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, 15 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 **7**
- 2.1. `create_graph()` . . . 8
- 2.2. `insert_vertex(G, info)` . . . 8
- 2.3. `remove_vertex(G, v)` . . . 9
- 2.4. `add_edge(G, v, w)` . . . 10
- 2.5. `remove_edge(G, v, w)` . . . 11
- 2.6. `add_vertex(G, info, N)` . . . 11
- 2.7. `vertex_iterator(G)` . . . 12
- 2.8. `neighbor_iterator(v)` . . . 13
- 2.9. `H_neighbor_iterator(v)` . . . 14
- 2.10. `degree(v)` . . . 14

3. La estructura h-grafo **17**
- 3.1. `G.insert_vertex(N = [], info = None)` . . . 20
- 3.2. `G.remove_vertex(v)` . . . 20
- 3.3. `G.insert_edge(v, w)` . . . 21
- 3.4. `G.remove_edge(v, w)` . . . 22

4. Implementación en C++ **27**
- 4.1. Interfaz de C++ . . . 27
  - 4.1.1. `Graph::vertex_iterator` . . . 27
  - 4.1.2. `Vertex::deg_iterator` y `Vertex::neighbor_iterator` . . . 28
  - 4.1.3. Interfaz de la clase `Graph` . . . 31
- 4.2. Estructura de representación en C++ . . . 32
  - 4.2.1. Clase `Graph` . . . 32
  - 4.2.2. Clase `Vertex` . . . 34
  - 4.2.3. Clase `Neighbor` . . . 34

5. Resultados de la experimentación **37**

6. Conclusiones y posibilidades de mejora **41**

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 bajo esta noción precisamente cuando su arboricidad es baja.

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. aplicar la estructura para resolver algún problema típico de grafos, y usarlo para
4. verificar que la eficiencia teórica coincide con la de nuestra implementación para algunos grafos.

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 6 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*; notar que $d(v) = |N(v)|$.

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 $G = H - v$.

Decimos que un grafo G es *k-degenerado* cuando todo subgrafo de G tiene al menos un vértice de grado menor o igual a k. La *degeneración* de G es el mínimo valor $d(G)$ tal que G es $d(G)$-degenerado. Para un vértice v, el *vecindario alto* de v es el conjunto $H(v)$ que contiene a todos los vecinos de grado al menos $d(v)$ lo llamamos el *vecindario alto*. Al valor $h(v) = |H(v)|$ lo llamamos el *h-index* de v. El *h-index* de G es el máximo $h(G)$ tal que hay al menos $h(G)$ vertices con $h(G)$-index mayor o igual a $h(G)$.

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 $E(G)$.

El siguiente resultado relaciona los parámetros de cantidad de aristas, grado, h-index, degeneración y arboricidad para un grafo G. Este resultado es importante, ya que guía las estrategias de los distintos algoritmos a fin de reducir la complejidad temporal para grafos ralos.

Teorema 1 (e.g. [**?**, **?**])**.** *Si G es un grafo con al menos dos vértices y δ es el mínimo entre los grados de los vértices, entonces*

$$\frac{\delta}{2} < \frac{|E|}{|V| - 1} \leq \alpha(G) \leq d(G) \leq 2\alpha(G) \leq 2h(G) \leq 2\sqrt{2|E|}$$

Recordemos que consideramos que grafo es ralo cuando lo densidad $|E(H)|/|V(H)|$ es pequeña para todo subgrafo H de G. Como mencionamos en la sección anterior, G es ralo cuando su arboricidad es baja. La veracidad de este hecho surge del siguiente teorema de Nash-Williams.

Teorema 2 ([**?**])**.** *Si G es un grafo con al menos dos vértices, entonces*

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

Los últimos teoremas de esta sección reflejan una estrategia válida para trabajar con grafos ralos, que es la de revisar únicamente los vecindarios altos de algunos vértices, ignorando sus otros vecinos.

Teorema 3 ([?]). *Sea $e_1, \ldots, e_m$ un ordenamiento de las aristas de un grafo G con $e_i = v_i w_i$. Si $h_i(v)$ es el valor de $h(v)$ en el subgrafo generador de G que contiene las aristas $e_1, \ldots, e_i$ (con $1 \leq i \leq m$), entonces*

$$\sum_{i=1}^{m} (h_i(v_i) + h_i(w_i)) \leq 4\alpha(G)m.$$

Intuitivamente, el teorema anterior indica que si se agregan las aristas a G en el orden indicado pagando $O(h_i(v) + h_i(w))$ tiempo para la arista $v_i w_i$, entonces el tiempo total del algoritmo es $O(m\alpha(G))$. Esta cota es importante para los algoritmos dinámicos donde se agregan aristas. En la versión "estática", donde el grafo es conocido en forma completa de antemano, el costo es el mismo.

Teorema 4 ([?]). *Si G es un grafo, entonces* $\sum_{vw \in E(G)} (h(v) + h(w)) \leq 4\alpha(G)m.$

Estos dos resultados aplican, de manera análoga, cuando uno considera vértices en lugar de aristas.

Teorema 5 ([?]). *Sea $v_1, \ldots, v_n$ un ordenamiento de los vértices de un grafo G. Si $h_i(v)$ es el valor de $h(v)$ en el subgrafo de G inducido por $v_1, \ldots, v_i$ (con $1 \leq i \leq n$), entonces*

$$\sum_{i=1}^{n} \sum_{w \in N(v_i)} h_i(w) \leq 4\alpha(G)m.$$

Teorema 6 ([?]). *Si G es un grafo, entonces,* $\sum_{v \in V(G)} \sum_{w \in N(v)} h(w) \leq 4\alpha(G)m.$

Reemplazar la figura por una pagina encuadrada en la que se muestre un ejemplo completo, incluyendo los resultados de los teoremas. Ademas, reemplazar el grafo por el que se usa en la Figura 3.1 (es mas facil reemplazar este grafo que esa figura)

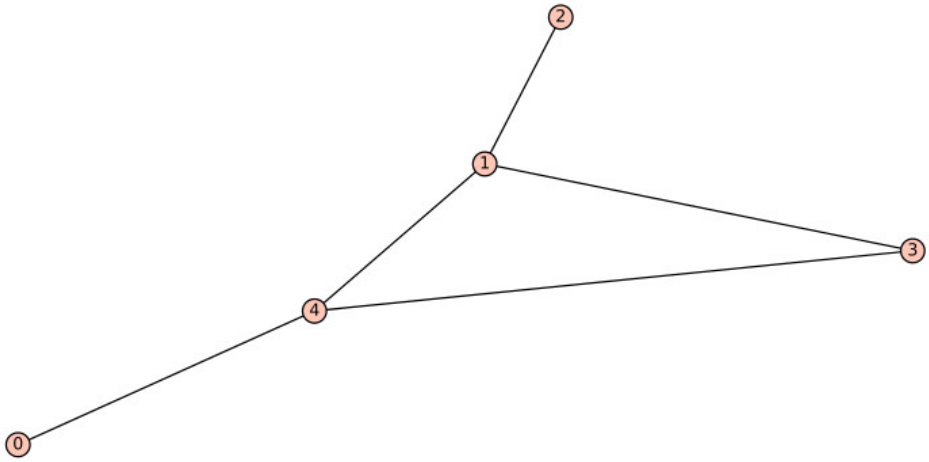

Figura 1.1: Grafo G con $V(G) = \{v_0, v_1, v_2, v_3, v_4\}$ y $E(G) = \{v_0v_4, v_1v_2, v_1v_3, v_1v_4, v_3v_4\}$. El vértice v_1 tiene grado 3 y su vecindario es $\{v_2, v_3, v_4\}$. La secuencia v_0, v_4, v_1, v_2 es un camino, mientras que v_4, v_1, v_3, v_4 es un ciclo. El grafo H con $V(H) = V(G)$ y $E(H) = \{v_0v_4, v_1v_3, v_1v_2\}\}$ 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`.

Por último, remarcamos que la interfaz que presentamos hace un uso extensivo de iteradores a fin de proveer una estructura de acceso y manipulación de los vértices y aristas del grafo. En términos simples, un iterador `it` presenta una secuencia S de elementos de un conjunto soporte C a través de dos operaciones básicas. La primer operación permite acceder al *primer* elemento de S, la segunda permite modificar `it`, "descartando" el primer elemento de S (vale remarcar que el conjunto soporte C no cambia). Cuando uno trabaja con tipos dinámicos, es posible que el conjunto soporte C cambie, lo que afecta a la secuencia S y altera el comportamiento de `it`. En general, puede ocurrir una de dos opciones:

- que `it` quede en un estado *inválido*, en cuyo caso cualquier uso de `it` resultará en un error de ejecución; esto ocurre usualmente cuando se elimina de C el primer elemento de S,

- que `it` quede *permutado*, en cuyo caso `it` es válido, pero que la secuencia recorrida S cambió más allá de lo esperable por la operación aplicada (e.g., obviamente que si se elimina de C un elemento de S, el mismo no será recorrido); esta situación ocurre cuando se cambian propiedades de los objetos de C que alteran el orden fundamental de recorrido de `it` (e.g., si se recorren los vértices de acuerdo a su grado, el agregado de aristas puede modificar el orden del recorrido).

Obviamente, el uso de iteradores inválidos está prohibido. En cambio, uno puede usar un iteradores después que queda permutado. Sin embargo, se desaconseja el uso de iteradores en recorridos que los permutan, ya que podría ocurrir que algunos elementos no se procesen o que algunos elementos se procesen más de una vez. En cualquier caso, la validez dependerá del uso que uno quiera darle al iterador.

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(1) (más la copia de la información).

```
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` para imprimir "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.

Asimismo, cualquier `neighbor_iterator` o `H_neighbor_iterator` al que aún le reste recorrer algún vecino de v quedará permutado.

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: $O(d(v)h(G))$; $O(|E(G)|\alpha(G))$ si se aplica a todos los vértices.

```
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` para imprimir 'b'. 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. Como resultado de esta operación, queda permutado cualquier `neighbor_iterator` o `H_neighbor_iterator` correspondiente a vecinos de v o w.

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: $O(h(v) + h(w))$

```
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` para imprimir 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. Asimismo, queda permutado cualquier `neighbor_iterator` o `H_neighbor_iterator` correspondiente a vecinos de 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: $O(h(v) + h(w))$

```
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` para imprimir 3. 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 al 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. Como resultado de esta operación queda permutado cualquier `neighbor_iterator` o `H_neighbor_iterator` que recorra vértices de `N`.

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: $O(d(v)h(G))$; $O(|E(G)|\alpha(G))$ si se aplica a todos los vértices.

```
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` para imprimir "12".

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 para manipular el vértice 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: $O(1)$; el recorrido de cada vértice cuesta $O(1)$ también.

```
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` para imprimir 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 dejar al iterador permutado.

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: $O(1)$; el recorrido de cada vecino cuesta $O(1)$ también.

```
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` para imprimir 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 "alto" 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 dejar al iterador permutado.

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: $O(1)$; el recorrido de cada vecino cuesta $O(1)$ también.

```
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` para imprimir 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` para imprimir 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 o h-index es baja. Esto se ve reflejado en las complejidades temporales esperadas, dado que las mismas dependen de la cantidad de aristas, el h-index 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, cómo está 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 1.1. 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 `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 inválida iteradores, aunque obviamente los permuta.

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 3.4) 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'` sin invalidar los iteradores que lo apuntan, entonces `v'->self_pointer` también 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.

Incluir ejemplo de una ejecución empezando con la estructura de la Figura 3.1).

3.4. `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 3.3). 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); G.update_neighborhood(w)
4    //Fase 2
5    G.update_neighbors(v); G.update_neighbors(w)
6    //Fase 3 (insercion fisica)
7    if v->deg > w->deg: swap(v, w)
8    if v->deg < w->deg: ubicar N := w->low(v->deg) recorriendo w->low
9    else: N := w->high
10   v' := N.prepend({neighbor_pointer=v})
11   w' := v->high.prepend({neighbor_pointer=w})
12   v'->self_pointer := N.first(); w'->self_pointer := v->high.first()
13   v'->list_pointer := N; w'->list_pointer := v->high
14   v->deg += 1; w->deg += 1
15
16 G.update_neighborhood(v):
17   N := []
18   for z' in v->high if v->deg = z'->neighbor_pointer->deg:
19     N.transfer(z')
20     //Obs: z'->self_pointer apunta a v dentro de z.high
21     z'->self_pointer->list_pointer := N
22   if not N.empty(): v->low.append(N)
23
24 G.update_neighbors(v):
25   for z' in v->high if v->deg < z'->neighbor_pointer->deg:
26     //Obs: z'->neighbor_pointer representa a z en G.vertices
27     z = z'->neighbor_pointer
28     //     z'->self_pointer representa a v en z->low,
29     v' := z'->self_pointer
30     //     y z'->list_pointer representa z->low(d(v))
31     N_d := z'->list_pointer
32
33     if v->deg + 1 < z->deg y z->low.next(N_d) != z->low(v->deg+1):
34       N := z->low.append([], pos = N_d)
35     if v->deg + 1 = z->deg:
36       N := z->high
37     N.transfer(v')
38     if 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); G.update_neighbors_delete(w)
10     //Deshacer Fase 2
11     G.update_neighborhood_delete(v); G.update_neighborhood_delete(w)
12
13 G.update_neighbors_delete(v):
14   for z' en v->high:
15     z := z'->neighbor_pointer
16     v' := z'->self_pointer
17     N := z'->list_pointer
18     P := if v->deg + 1 = z->deg then prev(N) else z->low.last()
19     if P != z->low(d(v)): P := z->low.append([], P)
20     P->transfer(z'->self_pointer)
21     if N = [] and v->deg+1 < z->deg: z->low.erase(N)
22     z'->list_pointer := P
23
24 G.update_neighborhood_delete(v):
25   if v->low.last() != v->low(v->deg): return
26   for z' in v->low.last():
27     v->high.transfer(z')
28     //z'->self_pointer representa a v en N(z).
29     z->self_pointer->list_pointer := v->high
30   v->low.erase(v->low.last())
```

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,

[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.

```
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.

al aplicar las función `H_neighbor_iterator`[3] (Sección 2.9). Análogamente, la clase `Vertex` ↪`::neighbor_iterator`, o simplemente `neighbor_iterator`, define los iteradores 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.

[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.

- `deg_iterator H_begin() const` y `deg_iterator H_end() const`: `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 de la clase `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 estructuras 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. Clase `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

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

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.

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 4.4 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. Clase `Vertex`

`Vertex` es la clase que define la estructura de un vértices del h-grafo. De acuerdo a lo que discutimos en la Sección 4.2.1, `Vertex` es *template* con un parámetro que corresponde a clase `Graph` de la que forma parte. Vale remarcar que el tipo `Vertex` se usa únicamente para la definición de clase `Graph`, quien no exhibe la interfaz interna de `Vertex`. En decir, la existencia de `Vertex` es invisible al usuario de `Graph`. Por conveniencia, entonces, la estructura interna de `Vertex` es pública, ya que sólo define una estructura que igualmente es conocida por `Graph`.

La clase `Vertex` posee la información asociada con el vértice v (campo `elem`), el grado $d(v)$ del vértice (campo `degree`) y el vecindario $N(v)$ (campo `neighborhood`). De acuerdo a lo discutido en la Sección 3, el vecindario en el h-grafo se almacena en una estructura especial con dos partes principales. Por un lado, hay una lista de listas $\mathcal{N}$, que contiene cada una de las listas no vacías de $N(v,0),\ldots,N(v,d(v)-1)$, en ese orden. Por otro lado, se guarda una lista conteniendo los vértices de $H(v)$. Recordemos que $N(v,i)$ contiene los vecinos de grado i, mientras que $H(v)$ contiene los vecinos con grado al menos $d(v)$. Para la representación en C++, por comodidad, elegimos almacenar $H(v)$ como si fuera una última lista de $\mathcal{N}$. En otras palabras, `neighborhood` es una lista de listas cuya última lista siempre representa a $H(v)$.

Recordemos que la clase `Graph` concentra las definiciones de todos los tipos que forman la estructura. Esto incluye qué estructura se usa para almacenar el vecindario $N(v)$ (tipo `Graph::Neighborhood`), qué estructura se usa para representar cada lista $N(v,i)$ y $H(v)$ dentro del vecindario (tipo `Graph::degNeighborhood`), y qué estructura se usa para representar un vecino del vecindario, i.e. el objeto que define el `self_pointer`, el `list_pointer` y el `neighbor_pointer` (tipo `Graph::Neighbor`). La clase `Graph` incluso define el tipo de vértice que se usa en `Graph::Vertex`. Por comodidad, renombramos estos tipos dentro de la estructura de `Vertex`. El Código 4.5 muestra la estructura de `Vertex`.

4.2.3. Clase `Neighbor`

Las instancias de esta clase se usan para representar a las adyacencias en el grafo. Supongamos que `G` representa a un grafo G con dos vértices v y w que están representados por los objetos `v` y `w` en `G.vertices`. Cuando v y w son adyacentes, una de las listas dentro de `v.neighborhood` contiene un objeto `w'` que representa a w y, análogamente, una de las listas dentro de `w.neighborhood` contiene un objeto `v'` que representa a v. Estos objetos `v'` y `w'` son instancias de la clase `Neighbor`.

Como vimos en la Capítulo 3, la instancia `w'` debe mantener tres iteradores que se usan como punteros restringidos:

```
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 del vecindario, las listas internas y los vertices.
8      using Neighborhood = typename Graph::Neighborhood;
9      using degNeighborhood = typename Graph::degNeighborhood;
10     using Vertices = typename Graph::Vertices;
11
12     //Estructura del vertice
13     elem_type elem;  //elemento
14     size_t degree;  //grado
15     Neighborhood neighborhood;  //vecindario
16 };
```

Código 4.5: Estructura del tipo Vertex en C++.

- `w'.neighbor`: iterador a `G.vertices` que apunta a `w`,
- `w'.list_pointer`: iterador a `w.neighborhood` apuntando a la lista `N` que contiene a `v'`,
- `w'.self_pointer`: iterador a `N` que apunta a `v'`.

Al igual que discutimos para la clase `Vertex`, la clase `Neighbor` también es un *template* con un parámetro que debe corresponder con la clase `Graph` que lo utiliza. Por comodidad, renombramos los tipos definidos dentro de la clase `Graph` dentro de `Neighbor`. El Código 4.6 muestra la estructura interna de `Neighbor`.

```
1  template<class Graph> 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      //Estructura para la arista vw dentro de v.neighborhood.
9      //Para w' en v.neighborhood mantenemos:
10     //- Puntero a w en G.vertices,
11     NeighborPtr neighbor;
12     //- Puntero a la lista que tiene a v',
13     ListPtr list_pointer;
14     //- Puntero a v'
15     SelfPtr self_pointer;
16 };
```

Código 4.6: Neighbor.h

5 Resultados de la experimentación

"tiempo vs complejidad"

En está sección describiremos la experimentación realizada a fin de demostrar la eficiencia del h-graph. La experimentación consiste en el tiempo que lleva a la implementación crear un grafo y encontrar los triángulos en él.

Para realizar la prueba implementamos un algoritmo sencillo que busca y arma una lista con todos los triángulos en un grafo dado. Por un lado tenemos una clase ejemplo Triangle que es la representación de como es un triángulo, esto lo usamos para que más adelante podamos distinguir un triángulo que ya fue encontrado. El método principal recibe un grafo de vértices del tipo marcables. Estos objetos que representan vértices entienden los métodos marcar(bool) que indica si el objeto está marcado o no y marcado() que retorna el valor de la marca. Se empieza recorriendo uno a uno los vértices v del grafo y por cada uno marca a sus vecinos w, una vez todos ellos están marcados se los vuelve a recorrer para ir procesando a su vez cada uno de sus vecinos z de grado mayor o igual con el fin de saber si z está marcado o no. Si está marcado significa que ese vértice es vecino tanto de v como de w por lo tanto (y si no es un triángulo que ya se encontró) se crea un Triangle con estos tres vértices y se lo agrega a la lista de resultado. Finalmente se desmarcan todos los vértices para poder iniciar el ciclo con el siguiente vértice del grafo. Una vez se procesan todos los vértices se obtiene la lista de triángulos.

Para crear los h-graph se generaron grafos aleatorios utilizando la librería Networkx de Python. En un archivo plano se guardaron los datos de los grafos generados (cantidad de aristas, aristas y nodos). Esos datos son consumidos por nuestra implementación para generar el h-graph.

A continuación mostraremos los gráficas del tiempo de ejecución en crear el grafo y encontrar los triángulos en función a la cantidad de aristas. El eje y refleja el tiempo en milisegundos, el eje x representa la cantidad de aristas. La experimentación fue realizada con diferentes tipos de grafos que tienen desde 10 a 505 nodos.

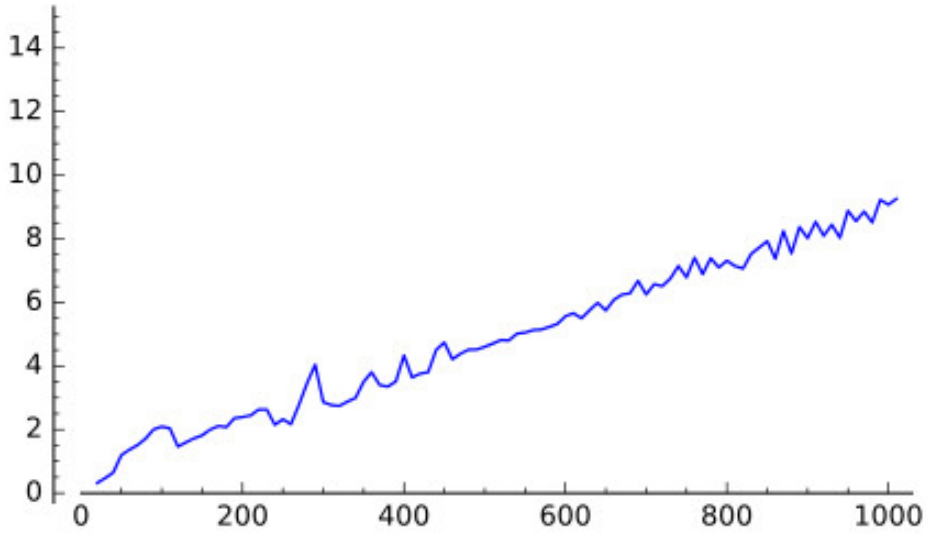

Gráfico del tiempo de ejecución de 100 grafos connected_watts_strogatz_graph

Figura 5.1: El tiempo máximo de 9.25799 ms se da en grafo con 505 nodos y 1010 aristas. El degeneracy * E de 3030 y el $E/\sqrt{E}$ de 31. El tiempo mínimo de 0.319994 ms se da en el grafo de 10 nodos y 20 aristas. El degeneracy * E es de 60 y el $E/\sqrt{E}$ de 4

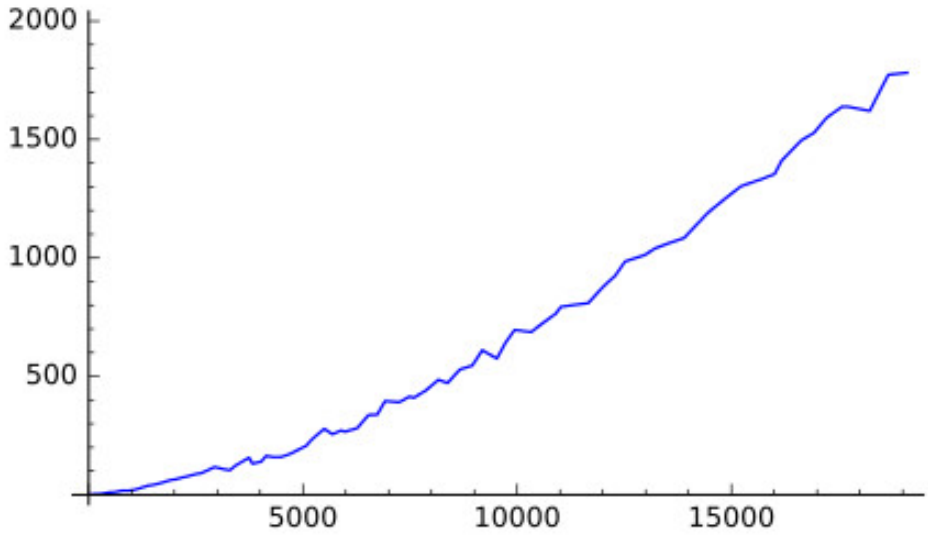

Gráfico del tiempo de ejecución de 100 grafos erdos_renyi_graph

Figura 5.2: El tiempo máximo de 1781.94 ms se da en grafo con 505 nodos y 19104 aristas. El degeneracy * E de 1146240 y el $E/\sqrt{E}$ de 138. El tiempo mínimo de 0.133815 ms se da en grafo de 10 nodos y 7 aristas. El degeneracy * E es de 7 y el $E/\sqrt{E}$ de 2

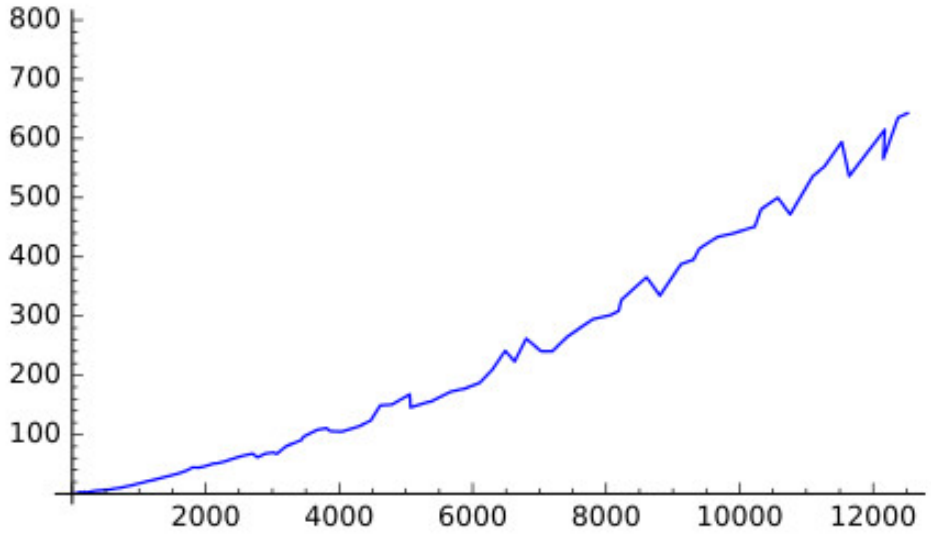

Gráfico del tiempo de ejecución de 100 grafos gnp_random_graph

Figura 5.3: El tiempo máximo de 642.788 ms se da en grafo con 505 nodos y 12518 aristas. El degeneracy * E de 475684 y el $E/\sqrt{E}$ de 111. El tiempo mínimo de 0.082068 ms se da en grafo de 10 nodos y 2 aristas. El degeneracy * E es de 2 y el $E/\sqrt{E}$ de 1

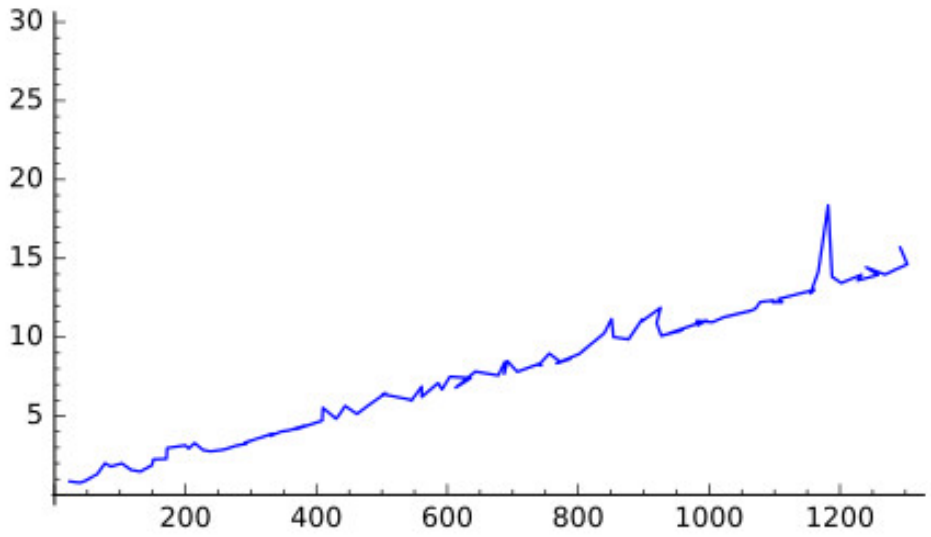

Gráfico del tiempo de ejecución de 100 newman_watts_strogatz_graph

Figura 5.4: El tiempo máximo de 18.3681 ms se da en el grafo con 455 nodos y 1182 aristas. El degeneracy*E de 4728 y el $E/\sqrt{E}$ de 34. El tiempo mínimo de 0.769438 ms se da en el grafo de 15 nodos y 39 aristas. El degeneracy*E de 156 y el $E/\sqrt{E}de6$

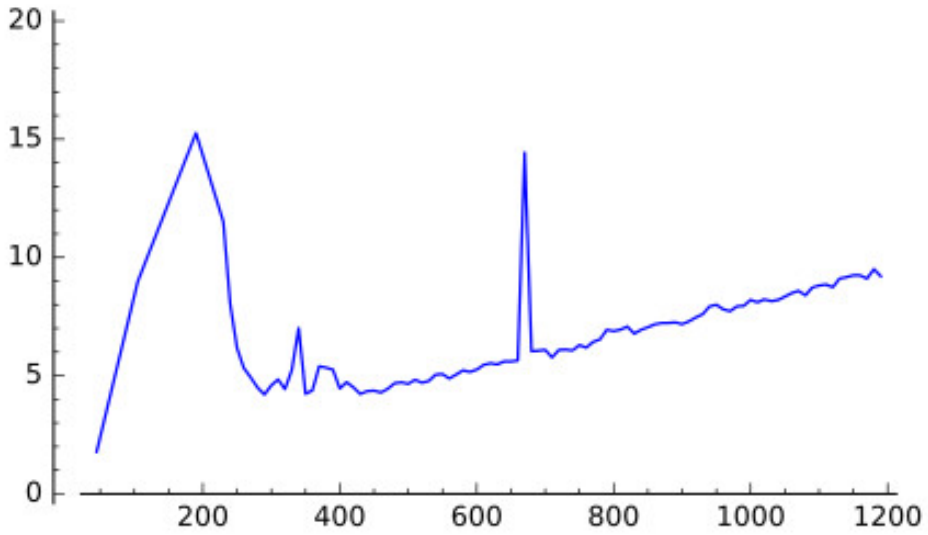

Figura 5.5: El tiempo máximo de 15.2598 ms se da en el grafo con 20 nodos y 190 aristas. El degeneracy * E de 3610 y el $E/\sqrt{E}$ de 13. El tiempo mínimo de 1.78106 ms se da en grafo de 10 nodos y 45 aristas. El degeneracy * E es de 405 y el $E/\sqrt{E}$ de 6

6 Conclusiones y posibilidades de mejora