Universidad Nacional de Córdoba

Trabajo especial de la Licenciatura en Ciencias de la Computación

Optimización del seguimiento de C. elegans en videos de microscopía

Presentado por

Sebastián Marín

Director: Jorge Sánchez
Profesor representante: Nicolás Wolovick
Colaboración con: Sergio Simonetta (Phylumtech)

Agosto 2022

Resumen

En este trabajo se optimiza un algoritmo que detecta gusanos en imágenes de microscopía. Se comienza con un análisis del algoritmo existente, tanto de su comportamiento como la distribución de los tiempos de ejecución. Luego, se prueban varias optimizaciones, comenzando por los cuello de botella más notables y luego con optimizaciones menores. Finalmente, se analiza el desempeño de estas optimizaciones en una Raspberry Pi, siendo esta la plataforma sobre la que se planifica utilizar el programa optimizado.

Palabras clave: C. elegans, detección, optimización, C++

Abstract

In this work we present an optimization for a program that detects worms in microscopic images. It starts with an analysis of the existing algorithm, regarding both its behaviour and the distribution of its execution time. Then, several optimizations are tested, starting with the heaviest bottle necks, and following with minor optimizations. Finally, the results are analyzed on a Raspberry Pi, which is the platform where the optimized program will be used on.

Keywords: C. elegans, detection, optimization, C++

Índice

1. Introducción

1.1. El gusano C. elegans

El análisis automático de imágenes es un área de sumo interés, con aplicaciones prácticas en una gran cantidad de áreas. En particular, en este trabajo nos centraremos en el uso de estas tecnologías sobre el gusano *Caenorhabditis elegans* (*C. elegans*). Este gusano es un organismo muy pequeño y sencillo que se usa popularmente como modelo animal para el estudio de varios aspectos de la ciencia, incluyendo el estudio de sus redes neuronales[1], genética del comportamiento[2], y reacción ante ciertos químicos[3].

Figura 1: Imagen de gusano C-elegans.

En la actualidad, se postula como un modelo útil para realizar las primeras pruebas de factibilidad de medicamentos para humanos, en las fases de descubrimiento de fármacos y ensayos preclínicos tempranos.

Una de las formas de medir el efecto de un medicamento, ya sea su toxicidad o eficacia, está basada en el análisis del comportamiento de los animales.

Para el caso del *C.elegans*, el análisis más simple consiste en medir su movimiento antes y después de estar expuesto a una droga, para ver si mejora, empeora, o se detiene totalmente (es decir, mueren). En general para el estudio del gusano ante diferentes contextos y situaciones resulta útil poder medir su movimiento, que es su principal reacción al entorno.

Una forma de medir el movimiento es tener un usuario mirando muestras bajo un microscopio y sacando conclusiones. Para facilitar esta tarea, lo normal es guardar grabaciones (es decir una secuencia de imágenes) de los gusanos, y luego hacer los estudios, probando o siguiendo algunas métricas bien definidas mediante la ayuda de programas no automatizados tal como el ImageJ[1].

Un problema que tiene este enfoque es que el usuario es lento al analizar las imágenes, y puede tener que realizar mucho trabajo tedioso, como por ejemplo medir la distancia recorrida en pixeles por los organismos. Específicamente para este tipo de tareas existe una rama de la computación conocida como visión por computadoras, donde podemos escribir programas para que realicen estas tareas en poco tiempo. Notar que con la tecnología actual, a veces intercambian este aumento de velocidad por la precisión de los resultados, por lo que sigue requiriendo un usuario que ajuste los programas e interprete correctamente los resultados.

[1] https://imagej.nih.gov/ij/

1.2. Phylumtech

Este trabajo nace del requerimiento de una empresa Start up Argentina llamada Phylumtech, con el objetivo de poder eficientizar un programa de reconocimiento y trackeo de nematodos. Phylumtech SA es una empresa originada en el CONICET y Fundación Instituto Leloir, fundada en el Año 2015 con inversión del CITES (Sunchales), que trabaja en el desarrollo de soluciones innovadoras para el descubrimiento de nuevos fármacos.

Figura 2: Logo de Phylumtech

Entre sus productos ha desarrollado un sistema de adquisición de imágenes y procesamiento automático del movimiento de gusanos, que pretende solucionar uno de los cuellos de botella tecnológicos existente: obtener procesamiento de las imágenes en tiempo real. Por ejemplo, uno de los equipos existentes en el mercado llamado Lifespan Machine [4], realiza un análisis profundo de los gusanos y demora del orden de horas a días en obtener los resultados.

Muchas veces, los resultados se hacen evidentes sin la necesidad de hacer un análisis profundo de las mismas (Por ejemplo: hubo un error al sacar las imágenes, los gusanos están muertos, están demasiado agrupados, etc). Por eso, el desarrollo de un programa más sencillo, que pierde un poco en la profundidad y precisión del análisis pero nos permita ver los resultados en real-time, es útil para obtener algunas conclusiones preliminares y decidir si es conveniente continuar con un experimento o no.

El software de detección de movimiento desarrollado por Phylumtech posee varios bloques que basan su funcionamiento en: obtener un conjunto de imágenes, identificar los gusanos que se encuentran en cada una, y luego realizar un seguimiento del movimiento mediante la detección de los gusanos coincidentes entre imágenes (por ejemplo, en la figura 3 hay dos imágenes, donde podemos ver que el gusano 19 de la foto de la izquierda es el mismo que el gusano 18 de la foto de la derecha.

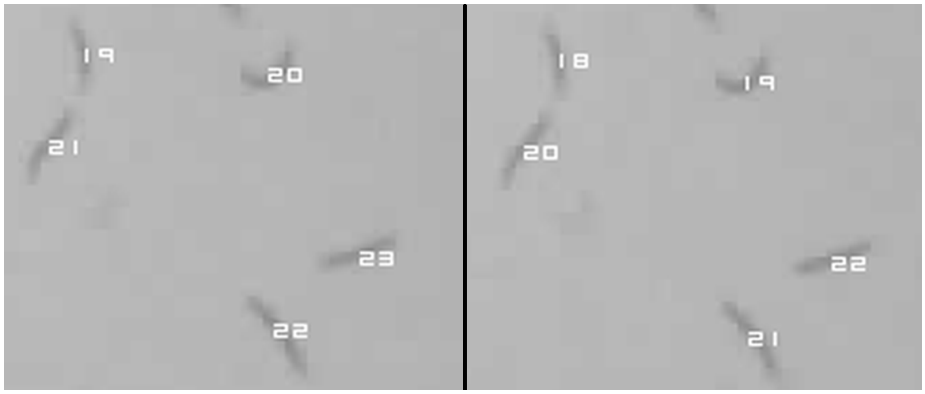


Figura 3: Enumeración de gusanos

.

Con esta información, luego se pueden calcular parámetros asociados al movimiento de cada gusano individual tal como la velocidad y distancia recorrida, y sacar conclusiones del progreso del experimento determinado.

1.3. Objetivo

En este trabajo de tesis nos concentramos en la porción funcional del programa que identifica los gusanos. Buscamos proponer una alternativa que funcione cuantitativamente más rápido que el anterior, manteniendo o mejorando su precisión.

1.4. Organización del trabajo

Debido a la naturaleza del problema, no hay una separación clara entre el análisis teórico y el experimental, sino que se complementan a lo largo del trabajo. Dividimos el proyecto en las siguientes etapas:

1. Contexto del programa: Se explica una visión general de qué es el programa que buscamos optimizar, y sobre qué tipos de experimentos se aplica.
2. Detector de gusanos: Se desarrolla en más profundidad el funcionamiento del programa, junto con un análisis preliminar de las variables que más impactan en su tiempo de ejecución.
3. Optimizaciones: Se presentan las optimizaciones que se probaron, incluyendo aquellas que obtuvieron resultados significativos y aquellas que no.
4. Análisis: Se analiza el programa obtenido como resultado de este trabajo
5. Conclusiones y posibles líneas de trabajo futuro

2. Contexto

2.1. El programa

Phylumtech tiene un dispositivo para analizar el movimiento de gusanos. Esto abarca todos los procesos desde la toma de imágenes hasta la presentación visual de los resultados. Una parte de este proceso se encarga de detectar los gusanos que tiene una imagen (Ejemplos de imágenes se pueden ver en la figura 6).

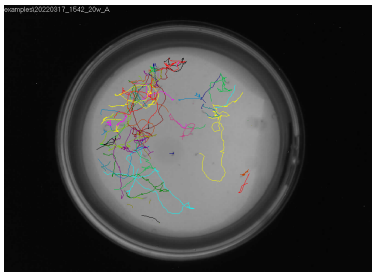


Figura 4: Trazas de movimiento de gusanos

El programa que detecta los gusanos recibe como entrada una imagen, en formato PGM, en escala de grises. Tiene 1600 x 1200

pixeles, cada uno tomando un valor entre 0 y 255, donde el 0 es negro y el 255 es blanco. Estas imágenes son tomadas por un sensor *OV2640* (Omnivision technologies) que captura imágenes de una placa de petri de 35mm, la cual esta iluminada por luz infrarroja.

Además se proveen como entrada algunos valores útiles o hiperparámetros del modelo (Por ejemplo, la posición de la placa de petri dentro de la imagen, y la longitud admitida para un gusano)

Como salida, este programa escribe en un archivo una cadena de bytes que representan la información correspondiente a los gusanos encontrados (por ejemplo, en qué posición se encuentre dentro de la imagen y qué tamaño tiene).

El dispositivo actual corre sobre una PC de manera secuencial, pero se desea el escalar el sistema, permitiendo utilizar varias cámaras en paralelo, e independizarse del uso de una PC asociada. El nuevo dispositivo debe ser autónomo y de bajo costo, lo cual presenta algunas restricciones con respecto al hardware utilizado. Esto tiene un impacto en el tipo y complejidad del procesamiento que se puede hacer.

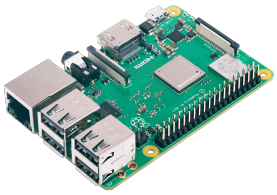


Figura 5: Raspberry Pi 3

Se eligió como dispositivo la Raspberry Pi modelo 3B+. Tiene instalado el sistema operativo *Raspbian Lite Debian Jessie*. Este programa está escrito en C++, con versión de compilador "C++ (Raspbian 10.2.1−6+rpi1) 10.2.1 20210110". Este compilador por defecto utiliza la versión *C++ 14*.

En este trabajo, lo que se busca es obtener una versión nueva del programa, que funcione cuantitativamente más rápido que el anterior, manteniendo una precisión igual o superior. Por precisión entendemos que detecta la misma cantidad o más de los gusanos que están en la imagen, y obteniendo la misma cantidad o menos de falsas detecciones, es decir interpretar como gusano algo que no lo es.

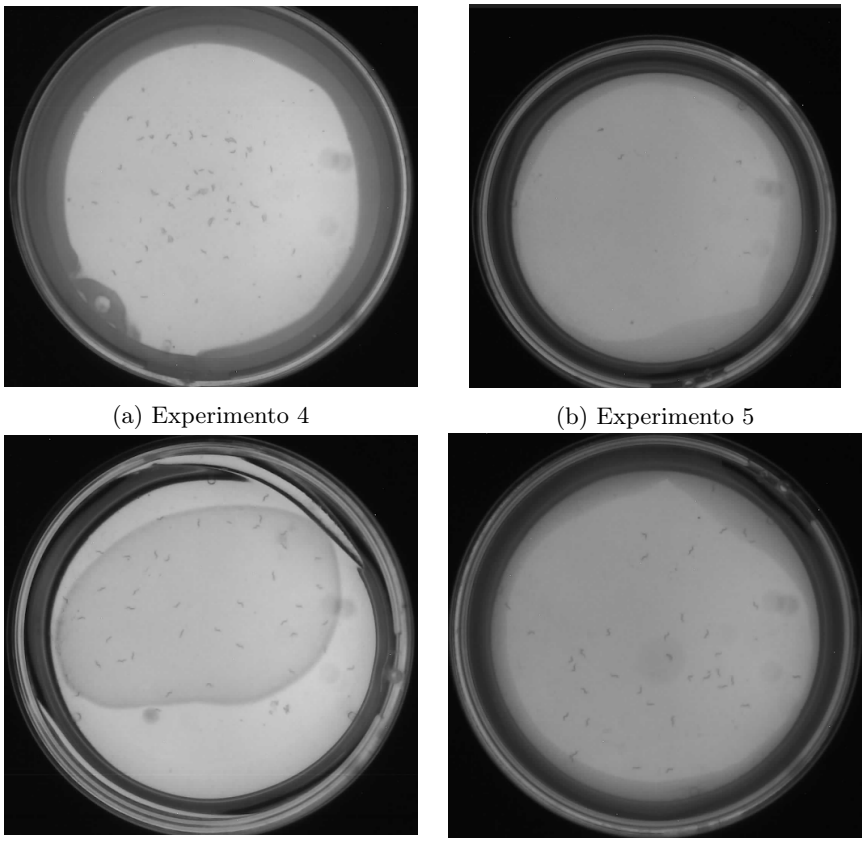
2.2. El conjunto de datos

Para medir estas mejoras, en cuanto a tiempo de ejecución o precisión de los resultados se eligieron 15 filmaciones de gusanos tomados de experimentos reales con los que trabajó Phylumtech.

Estas filmaciones vienen en la forma de al rededor de 300 imágenes cada una, y fueron elegidas estratégicamente para tener variedad en distintos factores, como la cantidad de gusanos o la cantidad de *manchas*, es decir figuras oscuras que no son gusanos, entre otros. También nos permiten evaluar distintos contextos donde nos interesa ver la precisión de los resultados, como por ejemplo qué pasa si los gusanos están mas cercas, si se mueven mucho o poco, etc.

Veamos algunos ejemplos. Observando la figura 6:

- El experimento 4 corresponde a un experimento con una alta cantidad de gusanos, de longitud relativamente pequeña.



(a) Experimento 4

(b) Experimento 5

(c) Experimento 14

(d) Experimento 11

Figura 6: Ejemplos de experimentos

- El experimento 5 tiene pocos gusanos y pocas secciones con pixeles oscuros.
- En el experimento 14 los gusanos están expuestos ante una bacteria, generando una diferencia en parte de la imagen entre el fondo uniforme claro, y fondo menos claro donde se encuentran los gusanos.
- El experimento 11 es el que tiene mayor cantidad de regiones donde hay un contraste alto entre los niveles de intensidad de las imágenes.

Estos datos (a veces referidos como *dataset*) los usaremos para medir los distintos valores que se observan a lo largo del trabajo.

3. Detector de gusanos

3.1. Comprensión y refactorización

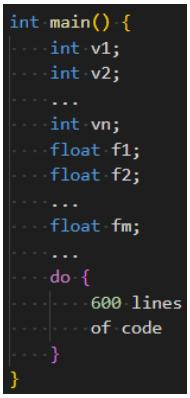


Figura 7: representación visual del código

Antes de comenzar a optimizar el algoritmo existente, hay una etapa de análisis para entender el algoritmo actual y su implementación.

Se comienza con un código escrito en C++ donde toda la lógica (más de mil líneas de código) está escrita en dos grandes funciones. Todas las variables utilizadas se declaran al principio de cada función (a excepción de algunas pocas variables globales) lo cual resulta poco amigable a la lectura (en la Figura 7 vemos una representación visual de esto).

Entonces, lo primero que se hizo fue identificar los bloques funcionales del código, y escribir funciones separadas que engloben su funcionamiento. Esta refactorización fue considerada valiosa por varios motivos, entre los cuales:

- Ayuda a la legibilidad del código y por lo tanto el entendimiento del algoritmo.
- Es más fácil ver qué variables o qué lógica no se usan (o no se necesitan) más. Es común dejar algunas líneas de código que son obsoletas luego de nuevos cambios.
- Al actualizar/modificar algunas partes, el hecho de tenerlo separado del resto del código hace más fácil ver su impacto en las variables locales y/o globales y reduce la probabilidad de introducir un comportamiento inesperado en la interacción con el resto del código.
- El tiempo invertido en reacomodar el código también es tiempo dedicado a revisar algunas de las decisiones de diseño más sutiles, y analizar potenciales bugs.

De hecho, durante esta refactorización del código se pudieron identificar algunos *off-by-one*. El off-by-one es un bug que ocurre cuando se accede en un arreglo en la primera posición que no le pertenece (En particular en C++ se indexa desde 0, entonces un arreglo de 4 valores usa los índices del 0 al 3 inclusive. Por lo tanto, el índice -1, o el índice correspondiente al tamaño alocado no le corresponden al arreglo)[2].

[2] En general se considera off-by-one cuando se itera una vez de más o de menos, no necesariamente ligado a memoria inválida. https://en.wikipedia.org/wiki/Off-by-one_error

Esto genera un comportamiento indefinido que puede o no causar un error dependiendo del contexto de la ejecución. Si se intenta acceder a una posición no alocada por el programa, lo más probable es que resulte en un *runtime error* (También conocido como *error de tiempo de ejecución*). También puede suceder que acceda a memoria alocada pero que pertenezca a una variable distinta al arreglo y modifique su valor. Dependiendo del uso de esta variable modificada puede o no verse un cambio relevante reflejado en otras partes del programa.

En particular uno de estos errores saltó con un runtime error de C++ llamado *stack smashing detected*[3]. Este error aparece cuando al finalizar la ejecución de una función se hace una verificación que ayuda a detectar si se accedió a memoria inválida. Al tener el código incorrecto encapsulado en una función pequeña, la ejecución del programa pudo detectar que estaba modificando memoria inesperada, mientras que al estar en una función mucho mas grande no se detectaba (probablemente porque la memoria accedida estaba alocada dentro de la función grande, perteneciendo a alguna variable diferente del arreglo).

3.2. Descripción del algoritmo actual

Luego de hacer este análisis (y refactorización), estos fueron los bloques principales identificados del código:

1. **Argumentos**: Se interpretan los argumentos de compilación. Esto incluye algunos valores como el nombre del archivo que tiene la imagen a analizar, la posición del círculo que contiene los gusanos dentro de la imagen, y algunos hiper-parámetros que usamos para determinar qué es un gusano y qué no.

2. **Lectura**: Se lee la imagen.

3. **Ver subcuadrados**: Se aprovecha el dato de que los gusanos se encuentran dentro de un círculo (la placa de petri) para iterar sólo sobre los pixeles dentro del mismo. En lugar de iterar pixel por pixel, se itera sobre subcuadrados (típicamente de tamaño entre 30x30 y 40x40 pixeles, es un valor ajustable del modelo). La figura 8 muestra una representación visual de esto.

4. **Prueba de interés**: Dado un subcuadrado, se realiza una primera prueba de si *es interesante*, es decir, si potencialmente puede tener un gusano adentro.

   En lugar de iterar pixel por pixel, se itera sobre 4 filas distribuidas uniformemente (Es decir, si el cuadrado es 40x40, las filas iteradas son la 0, 10, 20 y 30), y dentro de las filas elegidas se recorren todos sus pixeles. Para cada fila seleccionada, se calcula su intensidad promedio para encontrar *la fila más clara*. Análogamente, se seleccionan 4 columnas y se identifica la más clara.

[3] https://stackoverflow.com/questions/40416516/what-is-stack-smashing-c

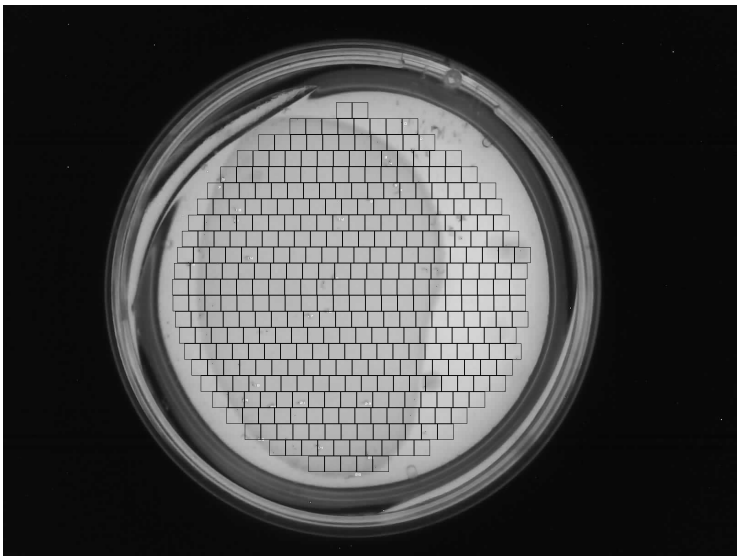


Figura 8: cuadrados iterados

Además, se apunta la intensidad del pixel más oscuro que se encontró al iterar sobre las 4 filas y columnas.

Si la diferencia entre la fila más clara y el pixel más oscuro es lo suficientemente grande, se considera que el cuadrado "es interesante", y por lo tanto puede contener un gusano.

El objetivo de este paso es ver que el pixel más oscuro sea notablemente diferente del entorno. En caso contrario, probablemente sea una mancha o espacio blanco del fondo. No iteramos sobre el cuadrado entero porque consideramos que los gusanos tienen una longitud suficiente para que podamos detectar su presencia viendo los intervalos elegidos, ahorrando tiempo de cómputo.

En la figura 9 se pueden ver dos cuadrados. En el de arriba, la columna más clara tiene casi exactamente la misma intensidad que el pixel más oscuro, mientras que en el de abajo, la columna mas clara tiene una intensidad notablemente distinta al pixel más oscuro.

5. **Buscar el centro**: Si el cuadrado es interesante, intentamos determinar la posición del potencial gusano. Partiendo del píxel más oscuro, se recorren en las diagonales iterando sobre los pixeles oscuros. Las 4 posiciones en horizontal y vertical más alejadas nos delimita 4 esquinas que forman un rectángulo donde se encuentra potencial gusano. Cuando hablamos del *centro del gusano* o *centro del potencial gusano* nos referimos al centro de este rectángulo.

   La Figura 10 muestra como se recorren las 4 diagonales buscando los

Figura 9: columna mas clara

pixeles oscuros. Empezando por el pixel más oscuro, se mueve hacia abajo-derecha, abajo-izquierda, arriba-derecha, arriba-izquierda, en ese orden. Luego, el centro del rectángulo delimitado es el centro del gusano.

Notar que no necesariamente encuentra todos los pixeles del gusano, ya sea porque algunos son muy claros, o como se puede observar en este caso, arriba a la izquierda los pixeles oscuros quedan desconectados (no adyacentes) por la calidad de la imagen.

Figura 10: Encontrando la forma del gusano

Una aclaración es que al iterar sobre los pixeles oscuros, a lo sumo se itera en horizontal tantos pixeles como el lado del cuadrado interesante, lo mismo en vertical. Esto nos evita posibles casos bordes donde se itera gran parte de la imagen total.

6. **Consultar repetido**: Se evalúa si la figura encontrada corresponde a un gusano ya identificado. Para esto simplemente se mide la distancia del centro de este potencial gusano contra el centro de todos los gusanos encontrados. Si está suficientemente lejos de todos los otros gusanos, consideramos que no corresponde a un gusano encontrado.

7. **Filtros**: Se define un cuadrado centrado en el centro del potencial gusano (del mismo tamaño de los que iteramos para buscar cuadrados interesantes). Sobre este cuadrado, se hacen la prueba de 3 filtros, que decidirán si consideramos que la figura es un gusano o no:

   7.1. **Filtro de *threshold***: se calcula el promedio y la desviación de las intensidades de los pixeles en el cuadrado. Usando estos valores, se hace un cálculo para ver si hay una diferencia notable entre las intensidades dentro del cuadrado. En caso contrario, probablemente se

Figura 11: Dos cuadrados encuentran el mismo gusano

trata de una mancha o una sombra, por lo que consideramos que el cuadrado no corresponde a un gusano.

En este filtro se aprovecha que los gusanos son una figura oscura en contraste con un fondo claro para ayudar a descartar algunas figuras o manchas que no sean un gusano.

En la figura 12 vemos que cerca del borde a pesar de haber diferencia entre la fila más clara y el pixel más oscuro, las intensidades no son tan distintas entre ellas.

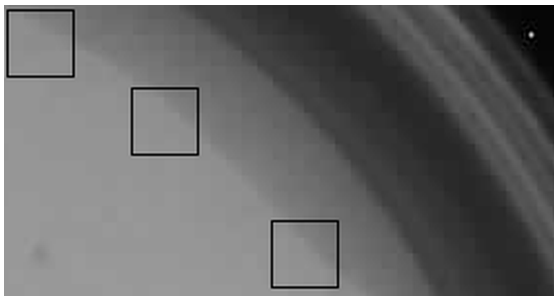


Figura 12: Cuadrados interesantes en los bordes

7.2. **Filtro de pixeles**: Se evalúa si la cantidad de píxeles oscuros del cuadrado está dentro de un rango aceptable. Consideramos que un pixel es oscuro si es más oscuro que la media de las intensidades del cuadrado.

7.3. **Filtro de longitud**: Se verifica que la longitud del gusano esté dentro de un rango aceptable. Para esto, se selecciona el mínimo y máximo índice en vertical y horizontal entre los pixeles identificados como oscuros al procesar el *filtro de pixeles*, y definimos la longitud del gusano como la distancia euclídea entre el mínimo índice en horizontal y vertical, y el máximo índice en horizontal y vertical.

8. **Cálculo de forma**: Si el gusano pasó las pruebas de los 3 filtros, se

Figura 13: Longitud del gusano

considera que efectivamente es un gusano. El paso siguiente es calcular un *resumen de la forma* del gusano.

Para esto se usa el rectángulo delimitado en el *filtro de longitud* (mínimo y máximo índice en horizontal y vertical). Se divide el rectángulo en 4x4 regiones uniformemente distribuidas, y se crea un cuadrado 4x4 donde a cada posición del cuadrado se le asigna el promedio de las intensidades de la región correspondiente en el rectángulo. Usando el promedio de los valores del cuadrado simplificado, se reemplaza cada valor del cuadrado por 1, si es mayor a este promedio calculado, y 0 a los que no. Luego, cada fila se convierte en un número binario de 4 bits, y se concatenan los números de las filas desde la de menor índice hasta la de mayor índice, efectivamente convirtiendo este cuadrado 4x4 en una cadena de 16 bits. A este número de 16 bits lo llamamos *la forma del gusano*.

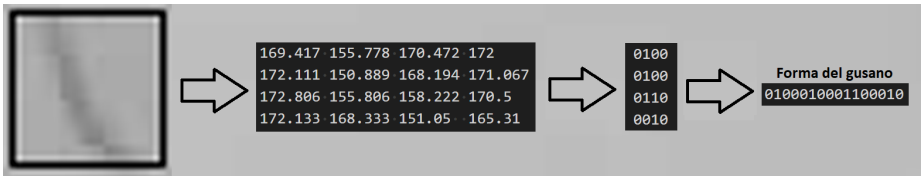


Figura 14: Calculo de la forma del gusano

9. **Escritura**: Finalmente, se escribe la información del gusano encontrado a un archivo. Esto incluye la posición de su centro, la forma que calculamos en el paso anterior, y algunos datos adicionales que fuimos calculando en los pasos anteriores.

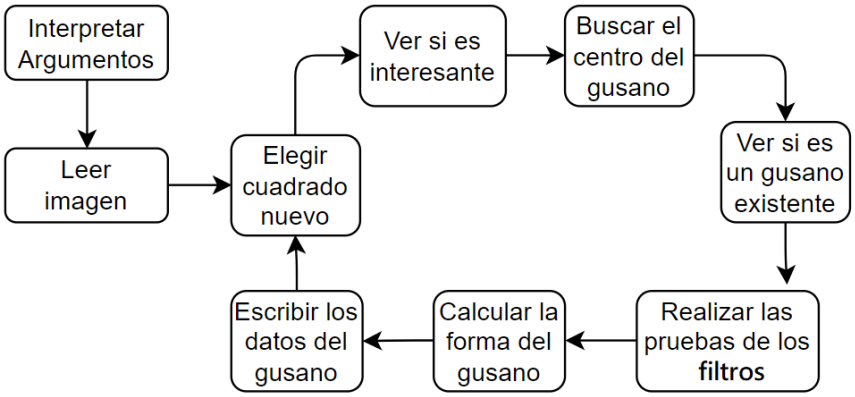


Figura 15: Resumen del algoritmo

3.3. Análisis de complejidad

Una vez identificada la estructura del código, se hace un análisis preliminar para identificar los cuellos de botella en el tiempo de ejecución. Para esto, uno de los aspectos a tener en cuenta es la complejidad teórica.

La complejidad teórica nos dice como escala la cantidad de operaciones que realiza un programa (y por lo tanto su tiempo de ejecución) en relación a las variables que lo afecta.

Algunas de estas *variables que lo afectan* son aleatorias o no del todo predecibles, por lo que se toman como referencia valores tomados de los experimentos reales para apoyar estas cuentas. Estas mediciones fueron realizados en una PC, por lo que las mediciones subjetivas al entorno (principalmente las mediciones de tiempo) pueden variar con respecto a los resultados que se evalúan más adelante en la Raspberry Pi.

Teniendo esto en cuenta, pasamos al análisis:

1. **Argumentos**: Interpretar los argumentos tiene complejidad despreciable, pues son pocos y no se usa ninguna lógica costosa

2. **Lectura**: Leer la imagen tiene como complejidad de la cantidad de píxeles: 1600 x 1200 $\simeq 2 \cdot 10^6$. Aprovechando que cada pixel es codificado por un número entre 0 y 255, se aprovecha una particularidad del lenguaje de programación: podemos interpretar cada pixel como un *caracter* o *número de 8 bit* en contraste al numero estándar (convencionalmente llamado número entero) de 32bits. Entonces, C++ al leer el bloque de memoria que representa nuestros pixeles, los lee como enteros por lo que efectivamente hace unas 4 veces menos operaciones de lo que uno esperaría.

   Entonces, esto resulta en alrededor de $5 \cdot 10^5$ operaciones.

3. **Ver subcuadrados**: Estos cuadrados los elegimos para que sean entre 30x30 y 40x40, pero para los fines de consistencia en el análisis, en todos los experimentos analizados usamos cuadrados de 35x35.

   La cantidad de operaciones que aporta esta iteración en total es la sumatoria de la cantidad de operaciones realizadas sobre cada cuadrado. Otra forma de ver esto, es como *cantidad de cuadrados · complejidad promedio de operaciones sobre un cuadrado.*

   Entonces, en los pasos siguientes se va a tener en cuenta la cantidad de cuadrados para definir bien la cantidad de operaciones que aportan. Esta cantidad de cuadrados es variable, pues solo se itera sobre el círculo que corresponde a la placa de petri (que no tiene la misma posición y tamaño en todas las imágenes, ya que es un parámetro que provee el usuario), pero teniendo en cuenta los experimentos seleccionados, en el peor caso hay cerca de 430 cuadrados.

4. **Prueba de interés**: Acá la cantidad de operaciones que consume es $\textit{Tamaño del lado del cuadrado} \cdot 4 \cdot 2 = \textit{Tamaño del lado del cuadrado} \cdot 8$, pues iteramos sobre 4 columnas (todos sus pixeles) y 4 filas (todos sus pixeles) dentro del cuadrado elegido. Tomando como tamaño de cuadrado 40, vemos que son $40 \cdot 8 = 320$ operaciones. Si tenemos en cuenta que hay 430 cuadrados, vemos que este paso aporta en total $320 \cdot 430$, cerca de $1{,}3 \cdot 10^5$ operaciones.

   Un dato que usaremos mas adelante es que la máxima cantidad de cuadrados interesantes encontrados fue 160.

5. **Buscar el centro**: Buscar el centro del potencial gusano: Este paso es más difícil de estimar porque depende de la cantidad de pixeles oscuros adyacentes al pixel más oscuro de un cuadrado, y que formen un camino en diagonal así que usamos la observación de que en el experimento donde este paso el costo fue máximo entre los experimentos analizados, hacemos cerca de $6 \cdot 10^4$ operaciones.

   Este paso en si no es tan costoso, siendo cerca de 10 veces menos operaciones que la lectura de la imagen.

6. **Consultar repetido** Detectar si el gusano ya fue encontrado: Teniendo en cuenta el dato que la máxima cantidad de gusanos encontrados fueron 40. Si suponemos que en cada cuadrado interesante se itera sobre todos los gusanos encontrados, tenemos que este paso aporta unas $40 \cdot 160 = 6400$ operaciones en total a la ejecución.

   Este número lo se considera despreciable siendo que aún en un caso pesimista más costoso que los experimentos vistos, hace cerca de 100 veces menos operaciones que leer la imagen y por lo tanto no va a afectar el tiempo de ejecución por más de un centésimo del tiempo que tarda en leer la imagen, aún menos notable en el tiempo total de ejecución (incluso si en cada cuadro interesante se detecta un gusano, sigue siendo $160 \cdot 160 \simeq 2{,}5 \cdot 10^4$ operaciones, cerca de 20 veces menos costoso que leer la imagen).

7. **Filtros**:

   7.1. **Filtro de *threshold***:

   En este paso se itera dos veces el tamaño de la caja, una para calcular el promedio y una para la desviación (Si bien se puede hacer en una sola iteración, nos costaría más operaciones durante esa iteración entonces a lo sumo no cambiaría notablemente). Si suponemos que la caja es de 40x40, entonces hacemos $160 \cdot 40 \cdot 40 \cdot 2$ veces en total.

   Para cada cuadro interesante (que no corresponde a un gusano encontrado previamente), iteramos el tamaño de la caja dos veces, obteniendo aproximadamente $5 \cdot 10^5$ operaciones, siendo potencialmente el primer paso igual o más lento que leer una imagen.

   En particular, en los experimentos realizados (teniendo en cuenta que el tamaño de cuadrado es de 35x35) se obtuvo que en el peor caso este filtro usa cerca de $3 \cdot 10^5$ operaciones.

   7.2. **Filtro de pixeles**:

   Este paso hace cerca de la misma cantidad de operaciones que el *filtro de threshold*, pues se itera sobre el cuadrado para calcular la cantidad de pixeles oscuros, pero en este paso lo hacemos una sola vez.

   Una aclaración importante, es que sólo hacemos esta verificación si ya pasamos el filtro anterior (en particular, necesitamos saber la intensidad promedio del cuadrado analizado). Entonces en la práctica, en el experimento donde más gusanos pasaron el chequeo del *filtro de threshold*, fueron 100.

   Asumiendo cuadrados de 40x40 la máxima cantidad de operaciones que nos cuesta este paso es alrededor de $98 \cdot 40 \cdot 40 \simeq 1{,}6 \cdot 10^5$ (En la practica, usando cuadrados de 35x35, fueron 127388,34 $\simeq 1{,}2 \cdot 10^5$ operaciones en el experimento donde este paso fue el más costoso).

   7.3. **Filtro de longitud**:

   Este paso no agrega un costo significativo, ya que al contar la cantidad de pixeles oscuros, también se guardan los 4 datos relevantes que se usan en este análisis. Si se quiere ser un poco más estricto, podemos tener en cuenta las 4 operaciones de mínimo/máximo que usamos para encontrar los *extremos del gusano*. Estas operaciones se hacen sobre cada intensidad marcada como oscuro en el filtro de pixeles. Nuevamente, el costo de este paso puede variar mucho según cada imagen, entonces observamos en el experimento donde este paso fue más costoso aporta alrededor de 127388,34$\cdot 4 \simeq 5 \cdot 10^5$ operaciones.

   Dicho esto, notar que son operaciones un poco más rápidas porque hacen buen uso de la cache al calcularse mientras se itera el filtro anterior. Un dato anecdótico fue que eliminando este paso resultó en tiempos de ejecución ligeramente más lentos (probablemente debido a que encuentra más gusanos), además de perder precisión en los resultados.

8. **Cálculo de forma**:

   Este es otro paso que depende de la forma del gusano, entonces usamos los datos de los experimentos para observar que en el peor caso, sólo consume cerca de $1{,}8 \cdot 10^4$ operaciones.

   Este es otro paso que consume cerca de 10 veces menos operaciones que leer una imagen por lo que no aporta una proporción notable al tiempo final de ejecución.

9. **Escritura**:

   En total, escribimos 17 bytes por gusano. Similar al lectura, el lenguaje de programación escribe de a bloques de 4 bytes, o *números de 32 bits*, por lo que podemos dividir el costo de operaciones de este paso en 4. Si tenemos en cuenta el peor caso donde encontramos 40 gusanos, tenemos cerca de $40 \cdot 17/4 \simeq 170$ operaciones, un número absolutamente despreciable en comparación con los pasos anteriores.

   Dicho esto, notar que este paso es **significativamente mas costoso** de lo que parece a simple vista, porque escribir valores a un archivo es mucho más lento que operar con números enteros (o incluso números de 8 bits).

   Como se escribe la información a medida que se encuentran los gusanos, cada una realiza una operación de escritura distinta, y las operaciones de escritura en C++ (y en general en cualquier lenguaje popular de programación) son muy costosas en tiempo de ejecución. Es difícil ver exactamente qué tan lento es este paso en particular, ya que la cantidad de operaciones no nos revela demasiada información, pero probamos eliminar la escritura del todo (solo las operaciones de escritura a archivo) en en promedio redujimos el 70 % del tiempo total de ejecución. Estas mediciones se pueden ver en la figura 16. Se obtuvieron ejecutando el código original, y el mismo sin las líneas que corresponden a la escritura del archivo.

Dataset:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
T-Write:	18.1	15.0	20.5	20.7	5.5	9.3	8.4	8.0	15.4	12.2	19.9	21.0	12.6	25.1	23.7	
T-No-Write:	3.3	3.3	3.7	3.5	2.6	4.4	3.5	3.5	3.9	3.9	5.0	4.5	4.3	4.5	4.7	
Difference:	18.1 %	21.9 %	18.0 %	17.1 %	46.8 %	47.8 %	42.0 %	43.8 %	25.1 %	31.8 %	25.2 %	21.5 %	34.4 %	17.9 %	20.0 %	28.7 %

Figura 16: Tiempo de ejecución con y sin escritura a archivo

   Esto es un dato notable porque si bien la cantidad de operaciones es un número despreciable en cuestión de operaciones de computadora, al final resulta en un porcentaje extremadamente alto del tiempo de ejecución total del programa.

Luego de este análisis preliminar, se estima que las partes que más tiempo consumen del tiempo de ejecución total (ya sea por mayoría de operaciones o por costo de las operaciones) son: La lectura del gusano, el cálculo de los filtros, y la escritura del gusano.

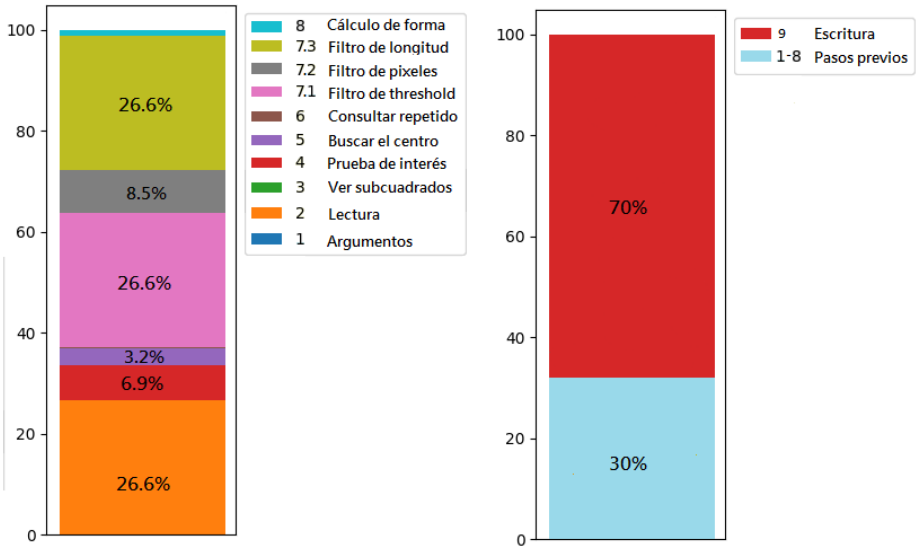


Figura 17: Visualización del costo de los pasos

Con la información recaudada entre la refactorización del código y el análisis de tiempo de ejecución, procedemos a buscar formas para optimizar el tiempo de ejecución.

4. Optimizaciones

4.1. Lectura, Escritura

De acuerdo al análisis preliminar del código, lo primero que se busca optimizar es la parte del proceso que corresponde a la escritura de la información de los gusanos, ya que corresponde a una gran parte del tiempo total de ejecución.

En esta parte se hicieron dos optimizaciones:

1. La primera, y la más importante es pasar de hacer una operación de escritura por gusano, a hacer una sola escritura al archivo.

   Para esto, se guardan localmente los bytes de cada gusano en un arreglo de caracteres (números de 8 bits), y justo antes de finalizar la ejecución del programa se escribe toda la información relevante de ese arreglo (Exactamente la cantidad de bytes correspondientes a los gusanos encontrados).

2. Reducimos la cantidad de bytes escritos por gusano.

   Creamos un encabezado al principio de la escritura, para decir cuantos gusanos se detectaron, a que índice de imagen dentro de la filmación co-

rresponde, y un *hash* que nos ayuda a detectar si hubo alguna corrupción o problema al escribir los datos.

Antes, cada gusano llevaba en su información el identificador de la foto, pero ahora ya no es necesario, efectivamente ahorrando 4 bytes de escritura por cada gusano encontrado. Así pasamos de ocupar 17 bytes por gusano a 13 bytes por gusano.

Si bien estos cambios no son una mejora notable en cantidad de operaciones del programa, realizamos una prueba (Resultados en la figura 18) y obtuvimos una mejora promedio de más del 300 % en tiempo de ejecución. Es decir, más de 3 veces más rápido.

Dataset:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
Multiple Writes:	18.1	15.0	20.5	20.7	5.5	9.3	8.4	8.0	15.4	12.2	19.9	21.0	12.6	25.1	23.7	
One Write:	3.9	4.0	4.4	4.1	3.2	5.1	4.0	4.2	4.5	4.7	5.7	5.2	5.0	5.1	5.5	
Improvement:	466.7 %	379.7 %	464.7 %	502.0 %	172.2 %	181.1 %	209.8 %	191.1 %	345.6 %	259.0 %	348.7 %	407.0 %	250.6 %	496.2 %	435.7 %	340.7 %

Figura 18: Optimización de escritura

Se probó además distintos métodos de escritura que nos provee el lenguaje: cout, printf, ofstream, pero el más veloz resultó ser el que se utilizaba actualmente, que es fwrite.

En cuanto a la lectura, similar al output, también se probaron distintos métodos de lectura: cin, scanf, ifstream, pero terminó ganando el que está actualmente que es fread.

Por último, existe una línea en C++, que se escribe asi:

```
ios::sync_with_stdio(false);
```

C++ esta diseñado para ser (en la mayoría de los casos) compatible con código de C. Un problema que surge de esta compatibilidad es que los métodos de *IO* (Input Output, es decir lectura y escritura) de C++: cin, cout ifstream, ofstream, funcionan de una manera diferente a los métodos de escritura tradicionales de C: scanf, printf, fread, fwrite.

Para evitar problemas al alternar entre ambos usos, existe una sincronización entre los distintos métodos de *IO*. Sin embargo, esta sincronización a veces resulta en una lectura/escritura más lenta. Como mencionamos más arriba, sólo usamos fread y fwrite para lectura y escritura, por lo que este método de sincronización no es necesario. La línea de C++ mencionada arriba desactiva esta sincronización, potencialmente acelerando el tiempo de lectura y escritura.

Sin embargo esta mejora no fue significativa en los experimentos probados.

4.2. Compilación

Habiendo terminado las optimizaciones de IO, las siguientes optimizaciones que se probaron apuntaban a aprovechar las distintas opciones de compilación de C++. En particular, el método de compilación actual era muy básico. Esencialmente se hace

```
g++ -o executablename filename
```

Mediante el uso de lo que se llama *flags* se puede modificar la forma que se compila el programa, es decir, la forma que el código escrito se convierte en instrucciones que interpreta el procesador. Estos cambios en la compilación pueden resultar en formas más (o menos) eficientes de realizar la misma lógica.

Por ejemplo, un código que hace

```
int res = 0;
for(int i = 0; i < n; i++) {
  res += i;
}
```

tiene exactamente el mismo efecto que un código que hace

```
int res = n * (n+1) / 2;
```

Dependiendo de la versión del compilador y los flags elegidos, se pueden aprovechar este tipo de comportamientos, que muchas veces no son muy visibles en el código, para acelerar el tiempo de ejecución.

Un flag de compilación muy conocido es el "O2" que mejora notablemente el tiempo de ejecución, sin efectos secundarios a excepción de algunos casos específicos. En particular, nos encontramos con un problema al usar esta optimización debido a las convenciones utilizadas para nombrar los archivos y directorios involucrados en el proyecto. Sin embargo, esto se pudo solucionar sin mayores inconvenientes con modificaciones menores en los nombres utilizados.

Entonces, la compilación ahora toma la forma de

```
g++ -O2 -o executablename filename
```

Sólo esta modificación de 3 caracteres nos mejoró un 160 % el tiempo de ejecución con respecto a la optimización de la escritura. En la figura 19 Vemos que en promedio corre cerca de un 500 % más rápido que el código original, y comparamos esto con la mejora del 340 % de la figura 18

Dataset:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
Multiple Writes:	18.1	15.0	20.5	20.7	5.5	9.3	8.4	8.0	15.4	12.2	19.9	21.0	12.6	25.1	23.7	
One Write:	3.9	4.0	4.4	4.1	3.2	5.1	4.0	4.2	4.5	4.7	5.7	5.2	5.0	5.1	5.5	
O2 Flag:	2.8	2.8	3.0	2.9	2.5	3.2	2.9	3.0	3.0	3.0	3.4	3.3	3.2	3.3	3.4	
Improvement:	644.5 %	529.4 %	683.0 %	713.7 %	219.9 %	286.2 %	295.6 %	270.6 %	516.4 %	400.9 %	577.5 %	636.1 %	392.2 %	770.8 %	695.0 %	508.8 %

Figura 19: Optimización del flag -O2

Con esto en mente, también se probaron otros argumentos de compilación, como O3 pero no mostraron mejoras notables.

Además de los *flags* de compilación, existen algunas líneas de código conocidas como *pragma* que cambian la forma que el compilador interpreta algunas partes del código, y que en los contextos correctos aceleran significativamente el tiempo de ejecución. Sin embargo, los que probamos tampoco aceleraron significativamente el tiempo de ejecución, en varios casos generando tiempos peores (Lo cual no es poco común si se aplican en contextos donde no corresponde usarlos).

4.3. Cambios en la la detección de gusanos

Luego de las optimizaciones previas, volvemos a analizar la distribución de las 3 partes principales del código: Lectura de la imagen, búsqueda de gusanos y escritura de los resultados.

Dataset:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
T1:	1.1	1.1	1.1	1.1	1.0	1.1	1.1	1.1	1.0	1.0	1.0	1.0	1.0	1.0	1.0	
T1 %:	39.2 %	39.1 %	36.3 %	36.5 %	40.7 %	32.8 %	37.5 %	36.7 %	33.1 %	32.8 %	29.0 %	30.7 %	30.4 %	31.2 %	29.6 %	34.4 %
T2:	1.7	1.8	1.9	1.8	1.4	2.2	1.8	1.8	1.9	1.9	2.3	2.1	2.1	2.1	2.2	
T2 %:	21.3 %	22.0 %	26.4 %	25.0 %	15.6 %	31.2 %	23.6 %	24.6 %	29.2 %	29.5 %	37.3 %	34.8 %	32.9 %	31.8 %	34.9 %	28.1 %
T3:	2.9	2.9	3.0	2.9	2.6	3.4	2.9	3.0	3.0	3.0	3.5	3.2	3.3	3.3	3.4	
T3 %:	39.5 %	38.8 %	37.3 %	38.6 %	43.7 %	36.1 %	39.0 %	38.7 %	37.6 %	37.7 %	33.7 %	34.5 %	36.8 %	37.0 %	35.5 %	37.5 %

Figura 20: Distribución del tiempo de ejecución

En la figura 20, las filas de T1 nos dicen el tiempo de ejecución luego de la lectura de la imagen, T2 nos dice el tiempo de ejecución luego de identificar los gusanos, pero antes de escribir los resultados a un archivo, y T3 es el tiempo final de ejecución luego de escribir los datos a un archivo.

Observamos que la parte del programa que se encarga de encontrar los gusanos (casi la totalidad de la lógica del código, sin contar unas pocas líneas de lectura y escritura) demora actualmente menos de 1/3 del tiempo total de ejecución. Esto no nos provee un panorama optimista con respecto a poder recortar notablemente en operaciones o tiempo de ejecución la búsqueda de gusanos, sin perder precisión de los resultados.

No obstante, se dedicó tiempo a intentar ver donde se puede recortar la cantidad de operaciones, o en su defecto aprovechar mejor la caché de la computadora[5] (La idea general, es que operaciones que usan memoria más cercana deberían funcionar mas rápido) u otras formas de usar operaciones menos costosas.

Estas fueron las modificaciones probadas:

- Al iterar para ver si el centro de un potencial gusano corresponde a un gusano ya encontrado, iteramos por los gusanos en orden de descubrimiento para ver si están lo suficientemente cerca para ser considerado el mismo. Una forma de ahorrar algunas operaciones acá es iterar desde el último encontrado hacia el primero. En particular si fue detectado en el cuadro anterior, terminaría casi instantáneamente, y si estamos una cierta cantidad de filas con menor índice al actual podemos dejar de buscar porque a partir de ahí todos los gusanos a revisar estarían lejos. Sin embargo, esto solo nos ahorra unas pocas operaciones en una parte poco costosa del problema.

- Otra pequeña optimización en términos de cache, no de operaciones totales, es que tenemos un buffer auxiliar al hacer el filtro del threshold para iterar las posiciones de un pequeño cuadrado de 30x30 a 40x40 en lugar de la imagen entera, ahorrando muchos *saltos de caché*. Este buffer no se usa más adelante, y se puede aprovechar en el cálculo del siguiente filtro y al calcular la forma del gusano.

Nuevamente, reducir un poco el tiempo de ejecución en estos pasos no resultan en un impacto notable en el tiempo total de ejecución del programa.

- Se redujeron algunas comparaciones al encontrar el centro del gusano. Antes, cuando se buscan las 4 esquinas, en cada paso se hace una consulta de mínimo y máximo en las 4 direcciones. Ahora, con esta optimización, cuando se itera en una diagonal solo se hace las 2 consultas de mínimo y máximo correspondientes. Por ejemplo, al iterar aumentando en vertical y horizontal, sólo se realizan las operaciones de máximo en ambos sentidos, evitando las dos de mínimo.

  En este caso, también estamos reduciendo ligeramente el tiempo de ejecución en un paso no tan costoso.

En la parte más significativa en cuanto a cantidad de operaciones, el cálculo de los filtros, se intentaron algunas modificaciones pero siempre obtuvieron una precisión peor.

Por ejemplo:

- Usar valores absolutos para decidir si un pixel es oscuro, en lugar de relativo al promedio de la caja donde se encuentra. Esto genera problemas, por ejemplo en la figura 21, hay pixeles en la mancha resaltada por el círculo rojo grande que son igual o más oscuros que los pixeles del gusano marcado por el círculo rojo pequeño. Entonces, o no diferenciamos los pixeles del gusano dentro de la mancha del resto de la mancha, o directamente no detectamos el gusano al costado.

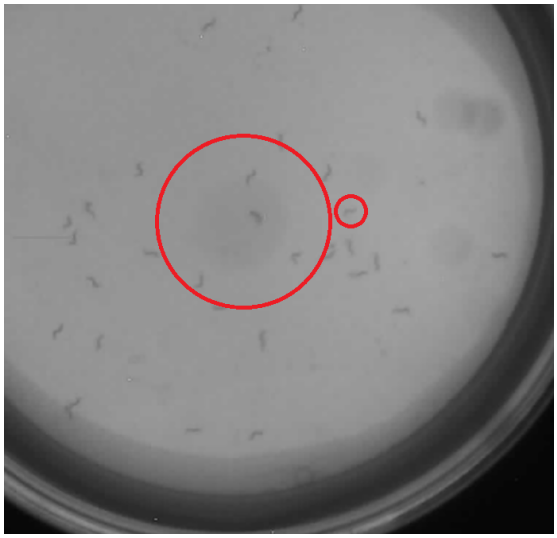


Figura 21: Hay pixeles en el círculo grande más oscuros que el gusano del círculo chico

- Memorizar pixeles correspondientes a gusanos ya encontrados para ahorrar preguntar si un gusano ya es existente. No generó una ganancia notable en tiempo de ejecución y a pesar de varias implementaciones diferentes del mismo concepto, siempre se terminó encontrando menos gusanos que antes.

- Cambiar de orden los filtros: Quizás primero verificar si la cantidad de pixeles oscuros era aceptable o la longitud, y luego el filtro de threshold mejoraría en tiempo de ejecución, ya que se detectarían antes más cuadrados que no son gusanos, y requeriría menos uso de los filtros siguientes. El problema con esto es que usamos el valor de la media del cuadrado para determinar si un pixel es oscuro o no, entonces no estamos ahorrando en operaciones al cambiar de orden los filtros.

- Con la misma idea del paso anterior, se probó sólo calcular la longitud del gusano si pasó el *filtro de pixeles*. Esto nos ahorraría varias operaciones, siendo que las 4 operaciones de mínimo/máximo se harían menos veces. Sin embargo, en la práctica no resulta en un tiempo de ejecución menor.

  En particular, se hizo una prueba en uno de los experimentos donde el promedio de fallas en el filtro de threshold era de 18, el de pixeles 59, y el de longitud 8, es decir, el filtro de pixeles detecta muchos cuadrados interesantes que no corresponden a gusanos. Sin embargo, hasta en este caso el tiempo de ejecución promedio aumentó ligeramente luego de aplicar el cambio. Esto se debe a que las operaciones dentro del bucle donde se iteran los pixeles del rectángulo interesante hacen uso eficiente de la cache, mientras que repetir el bucle dos veces es mucho mas lento, a pesar de que sólo en algunos casos se itera por segunda vez.

  Esto se probó cambiando de orden los filtros de longitud y pixeles, ya que usan valores independientes para sus cálculos pero no resultó en una diferencia notable en ninguno de los dos casos.

- Reducir operaciones en los bordes: Al ser oscuros los bordes de la imagen, es fácil identificarlos como cuadrados interesantes para luego descartarlos con los filtros.

  Se probaron distintas maneras de evitar llegar a considerarlos como cuadrados interesantes, pero cualquier lógica que se utiliza en los bordes termina corriendo en el resto de los cuadrados (aunque sea un solo *if*, es un *if* que se corre en todos los cuadrados del medio, que son más que los del borde) haciendo que no resulte en una mejora significante en el tiempo de ejecución. Además, en las pruebas no se identificaron algunos gusanos cerca del borde que antes si eran detectados.

  La conclusión de este paso es que al elegir el argumento que nos dicen en que posición esta la placa de petri dentro de la imagen, ser un poco más estrictos con lo que se considera que es el círculo nos ayuda a reducir un poco el tiempo de ejecución sin perder casi ningún gusano cerca del borde.

4.4. Otros

Otra potencial mejora que se consideró es la idea de paralelizar el código, que en teoría se espera que sea lograble pues muchas de las operaciones no dependen de resultados anteriores.

Sin embargo, no se ha puesto en práctica porque significa mucho trabajo debido a cómo esta implementado actualmente, sumado al hecho que algunas partes son paralelizables pero otras no. Por ejemplo, cada cuadrado se puede ver si es interesante en paralelo, pero para ver los filtros dependen de que sea un cuadrado interesante, y se necesita haber calculado el centro del potencial gusano. Esto complica aún mas el desarrollo de un algoritmo efectivamente paralelizable.

Además, sobre la Raspberry Pi donde se planea poner en marcha este problema, hay otros procesos corriendo, incluyendo un proceso que está atento a la interacción del usuario con el *front-end* (o visualización) y algunos otros procesos de Phylumtech, por lo que no nos garantiza que efectivamente se aprovechen en simultáneo los distintos núcleos que se necesitan para ver una mejora en el tiempo de ejecución.

Por estos motivos, se decidió que escapa el objetivo de este trabajo hacer una implementación que paralelice lo mejor que pueda los procesos que se llevan a cabo en este programa.

5. Análisis

Los números que se fueron presentando a lo largo del proyecto fueron obtenidos ejecutando el código en una PC. Esta decisión fue tomada por su velocidad superior, y conveniencia para el desarrollo en cuanto a herramientas visuales y otros factores.

Sin embargo, el objetivo de este programa es obtener un programa que se ejecute en una Raspberry Pi. Esto puede presentar diversas diferencias con los resultados anteriores, principalmente en la medición del tiempo de ejecución, ya que la lógica del código es la misma.

Además, la PC de uso personal del autor tiene el sistema operativo Windows, corriendo los programas en el *subsistema de Ubuntu para windows* (wsl), con potencialmente varios procesos corriendo de fondo que pueden interferir con las mediciones. Mientras tanto, la Raspberry Pi tiene una instalación fresca de un sistema operativo liviano, por lo que los resultados deberían ser más consistentes.

En esta sección, vamos a analizar los resultados obtenidos sobre la Raspberry Pi, y evaluando algunas diferencias notables con los resultados obtenidos en PC.

5.1. Mediciones

En la figura 22 se muestran algunos resultado relacionados a las optimizaciones que fueron probadas anteriormente. A continuación se explica como interpretar la tabla.

	Dataset:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
	G:	25.1	21.9	30.4	34.0	4.5	7.3	8.1	8.2	18.7	15.9	26.7	29.7	14.8	37.4	30.5	
	SQ	45.3	57.0	84.3	76.5	31.1	126.2	63.5	77.9	96.9	98.5	158.3	133.0	120.1	117.9	134.0	
	DD	15.7	17.7	21.7	16.6	3.3	5.5	7.3	6.2	13.3	12.2	20.8	24.9	8.8	16.7	17.7	
	DTH	4.3	11.0	14.8	21.1	11.5	81.3	13.1	59.7	49.0	10.8	64.5	14.9	63.9	20.2	18.1	
	DP	0.2	5.4	14.9	4.2	9.3	25.7	33.3	2.3	14.0	59.4	44.5	61.5	28.1	43.1	59.4	
	DL	0.0	1.8	1.9	0.6	2.4	6.5	1.7	1.6	1.1	0.1	1.7	1.9	5.0	0.4	8.2	
1	T:	43.2	44.3	47.1	46.4	37.9	54.5	46.5	46.2	50.7	50.7	61.9	57.1	54.8	55.4	58.1	50.3
2	T:	22.6	23.8	28.9	27.0	18.9	37.6	27.4	28.2	32.6	33.3	44.2	39.3	37.9	36.9	40.6	31.95
	Gain:	191.0 %	186.2 %	162.7 %	172.1 %	200.7 %	145.1 %	169.5 %	163.7 %	155.2 %	152.6 %	140.0 %	145.3 %	144.6 %	150.1 %	143.3 %	161.5 %
3	T1:	4.9	4.9	4.8	4.9	4.9	4.8	4.9	4.9	4.8	4.8	4.8	4.8	4.9	4.9	4.8	4.87
	T2:	25.1	26.5	31.1	29.7	20.2	41.3	29.7	30.2	34.8	36.7	49.2	43.7	41.9	41.1	45.3	35.09
	T3:	25.4	26.8	31.4	30.0	20.4	41.6	30.0	30.4	35.0	36.9	49.5	44.0	42.2	41.4	45.5	35.36
	Gain:	170.4 %	165.5 %	150.0 %	155.0 %	185.9 %	131.1 %	155.2 %	151.9 %	144.7 %	137.4 %	125.1 %	129.8 %	129.9 %	133.9 %	127.7 %	146.2 %
4	T1:	4.8	4.8	4.8	4.8	5.1	4.8	4.8	4.9	4.9	4.8	4.9	4.8	4.9	4.9	4.8	4.86
	T2:	11.1	11.2	12.6	11.9	9.8	15.0	12.0	12.2	13.6	13.7	17.4	15.9	15.4	15.1	16.1	13.54
	T3:	11.3	11.5	12.8	12.2	10.1	15.3	12.2	12.5	13.8	14.0	17.7	16.1	15.6	15.4	16.3	13.78
	Gain:	381.2 %	386.1 %	367.2 %	381.2 %	377.1 %	357.3 %	380.8 %	370.5 %	366.8 %	362.8 %	350.4 %	353.9 %	350.9 %	359.7 %	355.9 %	366.8 %
5	T1:	4.9	4.9	4.9	4.9	5.1	4.9	4.9	4.9	5.0	4.9	4.9	5.0	5.0	4.9	5.0	4.95
	T2:	11.1	11.3	12.5	11.8	9.6	14.8	11.9	12.1	13.5	13.5	17.0	15.7	15.3	14.8	15.9	13.39
	T3:	11.4	11.5	12.8	12.0	9.9	15.0	12.1	12.3	13.7	13.8	17.3	16.0	15.6	15.0	16.2	13.64
	Gain:	380.1 %	385.3 %	368.1 %	385.6 %	384.4 %	363.0 %	383.1 %	375.0 %	369.4 %	368.3 %	357.7 %	357.3 %	351.4 %	368.6 %	360.0 %	370.5 %
6	T1:	4.9	4.9	4.9	4.9	5.2	4.9	4.9	4.9	5.0	4.9	4.9	5.0	5.0	5.0	4.9	4.95
	T2:	11.4	11.4	12.6	12.0	9.8	14.8	11.9	12.2	13.6	13.6	17.1	15.9	15.3	15.1	15.9	13.50
	T3:	11.6	11.6	12.9	12.3	10.1	15.1	12.2	12.4	13.8	13.8	17.3	16.1	15.5	15.3	16.2	13.75
	Gain:	371.1 %	380.5 %	365.1 %	378.3 %	376.8 %	361.8 %	382.0 %	371.9 %	366.7 %	367.5 %	357.5 %	353.9 %	352.6 %	361.7 %	359.9 %	367.2 %
7	T1:	4.9	4.9	4.9	4.9	5.1	4.9	4.9	4.9	4.9	4.9	4.9	4.9	4.9	5.0	4.9	4.93
	T2:	11.3	11.3	12.7	12.0	9.7	15.0	12.0	12.1	13.5	13.7	17.1	16.0	15.3	15.3	16.1	13.53
	T3:	11.5	11.6	13.0	12.2	9.9	15.2	12.3	12.3	13.8	14.0	17.4	16.2	15.5	15.5	16.3	13.78
	Gain:	375.1 %	383.3 %	363.5 %	379.4 %	382.4 %	358.5 %	379.2 %	375.8 %	367.3 %	363.0 %	356.5 %	352.7 %	353.2 %	356.9 %	356.6 %	366.9 %
8	T:	10.8	11.2	11.9	11.6	9.7	14.9	11.9	12.2	13.2	13.4	16.9	15.3	15.1	14.5	15.5	13.21
	Gain:	398.3 %	397.1 %	395.0 %	401.3 %	390.4 %	366.7 %	390.9 %	379.7 %	382.6 %	378.7 %	366.4 %	372.7 %	361.7 %	383.5 %	374.4 %	382.6 %
9	T:	10.1	10.3	11.2	10.0	9.0	12.8	11.0	11.1	11.6	11.8	14.3	13.4	13.1	12.5	13.3	11.71
	Cost:	426.5 %	432.1 %	420.3 %	463.5 %	420.0 %	426.0 %	423.6 %	417.3 %	435.2 %	431.1 %	432.2 %	425.7 %	417.8 %	442.0 %	435.8 %	429.9 %
10	T2:	22.6	23.4	28.5	27.1	18.7	37.5	27.4	28.0	31.7	33.3	44.2	39.4	37.8	37.0	40.6	31.82
	Cost:	52.2 %	52.8 %	60.6 %	58.3 %	49.3 %	68.9 %	58.9 %	60.6 %	62.5 %	65.6 %	71.4 %	69.0 %	69.1 %	66.8 %	69.9 %	62.4 %

Figura 22: Compilación de resultados en Raspberry Pi

Las columnas corresponden a los diferentes experimentos, enumerados del 1 al 15 en la primera fila.

Sobre cada experimento, se muestran una serie de métricas, correspondientes al promedio sobre todas sus imágenes:

- G: Gusanos detectados
- SQ: Cuadrados interesantes
- DD: Veces que un cuadrado interesante corresponde a un gusano ya encontrado
- DTH: Veces que se calculó el filtro de threshold
- DP: Veces que se calculó el filtro de pixeles
- DL: Veces que se calculó el filtro de longitud

Estos valores no cambian a lo largo de las distintas (posibles) optimizaciones presentadas más abajo en la tabla.

Luego, se muestran mediciones correspondientes a 10 versiones del programa.

La primera columna de la izquierda nos muestra a qué versión de código corresponden los datos. La última columna a la derecha nos dice el promedio de los valores mostrados en la fila correspondiente.

En la segunda columna de la izquierda, se describen los valores que fueron evaluados sobre los distintos experimentos:

- T: El tiempo total de ejecución del programa
- T1: El tiempo apenas termina la lectura de la imagen
- T2: El tiempo cuando se termina la búsqueda de gusanos, pero antes de la escritura al archivo
- T3: El tiempo final de ejecución (La única diferencia con el ítem anterior es que se hace una escritura a archivo.
- Gain: La ganancia de tiempo sobre cada experimento con respecto al tiempo de ejecución del código original. Al final de estas filas, se destaca en amarillo la ganancia promedio sobre todos los experimentos
- Cost: El porcentaje del total del tiempo de ejecución que representó el paso correspondiente. Al final de esta fila, se destaca en amarillo el costo promedio sobre todos los experimentos

Estos tiempos se eligieron por ser los más significativos según el análisis preliminar del código.

Las 10 versiones del programa son las siguientes:

1. El programa con el que se comenzó el trabajo, sin modificar.
2. El mismo que el ítem anterior, sólo cambiando la mínima cantidad de líneas para hacer una sola escritura al final (en lugar de una escritura por gusano).
3. El código refactorizado. Se espera que tenga el mismo comportamiento que el código 2 (excepto los casos excepcionales donde saltaban los bugs del código 2), solo es el resultado de re-escribir el código para mejorar su entendimiento (de hecho, el código 2 se hizo luego del código 3, donde se probó por primera vez la optimización de una sola escritura al output).

   Cabe destacar que por los cambios del output y bugs, quizás el código 1 y 2 no hacen exactamente lo mismo que el código 3, pero su tiempo de ejecución debería ser lo suficientemente preciso para los fines de este análisis.
4. El mismo código del paso anterior, pero de ahora en adelante compilando con el flag -O2
5. El código del ítem anterior con las *pequeñas optimizaciones*:
   - Desactivar la sincronización con stdio.

- Usar el arreglo pequeño que creamos para calcular el filtro de threshold en los pasos siguientes para mejorar el uso de la cache.
- optimizar la revisión de gusanos repetidos.
- reducir las operaciones de mínimo/máximo al encontrar el centro del gusano

Consideramos estos 4 cambios en un mismo código porque individualmente su diferencia era lo suficientemente pequeña para ser despreciable, pero nos interesa analizar su efecto acumulado.

6. El código del ítem anterior, donde sólo hacemos las operaciones de mínimo máximo asociadas al filtro de longitud si el gusano aprobó el filtro de pixeles (En lugar de calcular ambos en la misma iteración del cuadrado). En este código calculamos primero el filtro de pixeles y luego el de longitud

7. Similar al ítem anterior pero primero calculamos el filtro de longitud y luego el de pixeles.

8. El código 5., pero eliminando algunas líneas de código que solo servían para este análisis. Por ejemplo, no se cuenta la cantidad de cuadrados interesantes, ni se mide el tiempo a lo largo del código, solo se mide luego de la escritura (es decir, el final oficial del programa).

9. El código 8., agregando dos líneas de *pragma* que modifican su compilación:.

```
#pragma GCC optimize("Ofast")
#pragma GCC optimize ("unroll-loops")
```

El pragma *Ofast* habilita algunas optimizaciones adicionales del compilador[6]. Tiene ciertos riesgos en contextos específicos pero no generó diferencias en los resultados del código.

El pragma *unroll-loops* habilita un fenómeno conocido como *loop unrolling*[7] que escencialmente reduce la cantidad de iteraciones en un bucle de iteración, agrupando distintas iteraciones en una. Una forma de visualizar esto es mediante el siguiente código:

```
// Antes:
for(int i = 0; i < n; i++)
    hacer_algo(i);
// Ahora:
for(int i=0; i < n; i += 2)
    hacer_algo(i);
    hacer_algo(i+1);
```

10. El código 1. (sin el flag de O2), pero eliminando las líneas que corresponden a la escritura de la información de los gusanos (es decir, esa información se pierde al terminar la ejecución del programa).

Una observación notable de la figura 22, es que los tiempos de lectura varían ligeramente en códigos donde no se modificó ninguna lógica que debiera afectarla. En general la medición de los tiempos tiene un cierto margen de error. Esto se debe a varios motivos:

- El código que mide el tiempo de ejecución. Se usan valores de alta resolución pero no es una medición perfecta.
- El contexto de ejecución del programa: Algunos procesos de fondo pueden influenciar en el estado de la cache y memoria en un momento dado.
- El error de precisión que viene de hacer cálculos en punto flotante, que se acumula a lo largo de las distintas etapas.

En general, este error de precisión varia entre 1 y 5 ms en los intentos medidos. Esto dificulta a veces la interpretación de los resultados, en especial en los casos donde el tiempo de ejecución varía muy poco, por lo que distintas ejecuciones pueden obtener diferentes resultados.

A continuación se explica más en detalle como se calcula el tiempo de ejecución, y algunas medidas utilizadas para ayudar a disminuir el impacto de este error de precisión.

1. En el código, se toman los clocks al principio del bloque del main, con el siguiente comando:

   ```
   clock_x begin = chrono::high_resolution_clock::now();
   ```

   donde clock_x se define como

   ```
   std::chrono::time_point<std::chrono::high_resolution_clock>
   ```

   Esto se debe a que los tiempos de ejecución son tan bajos que la precisión del clock_t estándar de C++ no es suficiente para nuestros análisis. Entonces se optó por esta variante de mayor precisión.

2. Más adelante en el código, se calcula el tiempo en un momento determinado de la siguiente manera

   ```
   clock_x now = chrono::high_resolution_clock::now();
   std::chrono::duration<double> elapsed = now - begin;
   double time = elapsed.count();
   ```

   Con esto obtenemos en la variable *time* el tiempo transcurrido.

3. Al finalizar la ejecución del programa, se imprime el valor de tiempo calculado, para no influenciar el tiempo de ejecución del resto del programa (Sólo se agregan las operaciones de iniciar el conteo del clock, y calcular el tiempo transcurrido, que son despreciables en el tiempo de ejecución total del programa).

4. Estos tiempos, se calculan para cada imagen de la filmación para obtener el tiempo promedio sobre un experimento.

5. Por último, debido a la variabilidad en los resultados, para cada experimento se calcularon los tiempos 10 veces, y el valor final es el promedio de estos 10 resultados.

5.2. Análisis de resultados

Con los resultados de la figura 22 vemos una diferencia notable con los resultados preliminares en la PC:

- En el código original ejecutado en PC, el tiempo de escritura consumía un 70 % del tiempo total de ejecución, mientras que en la Raspberry es menos del 40 %. Entonces en la PC la optimización de escritura resultó en una mejora del 360 % (más de 3 veces más rápido), mientras en la Raspberry sólo fue cerca del 160 % (Menos del doble).

- En contraste, la optimización del *flag* -O2 en la PC obtenía una ganancia de 160 %, mientras que en la Raspberry fue poco más del 200 %. Es decir, fue más significativa la optimización del O2 que la optimización de escritura.

Otras diferencias entre la PC y la Raspberry fueron:

- Los *pragma* fueron probados en PC, pero resultaron en un tiempo de ejecución ligeramente peor. Sin embargo, en la Raspberry Pi resultaron en una mejora de más del 45 % con respecto al mismo código donde no se usaron. Esto se debe a que son optimizaciones de bajo nivel por lo que pueden presentar diferencias notables en su desempeño dependiendo del compilador y la arquitectura donde se utilizan.

- La versión 8. fue ligeramente más lento en PC que la versión 5., con una diferencia de menos del 1 %, probablemente debido al error de precisión en la medición de los tiempos.

  Sin embargo, en la Raspberry consiguió una mejora del 12 %.

  Posiblemente se debe a que la Raspberry tiene menos memoria disponible, por lo que le afecta más ocuparla con las pequeñas cuentas auxiliares realizadas, mientras que en la PC no alcanza a ser relevante.

En el resto de las mediciones, los resultados de la PC sí fueron más similares a los de la Raspberry:

- En la re-factorización del código, se agregan más llamados a funciones, algunos anidados entre ellos, y se crean algunas estructuras de datos para mejorar la legibilidad del código y facilitar su mantenimiento. En particular, no hubo cambios significativos en el código en cuanto a eliminar variables no usadas o eliminar código innecesario. Esto a priori puede dar la sensación de que la mejora en legibilidad debe ser a costo de un peor tiempo de ejecución, pero resulta que no es el caso.

En las mediciones obtenidas, vemos que mejora en promedio cerca de un 20 % con respecto al código que hace lo mismo sin refactorizar.

Esto se debe principalmente a que al declarar las variables más localmente, se aprovecha mejor el uso de la cache. No se debe buscar memoria alocada previamente en posiciones posiblemente lejanas y se elimina la memoria local al final de la ejecución de cada función reduciendo ligeramente el consumo de memoria.

- El resto de las pequeñas optimizaciones, aplicadas en simultáneo mostraron una mejora minúscula, cerca del 4 % con respecto al código anterior. Esto coincide con los resultados esperados, ya que no eran cambios en las partes más costosas del código.

5.3. Observaciones

	Dataset:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
Raspbi	T1:	4.9	4.9	4.9	4.9	5.1	4.9	4.9	4.9	5.0	4.9	4.9	5.0	5.0	4.9	5.0	
	T1 %:	43.3 %	42.8 %	38.6 %	40.8 %	51.5 %	32.7 %	40.7 %	40.0 %	36.1 %	35.7 %	28.5 %	31.0 %	32.0 %	32.9 %	30.7 %	37.0 %
	T2:	11.1	11.3	12.5	11.8	9.6	14.8	11.9	12.1	13.5	13.5	17.0	15.7	15.3	14.8	15.9	
	T2 %:	54.6 %	55.1 %	59.3 %	57.1 %	45.6 %	65.6 %	57.3 %	58.0 %	62.1 %	62.6 %	69.7 %	67.5 %	66.4 %	65.4 %	67.7 %	61.1 %
	T3:	11.4	11.5	12.8	12.0	9.9	15.0	12.1	12.3	13.7	13.8	17.3	16.0	15.6	15.0	16.2	
	T3 %:	2.18 %	2.16 %	2.07 %	2.04 %	2.91 %	1.64 %	2.05 %	2.00 %	1.80 %	1.74 %	1.74 %	1.54 %	1.59 %	1.70 %	1.55 %	1.87 %
PC	T1:	1.1	1.1	1.1	1.1	1.0	1.1	1.1	1.1	1.0	1.0	1.0	1.0	1.0	1.0	1.0	
	T1 %:	39.2 %	39.1 %	36.3 %	36.5 %	40.7 %	32.8 %	37.5 %	36.7 %	33.1 %	32.8 %	29.0 %	30.7 %	30.4 %	31.2 %	29.6 %	34.4 %
	T2:	1.7	1.8	1.9	1.8	1.4	2.2	1.8	1.8	1.9	1.9	2.3	2.1	2.1	2.1	2.2	
	T2 %:	21.3 %	22.0 %	26.4 %	25.0 %	15.6 %	31.2 %	23.6 %	24.6 %	29.2 %	29.5 %	37.3 %	34.8 %	32.9 %	31.8 %	34.9 %	28.1 %
	T3:	2.9	2.9	3.0	2.9	2.6	3.4	2.9	3.0	3.0	3.0	3.5	3.2	3.3	3.3	3.4	
	T3 %:	39.5 %	38.8 %	37.3 %	38.6 %	43.7 %	36.1 %	39.0 %	38.7 %	37.6 %	37.7 %	33.7 %	34.5 %	36.8 %	37.0 %	35.5 %	37.5 %

Figura 23: Tiempo de ejecución en PC vs Raspberry

En la figura 23, se tomaron las mejores versiones en PC y en Raspberry (ambas siendo el código 5.), para evaluar luego de las optimizaciones cómo se distribuye el tiempo entre las 3 partes principales del programa: Lectura de la imagen, búsqueda de gusanos, escritura de la información (Sin contar las versiones 8. y 9., ya que no se analizó el tiempo en las distintas etapas).

Podemos observar que en la Raspberry, el tiempo de escritura, que originalmente consumía en promedio cerca del 40 % del tiempo de ejecución ahora consume menos del 1.87 %, sin llegar en ningún caso a consumir más del 3 %. Luego de las optimizaciones, ya no es más un paso relevante en el tiempo de ejecución total en la Raspberry.

Sin embargo, en PC sigue costando un 37 %, siendo el más costoso de los 3 pasos. Esto puede deberse a la ineficiencia del subsistema de Ubuntu para Windows, pero un análisis más profundo de este paso puede ser relevante cuando se desee incorporar el detector de gusanos a un programa que corra en otro sistema operativo.

Si se soluciona la diferencia del porcentaje de tiempo que ocupa la escritura entre PC y Raspberry, probablemente significa que también sería más rápido la lectura de la imagen, ya que son procesos similares.

En particular, usando los datos obtenidos en la Raspberry, la parte del código que se encarga de encontrar los gusanos se encuentra cerca del 60 % del tiempo

total de ejecución. Si tenemos en cuenta que el código optimizado demora cerca del 23 % del tiempo del código original (desde un promedio de 50.3 hasta un 11.7), futuras optimizaciones en la lógica de la búsqueda de los gusanos nos puede dar hasta un 14 % de mejora de tiempo con respecto al código original con el que se inició este trabajo (Y esto es si casi no se consume tiempo para detectar los gusanos, lo cual significa que probablemente la precisión decaiga notablemente).

Se consideró probar filtros nuevos o diferentes (que es la lógica más pesada en el código optimizado) para ayudar a reducir el tiempo de detección de los gusanos, pero se decidió que escapa los objetivos de este trabajo. No sería una mejora notable, además del hecho que no podemos garantizar que efectivamente obtengamos una solución mejor que no pierda precisión.

Si bien creemos que teóricamente esto se puede mejorar, no fue algo que pudimos resolver en este proyecto.

5.4. Limitaciones conocidas

- El algoritmo pierde precisión cuando los gusanos están muy cerca entre ellos. Algunas problemas que genera esto son:
  - A veces considera a los gusanos como *ya encontrados* cuando el centro se encuentra muy cercano al de un gusano detectado, o cuando intersecan.
  - Al haber varios gusanos en un solo cuadro interesante, solo elige el pixel más oscuro para analizar un potencial gusano, entonces algunos nunca se encuentra (como se observa en la figura 24).

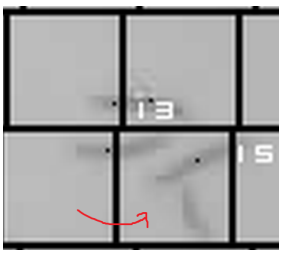


Figura 24: Ejemplo donde se pierden gusanos por proximidad

    Dicho esto, en la práctica pocas veces termina siendo la causa de no encontrar un gusano, porque muchas veces los gusanos están entre varios cuadrados, entonces uno de los cuadrados a donde pertenece los termina encontrando (como se observa en la figura 25).
  - Se mezclan al calcular los filtros al estar dentro del cuadrado que se analiza, identificando más pixeles oscuros de los que debería y posiblemente interfiriendo en el cálculo de su longitud.

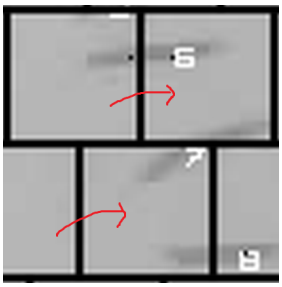


Figura 25: Ejemplo donde **no** se pierden gusanos por proximidad

- Otra limitación es que no se detectan los gusanos cuando se encuentran muy próximo al perímetro de la placa de petri (O sobre el mismo).

  A veces es *inevitable* porque incluso a ojo humano tampoco se pueden ver, pero aún en los casos donde son más visibles, al estar cerca del perímetro del circulo se agrega demasiado ruido al cálculo de los filtros.

  Si acomodamos los hiperparámetros para encontrar estos gusanos cerca del perímetro, termina resultando en un aumento notable de las falsas detecciones, empeorando la precisión general.

En particular, los gusanos que se encuentran cerca del borde de la placa de petri, y los gusanos que se superponen entre ellos generan pérdidas de precisión. Sin embargo, este fenómeno también se observa en otros métodos del estado del arte, incluyendo modelos más complejos que se utilizan en la industria.

5.5. Fortalezas del programa

- En los casos donde las limitaciones mencionadas no ocurren, el algoritmo funciona con una alta precisión.

  Es muy bueno para diferenciar gusanos de manchas o estructuras de pixeles oscuros que no corresponden a gusanos, aprovechando fuertemente las características esperadas del gusano en contra de las características esperadas del fondo para identificar a los gusanos y descartar a los que no lo son.

- Es totalmente determinista Cualquier fallo que pudiera haber se puede identificar (o al menos los desarrolladores de Phylumtech que tengan acceso al código) de donde viene y si se puede solucionar. Para una misma imagen y con los mismos hiperparámetros (o argumentos de compilación) va a dar siempre el mismo resultado

- Es muy veloz y fácil de utilizar. Phylumtech provee una interfaz donde con unos pocos clicks se puede entrenar los hiperparámetros del modelo sobre un experimento. Luego, los resultados del proceso entero, incluyendo la obtención de las imágenes se pueden ver en tiempo real.

- A diferencia de otros modelos tradicionales como *Machine Learning*, no necesitamos conseguir grandes cantidades de datos bien descriptos para poder entrenar y poner en funcionamiento al modelo.

6. Conclusiones

La empresa Phylumtech desarrolló un dispositivo para el seguimiento de gusanos en tiempo real. Una parte de este proceso consiste en la detección de gusanos sobre una imagen. En este trabajo, buscamos obtener resultados igual o más precisos (que detecte la misma cantidad o más de gusanos, con igual o menos falsas detecciones), con un tiempo de ejecución menor al actual.

El requerimiento nació de un cambio de tecnología, migrando de una PC a una Raspberry Pi, e incorporando el uso de múltiples cámaras en simultáneo, por lo que la optimización del algoritmo influye en la cantidad de cámaras que se pueden utilizar.

Se consiguió una mejora promedio de cerca del 430 %, es decir, más de 4 veces más rápido.

Para lograr esto, se comenzó con un análisis para identificar las partes del algoritmo que más inciden en el tiempo de ejecución, donde se observó que había una proporción similar del tiempo total de ejecución entre la lectura de la imagen, la búsqueda de los gusanos, y la escritura de los resultados.

Se redujo el impacto del tiempo de escritura pasando de ocupar cerca del 40 % del tiempo total de ejecución (en la Raspberry Pi) a menos del 2 %. Además, se aprovecharon algunas características del lenguaje C++ para modificar la forma en la que se compila el código, incluyendo *flags* de compilación y *pragmas*. Estos cambios, junto con unas pequeñas optimizaciones en la implementación del programa nos permitió obtener los mismos resultados que antes con menor tiempo de ejecución, por lo que también se cumplió el requisito de que la optimización resultante sea *igual o más precisa*.

Si bien creemos que se puede mejorar el tiempo de ejecución con algunas modificaciones en el funcionamiento de la búsqueda de los gusanos, esta mejora no se espera que sea significativa. En particular, la forma más directa de mejorar el tiempo de ejecución es con una mejora en la arquitectura física donde se ejecuta el programa. Por ejemplo, el tiempo de lectura de la imagen forma una proporción notable del tiempo total del programa, por lo que un dispositivo con lectura de datos más veloz (por ejemplo una mejor *SSD*) nos puede hacer una diferencia notable.

Con estas mejoras en mente, como trabajo futuro podrían evaluarse alternativas en el programa que resulten en un tiempo de ejecución ligeramente más lento, pero que puedan conseguir una precisión *mejor* que la actual, sin perder el objetivo de procesar la cantidad deseada de imágenes en *real time*.

Otro aspecto posible a explorar puede ser la evaluación de la *performance* en distintas arquitecturas. No sólo en el desempeño del programa en las distintas arquitecturas, sino posibles cambios que se adapten específicamente a la arquitectura utilizada.

Referencias

[1] Shen K. Ou CY. Neuronal polarity in c. elegans. 2011.

[2] Denise S. Walker, Yee Lian Chew, and William R. Schafer. 151Genetics of Behavior in C. elegans. In *The Oxford Handbook of Invertebrate Neurobiology*. Oxford University Press, 04 2019.

[3] Bazzicalupo P. Hilliard MA, Bargmann CI. C. elegans responds to chemical repellents by integrating sensory inputs from the head and the tail. 2002.

[4] Nash Z. et al. Stroustrup N., Ulmschneider B. The caenorhabditis elegans lifespan machine. 2013.

[5] R. Stacpoole and T. Jamil. Cache memories. *IEEE Potentials*, 19(2):24–29, 2000.

[6] gcc. Documentación de gcc de optimizaciones. url: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html.

[7] Chapter 12 - toolchain primer. In Jim Kukunas, editor, *Power and Performance*, pages 207–239. Morgan Kaufmann, Boston, 2015.