NetworkX-OOC: Una implementación Out Of Core para NetworkX

Giampieri, Leonardo
Hojman, Joaquin
Vazquez, Francisco

Abstract

En el presente trabajo analizamos las limitaciones de NetworkX para el análisis de grafos de gran tamaño. Identificamos que la biblioteca enfrenta serias restricciones debido a su dependencia exclusiva de la memoria RAM. Para abordar este problema, desarrollamos estructuras Out of Core que almacenan datos en disco y emulan las estructuras en memoria de Python, modificamos los algoritmos de NetworkX para integrar estas estructuras y creamos una clase de Grafo Out of Core capaz de manejar grafos grandes almacenando sus partes en disco. Esta solución permite cargar y analizar grafos que excedan la memoria principal de una máquina, utilizando entre el 7% y el 33% de la memoria que consume la versión original de NetworkX para grafos de más de 100 mil nodos. Si bien el tiempo de ejecución aumenta significativamente debido a los múltiples accesos en disco, un aspecto que podría mejorarse en futuras investigaciones, los grafos en disco ocupan solo un 10% de lo que utilizan en memoria con la versión nativa. Nuestra implementación es compatible hacia atrás, permitiendo a los usuarios de NetworkX aprovechar las nuevas capacidades sin cambios significativos en su código.

1 Introducción

En el contexto del análisis de grafos, NetworkX [1] ha emergido como una de las bibliotecas más utilizadas y versátiles para Python. Su popularidad radica en su facilidad de uso y su amplia gama de algoritmos y herramientas que permiten analizar y comprender las relaciones y estructuras en los datos interconectados. Sin embargo, a medida que las aplicaciones del análisis de grafos se expanden para abordar conjuntos de datos más grandes y complejos, NetworkX se encuentra con ciertas limitaciones relacionadas con la carga y el manejo de grafos de gran tamaño.

Proponemos la creación de estructuras Out Of Core, que con cambios mínimos a NetworkX permiten manejar grafos de gran tamaño, evitando las limitaciones de memoria. Se modificará la implementación interna de los algoritmos existentes, reemplazando las estructuras en memoria por implementaciones de las mismas que funcionen Out of Core, de modo que los usuarios puedan aprovechar las nuevas capacidades sin tener que realizar cambios significativos en su código.

2 Diseño

Para lograr que los algoritmos de NetworkX funcionen con grafos que no pueden ser almacenados en su totalidad en memoria es necesario reemplazar las estructuras que carguen datos en memoria por una versión de las mismas que guarde los datos en disco, es decir, estructuras Out of Core (OOC).

2.1 Out Of Core Dict

La principal estructura de la cual depende nuestra implementación es un OutOfCoreDict ($OOCDict$), que proporciona una interfaz similar a un diccionario de Python (MutableMapping), y que utiliza LevelDB [2] como implementación detrás de escena.

LevelDB es una key-value store que acepta claves y valores en bytes arbitrarios, y que internamente utiliza árboles LSM [3], optimizados para escrituras en bulk, lo cual permite una carga eficiente de grafos grandes. Esto nos interesa debido al objetivo mismo del trabajo, poder cargar cantidades de datos que no entran en memoria.

Nuestra implementación utiliza plyvel [4] como binding para LevelDB. En la sección 5 exploramos alternativas al uso de LevelDB.

Algorithm 1 OutOfCoreDict class

```
class OutOfCoreDict(MutableMapping):
    def __setitem__(self, key, value):
        self._wb[key] = value
        if len(self._wb) > 4000:
            self._flush_write_batch()

    def __getitem__(self, key):
        value = self._wb.get(key)
        if value is not None:
            return value
        data = self._inner.get(key, fill_cache=False)
        if data is None:
            raise KeyError(key)
        return data

    def __init__(self, dir_=DB_DIR) -> None:
        self._temp = tempfile.TemporaryDirectory(dir=dir_)
        self._inner = plyvel.DB(
            f"{self._temp.name}",
            create_if_missing=True,
        )
        self._wb = {}
```

Utilizamos Temp Directory [5] (*self._temp*) ya que LevelDB necesita un lugar para almacenar los datos que se guardan en el diccionario, y no estamos interesados en que estos persistan luego de que el proceso Python termine: esto busca replicar el comportamiento del diccionario nativo de Python.

Temp Directory usa por defecto "/tmp", que en algunos sistemas puede llegar a estar montado sobre tmpfs [6], file system que utiliza memoria y swap. Para los experimentos utilizamos un directorio (referido por DB_DIR) montado en disco para estresar la implementación y lograr una comparación razonable contra la nativa. Se permite cambiar este directorio mediante una variable de entorno. En la sección 5 exploramos posibilidades que nos permitirian persistir las estructuras de datos aun luego de terminar su uso.

Uno de los principales problemas que encontramos con la primera versión del *OOCDict* fue el tiempo que tomaba la carga del mismo. Lo que se observó fue que muchas de las llamadas al write de plyvel para guardar los elementos del *OOCDict* ejecutaban llamadas a la syscall write con pocos datos por llamada, de sólo algunas decenas de bytes. Para aumentar el throughput se decidió introducir un buffer en memoria de tamaño fijo a modo de caché de escritura, y recién cuando este se llena se hace una escritura de los mismos en batch. Esto resultó en una reducción significativa del tiempo de carga, debido a que se llaman muchas menos veces a la syscall write, y cada llamada se realiza con una mayor cantidad de datos a escribir.

La clase *OOCDict* implementa todos los métodos de un diccionario nativo de Python, pudiendo reemplazar a éste en los algoritmos donde haga falta.

Debido a que plyvel únicamente guarda bytes, debimos definir una forma de serialización/deserialización de la información almacenada. Se probó inicialmente con pickle [7], debido a la gran variedad de tipos de datos con los que permite trabajar, incluyendo estructuras complejas. Sin embargo, por el overhead de performance, se decidió utilizarlo de manera auxiliar, como se ilustra en la sección 2.5.2.

Se determinó entonces utilizar el módulo struct [8] para la serialización y deserialización de datos. Estas modificaciones nos permitieron ganar velocidad, pero por otro lado se perdió la posibilidad de que los datos almacenados sean de tipos arbitrarios.

Se implementaron entonces diversos adaptadores del *OOCDict*, cada uno especializado en serializar un tipo de dato específico y guardarlo en plyvel. La utilidad de estas estructuras es que abstraen la conversión de los datos desde y hacia bytes. Estas forman la base de varias estructuras que son utilizadas por los algoritmos OOC. El detalle de estas estructuras puede verse en la tabla 1 y cómo se relacionan con *OOCDict* en la figura 1.

Estructura	Tipo de dato de clave	Tipo de dato de valor
IntDict	Int	Int
IntFloatDict	Int	Float
EdgesDict	Int, Tupla de Ints	Int, Tupla de Ints

Table 1: Adaptadores del *OOCDict*

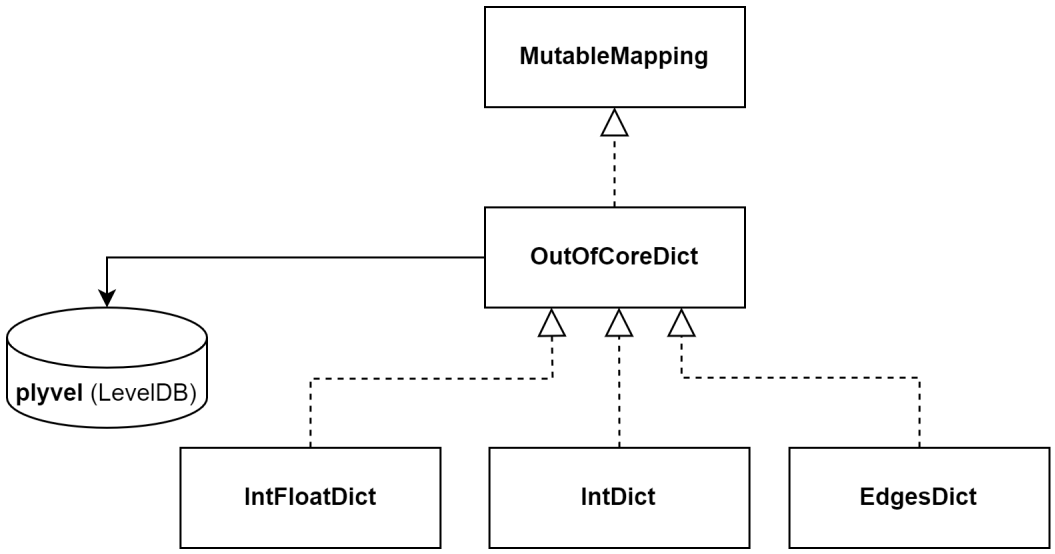

Figure 1: Diagrama de clases que muestra las relaciones de *OOCDict*, sus adaptadores y LevelDB

2.2 Lazy List

Fue necesario definir una estructura que implementara un subset de las operaciones típicas de una lista nativa de Python, llamada LazyList. Esta estructura tiene como atributo una ruta a un archivo temporal en disco que representa la lista, y es en ese archivo donde se guardan los elementos de la misma.

Además, tiene un atributo que indica el tipo de dato que se guarda en la lista, ya que los elementos que se guarden en la misma serán codificados como bytes usando esta información. Por ejemplo, si en la LazyList se guardan integers, la estructura sabe que cada 4 bytes escritos se tiene un dato de tipo integer.

De esta forma, utilizando mmap [9] y sabiendo los bytes que ocupa cada dato, se pueden manejar eficientemente estos archivos que representan las listas, aplicando las operaciones clásicas de la misma. De hecho, las operaciones de longitud, obtener un elemento, o guardar un elemento dado un índice o al final de la lista son todas O(1).

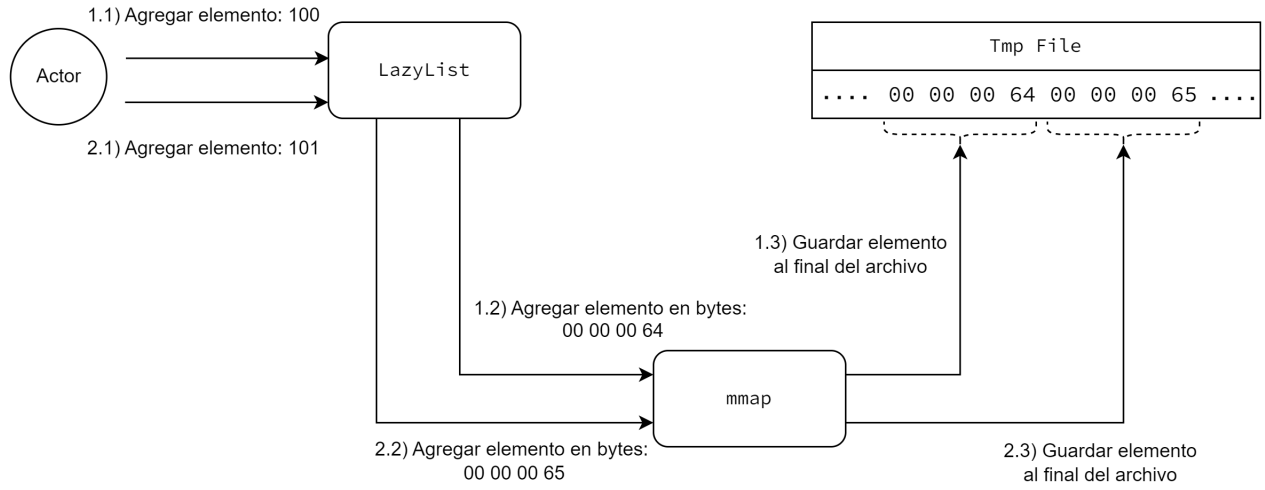

Figure 2: Diagrama de comunicación que ilustra el guardado de los elementos 100 y 101 al final de una LazyList

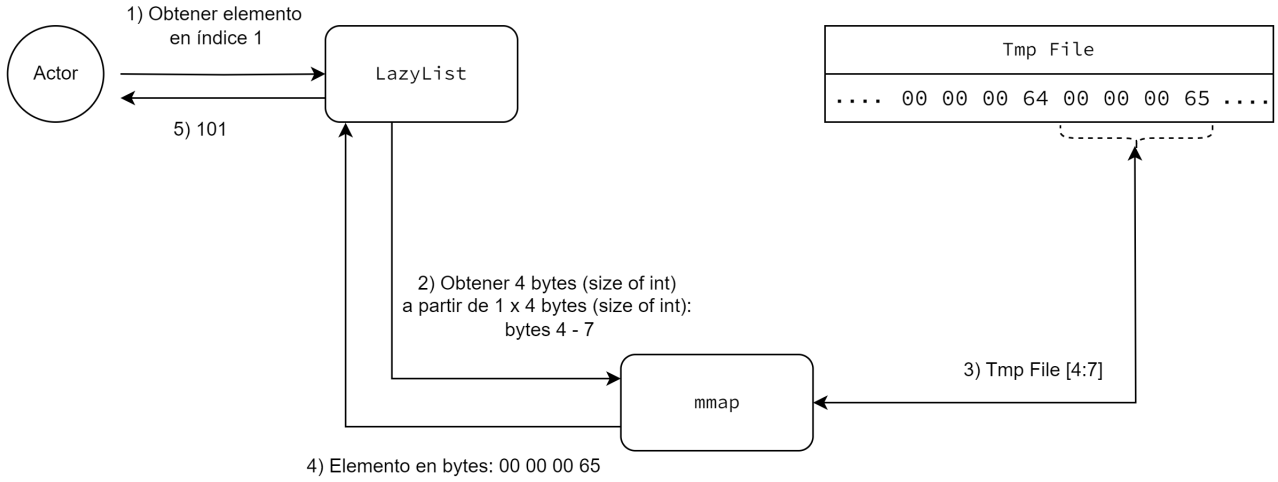

Figure 3: Diagrama de comunicación que ilustra la obtención del dato ubicado en la posición 1 en una LazyList

Por otra parte, nuestra implementación cuenta con una lista llamada OutOfCoreList (*OOCList*), basada en un OutOfCoreDict (ver sección 2.1). Cada par clave-valor del *OOCDict* interno representa para *OOCList* un par índice-elemento. La ventaja de esta implementación es la posibilidad de soportar listas "sparse" [10]. Actualmente, *OOCList* es la estructura que reemplaza a la lista nativa en nuestra implementación, exploramos alternativas para esto en la sección 5.

2.3 Out of Core Dict of Lists

En los algoritmos donde es necesario utilizar de forma OOC un diccionario de listas la estructura debe ser algo más compleja que un OutOfCoreDict. Un *OOCDict* acepta como valores una tira de bytes, por lo que podríamos codificar una lista de Python de esta manera y guardarla como valor en el diccionario. El problema es que *OOCDict* carga completamente en memoria cada valor accedido: para datos chicos como un int o float no habría problema, pero para una lista la misma podría no entrar en memoria. Por otro lado, una *OOCList*, donde la lista no está cargada en memoria, no puede ser codificada como bytes e insertada en un *OOCDict* por la complejidad de la misma.

Lo que se resolvió fue tener una estructura especializada, llamada OutOfCoreDictOfLists (*OOCDictOfLists*): internamente usa un *OOCDict*, donde la clave del mismo es un elemento, y el valor es una ruta a un archivo temporal en disco que representa la lista, la cual se maneja mediante una LazyList (ver sección 2.2).

El algoritmo 2 provee un ejemplo en código, mientras que la figura 4 ejemplifica el uso de la estructura mediante un diagrama de componentes.

Algorithm 2 Ejemplo de uso de OutOfCoreDictOfLists

```
import networkx as nx

g = nx.OutOfCoreGraph()
dol = g.int_dict_of_lists()

dol[1] = [22, 15]
dol[1].append(7)

dol[2] = []
dol[2].append(500)

dol[3] = [400, 401, 402]
dol[3].append(403)
dol[3].append(404)

print(dol[1][0])  # 22
print(dol[2][0])  # 500
print(dol[3][4])  # 404
```

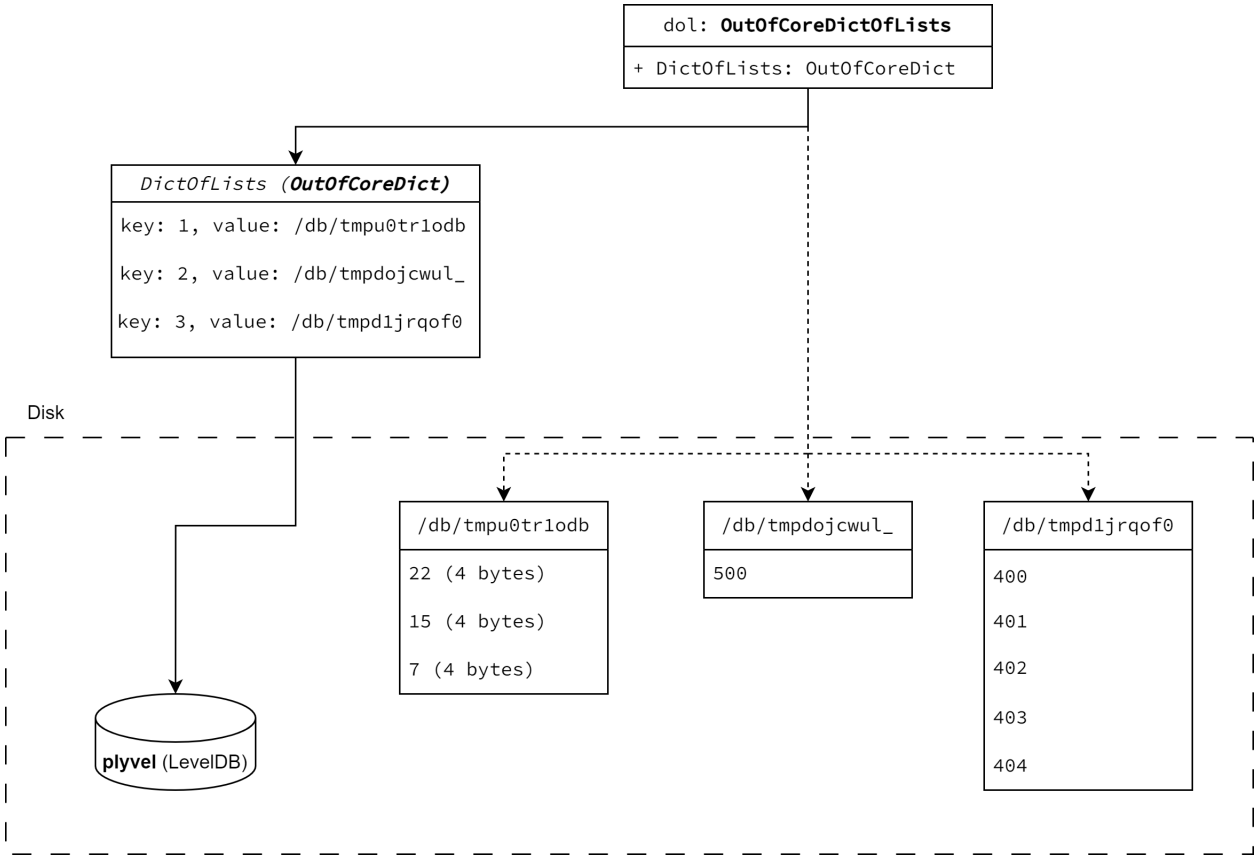

Figure 4: Diagrama de componentes del *OOCDictOfLists* para el caso que se ejemplifica en el algoritmo 2

2.4 Out of Core Set

Varias de las estructuras detalladas hasta aquí utilizan *OOCDict*. Utilizar un *OOCDict* para implementar un set resultaría en una estructura que ocupa espacio en guardar claves que no serán utilizadas. Es por esto que para desarrollar el OutOfCoreSet (*OOCSet*) se decidió utilizar una implementación específica; un BitMap de CRoaring [11], utilizando Pyroaring [12] como bindings de Python. Este optimiza el uso de memoria mediante algoritmos de compresión [13].

2.5 Out of Core Graph

Un Graph de NetworkX tiene listas de adyacencia y de nodos implementadas mediante diccionarios nativos de Python.

Un OutOfCoreGraph (*OOCGraph*) implementa la mismas funcionalidades que un Graph nativo, reemplazando sus diccionarios en memoria internos por *OOCDict* (en disco).

2.5.1 Lazy adjacency list y Lazy node list

Para la primera versión del *OOCGraph*, se reemplazaron los diccionarios de python internos que usa el Graph de NetworkX para representar las listas de nodos y adyacencia por instancias de *OOCDict*. En esta implementación, recuperar información implicaba deserializar estas estructuras por completo. Esto no solo era lento, ya que el mismo proceso de serialización/deserialización se llevaba a cabo por cada acceso al grafo, sino que también resultaba en spikes en el uso de memoria, propios de que la estructura que se estaba deserializando era un diccionario tan grande como aristas y/o nodos hubiera en el grafo.

Por lo tanto, se decidió implementar una solución que utiliza estructuras lazy, que únicamente operan sobre la información en disco en el momento necesario. Estas estructuras son una Lazy Node List, que almacena como clave un nodo y como valor los atributos del mismo, y una Lazy Adjacency List, que almacena como clave un par ordenado de nodos, y como valor los atributos.

Ambas implementan la interfaz MutableMapping y utilizan *OOCDict* para su persistencia. Al solicitar los atributos de un nodo o una arista, Lazy Node List y Lazy Adjacency List instancian un LazyNode o LazyEdge, los cuales funcionan como un proxy, como se esquematiza en la figura 5: cuando se pide a un LazyNode el atributo del nodo, este se busca en el *OOCDict* interno de la Lazy Node List. De igual manera, cuando a un LazyEdge se le solicita el atributo de una arista, se busca en el *OOCDict* interno de la Lazy Adjacency List mediante el par de nodos que conforman la arista, y se devuelve el valor como un diccionario de atributos.

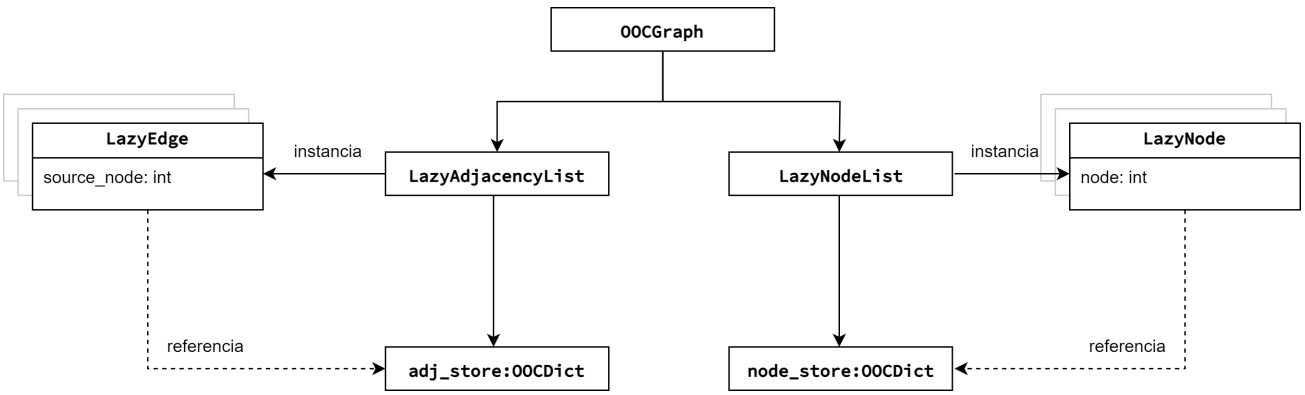

Figure 5: Diagrama de objetos que muestra el *OOCGraph* y sus estructuras internas

Un LazyEdge se instancia únicamente con un nodo fuente, y luego tiene la capacidad de devolver los atributos que tenga ese nodo fuente con el resto de los nodos con los que forme una arista, como se puede observar en la figura 6.

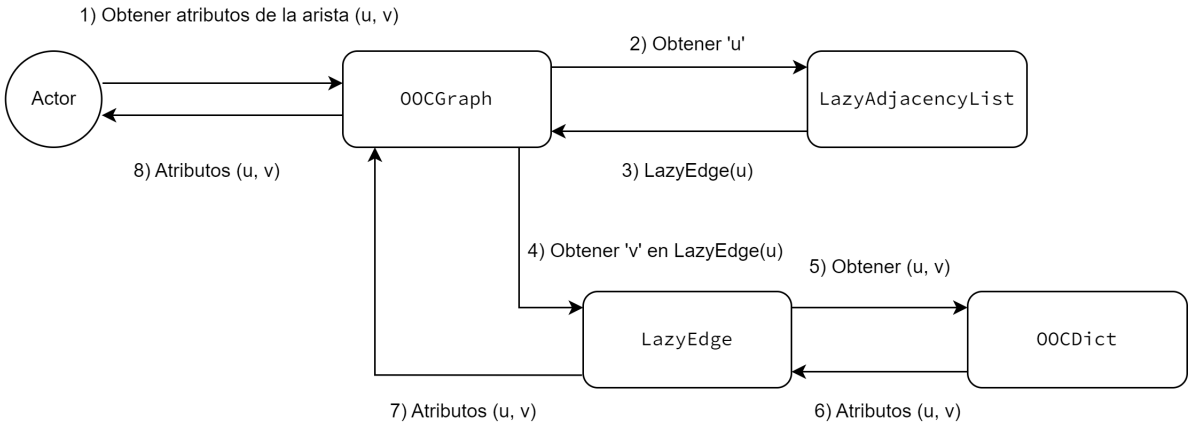

Figure 6: Diagrama de comunicación que ilustra como se obtienen atributos de un LazyEdge

2.5.2 Serialización/Deserialización de nodos y aristas

Como se expuso en la sección 2.1, pickle presenta un overhead de performance al momento de serializar datos, y es por eso que se decidió utilizar el módulo struct. En particular, se utiliza para serializar/deserializar los nodos y aristas del grafo en sus estructuras internas, permitiendo guardar únicamente datos del tipo entero, debido a que serializar este tipo de dato es más performante que serializar estructuras genéricas.

Lo expuesto implica una diferencia con el grafo de NetworkX nativo, que la única restricción que requiere para almacenar nodos y aristas es que estos sean hasheables. Exploramos algunas alternativas para esto en la sección 5.

Por otro lado, se mantuvo el uso de pickle para serializar/deserializar atributos de los nodos y aristas del grafo, ya que ciertos algoritmos guardan en estos atributos información necesaria de los nodos y aristas en formato de diccionario de Python con claves y valores genéricos.

2.6 Algoritmos

Los algoritmos de NetworkX no solo interactúan con el grafo sino que además usan estructuras de datos secundarias como diccionarios, listas o sets. Cómo las mismas pueden crecer proporcionalmente al tamaño del grafo, estas deben ser reemplazadas por estructuras OOC.

Se hizo necesario modificar la implementación de los algoritmos cambiando sus estructuras secundarias por su versión OOC, sin embargo, con esta solución se acopla la implementación nativa con las estructuras OOC, impidiendo ejecutar los algoritmos enteramente en memoria. Respetando DRY [14], también se deseaba evitar duplicar las implementaciones, una para grafos en memoria y otra OOC.

La decisión tomada fue que el mismo grafo le informará al algoritmo que estructura utilizar, es decir, el grafo ofrece factory methods para crear estas estructuras adicionales, que para el grafo nativo serán estructuras en memoria y para el *OOCGraph* serán estructuras OOC.

Tomando el ejemplo de la función auxiliar de Bellman-Ford, $\_inner\_bellman\_ford$, donde antes se instanciaban las estructuras directamente en memoria ahora se instancian obteniéndolas del factory method del grafo:

Algorithm 3 *_inner_bellman_ford*, en NetworkX (izquierda) y en NetworkX-OOC (derecha)

```
def _inner_bellman_ford(G):
    pred = {}
    rec = {}
    count = {}
    q = deque()
    in_q = set()
    ...
```

```
def _inner_bellman_ford(G):
    pred = G.int_dict_of_lists()
    rec = G.int_tuple_dict_of_edges()
    count = G.int_dict()
    q = G.int_deque()
    in_q = G.set_()
    ...
```

Luego, dentro de la clase grafo:

Algorithm 4 *NXGraph* (izquierda) y OutOfCoreGraph (derecha)

```
class Graph:

    def set_(self, *args):
        return set(*args)

    def int_list(self, *args):
        return list(*args)

    def int_dict(self, *args):
        return dict(*args)
    ...
```

```
class OutOfCoreGraph(Graph):

    def set_(self, *args):
        return OutOfCoreSet(*args)

    def int_list(self, *args):
        return OutOfCoreList(*args)

    def int_dict(self, *args):
        return IntDict(*args)
    ...
```

Con este diseño en mente se modificaron un total de 130 algoritmos de NetworkX para que funcionen de forma OOC. La evidencia de que la biblioteca fue efectivamente modificada en forma compatible hacia atrás es que los tests originales de los algoritmos de NetworkX funcionan en su versión nativa y en la versión OOC. Para conocer la totalidad de los mismos revise la sección "APÉNDICE A: Algoritmos implementados" o la lista referenciada en el readme del proyecto, en nuestro repositorio [15].

3 Experimentos

3.1 Grafos utilizados

Para el análisis de resultados utilizamos 5 grafos: 2 basados en datasets reales de redes sociales existentes y 3 generados artificialmente.

Los 3 grafos artificiales son de distinto tamaño y se generaron utilizando el modelo de Erdős–Rényi y NetworkX, con probabilidad de agregar una arista de

$$\frac{2 * Log(N)}{N} \tag{1}$$

siendo N la cantidad de nodos del grafo. Esto nos garantiza una única componente conexa [16], precondición requerida por algunos de los algoritmos de NetworkX analizados en el presente paper.

De esta forma los 5 grafos tienen estructuras similares, como se ve en la tabla 2.

Grafo	Detalles	Source
1	10.000 nodes / 92.412 edges	Erdős–Rényi model [17]
2	100.000 nodes / 1.150.875 edges	Erdős–Rényi model [18]
3	1.000.000 nodes / 13.819.815 edges	Erdős–Rényi model [19]
4	1.632.803 nodes / 30.622.564 edges	Pokec social network [20]
5	4.847.571 nodes / 68.993.773 edges	LiveJournal social network [21]

Table 2: Grafos utilizados para experimentos y análisis de resultados

3.1.1 Toma de métricas y herramientas

Se midió el consumo de memoria, la cantidad acumulada de operaciones input/output al momento de la medición del programa bajo test leyendo de /proc, y el uso de swap. Se midió el uso de disco de la carpeta donde se almacenan los archivos de LevelDB. Todas las métricas se tomaron a intervalos de sampleo de 10ms.

Sabiendo que cuando se usan estructuras OOC no debería haber uso de swap, este se midió a nivel global del sistema. El objetivo fue corroborar que efectivamente la implementación OOC no estaba usando más memoria de la esperada. Observamos que nunca se llegó al límite de la memoria disponible en el sistema, y que el comportamiento del swap fue inconsistente, lo cual sugiere que el swap medido es parte del ruido del sistema y no está relacionado con los algoritmos que analizamos.

3.2 Carga del grafo

3.2.1 Análisis temporal

La tabla 3 compara los tiempos de carga para los diferentes grafos, cargandolos en la versión nativa de NetworkX y en la versión OOC, medido en segundos:

Grafo	Nodos	*NXGraph* (s)	± (s)	*OOCGraph* (s)	± (s)	Ratio (OOC/NX)
1	10.000	0.16	0.01	0.82	0.06	5.1
2	100.000	2.67	0.14	16.37	0.49	6.1
3	1.000.000	38.37	2.25	297.94	4.73	7.7
4	1.632.803	101.42	4.93	646.61	16.60	6.4
5	4.847.571	190.78	29.56	1515.81	143.55	7.9

Table 3: Tiempos de cargas para los diferentes grafos. Se tomaron alrededor de 30 mediciones para cada caso del *NXGraph* y 20 mediciones para cada caso del *OOCGraph*.

Como era de esperarse, los tiempos de carga aumentan cuando el grafo se carga de forma OOC debido a las escrituras en disco: entre unas 5 y 8 veces aproximadamente más lento.

3.2.2 Análisis espacial

La tabla 4 compara el uso de memoria al cargar los diferentes grafos, en la versión nativa de NetworkX y en la versión OOC, en megabytes:

Grafo	Nodos	*NXGraph* (Mb)	± (Mb)	*OOCGraph* (Mb)	± (Mb)	Ratio (OOC/NX)
1	10.000	51.26	0.19	38.74	0.22	0.75
2	100.000	289.10	0.15	52.33	1.47	0.18
3	1.000.000	3022.98	0.10	84.33	2.27	0.03
4	1.632.803	4878.13	0.09	108.91	3.14	0.02
5	4.847.571	9743.20	2.54	161.19	3.57	0.01

Table 4: Uso de memoria en la carga de los diferentes grafos. Se tomaron 7 mediciones para cada caso.

Podemos apreciar que para el *OOCGraph* el crecimiento del uso de memoria es menor que para el grafo de NetworkX: se utiliza un 75% de la memoria original para grafos chicos, y luego entre 1% y 18% de la memoria original conforme los grafos aumentan de tamaño.

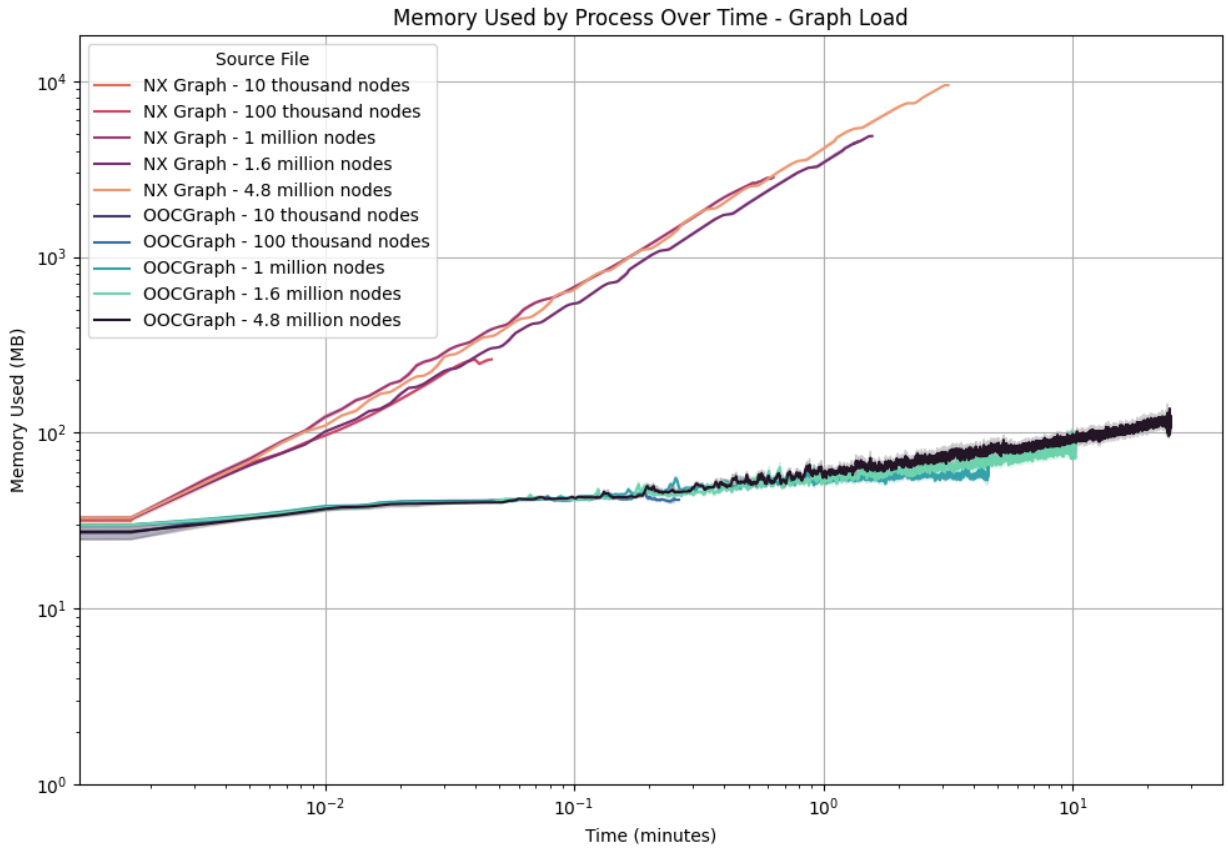

Figure 7: Uso de memoria durante la carga de los grafos mencionados en la tabla 2, con escala logarítmica en ambos ejes. El sombreado alrededor de las lineas corresponde al intervalo de confianza.

En el gráfico 7 podemos observar como el uso de memoria crece exponencialmente en el caso del grafo nativo de NetworkX, debido a que toda la información del mismo se persiste en memoria. Mientras que para *OOCGraph* se aprecia como el uso de memoria se mantiene relativamente constante, ya que las estructuras OOC guardan los datos en disco. El uso de memoria no es nulo en los *OOCGraph* dado que las estructuras internas hacen uso de caching para optimizar las operaciones de lectura y escritura.

3.2.3 Análisis espacial en disco

La tabla 5 muestra el uso de disco al cargar los diferentes grafos, en la versión nativa de NetworkX y en la versión OOC, en megabytes:

Grafo	**Nodos**	*OOCGraph* **(Mb)**	± **(Mb)**
1	10.000	2.64	0.07
2	100.000	30.33	0.75
3	1.000.000	294.53	0.59
4	1.632.803	522.85	8.89
5	4.847.571	988.89	14.12

Table 5: Uso de disco en la carga de los grafos OOC. Se tomaron 7 mediciones para cada caso.

En el caso del grafo de NetworkX no se utiliza el disco para cargar el grafo, cosa que sí sucede con *OOCGraph*. La diferencia entre el espacio en disco utilizado por *OOCGraph* y el uso de memoria para los mismos grafos utilizado en NetworkX nativo (ver tabla 4) puede explicarse teniendo en cuenta la información que se guarda en cada caso; mientras que NetworkX nativo utiliza estructuras de diccionarios anidadas, *OOCGraph* serializa y guarda la

representación entera de los nodos y aristas, lo cual es más liviano. Podemos observar que el espacio que se ocupa en disco para almacenar los grafos OOC es de apenas el aproximadamente 10% de lo que ocupan en memoria con la versión nativa.

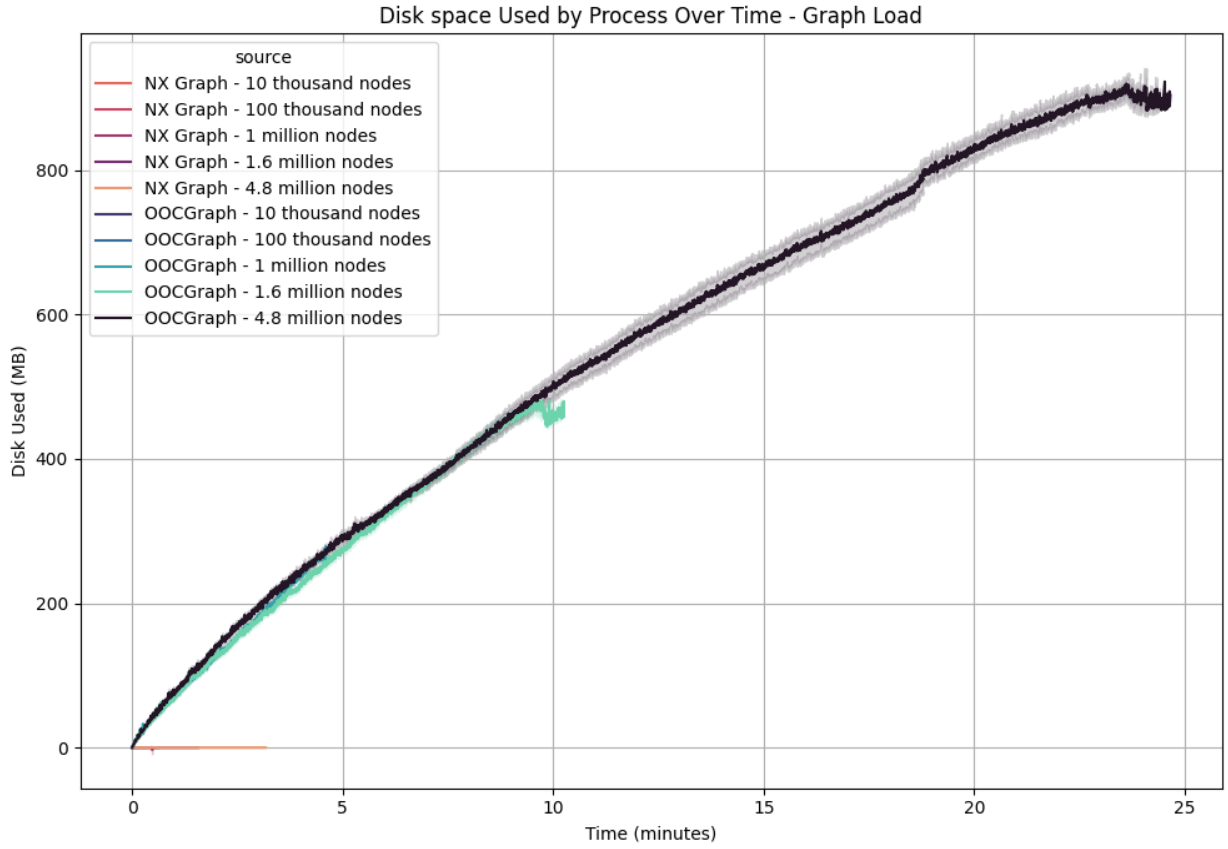

Figure 8: Uso de disco durante la carga de los grafos mencionados en la tabla 2, con escala lineal en ambos ejes. El sombreado alrededor de las lineas corresponde al intervalo de confianza.

En el gráfico 8 podemos observar el caso inverso al gráfico 7. El grafo nativo de NetworkX no usa disco, por lo que las líneas del mismo se mantienen constantes en 0. En el caso de *OOCGraph*, al utilizar estructuras que guardan los datos en disco, vemos como el uso del mismo se ve reflejado en el gráfico.

3.3 Análisis de algoritmos

Se presentan los resultados de los siguientes algoritmos:

- Predecessor (Unweighted)
- Degree Centrality
- Eigenvector Centrality
- Single Source Dijkstra
- Single Source Bellman-Ford

Los mismos fueron elegidos debido a que son de distintas familias y utilizan varias estructuras secundarias, como puede verse en la tabla 6.

Algoritmo	*OOCDict*	*OOCList*	*OOCSet*	*OOCDictOfLists*	*OOCDeque*
Predecessor	x	x		x	
Degree centrality	x				
Eigenvector centrality	x				
Single source dijkstra	x	x		x	
Single Source Bellman-Ford	x	x	x	x	x

Table 6: Estructuras de los algoritmos analizados

3.3.1 Predecessors

El algoritmo de Predecessors Unweighted de NetworkX [22] encuentra el camino más corto desde un nodo fuente a todos los demás nodos en un gráfico no pesado. Devuelve un diccionario donde cada clave es un nodo y su valor es el nodo predecesor en el camino más corto desde la fuente.

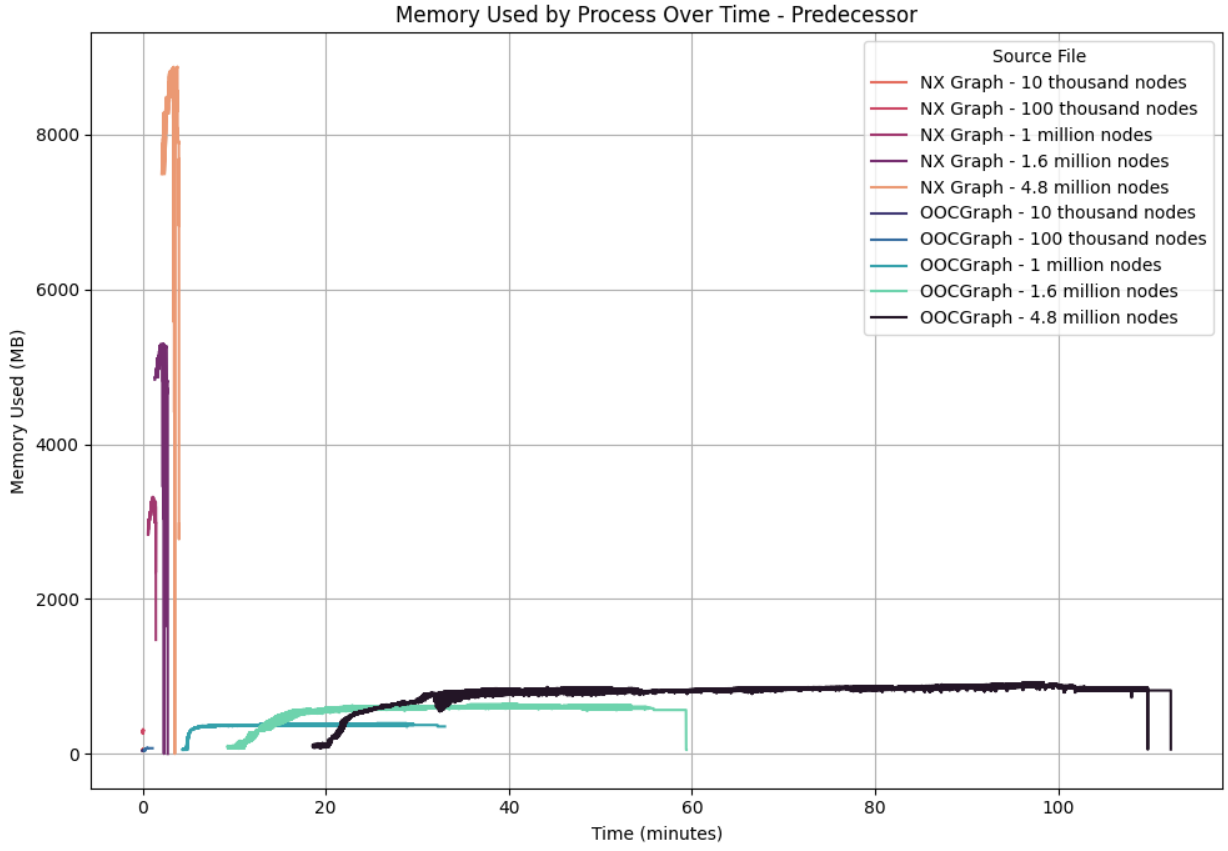

Figure 9: Memoria utilizada a traves del tiempo para el algoritmo de Predecessors, con escala lineal en ambos ejes

En la figura 9 podemos apreciar diversas ejecuciones de Predecessors, con grafos en su versión OOC y en la versión nativa de NetworkX. El gráfico muestra el uso de memoria a través de la ejecución del algoritmo. El algoritmo utiliza BFS para explorar los nodos y almacenar información de predecesores y niveles. La información se guarda en listas, diccionarios, y diccionarios de listas. Se observa el crecimiento de memoria en el caso de NetworkX nativo debido a que los datos para procesar el algoritmo se almacenan en memoria. En el caso de *OOCGraph*, el uso de memoria es mucho más eficiente porque las estructuras de datos no se cargan completamente en memoria, sino que únicamente lo hacen los datos que se necesitan en cada momento.

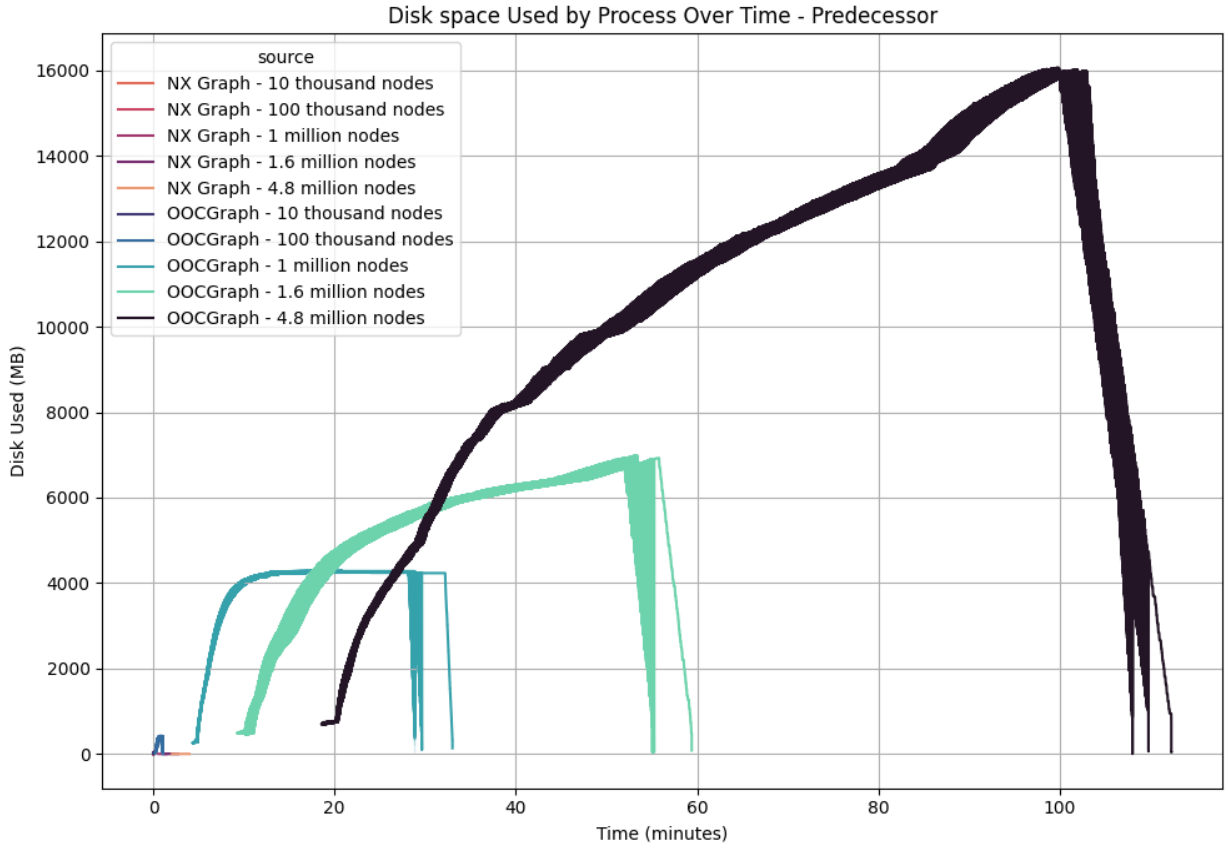

Figure 10: Espacio en disco utilizado a traves del tiempo para el algoritmo de Predecessors, con escala lineal en ambos ejes

En la figura 10 podemos ver el impacto en disco de utilizar estructuras OOC. Existe consumo de disco en el caso de los algoritmos con estructuras OOC, debido a que se almacena información en las listas, diccionarios, y diccionarios de listas OOC, mientras que los algoritmos nativos no utilizan disco.

El algoritmo carga todos los nodos como claves de un $OOCDictOfLists$ y como valor una lista con los predecesores entre cada nodo y la fuente. El tamaño de estas listas estará acotado por el diámetro del grafo [23]. Esto implica que el espacio en disco ocupado por estas listas depende del tamaño del grafo y de la estructura del mismo. Luego, si bien el algoritmo también utiliza como estructuras auxiliares $OOCList$ y $OOCDict$, la que realmente impacta en el uso de disco es el $OOCDictOfLists$.

En la tabla 7 se muestra el tiempo promedio de ejecución en segundos para varias ejecuciones del algoritmo de Predecessors en diversos grafos, tanto en su versión OOC como en su versión nativa de NetworkX, sin contar la carga del grafo. En la tabla 8 se presenta el uso promedio de memoria en megabytes para las mismas ejecuciones y versiones del algoritmo, también sin contar la carga del grafo.

Grafo	**Nodos**	$NXGraph$ **(s)**	**± (s)**	$OOCGraph$ **(s)**	**± (s)**	**Ratio (OOC/NX)**
1	10.000	0.0338	1.2039	3.1566	0.4544	93.4
2	100.000	1.2254	0.1052	47.0946	4.6136	38.4
3	1.000.000	42.5711	1.4445	1586.3832	139.4725	37.3
4	1.632.803	52.3224	7.6354	2832.3912	141.4864	54.1
5	4.847.571	71.8172	8.7653	5551.3701	117.6728	77.3

Table 7: Tiempos de ejecución de Predecessors para los diferentes grafos. Se tomaron 5 mediciones para cada caso del $NXGraph$ y 3 mediciones para cada caso del $OOCGraph$.

Grafo	Nodos	*NXGraph* (Mb)	± (Mb)	*OOCGraph* (Mb)	± (Mb)	Ratio (OOC/NX)
1	10.000	54.64	0.82	42.82	0.19	0.78
2	100.000	312.61	0.13	72.17	0.22	0.23
3	1.000.000	3317.03	0.27	377.54	14.03	0.11
4	1.632.803	5295.47	5.26	616.80	22.06	0.12
5	4.847.571	8871.40	0.17	894.62	22.75	0.10

Table 8: Uso de memoria en la ejecución de Predecessors para los diferentes grafos. Se tomaron 5 mediciones para cada caso del *NXGraph* y 3 mediciones para cada caso del *OOCGraph*.

Se observa un claro trade-off entre utilizar menos memoria cuando se utiliza el *OOCGraph* a cambio de que el tiempo de ejecución de los algoritmos se incremente notablemente. Con el grafo 3 se aprecia claramente que el grafo OOC utiliza un 10% de la memoria pero tarda aproximadamente 40 veces más en términos de tiempo. En la sección 5 exploramos algunos puntos de mejora posibles en cuanto a tiempo de ejecución.

3.3.2 Degree Centrality

El algoritmo de Degree Centrality en NetworkX [24] calcula la importancia de cada nodo en un grafo basándose en el grado [25] del mismo. Devuelve un diccionario donde cada clave es un nodo y su valor es la fracción de nodos a los que está conectado.

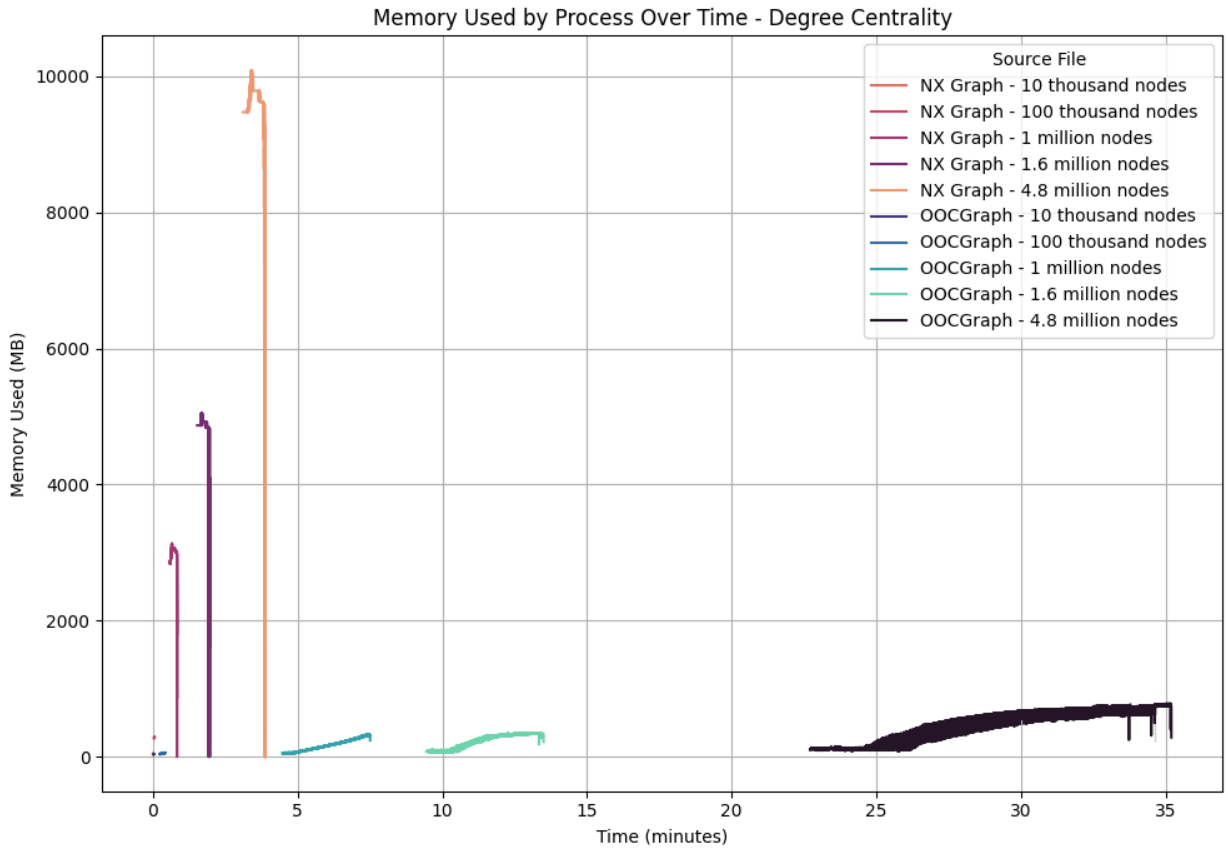

Figure 11: Memoria utilizada a traves del tiempo para el algoritmo de Degree Centrality, con escala lineal en ambos ejes

Degree Centrality tiene un tiempo de ejecución menor a otros algoritmos analizados, debido a que no tiene un procesamiento tan intensivo. En la figura 11 vemos que el caso del grafo con 4.8 millones de nodos en la versión nativa tarda menos de 5 segundos mientras que el equivalente OOC tarda unos 10 minutos.

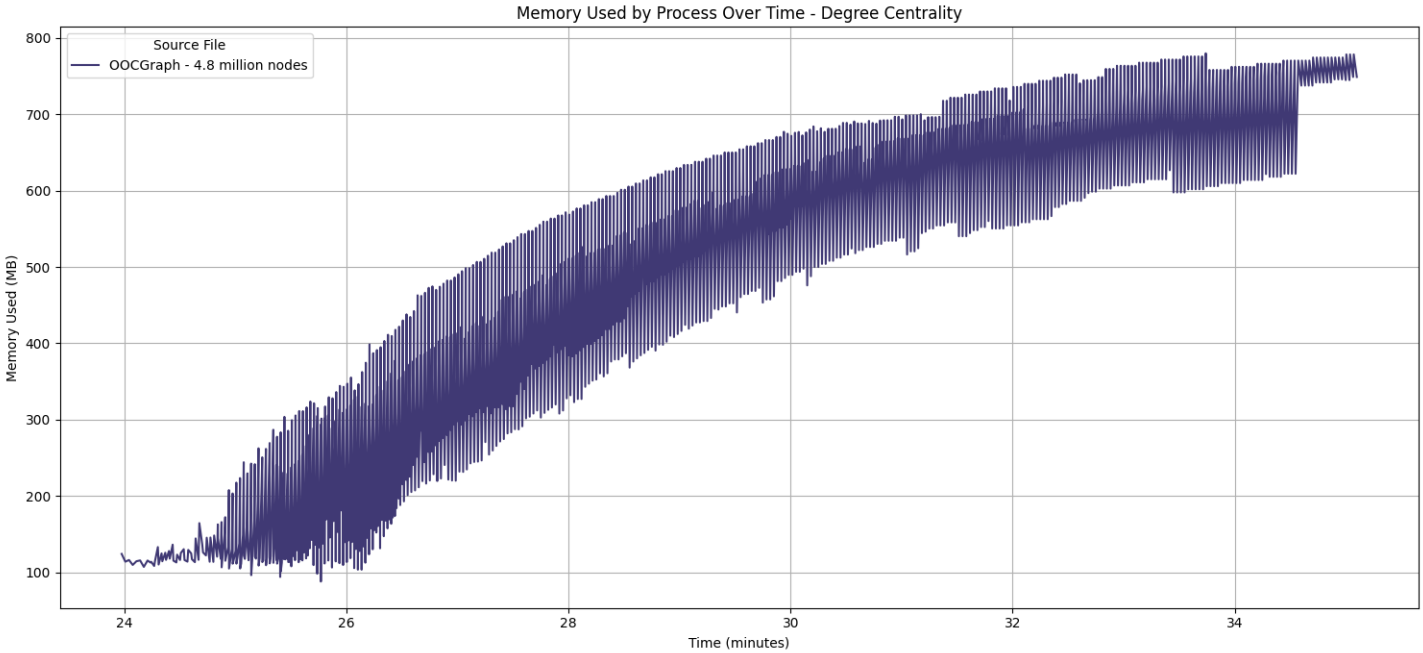

Figure 12: Memoria utilizada a traves del tiempo para el algoritmo de Degree Centrality, entre los minutos 24 y 35, para el grafo de 4.8 millones de nodos, con escala lineal en ambos ejes

Si observamos más en detalle el caso con 4.8 millones de nodos, en la figura 12, vemos que el uso de memoria tiene una tendencia creciente, y de manera periódica hay picos altos y bajos. Esto tiene una explicación doble; el algoritmo carga la centralidad de cada nodo en un *OOCDict* a medida que itera la lista de adyacencia del grafo. Para calcular esta centralidad se lee la lista de adyacencia del grafo, lectura que implica la carga de caches LRU de LevelDB, que ocupan espacio en memoria. A medida que las centralidades de cada nodo se van almacenando en el *OOCDict*, se llena el caché del mismo (ver sección 2.1).

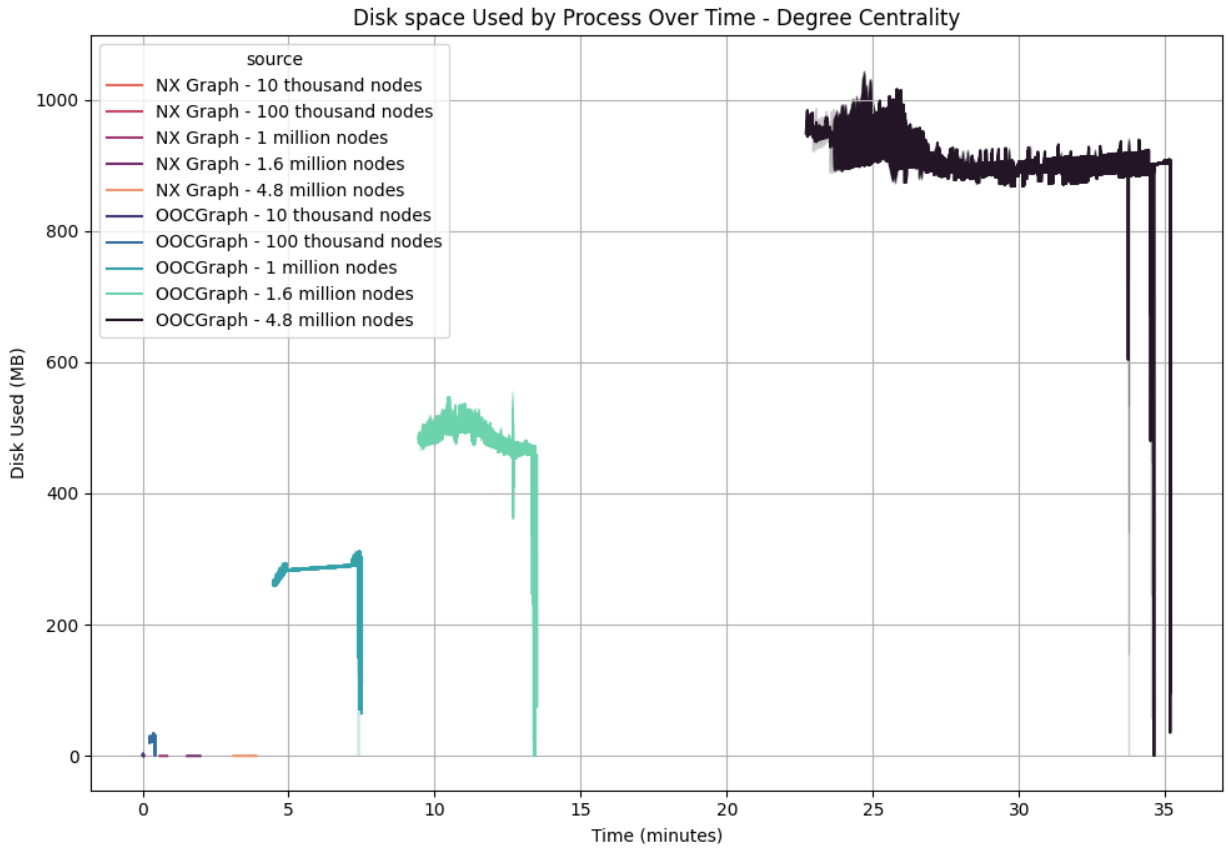

Figure 13: Espacio en disco utilizado a traves del tiempo para el algoritmo de Degree Centrality, con escala lineal en ambos ejes

En la tabla 9 se muestra el tiempo promedio de ejecución en segundos para varias ejecuciones del algoritmo de Degree Centrality en diversos grafos, tanto en su versión OOC como en su versión nativa de NetworkX, sin contar la carga del grafo. En la tabla 10 se presenta el uso promedio de memoria en megabytes para las mismas ejecuciones y versiones del algoritmo, también sin contar la carga del grafo.

Grafo	**Nodos**	*NXGraph* **(s)**	**± (s)**	*OOCGraph* **(s)**	**± (s)**	**Ratio (ooc/ Nx)**
1	10.000	0.0041	0.0001	0.6687	0.0180	163.1
2	100.000	0.0591	0.0011	10.3524	0.0372	175.2
3	1.000.000	0.8362	0.0027	157.5144	1.2258	188.4
4	1.632.803	1.3137	0.0073	188.9905	7.2324	143.9
5	4.847.571	3.7380	0.0609	576.0167	19.5377	154.1

Table 9: Tiempos de ejecución de Degree Centrality para los diferentes grafos. Se tomaron 5 mediciones para todos los casos.

Grafo	**Nodos**	*NXGraph* **(Mb)**	**± (Mb)**	*OOCGraph* **(Mb)**	**± (Mb)**	**Ratio (OOC/NX)**
1	10.000	52.04	0.19	39.37	0.30	0.75
2	100.000	301.97	0.20	60.17	0.57	0.20
3	1.000.000	3133.39	0.08	324.66	1.10	0.10
4	1.632.803	5067.17	12.79	349.02	6.84	0.07
5	4.847.571	10109.32	1.75	735.32	60.17	0.07

Table 10: Uso de memoria en la ejecución de Degree Centrality para los diferentes grafos. Se tomaron 5 mediciones para todos los casos.

Nuevamente observamos un trade-off entre utilizar menos memoria cuando se utiliza el *OOCGraph* a costa de que el tiempo de ejecución de los algoritmos se incremente notablemente.

3.3.3 Eigenvector Centrality

El algoritmo de Eigenvector Centrality en NetworkX [26] mide la influencia de un nodo en un grafo, teniendo en cuenta no sólo el número de aristas que tiene (como en Degree Centrality), sino también la importancia de los nodos a los que está conectado. Devuelve un diccionario donde cada clave es un nodo y su valor es una puntuación que indica su centralidad e importancia en la red.

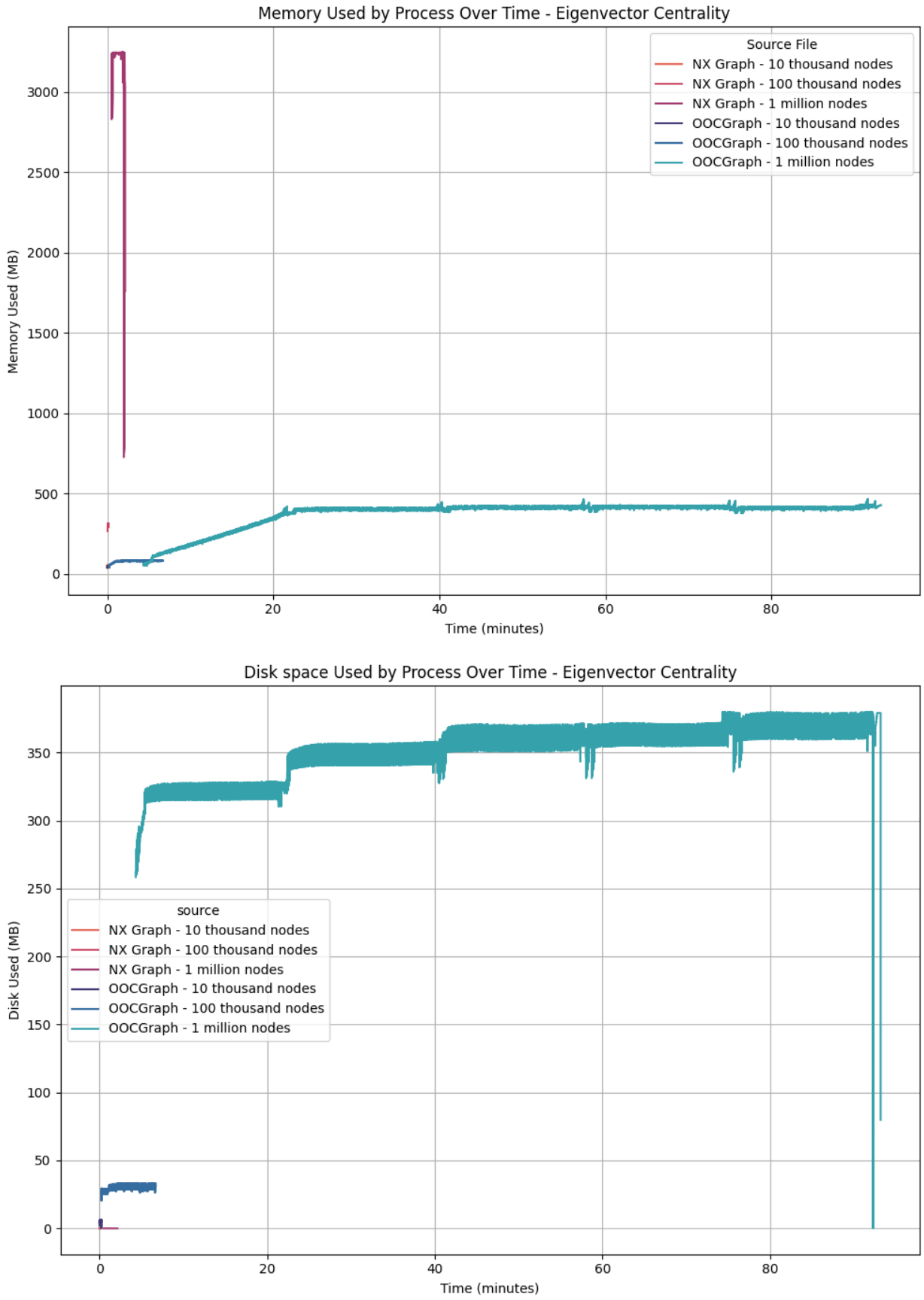

Figure 14: Memoria (superior) y espacio en disco (inferior) utilizados a traves del tiempo para el algoritmo de Eigenvector Centrality, con escala lineal en ambos ejes

Eigenvector Centrality calcula la centralidad del grafo de manera iterativa, finalizando cuando se llega al punto de convergencia o al alcanzar la cantidad máxima de iteraciones. El algoritmo utiliza un *OOCDict* inicial, indexado por los nodos del grafo, y por cada iteración, ajusta esta estructura hasta llegar a la centralidad de convergencia. Esto requiere, por cada iteración, generar una copia del *OOCDict*, con lo cual durante el procesamiento conviven ambos diccionarios.

Teniendo en cuenta las medidas de disco y memoria del *OOCGraph* de 1 millón de nodos, en la figura 14 se observa que se alinean saltos pronunciados en el uso de disco, en los minutos 20, 40, 60 y 80, con variaciones en el uso de memoria. Cada uno de estos coincide con la copia de la estructura de centralidad calculada en la iteración anterior. En el minuto 20, al ser la primera iteración del algoritmo, el impacto en disco es mayor ya que se crea la copia por primera vez. En las siguientes iteraciones, como esa estructura auxiliar se elimina y luego se vuelve a generar una copia, el impacto es mucho menor. De hecho, se puede ver una pequeña baja en uso de disco y luego una estabilización.

En la tabla 11 se muestra el tiempo promedio de ejecución en segundos para varias ejecuciones del algoritmo de Eigenvector Centrality en diversos grafos, tanto en su versión OOC como en su versión nativa de NetworkX, sin contar la carga del grafo. En la tabla 12 se presenta el uso promedio de memoria en megabytes para las mismas ejecuciones y versiones del algoritmo, también sin contar la carga del grafo.

Grafo	**Nodos**	*NXGraph* **(s)**	± **(s)**	*OOCGraph* **(s)**	± **(s)**	**Ratio (ooc/ Nx)**
1	10.000	0.3352	0.0132	13.9997	0.1646	41.76
2	100.000	6.4924	0.2515	396.7913	5.1044	60.11
3	1.000.000	85.7446	3.1576	5435.7982	39.2811	63.39

Table 11: Tiempos de ejecución de Eigenvector Centrality para los diferentes grafos. Se tomaron 6 mediciones para cada caso del *NXGraph* y 2 para cada caso del *OOCGraph*.

Grafo	**Nodos**	*NXGraph* **(Mb)**	± **(Mb)**	*OOCGraph* **(Mb)**	± **(Mb)**	**Ratio (OOC/NX)**
1	10.000	53.2	0.06	50.7	0.08	0.95
2	100.000	315.28	0.26	84.57	0.16	0.26
3	1.000.000	3315.84	0.54	460.80	11.85	0.14

Table 12: Uso de memoria en la ejecución de Eigenvector Centrality para los diferentes grafos. Se tomaron 6 mediciones para cada caso del *NXGraph* y 2 para cada caso del *OOCGraph*.

3.3.4 Single Source Dijkstra

El algoritmo de Dijkstra en NetworkX [27] encuentra el camino más corto desde un nodo fuente a todos los demás nodos en un grafo. Devuelve un diccionario en el que las claves son los nodos del grafo y los valores son las distancias desde el nodo fuente.

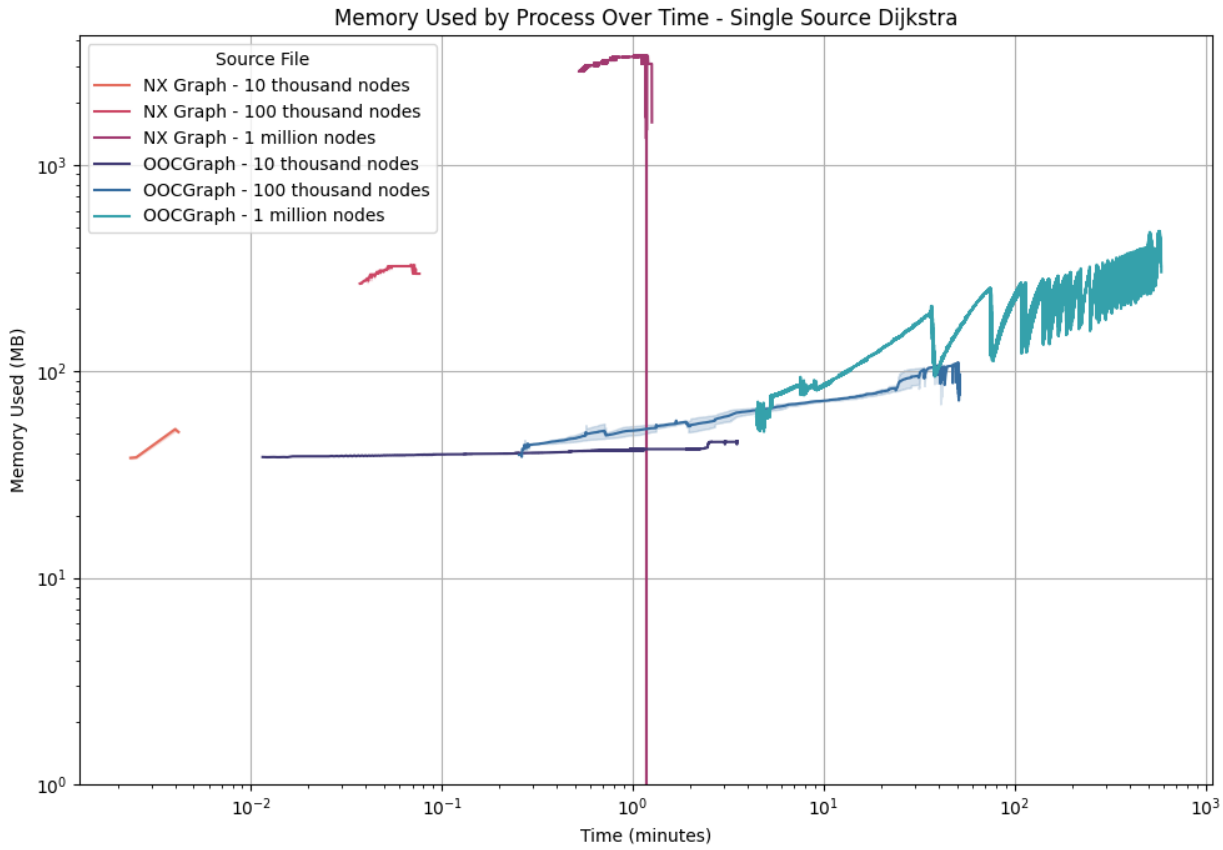

Figure 15: Memoria utilizada a traves del tiempo para el algoritmo de Single Source Dijkstra, con escala logaritmica en ambos ejes

Durante el procesamiento de Single Source Dijkstra se accede a la lista de adyacencia del grafo para lograr generar los caminos mínimos, accesos que no son lineales. Para optimizar este acceso a ciertas claves, LevelDB utiliza caches LRU. Como en el caso de este algoritmo los nodos que acceden a la lista de adyacencia son random, y por lo general no se vuelve a acceder a la misma adyacencia (no se vuelve a indexar el mismo nodo), ocurre que los cachés se llenan y se vacían, generando la aparición de los picos y valles que se observan en la figura 15 para el caso OOC.

Se observa que conforme el algoritmo va progresando, la aparición de picos y valles comienza a acelerarse. Esto se debe a que en el grafo hay más nodos ya recorridos y menos caminos por recorrer, por ende hay menos escrituras en disco, lo que implica menos procesamiento, que finalmente acelera el progreso del algoritmo.

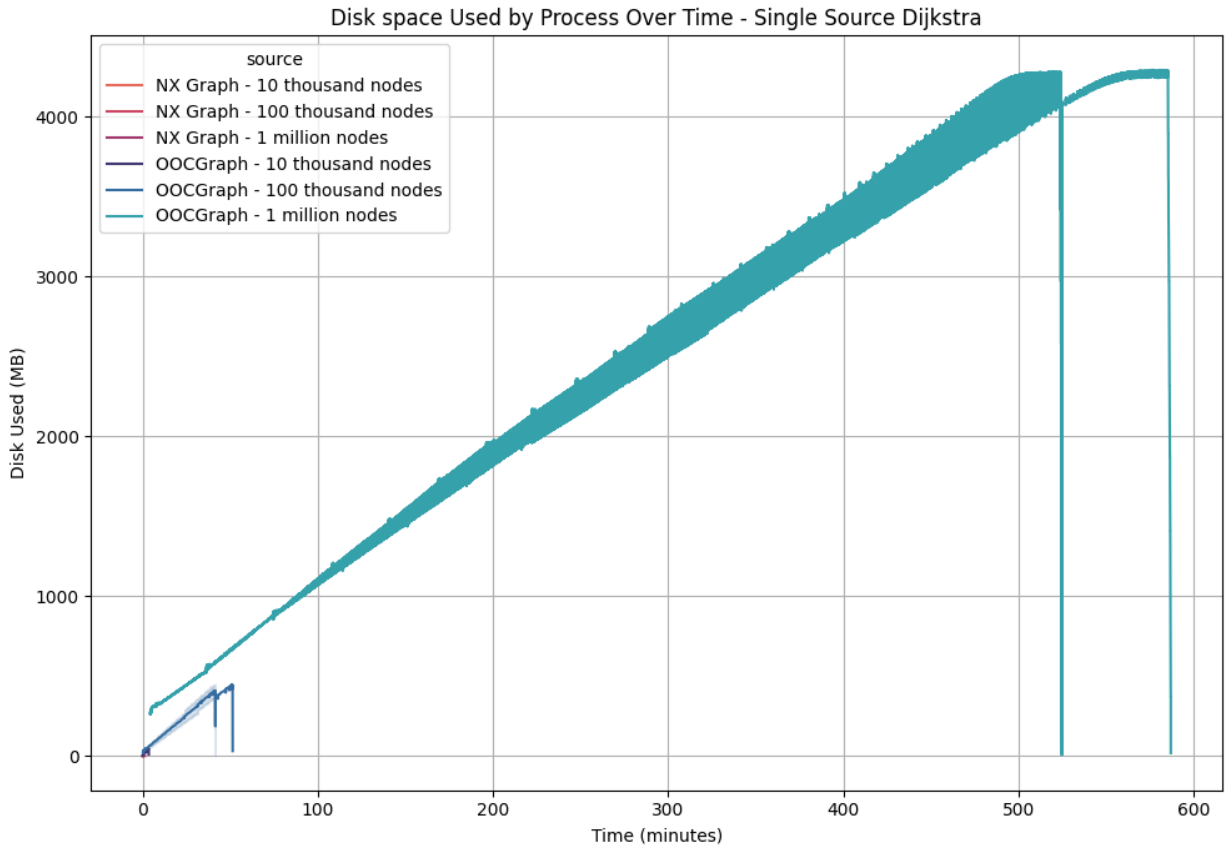

Figure 16: Espacio en disco utilizado a traves del tiempo para el algoritmo de Single Source Dijkstra, con escala lineal en ambos ejes

Se observa un crecimiento cuasi-lineal en la figura 16, acelerado en el primer tercio del gráfico, con una pendiente más alta, y luego se ve una leve desaceleración. La explicación está en línea con la del uso de memoria, conforme avanza el algoritmo se realizan menos escrituras a disco.

En la tabla 13 se muestra el tiempo promedio de ejecución en segundos para varias ejecuciones del algoritmo de Single Source Dijkstra en diversos grafos, tanto en su versión OOC como en su versión nativa de NetworkX, sin contar la carga del grafo. En la tabla 14 se presenta el uso promedio de memoria en megabytes para las mismas ejecuciones y versiones del algoritmo, también sin contar la carga del grafo.

Grafo	**Nodos**	*NXGraph* **(s)**	**± (s)**	*OOCGraph* **(s)**	**± (s)**	**Ratio (ooc/ Nx)**
1	10.000	0.0750	0.0036	203.6936	20.9666	2715.91
2	100.000	1.7674	0.0416	2868.9704	430.6724	1623.27
3	1.000.000	34.7484	1.3309	34224.5996	2780.0263	984.92

Table 13: Tiempos de ejecución de Single Source Dijkstra para los diferentes grafos. Se tomaron 6 mediciones para cada caso del *NXGraph* y 2 para cada caso del *OOCGraph*.

Grafo	**Nodos**	*NXGraph* **(Mb)**	**± (Mb)**	*OOCGraph* **(Mb)**	**± (Mb)**	**Ratio (OOC/NX)**
1	10.000	54.55	0.13	46.01	0.002	0.84
2	100.000	328.13	0.16	109.91	0.54	0.33
3	1.000.000	3456.48	0.17	481.28	3.99	0.14

Table 14: Uso de memoria en la ejecución de Single Source Dijkstra para los diferentes grafos. Se tomaron 6 mediciones para cada caso del *NXGraph* y 2 para cada caso del *OOCGraph*.

3.3.5 Single Source Bellman-Ford

El algoritmo de Single Source Bellman-Ford de NetworkX [28] encuentra el camino más corto desde un nodo fuente a todos los demás nodos en un grafo, incluso si el grafo tiene aristas con pesos negativos, donde el costo total de recorrer un ciclo es negativo. Devuelve un diccionario en el que las claves son los nodos del grafo y los valores son las distancias desde el nodo fuente, e informa si hay ciclos negativos en el grafo.

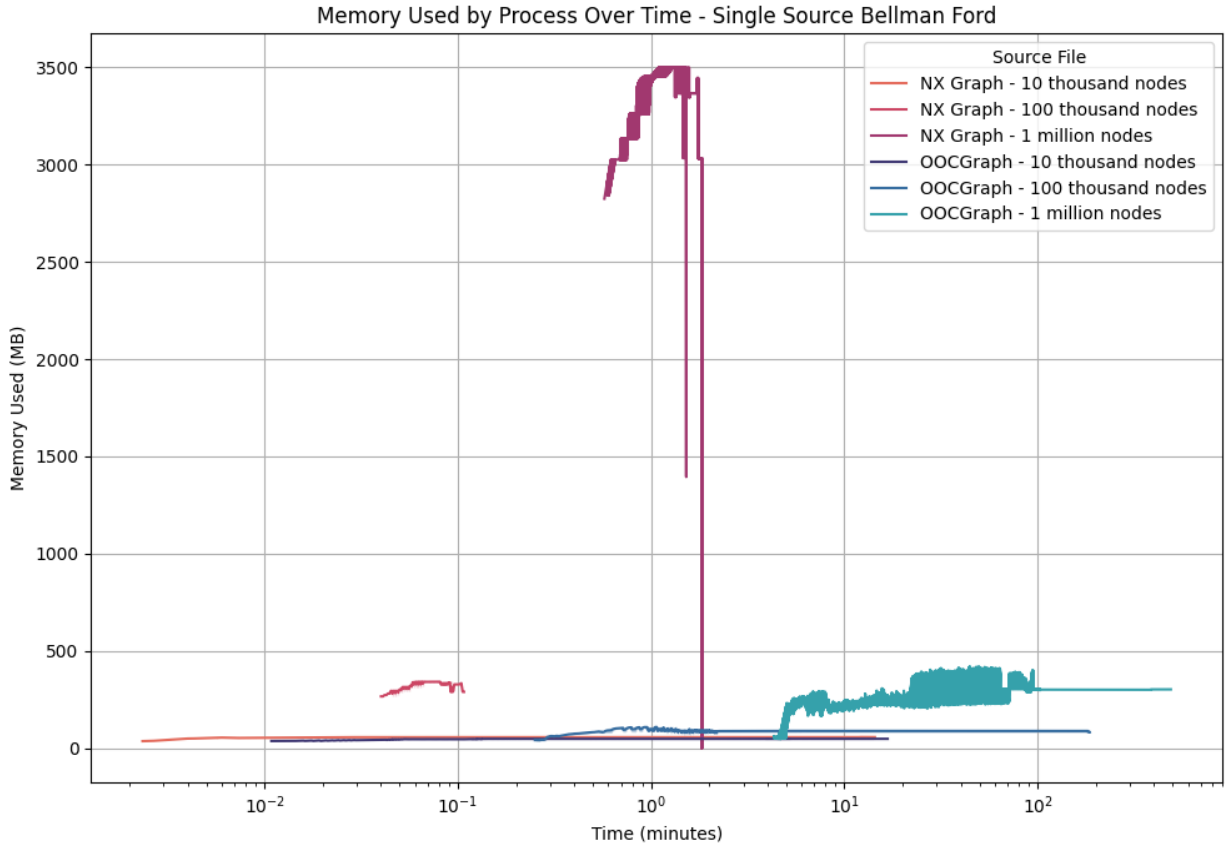

Figure 17: Memoria utilizada a traves del tiempo para el algoritmo de Single Source Bellman-Ford, con escala logaritmica en el eje X y lineal en el eje Y

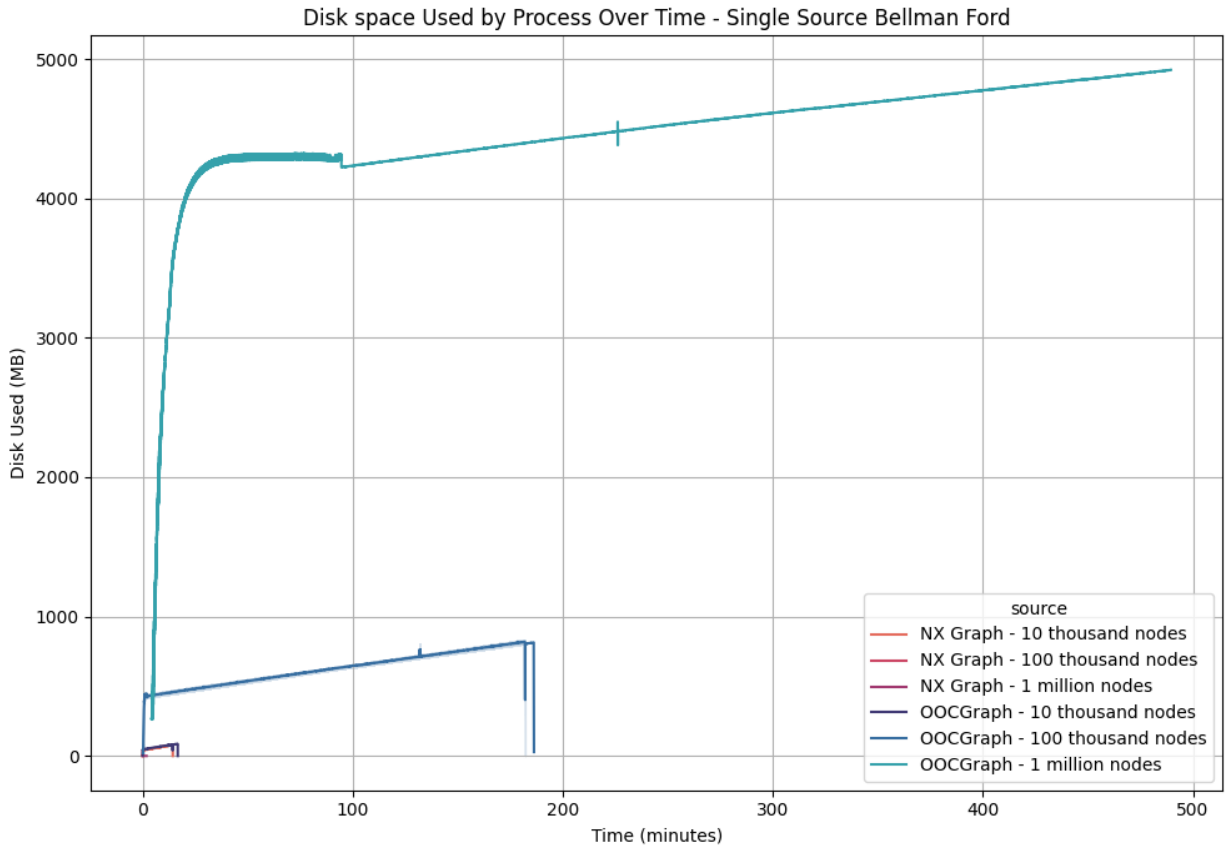

Figure 18: Espacio en disco utilizado a traves del tiempo para el algoritmo de Single Source Bellman-Ford, con escala lineal en ambos ejes

En la figura 18 se observa que existe un uso intensivo de disco durante el principio de la ejecución (hasta el minuto 100, aproximadamente). Esto se debe a que en ese periodo de tiempo Bellman-Ford carga diversas estructuras OOC con datos: itera la lista de adyacencia, crea los diccionarios de predecesores para los nodos no recorridos, actualiza el diccionario de distancias, etc. A medida que el algoritmo progresa y recorre los nodos, las escrituras disminuyen, de forma similar a lo que ocurría en Single Source Dijkstra (ver sección 3.3.4). Esto coincide con el amesetamiento del uso de disco.

Una vez terminado ese procesamiento, Bellman-Ford genera los caminos mínimos a partir de la lista de predecesores, que se observa en la figura como un crecimiento cuasi-lineal en el uso de disco, a partir del minuto 100 aproximadamente.

En la tabla 15 se muestra el tiempo promedio de ejecución en segundos para varias ejecuciones del algoritmo de Single Source Bellman-Ford en diversos grafos, tanto en su versión OOC como en su versión nativa de NetworkX, sin contar la carga del grafo. En la tabla 16 se presenta el uso promedio de memoria en megabytes para las mismas ejecuciones y versiones del algoritmo, también sin contar la carga del grafo.

Grafo	**Nodos**	*NXGraph* **(s)**	± **(s)**	*OOCGraph* **(s)**	± **(s)**	**Ratio (ooc/ Nx)**
1	10.000	0.1299	0.0023	966.3301	109.1089	7156.35
2	100.000	2.9494	0.2237	11464.4196	173.6607	4305.71
3	1.000.000	65.6851	7.3296	122257.3702	4250.2523	2325.61

Table 15: Tiempos de ejecución de Single Source Bellman-Ford para los diferentes grafos. Se tomaron 2 mediciones para cada caso.

Grafo	Nodos	*NXGraph* (Mb)	± (Mb)	*OOCGraph* (Mb)	± (Mb)	Ratio (OOC/NX)
1	10.000	56.38	1.45	50.44	0.08	0.89
2	100.000	342.69	0.16	78.17	0.12	0.22
3	1.000.000	3575.36	0.10	430.08	12.71	0.12

Table 16: Uso de memoria en la ejecución de Single Source Bellman-Ford para los diferentes grafos. Se tomaron 2 mediciones para cada caso.

4 Conclusiones

Actualmente, NetworkX es una de las bibliotecas más utilizadas y versátiles para Python en el contexto del análisis de grafos. Sin embargo, a medida que el tamaño de los grafos que se intentan analizar aumenta, NetworkX se encuentra con limitaciones relacionadas con la carga y el manejo de dichos grafos. Observamos a través de los experimentos que, tal como está la librería actualmente, conforme se procesan grafos de mayor tamaño se tiene la necesidad de poseer cada vez una mayor cantidad de memoria disponible para poder ejecutar los algoritmos, mayormente debido a que la biblioteca no utiliza el espacio del disco para el procesamiento.

Para solucionar las limitaciones de carga de NetworkX se crearon una serie de estructuras OOC que funcionan guardando los datos en disco, cuya finalidad es emular el funcionamiento de las estructuras en memoria de Python, para luego modificar los algoritmos de NetworkX reemplazando estas últimas por su versión OOC. Luego, se creó una nueva clase de grafo OOC, capaz de gestionar grafos de gran tamaño permitiendo su almacenamiento y acceso por partes en el disco. La combinación de estructuras y grafo OOC permite que los grafos que excedan la memoria principal de una máquina puedan ser cargados y analizados de manera eficiente, evitando la limitación de la memoria.

La biblioteca fue implementada de forma compatible hacia atrás, por lo que los usuarios pueden usar tanto la versión original de NetworkX como las funcionalidades OOC.

Grafos de millones de nodos que en la versión original de NetworkX requieren decenas o centenas de GB de RAM se almacenan en disco con nuestra versión OOC utilizando entre el 7% y el 33% de la memoria de la versión original para grafos de más de 100 mil nodos en los algoritmos analizados, lo que permite el procesamiento y análisis sin grandes recursos. Además, los grafos en disco ocupan apenas un 10% de lo que utilizan en memoria con la versión nativa. Sin embargo, el tiempo de ejecución aumenta notablemente debido a los múltiples accesos en disco, lo cual podría optimizarse en trabajos futuros. Por otra parte, se mantuvieron las prácticas y convenciones de nomenclatura de NetworkX, permitiendo a los usuarios existentes aprovechar las nuevas capacidades sin cambios significativos en su código.

5 Trabajo futuro

Soporte del grafo más allá de enteros

Una limitación que tiene la clase OutOfCoreGraph es que únicamente acepta el tipo de dato entero para nodos o aristas. Esta característica también la comparten las estructuras OOC (aunque algunas de ellas aceptan floats).

Para agregar la posibilidad de guardar otros tipos de datos en el grafo y estructuras, deberían modificarse tanto la clase LazyNode (para el *OOCGraph*) como las diversas estructuras OOC para que sean capaces de serializar/deserializar diferentes tipos de datos.

Una mejor solución sería implementar un wrapper sobre el *OOCGraph*, que reciba cualquier tipo de dato como nodo o arista y lo mapee a un integer, que es el dato que realmente se guardará en el grafo. Este wrapper debería funcionar de forma OOC y tener la capacidad de serializar y deserializar diversos tipos de datos. Como punto de partida podría usarse, por ejemplo, un OutOfCoreDict.

Optimización de algoritmos

La presente implementación no reescribe los algoritmos de NetworkX para hacerlos más eficientes, sino que reemplaza las estructuras en memoria por su variante OOC. Tal como están desarrollados, los algoritmos asumen costo

uniforme de acceso a memoria, lo cual no es cierto ya que ciertas operaciones suceden en la caché de las estructuras OOC, y otras necesitan acceder a disco para recuperar la información.

Existen implementaciones de algoritmos como BFS/DFS [29] optimizados específicamente para minimizar la cantidad de operaciones de I/O. Estas implementaciones podrían reemplazar a las actuales para optimizar el acceso a disco y por ende la performance de los algoritmos que utilizan estructuras OOC.

Otra opción es buscar optimizar las estructuras OOC. Las estructuras OOC que utilizan internamente un *OOCDict* utilizan internamente LevelDB, por lo que en vistas de mejorar los resultados obtenidos en los experimentos se podrían analizar algunas alternativas más eficientes, o incluso una o múltiples implementaciones custom de dicha biblioteca, que quizás podrían ser específicas para cada estructura o algoritmo.

Alternativa para OutOfCoreList

La estructura OOC que reemplaza a la lista de Python en nuestra implementación de los algoritmos de NetworkX, OutOfCoreList, utiliza internamente un OutOfCoreDict. En vistas de mejorar la performance de la implementación, podria reemplazarse en los algoritmos *OOCList* por LazyList, más eficiente debido que no cuenta con el overhead que agrega LevelDB al *OOCDict*.

Persistencia de datos

Nuestra implementación descarta todos los archivos utilizados por LevelDB en las estructuras OOC que lo utilizan internamente luego de terminado el proceso. Una posible mejora implicaría cambiar el diseño para persistir la información de las estructuras de datos más allá de la vida del proceso, implementado un sistema de checkpointing [30]. Esto nos permitiría, por ejemplo, no tener que cargar de cero un *OOCGraph* que ya cargamos previamente, ahorrando el tiempo de carga del mismo.

6 Agradecimientos

Queremos agradecer a Martin Di Paola por su apoyo, guía y por las inspiradoras discusiones que nos permitieron llevar a cabo este proyecto. Este proyecto fue desarrollado por estudiantes de la Universidad de Buenos Aires (UBA) como trabajo final de la carrera para obtener el título de Ingenieros en Informática y no tiene relación alguna con los desarrolladores originales de NetworkX.

References

[1] NetworkX: https://networkx.org/

[2] LevelDB: https://github.com/google/leveldb

[3] LSM Trees: https://en.wikipedia.org/wiki/Log-structured_merge-tree

[4] Plyvel: https://plyvel.readthedocs.io/

[5] tempfile: https://docs.python.org/3/library/tempfile.html

[6] tmpfs: https://docs.kernel.org/filesystems/tmpfs.html

[7] Pickle: https://docs.python.org/3/library/pickle.html

[8] Struct: https://docs.python.org/3/library/struct.html

[9] mmap: https://docs.python.org/3/library/mmap.html

[10] Sparse arrays: https://scientific-python.org/grants/sparse_arrays/

[11] Cyroaring: https://github.com/RoaringBitmap/CRoaring

[12] Pyroaring: https://pypi.org/project/pyroaring/

[13] PyRoaring BitMap Benchmarks: Pyroaring Repository

[14] DRY: https://en.wikipedia.org/wiki/Don%27t_repeat_yourself

[15] NetworkX OOC: https://github.com/leogm99/networkx-ooc

[16] Erdös-Rényi model properties: Wikipedia Erdös-Rényi, Erdös-Rényi paper

[17] Erdös-Rényi Graph Ten: Google Drive Erdös-Rényi Graph Ten

[18] Erdös-Rényi Graph Hundred: Google Drive Erdös-Rényi Graph Hundred

[19] Erdös-Rényi Graph Million: Google Drive Erdös-Rényi Graph Million

[20] Stanford Large Network Dataset Collection - Pokec social network: https://snap.stanford.edu/data/soc-Pokec.html

[21] Stanford Large Network Dataset Collection - LiveJournal social network: http://snap.stanford.edu/data/soc-LiveJournal1.html

[22] Predecessors Algorithm: Predecessors Algorithm NetworkX source code

[23] Graph diameter: https://en.wikipedia.org/wiki/Distance_(graph_theory)#Related_concepts

[24] Degree Centrality Algorithm: Degree Centrality Algorithm NetworkX source code

[25] Node degree: https://es.wikipedia.org/wiki/Grado_(teor%C3%ADa_de_grafos)

[26] Eigenvector Centrality Algorithm: Eigenvector Centrality Algorithm NetworkX source code

[27] Single Source Dijkstra Algorithm: Single Source Dijkstra Algorithm NetworkX source code

[28] Single Source Bellman-Ford Algorithm: Single Source Bellman-Ford Algorithm NetworkX source code

[29] External Memory Graph Traversal: https://en.wikipedia.org/wiki/External_memory_graph_traversal

[30] Checkpointing: https://en.wikipedia.org/wiki/Application_checkpointing

APÉNDICE A: Algoritmos implementados

Se listan a continuación las familias de algoritmos y sus subcategorías que tienen algoritmos implementados de forma totalmente Out Of Core. Para ver el detalle de cuáles algoritmos se implementaron para cada categoría/subcategoría, vea el detalle en la lista referenciada en el readme del proyecto, en nuestro repositorio [15].

Algorithms:

1. **Approximation**
   - MaxCut
   - Clustering
   - Vertex Cover
   - Connectivity
2. **Bipartite**
   - Basic
   - Clustering
   - Redundancy
   - Centrality
   - Matrix
   - Covering
   - Matching
   - Projections
   - Generators
3. **Boundary**
4. **Bridges**
   - VoteRank
   - Harmonic Centrality
   - Reaching
   - Laplacian
   - Load
   - Percolation
   - Trophic
   - Group Centrality
6. **Components**
   - Connected
   - Weakly connected
7. **Cuts**
8. **Distance Measures**
9. **Simple path**
10. **Shortest path**
    - Unweighted

5. **Centrality**
   - Betweeness
   - Degree
   - Eigenvector
   - Closeness
   - (Shortest Path) Betweenness
   - Weighted
   - Generic
   - Dense
   - Astar