UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

MEJORAR LA USABILIDAD Y EFECTIVIDAD DE UNA HERRAMIENTA DE PERFILAMIENTO DE MEMORIA

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERA CIVIL EN COMPUTACIÓN

ARACELI NICOLE QUEIROLO CÓRDOVA

PROFESOR GUÍA:
ALEXANDRE BERGEL
PROFESOR CO-GUÍA:
ALISON FERNÁNDEZ BLANCO

MIEMBROS DE LA COMISIÓN:
JOSÉ PINO URTUBIA
EDGARD PINEDA LEONE

Este trabajo ha sido parcialmente financiado por FONDECYT Regular 1200067, etapa 2020

SANTIAGO DE CHILE
2021

RESUMEN DE LA MEMORIA PARA OPTAR
AL TÍTULO DE INGENIERA CIVIL EN COMPUTACIÓN
POR: ARACELI NICOLE QUEIROLO CÓRDOVA
FECHA: MARZO 2021
PROF. GUÍA: ALEXANDRE BERGEL ALISON FERNÁNDEZ BLANCO

MEJORAR LA USABILIDAD Y EFECTIVIDAD DE UNA HERRAMIENTA DE PERFILAMIENTO DE MEMORIA

Comprender y abordar los problemas de memoria es una tarea importante, pero tediosa y compleja, y es exacerbada cuando el software tiene que manejar una gran cantidad de datos.

A pesar de la diversidad de herramientas diseñadas para esta actividad, los desarrolladores aun tienen problemas para interpretar sus resultados, lo que podría disuadirlos de utilizar un perfilador de memoria. Para reducir la barrera al utilizarlos e interpretarlos, la alumna de doctorado Alison Fernández desarrolló una herramienta de perfilamiento de consumo de memoria de aplicaciones hechas en Python que permite una visualización en Pharo, mostrando las llamadas entre métodos mediante grafos y el consumo de memoria vs cantidad de ejecuciones de cada método con un gráfico de dispersión. Así mismo se presentan pistas visuales en el código fuente para resaltar el consumo de memoria, entre otros.

Tal herramienta requiere ser evaluada para estudiar y mejorar su usabilidad y utilidad, pero, pese a la relevancia de analizar cómo las características de las técnicas visuales apoyan a los desarrolladores con sus necesidades, no se han realizado estudios similares, de los que se tenga conocimiento, para proporcionar preguntas relacionadas con la tarea de analizar la memoria ni que las considere en métodos de investigación de visualizaciones.

El presente trabajo muestra el diseño y ejecución de una metodología para evaluar el perfilador de memoria mencionado. Esta consiste en un experimento controlado con practicantes de Python, los cuales son 4 desarrolladores expertos y 8 no expertos. Se contempla el rendimiento de los participantes al usar la herramienta, se les hace preguntas abiertas sobre lo que han aprendido con ella, se compara con un perfilador estándar de Python y se evalúa la carga cognitiva al momento de usar el perfilador bajo estudio, así como su usabilidad.

Como resultado, se identifica un total de 372 preguntas relacionadas con el análisis del consumo de memoria que se dividen en 4 categorías. También se describen cómo los desarrolladores responden estas preguntas utilizando las características de los perfiladores escogidos.

Respecto a la experiencia de usuario, usando SUS, tanto expertos como no expertos evalúan positivamente el perfilador bajo estudio (sobre el promedio) por sobre el perfilador estándar (bajo el promedio). Sobre la carga cognitiva, medido con NASA TLX, tienen valores similares entre expertos y no expertos en el conjunto. Sin embargo, con el perfilador bajo estudio, el grupo de expertos considera que la carga por frustración y rendimiento es lo que tiene menos peso, mientras que la demanda mental tiene mayor preponderancia; en cambio, para los no expertos la carga por frustración y demanda física es lo que tiene menor peso, mientras que la demanda mental y temporal tienen mayor. Con el perfilador clásico, en ambos grupos aumenta la carga asociada a la demanda mental y la frustración, sin embargo, se mantiene menor la carga temporal que con el otro perfilador.

A mi mamá, papá y hermana.

Agradecimientos

En primer lugar, a Dios, ya que "*sin Él nada de lo que ha sido hecho, fue hecho*", porque Él me acompaña en cada proceso de mi vida, me ha dado la oportunidad de vivir esta experiencia, y a otorgado tantas bendiciones, entre ellas, las que agradezco a continuación.

A mi familia (mis padres Carlos y Patricia, mi hermana y confidente Shony), por su amor, entrega y paciencia, por su apoyo, ayudarme a atender mis necesidades para que mi paso por la universidad fuera lo mejor posible, por cuidarme, y más aun en pandemia. A mis tías Nancy y Tati, y primos Vado, Lolo y Nat por recibirme, cuidarme, los viajes y recibirme en Santiago cuando comencé mi vida universitaria. Y a todos mis familiares: abuelas, tíos (especialmente Giovanni y Fabiola), primos, mi hermano Ariel, mis sobrinos y cuñada, por todo lo que han hecho por mí y mi familia más cercana. También a mis compañeros de trasnoche y motivos de alegría por cerca de 12 años, Blacky, Mimi, Pelusa y Copito.

A mis amigos del colegio (más a Naty y Aloysha), universidad (Belén, Pía, Pili, Cami, Cynthia, Dani, José, Felipe, Vale, Maripi, Franco, Benja) y prácticas, por ayudarme, acompañarme en el proceso, los estudios juntos y pasar buenos ratos. A mis amigos del MUA UChile (mis queridos "Sobrinos de Bello", especialmente a Kevgar y Polyn, también a Ángela, Vale, Dama, Marcelo, Exe, Dante y Coni G), del MUA Santiago y de la Iglesia Adventista Santiago Centro (en especial a la familia Guzmán Pinto, Alcides, Marisol y Pedro, Dani, Karen y su esposo, pastor Esteban Díaz y su esposa), por sus oraciones, por estudiar juntos, por darme ánimos cuando me cansaba, los momentos de desconexión para conectarse con los amigos, hermanos y con Dios, por todo su apoyo, cuidados y ayuda brindada.

A los profesores del colegio, por la confianza que han puesto en mí y ayudarme a cimentar una base para mi paso por la universidad. Así también a los profesores de la universidad por compartir sus conocimientos y la motivación de algunos.

A mi profesor guía, Alexandre Bergel, y a Alison Fernández por confiar en mí y darme la oportunidad de trabajar en este proyecto. A los miembros de ISCLab por apoyarme y ayudarme en este trabajo. A los que participaron en los pilotos y experimentos, por su buena voluntad, haciendo posible este trabajo. También a quienes me ayudaron a convocar personas para estos estudios.

También a todos los que componen el Departamento de Ciencias de la Computación por recibirme. Por último, al proyecto FONDECYT Regular 1200067, por el financiamiento.

Tabla de Contenido

Índice de Tablas

Índice de Ilustraciones

Capítulo 1

Introducción

La actualidad exige cierta eficiencia en las soluciones tecnológicas que obtienen, como tiempo de respuesta, soporte de alta concurrencia en algunos casos, entre otros. Las plataformas modernas de ejecución de software están diseñadas para hacer frente de manera eficiente a varios problemas o necesidades, pero garantizar que el consumo de memoria no sea excesivo es una actividad que sigue siendo manual en muchos casos. Es muy conocido que la depuración de problemas de memoria es una actividad tediosa y propensa a errores [1], [6].

Existen herramientas que permiten crear perfiles de consumo de memoria que se basan en la introspección de la memoria, pero saber la causa requiere métricas en varias dimensiones y la mayoría de los perfiles populares de memoria y ejecución ofrecen representaciones de informes de memoria desconectados, lo que lleva a una complicada comprensión y explotación del software [6]. Además, se sabe que la visualización es efectiva para ayudar a los profesionales a llevar a cabo la comprensión del software y tareas de mantenimiento [4], [9].

A su vez, Python es uno de los lenguajes de programación más usados, de más rápido crecimiento[1] y más amado[2] en la actualidad. Por lo tanto, este lenguaje tiene una gran demanda debido a las bibliotecas proporcionadas para diferentes tendencias (como aprendizaje automático, ciencia de datos, entre otros), utilizadas en el mundo académico y en varias empresas [15], además que también ofrece varias opciones para el análisis del consumo de memoria.

Para ello, como parte del trabajo de la alumna de doctorado Alison Fernández Blanco, se desarrolló una herramienta de perfilamiento de consumo de memoria de aplicaciones hechas en Python que permite una visualización en el ambiente de programación de Pharo[3], mostrando un grafo de las llamadas entre métodos de un programa hecho en Python y un gráfico indicando también, con señales visuales, el consumo de memoria de cada método, líneas de código, entre otros. Tal herramienta, denominada AlProfiler, requiere ser evaluada para estudiar y mejorar su usabilidad y utilidad.

[1] `https://insights.stackoverflow.com/survey/2019`
[2] `https://stackoverflow.blog/2020/05/27/2020-stack-overflow-developer-survey-results/`
[3] `https://pharo.org/`

Por ello, para el presente trabajo de título se busca, en conjunto con la alumna de doctorado, mejorar la usabilidad y efectividad de la herramienta de perfilamiento de memoria mencionada anteriormente, llevando a cabo una experiencia de evaluación con usuarios, definiendo los aspectos necesarios para ello (como los criterios de efectividad, los usuarios, cómo realizar el estudio, pilotos, carga cognitiva, entre otros), y analizar los resultados que se obtengan posteriormente.

En general, el objetivo es finalizar el prototipo inicial de una herramienta visual de perfilamiento de memoria y evaluarla empíricamente. Más específicamente, se busca:

1. Hacer que esta herramienta se pueda usar lo más fácilmente posible por los probables usuarios.
2. Definir criterios de efectividad de la herramienta.
3. Definir y ejecutar estudios pilotos para poder mejorar la herramienta.
4. Definir un protocolo experimental para poder evaluar la herramienta. Este protocolo ha de ser basado en técnicas mixtas como estudio de casos y experiencia controlada.
5. Ejecutar una evaluación de la herramienta con usuarios.
6. Analizar los resultados para concluir sobre la efectividad y el uso de la herramienta.

Como resultado, se evalua la efectividad de la herramienta propuesta mediante la introducción de un método de investigación que implica la detección de preguntas formuladas por los desarrolladores durante el análisis del consumo de memoria y cómo utilizan esta visualización para responder a estas preguntas. También se compara esta visualización con el perfilador clásico Tracemalloc, considerando el impacto en la experiencia del usuario.

Esta metodología consiste en realizar sesiones de trabajo en las que participan 8 desarrolladores no expertos y 4 expertos. En primer lugar, se examina en detalle cómo los desarrolladores emplean la herramienta bajo estudio para analizar el consumo de memoria y, también, se analiza cómo esta herramienta afecta la experiencia del usuario de los participantes.

Como resultados, se identifican 372 preguntas relacionadas con el análisis de consumo de memoria, las que se dividen en 4 categorías. También se nota cómo los desarrolladores responden estas preguntas utilizando las características del este perfilador visual.

Respecto a la experiencia de usuario, usando un sistema de escala de usabilidad, tanto expertos como no expertos evalúan positivamente AlProfiler (73.75, lo que está sobre el promedio de lo aceptable) por sobre el perfilador estándar (42.92, lo que está bajo el promedio de lo aceptable), indicando lo usable que es AlProfiler.

Sobre la carga cognitiva, medido con el índice de carga de tareas NASA TLX, tienen puntajes similares entre expertos y no expertos en el conjunto. Sin embargo, se le otorgan valores más bajos a la carga asociada con AlProfiler que con un perfilador clásico, especialmente con los desarrolladores no expertos. Por otro lado, con un perfilador clásico son menores los valores de la carga cognitiva con el grupo de no expertos que con los expertos, pero superior a los de AlProfiler.

En forma separada, con AlProfiler, los desarrolladores no expertos le dan más peso a la demanda mental y temporal en términos de carga cognitiva, mientras que le dan muy poco a frustración y a la demanda física; por otro lado, los desarrolladores expertos también le dan mayor preponderancia a la demanda mental, pero menos a frustración y rendimiento. A su vez, con el perfilador clásico, en ambos grupos aumenta el puntaje de la carga asociada a la demanda mental y la frustración, pero se mantiene menor la carga temporal que con el otro perfilador.

A continuación, en el Capítulo 2 se presentan las herramientas clásicas de perfilamiento de memoria en Python, trabajos relacionados a la herramienta y metodologías utilizadas para estudios de este estilo de herramienta, y una breve descripción de AlProfiler. En el Capítulo 3 se describe el problema que se trata en este documento. En el Capítulo 4 se describe la herramienta para comprender los capítulos siguientes. En el Capítulo 5 se describe la solución, la que se divide en búsqueda de la metodología apropiada y la ejecución de la misma, mostrando los estudios realizados, resultados obtenidos y discusiones al respecto. Finalmente, en el Capítulo 6 se hacen las conclusiones del presente trabajo.

Capítulo 2

Estado del arte

2.1. Herramientas clásicas

Las herramientas de análisis de memoria son esenciales para encontrar y corregir anomalías en el uso de memoria de los sistemas de software. Sin embargo, en la situación actual, algunas herramientas de perfilamiento de memoria de código abierto para Python existentes son:

1. **PySizer**[1]: Luego de hacer un escaneo, muestra una consola de intérprete para analizar los resultados, los que se muestran en forma de tabla y textos, como en la Figura 2.1.

```
 Size    Total Object
12188    12188 dict at -0x4830bf0c
 3716    15904 dict at -0x48312994
 3410    19314 str at 0x814f898
 2672    21986 dict at -0x482a21c4
 2641    24627 str at 0x8149808
 2514    27141 str at 0x8151280
 2429    29570 str at 0x8168528
 2163    31733 str at 0x81fdf28
 2048    33781 dict at -0x482a20fc
 1796    35577 dict at -0x4828cd7c
```

Figura 2.1: Ejemplo de resultado en la documentación de PySizer.

2. **Guppy**[2]: Imprime en la salida una tabla de uso de memoria agrupada por tipo de objeto, como se ve en la Figura 2.2. Este tipo de creación de perfiles puede ser difícil de usar si se tiene una aplicación grande que utiliza un número relativamente pequeño de tipos de objetos.
3. **Módulo de python memory-profiler**[3]: Colocando el decorador @profile alrededor de cualquier método que se quiera estudiar, se muestra el uso de memoria por línea en forma de tabla en el output, como en la Figura 2.3. Puede ser útil si se desea perfilar una sección de código con uso intensivo de memoria, pero no mucho si no se tiene idea de cuál es el mayor uso de memoria o si el código es demasiado grande para analizarlo.

[1] `https://pysizer.8325.org/doc/tutorial.html`
[2] `https://github.com/zhuyifei1999/guppy3`
[3] `https://pypi.org/project/memory-profiler/`

```
>>> from guppy import hpy; h=hpy()
>>> h.heap()
Partition of a set of 30976 objects. Total size = 3544220 bytes.
 Index  Count   %     Size   % Cumulative  % Kind (class / dict of class)
     0   8292  27   739022  21    739022  21 str
     1   7834  25   625624  18   1364646  39 tuple
     2   2079   7   300624   8   1665270  47 types.CodeType
     3    400   1   297088   8   1962358  55 type
     4   4168  13   279278   8   2241636  63 bytes
     5   1869   6   269136   8   2510772  71 function
     6    400   1   228464   6   2739236  77 dict of type
     7     79   0   139704   4   2878940  81 dict of module
     8   1061   3    93368   3   2972308  84 types.WrapperDescriptorType
     9    172   1    81712   2   3054020  86 dict (no owner)
<89 more rows. Type e.g. '_.more' to view.>
>>> h.heap().byid[0].sp
 0: h.Root.i0_modules['os'].__dict__
>>> h.iso(1,[],{})
Partition of a set of 3 objects. Total size = 348 bytes.
 Index  Count   %     Size   % Cumulative  % Kind (class / dict of class)
     0      1  33      248  71       248  71 dict (no owner)
     1      1  33       72  21       320  92 list
     2      1  33       28   8       348 100 int
```

Figura 2.2: Ejemplo de resultado en la documentación de Guppy.

```
Line #    Mem usage    Increment  Occurences   Line Contents
=============================================================
     3   38.816 MiB   38.816 MiB           1   @profile
     4                                         def my_func():
     5   46.492 MiB    7.676 MiB           1       a = [1] * (10 ** 6)
     6  199.117 MiB  152.625 MiB           1       b = [2] * (2 * 10 ** 7)
     7   46.629 MiB -152.488 MiB           1       del b
     8   46.629 MiB    0.000 MiB           1       return a
```

Figura 2.3: Ejemplo de resultado en la documentación de memory-profiler.

4. **Muppy**[4]: Un objeto rastreador crea resúmenes del consumo de memoria por tipos de objetos, los compara e imprime en la consola en forma de tablas o árboles, como en la Figura 2.4.
5. **Memprof**[5]: Es un generador de perfiles de memoria para Python que registra y traza el uso de memoria de todas las variables durante la ejecución de los métodos decorados con @memprof, creando archivos de registro con textos. Se pueden formar gráficos con esos archivos indicando el consumo de memoria a través del tiempo, pero no claramente la relación entre los distintos métodos. Se puede apreciar un ejemplo en la Figura 2.5.
6. **Tracemalloc**[6]: Es una herramienta de depuración para rastrear bloques de memoria asignados por Python, proporcionando el rastreo de dónde se asignó un objeto, estadísticas sobre bloques de memoria asignados y calcula diferencias para detectar pérdida de memoria entre 2 *snapshots*, los que se muestran en forma escrita. Un ejemplo de esta herramienta está en la Figura 2.6.

[4] `https://pythonhosted.org/Pympler/muppy.html`
[5] `https://github.com/jmdana/memprof`
[6] `https://docs.python.org/3/library/tracemalloc.html`

```
>>> sum2 = summary.summarize(muppy.get_objects())
>>> diff = summary.get_diff(sum1, sum2)
>>> summary.print_(diff)
                                types |   # objects |   total size
============================== | =========== | ============
                               list |        1097 |      1.07 MB
                                str |        1105 |     68.21 KB
                               dict |          14 |     21.08 KB
                 wrapper_descriptor |         215 |     16.80 KB
                                int |         121 |      2.84 KB
                              tuple |          30 |      2.02 KB
                 member_descriptor |          25 |      1.76 KB
                           weakref |          14 |      1.20 KB
                getset_descriptor |          15 |      1.05 KB
                method_descriptor |          12 |    864     B
  frame (codename: get_objects) |           1 |    488     B
     builtin_function_or_method |           6 |    432     B
     frame (codename: <module>) |           1 |    424     B
         classmethod_descriptor |           3 |    216     B
                           code |           1 |    120     B
```

(a) Tabla.

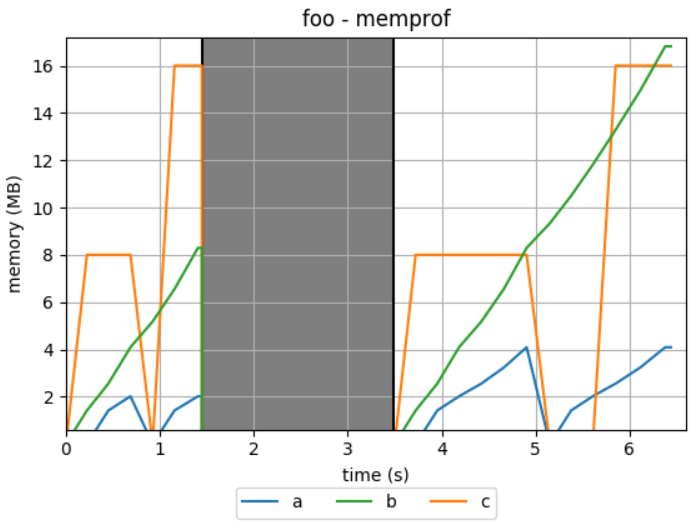
```
>>> cb.print_tree()
<type 'str'>-+-<type 'dict'>-+-<type 'list'>
             |               +-<type 'list'>
             |               +-<type 'list'>
             |
             +-<type 'dict'>-+-<type 'module'>
             |               +-<type 'list'>
             |               +-<type 'frame'>
             |               +-<type 'function'>
             |               +-<type 'list'>
             |               +-<type 'frame'>
             |               +-<type 'list'>
             |               +-<type 'function'>
             |               +-<type 'frame'>
             |
             +-<type 'list'>--<type 'dict'>
             +-<type 'tuple'>--<type 'dict'>
             +-<type 'dict'>--<class 'muppy.refbrowser.ConsoleBrowser'>
```

(b) Árbol.

Figura 2.4: Ejemplo de resultado en la documentación de Muppy.

```
0.000607
__annotations__ 240
FooClass        1056
0.328528
a       732816
b       1486000
c       8388672
__annotations__ 240
FooClass        1056
0.658483
a       1486000
b       2678096
c       8388672
__annotations__ 240
FooClass        1056
0.986977
a       2115952
b       4290016
c       8388672
__annotations__ 240
FooClass        1056
1.339916
a       732816
b       5429648
c       64
__annotations__ 240
FooClass        1056
```

(a) Texto.



(b) Gráfico.

Figura 2.5: Ejemplo de resultado en la documentación de Memprof.

```
[ Top 10 ]
<frozen importlib._bootstrap>:716: size=4855 KiB, count=39328, average=126 B
<frozen importlib._bootstrap>:284: size=521 KiB, count=3199, average=167 B
/usr/lib/python3.4/collections/__init__.py:368: size=244 KiB, count=2315, average=108 B
/usr/lib/python3.4/unittest/case.py:381: size=185 KiB, count=779, average=243 B
/usr/lib/python3.4/unittest/case.py:402: size=154 KiB, count=378, average=416 B
/usr/lib/python3.4/abc.py:133: size=88.7 KiB, count=347, average=262 B
<frozen importlib._bootstrap>:1446: size=70.4 KiB, count=911, average=79 B
<frozen importlib._bootstrap>:1454: size=52.0 KiB, count=25, average=2131 B
<string>:5: size=49.7 KiB, count=148, average=344 B
/usr/lib/python3.4/sysconfig.py:411: size=48.0 KiB, count=1, average=48.0 KiB
```

Figura 2.6: Ejemplo de resultado en la documentación de Tracemalloc.

7. **Objgraph**[7]: Es un módulo que permite explorar, visualmente, objetos del código en Python, como se ve en la Figura 2.7.

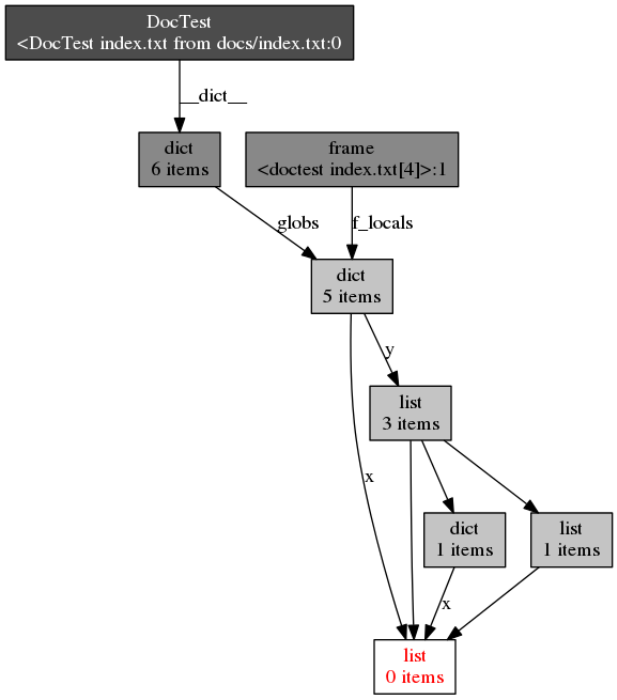


Figura 2.7: Ejemplo de resultado en la documentación de Objgraph.

Muchas de estas herramientas muestran texto en forma de tablas, con pocos o sin aspectos visuales, haciendo complejo el explorar detalles en demanda. También requieren hacerles cambios manualmente al código para poder usarse. En consecuencia, se considera que estas alternativas requieren el conocimiento y la experiencia de un desarrollador experto en abordar problemas de memoria.

2.2. Trabajos anteriores

De acuerdo a lo escrito en el artículo "Evaluating an Interactive Memory Analysis Tool: Findings from a Cognitive Walkthrough and a User Study" [17], aunque existen numerosas herramientas disponibles, casi no existen estudios empíricos sobre su utilidad para los desarrolladores en escenarios de uso típicos, limitándose a informar las métricas de rendimiento. A partir de los resultados obtenidos en este documento, algunas sugerencias fueron utilizar mecanismos flexibles de desglose, ocultar la complejidad utilizando la configuración predeterminada específica de la tarea, seleccionar y explicar cuidadosamente la terminología del análisis de memoria, mostrar detalles y resultados de análisis avanzados solo bajo demanda, usar la automatización para liberar a los usuarios de tareas complejas, proporcionar orientación para apoyar el aprendizaje exploratorio de las capacidades de análisis y proporcionar integración IDE para guiar el diagnóstico de errores de memoria.

[7] `https://mg.pov.lt/objgraph/`

Los perfiladores de memoria ayudan a los usuarios a analizar el consumo de memoria y a abordar problemas de memoria (por ejemplo, pérdidas de memoria). Estos perfiladores recopilan métricas relacionadas con el consumo de memoria de la ejecución del programa y las informan al usuario, lo cual se puede realizar de distintas maneras.

A su vez, la comunidad de Interacción Humano-Computador y Visualización de Software ha producido visualizaciones y formas de uso que se han probado efectivas. Algunos trabajos en esta área, en los que también ha trabajado el profesor guía, se explicitan en los papers "Effective Visualization of Object Allocation Sites" [6], "Evaluating a Visual Approach for Understanding JavaScript Source Code" [9] y "Performance Evolution Matrix: Visualizing Performance Variations along Software Versions" [8]. En estos documentos se encuentran herramientas relacionadas.

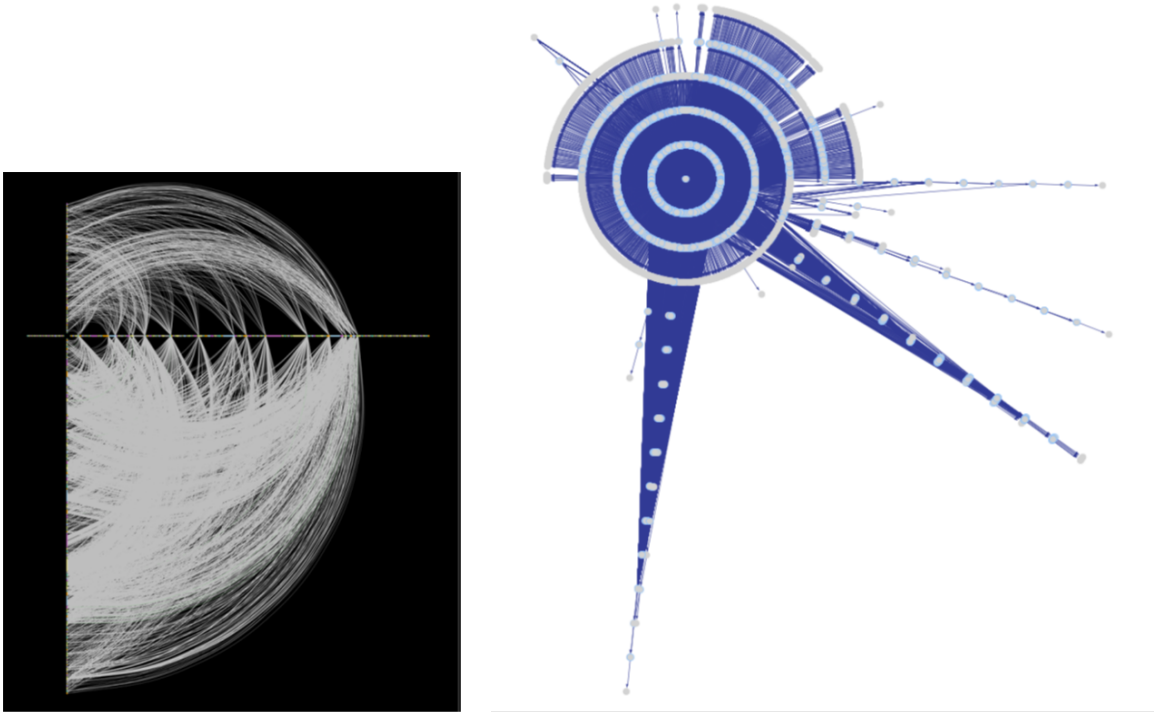
También, en otros artículos se ha trabajado en entender el consumo de memoria de una aplicación usando visualizaciones. En "Visualizing the Allocation and Death of Objects" [3] se visualiza el método de asignación y desasignación de cada objeto usando gráficos de colmena, relacionando tanto la creación como la muerte de un objeto, como se ve en la Figura 2.8(a). En "Heapviz: Interactive Heap Visualization for Program Understanding and Debugging" [10] muestra la relación de objetos en el *heap* con un enlace de nodo gráfico usando un diseño radial con una visualización compacta, ilustrándose en la Figura 2.8(b). En "Abstracting Runtime Heaps for Program Understanding" [13] se identifican problemas de memoria visualizando la relación de variables con un árbol, como en la Figura 2.8(c). En "Abstract Visualization of Runtime Memory Behavior" [2] se visualiza mediante glifos de datos la actividad de caché durante la ejecución de un programa, como el ejemplo de la Figura 2.8(d).

2.3. Metodología

Seleccionar un método de investigación para el estudio empírico es problemático porque los beneficios y desafíos de usar cada método aún no están bien catalogados, por lo que, para evaluar la herramienta, se cuenta con la guía "Selecting Empirical Methods for Software Engineering Research" [12] en el que basarse para hacer el diseño experimental.

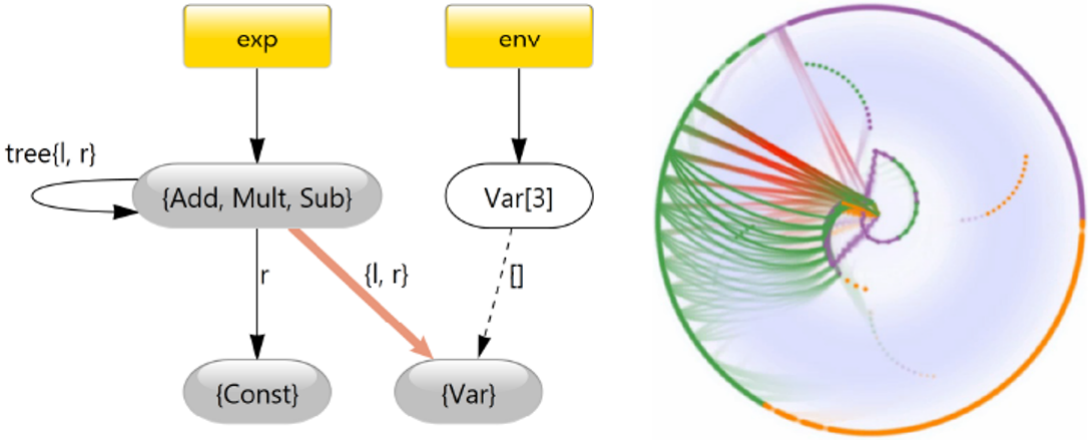
Existen diferentes estrategias que se utilizan para evaluar las visualizaciones de software, como se menciona en "A systematic literature review of software visualization evaluation" [14], donde se llama a los investigadores que proponen nuevas visualizaciones de software para proporcionar evidencia de su efectividad mediante la realización de estudios de casos exhaustivos para enfoques que deben estudiarse in situ, y cuando las variables pueden controlarse; experimentos con participantes seleccionados al azar de la audiencia objetivo y sistemas de software de código abierto del mundo real para promover la reproducibilidad y la replicabilidad.

La mayoría de los autores de algunos estudios proporcionan escenarios de uso, que describen cómo utilizar la visualización en aplicaciones de software de código abierto, por ejemplo en "Detailed heap profiling" [11]. Por otro lado, un menor número de artículos presenta estudios de usuarios o experimentos que involucran a participantes del público objetivo y analizan cómo los usuarios emplean la visualización teniendo en cuenta ciertos factores, como es el caso de "Effective Visualization of Object Allocation Sites" [6]. Por ejemplo, en "Evaluating an Interactive Memory Analysis Tool: Findings from a Cognitive Walkthrough and a User



(a) Grafos de colmena.

(b) Diseño radial.



(c) Árbol.

(d) Glifos.

Figura 2.8: Ejemplos de visualizadores.

Study" [17], donde se evalúa la utilidad de una herramienta utilizando un recorrido cognitivo y un estudio de usuario que muestra cómo los estudiantes usan la herramienta para realizar tareas relacionadas con el análisis del consumo de memoria.

Además, también es importante destacar la relevancia de analizar cómo las características de las técnicas visuales apoyan a los desarrolladores con sus necesidades, como se menciona en "Towards actionable visualisation in software development" [5]. En consecuencia, existen algunos artículos que representan las necesidades de los desarrolladores de incluir las preguntas formuladas por ellos durante el desarrollo de una tarea de software, como en "Asking

and Answering Questions during a Programming Change Task" [16], donde se identificaron 44 preguntas que los desarrolladores hicieron durante las tareas de cambio de software, clasificándose en 4 categorías: encontrar puntos de enfoque (con 5 preguntas), aprovechar esos puntos (con 15 preguntas), comprender un subgrafo (con 13 preguntas) y preguntas sobre grupos de subgrafos (con 11 preguntas); y en "Live Programming and Software Evolution: Questions during a Programming Change Task" [7], donde se hizo una réplica parcial del artículo anterior con un entorno de programación en vivo, reportando 8 nuevas preguntas y observando que las instalaciones de programación en vivo tienen un impacto en la forma en que los desarrolladores hacen preguntas sobre el código fuente y usan las herramientas para obtener el conocimiento correspondiente.

2.4. AlProfiler

En este trabajo se tomará como base un perfilador de consumo de memoria llamado AlProfiler, realizado por Alison Fernández Blanco, alumna de doctorado, cuyo código se encuentra disponible para los efectos de este trabajo, y con el que se han realizado algunos pilotos de estudio que han permitido realizarle algunos ajustes. La herramienta ofrece visualizaciones interactivas con la capacidad de explorar entre elementos y con claves visuales en el código fuente.

En la Figura 2.9 se ve parte del resultado de aplicar este perfilador sobre un archivo en lenguaje Python. La herramienta ofrece visualizaciones interactivas con la capacidad de explorar entre elementos y con claves visuales en el código fuente.

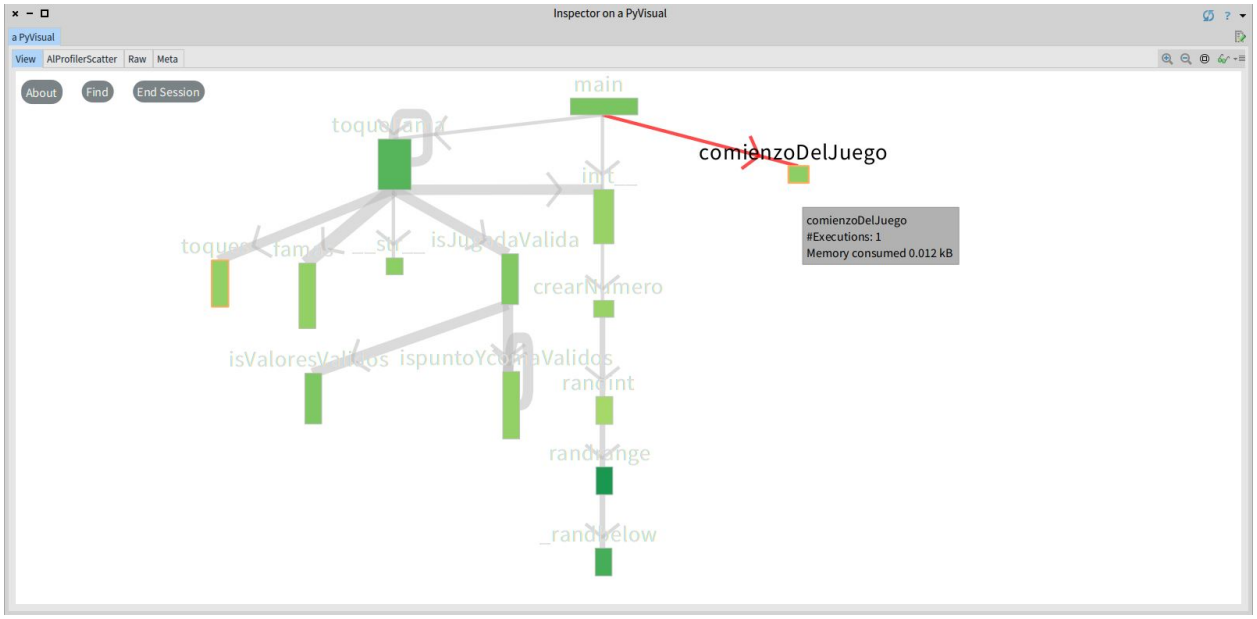


Figura 2.9: Ejemplo de resultado final del código actual.

Capítulo 3

Problema

Mantener bajo control el uso de memoria empleado en la ejecución de una aplicación es un desafío, ya que analizar el consumo de memoria implica varias métricas y requiere una comprensión profunda del código analizado. A pesar de la existencia de herramientas que permiten hacer perfilamiento de memoria, los enfoques actuales no consideran en detalle lo que un desarrollador necesita saber sobre la aplicación durante el análisis del consumo de memoria y cómo el desarrollador nota esa información, complicando a los desarrolladores no expertos al momento de interpretar esos resultados.

La hipótesis es que la creación de perfiles de memoria es una actividad compleja, existiendo una gran brecha para relacionar el informe de creación de perfiles de consumo de memoria con el código fuente; los perfiladores estándar de Python, en general, son complejos de usar, difíciles de interpretar y requieren modificación de la aplicación para usarse, ajustándose posiblemente más a las expectativas de programadores expertos, pero es probable que estas herramientas desanimen a los programadores no expertos que necesitan producir programas eficientes para procesar datos; por lo tanto, las barreras para usar un generador de perfiles de memoria para Python se pueden reducir mediante una visualización simple y una huella de memoria genérica.

Para probar esa hipótesis se crea el perfilador AlProfiler. Sin embargo, pese a la relevancia de analizar cómo las características de las técnicas visuales apoyan a los desarrolladores con sus necesidades, no se han realizado estudios similares, de los que se tenga conocimiento, para proporcionar preguntas relacionadas con la tarea de analizar la memoria ni que las considere en métodos de investigación de visualizaciones.

Por lo tanto, el problema principal que se trata en este documento es diseñar y ejecutar una metodología para evaluar el perfilador de memoria AlProfiler. En particular, para que la metodología sea de calidad deseada, esta debe:

- Contemplar cómo utilizan la herramienta de los participantes.
- Evaluar si el nivel de experiencia de los participantes tiene un impacto.
- Comparar AlProfiler con un perfilador estándar de Python.
- Evaluar la carga cognitiva al momento de usar AlProfiler.

Capítulo 4

AlProfiler

Durante la ejecución de una aplicación, se ejecutan varios métodos y se realizan muchas llamadas entre ellos. Representar esta información, por ejemplo, mediante el uso de una visualización de árbol de contexto de llamada podría presentar una gran cantidad de elementos visuales relacionados, produciendo una visualización compleja de entender.

Para presentar otra alternativa que aborde este problema, la herramienta AlProfiler utiliza un gráfico de llamada para mostrar la relación de llamadas entre métodos, la cantidad de memoria que consume cada una y ver si hace uso de una gran cantidad de memoria por ejecutar un método muchas veces (para lo cual se tiene la cantidad de ejecuciones de cada método); apoyado también del código fuente de cada función con las señales que indican en qué línea se asigna memoria.

Esta herramienta fue desarrollada por la alumna Alison Fernández Blanco, como parte de su trabajo de doctorado. En este capítulo, se describe la herramienta para entender la visualización, cómo se ejecuta y así entender los capítulos que siguientes.

4.1. Visualización

En la vista principal se ve un mapeo visual de la aplicación probada en forma de grafo, con la opción de explorar un diagrama de dispersión de memoria, como la Figura 4.1.

En ese ejemplo, el programa modela una porción de un litoral, considerando un perfil que contiene mar y un sector continental gobernado por cordilleras, con una planta ubicada en el borde entre el mar y las montañas, siendo la intención del programa graficar la temperatura atmosférica.

Se grafican las temperaturas en 2 horas distintas y se imprime la matriz resultante en consola. Se puede ver el código de ejemplo en el **Anexo A**. Se utilizan las bibliotecas Numpy, Matplotlib, Math y Random. Al ejecutar la herramienta con el mismo código, pero sin dibujar los gráficos, se obtiene la Figura 4.2.

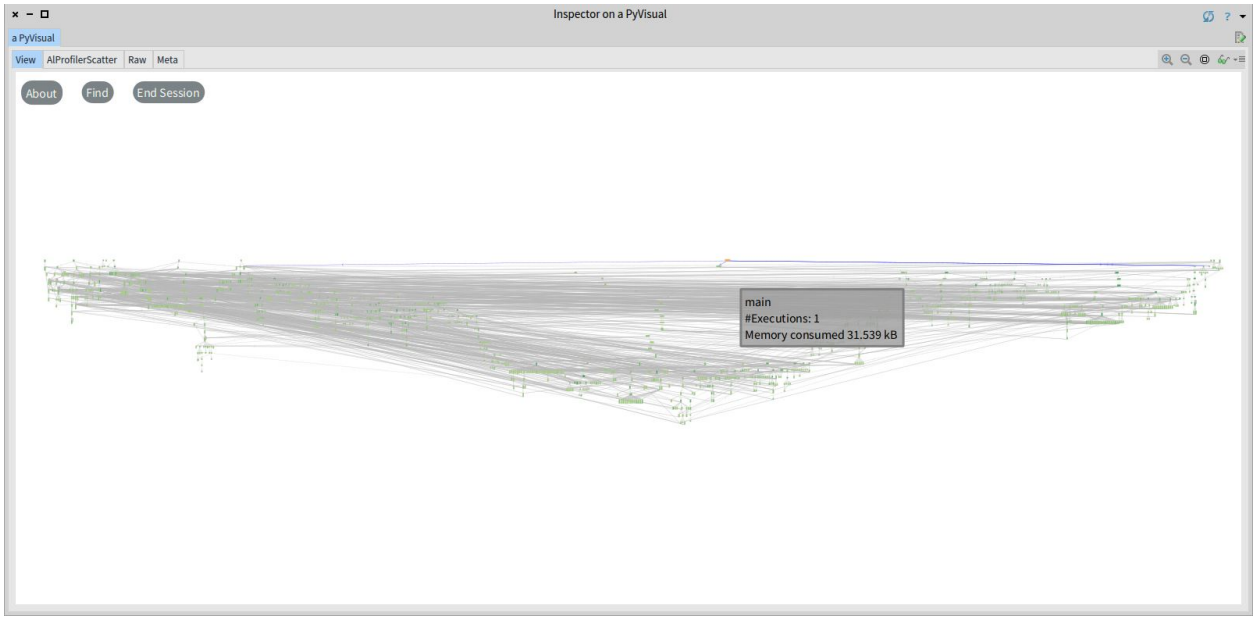


Figura 4.1: Ejemplo de resultado final de código de ejemplo.

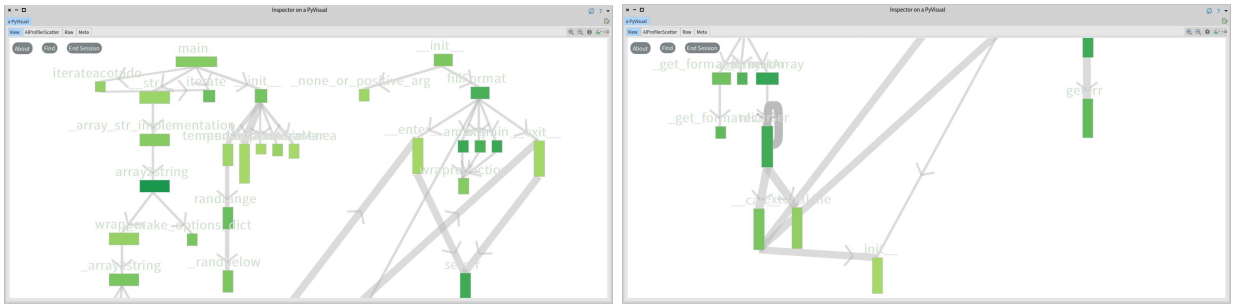


Figura 4.2: Ejemplo de resultado final de código de ejemplo, sin dibujar los gráficos de temperaturas.

Nodos: Cada nodo representa un método invocado durante la ejecución de la aplicación, donde:

- El ancho representa la memoria consumida por el respectivo método.
- La altura representa la cantidad de ejecuciones.
- El color representa la cantidad de líneas de código: menos líneas, más claro es.

Aristas: Representan la relación de llamadas entre funciones, usando una flecha para indicar la dirección de la llamada y el ancho como señal visual del número de llamadas entre dos funciones. En este ejemplo, el método *__enter__* es llamado más veces que *iterate*.

Disposición: La visualización utiliza por defecto un diseño vertical que coloca los nodos para representar las funciones que llaman a otras y el orden de ejecución. En consecuencia, la relación entre los métodos que llaman y que son llamados se realizan de arriba a abajo y el orden de ejecución se da de izquierda a derecha.

Interacciones: Los nodos del gráfico de llamadas ofrecen las siguientes interacciones:

- El nombre de la función está encima del nodo y desaparece si se usa zoom para alejarse.
- **Buscar**: con el botón *Find* se puede buscar un método específico de la aplicación probada. El resultado de la búsqueda se presenta resaltando el nodo buscado en color rojo por un breve período de tiempo, como se ve en la Figura 4.3 en que se busca el método *calorChimenea*.

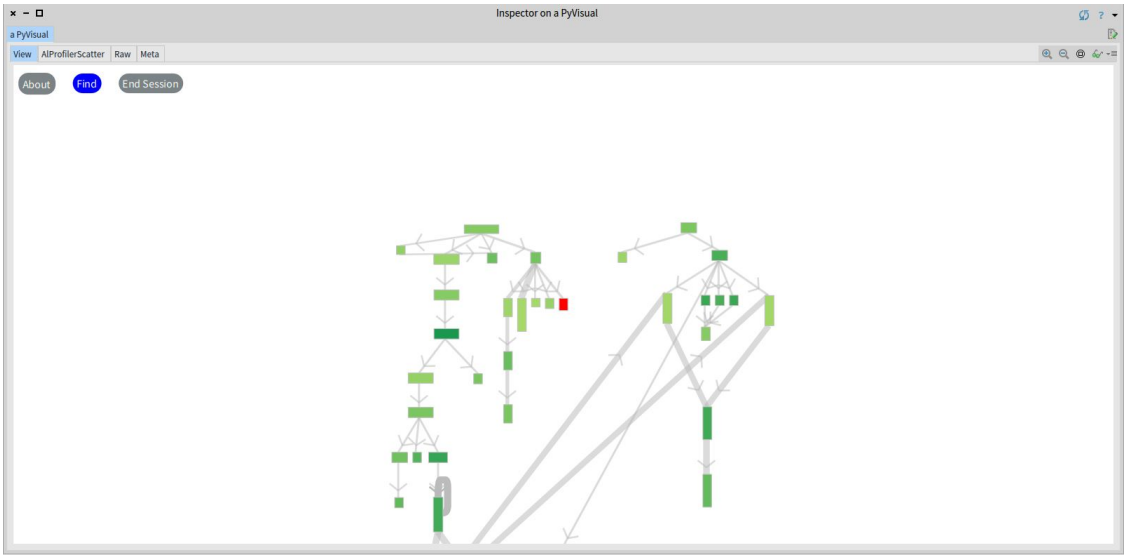


Figura 4.3: Buscar *calorChimenea*.

- Cuando el cursor pasa sobre un nodo, aparece una ventana emergente que muestra el nombre, número de ejecuciones y cantidad de memoria consumida en kilobytes (kB), resalta en naranjo el borde del nodo, las aristas entrantes en rojo y las salientes en azul, como muestra la Figura 4.4.
- Al mantener presionado un nodo, se puede arrastrar junto a los nodos involucrados con las aristas azules. Al hacer click en el nodo se presentan otras interacciones:
  - **Código fuente**: Se muestra el código del método en cuestión, resaltando las líneas que asignan memoria. El color del resaltado se desvanece de naranja oscuro a gris claro, dejando en blanco donde no se asigna memoria, como en la Figura 4.4.
  - **Navegación**: se permite navegar con un grafo de sub-llamadas del nodo seleccionado, el cual se encapsula en un cuadro al centro, a la izquierda con un borde rojo se anidan los nodos que lo llaman y, a la derecha, con un borde azul están anidados los nodos que son llamados por él. Cada nodo de este gráfico admite las mismas interacciones mencionadas, como en la Figura 4.5, donde se seleccionó el método *main* y luego la función *__str__* del grafo de sub-llamadas de *main*.

Diagrama de dispersión de memoria: La relación entre la memoria consumida y la cantidad de ejecuciones de las funciones se indica en un diagrama de dispersión, donde cada método es un punto colocado de acuerdo con su memoria consumida y su número de ejecuciones en los ejes Y y X, respectivamente, admitiendo las mismas interacciones anteriores, como se ve en la Figura 4.6:

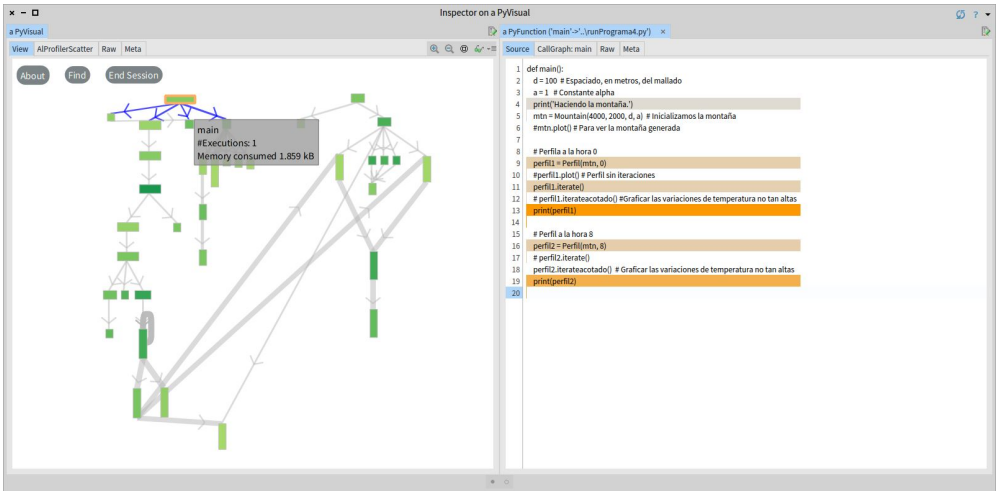


Figura 4.4: Código de *main*.

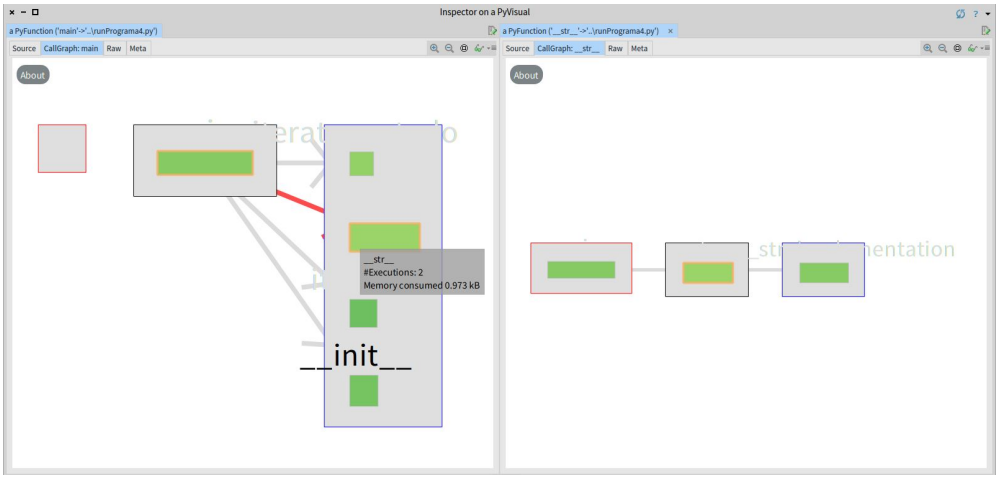


Figura 4.5: Grafos de sub-llamadas.

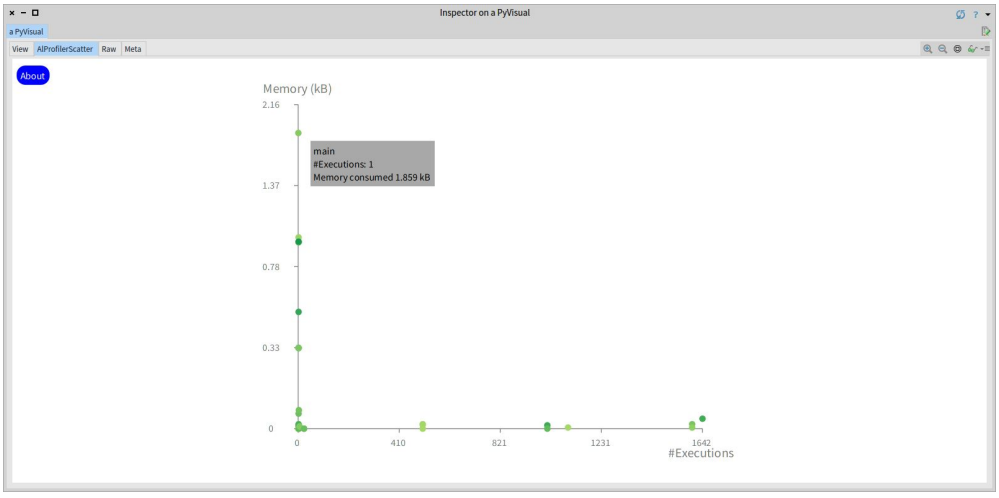


Figura 4.6: Gráfico de dispersión de memoria.

4.2. Ejecución

Para automatizar el procedimiento para ejecutar el perfilador se tiene un batch para Windows y un bash para Linux/MAC, los cuales fueron creados por la autora del presente documento.

En ellos, se verifica que el usuario tenga instalado Python 3 y, de acuerdo a eso, se utiliza `python/pip` o `python3/pip3` para las siguientes instrucciones. Si no se tiene, se solicita al usuario que lo instale. Posteriormente, se instala la biblioteca psutil en caso de no tenerse, puesto que es necesaria para la ejecución. A continuación, descarga la herramienta de Github y realiza la ejecución, configurando los *paths* correctos para llevarlo a cabo.

En Github, la herramienta tiene un tutorial para que los usuarios sepan cómo realizar la ejecución, abrir y entender la visualización final. Éste se encuentra en inglés, considerando que la herramienta se puede utilizar también a nivel internacional, tanto para experimentos como para trabajos. En este tutorial, se describen:

- **Requisitos**: Tener instalado alguna versión de Python 3 y Git, además de solicitar descargar un directorio desde Dropbox, el cual contiene el *batch/bash* antes mencionado, la máquina virtual de Pharo y las imágenes necesarias.
- **Cómo utilizarlo**:
  - Instrucciones de cómo ejecutar la herramienta.
  - Instrucciones para abrir Pharo y, dentro de la aplicación, abrir la visualización resultante de ejecutar la herramienta.
- **Visualización**: Explicación de lo que se muestra en la visualización obtenida.

Para ejecutar la herramienta, en una terminal se escribe `runProfiler.bat/sh`, el nombre del archivo que contiene el código Python que se quiere perfilar y el nombre del método que corre ese archivo, más los argumentos de ese método en caso de ser necesario. Todo lo anterior debe ir con las direcciones correspondientes. Por ejemplo, en caso que todos los archivos necesarios estén en un mismo directorio y se ejecuta la herramienta desde él, la instrucción que se anota en la terminal sería, por ejemplo, `./runProfiler.bat example.py main`. En ella se mostrará con '*' el avance del perfilamiento.

Una vez terminado el proceso, se crean 3 archivos csv en una carpeta con el nombre de *AlProfiler_* [nombre del archivo con el código probado]. A continuación, se ejecuta *Pharo* y se abre el directorio mencionado. Entonces, se muestra la visualización para su exploración.

Capítulo 5

Solución

5.1. Búsqueda de metodología

En la presente sección se muestran las actividades realizadas para llegar a la metodología escogida para evaluar la herramienta en estudio. En la sección 5.1.1 se detallan los estudios pilotos realizados para obtener comentarios de desarrolladores de software que permitan mejorar la herramienta y definir las sesiones de trabajo, y los resultados llegados. En la sección 5.1.2 se describe una herramienta de logs creada para recopilar información de los experimentos a realizar, y así detectar los eventos y cuánto tiempo pasa el usuario en cada vista del perfilador. En la sección 5.1.3 se explican las características de los perfiladores actuales de memoria en Python para comprender cómo funcionan estos actualmente y poder elegir el que se usa en los experimentos a realizar posteriormente y hacer las comparaciones. Finalmente, en la sección 5.1.4 se muestra la definición de la metodología escogida, en base a los resultados obtenidos, detallándose su ejecución en la sección 5.2.

5.1.1. Pilotos

Se realiza una serie de estudios pilotos para recabar comentarios de personas que programen en Python. A continuación se describe el diseño de los pilotos, los resultados, observaciones y opiniones. Los resultados indican la viabilidad de usar la visualización para analizar el consumo de memoria de un programa realizado en Python.

Participantes

Se seleccionan ocho personas para las sesiones. De ellos, cinco son estudiantes de pregrado (cuatro de Ingeniería Civil en Computación y uno en Ingeniería Civil Industrial), dos son estudiantes de magíster (uno de Ingeniería Civil en Matemática y uno en Ingeniería Civil en Computación e Ingeniería Civil en Matemática) y uno es egresado de Ingeniería Civil Mecánica de la Universidad de Chile. De ellos, siete son conocidos de la estudiante y uno es un conocido de amigos de la estudiante.

La información general de cada participante se encuentra en la Tabla 5.1, donde se muestra los años de experiencia de programación en Python, nivel de experto en un valor de 1 a 5 (de

principiante a experto, respectivamente) de acuerdo a la apreciación de ellos mismos, cómo se enfrentan a un problema de consumo de memoria por parte de un código propio, si trabajan en una compañía o proyecto open source y una referencia a su respectivo programa utilizado en las sesiones de trabajo.

ID	Años	Exp	Consumo de memoria	Trabajo	Código
P1	4	3	Examinar su código		Tarea 1.1, 1.2 y 1.3
P2	3	2	Aumentar la memoria y examinar su código		Tarea 2
P3	5	4	Cortar la ejecución del programa		Tarea 3
P4	5	4	Examinar su código y buscar en Internet		Tarea 4.1 Tarea 4.2
P5	7	3	Cortar la ejecución del programa	Proyecto open source	Tarea 5
P6	7	3	Aumentar la capacidad RAM y buscar en Internet		Tarea 6
P7	3	3	Reevaluar cómo resolver el problema	Compañía	Tarea 7
P8	6	3	Examinar su código y buscar en Internet	Compañía	Tarea 8

Tabla 5.1: Información general de los participantes de los pilotos.

Como se muestra en la Tabla 5.1, respecto a cómo se enfrentan al consumo excesivo de memoria:

- El primer participante revisa sus algoritmos para ver dónde se produce el problema.
- El segundo solicita más memoria para correr su aplicación. De no poderse, revisa la implementación de su código para ver si existe otra forma de hacer lo mismo, consumiendo menos memoria, como pasar de un método iterativo a uno recursivo o viceversa.
- Los participantes 3 y 5, cuando se enfrentan a un problema de consumo de memoria, usualmente detienen la ejecución del programa y lo vuelven a correr en otro momento.
- Los participantes 4 y 8 revisan su código para analizar si codificaron algo que produce el consumo excesivo de memoria y hacen consultas en Internet para estudiar mejor cuál es el problema y cómo solucionarlo.
- El participante 6, además de hacer consultas por Internet, aumenta la capacidad de memoria en RAM de su equipo en caso de ser necesario.
- El participante 7 revisa su código y analiza si esa es la mejor manera de resolver el problema que iba a enfrentar al momento de realizarlo.
- Los participantes 1, 2, 7 han programado en Python más años, pero en los últimos años mencionados en la Tabla 5.1 se han dedicado un poco más a este lenguaje.
- En la Tabla 5.1 se explica en qué consisten los códigos de cada participante.

Códigos

A cada participante se le pide seleccionar un código escrito por ellos mismos para las sesiones de los pilotos. La descripción de cada uno se presenta en la Tabla 5.2.

Código	Descripción
Tarea 1.1	Dentro de una matriz de 20×20, busca los 4 valores en línea (en vertical, diagonal u horizontal), cuya multiplicación sea la mayor posible.
Tarea 1.2	Dada una matriz, la muestra como un grafo. Luego, resuelve un modelo matemático basado en el grafo.
Tarea 1.3	Tarea 1.2, pero sin dibujar el grafo.
Tarea 2	Arma grafos para, después, sacar el minimal, en el caso particular de hacer una maratón de películas, donde las películas son los nodos.
Tarea 3	Coloca mensajes codificados dentro de imágenes y los decodifica.
Tarea 4.1	Clasifica 4 dataset de tweets si son tristes, divertidos, de enojo o de miedo.
Tarea 4.2	Calcula el sueldo bruto de un trabajador.
Tarea 5	Abre un proceso de un código escrito en C y le manda mensajes por un protocolo aleatorio.
Tarea 6	Hace un grafo con pesos y aristas aleatorias, sin repetirse, en el que ejecuta Dijkstra posteriormente.
Tarea 7	Hace una neurona y la entrena con el dataset de Iris.
Tarea 8	Simula un cubo rubik, el cual puede armar uno mismo o el programa simula el armado.

Tabla 5.2: Códigos de los participantes de los pilotos.

Sesiones

Una sesión involucra a un participante y su código, llevándose a cabo con una metodología con los dos primeros participantes y con otra para los demás. Estas se pueden resumir de la siguiente manera:

- **Preguntas generales**: el participante aborda 6 preguntas sobre su experiencia general y su experiencia en analizar el consumo de memoria.
- **Tutorial**: el participante lee el tutorial, hace consultas y sugerencias respecto a él.
- **Exploración**: de P3 a P8, el participante explora la herramienta con un ejemplo y puede hacer cualquier pregunta en caso de duda. Luego responde 3 preguntas para verificar que entiende el tutorial: ¿cuál es el método que consume más memoria?, ¿cuántas veces se ejecuta el método X?, ¿en qué líneas del código del método Y se asigna memoria?
- **Tareas**: el participante corre el *profiler* en su propio código y visualiza el resultado. A continuación, responde:
  - Primeros 2 participantes: si entiende la visualización, la información expuesta en ella y si tuvo algún problema usándola.
  - Últimos 6 participantes: qué aprendió de su aplicación con esta herramienta y si ha encontrado una oportunidad de mejora en su aplicación.
- **Comentarios**: el participante da sugerencias y comentarios de manera informal y oral.

Las modificaciones de la metodología utilizada se deben a los comentarios presentados por los participantes e ir probando nuevas alternativas.

Recolección de datos

Cada sesión es por videollamada y se graba (con la autorización de cada participante) a través de la plataforma de Zoom, para observar cómo utilizan la herramienta, mantener un registro y así extraer más información de la interacción entre el participante y la visualización.

Posterior a las sesiones, se transcriben para obtener las preguntas que frecuentemente trataban de responder usando la herramienta.

Resultados

Esta sección describe los resultados de los pilotos en la realización de las tareas:

- **Comprensión de la visualización**: todos los participantes respondieron correctamente las preguntas en el momento de **Exploración**, entendieron en gran medida y utilizaron los diferentes elementos de visualización, interacciones y la información mostrada, según se puede deducir de los comentarios hechos sobre sus resultados.
- **Caracterizar el consumo de memoria**: todos los participantes caracterizaron el consumo de memoria de sus respectivos códigos. Algunos de ellos estaban sorprendidos por la cantidad de llamadas entre algunos métodos y el impacto que tienen en el uso de memoria las bibliotecas utilizadas por su código (no esperaban que se llamaran tantas funciones dentro de ellas ni la cantidad de memoria que utilizaban), otros estaban preocupados por la cantidad de uso de memoria en algunas funciones.
  - P1 vio el flujo de llamadas desde *main* siguiendo una rama hasta cierto punto en la Tarea 1.2 y Tarea 1.3, ve la memoria consumida y la cantidad de ejecuciones, viendo que lo que más memoria consume son los métodos que dibujan los grafos, lo que también se comprueba entre esas tareas.
  - P2 mencionó el recorrido del uso de la memoria según lo que observó en el grafo principal y los de sub-llamadas.
  - P3 habló del uso de memoria de los métodos propios y algunos otros, indicando que el programa hizo varias acciones antes de abrir la imagen.
  - P4 habló de la memoria ejecutada por cada función y cómo se llaman basándose en los grafos de sub-llamadas.
  - P5 se enfocó, principalmente, en un método escrito por ella, el que llama a muchas funciones varias veces, haciéndole sentido que use mucha memoria. También dice que puede hacer un análisis de las llamadas entre funciones y le llama la atención que no hay tanta variación en ancho, largo y colores de los nodos como esperaba.
  - P6 se concentró en ver las líneas que consumían más memoria dentro de su código, siguiendo las ramas del grafo.
  - P7 ve métodos que él escribió, su memoria consumida y cantidad de ejecuciones, las llamadas realizadas, el código fuente y trata de relacionar las líneas destacadas con los nodos del grafo de sub-llamadas.

  - P8 caracterizó los métodos que iban consumiendo más memoria dentro de algunas ramas en particular, vio la relación entre las líneas destacadas del código fuente con los nodos en el grafo principal y el flujo de las llamadas.
- **Mejoras del consumo de memoria**: De los 6 participantes que respondieron a la pregunta asociada, uno respondió que no ve forma de mejorar el código, uno contestó que podría reducir el uso de memoria y dijo algunas posibles soluciones, como ver la posibilidad de usar otra biblioteca que no sea tan costosa en términos de consumo de memoria, y los demás respondieron cómo usarían la herramienta para ver si pueden mejorar su aplicación.

Observaciones: En el caso de obtener diagramas muy grandes, se focalizaban en analizar algunas ramas más específicas, principalmente las que tuvieran los métodos que ellos mismos programaron. Los participantes usaron ampliamente las ventanas emergentes para obtener el nombre de la función, cantidad de memoria utilizada, el número de ejecuciones, el resaltado de las aristas para enfocarse en la relación entre los métodos, la navegación para obtener el gráfico de sub-llamadas y el código fuente para obtener las líneas resaltadas y, finalmente, la búsqueda para acceder rápidamente a algunas funciones específicas.

A continuación, se ven algunas de las imágenes resultantes de utilizar la herramienta con los distintos códigos.

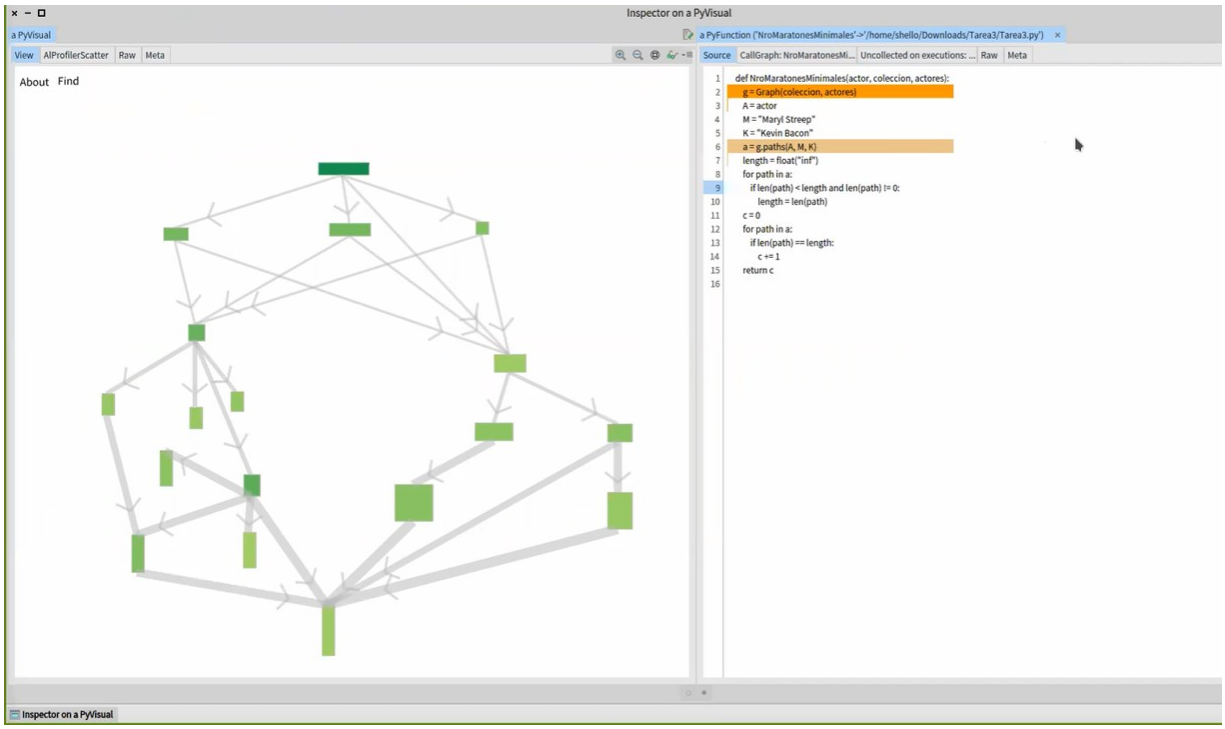
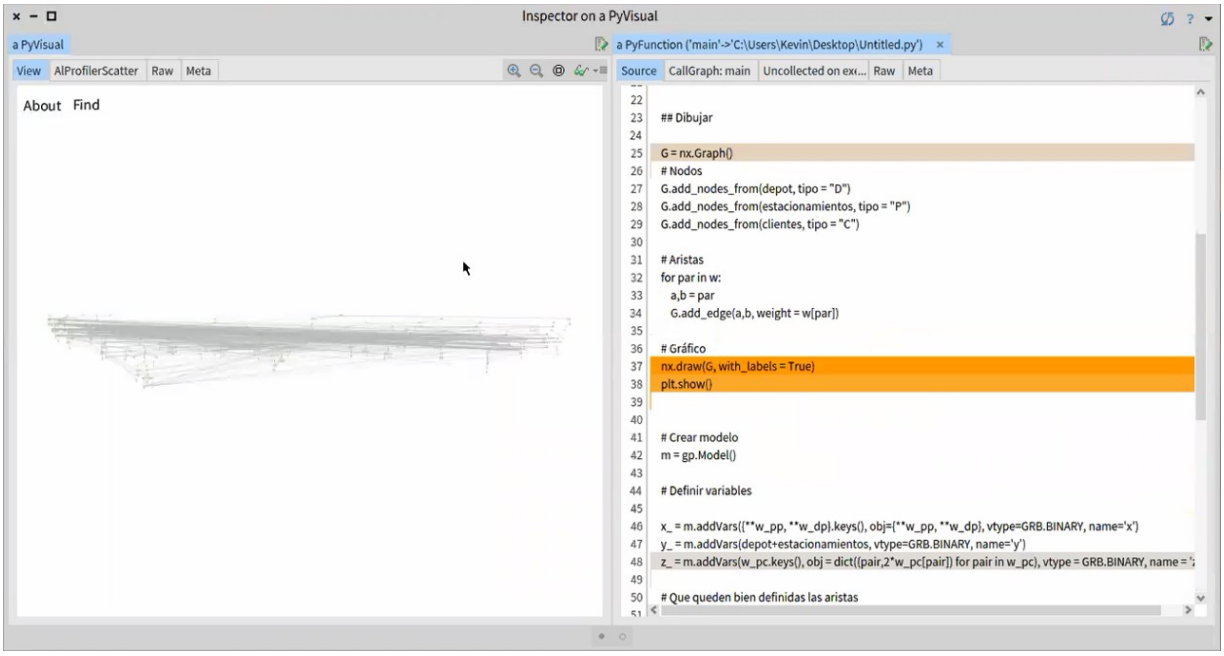
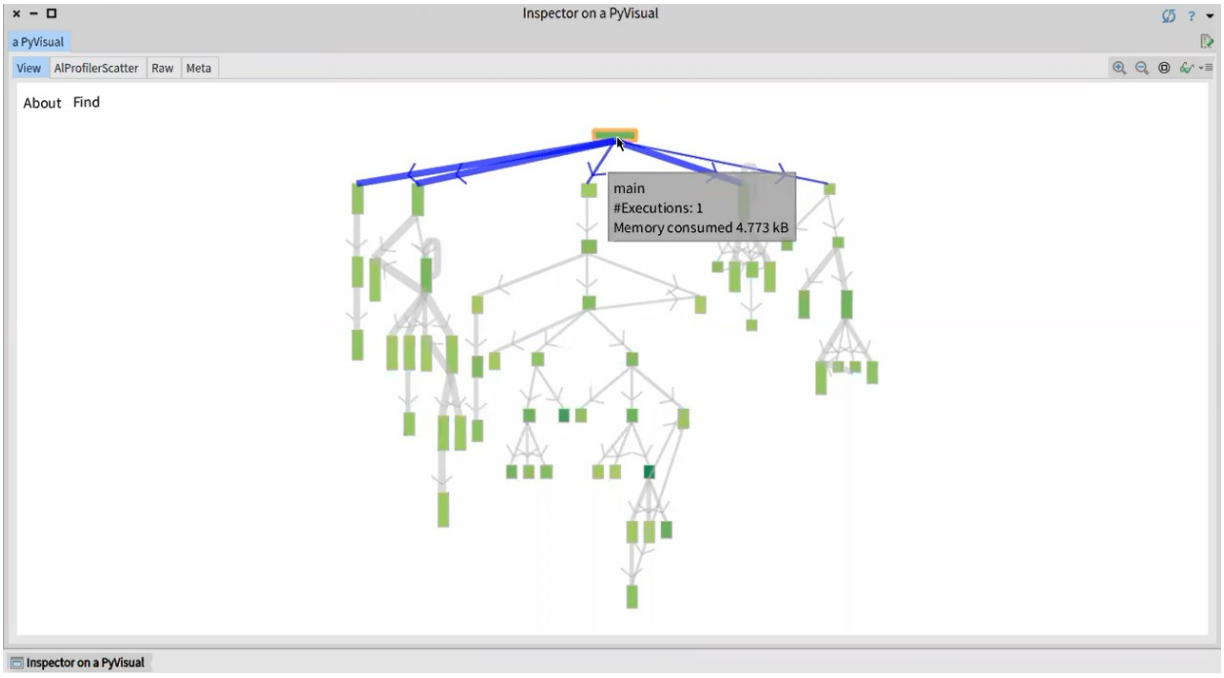


Figura 5.1: Resultado del participante P2. Vista principal y código fuente de de *NroMaratonesMinimales*.

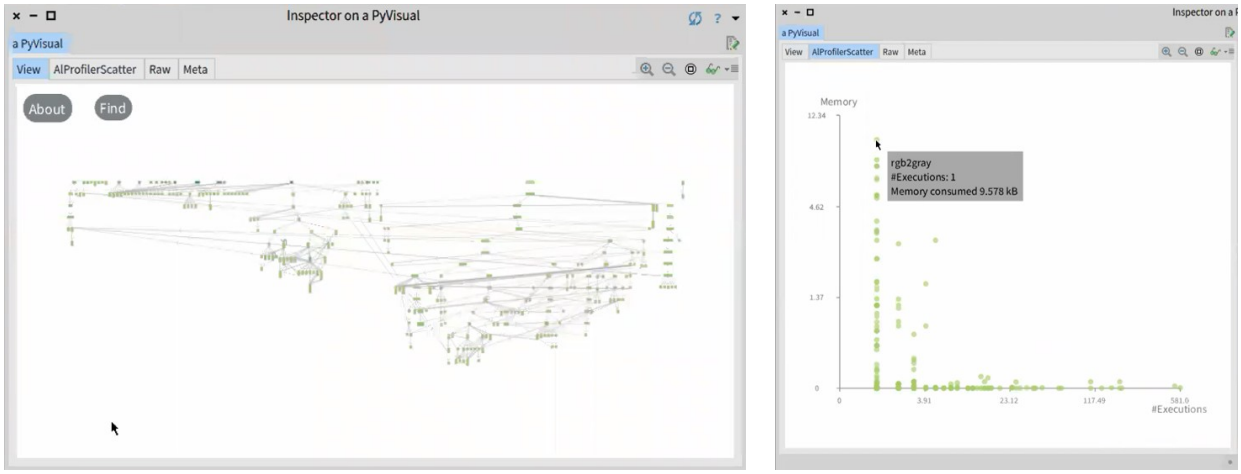


(a) Tarea 1.2. Vista principal y código fuente de *main*.



(b) Tarea 1.3. Vista principal.

Figura 5.2: Resultado del participante P1. Entre las bibliotecas que utilizó se encuentra matplotlib, networkx (en Tarea 1.2), gurobi, numpy, itertools.



(a) Vista principal.

(b) Diagrama de dispersión.

Figura 5.3: Resultado del participante P3. Entre las bibliotecas usadas están numpy, pai_io y argparse.

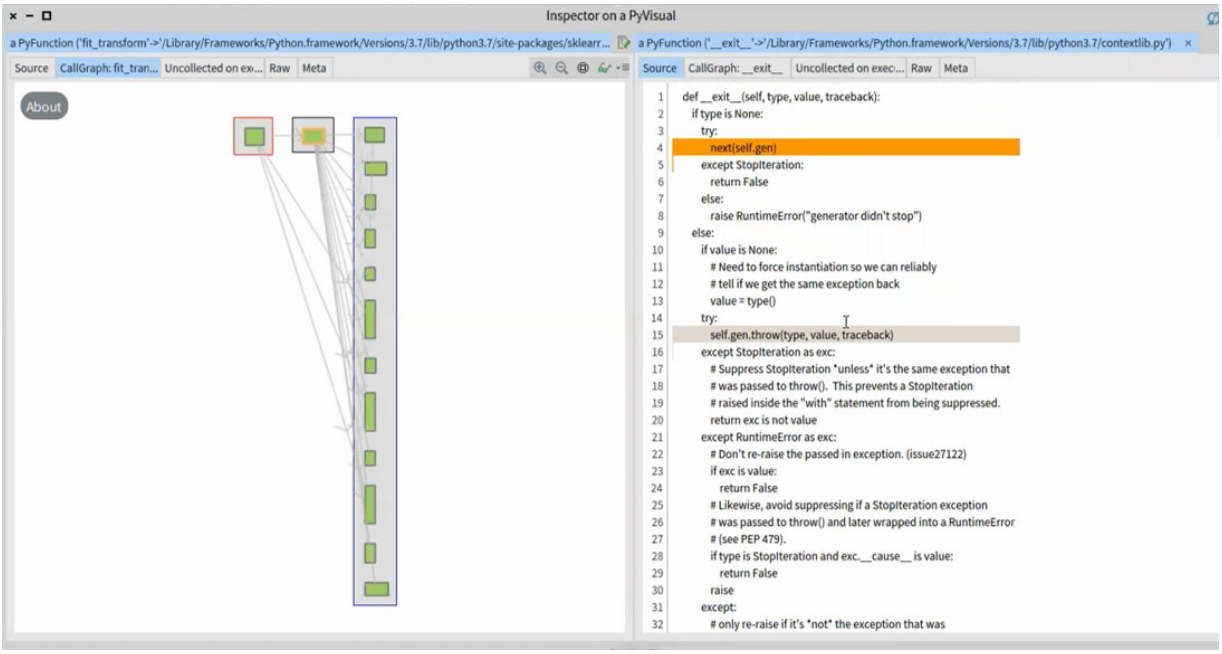


Figura 5.4: Resultado del participante P4 con la tarea 4.1. Entre las bibliotecas usadas están TensorFlow. Gráfico de sub-llamadas de *fit_transform* y código fuente de *__exit__*.

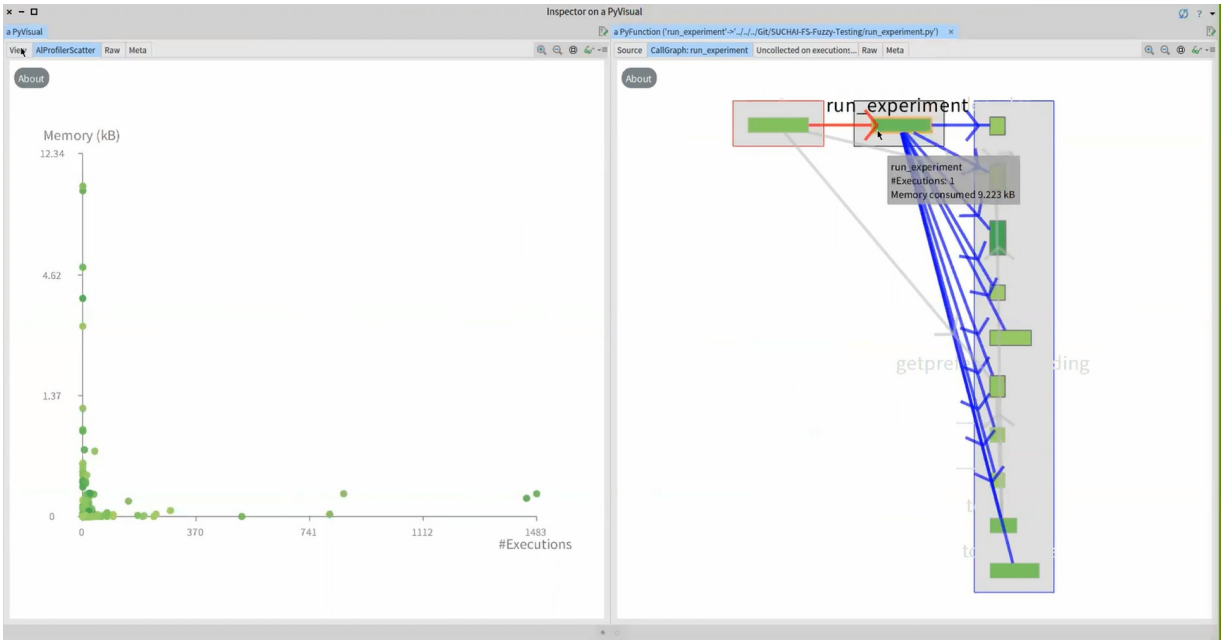


Figura 5.5: Resultado del participante P5. Entre las bibliotecas usadas están pandas, json, argparse, subprocess. Diagrama de dispersión y gráfico de sub-llamadas de *run_experiment*.

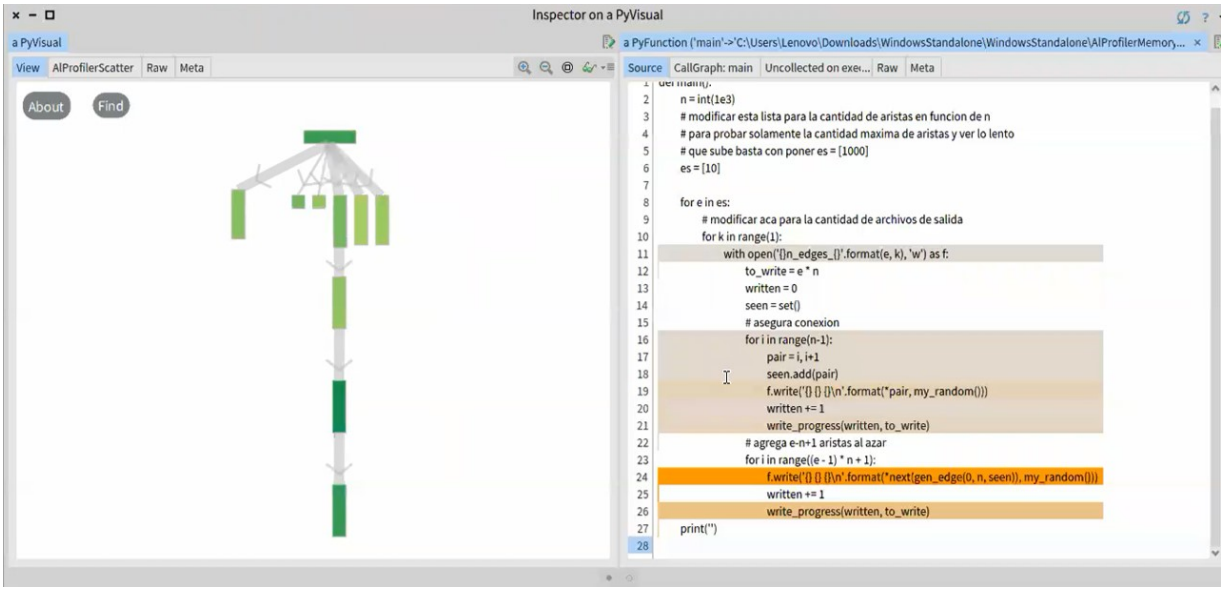
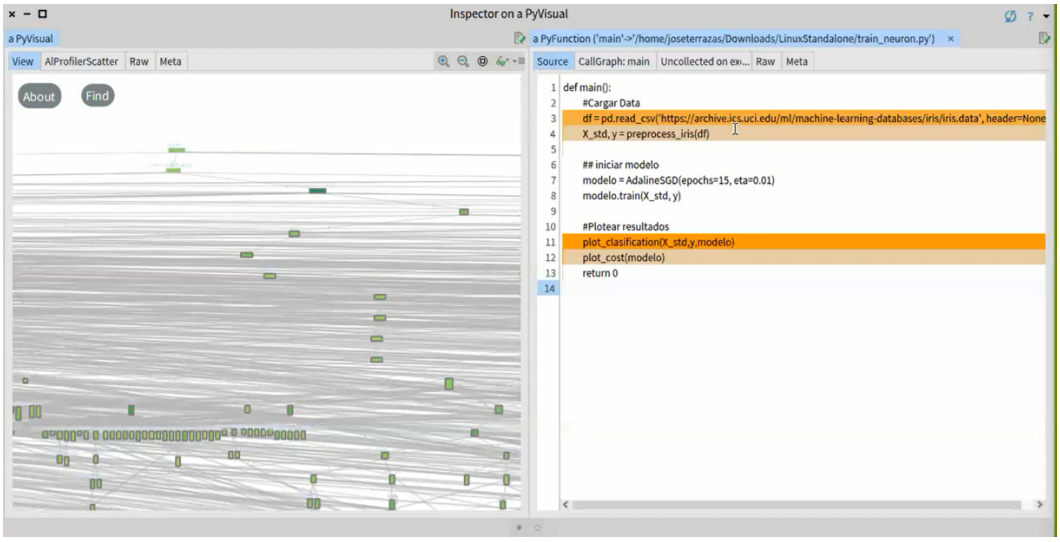
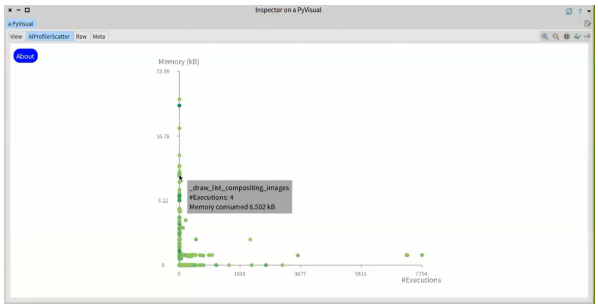


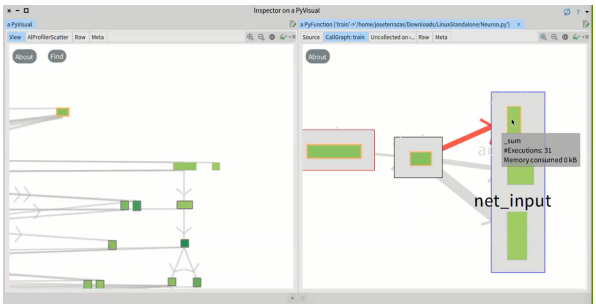
Figura 5.6: Resultado del participante P6. Vista principal y código fuente de *main*.



(a) Vista principal y código fuente de *main*.



(b) Diagrama de dispersión.



(c) Vista principal y gráfico de sub-llamadas de *train*.

Figura 5.7: Resultado del participante P7. Entre las bibliotecas usadas están numpy, matplotlib, pandas, mlxtend.

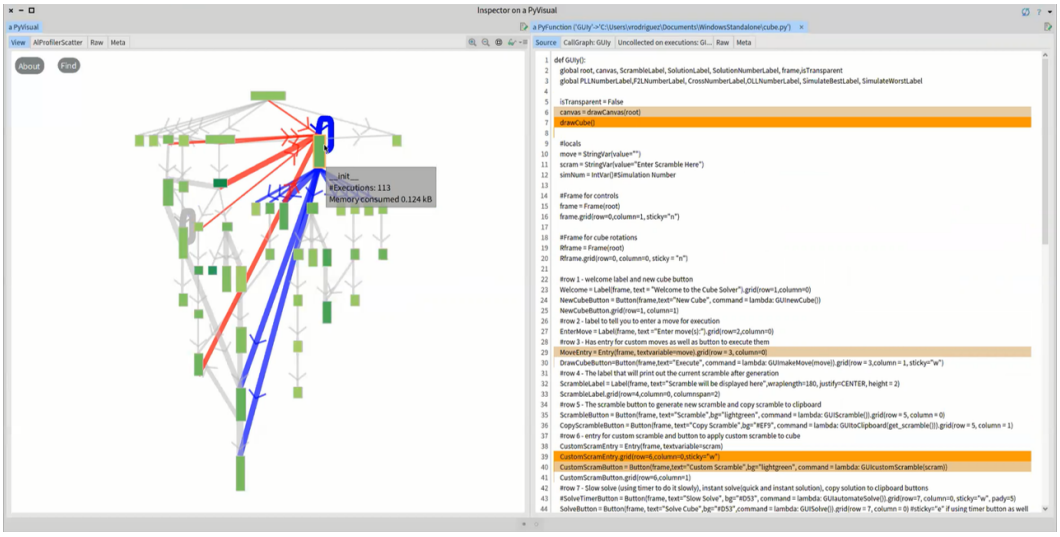


Figura 5.8: Resultado del participante P8. Entre las bibliotecas usadas están tkinter, copy, webbrowser. Vista principal y código fuente de *GUIy*.

Preguntas: Las preguntas que se presentan a continuación se dividen en las que ellos hicieron audibles (A) y las que se deducen por las acciones que realizaban (D). Resumiendo:

- **Flujo de llamadas**: Preguntas del tipo *¿qué métodos llaman a X?* (D) o *¿qué métodos son llamados por X?* (D). 6 participantes le dieron importancia, iban al gráfico de sub-llamadas de algún método y decían los métodos que llamaban o eran llamados, pasaban el mouse sobre los nodos y decían en voz alta a quienes llamaba el método, también indicaban que les llamaba la atención la gran cantidad de llamadas que tenía alguna función o que solo querían saber a qué funciones llamaba un método X.
- **Máximos**: Preguntas del tipo *¿qué método consume más memoria?* (D) y *¿qué método se ejecuta más veces?* (D). 7 participantes mencionaron los métodos que cumplían con estas características. 3 de ellos manifestaron sorpresa al respecto: a P1 le llamó la atención que el método que se ejecutaba más veces indicara que, prácticamente, no consumió memoria, al igual que P2; y a P4 le encontró sentido cuál era el método que se ejecutaba más veces, pero le sorprendió la cantidad.
- **Memoria**: Preguntas del tipo *¿cuáles métodos son más costosos en términos de consumo de memoria?* (D), o que reciben comentarios como *"estos otros consumen harta memoria también"*, no buscando el máximo necesariamente. 5 participantes hicieron comentarios al respecto, diciendo las frases ya mencionadas. En particular, a P1 le sorprendió que los métodos que dibujaban los grafos en Tarea 1.2 consumían más memoria que los que resolvían el problema de programación lineal entera.
- **Código**: Preguntas del tipo *¿qué partes del código del método X consumen memoria?* (D) y *¿cuál es el código del método X?* (D). Todos los participantes hicieron, al menos, un comentario al respecto, pero los participantes P2, P3, P4 y P6 realizaron más clicks para ver el código e indicaron en qué líneas se asignaba mayor cantidad de memoria.
- **Líneas de código**: Preguntas del tipo *¿cuál es la función con más líneas de código?* (D). Un participante mencionó cuál sería el método con más líneas de código.
- **Código en grafo**: Preguntas del tipo *¿dónde se encuentran los métodos destacados del código fuente dentro del grafo?* (D). Es decir, buscar cómo se relacionan las líneas que asignan memoria dentro del código de una función en particular con las demás funciones que se ven en el grafo de la vista principal. El último participante intentó hacer esto en su sesión.
- **Datos de los métodos**: Preguntas del tipo *¿cuántas veces se ejecuta el método X?* (A y D) y *¿cuánta memoria consume el método X?* (D). Todos los participantes hicieron comentarios al respecto, pasando el mouse sobre los nodos por un tiempo más prolongado, mencionando los valores ahí presentados, diciendo qué métodos tenían consumo de memoria o cantidad de ejecuciones similar, o preguntándose en voz alta.
- **Filtro**: Cuatro participantes hicieron comentarios respecto a preguntas del tipo *¿cuáles son mis métodos?* (D). Ellos manifestaban que querían buscar los métodos que ellos mismos escribieron, considerando que obtuvieron un grafo con muchos nodos representando a muchos más métodos de los que ellos mismos programaron.
- **Específicas**: El último participante hizo preguntas más específicas relacionadas con su código, como *"¿por qué la definición de este elemento consume memoria?"* (A), *"si yo hubiera apretado botones distintos, el árbol sería diferente ¿o no?, pero más en el número de ejecuciones y ese tipo de cosas, ¿o no?, el grafo propiamente tal no debiera ser tan distinto"* (A), *"si yo hubiera hecho otras operaciones, ¿también podría haber*

cambiado la memoria consumida?" (A), *"¿es lineal este cambio?, es decir, si yo ocupara el programa de tal manera que yo duplicara la memoria en este método, ¿en todos los demás se duplicaría la memoria ocupada?"* [indicando un método en particular] (A).

- **Herramienta**: Preguntas relacionadas con la herramienta en sí, pero que influyen en lo que ellos querían averiguar de su aplicación con la herramienta, como *"¿no se puede ver en tiempo real?"* (A), *"¿se puede guardar como para comparar árboles después?"* (A), y *"¿el método X toma la memoria que ocupó el método Y o van separados?"* (A y D). 3 participantes hicieron comentarios y acciones al respecto.

Principalmente, los participantes intentaron averiguar cuánta memoria y cantidad de ejecuciones tenían algunos métodos, cómo era el flujo de las llamadas, y en qué líneas del código de las funciones se asignaba más memoria, para ponerle más atención a esas líneas y ver su relación con el resto de la aplicación en caso de querer mejorar su código.

Retroalimentación: Todos los participantes mostraron interés por las características de la visualización e hicieron algunas sugerencias:

- **Filtro**: Dos participantes sugirieron que se dé la opción al usuario de visualizar los métodos utilizados de las bibliotecas en lugar de mostrarlas inmediatamente, puesto que les interesaría ver lo que ellos implementaron primero y, si lo encuentran necesario, analizar qué utilizaron de las componentes externas.
- **Estética**: Cuatro participantes hicieron comentarios sobre la estética de la visualización. Si bien opinaron que la visualización era aceptable, aun se puede mejorar. Por ejemplo, en la vista principal, los nombres de los métodos eran muy claros y se mezclaban al ser largos y los nodos cercanos unos de otros. Algunos cambios fueron implementados por estas sugerencias.
- **Propuestas**: Un participante mencionó que, para futuras versiones, sería interesante que el mismo programa entregue un diagnóstico o sugerencias para arreglar el código; también dijo que sería bueno que se muestre cómo varía la información en tiempo real. Otro participante también manifestó que si la herramienta fuera un *plugin* de algún IDE, lo utilizaría.

Problemas: Algunos problemas que enfrentaron los participantes en las sesiones fueron:

- **Ralentización**: Dos participantes mencionaron que era muy lento de ejecutar la herramienta. En particular, uno de ellos esperó, aproximadamente, una hora y media en esperar a que se ejecutara la herramienta sobre su código, y media hora en poder abrir la visualización en Pharo, mientras que su aplicación demoraba menos de 2 minutos en ejecutarse sola.
- **Formato de los argumentos de entrada**: Dos participantes no pudieron ejecutar el *batch/bash* al entregarles los argumentos de su aplicación, por lo que tuvieron que modificar su código para poner esos argumentos por defecto y no tener problemas al ejecutar la herramienta. En un caso, los argumentos eran flags y, en el otro, eran *paths* en formato de Strings.
- **Direcciones de archivos**: Un participante no pudo ejecutar el *bash* porque, dentro de su código, llamaba a archivos externos para leer y/o escribir, y la herramienta no los encontraba. Para poder continuar, tuvo que modificar su código, cambiando la

ubicación de esos archivos a partir de la ubicación desde donde se estaba ejecutando la herramienta. Se hicieron modificaciones para solucionar este problema.

5.1.2. Logs

Para poder estudiar los experimentos y obtener conclusiones de mejor manera, se crea un sistema de logs utilizando *Pair Programming* con la estudiante de doctorado, ya que ella conoce más a profundidad su proyecto y el lenguaje en que está escrito, el cual es, principalmente, en el lenguaje *Pharo*, inspirado en el lenguaje de programación *Smalltalk*.

Cuando se selecciona el directorio para visualizar dentro de la aplicación de Pharo, automáticamente comienza a detectar movimientos, los que se detienen al presionar el botón *End Session* en la visualización, escribiendo las acciones del usuario en un archivo CSV. Cada vez que se abre una visualización, se borra el anterior y se escribe uno nuevo.

Lo que detecta, y finalmente se escribe en el archivo, es:

- La hora y posición del mouse en ese momento.
- Si está en alguna vista dentro de la visualización (si está afuera, no detecta movimientos): *Principal* (grafo principal), *Scatter* (diagrama de dispersión de memoria), *Source code X* (código fuente) y *Call graph X* (grafo de sub-llamadas), donde X es un método seleccionado.
- El evento realizado: *Over* (por defecto, cuando no se hace click) o *Click*.
- El elemento sobre el que se realiza la acción: *Nothing* (por defecto) o el nombre del método sobre el que se hace la acción.
- En el caso de hacer click, sólo lo registra si es sobre un elemento de la vista (nodo).

Con este sistema se puede medir con más precisión el tiempo que se estuvo sobre alguna vista, ver en cuál se pasa más tiempo, los métodos más visitados, flujo seguido, entre otros.

5.1.3. Estudio de herramientas clásicas

Para poder realizar los experimentos, es necesario entender cómo funcionan o se usan las herramientas actuales para el perfilamiento de memoria para Python. Para ello, se prueban las herramientas descritas en la sección **2.1**, con 2 códigos escritos en Python 3 cada una.

En la Tabla 5.3 se indica el número de la herramienta según la sección **2.1**, en qué se enfoca en presentarle al usuario, las bibliotecas necesarias para su funcionamiento, cómo se instala, si permite mostrar información de manera gráfica, si la documentación entregada es suficiente para poder utilizar la herramienta (bajo el punto de vista de la estudiante), si hay que agregar *flags* a la línea de comando para ejecutar la herramienta, cuántas opciones de cambios en el código ofrecen para obtener información (sin contar la que importa la herramienta) y si hay similitudes entre las respuestas obtenidas de la herramienta y las de AlProfiler.

Observaciones:

- **Documentación**: Un círculo negro representan que la herramienta tiene la información necesaria en su documentación, incluyendo ejemplos. Un círculo blanco representa que

la documentación ofrece bastante información, sin embargo, aun es necesario consultar en otras páginas para obtener lo suficiente para ocupar la herramienta.

- **Flags**: Los ✓indican que se necesitan *flags* en la línea de comando.
- **Similitud**: Toma en cuenta la información que se saca en común entre la herramienta y AlProfiler. Los ✓representan que parte de la información entregada en las pruebas hechas concuerda entre ambas herramientas. Los - representan que fue complicado obtener concordancia y entender bien en la información entregada por ambas herramientas.

ID	Foco	Biblioteca extra	Inst.	Grafica	Doc	Flags	Cambios	Sim.
1	Objetos	PyDot Pyrex annotate	Descarga	Sí				
2	Objetos	Tkinter ipdb*	pip	Browser gráfico	○	-	1 - 10+	-
3	Código	psutil	pip	Memoria en el tiempo	•	✓	3	✓
4	Objetos	Tcl/Tk	pip*	Browser interactivo	•	-	10+	-
5	Objetos	-	pip	Memoria en el tiempo	•	✓	1	-
6	Código	-	-	-	•	-	10+	✓
7	Objetos	Graphviz xdot	pip	Relación entre objetos	•	-	9	-

Tabla 5.3: Perfiladores clásicos.

Para comprender más la Tabla 5.3:

- **PySizer** (1): Se descarga como un directorio comprimido. Se encontró solo para versiones de Python 2.X, por lo que no se hicieron pruebas con esta herramienta, razón por la cual desde **Doc** en adelante no están llenos.
- **Guppy** (2):
  - **Biblioteca extra**: Tkinter es para que se muestre la parte gráfica e interactiva mencionada en la tabla, no es necesaria para el resto de la funcionalidad. **ipdb** es un componente externo sugerido para obtener información; exporta funciones para acceder al depurador de IPython, que cuenta con finalización de pestañas, resaltado de sintaxis, mejores trazas y mejor introspección con la misma interfaz[1].
  - **Instalación**: Se usa `pip install guppy3` para obtener la versión correcta.
  - **Documentación**: La documentación encontrada ofrece mucha información, pero no se menciona todos los comandos que se pueden usar con `heap` (objeto dentro de la herramienta), ni cómo usar *ipdb*; sin embargo, ofrece otras páginas que pueden ayudar al usuario a detectar problemas con la memoria usando esta herramienta.
  - **Cambios**: Se puede hacer uso de una línea para activar la terminal de *ipdb* y usar el resto de las opciones de esa herramienta dentro de la terminal, u ocupar algunas

[1] `https://pypi.org/project/ipdb/`

de las más de 10 opciones dentro del mismo código. Dependiendo de si se quiere ver cambios en el consumo entre métodos o dentro de un mismo método, se puede elegir la opción más apropiada.

  - **Similitud**: En las pruebas se indicó que los objetos de Numpy, los involucrados en escrituras de archivos y Matplotlib consumen más memoria, y en AlProfiler los métodos de esas bibliotecas, o las líneas resaltadas de los códigos que incluían instrucciones respecto a ellas, influyen en gran medida en la cantidad de memoria consumida. Sin embargo, no hay datos exactos que concuerden entre ambas herramientas.

- **Memory_profiler** (3):
  - **Biblioteca extra**: Es un módulo que depende de psutil.
  - **Instalación**: Se pueden instalar con `pip install memory_profiler`.
  - **Documentación**: Tiene la información necesaria, incluyendo ejemplos.
  - **Cambios**: Ofrece un decorador con distintos argumentos para los métodos que se quieren estudiar, una línea para escribir esa información en un archivo de logs y otra para medir el uso de memoria durante un intervalo de tiempo.
  - **Similitud**: Muestra que se asigna memoria en las líneas que aparecen destacadas con naranjo más fuerte en los códigos fuentes de la visualización con AlProfiler.
- **Muppy** (4):
  - **Biblioteca extra**: Las dependencias de módulos Python externas escritas son para que se muestre la parte gráfica e interactiva mencionada en la tabla, no son necesarias para el resto de la funcionalidad.
  - **Instalación**: Se pueden instalar con la instrucción `pip install pympler`.
  - **Documentación**: Tiene la información necesaria, incluyendo ejemplos.
  - **Cambios**: Se pueden ver dentro de la misma documentación las más de 10 opciones que ofrece para modificar el código.
  - **Similitud**: En ambas pruebas indicó que se crean muchas listas y que son las que más memoria consumen, pero no se puede definir bien, en un principio, qué listas son. También es un poco confuso el árbol que crea para relacionar un objeto escogido con los demás objetos y funciones.
- **Memprof** (5):
  - **Biblioteca extra**: No necesita de más dependencias externas.
  - **Instalación**: Se pueden instalar con la instrucción `pip install memprof`.
  - **Documentación**: Tiene la información necesaria, incluyendo ejemplos.
  - **Cambios**: Ofrece un decorador con distintos argumentos para colocar sobre cada método que se quiera estudiar.
  - **Similitud**: Mostraba, en general, el nombre de los objetos con los que se les había llamado en el código, facilitando saber el tiempo en que se creaban. Sin embargo, aun es complicado asociar esa información con la entregada por AlProfiler.
- **Tracemalloc** (6):
  - **Biblioteca extra**: No necesita de más dependencias externas.
  - **Instalación**: Viene incluido dentro de Python, por lo que no requiere ser instalado.
  - **Documentación**: Tiene la información necesaria, incluyendo ejemplos.

  - **Cambios**: Se pueden ver dentro de la misma documentación las más de 10 opciones que ofrece para modificar el código.
  - **Similitud**: Algunas líneas que resultan de ocupar la herramienta aparecen destacadas en los códigos fuentes de la visualización con AlProfiler.
- **Objgraph** (7):
  - **Biblioteca extra**: Las mencionadas son para que se muestre la parte gráfica e interactiva mencionada en la tabla, no son necesarias para el resto de la funcionalidad.
  - **Instalación**: Se pueden instalar con la instrucción `pip install objgraph`.
  - **Documentación**: Tiene la información necesaria, incluyendo ejemplos.
  - **Cambios**: Se pueden ver dentro de la misma documentación las 9 opciones que ofrece para modificar el código.
  - **Similitud**: Indica qué objetos son los que consumen más memoria, pero no se puede definir bien a qué corresponden esos objetos en un principio.

En base a las pruebas hechas, se pueden notar las diferencias, en algunos casos desventajas, que tienen en comparación con AlProfiler:

Clásicas

- Es necesario modificar el código propio para ejecutar la herramienta.
- En algunos casos se necesita una biblioteca externa para ejecutar la herramienta en su completitud.
- La mayoría se centra en mostrar el consumo de memoria de los objetos. En 2 casos se muestra el consumo por líneas del código.
- Solo muestra una forma de representación, en caso de hacerlo.
- Los gráficos de los que muestran son de memoria en el tiempo y no permiten interacciones.
- Los que muestran grafos y browsers son de cómo se relacionan los objetos, los que pueden ser grandes si hay muchos objetos y la profundidad que se les dé.
- En (3) se puede ver el código fuente solo del método seleccionado, y en (6) se ven las líneas que más ocupan memoria, sean propias o externas. En los demás no se puede ver el código desde la herramienta.

AlProfiler

- El ideal es no necesitar hacer modificaciones al código propio para ejecutarse.
- El *batch/bash* descarga las bibliotecas necesarias para que el usuario no tenga que hacerlo.
- Muestra el consumo de memoria y el número de ejecuciones por métodos, también el código fuente por función, resaltando las líneas de más consumo.
- Muestra un grafo de llamadas, de sub-llamadas y un gráfico de dispersión.
- El gráfico de dispersión es de memoria versus cantidad de ejecuciones, y permite ciertas interacciones.
- Los grafos son interactivos y muestran las llamadas entre los métodos en el caso del principal, y por cada función se tiene también el grafo de sub-llamadas.
- En la herramienta se puede acceder al código fuente de cada método, tanto propio como de las dependencias externas, junto con un resaltado de las líneas que consumen memoria que varía de color según la cantidad consumida.

5.1.4. Resultados y metodología

Para explicar lo que se quiere medir u observar, se define lo siguiente:

- **Experto**: Persona con un profundo conocimiento en programación. Es capaz de relacionar una falla de software con un consumo anormal de memoria y de usar o crear un perfilador de memoria.
- **No experto**: Para efectos de esta memoria, un no experto es una persona con cierta experiencia en programación, pero que no son del área de la computación y que, por lo tanto, no tienen conocimientos profundos en programación ni en abordaje de problemas de memoria. Por ejemplo, periodistas, analistas científicos, entre otros.

Resultados

En resumen, según los pilotos realizados, la herramienta se considera útil para analizar el consumo de memoria de una aplicación; los participantes de estos entendieron la visualización, caracterizaron el consumo de memoria de sus aplicaciones y dieron ideas de cómo mejorarlas en base a lo visto en la herramienta, además de dar retroalimentación para mejorar el perfilador. Durante las sesiones, los participantes realizan preguntas, principalmente, sobre el flujo de las llamadas, cuáles son los métodos que más consumen memoria y cómo se distribuye en el código.

Además de los resultados de los pilotos, se tiene el perfilador visual AlProfiler para el consumo de memoria de aplicaciones en Python y un sistema de logs para registrar las acciones de los usuarios. Así también, se tienen los perfiladores de memoria estándar para aplicaciones hechas en Python, las cuales no proveen material interactivo, más bien texto, con opciones de crear gráficos, para realizar perfilamiento.

Metodología

Finalmente, en base a lo anterior, la metodología elegida para proceder con los experimentos es la siguiente:

- **Preguntas generales**: el participante aborda 6 preguntas sobre su experiencia general y su experiencia en analizar el consumo de memoria.
- **Tutorial**: el participante lee el tutorial y realiza consultas en caso de tenerlas.
- **Exploración**: el participante explora la herramienta con un código de ejemplo y puede hacer consultas. Luego responde 3 preguntas para verificar que entiende el tutorial: ¿cuál es el método que consume más memoria?, ¿cuántas veces se ejecuta el método X?, ¿en qué líneas del código del método Y se asigna memoria?
- **Tareas con AlProfiler**: el participante corre el *profiler* en su propio código y visualiza el resultado. A continuación, responde qué aprendió de su aplicación con esta herramienta y si ha encontrado una oportunidad de mejora en su aplicación.
- **Encuestas sobre AlProfiler**: el participante responde un formulario para medir la carga cognitiva durante la realización de la tarea y la usabilidad de la herramienta.
- **Comentarios sobre AlProfiler**: el participante da sugerencias y comentarios de manera informal y oral.

- **Tareas con perfilador clásico**: el participante corre un perfilador clásico en su propio código y visualiza el resultado. A continuación, responde las mismas preguntas de **Tareas con AlProfiler**.
- **Encuestas sobre perfilador clásico**: el participante responde el mismo formulario de **Encuestas sobre AlProfiler**.
- **Comentarios sobre perfilador clásico**: el participante da comentarios de manera informal y oral.

Se decide usar Tracemalloc de entre las herramientas clásicas si el usuario no usa alguna en particular y, en base a las pruebas realizadas y los resultados obtenidos anteriormente, Tracemalloc posee varias ventajas en relación al resto: no necesita otras dependencias para ejecutarse, no necesita instalarse por venir incluido en Python, no necesita *flags* en la línea de comandos cuando se ejecuta, la documentación es bastante completa y se puede obtener cierta información coincidente con lo que entrega AlProfiler. En caso que el usuario use un perfilador clásico, se le hace probar ambas herramientas.

Las preguntas a responder por el usuario se encuentran en la Tabla 5.4.

Pregunta	**Motivo**
¿Es la memoria una potencial amenaza para el código?	El participante se encarga de describir una idea de cuánta memoria consume la aplicación y sus expectativas sobre lo que más consume durante la ejecución de su aplicación.
¿Puede caracterizar el consumo de memoria de la aplicación?	El participante identifica la información relevante para el análisis del consumo de memoria de su aplicación.
¿Qué aprendió de la aplicación? ¿Algo sorprendente?	El participante compara lo que le entrega el perfilador con la idea que tenía, y describe anomalías o nueva información relacionada con el consumo de memoria en su aplicación.
¿Encontró una oportunidad de mejora en la aplicación?	El participante detecta lo que podría ser modificado y hace suposiciones sobre cómo disminuir el consumo de memoria de la aplicación.

Tabla 5.4: Información general de los participantes no expertos de los experimentos.

Experiencia del usuario

- **Carga cognitiva**: Para medir esta carga luego de analizar el código en cada participante se escoge una herramienta destinada para este propósito, que es *NASA Task Load Index* (NASA TLX)[2], una herramienta de evaluación subjetiva de la carga de trabajo. En este cuestionario se debe calificar la demanda mental (actividad mental y perceptiva requerida), física (actividad física requerida), temporal (presión de tiempo sentida por el ritmo de la tarea), rendimiento (éxito al realizar la tarea y satisfacción por él, considerando que mientras más bajo es el puntaje, mayor es el éxito), esfuerzo (para lograr el nivel de rendimiento llegado) y frustración (qué tan irritado, estresado y molesto se sintió durante la tarea) durante la tarea.

[2] `https://humansystems.arc.nasa.gov/groups/TLX/`

- **Usabilidad**: Para medir la usabilidad se escoge el *System Usability Scale* (SUS)[3], un cuestionario de 10 ítems con 5 opciones de respuesta, de "Totalmente de acuerdo" a "Totalmente en desacuerdo", para evaluar la facilidad de uso percibida de cada herramienta. Se elige este cuestionario por ser una herramienta válida para medir la usabilidad, riguroso, de pocas preguntas, siendo simple para los usuarios poder responderla, puede utilizarse en muestras pequeñas con resultados fiables.

5.2. Ejecución de la metodología

Se realiza una serie de experimentos para recabar comentarios de personas que programen en Python. A continuación se describe el diseño de los experimentos, los resultados, observaciones y opiniones.

5.2.1. Experimentos

Participantes

Se seleccionan 12 personas para las sesiones. De estos, 8 son del grupo no experto y 4 son del grupo experto. La información general de cada uno se encuentra en la Tablas 5.5 y 5.6, donde se muestra la edad, los años de experiencia de programación en Python, nivel de experto en un valor de 1 a 5 (de principiante a experto, respectivamente) de acuerdo a la apreciación de ellos mismos, cómo se enfrentan a un problema de consumo de memoria por parte de un código propio y si trabajan en una compañía o proyecto open source. El valor *ID* representa a cada participante para facilitar su identificación para más adelante.

Como se ve en la Tabla 5.5, respecto a cómo se enfrentan al consumo excesivo de memoria:

- N1 modifica la idea de lo que quiere resolver o medir con su aplicación y modifica su código en base a ello.
- N2 y N5 participante revisan su código, ven qué es lo que está consumiendo la memoria, como el tener procesos abiertos, estructuras pesadas o utilizar funciones innecesarias, y así tratar de evitar que en una próxima ejecución suceda lo mismo.
- N3 no ha tenido mayores problemas con el consumo de memoria pero, de tenerlos, examinaría su código y revisaría las estructuras de datos, trataría de usar algo más cercano al bajo nivel o estructuras de las cuales conozca el tamaño.
- N4 usualmente detiene la ejecución del programa y lo vuelve a correr en otro momento.
- N6 examina su código o lo divide en 4 partes para correrlas por separado.
- N7 solo trabaja con base de datos, por lo que segmenta esos datos y realiza subiteración para solucionar el problema del consumo excesivo de memoria.
- N8 cierra lo que no necesite para correr su programa y espera que termine de ejecutarse.

N1 y N3 trabajan en laboratorios, N6 y N7 en centros de investigación, mientras que el resto trabaja realiza aplicaciones en Python en contexto académico.

[3] `https://www.usability.gov/how-to-and-tools/methods/system-usability-scale.html`

ID	Edad	Grado	Años	Exp	Consumo de memoria	Trabajo
N1	27	MSc. Geología	9	2	Reevaluar cómo resolver el problema	Laboratorio
N2	28	MSc. Ing. Civil Eléctrica	4	3	Examinar su código	
N3	27	Ing. Civil Eléctrica	8	3	Examinar su código	Laboratorio
N4	26	Ing. Aeroespacial - Doctorado de Ing. Civil Eléctrica	2.5	3	Cortar la ejecución del programa	
N5	24	Ing. Civil Eléctrica	6	4	Examinar su código	
N6	29	Ing. Física	5	3	Examinar su código o dividirlo	Centro de investigación
N7	26	MSc. Ing. Civil Matemática e Industrial	3	3	Segmentar base de datos	Centro de investigación
N8	26	Ing. Civil Metalúrgica - Diplomado de Ciencias de Datos para la Gestión	1	2.5	Ejecutar solo el programa	

Tabla 5.5: Información general de los participantes no expertos de los experimentos.

ID	Edad	Grado	Años	Exp	Consumo de memoria	Trabajo
E1	27	Ing. en Computación	1.5	3	Examinar su código	Proyecto open source
E2	25	Ing. Civil en Computación	4	2.5	Examinar su código	Pharo
E3	25	Ing. Civil en Computación	5	4	Buscar en Internet	
E4	31	MSc. Ing Civil Informática	6	4	Vectorización	

Tabla 5.6: Información general de los participantes expertos de los experimentos.

Como se muestra en la Tabla 5.6, respecto a cómo se enfrentan al consumo excesivo de memoria:

- E1 revisa su código, ve qué es lo que está consumiendo la memoria, como tener procesos abiertos, estructuras pesadas o utilizar funciones innecesarias, y así tratar de evitar que vuelva a ocurrir.
- E2 no ha tenido mayores problemas con el consumo de memoria pero, de tenerlos, examinaría su código y así tratar de liberar la memoria adecuadamente.

- E3 busca en internet qué le puede estar ocasionando el problema y cómo lo puede solucionar.
- E4 ha tenido problemas con la memoria cuando trabaja con matrices grandes. En general, ha tenido poca experiencia con estos problemas en Python, y ha usado un perfilador pocas veces. Con otros lenguajes, ha usado vectorización, por ejemplo.

E1 trabaja en un proyecto opensource, E2 se desempeña en Pharo, mientras que el resto trabaja realiza aplicaciones en Python en contexto académico.

Códigos

A cada participante se le pide seleccionar un código escrito por ellos mismos para las sesiones de los experimentos. La descripción de cada uno se presenta en la Tabla 5.7. En el caso del participante N4, se usa 2 códigos, razón por la cual se repite en la tabla mencionada.

ID	Descripción
N1	Leer unos archivos de nubes de puntos y trasladarlos según 3 vectores, calcular la distancia entre los puntos, normalizarlos, obtener el promedio y error.
N2	Aplicar filtros de difuminado en una imagen, seleccionando el tamaño y el tipo de filtro.
N3	Simular sensores, pasar los resultados por un algoritmo de estimación (filtro de Kalman) y graficarlos.
N4	Calcular la trayectoria óptima de un satélite o módulo para llegar de un punto orbital a otro, considerando ciertas restricciones de un propulsor.
N4	Calcular los parámetros de un fluido al pasar por un tubo.
N5	Graficar la temperatura que hay dentro de un volumen de agua que hay dentro de un estanque.
N6	Cambiar el formato y algunos datos dentro de 12 archivos.
N7	Aplicar un modelo matemático con ejercicios de un SIMCE, cursos y notas en ese SIMCE de una base de datos.
N8	Depurar una base de datos para posteriores usos.
E1	Detección de fuego en una imagen.
E2	Predecir si corresponden a una anomalía, como un movimiento brusco, los valores de un JSON con 10 parámetros, que representan los movimientos de un vehículo.
E3	Algoritmo genético que transforma un número binario en decimal.
E4	Calcular *Simulated Annealing* (método de optimización de metaheurística), para un problema de acústica.

Tabla 5.7: Códigos de los participantes de los experimentos.

Sesiones

Una sesión involucra a un participante y su código, llevándose a cabo con la metodología que se menciona en la sección 5.1.4.

Como observación, el participante E4 utiliza como perfilador de memoria memory-profiler, la que también aparece en la sección 2.1, por lo que se le pide que use ambas herramientas durante la sesión para hacer las comparaciones.

Recolección de datos y métricas

Extracción de preguntas y eventos. Para encontrar lo que un usuario necesita saber sobre un programa hecho en Python para analizar el consumo de memoria y cómo nota esta información usando los perfiladores seleccionados, cada sesión es por videollamada y grabada, con la autorización de los participantes; se rastrean los eventos de interacciones realizadas en el entorno con el sistema de logs y se usa la técnica de pensar en voz alta.

Se enfoca en el desempeño de los participantes al momento de explorar ambos perfiladores con sus códigos para identificar las preguntas hechas por ellos y detectar sus interacciones con los perfiladores para responderlas. Para ello, se examina en detalle las grabaciones y registros de eventos de cada sesión siguiendo un método de análisis similar al estudio de Kubelka [7].

Primero, se transcribe la sesión de trabajo y se genera un archivo de texto estructurado que la resume, teniendo en cuenta el perfilador de memoria empleado. En ese archivo se presentan las potenciales preguntas que hace el participante en un rango de tiempo, se detecta cómo interactúa con la visualización y qué mecanismos de interacción se activan. Además, se agrega una descripción general de la intención del usuario basada en la pregunta y sus acciones.

La pregunta específica que cada participante trata de responder se define en función de la acción del usuario y la respuesta de la herramienta. Por ejemplo, N1 explora el diagrama de dispersión para identificar cuáles son los métodos que más consumen memoria, traduciéndose en "*¿Qué funciones consumen más memoria?*". A veces, el participante menciona directamente la pregunta, como N7, que pasa el cursor sobre la función *main* y examina los nodos conectados con flechas azules (salientes) mientras dice "*¿Qué funciones llama main?*".

Luego, se generalizan las preguntas identificando preguntas específicas similares. Por ejemplo, se reemplaza "*¿Qué funciones llama MAIN?*" con "*¿A qué funciones llama este método?*", referenciando a cualquier función en la ejecución de un programa. Posteriormente, se mapean algunas preguntas con las preguntas mencionadas por Sillito [16] y Kubelka [7] debido a la presencia de preguntas relacionadas con las encontradas en las sesiones (como las de conexiones entrantes y salientes). Por ejemplo, se usa "*¿Dónde se llama este método o tipo referenciado?*", propuesto por Sillito, en lugar de "*¿Qué funciones llaman a este método?*". Sin embargo, ya que estas sesiones están enmarcadas en el contexto específico de análisis del consumo de memoria, distinto al de los artículos, hay preguntas que no son posible mapear por las propuestas de los artículos mencionados.

Extracción de la experiencia del usuario. Para saber cómo afectan la representación de perfiles y la expertiz del usuario a la experiencia de los participantes en estas sesiones, se usan como base las siguientes métricas:

- **Carga cognitiva**: Cada participante responde a las preguntas de *NASA TLX* para evaluar su carga cognitiva después del análisis de su código. Se les pide que califiquen su demanda mental, física, temporal, rendimiento, esfuerzo y frustración.

- **Usabilidad**: Para medir la usabilidad se utiliza el SUS para evaluar la facilidad de uso percibida de cada herramienta. Por ende, se pide a los participantes que califiquen las dimensiones relacionadas con la usabilidad y la capacidad de aprendizaje.

Se recopilan las respuestas utilizando formularios en línea durante las sesiones. Como observación, se consideran los puntajes que E4 da al perfilador que el usa este informe.

5.2.2. Resultados

En la presente sección, se presentan los resultados obtenidos de los experimentos realizados. En la Figura 5.9 sale un ejemplo de la visualización obtenida por los participantes.

PREGUNTAS DE LOS PARTICIPANTES

Se identifican 33 preguntas diferentes (ver Tabla 5.8) y 372 ocurrencias de preguntas de las 12 sesiones, donde los no expertos realizan 278 ocurrencias de preguntas y 94 los expertos, detallándose éstas en la Tabla 5.9.

En la Tabla 5.9 se proporcionan las preguntas que ocurren durante cada sesión con desarrolladores expertos y no expertos usando ambos perfiladores, categorizadas según la clasificación definida por Sillito [16]. Cada grupo está formado por columnas que resumen las preguntas realizadas con el formato [total de ocurrencias (ocurrencias en AlProfiler/ocurrencias en el perfilador clásico)]. Adicionalmente, cada grupo muestra la columna T que representa el total de ocurrencias dada una pregunta o categoría y la columna N que provee el porcentaje que representa T considerando el monto total de ocurrencias en el grupo (expertos y no expertos).

De la Tabla 5.9, el promedio de preguntas entre los no expertos es de 34.75 preguntas (22.63 con AlProfiler y 12.13 con un perfilador clásico), y 23.5 en el caso de los expertos (17.5 con AlProfiler y 6 con un perfilador clásico).

También se puede ver que, con AlProfiler, las preguntas que más hacen los desarrolladores no expertos son las relacionadas a encontrar los máximos de memoria, cuánto consumen esos métodos y cómo se distribuye entre las líneas de código. A su vez, los desarrolladores expertos son parecidos, las preguntas más repetidas son para encontrar los máximos de memoria y para conocer cuánta memoria y cuántas veces se ejecutan distintos métodos de su aplicación.

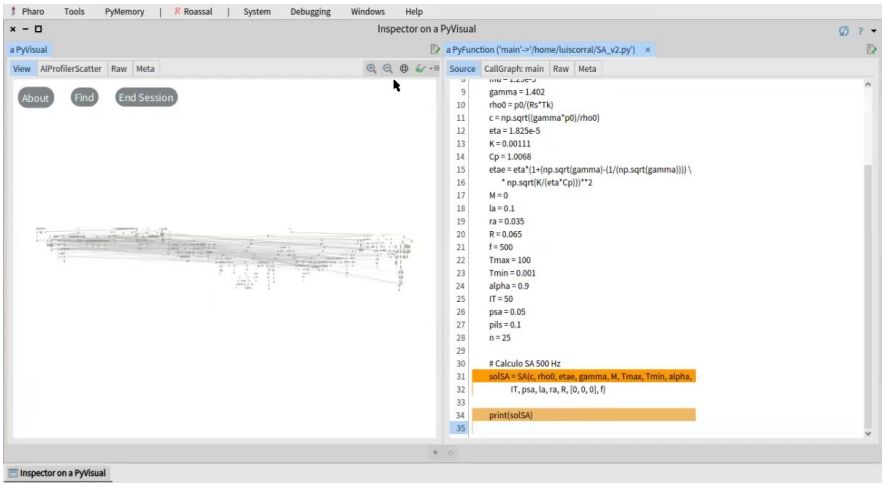
Por el contrario, con el perfilador clásico, entre las preguntas que más se repite entre los desarrolladores no expertos son cuáles son los métodos que más consumen memoria y cómo es la definición de distintos métodos. A su vez, para los expertos, cerca de la mitad del total de preguntas se enfocó en encontrar las entidades que más consumen memoria.

La Figura 5.10 ilustra la distribución de ocurrencias de preguntas hechas por los participantes con las categorías. Cabe notar que, en conjunto, las preguntas de EFP son las más frecuentes con AlProfiler y las de FFP con el perfilador clásico, mientras que las preguntas de QGS son menos frecuentes, independiente de la experiencia del usuario y la herramienta.

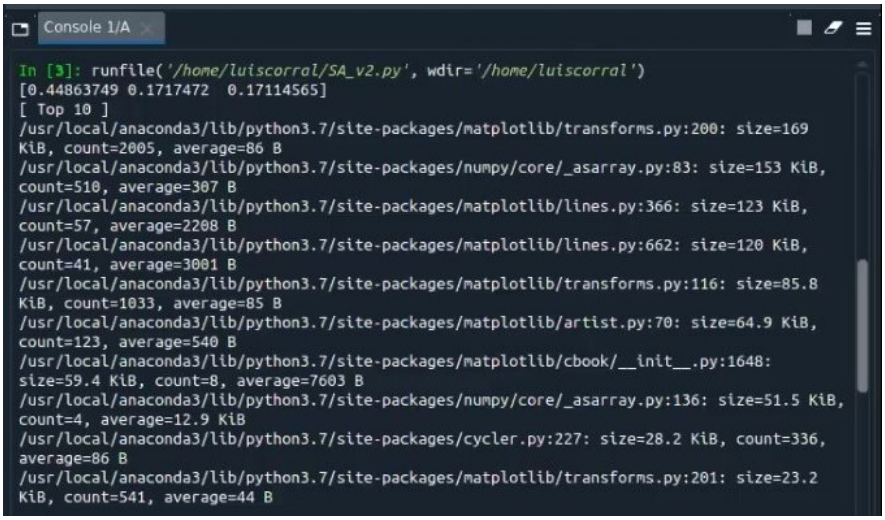
Analizando por categoría, se tiene lo siguiente:

Encontrando un punto focal (FFP)

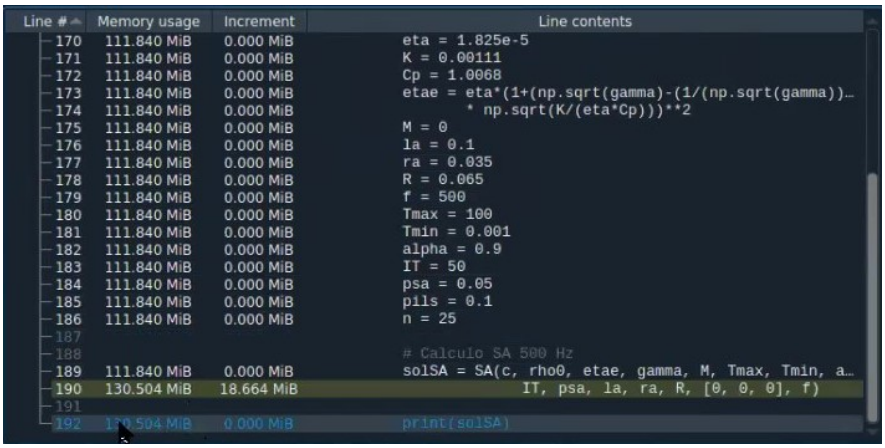
Estas preguntas suelen implicar encontrar entidades (funciones, líneas de código) como puntos de partida en función de alguna característica, como que tengan cierto comportamiento, que consuman mucha memoria, se ejecuten varias veces o que estén en cierta ubicación.



(a) Parte de la visualización con AlProfiler, en el *gráfico de llamadas* y un *código fuente*.



(b) Parte de los resultados con Tracemalloc, con el ejemplo de obtener las 10 líneas que consumen más memoria



(c) Parte de los resultados con memory-profiler.

Figura 5.9: Resultado del participante E4.

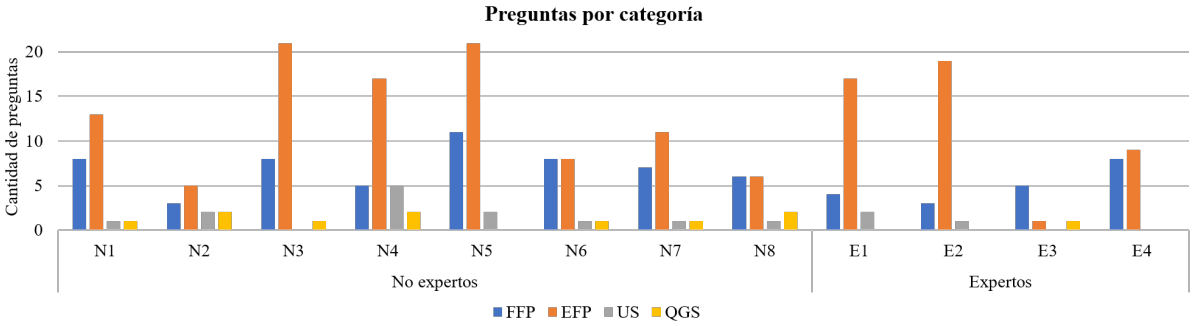
ID	Preguntas	Categoría
1	¿Qué entidades (funciones, líneas de código, instancias) están involucradas en la implementación de este comportamiento?	***Encontrado un punto focal (FFP)***
2	¿Cuáles funciones son las más ejecutadas?	
3	¿Cuáles funciones son las más llamadas por diferentes funciones?	
4	¿Qué entidades (funciones, líneas de código, instancias) asignan más memoria?	
5	¿Qué funciones contienen más líneas de código?	
6	¿Qué entidades (funciones, líneas de código, clases) pertenecen a este archivo o módulo?	
7	¿Hay una entidad denominada algo así en esa unidad (proyecto, paquete o clase, por ejemplo)?	
8	¿En qué parte del código se realizan las asignaciones de memoria en esta función?	***Expandiendo en el punto focal (EFP)***
9	¿Dónde fueron creadas las instancias de estas clases?	
10	¿Cuándo se van al garbage collector estos objetos?	
11	¿Dónde es llamado o referenciado este método?	
12	¿Cuándo durante la ejecución se llama a este método?	
13	¿Qué funciones son llamadas por esta función?	
14	¿Cuáles son las partes de esta entidad (función, instancia, tipo)?	
15	¿Cómo se ve la declaración o definición de esto?	
16	¿Qué datos son modificados en este código?	
17	¿Cuánta memoria, directa o indirectamente, consume esta entidad (función, instancia, línea de código)?	
18	¿Cuántas veces es esta entidad (función o línea de código) ejecutada?	
19	¿Qué parte (función, línea de código, instancia) de esta función consume más memoria?	
20	¿Cuántas llamadas recursivas ocurren durante esta operación?	
21	¿En qué parte del código se realizan las asignaciones de memoria en las funciones que consumen menos memoria?	
22	¿Cuánta memoria consumen juntas estas partes del código para esta función?	***Entendiendo un subgrafo (US)***
23	¿Cómo se ve esta estructura de datos en tiempo de ejecución?	
24	¿Qué ruta de ejecución se está tomando en este caso?	
25	¿En qué circunstancias se llama a este método o se lanza una excepción?	
26	¿Cuál será el impacto en el consumo de memoria de este cambio?	***Preguntas sobre grupos de subgrafos (QGS)***
27	¿Cómo se distribuyen las funciones según la memoria consumida y el número de ejecuciones?	
28	¿Cuál es la diferencia entre estas partes similares del código (por ejemplo, entre conjuntos de métodos)?	
29	¿Cuáles son las diferencias en el consumo de memoria entre este momento y ese momento?	
30	¿Qué funciones son las primeras en ser llamadas por esta función?	
31	¿Qué funciones son las últimas en ser llamadas por esta función?	
32	¿Cuál es la diferencia en el consumo de memoria entre varias ejecuciones del código?	
33	¿Cuánto es la memoria total?	

Tabla 5.8: Preguntas identificadas durante las sesiones con no expertos y expertos.

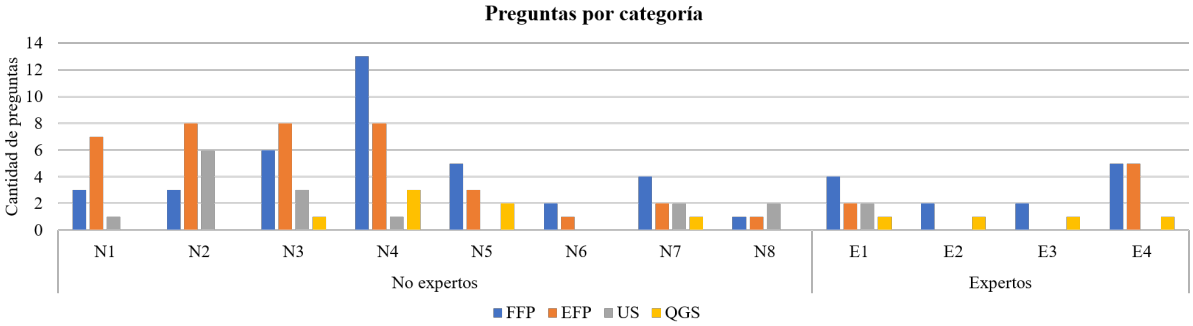
En el grupo de los no expertos se repite la sorpresa por la gran cantidad de funciones llamadas en la visualización, muchas desconocidas para ellos. Esto es debido a que, generalmente, ellos no consideran el impacto de las bibliotecas externas (como numpy, pandas o matplotlib). En consecuencia, las preguntas de FFP indican puntos de partida durante el análisis del consumo de memoria.

Preguntas por categoría	No expertos										Expertos					
	N1	**N2**	**N3**	**N4**	**N5**	**N6**	**N7**	**N8**	**T**	**N [%]**	**E1**	**E2**	**E3**	**E4**	**T**	**N [%]**
1	1 (1/0)	2 (1/1)	2 (2/0)	4 (1/3)	3 (2/1)	1 (1/0)	7 (5/2)	1 (1/0)	21 (14/7)	7,6 (7,7/7,2)	2 (2/0)	2 (2/0)	3 (3/0)		7 (7/0)	7,3 (10/0)
2	1 (1/0)	1 (1/0)	2 (2/0)	1 (1/0)	3 (3/0)	1 (1/0)			9 (9/0)	3,2 (5/0)	1 (1/0)	1 (1/0)		1 (1/0)	3 (3/0)	3,1 (4,3/0)
3					1 (1/0)				1 (1/0)	0,4 (0,6/0)					0	0
4	8 (5/3)	2 (1/1)	6 (2/4)	8 (2/6)	6 (4/2)	3 (2/1)	3 (1/2)	2 (2/0)	38 (19/19)	13,7 (10,5/19,6)	5 (1/4)	2 (0/2)	3 (1/2)	10 (6/4)	20 (8/12)	20,8 (11,4/46,2)
5			1 (1/0)					2 (2/0)	3 (3/0)	1,1 (1,7/0)			1 (1/0)		1 (1/0)	1 (1,4/0)
6	1 (1/0)	1 (0/1)	2 (0/2)	4 (0/4)	2 (0/2)	3 (2/1)	1 (1/0)	2 (1/0)	16 (5/11)	5,8 (2,8/11,3)				2 (1/1)	2 (1/1)	2,1 (1,4/3,8)
7			1 (1/0)	1 (1/0)	1 (1/0)	2 (2/0)			5 (5/0)	1,8 (2,8/0)					0	0
Total en esta categoría (FFP)	11 (8/3)	6 (3/3)	14 (8/6)	18 (5/13)	16 (11/5)	10 (8/2)	11 (7/4)	7 (6/0)	93 (56/37)	33,5 (30,9/38,1)	8 (4/4)	5 (3/2)	7 (5/2)	13 (8/5)	33 (20/13)	34,4 (28,6/50)
8	9 (9/0)	1 (1/0)	1 (1/0)	1 (1/0)	2 (2/0)		2 (2/0)	2 (2/0)	18 (18/0)	6,5 (9,9/0)	4 (4/0)				4 (4/0)	4,2 (5,7/0)
9			1 (1/0)	1 (1/0)					2 (2/0)	0,7 (1,1/0)					0	0
10				3 (2/1)			1 (0/1)		4 (2/2)	1,4 (1,1/2,1)					0	0
11		1 (1/0)	3 (2/1)	1 (1/0)	2 (2/0)		2 (2/0)		9 (8/1)	3,2 (4,4/1)	3 (2/1)	1 (1/0)		1 (1/0)	5 (4/1)	5,2 (5,7/3,8)
12	1 (0/1)			2 (0/2)		1 (0/1)			4 (0/4)	1,4 (0/4,1)					0	0
13			2 (2/0)	3 (3/0)	3 (3/0)		2 (2/0)	1 (1/0)	11 (11/0)	4 (6,1/0)	2 (2/0)	3 (3/0)		1 (1/0)	6 (6/0)	6,3 (8,6/0)
14									0	0				2 (2/0)	2 (2/0)	2,1 (2,9/0)
15	6 (1/5)	8 (2/6)	8 (1/7)	7 (4/3)	3 (2/1)	3 (1/2)	1 (1/0)	2 (2/0)	38 (16/22)	13,7 (8,8/22,7)	2 (2/0)			1 (1/0)	3 (3/0)	3,1 (4,3/0)
16			1 (1/0)	1 (1/0)	1 (1/0)	1 (1/0)	1 (1/0)	1 (1/0)	6 (6/0)	2,2 (3,3/0)	1 (1/0)				1 (1/0)	1 (1,4/0)
17	2 (2/0)	1 (1/0)	5 (5/0)	4 (2/2)	8 (6/2)	3 (3/0)	2 (1/1)	1 (0/1)	26 (20/6)	9,4 (11/6,2)	4 (3/1)	6 (6/0)		2 (0/2)	12 (9/3)	12,5 (12,9/11,5)
18	1 (0/1)	2 (0/2)	8 (8/0)	1 (1/0)	5 (5/0)	1 (1/0)	2 (2/0)		20 (17/3)	7,2 (9,4/3,1)	3 (3/0)	6 (6/0)	1 (1/0)	6 (3/3)	16 (13/3)	16,7 (18,6/11,5)
19									0	0		3 (3/0)		1 (1/0)	4 (4/0)	4,2 (5,7/0)
20				1 (1/0)					1 (1/0)	0,4 (0,6/0)					0	0
21	1 (1/0)								1 (1/0)	0,4 (0,6/0)					0	0
Total en esta categoría (EFP)	20 (13/7)	13 (5/8)	29 (21/8)	25 (17/8)	24 (21/3)	9 (8/1)	13 (11/2)	7 (6/1)	140 (102/38)	50,4 (56,4/39,2)	19 (17/2)	19 (19/0)	1 (1/0)	14 (9/5)	51 (46/5)	55,2 (65,7/26,9)
22	2 (1/1)	4 (0/4)	3 (0/3)	1 (1/0)	1 (1/0)		1 (0/1)	1 (0/1)	13 (3/10)	4,7 (1,7/10,3)	1 (0/1)				1 (0/1)	1 (0/3,8)
23				1 (1/0)					1 (1/0)	0,4 (0,6/0)					0	0
24		3 (1/2)		1 (1/0)	1 (1/0)		1 (0/1)	1 (0/1)	7 (3/4)	2,5 (1,7/4,1)	1 (0/1)	1 (1/0)			2 (1/1)	2,1 (1,4/3,8)
25		1 (1/0)		3 (2/1)		1 (1/0)	1 (1/0)	1 (1/0)	7 (6/1)	2,5 (3,3/1)	2 (2/0)				2 (2/0)	2,1 (2,9/0)
Total en esta categoría (US)	2 (1/1)	8 (2/6)	3 (0/3)	6 (5/1)	2 (2/0)	1 (1/0)	3 (1/2)	3 (1/2)	28 (13/15)	10,1 (7,2/15,5)	4 (2/2)	1 (1/0)	0	0	5 (3/2)	5,2 (4,3/7,7)
26		1 (1/0)		1 (0/1)	1 (0/1)				3 (1/2)	1,1 (0,6/2,1)					0	0
27	1 (1/0)		1 (1/0)	1 (1/0)		1 (1/0)			4 (4/0)	1,4 (2,2/0)			1 (1/0)		1 (1/0)	1 (1,4/0)
28		1 (1/0)		1 (1/0)			1 (1/0)		3 (3/0)	1,1 (1,7/0)					0	0
29			1 (0/1)	2 (0/2)	1 (0/1)		1 (0/1)		5 (0/5)	1,8 (0/5,2)	1 (0/1)				1 (0/1)	1 (0/3,8)
30								1 (1/0)	1 (1/0)	0,4 (0,6/0)					0	0
31								1 (1/0)	1 (1/0)	0,4 (0,6/0)					0	0
32									0	0			1 (0/1)		1 (0/1)	1 (0/3,8)
33									0	0		1 (0/1)		1 (0/1)	2 (0/2)	2,1 (0/7,7)
Total en esta categoría (QGS)	1 (1/0)	2 (2/0)	2 (1/1)	5 (2/3)	2 (0/2)	1 (1/0)	2 (1/1)	2 (2/0)	17 (10/7)	6,1 (5,5/7,2)	1 (0/1)	1 (0/1)	2 (1/1)	1 (0/1)	5 (1/4)	5,2 (1,4/15,4)
Total	34 (23/11)	29 (12/17)	48 (30/18)	54 (29/25)	44 (34/10)	21 (18/3)	29 (20/9)	19 (15/4)	278 (181/97)	100	32 (23/9)	26 (23/3)	10 (7/3)	28 (17/11)	94 (70/24)	100

Tabla 5.9: Ocurrencias de las preguntas de la Tabla 5.8 en formato [total de ocurrencias (ocurrencias en AlProfiler/ocurrencias en perfilador clásico)]. Siendo los grupos experto y no experto, N es [porcentaje del total del grupo con ambas herramientas (porcentaje con AlProfiler/porcentaje con un perfilador clásico)], se calcula como T de la pregunta dividido en el total de preguntas del grupo.



(a) AlProfiler.



(b) Perfilador clásico.

Figura 5.10: Total de preguntas por categoría al usar los perfiladores por ambos grupos.

Algunos participantes buscan métodos por su funcionalidad y examinan la memoria consumida por ellos. Por ejemplo, N5 menciona su enfoque en sus funciones auxiliares: "*Aquí estos me quedaron medio juntos, así que los voy a separar. Estas son las [...] 6 funciones auxiliares que tenía, los voy a ver*".

Otros buscan las funciones que consumen mucha memoria o que se ejecutan varias veces y las comparan con sus supuestos. Por ejemplo, E1 detecta las líneas de código que asignan más memoria en la función principal: "*Está usando gran cantidad al leer archivo, al detectar fuego y al mostrarlo. Pero esto debe ser, me imagino, dependiendo de cuánto tiempo voy a mostrarlo*", o E4 que se percata que las funciones que consumen más memoria son las asociadas a crear gráficos: "*Entonces claro, lo que más consume memoria son la visualización de los gráficos, porque la función objetivo que tengo se ejecuta varias veces, pero son unas matrices chiquititas, entonces no consumen tanta memoria*". Por otro lado, N6 se sorprende al ver algunas de las funciones que consumen bastante memoria: "*Bueno, ahí veo que esta función [...] ocupa igual harta memoria, siendo que es algo súper simple lo que hace, pero puede que ocupe harta memoria por la cantidad de veces que la utilizo. Pero género me acuerdo que no la utilizo tanto, lo uso como 2 veces, ¿por qué? [...]*".

Independiente de la experiencia del participante, las preguntas más comunes en esta categoría son la 1 y 4 en el caso de AlProfiler, y solo la 4 en el caso del perfilador clásico.

Expandiendo en el punto focal (EFP)

Estas preguntas implican obtener información más detallada de una entidad dada, generalmente explorando sus características o relaciones. Estas preguntas se responden explorando el código fuente de una entidad, observando la línea de código y la cantidad de memoria asignada por ella, y considerando cuántas veces se ejecuta y memoria consumida por la entidad.

Por lo tanto, la mayoría de los participantes revisan el código fuente de algunas funciones para identificar las líneas de código que asignan más memoria. Por ejemplo, N5, que dice: "*Acá me gustaría ver, por ejemplo, cuánto consume esta línea*", o N2, que navega sobre el código fuente de una de sus funciones para detectar los sitios de asignación de memoria: "*Acá sale, por ejemplo, que esta es la que usa más memoria [...], que es cuando se aplica el [...] filtro mediano, ocupa más memoria que al ocupar un blur normal*".

Además, la mayoría de los participantes examinan las relaciones de llamadas. Por ejemplo, N4 dice "*Estoy viendo cuáles son las funciones que llama el main*". O también ven la cantidad de memoria consumida y ejecuciones de algún método en particular, por ejemplo, N3 ve cuántas veces se ejecuta un método y lo asocia con los métodos que lo llaman: "*Se ejecuta ¡2100! Ya, eso me llama la atención igual por qué se ejecuta tantas veces, pero, es probablemente porque lo utilizan más de alguna función [...] Claro, justamente se utiliza desde predict_nominal y desde [...]*".

Con AlProfiler, la pregunta más común para los desarrolladores expertos es la 18, mientras que para los no expertos son la 8 y 17. Con el perfilador clásico, la pregunta 15 es la más común entre los no expertos, pero son muy escasas para los expertos.

Entendiendo un subgrafo (US)

Estas preguntas se enfocan sobre múltiples entidades y relaciones, como identificar la cantidad de memoria que algunas partes del código consumen juntas, comprender por qué se llaman algunas funciones y rastrear la ruta de ejecución para algunos casos. Los participantes hacen preguntas de esta categoría para confirmar los supuestos sobre la ejecución de sus aplicaciones y para comprender nueva información sobre grupos específicos de entidades.

Varios participantes analizan las razones de ciertos fenómenos relacionados con el número de ejecuciones o memoria consumida por algunas entidades. Por ejemplo, N4 se centra en analizar un grupo de funciones encargadas de calcular unos parámetros: "*Me sorprendió que, para ser un método que va guardando datos en cada una de esas iteraciones consume, a mí parecer, poca memoria. Debe ser donde no estoy guardando todos los datos que calcula entre medio, sino que son variables que se crean dentro de la función y después se eliminan*". Además, N7 confirma las características de un grupo de funciones: *"Estas son las funciones objetivo que yo definí, f1 y f2, dependiendo si son para el grupo de tratamiento o control [...], se ejecutan 326, 63, y el otro debiera ser la suma. Efectivamente, la suma de ejecuciones*".

Independiente de la experiencia del participante, la pregunta más común en esta categoría es la 25 con AlProfiler. Por otro lado, con el perfilador clásico, la pregunta más común es la 22 para los no expertos, mientras que para los expertos, estas preguntas son muy escasas.

Preguntas sobre subgrafos (QGS)

Esta categoría es la menos frecuente y presenta preguntas sobre grupos de entidades que implican, frecuentemente, analizar las diferencias entre grupos.

Varios participantes se muestran interesados en detectar qué cambios en la memoria se realizaron después o antes de ciertas operaciones. Por ejemplo, N2 comenta sobre el impacto de los cambios al usar su aplicación: "*El blurring, es el que se ejecutó más veces y también dependía de cuántas veces movía el trackbar, o sea, si yo cambiaba esto, lo más probable es que hubiesen más ejecuciones, pero la memoria hubiese sido la misma, ¿o no?* ".

Además, cuatro no expertos y un experto también ponen su atención en cómo se distribuyen las funciones respecto a la memoria y las ejecuciones, por ejemplo, métodos que ubican la mayor parte de la memoria pero se ejecutan menos veces en comparación con el resto. Además, tres no expertos compararon partes del código con el fin de identificar las oportunidades de optimización, por ejemplo, N4 se pregunta sobre la diferencia entre dos líneas que cumplen funciones similares en el código: "*Todas estas cosas que están acá tienen el mismo largo, entonces eso me llama la atención de por qué, si tienen el mismo largo, tengo menos peso acá que acá*". Algunos no expertos proponen algunos cambios en el código o la entrada para analizar las diferencias en la memoria consumida por su aplicación.

En conjunto, la mayor cantidad de estas preguntas son hechas por no expertos (entre 1 y 5 ocurrencias) siendo más con AlProfiler, mientras que para el desarrollador experto este tipo de preguntas son muy escasos (entre 1 y 2 ocurrencias), siendo más en el perfilador clásico.

EMPLEO DE LA HERRAMIENTA

Las siguientes secciones proporcionan información más detallada basada en el generador de perfiles de memoria empleado, comenzando por AlProfiler y continuando con el perfilador de memoria clásico utilizado.

AlProfiler

Durante la realización de las **Tareas con AlProfiler**, se identifican 29 preguntas diferentes y 251 ocurrencias de preguntas. De esas, 76 preguntas son de FFP y 148 de EFP, siendo estas categorías las más populares. Por otro lado, 16 son de US Y 11 de QGS. Las preguntas más frecuentes son la 1, 4, 8, 17 y 18 de la Tabla 5.9.

Considerando 4 vistas de la herramienta (*gráfico de llamada* o principal, *gráfico de dispersión* o scatter, *gráfico de subllamada* o subcallgraph, y *código fuente* o source) descritas en el Capítulo 2.4, en la Figura 5.11 se muestra el porcentaje del tiempo pasado por el participante realizando eventos (pasando el cursor sobre los distintos elementos o haciendo click) en cada vista. De estos eventos, para desarrolladores no expertos, el 74 % se realiza sobre el *gráfico de llamadas* y el 11 % sobre el *gráfico de dispersión*. Por otro lado, para los desarrolladores expertos, el 65 % de estos eventos se realiza sobre el *gráfico de llamadas*, seguido del 24 % sobre el *gráfico de subllamadas*.

Con base en los datos recopilados durante las sesiones, se notan acciones en común para responder las preguntas identificadas en función de su categoría:

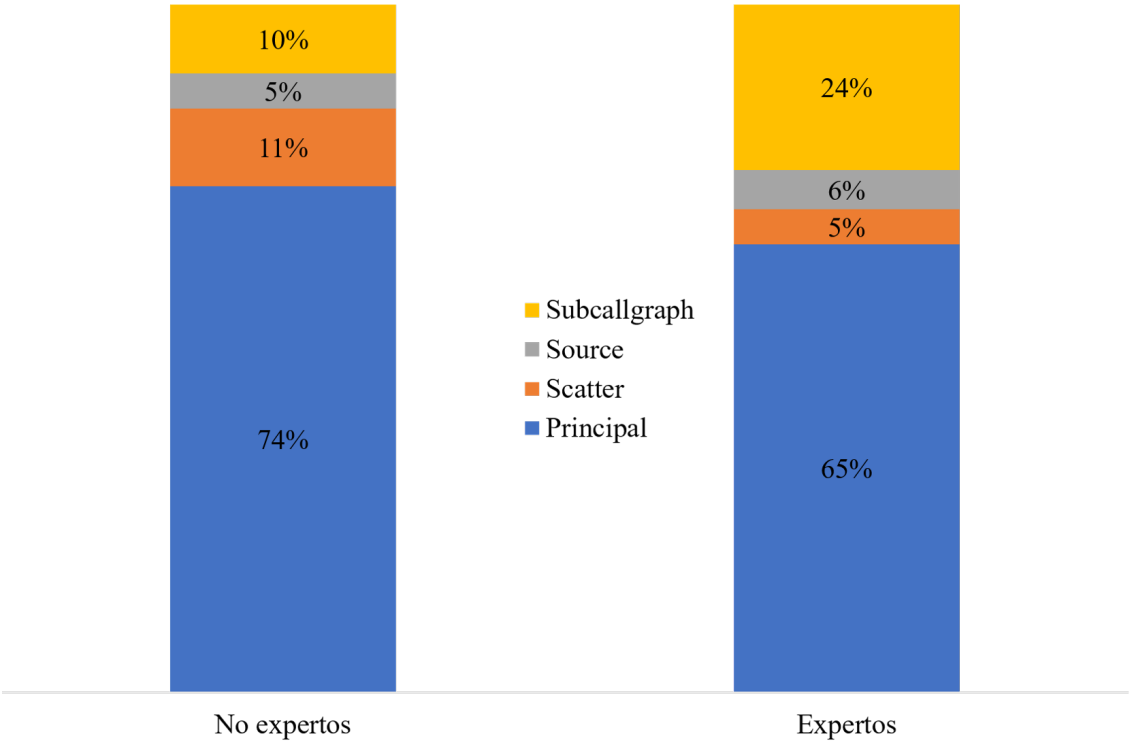


Figura 5.11: Porcentaje de tiempo pasado en las distintas vistas de AlProfiler realizando acciones por la experiencia de usuario.

- **FFP**. Para responder a estas preguntas, los participantes generalmente se enfocan en explorar el *gráfico de llamadas* para detectar entidades relevantes. Los participantes buscan y exploran los nodos (funciones) pasando el cursor sobre ellos y comprobando sus características (nombre, cantidad de memoria consumida, número de ejecuciones). Además, 8 participantes (2 expertos y 6 no expertos) exploran la vista del *diagrama de dispersión* para responder rápidamente preguntas de esta categoría. Por otro lado, 3 participantes utilizaron el botón de búsqueda en la vista del *gráfico de llamadas* para responder la pregunta 7.
- **EFP**. En su mayoría, los participantes exploraron la vista del *código fuente* para entender la definición de algunas funciones, identificar los datos modificados y las asignaciones de memoria realizadas. La vista de *gráfico de llamadas* se emplea con frecuencia para identificar las relaciones de llamadas entre funciones. Sin embargo, N5, E1, E2 y E4 generalmente seleccionan la vista de *gráfico de subllamada* para responder las mismas preguntas sobre un grupo específico de funciones. Además, los participantes usaron regularmente el *gráfico de llamadas* y el *diagrama de dispersión* para responder las preguntas 17 y 18. En esta fase, N4 realiza la pregunta "*¿cuándo se liberan de la memoria estas instancias?*", relacionada con la pregunta 10, pero esta herramienta no proporciona esta información para ayudar al participante a responder esta pregunta.
- **US**. Las preguntas relacionadas con la comprensión del rastreo de ejecución y las razones para llamar a alguna función se responden explorando las relaciones en un *gráfico de llamadas* y examinando el *código fuente*. Los participantes estiman la cantidad de memoria consumida por algunas partes del código examinando las líneas resaltadas en

la vista del *código fuente* y asociándolas con los métodos a los que está conectada la función en el *gráfico de llamadas o subllamadas*. Un participante menciona la pregunta 23, sin embargo, la información requerida no se encuentra en las métricas del perfilador.

- **QGS**. Los participantes seleccionan la vista del *diagrama de dispersión* para responder a la pregunta 27 al describir y examinar la distribución de los grupos, principalmente en los extremos del diagrama. Para identificar las diferencias entre las partes del código, los participantes utilizaron la vista del *código fuente*. Un participante examina y compara grupos de funciones en la vista del *gráfico de llamadas* para responder las preguntas 30 y 31. El participante que hizo la pregunta 26 no pudo responderla porque tendría que modificar el código y correr el perfilador nuevamente.

Respecto a la pregunta de si ven una oportunidad de mejora en su aplicación, responden:

- Tres que no directamente (2 expertos y 1 no experto), pues utilizan los métodos que necesitan por defecto de las bibliotecas y no ven mayores problemas en su código. Tal vez podrían cambiar la dependencia externa.
- Cinco participantes (4 no expertos y 1 experto) responden cómo mejorarían su código, por ejemplo, N3 dice que trataría de llamar de forma más "inteligente" a los métodos que se están llamando más y ejecutar en distintos threads las ramas que están separadas; N4 menciona que podría subir algún dato, que se está calculando más de una vez, de nivel y que, cuando se calcule, pasarlo a cada clase en lugar que cada clase lo tenga que recalcular; N6 dice que podría cambiar ciclos for por un map; N8 que arreglaría la estructura de su código para realizar las acciones dentro de él menos veces; E4 disminuiría la cantidad de veces que realiza gráficos; y N7 dice que puede concluir algunas cosas de lo que ve con la herramienta, dándose cuenta que algunos métodos no se están ejecutando la cantidad de veces que debería, lo que puede ser señal de que debe arreglar cómo modeló el problema.
- El resto responde cómo usarían la herramienta para ver si pueden mejorar su aplicación o no, como usar la información entregada, reducir llamadas o estudiar mejor las que consumen más memoria para ver cómo optimizarlas.

Retroalimentación: Todos los participantes muestran interés por las características de la visualización, por lo que realizan algunas sugerencias y comentarios:

- **Visión general**: Todos los participantes comentan lo útil que les parece la herramienta, el aporte que son las características que ofrece, es amigable, y que con ella podrían sacar varias conclusiones de sus aplicaciones que antes no se habían cuestionado.
- **Grafo en la vista principal**: Cinco participantes consideran que ver un grafo tan grande al principio no es tan intuitivo y puede desanimar a usuarios no acostumbrados a usar perfiladores de memoria. Tres participantes sugieren que se dé la opción al usuario de visualizar los métodos utilizados de las bibliotecas en lugar de mostrarlas inmediatamente, puesto que les interesaría ver lo que ellos implementaron primeramente y, si lo encuentran necesario, analizar qué utilizaron de las dependencias externas. Un participante sugiere estirar la malla para que no se sobrecargue la visualización o agregar cierta arquitectura a esta para los nodos que reciben muchas llamadas.
- **Estética**: Cinco participantes hacen comentarios sobre la estética de la visualización, que siendo aceptable, aun se puede mejorar. Por ejemplo, cambiar el color de las letras

para que destaquen más, relación de tamaño entre nodos podría ser más marcada, que los puntos del gráfico de dispersión achiquen a medida que se haga zoom, entre otros.

- **Avances**: Dos participantes sugieren colocar mensajes de avances en la ejecución de la aplicación para ver cuánto lleva avanzado, y un mensaje de carga cuando se abre la visualización en Pharo, en caso de demorarse más de lo esperado.
- **Propuestas**: Los participantes proponen tener una línea temporal para ver cuánta memoria se consume por cada instante en la ejecución (2 participantes), tener también la memoria que consume cada línea de código en lugar de solo el color (1 participante), indicar cuánto tiempo están consumiendo memoria las instancias de la aplicación (1 participante), dar la posibilidad de unir nodos o ver la relación entre ramas separadas del grafo principal (1 participante) y explotar más el gráfico de dispersión agregando otras alternativas de gráficas (1 participante).

Problemas: Algunos problemas que se presentaron en las sesiones fueron:

- **Ralentización**: Cuatro participantes tuvieron que esperar más de 1 hora en ejecutar la herramienta sobre su código, mientras que su aplicación demoraba menos de 2 minutos en ejecutarse sola. A 5 participantes se les congelaba la imagen en Pharo en ciertos momentos de la visualización.
- **Argumentos de entrada**: Tres participantes tuvieron problemas con los argumentos de entrada que debían entregar al *batch/bash* para ejecutar su aplicación, pero se pudo encontrar solución después.
- **Falta de main**: Varios participantes no tenían una función principal en su código, pero lo tenían estructurado de tal manera que no fue complicado agregarle este método, a excepción de uno, quién tuvo que dejar parte de su código fuera del perfilamiento para poder ejecutar correctamente su código.

Perfilador clásico

Se detectan 16 preguntas diferentes y 123 ocurrencias de preguntas. En esta fase, los participantes hicieron 4 nuevas preguntas, la 12, 29, 32 y 33 de la Tabla 5.9. Las preguntas de FFP y EFP son las más populares, con 50 y 45 ocurrencias, respectivamente. En contraste con la fase anterior, las ocurrencias de preguntas de EFP no duplican las preguntas de FFP, si no que son menos. Las preguntas con más ocurrencias son la 4 y 15 de la Tabla 5.9.

Con base en los datos recopilados durante las sesiones, los participantes usan los ejemplos provistos en la documentación de la herramienta de Tracemalloc. Se notan acciones en común para responder las preguntas identificadas en función de su categoría:

- **FFP**. Para responder a estas preguntas, los participantes se enfocan en explorar los resultados obtenidos de probar los ejemplos que entregan las 10 o más líneas de código que consumen más memoria. Como casos particulares, E1 y N3 también usan los ejemplos para obtener la diferencia de memoria entre dos momentos de la aplicación y el de rastreo del bloque de memoria más grande para responder la pregunta 4, N7 también usa el ejemplo de obtener la diferencia de memoria entre dos momentos de la aplicación para responder la pregunta 1, N8 y E4 usan también el ejemplo del de rastreo del bloque de memoria más grande para responder las preguntas 4 y 6. Además,

usando memory-profiler, E4 detecta las líneas de la función principal que consumen más memoria.

- **EFP**. Los participantes se enfocan en explorar los resultados obtenidos de probar los ejemplos que entregan las 10 o más líneas de código que consumen más memoria, viendo los resultados entregados en la terminal o yendo a la línea que ésta muestra en el código para responder, por ejemplo, la pregunta 16. Como casos particulares, N3 va al código luego de usar el ejemplo del rastreo del bloque de memoria más grande, N7 usa el ejemplo de la diferencia en el consumo de memoria entre distintos momentos para responder la pregunta 9 y el de rastro del bloque de memoria para la 16, así como N8; y en el caso de memory-profiler, E4 puede responder la pregunta 17, pero no la 18: "*Yo creo que es muy similar a los valores generados en el otro perfilador, pero aquí no aparece la cantidad de iteraciones. Eso tiene como desventajas éste*". Si bien durante la parte de **Tareas con AlProfiler** no formulan todas las preguntas de esta categoría, en las otras partes de la sesión los participantes expresan que hay más información que les interesaría saber pero que Tracemalloc no entrega, por ejemplo, N4 expone: "*De hecho, lo bueno que tiene es que uno aprieta aquí y te lleva al código y la línea, y te muestra el tamaño, pero no te muestra ninguna información más, de dónde se llamó a éste, después de ese qué viene, por qué es tan pesado [...]*".
- **US**. La pregunta 22 se responden usando los valores de los resultados de los ejemplos aplicados al código del participante y yendo al código de acuerdo a las líneas indicadas en la terminal como resultado de aplicar el perfilador a la herramienta. La pregunta 24 se responde usando el ejemplo del rastreo de bloques de memoria. La 25 se responde usando el ejemplo de la diferencia de memoria entre dos momentos de la aplicación, eligiendo la sección a medir la diferencia y comparando con supuestos, como N7 que dice "*Efectivamente, [...] claramente estoy llamándolo menos porque ya casi no lo uso, ya cargué la base de datos, ahora estoy llamando las copias de esa base de datos, lo mismo con todas las otras que eran para crear la base de datos, era de esperarse, porque estoy optimizando, no estoy cargando datos.*"
- **QGS**. Para la pregunta 29, E1, N3, N5 y N7 hacen uso del ejemplo que permite ver la diferencia de consumo de memoria entre dos momentos de la aplicación, mientras que N4 usa el ejemplo de obtener las líneas de código que más consumen memoria cambiando los métodos en los que escribe las instrucciones necesarias, respondiendo así también la pregunta 26, como N5. Para responder la pregunta 33, E2 hace uso del ejemplo de registrar el tamaño actual y máximo de todos los bloques de memoria, permitiéndole saber la memoria total, mientras que E4 usa memory-profiler para obtener la memoria por línea y así poder hacer algún cálculo. Para responder la pregunta 32, E3 hace uso del mismo ejemplo que E2, corriendo 2 veces su aplicación para ver si había alguna diferencia en la cantidad de memoria consumida: "*las dos veces que ejecutó el método, esto depende de números aleatorios, entonces no tengo como seguridad que siempre va a ocupar la misma cantidad memoria, de hecho siempre debería ocupar distinta, pero los resultados entre las dos ejecuciones dan bien parecidos*".

Respecto a la pregunta de si ven una oportunidad de mejora en su aplicación:

- Diez participantes dicen que no (4 expertos y 6 no expertos), pues utilizan los métodos que necesitan o no saben interpretar los resultados de la herramienta. Uno de ellos dice que tal vez podría cambiar la biblioteca, otro que tal vez achicar los argumentos de la

entrada, y otro que tendría que estudiar mejor el perfilador usado (Tracemalloc) para saber cómo interpretarlo y tomar decisiones.

- El resto responde cómo usarían la herramienta para ver si pueden mejorar su aplicación, como usar la información entregada o estudiar mejor las líneas que consumen más memoria para ver cómo optimizarlas y aprender a usar mejor la herramienta, sobre todo la parte de ver las diferencias de memoria en distintos momentos.

Comentarios: Diez de los participantes muestran interés por las características de la herramienta y realizan algunos comentarios:

- **Ventajas**: Cinco de los participantes comentan que la herramienta tiene algunas virtudes por sobre AlProfiler. Varios de ellos concuerdan en que la herramienta es rápida y no es tan compleja de usar, ofreciendo múltiples características que pueden ser útiles para analizar el consumo de memoria, además de venir incluido dentro de las bibliotecas de Python, en el caso de Tracemalloc. E4, que usa otro perfilador, destaca la rapidez de la herramienta y su utilidad para comparar algoritmos de ordenamiento.
- **Desventajas**: A pesar de las ventajas descritas anteriormente, todos los participantes concuerdan en que la herramienta es poco intuitiva y/o aceptable a la vista al momento de mostrar los resultados, además que la documentación no es muy amigable. Por ejemplo, uno de los participantes considera que no es consistente que se muestre en los resultados que una de las líneas de código que más consume memoria sea de la propia herramienta, otro comenta que es complicado relacionar elementos entre sí, y 3 opinan que es tedioso el tener que modificar tanto el código para poder analizar el consumo de memoria. E4, que usa otro perfilador, expresa que está dispuesto a cambiar el que está utilizando, y N7 dice: "*Supongo que primera y última vez que lo voy a usar*", incluso agrega: "*¿Me puedo quedar con la otra aplicación?*", refiriéndose a AlProfiler.
- **Más por ver**: Uno de los participantes comenta que le interesaría ver la memoria versus la cantidad de iteraciones en este perfilador (Tracemalloc), pero que tendría que estudiar más la documentación para ver si se puede obtener esa información.

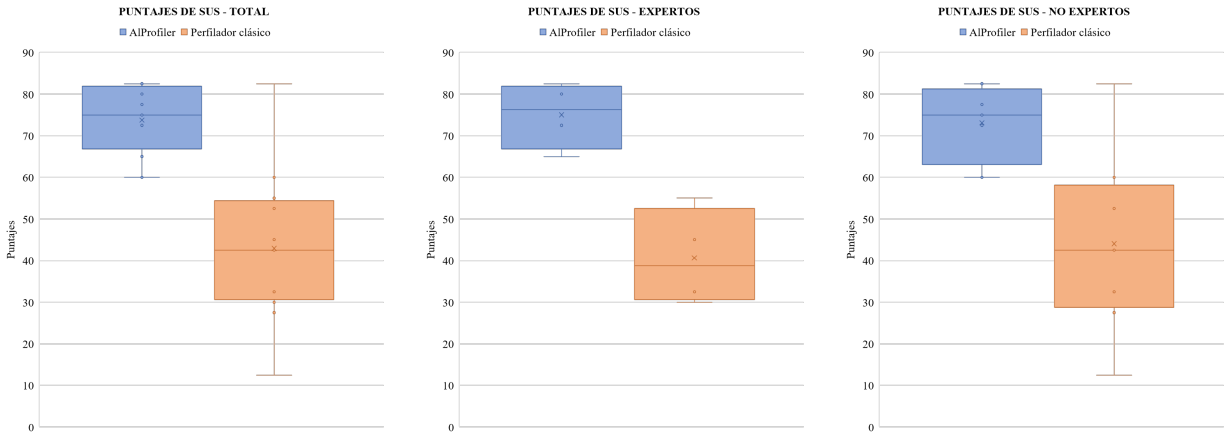
EXPERIENCIA DE LOS PARTICIPANTES

En esta sección se presentan los resultados de la escala de usabilidad del sistema (SUS) y los cuestionarios de NASA TLX.

La Figura 5.12 muestra los puntajes SUS para los perfiladores de memoria utilizados en una escala de 0 a 100. Del total, el puntaje SUS promedio para AlProfiler fue 73.75, con un máximo de 82.5 y un mínimo de 60, en tanto que el perfilador clásico presenta una media de 42.92, con un máximo de 82.5 y un mínimo de 12.5. En cuanto a los desarrolladores no expertos, la puntuación media de AlProfiler fue de 73.13 y de Tracemalloc (el perfilador con el que se hicieron las pruebas en todo este grupo) de 44.06. Por otro lado, con los desarrolladores expertos, la puntuación media de AlProfiler fue de 75 y del perfilador clásico de 40.63.

Por lo tanto, de acuerdo a lo propuesto como forma de interpretación[4], AlProfiler está por encima del promedio tanto para usuarios expertos como no expertos, lo cual es bueno, mientras que los perfiladores clásicos utilizados estuvieron bajo el promedio.

[4] `https://measuringu.com/sus/`



(a) Puntaje SUS del total de participantes.

(b) Puntaje SUS de los expertos.

(c) Puntaje SUS de los no expertos.

Figura 5.12: Puntaje SUS de utilizar ambos tipos de perfiladores (mayor puntaje es mejor).

Las medias de los puntajes SUS son similares entre expertos y no expertos, diferenciándose un poco más en la distribución de los puntajes. Si bien, no hay mayores diferencias entre estos grupos, se puede concluir la notable mayor usabilidad de AlProfiler por sobre las herramientas clásicas utilizadas por los participantes.

En la Figura 5.13 se muestra el puntaje total de NASA TLX por usar AlProfiler y un perfilador clásico, donde la escala va de 0 a 100. En las Figuras 5.14, 5.15, 5.16, 5.17, 5.18 y 5.19 se muestran los puntajes de NASA TLX separados por las subescalas que evalúa. El promedio por AlProfiler es 37.39, 39.33 y 36.42 para todos los participantes, expertos y no expertos, respectivamente, siendo el máximo 63.34 en expertos y 53.33 en no expertos, y el mínimo 17.33 en expertos y 13.34 en no expertos. Mientras tanto, con el perfilador clásico, el promedio es 47.56, 50.83 y 45.92 para todos los participantes, expertos y no expertos, respectivamente, siendo el máximo 79.34 en expertos y 84 en no expertos, y el mínimo 17.33 en expertos y 10.67 en no expertos.

En la Figura 5.13 se ve que no hay mayores diferencias en el conjunto de categorías que evalúa el NASA TLX entre expertos y no expertos. Sin embargo, se puede apreciar que, por lo general, AlProfiler implica menos carga para estos participantes que un perfilador clásico.

Respecto a las Figuras 5.14, 5.15, 5.16, 5.17, 5.18 y 5.19, se puede ver que, por lo general, la carga asociada a las subescalas que mide NASA TLX es menor con AlProfiler que con un perfilador clásico, a excepción de lo asociado a la demanda temporal, donde es menor con un perfilador estándar.

Con AlProfiler, frustración y demanda física tienen menos peso en la carga cognitiva en general, mientras que la demanda mental y temporal tienen un mayor peso. A su vez, con un perfilador clásico, la frustración y la demanda mental tienen mayor peso en la carga cognitiva, mientras que la demanda física y temporal son las que tienen menor peso.

Respecto a la diferencia entre usuarios expertos y no expertos, se pueden notar varias diferencias entre las categorías que mide esta encuesta, en particular:

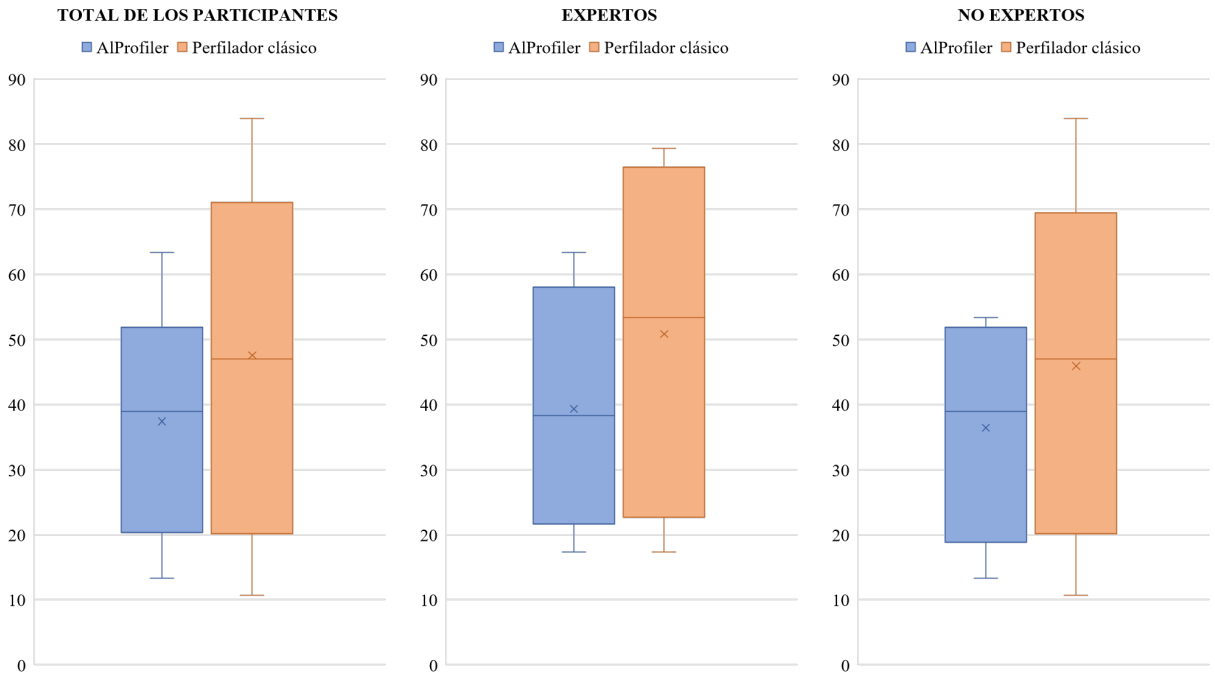
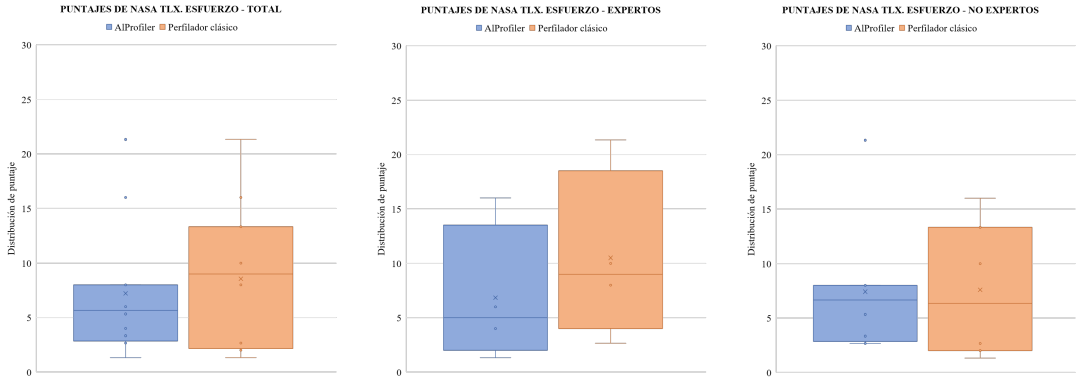


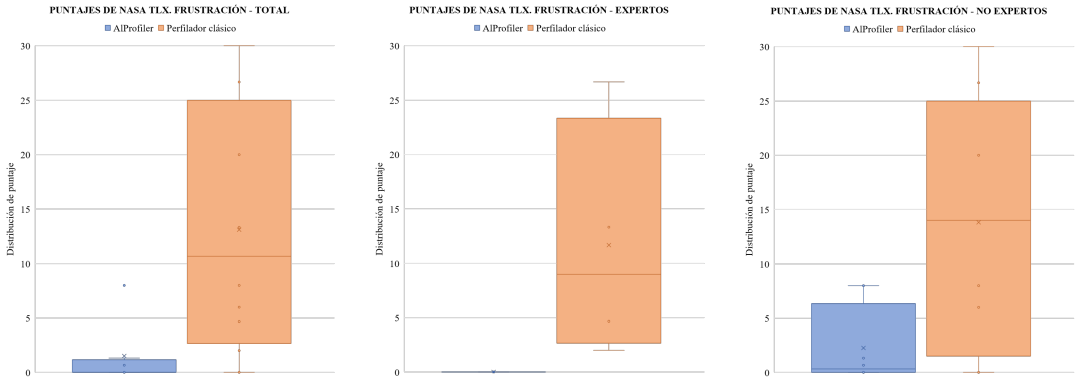
Figura 5.13: Puntaje NASA TLX de utilizar ambos tipos de perfiladores (menor puntaje es mejor).



(a) Total de participantes. (b) Expertos. (c) No expertos.

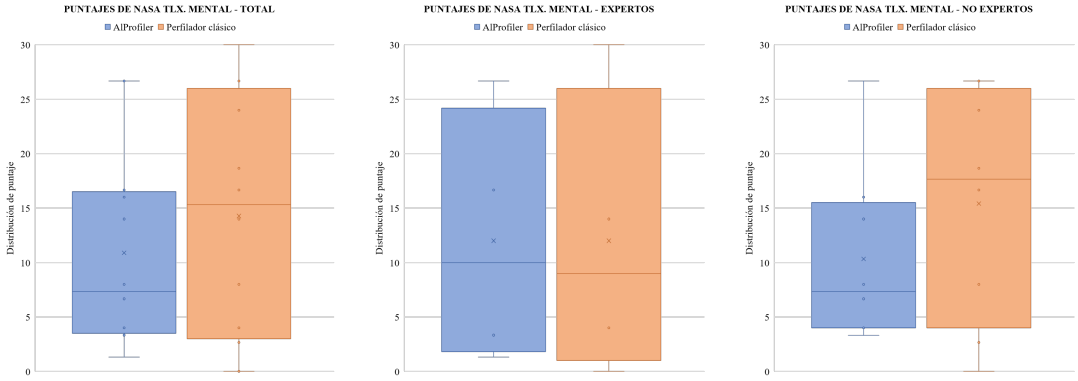
Figura 5.14: NASA TLX de utilizar ambos tipos de perfiladores por el esfuerzo (menor puntaje es mejor).

- Los participantes expertos consideran que la carga asociada a la frustración no tiene impacto en la carga que ellos sienten mientras usan AlProfiler, al contrario de los no expertos. Sin embargo, en ambos casos, esa carga es bastante menor en comparación a cómo la sienten con un perfilador clásico.
- Al revés, los participantes no expertos consideran que la carga asociada a la demanda física no pesa mucho en la carga general, con ninguno de los perfiladores (aunque en AlProfiler menos que en el perfilador estándar), mientras que los expertos consideran que tiene mayor peso.



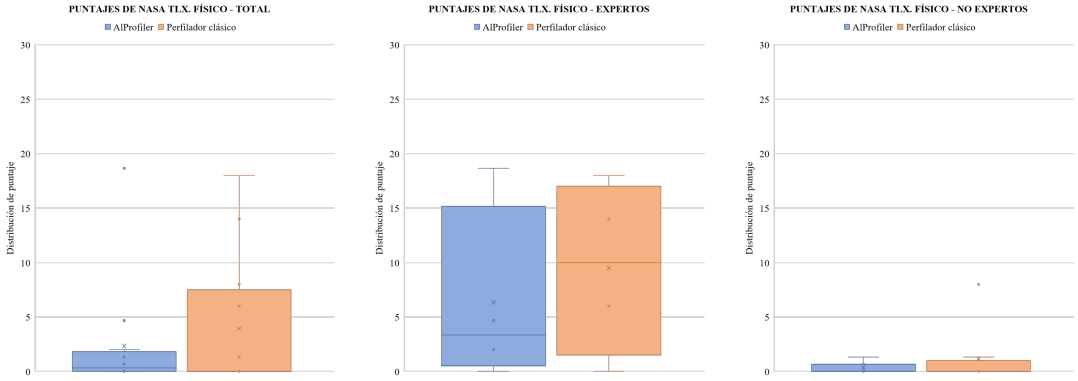
(a) Total de participantes. (b) Expertos. (c) No expertos.

Figura 5.15: NASA TLX de utilizar ambos tipos de perfiladores por frustración (menor puntaje es mejor).



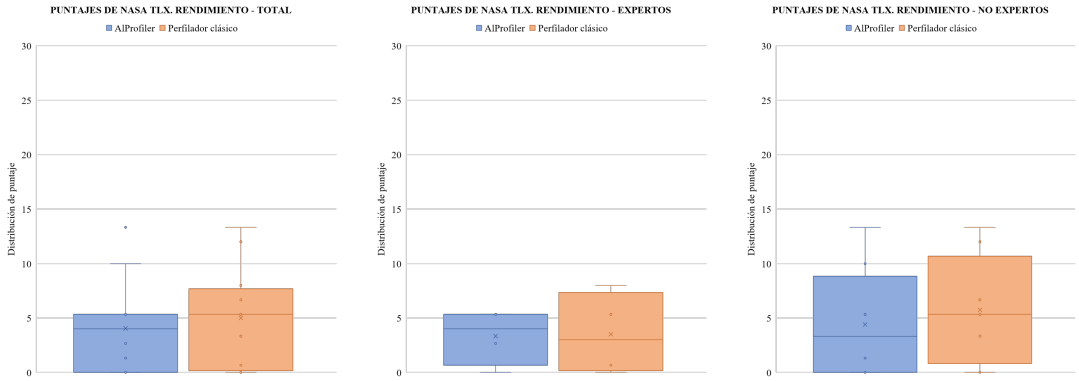
(a) Total de participantes. (b) Expertos. (c) No expertos.

Figura 5.16: NASA TLX de utilizar ambos tipos de perfiladores por demanda mental (menor puntaje es mejor).



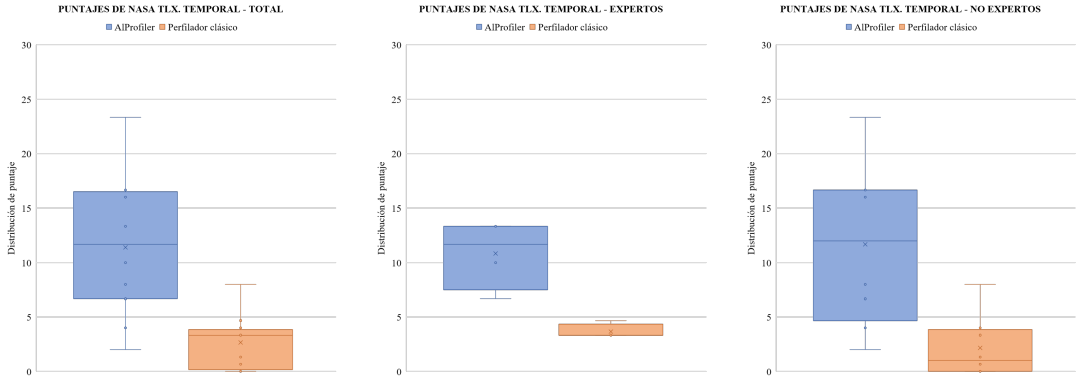
(a) Total de participantes. (b) Expertos. (c) No expertos.

Figura 5.17: NASA TLX de utilizar ambos tipos de perfiladores por demanda física (menor puntaje es mejor).



(a) Total de participantes. (b) Expertos. (c) No expertos.

Figura 5.18: NASA TLX de utilizar ambos tipos de perfiladores por rendimiento (menor puntaje es mejor).



(a) Total de participantes. (b) Expertos. (c) No expertos.

Figura 5.19: NASA TLX de utilizar ambos tipos de perfiladores por demanda temporal (menor puntaje es mejor).

- Hay mayores diferencias en la distribución respecto a la carga asociada al esfuerzo entre los expertos (AlProfiler un poco menor que al usar un perfilador clásico) que entre los no expertos (en general, AlProfiler un poco menor que con el clásico).
- También hay mayores diferencias en la distribución respecto a la carga asociada a la demanda mental entre los expertos (AlProfiler similar al usar un perfilador clásico) que entre los no expertos (en general, es menor con AlProfiler que con el clásico).
- Al revés, hay más diferencias en la distribución de los puntajes entre los no expertos con respecto al rendimiento. Sin embargo, en ambos grupos, los puntajes son similares entre ambas herramientas, siendo un poco menor con AlProfiler que con el clásico, en general.
- También hay mayores diferencias en la distribución de los puntajes entre los no expertos cuando la carga se asocia a la demanda temporal. Sin embargo, en ambos grupos, la carga es mayor con AlProfiler que con el perfilador clásico.

5.2.3. Discusión

Todos los participantes pueden caracterizar el consumo de memoria de sus aplicaciones con AlProfiler, utilizando los diferentes elementos de la visualización, interacciones e información mostrada. Principalmente, primero exploran la visualización para encontrar puntos de partida, luego expanden en esos puntos encontrados mientras buscan otros, y ven las relaciones que hay entre los métodos y el conjunto según estimen necesario. Por ejemplo, N8 busca cuáles son los primeras y últimas funciones en ejecutarse, y las compara con sus supuestos: "*Me gustaría conocer las primeras celdas o las primeras acciones que se ejecutan y, por último, revisar las últimas ejecuciones, que sé cuáles son, que son la conversión a archivo. Eso es lo que estoy buscando ahora*".

Respondiendo a la pregunta sobre qué aprendió de la aplicación y si hubo algo que le haya sorprendido, algo frecuente entre los desarrolladores no expertos fue la sorpresa por la gran cantidad de funciones desconocidas, pues no consideraban el impacto que tienen las dependencias externas sobre el consumo de memoria. Además:

- N1 dice que ahora puede determinar aproximadamente dónde se está consumiendo mayor memoria en su aplicación, siendo que antes era algo que no se lo había cuestionado.
- N2 obtuvo una visualización con 3 métodos, por lo que no le llamó la atención el impacto de bibliotecas, pero dejó claro que hay funciones que quedaron fuera del perfilamiento por cómo tenía escrito su código. De todas formas, le llama la atención una línea en particular que ocupa más memoria que otras, pero, comparando con información que poseía de su aplicación, hace sentido esta distribución de consumo de memoria.
- Llama la atención de N3 funciones que son llamadas por muchos métodos, se ejecutan varias veces o tienen la mayor cantidad de líneas de código, y no le son familiares. También las funciones que se ejecutan más veces de lo supuesto por las iteraciones del código pero, viendo su relación con otros métodos, termina por entenderse. Así también, le es interesante la cantidad de memoria consumida por el método principal.
- N4 detecta qué métodos son los que ocupan más memoria, lo que trata de explicar con la información que posee de su código. También le interesa los métodos que se ejecutan más veces, ya que menciona que le puede servir para ver que esos no se llamen tanto y disminuir el tiempo de ejecución de la aplicación, por ejemplo, que no tenga tantos *if*, *while* o *for*, solo cálculos. Además, busca un método que él crea para ser iterativo y llama su atención el cómo se ve en la estructura. También le interesa saber la memoria total consumida por su aplicación, para analizar la posibilidad de tenerla en dispositivos con memoria más limitada, como una Raspberry.
- N5 detecta la memoria consumida y el número de ejecuciones de métodos importantes para él, y el consumo total por parte de su aplicación, algo que antes no tenía tan claro por no usar perfiladores de memoria. Comprueba que funciona como espera, en general, al ver que sus funciones auxiliares no consumen tanta memoria.
- N6 se percata que debería mejorar los ciclos *for* porque produce muchas llamadas, provocando el consumo de bastante memoria. Podría cambiarlo por un *map* y ver si así mejora la visualización.
- N7 menciona que hay varias funciones que esperaba que llamara de las bibliotecas, pero otras muchas que no. También se da cuenta que consume menos memoria de lo esperado, pero el número de ejecuciones de varias de las funciones que le interesan son

las esperadas. En un método en particular, llama su atención que se ejecutara menos veces que otra, siendo que, teóricamente, deberían ser similares.

- N8 se da cuenta que la estructuración es vital, debiendo entonces agrupar de buena forma el código para facilitar el flujo de información. También, pensando que era más simple, le sorprende la cantidad de llamadas a métodos de bibliotecas.

Entre los desarrolladores expertos, tiende a cumplirse en cierta medida lo que suponían de sus aplicaciones. Aun así, hay nueva información para ellos. En particular:

- E1 dice que sabía que algunos métodos de archivos o dependencias externas podían consumir más memoria, pero como no sabe cómo funcionan estas bibliotecas, esta herramienta le da una información más sobre cómo funciona y cuánta memoria están usando esos métodos.
- E2 detecta las funciones que más consumen memoria, verificando lo que ella suponía.
- E3 se percata que hay algunos métodos que consumen menos memoria de la esperada.
- E4 indica que le interesa la cantidad de iteraciones de los métodos, ya que es información importante para él y que no proporciona tan claro otros perfiladores. También ahora puede saber la cantidad de memoria y ejecuciones de métodos importantes para él, lo que no se escapa tanto de lo que suponía.

Respecto a las vistas de AlProfiler, podría decirse que los *gráficos de subllamadas* son más complejos o más complicados de llegar, puesto que fueron más explotados por los desarrolladores expertos, mientras que el *gráfico de dispersión* fue más explotado por los desarrolladores no expertos, siendo esta una vista más simple para ellos. Incluso, N1 menciona "*acá me queda más claro lo que estaba haciendo*" cuando se encontraba en esa vista, y N4 dice "*y lo otro que me gustó, [...] lo del tema de este gráfico*".

Como estos participantes no usan, o usan muy poco, perfiladores de memoria, el tipo de preguntas más frecuentes entre ambos grupos son para detectar puntos de partida (FFP) y estudiar esos puntos (EFP).

Por eso también, los participantes hacen más preguntas relacionadas a lo que han leído o visto de la herramienta, pocas son las que no se pueden responder. Por ejemplo, ningún participante hace la pregunta 5, 7 o 27 con el perfilador clásico porque los ejemplos que usan de la documentación tienen otro objetivo y no pueden dar esas respuestas; o ninguno hace la 32 con AlProfiler porque eso no se puede responder con una sola ejecución del código con la herramienta, o la 29, porque la herramienta no propone esa característica de entregar valores de diferencia de consumo de memoria.

Sobre las preguntas hechas por los participantes:

- Por lo tanto, para conocer más su código y analizar el consumo de memoria con AlProfiler, tanto expertos como no expertos repiten preguntas sobre qué métodos tiene la aplicación, cuáles son los máximos de memoria y cantidad de ejecuciones, o características de métodos en particular (memoria consumida, cantidad de ejecuciones (aunque esto es más para expertos que no expertos), cómo se distribuye la memoria dentro de las funciones, a qué funciones llama o por cuáles es referenciado).

- Con los perfiladores clásicos, los desarrolladores expertos se enfocan más en encontrar los puntos que más consumen memoria, así como los desarrolladores no expertos. Sin embargo, como estos últimos no ven tan claros los resultados en un principio, o simplemente no los entienden, también se preguntan bastante a qué corresponde lo que están recibiendo como respuesta de la herramienta, resultado en las preguntas 6 y 15. Después ambos grupos se preguntan, aunque en menor cantidad, cuánta memoria consume alguna línea en particular o cuántas veces se ejecuta.
- El que las preguntas 6 y 15 sean más escasas entre expertos, puede indicar que los desarrolladores expertos conocen más su aplicación y tienden a entender mejor las herramientas, y se enfocan en otras características en vez de entender cómo está escrito un método de una biblioteca, por ejemplo, al contrario de un participante no experto.
- La pregunta 5 es porque les llamaba la atención los nodos de colores más oscuros, pero no parece ser tan importante esta información por lo escasa de la pregunta.
- Hay preguntas que tienen que ver con características propias o intereses particulares de algún participante, más entre no expertos, como la 3, 20, 21, 23, 27, 30 y 31.
- Respecto a las preguntas de entender relaciones (US y QGS), los desarrolladores no expertos aprovechan AlProfiler para hacer más preguntas al respecto, puesto que le es más útil, según ellos mismos expresan después; mientras que los desarrolladores expertos le dan un mayor porcentaje a esas preguntas en un perfilador clásico que en AlProfiler.

Sobre la usabilidad, se puede ver que es más usable AlProfiler que los perfiladores clásicos usados para los dos tipos de desarrolladores, puesto que ambos grupos evalúan positivamente el primero, quedando con un promedio por sobre el promedio de lo aceptable. En cambio, los perfiladores clásicos quedan con un promedio bajo lo aceptable. Eso también se ve reflejado en la retroalimentación y comentarios de los participantes.

En relación a la carga cognitiva:

- Se ve el impacto positivo que tiene AlProfiler por sobre el perfilador clásico para todos los participantes, pues ellos dan más puntaje (más carga) al perfilador estándar en todos los aspectos de NASA TLX, a excepción de la demanda temporal. Esto puede ser debido a la rapidez de la herramienta para dar resultados, y que ellos deciden no dedicar mucho tiempo a probar la aplicación puesto que no es muy amigable a la vista, sobre todo para el grupo de desarrolladores no expertos. Esto también se ve reflejado al ver cómo aumenta la carga asociada a la frustración en ambos grupos con el perfilador clásico.
- En muchos casos entre los desarrolladores no expertos se obtenían visualizaciones de gran tamaño con AlProfiler, provocando que se congelara la imagen en ciertos momentos, lo que puede influir en la carga cognitiva de los participantes, como se ve en frustración, que los expertos no le dan peso, en cambio los no expertos le dan un poco más.
- Respecto a los perfiladores clásicos, la carga asociada a frustración es alta y similar en ambos grupos, un poco más baja con expertos. Esto podría explicarse con los comentarios que hacen los participantes, entre no expertos que no se entiende mucho los resultados, y entre ambos grupos que dicen que no proporciona tanta información como AlProfiler.

- Como la exploración de las herramientas fue una tarea que requería más demanda mental, los participantes dieron más peso a ese aspecto en la carga cognitiva. Usualmente, los desarrolladores expertos se pueden manejar más con herramientas en terminales, como los perfiladores clásicos, y eso se puede ver también en que el promedio de la demanda mental con los perfiladores clásicos es menor en el grupo de expertos que en el de no expertos; a su vez, los desarrolladores no expertos se sienten más cómodos con herramientas visuales, viéndose un promedio más bajo en la demanda mental con AlProfiler que el grupo de expertos. Esto también se puede reflejar en los comentarios y en el tipo de preguntas que responden los participantes, por ejemplo, la pregunta 15 la hacen los no expertos cuando usan Tracemalloc porque no entendían a qué líneas correspondían los resultados entregados por la herramienta, preguntándose entonces qué son y cómo se ven, al contrario de los expertos.
- La carga cognitiva de rendimiento es menor en expertos que en no expertos, lo que también podría reflejar el mejor manejo o entendimiento que tienen los desarrolladores expertos, aunque es bajo en ambos grupos esta carga.
- Para los desarrolladores expertos, estas tareas tienen un componente físico considerable en comparación a los no expertos, esto puede ser que los no expertos le dan más importancia a otros aspectos al medir la carga cognitiva. También puede asociarse a cómo responden algunas preguntas, como la 6 y 7, que los no expertos a veces usaban la opción *FIND* de la herramienta AlProfiler, en cambio los expertos solo pasaban el cursor sobre los nodos del *gráfico de llamadas*, tomando más tiempo en buscar. También, algunos desarrolladores no expertos consultaban el tutorial para ver qué les faltaba por probar, los expertos no.

5.2.4. Validación

En esta sección se muestra la validez de este estudio.

Primeramente, en base a lo expuesto en el Capítulo 3 sobre la calidad de la metodología para evaluar el perfilador de memoria AlProfiler, el estudio realizado tiene en cuenta las necesidades y acciones de los desarrolladores durante el análisis del consumo de memoria. Se llama a desarrolladores expertos y no expertos para ver si existen diferencias en su comportamiento y cómo califican la herramienta usando un código bien conocido por ellos. También se les pide usar un perfilador clásico de memoria para notar diferencias con AlProfiler.

Se usan encuestas para medir la usabilidad de la herramienta y la carga cognitiva al momento de utilizar los perfiladores, encuestas cuya eficacia es bien conocida y sabiendo que estos participantes pueden representar usuarios finales.

Como la mayoría de los estudios empíricos, el presente trabajo también puede enfrentarse a amenazas a la validez. Para lidiar apropiadamente con ellos, se describe, usando la estructura de *conclusión, interna, de constructo* y *externa*, lo siguiente:

Validez de conclusión: Se refiere a los factores que pueden llevar el esfuerzo puesto en este trabajo a una conclusión incorrecta sobre una relación en las observaciones.

Las conclusiones mostradas se basan en un estudio de usuarios que involucra a un grupo de participantes. En este caso, esa cantidad de participantes es limitada. Se trata de mitigar esta amenaza eligiendo participantes con antecedentes diversos. Ellos son de distintas especialidades, con distintos grados, familiarizados con programación en Python, con distintos niveles de experiencia y años programando en este lenguaje. En total, son 12 personas: 8 que no son directamente de la especialidad de computación/informática, de los cuales son 6 hombres y 2 mujeres; y 4 que son de la especialidad de computación/informática, de los cuales 2 son hombres y 2 son mujeres.

Validez interna: Esta amenaza presenta influencias que podrían afectar a la variable independiente. La validez interna se refiere a qué tan bien se diseñó y realizó el experimento (por ejemplo, identificando posibles variables independientes).

Como se menciona con anterioridad, los participantes son de distintas especialidades y con distintos niveles de experiencia, pero todos están familiarizados con el lenguaje de programación Python. Las aplicaciones que utilizan los usuarios son códigos escritos por ellos mismos y que han usado en situaciones académicas, laborales, entre otros, experiencias reales en general. Por lo tanto, se evita tener variables independientes relacionadas con los códigos utilizados por los participantes.

Además, para mantener la validez del estudio, después de cada sesión se realiza la transcripción de esta utilizando distintas herramientas para permitir moverse entre informes, transcripciones y grabaciones, y así minimizar los sesgos de interpretación. En particular, se hace uso de las herramientas para generar archivos de subtítulos de las grabaciones, los cuales pasan posteriormente por un proceso de revisión y mejora de las partes que no hayan quedado bien escritas; también se usa la herramienta YouTube Marks, un proyecto creado por Martín Dias para facilitar el análisis de videos colocándoles tags y agregándoles comentarios a esos tags, para así dividir e identificar con mayor facilidad las partes de las sesiones; además de la creación de documentos que están disponibles para la creadora de la herramienta bajo estudio.

La identificación de preguntas basadas en el comportamiento de un participante puede ser inexacta, ya que ellos no siempre verbalizan sus pensamientos de manera explícita; la generalización de las preguntas también sufre de incertidumbre. Para minimizar la inexactitud, se contrasta diferentes escenarios y el uso de la herramienta respondiendo las mismas preguntas, además de discutir estos temas con la creadora de esta herramienta.

Validez de constructo: Se refiere a la generalización del resultado del experimento al concepto o teoría detrás del experimento.

Para propósitos de este estudio, se enfoca en el lenguaje de programación de Python, para analizar el consumo de memoria de aplicaciones escritas en este lenguaje. También se busca que los participantes conozcan el código que están eligiendo para los experimentos y el resultado de cada sesión se mide cuidadosamente mediante la observación y los métodos mencionados en la amenaza anterior.

Validez externa: Se refiere a la generalización del resultado del experimento a otros entornos distintos de aquel en el que se realiza el estudio.

Si bien se centra en aplicaciones hechas en Python, se trata de cubrir diversas categorías de aplicaciones, según las especialidades de cada participante.

La generalización de los resultados está restringida, debido a la dificultad de realizar tales estudios a gran escala, puesto que la transcripción y obtención de preguntas de una sesión requiere mucho tiempo. Aunque el número de sesiones es similar a algunos trabajos relacionados (por ejemplo, Kubelka [7], 17 sesiones; Fernandez [6], 8 sesiones), y se busca la variabilidad de los participantes para que representen a los usuarios finales de esta herramienta, no es muy seguro la generalización de resultados debido a cantidad de sesiones y participantes; equitatividad de los participantes (no expertos a expertos tienen una relación de 2:1); la falta de diversidad en tareas; y la duración de las sesiones.

También, los proyectos más grandes podrían causar mayor desorden visual o más problemas para visualizar la imagen final, siendo que ya con algunos participantes se tuvo que esperar más de una hora y media para poder apreciar los grafos, y con algunos se congelaba la imagen en ocasiones.

A pesar de que la validez externa es limitada, se cree que este estudio aporta evidencia relevante sobre la viabilidad de AlProfiler y la validez de esta metodología.

Capítulo 6

Conclusión

Este documento presentó el diseño y ejecución de una metodología para evaluar la herramienta AlProfiler, desarrollada por la alumna de doctorado Alison Fernández. Previamente, se realizaron pilotos con 8 programadores en Python, tanto del área de computación como de otras; se hicieron pruebas con perfiladores clásicos para conocer su funcionamiento y elegir uno para los posteriores experimentos, y se implementó un sistema de logs para facilitar la obtención de información de las sesiones de trabajo.

Finalmente, la metodología escogida consiste en un estudio de usuario que involucra a 12 practicantes de Python (4 de la especialidad de computación/informática (expertos), y 8 de otras áreas (no expertos)). En este estudio utilizan la herramienta y un perfilador clásico para identificar, comprender y mejorar el uso de memoria de sus aplicaciones hechas en Python.

En base a lo descrito anteriormente, se pudo cumplir el objetivo mencionado en la sección 1 de finalizar el prototipo inicial de una herramienta visual de perfilamiento de memoria y evaluarla empíricamente. Más específicamente, respecto a lo que se buscaba:

1. Se aportó para que la herramienta se pudiera usar lo más fácilmente posible por los probables usuarios al agregar un tutorial y un sistema para ejecutar la herramienta con una aplicación hecha en Python con una línea de comando.
2. Se pudo determinar, con los casos de estudios realizados, la efectividad de la herramienta por sobre un perfilador clásico, considerando la usabilidad determinada por los usuarios y la carga cognitiva al realizar la tarea.
3. Se pudieron ejecutar 8 estudios pilotos para mejorar la herramienta y definir la metodología a usar.
4. Se pudo definir un protocolo experimental para poder evaluar la herramienta, el que corresponde a la metodología ya mencionada.
5. Se pudo ejecutar esa evaluación con la herramienta con usuarios que son practicantes de Python.
6. Se analizaron los resultados, como se menciona en los siguientes párrafos.

Como resultados, se identifican 372 preguntas relacionadas con el análisis de consumo de memoria. También se destaca que los participantes pudieron identificar y caracterizar el consumo de memoria de sus aplicaciones, y se identifican patrones de interacción con los perfiladores para responder las preguntas encontradas.

En general, los participantes suelen primero conocer su código al ver qué funciones posee su aplicación y buscar funciones importantes, para luego estudiar algunos métodos relevantes. Entre las actividades más repetidas están el buscar cuáles son las funciones que más consumen memoria o se ejecutan más, y consultar sus características (memoria consumida, cantidad de ejecuciones, distribución de la memoria, llamadas con otros métodos).

También es importante para los desarrolladores entender en qué consisten las funciones que están tratando. Los participantes expertos pueden comprender más fácilmente a qué corresponden algunas funciones al conocer su nombre o ubicación dentro de alguna biblioteca, por ejemplo, pero no tanto los no expertos, aumentando entre ellos preguntas relacionadas a cómo es la definición sobre cierta entidad, sobre todo con un perfilador clásico.

Así mismo, se obtuvo que tanto expertos como no expertos evaluaron positivamente la herramienta y negativamente el perfilador estándar utilizado. Además, la carga cognitiva con el perfilador bajo estudio fue menor que con el perfilador clásico, más para el grupo de no expertos, viéndose que afecta positivamente en la experiencia de usuario de estos participantes. Para los participantes, al medir la carga cognitiva, la demanda mental tiene mayor preponderancia y la frustración es lo que tiene menor peso, junto con la demanda física (en el caso de los no expertos) y por rendimiento (en caso de los expertos). También, la demanda temporal era menor con el perfilador clásico que con AlProfiler.

Este trabajo es relevante porque analizar cómo las características de las técnicas visuales apoyan a los desarrolladores con sus necesidades es una tarea importante, en particular, para realizar perfiles de consumo de memoria, puesto que es una tarea compleja por implicar varias métricas y requerir una comprensión profunda del código analizado, sobre todo para desarrolladores que no son específicamente del área de la computación, y más aun considerando que se puede requerir cierta eficiencia en sus códigos. Sin embargo, no se han realizado estudios, de los que se tenga conocimiento, para proporcionar preguntas relacionadas con la tarea de analizar la memoria ni que las considere en métodos de investigación de visualizaciones, por lo que trabajos como los presentados en este documento cobran mayor relevancia.

Dentro de los aprendizajes obtenidos cuentan el aprender a diseñar y aplicar estos estudios de usuarios, dirigir las sesiones de trabajo, sobre todo considerando la complejidad de este tiempo de pandemia.

6.1. Trabajo futuro

Como AlProfiler es un prototipo actualmente, falta desarrollar más esta herramienta.

Por lo tanto, respecto al trabajo futuro, se pueden seguir haciendo más estudios de usuarios de tal manera de aumentar la cantidad de participantes y equiparar la cantidad de usuarios expertos con no expertos para tener un grupo de desarrolladores más representativos de los

usuarios finales, que es lo que limita la validez externa, y así notar diferencias de manera más fidedigna entre ambos grupos y poder generalizar los resultados, enfocándose en lo que los usuarios intentan responder con la herramienta y cómo la utilizan para responder esas preguntas, además de la experiencia de usuario.

También se pueden agregar más tareas, como solicitar a los usuarios que hagan cambios en su código si identifican que hay posibilidades de mejora para comprobar que se puede optimizar, siempre que se tenga en consideración el tiempo de ejecución de la herramienta y de exploración, además de que los proyectos grandes provocan mayor desorden visual o problemas para visualizar, por lo que es más recomendable elegir una metodología que se ajuste más a los tiempos y capacidades de los usuarios.

Para apoyar los casos de estudios, se puede usar otro sistema para medir la usabilidad de la herramienta, como el *Software Usability Measurement Inventory* (SUMI, cuestionario de 50 preguntas para averiguar la experiencia de usuario en 5 áreas: efectividad, eficiencia, ayuda, control, y aprendizaje) o el *Usefulness, Satisfaction, and Ease of use* (USE, cuestionario de 30 preguntas que evalúa la actitud del usuario hacia un producto en las categorías de utilidad, satisfacción, facilidad de uso y de aprendizaje), ya que se puede hacer un análisis de cada categoría de preguntas de los cuestionarios, permitiendo sacar más conclusiones.

Fuera de eso, también se puede utilizar la información obtenida en este estudio para mejorar la herramienta y enriquecerla con otras características que permitan medir el consumo de memoria, considerando así mismo los comentarios, ideas y retroalimentación que han dado los participantes al usar AlProfiler. Una vez obtenida una nueva versión, se puede hacer nuevos estudios de usuarios para verificar su efectividad.

Bibliografía

[1] Chis, A., Mitchell, N., Schonberg, E., Sevitsky, G., O'Sullivan, P., Parsons, T. y Murphy, J., Patterns of Memory Inefficiency. En: 25Th European Conference on Object-oriented Programming (ECOOP). Lancaster, UK, Association Internationale pour les Technologies Objets. pp.383-407., 2011. DOI: 10.1007/978-3-642-22655-7_18.

[2] Choudhury, A. y Rosen, P., Abstract visualization of runtime memory behavior. En: 6Th International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT). Williamsburg, VA, USA, IEEE. pp.1-8., 2011. DOI: 10.1109/Vissof.2011.6069452.

[3] Veroy, R., Ricci, N. y Guyer, S., Visualizing the Allocation and Death of Objects. En: 1st IEEE Working Conference on Software Visualization (VISSOFT). Eindhoven, Países Bajos, IEEE. pp.1-4., 2013. DOI: 10.1109/VISSOFT.2013.6650538.

[4] Bacher, I., Mac N., B. y Kelleher, J., On Using Tree Visualisation Techniques to Support Source Code Comprehension. En: IEEE Working Conference on Software Visualization (VISSOFT). Raleigh, NC, USA, IEEE. pp.91-95., 2016. DOI: 10.1109/VISSOFT.2016.8.

[5] Merino, L., Ghafari, M. y Nierstrasz, O., Towards Actionable Visualisation in Software Development. En: IEEE Working Conference on Software Visualization (VISSOFT). Raleigh, NC, USA, IEEE. pp.61-70., 2016.

[6] Fernandez, A., Sandoval A., J. y Bergel, A., Effective Visualization of Object Allocation Sites. En: IEEE Working Conference on Software Visualization (VISSOFT). Madrid, España, IEEE. pp.43-53., 2018. DOI: 10.1109/VISSOFT.2018.00013.

[7] Kubelka, J., Robbes, R. y Bergel, A., Live Programming and Software Evolution: Questions During a Programming Change Task. En: 27Th International Conference on Program Comprehension (ICPC). Montreal, QC, Canadá, IEEE/ACM. pp.30-41., 2019. DOI: 10.1109/ICPC.2019.00017.

[8] Sandoval A., J, Beck, F. y Bergel, A., Performance Evolution Matrix: Visualizing Performance Variations Along Software Versions. En: IEEE Working Conference on Software Visualization (VISSOFT). Cleveland, OH, USA, IEEE. pp.1-11., 2019. DOI: 10.1109/VISSOFT.2019.00009.

[9] Dias, M., Orellana, D., Vidal, S., Merino, L. y Bergel, A., Evaluating a Visual Approach for Understanding JavaScript Source Code. En: 28Th International Conference on Program Comprehension (ICPC). Seoul, República de Corea, ACM. pp.128-138., 2020. DOI: 10.1145/3387904.3389275.

[10] Aftandilian, E., Kelley, S., Gramazio, C., Ricci, N., Su, S. y Guyer, S. 2010, "Heapviz: Interactive Heap Visualization for Program Understanding and Debugging. Information Visualization," vol. 12(2), págs. 53-62, DOI: 10.1145/1879211.1879222.

[11] Byme, S. y Larus, J. 2018, "Detailed heap profiling. ACM Sigplan Notices," vol. 53(5), págs. 1-13, DOI: 10.1145/3299706.3210564.

[12] Easterbrook, S., Singer, J., Storey, M. A. y Damian, D. 2008, en, Selecting Empirical Methods for Software Engineering Research. Londres, Springer. pp.285-311, ISBN: 978-1-84800-043-8. DOI: 10.1007/978-1-84800-044-5_11.

[13] Marron, M., Sanchez, C., Su, Z. y Fähndrich, M. 2012, "Abstracting Runtime Heaps for Program Understanding. IEEE Transactions on Software Engineering," vol. 39(6), s.p. DOI: 10.1109/Tse.2012.69.

[14] Merino, L., Ghafari, M., Anslow, C. y Nierstrasz, O. 2018, "A Systematic Literature Review of Software Visualization Evaluation. Journal of Systems and Software," vol. 144, págs. 165-180, DOI: 10.1016/j.jss.2018.06.027.

[15] Millman, K. y Aivazis, M. 2011, "Python for Scientists and Engineers. Computing in Science & Engineering.," vol. 13(2), págs. 9-12, DOI: 10.1109/Mcse.2011.36.

[16] Sillito, J., Murphy, G. y Volder, K. 2008, "Asking and Answering Questions during a Programming Change Task. IEEE Transactions on Software Engineering," vol. 34(4), págs. 434-451, DOI: 10.1109/Tse.2008.26.

[17] Weninger, M., Grünbacher, P., Gander, E. y Schörgenhumer, A. 2020, "Evaluating an Interactive Memory Analysis Tool: Findings from a Cognitive Walkthrough and a User Study. Proceedings of the ACM on Human-Computer Interaction.," vol. 4(75), págs. 1-37, DOI: 10.1145/3394977.

Anexo A

Código de ejemplo.

Listing A.1: Código de ejemplo.

```
# Constantes
ITERATIONS = 1000

# Funciones para las condiciones de borde

# temperaturaMar: num -> num
# Entrega la temperatura del mar dada una cierta hora
def temperaturaMar(hora):

    if hora <= 8:
        return 4
    elif hora <= 16:
        return 2 * hora - 12
    else:
        return -2 * hora + 52

# calorChimenea: num -> num
# Entrega la temperatura emitido por la chimenea dada una cierta
# hora
def calorChimenea(hora):
    t = 500 * (m.cos(m.pi * hora / 12) + 2)
    return t

# temperaturaBordes: num num -> num
# Entrega la temperatura de los bordes del perfil, que decae 6
# grados cada 1000 m de forma lineal segun la temperatura del mar
# en esa hora
def temperaturaBordes(altura, tempini):
    t = (-6 * altura / 1000) + tempini
    return t
```

```
class Mountain(object):
    """
    Clase Mountain.
    """

    def __init__(self, ancho, alto, h, alpha):
        """
        Constructor
        :param ancho: ancho del perfil
        :param alto: alto del perfil
        :param h: espaciado del mallado
        :param alpha: constante que gobierna la generacion de
        montanas
        """

        # Almacenamos valores
        self.ancho = ancho
        self.alto = alto
        self.dh = h
        self.a = alpha

        # Almacenamos cantidad de celdas de mi matriz
        self.w = int(ancho / h)
        self.h = int(alto / h)

        # Creamos la matriz (mallado)
        self.matrix = np.zeros((self.h, self.w))

        # Elegimos, al azar, cuanto mar sera, de un 30%-40%.
        # Obtenemos el valor entero, ya que la matriz tiene
        # una cantidad entera de puntos.
        self.ptoPartida = 1+int(4000*r.uniform(3,4)/10)/self.dh
        # Desde aqui es continente

        # Inicializamos la montana
        for j in range(int(self.ptoPartida), self.w):
            azar = (self.h-1)-r.randint(0,self.h-1)*self.a
            # Seleccionamos, al azar, la altura de la montana en
            # cada columna y se multiplica por el alpha.
            for i in range(azar, self.h):
                self.matrix[i,j] = np.nan
                # Temperatura de la montana

    def plot(self):
        """
        Plotea la matriz
        :return:
```

```
        """
        fig = plt.figure()
        ax = fig.add_subplot(111)
        extnt = [0, self.ancho, 0, self.alto];

        # Se agrega grafico al plot
        cax = ax.imshow(self.matrix, extent=extnt)
        plt.xlabel("x[m]")
        plt.ylabel("y[m]")
        plt.title("Cordilleras aleatorias")
        plt.show()

class Perfil(object):
    """
    Clase Perfil, itera.
    """

    def __init__(self, matrix, hora):
        """
        Constructor
        :param matrix: matriz inicial del perfil
        :param hora: hora en el que se analiza el perfil
        """

        # Almacenamos valores
        self.ancho = matrix.ancho
        self.alto = matrix.alto
        self.dh = matrix.dh
        self.hr = hora

        # Almacenamos cantidad de celdas de mi matriz
        self.w = matrix.w
        self.h = matrix.h

        # Almacenamos otros valores
        self.ptoContinente = int(matrix.ptoPartida)
        # Punto en el que parte el continente
        self.chimenea = int(100 / self.dh) + 1
        # Cantidad de puntos que corresponde a la chimenea
        tempMar = temperaturaMar(self.hr)
        # Temperatura del mar en un cierto horario
        self.matrix = matrix.matrix
        # Matriz del perfil

        # Condiciones iniciales (sin chimenea)

        for i in range(0, self.h):
```

```
        for j in range(0, self.w):
            if  np.isnan(self.matrix[i,j]):
            # Ver que no caiga en montana
                continue
            else:
                self.matrix[i,j]=temperaturaBordes(self.alto
                                    -(i*self.dh),tempMar)

    for j in range(self.ptoContinente-self.chimenea,
                                    self.ptoContinente):
    # Agregar chimenea
        self.matrix[self.h-1,j]=calorChimenea(self.hr)

def __str__(self):
    """
    Imprime la matriz
    :return:
    """
    print(self.matrix)
    return ''

def iterate(self):
    """
    Itera la matriz
    :return:
    """
    contador = 0 # Contar la cantidad de iteraciones
    while (contador < ITERATIONS):
        # Los bordes del perfil y la montana se mantienen
        # constantes.
        # Se itera sobre los puntos interiores, no se
        # consideran los anteriormente mencionados.
        for i in range(1, self.h - 1):
            for j in range(1, self.w - 1):
                # El punto es montana
                if np.isnan(self.matrix[i,j]):
                    continue
                # El punto tiene montana a ambos lados y abajo
                elif np.isnan(self.matrix[i,j-1]) and
                np.isnan(self.matrix[i,j+1]) and
                np.isnan(self.matrix[i+1,j]):
                    self.matrix[i,j]=0.25*(self.matrix[i-1,j]
                                                +45)
                # El punto tiene montana a la derecha y abajo
                elif np.isnan(self.matrix[i,j+1]) and
                np.isnan(self.matrix[i+1,j]):
                    self.matrix[i,j]=0.25*(self.matrix[i-1,j]
```

```
                                    +30+self.matrix[i,j-1])
                    # Tiene montana a la izquierda y abajo
                    elif np.isnan(self.matrix[i,j-1])
                    and np.isnan(self.matrix[i+1,j]):
                        self.matrix[i,j]=0.25*(self.matrix[i-1,j]
                                    +30+self.matrix[i,j+1])
                    # El punto tiene montana abajo
                    elif np.isnan(self.matrix[i+1,j]):
                        self.matrix[i,j]=0.25*(self.matrix[i-1,j]
                                    +15+self.matrix[i,j+1]
                                    +self.matrix[i,j-1])
                    # El punto tiene montana a ambos lados
                    elif np.isnan(self.matrix[i,j-1]) and
                    np.isnan(self.matrix[i,j+1]):
                        self.matrix[i,j]=0.25*(self.matrix[i+1,j]+
                                    self.matrix[i-1,j]+30)
                    # El punto tiene montana a la derecha
                    elif np.isnan(self.matrix[i,j+1]):
                        self.matrix[i,j]=0.25*(self.matrix[i+1,j]+
                                    self.matrix[i-1,j]+15+
                                    self.matrix[i,j-1])
                    # El punto tiene montana a la izquierda
                    elif np.isnan(self.matrix[i,j-1]):
                        self.matrix[i,j]=0.25*(self.matrix[i+1,j]+
                                    self.matrix[i-1,j]+15+
                                    self.matrix[i,j+1])
                    # El punto no tiene montana a su alrededor
                    else:
                        self.matrix[i,j]=0.25*(self.matrix[i+1,j]+
                                    self.matrix[i-1,j]+
                                    self.matrix[i,j+1]+
                                    self.matrix[i,j-1])
            contador += 1

    def iterateacotado(self):
        """
        Para graficar de forma distinta y obtener la variacion de
        distancias alejadas de la chimenea
        :return:
        """
        self.iterate()
        maximo = 40
        for i in range(1, self.h):
            for j in range(1, self.w - 1):
                if self.matrix[i,j] > maximo:
                    self.matrix[i,j]=maximo
```

```
    def plot(self):
        """
        Plotea la solucion
        :return:
        """
        fig = plt.figure()
        ax = fig.add_subplot(111)
        extnt = [0, self.ancho, 0, self.alto];

        # Se agrega grafico al plot
        cax = ax.imshow(self.matrix, extent=extnt)
        fig.colorbar(cax)
        plt.xlabel("x[m]")
        plt.ylabel("y[m]")
        if self.hr==0:
            plt.title("Modelacion a la hora 0")
        elif self.hr==8:
            plt.title("Modelacion a la hora 8")
        elif self.hr==12:
            plt.title("Modelacion a la hora 12")
        elif self.hr==16:
            plt.title("Modelacion a la hora 16")
        elif self.hr==20:
            plt.title("Modelacion a la hora 20")
        else:
            plt.title("Modelacion a una cierta hora")
        plt.show()

def main():
    d = 100  # Espaciado, en metros, del mallado
    a = 1    # Constante alpha
    mtn = Mountain(4000, 2000, d, a)  # Inicializamos la montana
    #mtn.plot() # Para ver la montana generada

    # Perfila a la hora 0
    perfil1 = Perfil(mtn, 0)
    perfil1.plot() # Perfil sin iteraciones
    perfil1.iterate()
    print(perfil1)

    # Perfil a la hora 8
    perfil2 = Perfil(mtn, 8)
    perfil2.iterateacotado()  # Graficar las variaciones de
                              # temperatura no tan altas
    print(perfil2)
    perfil2.plot()
```