamente la arista entre v y w, donde w es el vértice apuntado por el iterador. Tener en cuenta que el `H_neighbor_iterator` asociado a v que apunta a w se invalida cuando se elimina vw. Más aún, las operaciones que alteren a $N(v) \cup N(w)$ (i.e., inserciones y remociones de vecinos de v o w) pueden alterar el orden en el que se recorren los vértices. En particular, el iterador podría dejar de recorrer $H(v)$, con lo cual se desaconseja su uso ante alteraciones.

El Pseudocódigo 2.9 muestra cómo usar `H_neighbor_iterator`.

Parámetros: `v` representa un vértice v.

Propósito: proveer al usuario de un mecanismo para iterar los vecinos de grado al menos $d(v)$.

Retorna: un iterador `i` apuntando al "primer" vértice de $H(v)$.

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 imprime 3.

2.10. `degree(v)`

Dado un `vertex_iterator v` que representa un vértice v de un grafo G, devuelve el grado de v en G. El Pseudocódigo 2.10 muestra cómo usar `degree`.

Parámetros: `v` representando un vértice v.

Retorna: $d(v)$.

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 algoritmo imprime $d(v) = 2$.

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 está diseñada 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 artículo que introduce la estructura [**?**]. La idea es explicar, a continuación, como esta compuesta la estructura interna del h-grafo y luego detallar cada uno de sus operaciones.

La estructura h-grafo `G` que representa a un grafo G es simplemente un lista `G.vertices` que tiene un objeto `v` para cada vértice v del grafo G. Este objeto *representa* a v y mantiene la tripla $(d(v), \mathcal{N}(v), H(v))$, donde $\mathcal{N}(v)$ es la secuencia que se obtiene de eliminar los conjuntos vacíos de $N(v,1), \ldots, N(v,d(v)-1)$. Recordemos que:

- $d(v)$ es el grado de v, i.e., su cantidad de vecinos,
- $N(v,k)$ es el k-vecindario de v, i.e., el conjunto de vecinos de v que tienen grado k, y
- $H(v)$ es el *vecindario alto* de v, i.e., el conjunto de vecinos de v que tienen grado al menos $d(v)$.

Asimismo, recordemos que nuestra implementación almacena cierta información asociada a cada vértice v.

La Figura 3.1 muestra cómo se representa el h-grafo `G` de G. Obviamente, el objeto `v` que representa a $v \in V(G)$ mantiene un número `v.deg` para representar a $d(v)$ y un puntero `v.info` para representar la información asociada a v. Para representar a $H(v)$ utiliza una lista doblemente enlazada `v.high`. Por último, para representar $\mathcal{N}(v)$, el h-grafo mantiene una lista doblemente enlazada `v.low`, donde cada elemento `v.low(k)` representa un k-vecindario $N(v,k)$. En esta lista los k-vecindarios de v aparecen ordenados de acuerdo al grado de sus elementos; observar que `v.low(k)` no es necesariamente el k-ésimo elemento, ya que los k-vecindarios vacíos no se almacenan en `v.low`. Por último, cada objeto `w` de alguna lista `v.low(k)` $\cup$ `v.high`, que representa a $w \in N(v)$, mantiene los punteros (iteradores, ver abajo) `w.neighbor_pointer`, `w.list_pointer` y `w.self_pointer` tales que:

- `w.neighbor_pointer` apunta al objeto de `G.V` que representa a w,

- `w.list_pointer` apunta a la lista `N` de `w.low` ∪ `w.high` que contiene al objeto `v'` que representa a v, y
- `w.self_pointer` apunta a la posición de `v'` dentro de `N`. En otras palabras, `w.self_pointer` apunta a la encarnación de v en el vecindario de w.

$$v_1\boxed{1} = \begin{cases} \mathtt{d} = 1 \\ \mathcal{N} = \emptyset \\ \mathtt{H}\boxed{7} = [v_2\boxed{8} : \mathtt{s} = 10, \mathtt{l} = 9, \mathtt{n} = 2] \end{cases}$$

$$v_2\boxed{2} = \begin{cases} \mathtt{d} = 4 \\ \mathcal{N} = \begin{cases} \mathtt{N}(1)\boxed{9} = [v_1\boxed{10} : \mathtt{s} = 8, \mathtt{l} = 7, \mathtt{n} = 1], \\ \mathtt{N}(2)\boxed{11} = [v_3\boxed{12} : \mathtt{s} = 17, \mathtt{l} = 16, \mathtt{n} = 3, v_4\boxed{13} : \mathtt{s} = 20, \mathtt{l} = 19, \mathtt{n} = 4] \end{cases} \\ \mathtt{H}\boxed{14} = [v_5\boxed{15} : \mathtt{s} = 28, \mathtt{l} = 27, \mathtt{n} = 5] \end{cases}$$

$$v_3\boxed{3} = \begin{cases} \mathtt{d} = 2 \\ \mathcal{N} = \emptyset \\ \mathtt{H}\boxed{16} = [v_2\boxed{17} : \mathtt{s} = 12, \mathtt{l} = 11, \mathtt{n} = 2, v_5\boxed{18} : \mathtt{s} = 25, \mathtt{l} = 24, \mathtt{n} = 5] \end{cases}$$

$$v_4\boxed{4} = \begin{cases} \mathtt{d} = 2 \\ \mathcal{N} = \emptyset \\ \mathtt{H}\boxed{19} = [v_2\boxed{20} : \mathtt{s} = 13, \mathtt{l} = 11, \mathtt{n} = 2, v_5\boxed{21} : \mathtt{s} = 26, \mathtt{l} = 24, \mathtt{n} = 5] \end{cases}$$

$$v_5\boxed{5} = \begin{cases} \mathtt{d} = 4 \\ \mathcal{N} = \begin{cases} \mathtt{N}(1)\boxed{22} = [v_6\boxed{23} : \mathtt{s} = 30, \mathtt{l} = 29, \mathtt{n} = 6] \\ \mathtt{N}(2)\boxed{24} = [v_3\boxed{25} : \mathtt{s} = 18, \mathtt{l} = 16, \mathtt{n} = 3, v_4\boxed{26} : \mathtt{s} = 21, \mathtt{l} = 19, \mathtt{n} = 4] \end{cases} \\ \mathtt{H}\boxed{27} = [v_2\boxed{28} : \mathtt{s} = 15, \mathtt{l} = 14, \mathtt{n} = 2] \end{cases}$$

$$v_6\boxed{6} = \begin{cases} \mathtt{d} = 1 \\ \mathcal{N} = \emptyset \\ \mathtt{H}\boxed{29} = [v_5\boxed{30} : \mathtt{s} = 23, \mathtt{l} = 22, \mathtt{n} = 5] \end{cases}$$

Figura 3.1: Estado de los objetos almacenados por un h-grafo para representar al grafo de la Figura **??**. A la izquierda de cada signo = se muestra un objeto representando al vértice v_i, para $i = 1, \ldots, 6$. La posición de cada uno de estos objetos en la memoria de la máquina se muestra en una caja; por ejemplo, el objeto representando a v_1 ocupa la celda 1 de la memoria. La información que se mantiene para cada vértice se muestra a la derecha de su signo =. Aquí también la posición de cada objeto en la memoria aparece en una caja, pero sólo para aquellos objetos que están apuntados por algún puntero. Por ejemplo, el objeto `H` correspondiente a v_1 ocupa la celda 7. Las letras `s`, `l` y `n` representan a los `self_pointer`, `list_pointer` y `neighbor_pointer`, respectivamente.

Como discutimos en la Sección 1, uno de los objetivos del trabajo es utilizar las estructuras que provee la biblioteca estándar del lenguaje de implementación. En particular, queremos utilizar el tipo lista que esta biblioteca provee. Como parte de nuestra estructura, necesitamos almacenar punteros a los "nodos" de la lista. Este es un inconveniente

no menor, ya que no tenemos acceso a la estructura interna de la lista. Sin embargo, observando el uso que se hace de dichos punteros, vemos que alcanza con que el tipo lista provea algunas funciones que permitan crear, borrar y mover los objetos (i.e., nodos) eficientemente, para lo cual no necesitamos conocer la estructura de la lista. Cada lenguaje provee un mecanismo distinto para resolver este problema. En el lenguaje que elegimos (C++), la solución es usar iteradores de la lista, ya que proveen esta interfaz de *puntero restringido*.

Bajo la misma filosofía de ocultamiento de la información, la estructura del tipo GRAFO está oculta al usuario. En particular, los objetos de cada grafo son privados, en el sentido de que un usuario no pueden manipular estos objetos directamente. En su lugar, el usuario modifica v a través de un puntero que obtiene al insertar el vértice (ver Sección 2.7). Siguiendo el ejemplo de la biblioteca estándar de C++, vamos a utilizar iteradores para proveer este comportamiento de puntero restringido. Es decir, usamos iteradores para brindar acceso en $O(1)$ a los elemento sin mostrar la estructura interna al usuario. Vamos a suponer que existe una operación `vertex_iterator` que dado un objeto `v` de `G.vertices` crea un iterador `it_v` que apunta a `v`. Luego, usando el operador `it_v->X` podemos acceder al campo (o método) `X` de `v`. Asimismo, suponemos la existencia de las funciones

- `v.neighbors()` que retorna un iterador a los vértices de `G.vertices` que pertenecen a `v.low` $\cup$ `v.high`, y
- `v.H_neighbors()` que retorna un iterador a los vértices de `G.vertices` que pertenecen a `v.high`.

En el resto de esta sección describimos cómo implementar las operaciones principales del TAD GRAFO (ver Seccion 2) usando la estructura h-grafo. Para que quede claro que estamos usando la estructura h-grafo, vamos a escribir la signatura de cada operación `oper(G, ···)` usando la notación punto, i.e., `G.oper(···)` a fin de remarcar que tenemos acceso a la estructura interna de `G`. Asimismo, vamos a suponer la existencia de las siguientes funciones básicas sobre listas. Vale remarcar que estas funciones existen en la biblioteca estándar de nuestro lenguaje de implementación.

- `L.append(what, pos = None)`: agrega `what` inmediatamente atrás de la posición de `L` apuntada por el iterador `pos`. Si `pos = None`, entonces lo agrega al final.
- `L.prepend(what, pos = None)`: agrega `what` inmediatamente antes de la posición de `L` apuntada por el iterador`pos`. Si `pos = None`, entonces lo agrega al inicio.
- `L.first()`: retorna un iterador al primer elemento de `L`.
- `L.last()`: retorna un iterador al último elemento de `L`.
- `L.empty()`: indica si la `L` está vacía.
- `L.erase(pos)`: elimina el elemento de `L` apuntado por el iterador `pos`.

- `L.transfer(z)`: mueve el objeto apuntado por el iterador `z` hacia el inicio de `L`. La lista que antes contenía a `z` ya no lo contiene más. Esta operación no afecta a los iteradores.

3.1. `G.insert_vertex(N = ∅, info = None)`

El proceso para insertar un nuevo vértice v en G se divide en dos partes. Primero, se inserta un nuevo objeto `v` que representa a v en el grafo $H = G + v$. Segundo, se inserta una arista vw en H para cada w que este representado por uno de los objetos de N. Claramente, el grafo H así obtenido es tal que $N(v)$ se representa con `N`. El Pseudocódigo 3.1 muestra cómo está implementado `insert_vertex`.

```
1 G.insert_vertex(N = [], info = None):
2   v := G.vertices.append({deg = 0, high = [], low = [], info = info})
3   for w in N:
4     G.insert_edge(v, w)
5   return vertex_iterator(v)
```

Pseudocódigo 3.1: Implementación de `insert_vertex`.

3.2. `G.remove_vertex(v)`

Para remover un vértice v de G revertimos el proceso de inserción. Es decir, en primer lugar removemos las aristas que inciden en v y luego eliminamos físicamente al objeto `v` de `G.vertices` que representa a v. Para eliminar las aristas, se recorre el vecindario de v y se aplica la función `G.remove_edge(v,w)` (ver Sección 3.3.1) para cada $w \in N(v)$. El Pseudocódigo 3.2 muestra cómo está implementado `remove_vertex`.

```
1 G.remove_vertex(v):
2   for w in v->neighbors():
3     G.remove_edge(v,w)
4   G.vertices.erase(v)
```

Pseudocódigo 3.2: Implementación de `remove_vertex`.

3.3. G.insert_edge(v, w)

El algoritmo para insertar una nueva arista vw al grafo G (para $v, w \in V(G)$) tiene tres fases bien delimitadas que describimos a continuación. En esta descripción suponemos que `v` y `w` son los `vertex_iterator` que representan a v y 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 vértices en un nueva lista `N` que represente a $N(v, d(v))$. Luego, para actualizar el vecindario de v, agregamos `N` al final de `v->low` y movemos cada `z'` de `v->high` con `z->deg = v->deg` hacia `N`, donde `z = z'->neighbor_pointer` es el objeto que representa a z en `G.vertices`. Al mover `z'`, tenemos que actualizar los iteradores asociados `z'` y al objeto `v'` que representa a v en el vecindario de `z`. Tanto `z'->neighbor_pointer` (que apunta a `z`), como `z'->list_pointer` (que indica la lista de `z->low` $\cup$ `z->high` donde se encuentra `v'`) y `z->self_pointer` (que indica la posición de `v'` en `z'->list_pointer`) son correctos. Análogamente, `v'->neighbor_pointer` es correcto y, si uno tiene cuidado de mover `z'` de forma tal de no invalidar los iteradores que lo apuntan, entonces también `v'->self_pointer` es correcto. En cambio, hay que actualizar `v'->list_pointer` a fin de apuntar a la nueva lista `N`. Notemos que `v'` se obtiene eficientemente accediendo a `z'->self_pointer`. Una vez actualizado el vecindario de v, se procede análogamente con w. Es decir, agregamos la 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 sólo 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 $N(z)$, para 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 `v'` que representa a v no debe actualizarse. En cambio, si $v \notin H(z)$, entonces $d(v) < d(z)$ en G y $v \in N(z, d(v)) \in \mathcal{N}(z)$. Por lo tanto, tenemos que mover `v'` desde `z->low(d(v))` hacia `z->low(d(v)+1)` (o `z->high` si `v->deg + 1 = z->deg`) para todo `z = z'->neighbor_pointer` tal que `z'` pertenece a `v->high`. Notemos que `v'` es el objeto apuntado por `z'->self_pointer`. Por lo tanto, podemos acceder eficientemente a `v'` para moverlo. Al igual que en la primer fase, tenemos que actualizar `z'->list_pointer` para apuntar a la nueva lista que contiene a `v'`. 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. Supongamos que $d(v) \leq d(w)$. Primero creamos un objeto `w'` en `v->high` que represente a w. Luego, buscamos la lista `N` que representa a $N(w, d(v))$, recorriendo `w->low` (si $d(v) < d(w)$) o tomando `N = w->high` (si $d(v) = d(w)$). Finalmente, insertamos un objeto `v'` que represente a v en $N(w)$ e incrementamos `v->deg` y `w->deg` en 1.

El Pseudocódigo 3.3 resume la implementación del algoritmo.

Ejemplo de una ejecucion.

3.3.1. `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 3.3). Sean `v` y `w` los `vertex_iterator` que representan a v y w.

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, intercambiando v con w de ser necesario, que $d(v) \leq d(w)$, i.e., $w \in H(v)$. El primer paso es recorrer `v->high` hasta localizar al objeto `w'` que representa a w. En el segundo paso, usamos `w'->list_pointer` y `w'->self_pointer` para acceder y borrar al objeto que representa a v en la lista `w->low(d(v))` (o `w->high` cuando $d(v) = d(w)$). Por último, eliminamos `w'` de `v->high` y decrementamos `v->deg` y `w->deg` en 1.

Deshacer segunda fase. Al igual que para la inserción, la segunda fase consiste en actualizar $N(z)$ para 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 al objeto `v'` que representa a v en $N(z)$. En cambio, si $d(v) < d(z)$, entonces tenemos que mover `v'` desde `z->low(d(v)+1)` (si $d(v)+1 < d(z)$) o `z->high` (si $d(v)+1 = d(z)$) hacia la lista `z->low(d(v))`, donde `z` el el objeto que representa a z en `G.vertices`. Notemos que $z \in H(v)$ en este caso. En consecuencia, obtenemos cada `z = z'->neighbor_pointer` recorriendo `v->high`. Para obtener y mover a `v'`, utilizamos `z'->list_pointer` y `z'->self_pointer` como en la segunda fase de la inserción. Finalmente, actualizamos `z'->list_pointer` para reflejar el movimiento.

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.

El Pseudocódigo 3.4 resume la implementación del algoritmo.

```
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    if v->deg > w->deg: swap(v, w)
10   w' = v->high.prepend({neighbor_pointer=w})
11   if v->deg < w->deg: ubicar N := w->low(v->deg) recorriendo w->low
12   else: N := w->high
13   v' = N.prepend({neighbor_pointer=v})
14   v'->self_pointer := N.first()
15   v'->list_pointer := N
16   w'->self_pointer := v->high.first()
17   w'->list_pointer := v->high
18   v->deg += 1; w->deg += 1
19
20 G.update_neighborhood(v):
21   N := []
22   for z' in v->high if v->deg = z'->neighbor_pointer->deg:
23     N.transfer(z')
24     //Obs: z'->self_pointer apunta a v dentro de z.high
25     z'->self_pointer->list_pointer := N
26   if not N.empty(): v->low.append(N)
27
28 G.update_neighbors(v):
29   for z' in v->high if v->deg < z'->neighbor_pointer->deg:
30     //Obs: z'->neighbor_pointer representa a z en G.vertices
31     z = z'->neighbor_pointer
32     //     z'->self_pointer representa a v en z->low,
33     v' := z'->self_pointer
34     //     y z'->list_pointer representa z->low(d(v))
35     N_d := z'->list_pointer
36
37     if v->deg + 1 < z->deg y z->low.next(N_d) != z->low(v->deg+1):
38       N := z->low.append([], pos = N_d)
39     if v->deg + 1 = z->deg:
40       N := z->high
41     N.transfer(v')
42     Si N_d = []: z->low.erase(N_d)
```

Pseudocódigo 3.3: Implementación de `insert_edge`.

```
1  G.remove_edge(v,w) {
2      if v->deg > w->deg: swap(v,w)
3      //Deshacer Fase 3
4      buscar w' tal que w'->neighbor_pointer = w en v->high
5      w'->list_pointer->erase(w'->self_pointer)
6      v->high.erase(w')
7      v->deg -= 1; w->deg -= 1
8      //Deshacer Fase 2
9      G.update_neighbors_delete(v)
10     G.update_neighbors_delete(w)
11     //Deshacer Fase 2
12     G.update_neighborhood_delete(v)
13     G.update_neighborhood_delete(w)
14 }
15
16 G.update_neighbors_delete(v) {
17   for z' en v->high:
18     z := z'->neighbor_pointer
19     v' := z'->self_pointer
20     N := z'->list_pointer
21     P := if v->deg + 1 = z->deg then prev(N) else z->low.last()
22     if P != z->low(d(v)): P := z->low.append([], P)
23     P->transfer(z'->self_pointer)
24     if N = [] and v->deg+1 < z->deg: z->low.erase(N)
25     z'->list_pointer := P
26 }
27
28 G.update_neighborhood_delete(v){
29   if v->low.last() != v->low(v->deg): return
30   for z' in v->low.last():
31     v->high.transfer(z')
32     //z'->self_pointer representa a v en N(z).
33     z->self_pointer->list_pointer := v->high
34   v->low.erase(v->low.last())
35 }
```

Pseudocódigo 3.4: Implementación de `remove_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++. Este capítulo imita la estructura de los Capítulos 2 y 3, salvo que se omite la implementación de los algoritmos.[1] Primero describimos la interfaz publica (Sección 4.1) y luego la la estructura interna (Sección 4.2).

4.1. Interfaz de C++

En esta sección describiremos la interfaz pública de la clase `Graph` que implementa al TAD Grafo usando la estructura h-grafo. La interfaz publica de `Graph` contiene, en lenguaje C++, las funciones que fueron descriptas en alto nivel en la Sección 2. Tal cual se observa y explicita en las Secciónes 2 y 3, el usuario no tiene acceso directo a los objetos que representan a los vértices del grafo en la estructura interna, sino que accede a los mismos a través de *punteros restringidos*. Siguiendo la política de la biblioteca estándar de C++, estos punteros se representan usando iteradores. Es deseable que estos iteradores se encuadren en la jerarquía estándar de iteradores de C++.

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 en este trabajo. (Si bien no forma parte de este trabajo, es posible implementar otros iteradores, e.g. iteradores para recorrer y borrar aristas.)

4.1.1. `Graph::vertex_iterator`

La clase `Graph::vertex_iterator`, o simplemente `vertex_iterator`, define los iteradores que apuntan al conjunto de los vértices del grafo. Este iterador es el que se obtiene, por ejemplo, al aplicar las funciones `vertex_iterator` (Sección 2.7) y `add_vertex` (Sección 2.6). Formalmente, la clase `vertex_iterator` implementa un iterador bidireccional

[1] Los mismos pueden encontrarse en el código C++ que se adjunta a este documento.

constante de acuerdo a la jerarquía de C++.[2] En consecuencia, su interfaz 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, la clase `vertex_iterator` provee las siguientes funciones que permiten acceder al vecindario del vértice v apuntado por el iterador:

- `operator*() const`: retorna la información asociada al vértice.
- `neighbor_iterator begin() const` y `neighbor_iterator end() const`: implementan la función `neighbor_iterator` (ver Sección 2.8) 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 $N(v)$. La clase `neighbor_iterator` se describe en la Sección 4.1.2.
- `deg_iterator H_begin() const` y `deg_iterator H_end() const`: implementan la función `H_neighbor_iterator` (Sección 2.9) 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 clase `deg_iterator` se describe en la Sección 4.1.2.

Hay dos formas para obtener una instancia de la clase `vertex_iterator`:

1. Invocando el método `Graph::add_vertex`, que implementa la función `add_vertex` (Seccion 2.6) para agregar un nuevo vértice, y
2. Invocando alguno de los métodos `Graph::begin` o `Graph::end` que juntos implementan `vertex_iterator` (Sección 4.1.1).

El Código 4.1 muestra cómo usar un `vertex_iterator`.

4.1.2. `Vertex::deg_iterator` y `Vertex::neighbor_iterator`

La clase `Vertex::deg_iterator`, o simplemente `deg_iterator`, define los iteradores que permiten recorrer $H(v)$ para un vértice v. Este iterador es el que se obtiene, por ejemplo, al aplicar las función `H_neighbor_iterator`[3] (Sección 2.9). Análogamente, la clase `Vertex::neighbor_iterator`, o simplemente `neighbor_iterator`, define los iteradores

[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. Al ser constante, el iterador no permite modificar la información asociada a un vértice.

[3] La razón por la cual la clase se llama `deg_iterator` en lugar de `H_iterator` es que esta clase se puede usar para recorrer $N(v, d)$ para cualquier grado $d < d(v)$ (de hecho, esa es una funcionalidad descrita en [**?**]). Sin embargo, no implementamos dicha funcionalidad en esta versión de forma pública.

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

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

que permiten recorrer $N(v)$; se obtienen, por ejemplo, al aplicar la función `neighbor_iterator` (Sección 2.8). Formalmente, ambas clases proveen todas las funciones requeridas para ser considerados iteradores bidireccionales constantes de C++.

En principio, la única funcionalidad de ambos iteradores es poder acceder al vértice w de $N(v)$ (o $H(v)$) que está siendo apuntado. Hay dos opciones para implementar esta solución. La primera, implementar el **`operator*`** a fin de que retorne un `vertex_iterator` que apunte a w. La segunda, permitir el casteo automático desde `deg_iterator` y `neighbor_iterator` hacia `vertex_iterator`. Para evitar múltiples indirecciones, elegimos la segunda opción, dejando el **`operator*`** como forma de acceder a la información asociada a w. Además, cada iterador provee métodos para acceder al vecindario de w. En resumen, se proveen las siguientes operaciones:

- **`operator*() const`**: retorna la información asociada a w.
- **`operator`** `vertex_iterator() ` **`const`**: casteo automático de un `deg_iterator` (o `neighbor_iterator`) a un `vertex_iterator` que apunta a w.
- `vertex_iterator as_vertex_iterator() ` **`const`**: casteo explícito.
- `neighbor_iterator begin() ` **`const`** y `neighbor_iterator end() ` **`const`**: aplican `w->begin()` y `w->end()`, donde `w` es el `vertex_iterator` que apunta a w. Conceptualmente, implementan la función `neighbor_iterator` (Sección 2.8) con parámetro w.
- `deg_iterator H_begin() ` **`const`** y `deg_iterator H_end() ` **`const`**: aplican `w->H_begin()` e `w->H_end()` donde `w` es el `vertex_iterator` que apunta a w. Conceptualmente, implementan la función `H_neighbor_iterator` (Sección 2.9) con parámetro w.

Hay dos formas de obtener una instancia de las clase `deg_iterator`:

1. invocando `H_begin()` o `H_end()` sobre cualquier tipo de iterador (todos los iteradores a vértices incluyen esta función para evitar indirecciones), y
2. invocando `Graph::H_begin(v)` o `Graph::H_end(v)`, que toma un `vertex_iterator v` como parámetro.

Análogamente, cambiando `H_begin` y `H_end` por `N_begin` y `N_end`, respectivamente, podemos obtener instancias de la clase `neighbor_iterator`. El Código 4.2 muestra cómo usar estos iteradores.

```
1  //Imprime N(v)
2  void print_high_neighborhood(const Graph<int>& G, 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_Ns_of_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 neighbor_iterator
15             for(auto z = w.H_begin(); z != w.H_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 `vertex_iterator` y usando un `deg_iterator`.

4.1.3. Interfaz del tipo Graph

Recordemos que el TAD Grafo es un tipo paramétrico, ya que debe almacenar información arbitraria en los vértices. En consecuencia, la clase `Graph` es un *template* que depende de un parámetro `Elem`. Hay que tener en cuenta que las funciones que agregan

vértices copian la información y, por lo tanto, las instancias de `Elem` deben ser copiables. El resto de la interfaz publica se obtiene de traducir las operaciones del TAD GRAFO a C++ usando los iteradores definidos en las secciones anteriores. El Código 4.3 incluye las funciones principales de la interfaz de `Graph`.

```
1  //add_vertex(*this, e) (Sección 2.2)
2  vertex_iterator add_vertex(const Elem& e);
3
4  //remove_vertex(*this, v) (Sección 2.3)
5  void remove_vertex(vertex_iterator v);
6
7  //add_edge(*this, v, w) (Sección 2.4)
8  void add_edge(vertex_iterator v, vertex_iterator w);
9
10 //remove_edge(*this, v, w) (Sección 2.5)
11 void remove_edge(vertex_iterator v, vertex_iterator w);
12
13 //add_vertex(*this, e, N) y add_vertex(*this, e, N=[begin,end)) (Sección 2.6)
14 vertex_iterator add_vertex(const Elem& e, initializer_list<vertex_iterator> N)
15 template<typename iter>
16 vertex_iterator add_vertex(const Elem& e, iter begin, iter end);
17
18 //vertex_iterator(*this) (Sección 2.7)
19 vertex_iterator begin() const;
20 vertex_iterator end() const;
21
22 //neighbor_iterator(*this, v) (Sección 2.8)
23 neighbor_iterator N_begin(vertex_iterator v) const;
24 neighbor_iterator N_end(vertex_iterator v) const;
25
26 //H_neighbor_iterator(*this, v) (Sección 2.9)
27 deg_iterator H_begin(vertex_iterator v) const;
28 deg_iterator H_end(vertex_iterator v) const;
```

Código 4.3: Interfaz de la clase `Graph`.

4.2. Estructura de representación en C++

En esta sección describimos la estructura interna de la clase `Graph` de C++ que emula la estructura de un h-grafo vista en la Sección 3. La implementación de estas estruc-

turas se encuentra dividida en tres clases, cada una de las cuales se encuentra en su correspondiente archivo fuente:

- `Graph.h`: contiene la clase `Graph` que implementa la estructura principal del h-grafo.
- `Vertex.h`: contiene la clase `Vertex` que contiene la descripción de lo que es un vértice.
- `Neighbor.h`: contiene la clase `Neighbor` que representa lo que es un vecino de un vértice.

Las secciones siguientes describen las tres clases principales del h-grafo.

4.2.1. `Graph`

Recordemos que el clase `Graph` de C++ es un *template* que permite asociar información de algún tipo a cada vértice. Como vimos en la Sección 3, la representación de un h-grafo es sencilla en términos conceptuales, ya que simplemente es una lista de vértices. Esta lista está implementada con el tipo `std::list` y, dado que los vértices se implementan usando la clase `Vertex` (ver Sección 4.2.2), el tipo completo del miembro `vertices` es `std::list<Vertex>`. La clase `Graph` define también lo que es un `Graph::vertex_iterator`, que es una clase interna de `Graph`.

A pesar de la aparente simpleza, los tipos auxiliares (i.e., `Vertex` y `Neighbor`) deben estar al tanto de la estructura en la que se almacenan los vértices. En efecto, cada instancia de `Neighbor` mantiene un iterador (`neighbor_pointer`) que apunta a una posición de `vertices`. Análogamente, cada instancia de `Vertex` necesita acceso al tipo paramétrico `Elem` de `Graph` para saber qué tiene que guardar como información. Para comunicar a las otras clases que `vertices` es de tipo `std::list`, hay al menos tres opciones. La primera es copiar en cada clase los tipos no parametricos (e.g., el tipo de `vertices`). Esto no es muy flexible ya que, si la estructura cambiara[4], tendríamos que arreglar todos las clases auxiliares. Una segunda opción es que las clases auxiliares sean *templates* a fin de indicar los distintos parámetros que necesitan. Una tercer opción más flexible es que las clases auxiliares sean *templates* que dependan de la clase `Graph`. De esta forma, podemos guardar todos los rasgos (*traits*) en la clase `Graph`. Así, además de definir la estructura de `vertices`, la clase `Graph` contiene la definición de todos los renombres que son compartidos por las clases auxiliares (e.g., define el tipo donde se almacenan los vecindarios dentro de un vértice). En resumen, como veremos más adelante, las clases `Vertices` y `Neighbor` son *templates* que se instancian como `Vertex<Graph>` y `Neighbor<Graph>`. Más aún, la clase `Graph` es **`friend`** de estas dos clases a fin de que puedan manipular la estructura interna.

[4]Por ejemplo para representar grafos estáticos conviene usar un `std::vector`.

Volviendo un poco para atrás, la estructura de representación de `Graph` está dada por `vertices` cuyo tipo, habíamos dicho, es `std::list<Vertex>`. El inconveniente es que, de acuerdo al párrafo anterior, el tipo `Vertex` está incompleto, ya que `Vertex` es un *template*. Estrictamente hablando:

- `vertices` es de tipo `std::list<Vertex<Graph<Elem>>>`

Esto no es un inconveniente para C++, ya que el tipo está bien definido más allá de que estemos definiendo la clase `Graph`. Queremos remarcar que, más allá del patrón de implementación, la estructura no deja de ser una lista de vértices.

El Código 4.6 muestra la estructura de la clase `Graph`.

```
1  template<class Elem> class Graph
2  {
3      using elem_type = Elem;
4      //Vertex y Neighbor dependen de los rasgos de Graph<Elem>
5      friend class Vertex<Graph>;
6      friend class Neighbor<Graph>;
7      using Vertex = Vertex<Graph>;
8      using Neighbor = Neighbor<Graph>;
9
10     //Rasgos de Graph<Elem> que define todas las estructuras
11     using Vertices = std::list<Vertex>;
12     using degNeighborhood = std::list<Neighbor>;
13     using Neighborhood = std::list<degNeighborhood>;
14     using neighbor_iterator = typename Vertex::neighbor_iterator;
15     using deg_iterator = typename Vertex::deg_iterator;
16
17     //La estructura de representacion es una lista de vertices
18     Vertices vertices;
19  };
```

Código 4.4: Estructura del tipo Grafo en C++.

4.2.2. Vertex (Vertex.h)

De acuerdo a lo visto en la Sección 4.2.1, `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 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 grados.
12     using Neighborhood = typename Graph::Neighborhood;
13
14     elem_type elem;  //elemento
15     size_t degree;  //grado
16     Neighborhood neighborhood;  //vecindario
17 }
```

Código 4.5: 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 3.3.1

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 4.1 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    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) que lo contiene.
18            */
19           SelfPtr self_pointer;
20
21           /**
22            * Es un puntero directo a la lista de L(w) que contiene a v. (Solo tiene en cuenta los l
23            * cuando v esta en high_neighborhood de w, dejamos low_neighborhood.end())
24            */
25           ListPtr list_pointer;
26   }
```

Código 4.6: 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