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

OPTIMIZACIÓN Y PARALELIZACIÓN DE UN ALGORITMO DE GENERACIÓN DE SKELETONS A PARTIR DE MALLAS GEOMÉTRICAS APLICADO A ESTRUCTURAS BIOLÓGICAS

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

IVÁN YERKO ROJAS HERNÁNDEZ

PROFESOR GUÍA:
NANCY HITSCHFELD KAHLER

MIEMBROS DE LA COMISIÓN:
STEFFEN HÄRTEL
RODRIGO PAREDES MORALEDA

Este trabajo ha sido financiado por el proyecto Fondecyt 1120495

SANTIAGO DE CHILE
2014

Resumen

El estudio cuantitativo de estructuras microscópicas 3D requiere de herramientas computacionales, tanto para realizar mediciones como para su visualización, dada su complejidad y gran volumen. Una de las herramientas para medir y visualizar estas estructuras es el *skeleton*. Un *skeleton* es la representacion simplificada de la estructura en forma de grafo, compuesta por nodos y segmentos. Si bien existen múltiples algoritmos para su generación, estos buscan generalmente mantener propiedades topológicas y geométricas del objeto de estudio.

Actualmente se cuenta con la implementación de un algoritmo de generación de *skeletons* [2], basado en el algoritmo propuesto por Au et al [3]. Esta implementación, si bien entrega resultados satisfactorios, presenta tiempos de cálculo muy extensos. Dado lo anterior, es que esta memoria tiene como objetivo analizar y optimizar el tiempo de ejecución de esta implementación.

En este trabajo se realizaron optimizaciones seriales y la paralelización del cálculo. La optimización serial incluyó: (1) implementación del algoritmo con una nueva estructura de datos: *Halfedge*, (2) optimización en la actualización de costos de arcos, (3) optimización en el uso de la cola de costos y (4) optimización de estructuras de datos. La paralelización fue realizada sobre una de las etapas más demandantes del algoritmo usando la unidad de procesamiento gráfico (GPU). Para validar las optimizaciones y paralelización, se realizaron pruebas de la correctitud y *speed-up* alcanzado en: (1) modelos 3D creados simples, (2) modelos sintéticos de estructuras biológicas y (3) modelos de estructuras biológicas obtenidas de imágenes de microscopía.

Con las optimizaciones y paralelización implementados, se logró una mejora sustancial en el tiempo, pasando de días a minutos o incluso segundos. Además, se verificó que estas mejoras mantienen los *skeletons* resultantes bien definidos, vale decir, mantienen las propiedades que deben cumplir.

Agradecimientos

Quisiera agradecer al Profesor Steffen Härtel, a la Profesora Nancy Hitschfeld y a Mauricio Cerda por el apoyo y entrega durante el desarrollo de este trabajo de título.

Además, quisiera agraceder el financiamiento de este trabajo a la Comisión Nacional de Investigación Científica y Tecnológica CONICYT a través de los proyecto Fondecyt 1120495, 1120579 y 3140447.

Quisiera agradecer a mi familia y amigos por todo el apoyo que he recibido durante mis años de estudio y de vida.

Por último, quisiera agradecer a Jorge Jara, Karina Figueroa, Omar Ramírez, Jorge Toledo y Estibaliz Ampuero por toda la ayuda e información entregada durante todo el proyecto.

Tabla de Contenido

Índice de Figuras

Índice de Tablas

Capítulo 1

Introducción

1.1. Antecedentes Generales

Un *skeleton* es una estructura en una dimensión y es utilizada como una abstracción que captura la geometría y topología de un objeto en 2 y 3 dimensiones. El *skeleton* es muy útil cuando se analiza la estructura de un objeto, y es requerido en diversas aplicaciones, como animación [12], reconocimiento de patrones [14] y análisis de imágenes [13].

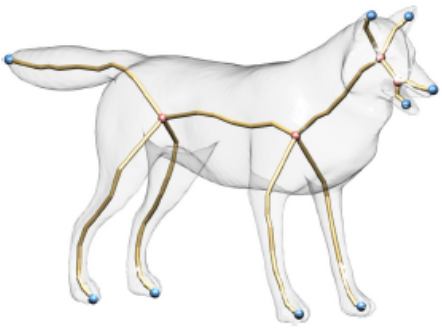


Figura 1.1: Ejemplo de *skeleton*. Estructura original de un lobo y su *skeleton* (en amarillo) [3].

Un *skeleton* debe garantizar propiedades respecto al objeto original, tales como: homotopía, invariabilidad bajo transformaciones isométricas, carácter unidimensional, centralidad, robustez y suavidad [1]. Homotopía se refiere al hecho de que el *skeleton* debe ser topológicamente equivalente al objeto original: debe tener el mismo número de componentes conectados y túneles. Invariabilidad consiste en que dada una transformación isométrica (rotación, traslación y simetría), el *skeleton* de un objeto transformado debe ser igual a la transformación de un *skeleton* del objeto original. Carácter unidimensional expresa que el *skeleton* posee una sola dimensión. En el caso discreto, a lo más

debe tener un voxel[1] de espesor, excepto en las uniones donde convergen dos o más segmentos. Centralidad define que el *skeleton* debe estar centrado dentro del objeto original. Robustez describe la sensibilidad al ruido en el borde del objeto. Finalmente, Suavidad se presenta cuando al moverse a lo largo del *skeleton* se minimiza la variación en la dirección de la curva tangente.

Existen diferentes algoritmos de generación de *skeletons* para objetos en 2 y 3 dimensiones [1]. Entre estos algoritmos podemos encontrar los de adelgazamiento y propagación de la frontera [15], los basado en campos de distancia [16], los de clase geométrica [17] y los de clase de funciones de campo general [18]. Los algoritmos de adelgazamiento producen un *skeleton* por eliminación iterativa de *voxels* del contorno de un objeto hasta obtener la delgadez deseada. A su vez, los algoritmos basados en campos de distancia generan el *skeleton* a partir del conjunto de puntos interiores que poseen la menor distancia con respecto a los límites/bordes del objeto. En los algoritmos de clase geométrica la generación del *skeleton* se aplica sobre objetos representados por mallas poligonales o conjuntos de puntos dispersos. Por último, los algoritmos de funciones de campo general hacen uso de distintas funciones para obtener campos y crear el *skeleton*.

El algoritmo estudiado en este trabajo corresponde a uno de los algoritmos de clase geométrica, ya que utiliza una malla de superficie como entrada, representada por triángulos, arcos y vértices. Este algoritmo elimina arcos de tal manera que produzca el menor efecto en la forma de la estructura, hasta lograr una estructura sin caras.

1.2. Motivación

El laboratorio de análisis científico de imágenes (SCIAN-Lab) [4] es un laboratorio ubicado en la Facultad de Medicina de la Universidad de Chile, que forma parte del Instituto Milenio de Neurociencia Biomédica (BNI)[2].

SCIAN-Lab desarrolla herramientas matemáticas y algoritmos computacionales con el fin de acceder a las características dinámicas, morfológicas y topológicas en sistemas experimentales que abordan los ámbitos biofísicos, biológicos y/o médicos. Gracias al uso de microscopía digital y al procesamiento de imágenes, busca descifrar la interacción entre una estructura y su función en sistemas de modelos de lípidos y fenómenos biológicos a nivel sub-celular, celular y supra-celular.

Estructuras biológicas se representan mediante un conjunto de *píxeles* (2D) o voxeles (3D). La construcción de estructuras 3D se realiza a través de un *stack* de imágenes. Un *stack* es un conjunto

[1] Representación tridimensional de un píxel.
[2] www.bni.cl

de imágenes en el espacio xy del objeto de estudio en el eje z, lo que posteriormente permite crear un modelo con volumen (ver Figura 1.2(a)).

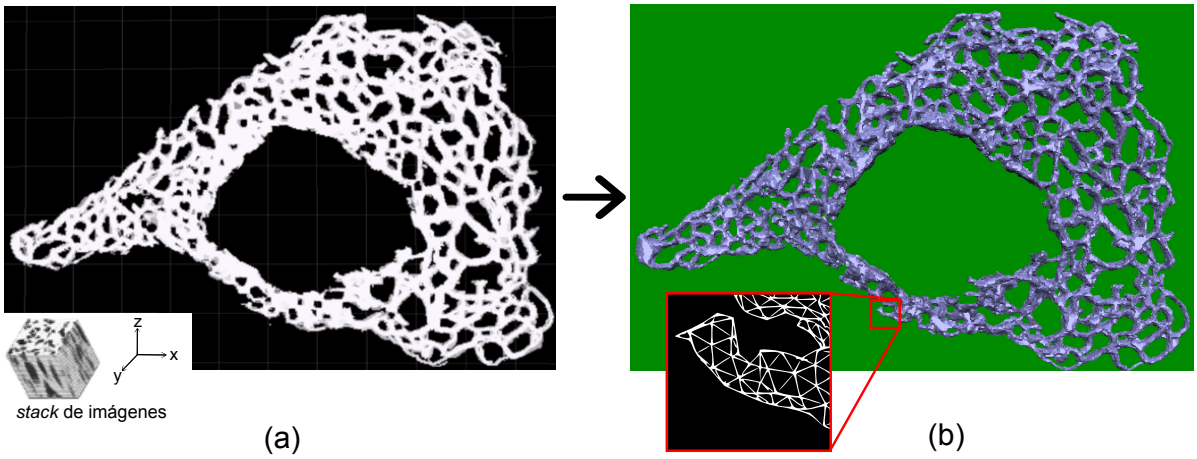


Figura 1.2: Generación de la estructura 3D de un retículo endoplasmático. A la izquierda se muestra una imágen perteneciente al *stack* de imágenes de un retículo (tejido en forma de red). A la derecha está la estructura 3D generada a partir del *stack*, formando una superficie de triángulos.

En los estudios que realiza SCIAN-Lab, existen casos en que los sistemas biológicos son de una complejidad y magnitud significativa (Figura 1.2). Son en estos estudios que se utilizan modelos de *skeleton*.

En la actualidad, el algoritmo generador de *skeleton* implementado usa como entrada una malla de superficie de una estructura (ver Figura 1.2(b)). Este algoritmo fue implementado por Liliana Alcayaga [2] [5].

El algoritmo de generación del skeleton se separa en tres etapas [3] (ver Figura 1.3):

1. **Contracción de Malla**: Proceso en el cual la malla de superficie se contrae hasta llegar a un volumen cercano a cero, manteniendo las propiedades topológicas de la estructura original (túneles y conectividad) (Figura 1.3 a $\rightarrow$ b).

2. **Colapso de la Malla** (o cirugía de conectividad): A partir de una malla contraída se comienza a colapsar arcos, eliminando caras y aristas, hasta obtener un grafo 1D llamado *skeleton* (Figura 1.3 b $\rightarrow$ c).

3. **Corrección del *Skeleton***: El colapso de la malla puede generar *skeletons* que no estén centrados o que partes de ellos estén fuera de la malla. Se realiza un proceso de corrección que implica el desplazamiento de los nodos hacia el centro de la región de la malla que colapsó a ese nodo (Figura 1.3 c $\rightarrow$ d).

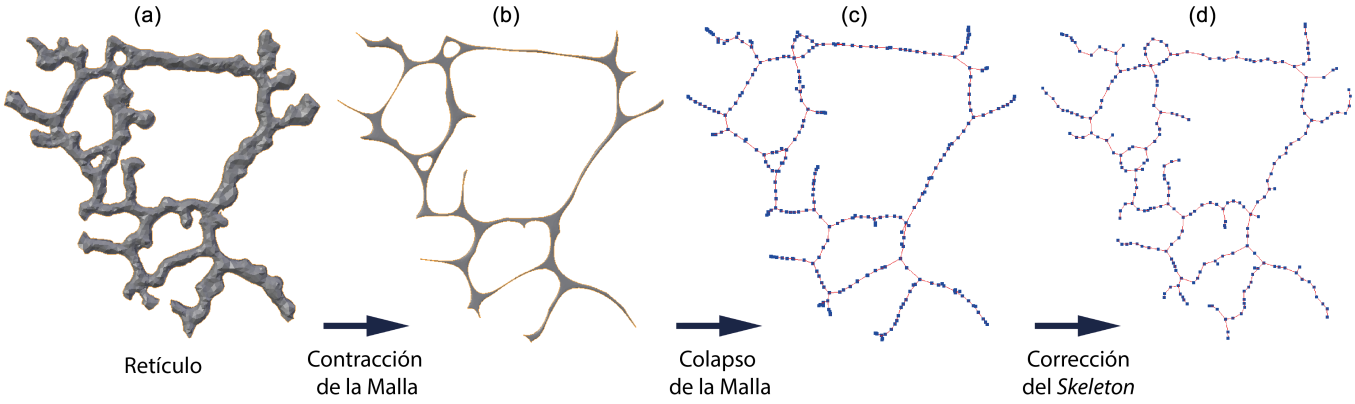


Figura 1.3: Etapas del algoritmo de generación de *skeletons*.

Aunque el algoritmo funciona correctamente, su operación es muy lenta para mallas con muchos nodos (un número mayor a 50 mil), llegando a superar días de procesamiento. En la Tabla 1.1 se puede observar el tiempo de cálculo del algoritmo implementado. Los investigadores no obtienen los resultados requeridos en un tiempo razonable. Por lo tanto, se necesita optimizar el tiempo de cálculo del algoritmo, con el fin de minimizar el tiempo de ejecución, y así poder realizar estudios científicos cuantitativos.

Malla	**N° de vértices**	**Tiempo de cálculo (hh:mm:ss)**			
		Contracción de la Malla	Colapso de la Malla	Corrección del Skeleton	Tiempo Total
neurona	13.508	00:00:01	00:08:27	00:00:00	00:08:28
cresta neural	84.513	00:06:16	17:26:48	00:00:00	17:33:04
retículo	92.452	00:00:06	102:09:00	00:00:02	102:09:08

Tabla 1.1: Tiempo de cálculo de la implementación [2].

1.3. Objetivos

1.3.1. Objetivo General

Optimizar y paralelizar la implementación del algoritmo de generación de *skeletons* basados en mallas de superficies [2], para mejorar su desempeño.

1.3.2. Objetivos Específicos

1. Evaluar la versión actual de la implementación del algoritmo [2], estudiando su complejidad y tiempo de cálculo para distintas entradas.
2. Proponer e implementar optimizaciones seriales que mejoren el tiempo de cálculo de las secciones más demandantes en tiempo.
3. Evaluar distintas arquitecturas de paralelización que se adapten a las secciones más demandantes del algoritmo.
4. Proponer e implementar una versión paralela de las secciones que toman más tiempo de cálculo y validar la solución propuesta, comparándola con la versión serial.

1.4. Contenido de la Memoria

En el Capítulo 1 de Introducción se presentan los antecedentes generales del trabajo, la motivación de su desarrollo y sus objetivos. En el siguiente capítulo de Antecedentes, se analiza la implementación [2], estudiando su rendimiento y secciones más demandantes. Terminado el análisis inicial, se prosigue con la optimización de la versión serial en el Capítulo 3, implementando cuatro mejoras. En el Capítulo 4 se paraleliza una de las etapas más demandantes del algoritmo utilizando la Unidad de Procesamiento Gráfico (GPU). Por último, se concluye y se plantean propuestas para trabajo futuro en el Capítulo 5.

Capítulo 2

Antecedentes

En este capítulo se analiza la implementación a mejorar [2], estudiando su rendimiento y secciones más demandantes para así luego iniciar la optimización serial.

2.1. Descripción de las Etapas del Algoritmo

Para comenzar, se hizo un análisis tanto del algoritmo como de la implementación 2011 [2] en cada una de sus etapas: Contracción de la Malla, Colapso de la Malla y Corrección del *Skeleton* [3]. Gracias a este análisis se pudo entender de mejor manera la complejidad computacional de las tres etapas de la implementación [2].

2.1.1. Etapa Contracción de la Malla

En la contracción, por cada uno de los n vértices de la malla inicial, se calcula su nueva posición mediante la obtención del promedio de las posiciones de los n_1 vértices vecinos. Experimentalmente se estimó que la cantidad de vecinos n_1 es aproximadamente seis. El cálculo de las nuevas coordenadas se repite para todos los vértices hasta que la contracción no produzca una diferencia considerable (medido mediante un umbral). Como la cantidad de repeticiones es $\ll n$, el orden de esta etapa es $O(n)$.

$$v_i^{k+1} = v_i^k \cdot 0.5 + \left(\sum_{vecinos} \frac{1}{n_i} v_i^k\right) \cdot 0.5.$$

En la ecuación anterior, se muestra el cálculo de la nueva coordenada de un vértice. Ese cálculo depende tanto de su coordenada actual como de las coordenadas de sus vécinos, mediante el promedio de sus coordenadas.

2.1.2. Etapa Colapso de la Malla

En esta etapa, antes de cada colapso se calculan los costos de los arcos y se construye la cola de costos de los arcos. El cálculo del costo de un arco depende del arco mismo y sus vecinos, siendo su costo n_1. Ya calculado los costos de los m arcos, estos arcos se insertan en la cola de costos de los arcos, siendo $log(m)$ la inserción de cada arco en la cola. Por lo tanto, el costo total de construcción de la cola es $mlog(m)$. Luego de la construcción de la cola, se colapsa el arco de menor costo, teniendo que procesar los vértices y arcos vecinos (n_1). Terminado el colapso del arco, se procede a corregir fallas. En esta corrección se buscan elementos repetidos dentro de una región (anillo de distancia 3) centrada en el vértice que queda luego del colapso, comparando todos los elementos contra si mismos. Siendo n_2 la cantidad de elementos en ese anillo (aproximadamente 36 vértices y arcos cada uno), la corrección tiene un costo n_2^2. Todo ese proceso de colapso de un arco se repite para todos los colapsos, hasta un máximo de m. Teniendo en cuenta que antes de cada colapso se construye nuevamente la cola, la etapa de Colapso de la Malla tiene un costo de $m \cdot (mlog(m) + n_2^2)$. Como $n_2^2 \ll m$ el orden de la etapa es $O(m^2log(m))$.

2.1.3. Etapa Corrección del *Skeleton*

En esta etapa se centran cada uno de los n_3 nodos del *skeleton* generados en la etapa Colapso de la Malla, por lo que su orden es $O(n_3)$, con $n_3 << m$ y n.

Consideradas las 3 etapas anteriores, el costo total del algoritmo es $O(m^2log(m))$, dominado por la etapa de Colapso de la Malla.

2.2. Identificación del Proceso más Demandante

La búsqueda e identificación de la etapa más demandante se centró en el tiempo de cálculo. Para iniciar con el análisis, se realizaron pruebas con una malla simple (que se identifica con el nombre de *Toroide*, ver Figura 2.1). A esta malla se le fue aumentando la cantidad de puntos para probar con mallas de distintos tamaños.

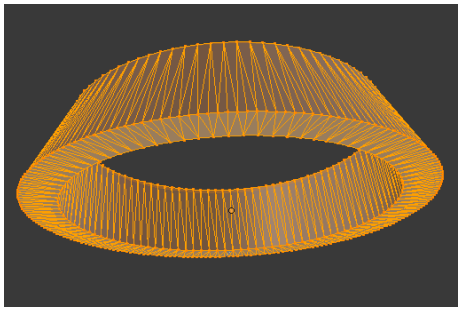


Figura 2.1: Vista lateral de la malla *Toroide*.

N° de vértices	**Tiempo de cálculo (hh:mm:ss)**			
	Contracción de la Malla	Colapso de la Malla	Refinamiento del *skeleton*	Tiempo total
400	00:00:00	00:00:00	00:00:00	00:00:00
8.000	00:00:00	00:00:49	00:00:00	00:00:49
20.000	00:00:00	00:06:12	00:00:00	00:06:12
28.000	00:00:00	00:12:43	00:00:00	00:12:43
40.000	00:00:00	00:26:30	00:00:00	00:26:30
80.000	00:00:00	01:41:51	00:00:00	00:41:51

Tabla 2.1: Tiempo de procesamiento de las etapas del algoritmo para la malla *Toroide*.

Con los resultados mostrados en la Tabla 2.1 se puede observar que la etapa más demandante en tiempo es el Colapso de la Malla. Para corroborar estos resultado, se realizaron además las mismas mediciones para mallas simplificadas de neuronas de pez cebra (Figura 2.2), pues son ejemplos más realistas y cercanos a lo utilizado en el laboratorio.

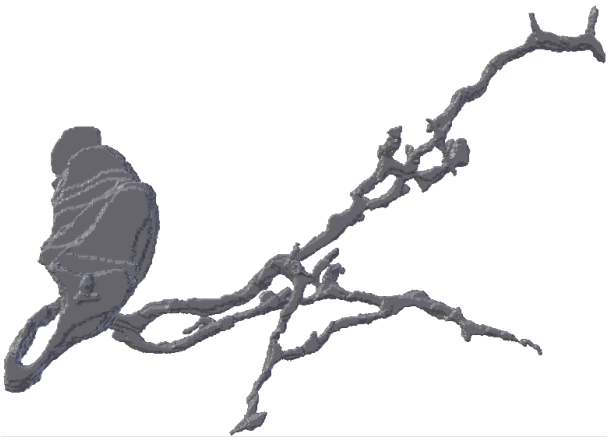


Figura 2.2: Ejemplo de malla de neurona de pez cebra.

Malla	**N° de vértices**	**Tiempo de cálculo (hh:mm:ss)**			
		Contracción de la Malla	Colapso de la Malla	Refinamiento del *Skeleton*	Tiempo Total
neurona1	39.276	00:00:14	02:27:18	00:00:00	02:27:32
neurona2	40.338	00:00:16	02:34:13	00:00:00	02:34:29

Tabla 2.2: Tiempo de procesamiento de las etapas del algoritmo para las mallas de neuronas.

Observando los resultados de la Tabla 2.2, nuevamente la etapa Colapso de la Malla mostró un tiempo de cálculo considerablemente mayor que los otras etapas del proceso. Por lo tanto, la optimización del algoritmo en una primera fase se centró en la etapa Colapso de la Malla, tal como se describe en el capítulo 3.

2.3. Análisis de la Etapa más Demandante

Partiendo de la elección de la etapa Colapso de la Malla, ésta se puede descomponer en distintos pasos, siendo los más importantes: (1) el cálculo de los costos de los arcos, (2) el colapso del arco elegido y (3) la revisión y reparación de la malla luego de cada colapso. Estos pasos se encuentras resumidos en la Figura 2.3.

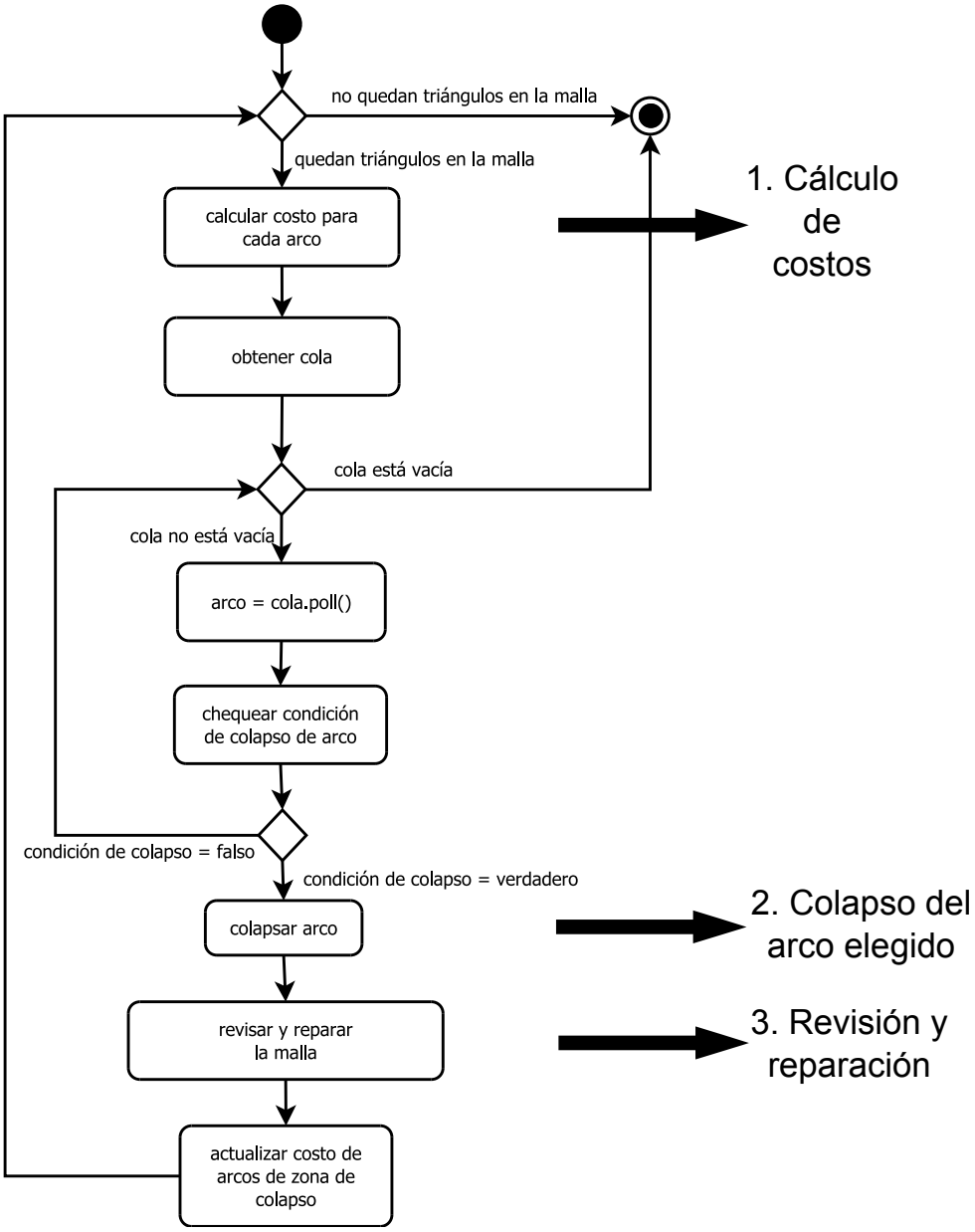


Figura 2.3: Diagrama de actividades de la implementación [2] de la etapa Colpaso de la Malla.

De los pasos principales, se inició el análisis con el cálculo de los costos, y la revisión y reparación. Para cada uno de esos pasos se midió su tiempo de cálculo, incluyendo su porcentaje con respecto al tiempo total de la etapa Colapso de la Malla. Estas mediciones se realizaron tanto para la malla simple (*Toroide*) como para las de neuronas (Figuras 2.1 y 2.2).

N° de vértices	N° de aristas	N° de triángulos	Tiempo de cálculo (hh:mm:ss)		
			Cálculo de los Costos	Revisión y Reparación	Colapso de la Malla
400	1.200	800	00:00:00 (38 %)	00:00:00 (58 %)	00:00:00 (100 %)
8.000	24.000	16.000	00:00:49 (80 %)	00:00:07 (14 %)	00:00:49 (100 %)
20.000	60.000	40.000	00:05:54 (95 %)	00:00:09 (2 %)	00:06:12 (100 %)
28.000	84.000	56.000	00:11:44 (92 %)	00:00:45 (5 %)	00:12:43 (100 %)
40.000	120.000	80.000	00:25:21 (95 %)	00:00:44 (2 %)	00:26:30 (100 %)
80.000	240.000	160.000	01:39:44 (97 %)	00:01:00 (0 %)	01:41:51 (100 %)

Tabla 2.3: Tiempo de cálculo de las partes principales del Colapso de la Malla para la malla *Toroide*.

Los resultados de la Tabla 2.3 muestran que el paso más demandante en tiempo de cálculo es el de cálculo de los costos de los arcos.

Malla	N° de vértices	N° de aristas	N° de triángulos	Tiempo de cálculo (hh:mm:ss)		
				Cálculo de los Costos	Revisión y Reparación	Colapso de la Malla
neurona1	39.276	117.840	78.560	00:27:29 (18,7 %)	01:59:26 (81,1 %)	02:27:18 (100 %)
neurona2	40.338	121.026	80.684	00:30:54 (20,0 %)	02:02:53 (79,7 %)	02:34:13 (100 %)

Tabla 2.4: Tiempo de cálculo de las partes principales del Colapso de la Malla para las mallas de neuronas.

Con las mediciones de la Tabla 2.4 se muestra que en mallas tipo neuronas el paso más demandante en el Colapso de la Malla, es la revisión y reparación de la malla. En la medición de la Tabla 2.3, la geometría de las mallas es más simple, mientras que en la segunda la geometría de las mallas es más irregular y con formas más complejas. Esto lleva a pensar que mientras más compleja sea la geometría de la malla, más fallas se introducen durante el colapso que deben ser reparadas (triángulos y arcos repetidos, y la falta de referencias a vértices, arcos y triángulos vecinos).

Con estas mediciones, se concluye que son los pasos Cálculo de Costos y Revisión y Reparación, los más demandantes en tiempo de cálculo de la etapa Colapso de la Malla. Así, se

decide analizar cada uno de estos con más profundidad en las secciones 2.3.1 y 2.3.2.

2.3.1. Cálculo del Costo de los Arcos

La etapa Colapso de la Malla tiene como objetivo principal lograr una correspondencia entre el *skeleton* resultante y la malla original. Su procesamiento consiste en ir eliminando arcos y caras, para ir simplificando la malla.

La elección de los arcos a eliminar está determinada por un costo asociado al arco, siendo elegido el que tenga el menor costo. Este costo es calculado usando tanto las coordenadas de los vértices del arco como las de los vértices vecinos (a distancia de un arco), y determina "cuán costoso" es colapsar un arco. La idea es eliminar el arco que implique el menor movimiento medido en distancia.

El costo de un arco se cálcula sumando dos costos: *Shapecost* y *Samplingcost*. Siendo el colapso desde el vértice i hacia el vértice j ($i \rightarrow j$), el *Shapecost* es la suma de las distancias cuadráticas desde el vértice v_j hacia los arcos adyacentes a los vértices i y j. Por otra parte, el *Samplingcost* es la distancia total que los arcos adyacentes del vértice i viajan durante el colapso ($i \rightarrow j$).

Teniendo los costos *Shapecost* y *Samplingcost*, el costo total de un arco es:

$$Totalcost(i,j) = w_a \cdot Shapecost(i,j) + w_b \cdot Samplingcost(i,j),$$

donde w_a y w_b son constantes con valor 1,0 y 0,1, respectivamente.

El colapso de un arco implica eliminar el arco y mover los arcos vecinos, modificando la malla, como se ilustra en la Figura 2.4. El costo de un arco depende tanto de su posición como de los arcos adyacentes a éste. Por lo tanto, si no hay cambios en el arco ni en sus arcos adyacentes, su costo no cambia.

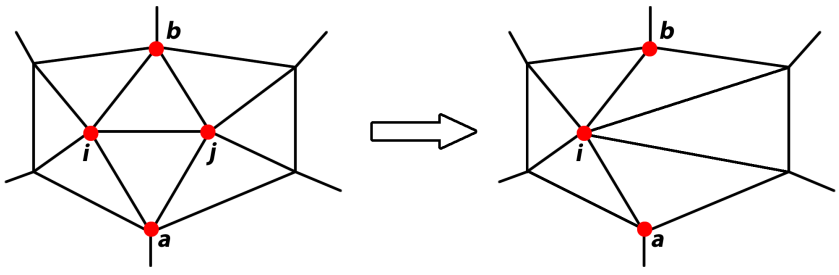


Figura 2.4: Colapso simple ($j \rightarrow i$): el arco (i,j) será el arco eliminado. Los arcos adyacentes a éste comparten en total 4 vértices (los marcados en rojo en la imagen).

El costo de cada arco se calcula al inicio del proceso, y cada vez que se elimina un arco se debe recalcular el costo de los arcos afectados. Sin embargo, en la implementación [2] se detectó que los cálculos de los costos se estaban recalculando para todos los arcos durante todo el proceso. Como se explica en la Sección 3.2, esta repetición del cálculo es innecesaria dada la invarianza de gran parte de la malla. En efecto, sólo se afecta/modifica el costo de los arcos dentro de una región limitada donde se eliminó el arco.

2.3.2. Revisión y Reparación de la Malla

En el análisis de los casos a revisar se busca la presencia de múltiples tipos de fallas que se producen durante una eliminación de un arco, entre ellas: triángulos o arcos repetidos, la falta de algún triángulo vecino a un arco y que su referencia esté incorrecta, y casos en que un arco apunta a un triángulo derecho y otro izquierdo pero que son el mismo. La idea fue ir eliminando la necesidad de buscar esas fallas e ir previniendo su ocurrencia. A continuación se detalla una de las revisiones, la corrección de los triángulos repetidos.

Triángulos Repetidos

Cuando se colapsa/elimina un arco pueden aparecer casos a reparar, buscando mantener la correctitud de la malla, como se buscó en la implementación [2]. Una manera de predecir donde podrían surgir estas fallas, es medir la cantidad de vértices que comparten arcos que contienen a los vértices del arco a colapsar. En particular, en el caso simple de colapso esos arcos comparten sólo 4 vértices (ver Figura 2.4). De esta manera, cuando se comparten más de 4 vértices (ver Figura 2.5), es probable generar un triángulo repetido luego del colapso.

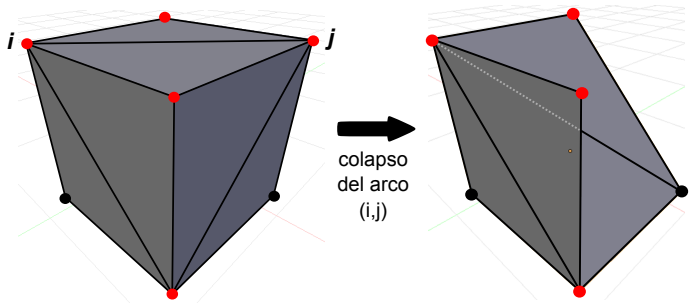


Figura 2.5: Colapso con condición de más de 4 vértices compartidos presente: el arco (i,j) es el arco a eliminar. Los arcos adyacentes a éste comparten 5 vértices (marcados en rojo).

Con esta condición, se planteó la siguiente idea: cuando la cantidad de vértices compartidos

por los arcos superase el umbral de aceptación (4 vértices), se realizaría el proceso de revisión y reparación. En caso contrario, no se haría ese paso, asumiendo que no arrastraría ninguna falla.

Una primera aproximación fue entender si era posible encontrar condiciones geométricas para evitar las verificaciones. Se identificó que el mayor problema es que las condiciones geométricas, como la idea de los 4 vértices propuesta, son válidas en mallas conformes [1], y la etapa Colapso de la Malla va generando una malla no conforme que resulta en un *skeleton*. Se verificó este punto, implementando la condición de los 4 vértices, sin lograr una convergencia de la implementación 2011 [2]. Con esto, se descartó la idea de buscar condiciones geométricas que evitara realizar verificaciones.

2.4. Tecnologías a utilizar en la Paralelización

Con el objetivo de paralelizar la implementación del algoritmo, se han revisado dos opciones: *Graphics processing unit* (GPU) y *Multi-threading*.

2.4.1. Cálculo en GPU

La GPU es una unidad de procesamiento gráfico. La GPU permite acelerar operaciones de cálculo. Gracias a sus múltiples núcleos, posibilita paralelizar tareas y así cumplir más rápido las operaciones.

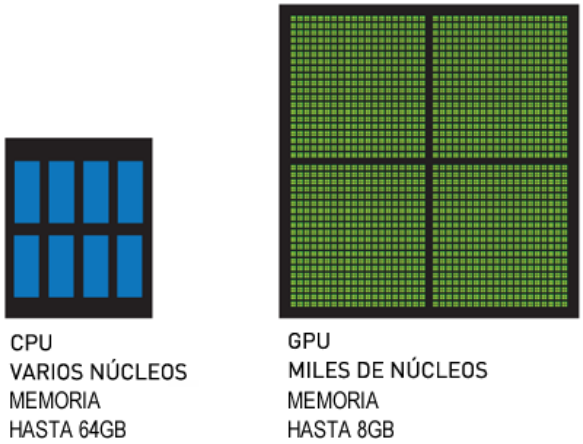


Figura 2.6: Comparación entre CPU y GPU. Mientras que la CPU tiene menos de una decena de núcleos, una GPU consta de miles de ellos [8].

[1] Cada arco tiene asociado sólo dos triángulos, dos triángulos no pueden tener tres vértices en común y los triángulos no pueden intersectar salvo en sus arcos.

Por lo general, la GPU es usada para resolver problemas que se pueden expresar como tareas idénticas para cada núcleo, con baja comunicación entre ellos, por ejemplo algunas operaciones matriciales. Algunos de los problemas en que se usa son: transformaciones sobre imágenes, ordenamiento de elementos, operaciones vectoriales y operaciones matriciales.

La desventaja principal del uso de GPU es la limitación en el uso de estructuras de datos avanzadas. De esta manera, no permite trabajar con objetos y aprovechar los beneficios de la abstracción de los datos en un objeto con variables y métodos, es difícil usar estructuras de datos más elaboradas, y tener comunicación entre núcleos.

Programación en GPU

Existen 2 alternativas de programación principales en GPU: *Compute Unified Device Architecture* (CUDA) [6] [7] y *Open Computing Language* (OpenCL) [9] [10]. CUDA es una plataforma de computación paralela y un modelo de programación que permite incrementar el rendimiento computacional mediante el aprovechamiento de la GPU. La ventaja de CUDA es que cuenta con múltiples ejemplos ya hechos. Aunque posee como desventaja el requisito de trabajar sólo en tarjetas de la marca NVIDIA, su funcionamiento está optimizado para trabajar en esas tarjetas. Por otra parte, OpenCL es también una plataforma de computación paralela, pero libre y abierta. Su principal ventaja es que es multiplataforma, pudiendo trabajar sobre cualquier tarjeta gráfica compatible (al contrario de la limitación de CUDA). Sin embargo, OpenCL posee dos desventajas principales: (1) al ser multiplataforma, es menos eficiente que las plataformas que están optimizadas para dispositivos específicos, y (2) posee un desarrollo más reciente que CUDA, por lo que existen menos ejemplos documentados y librerías que trabajen en conjunto con OpenCL.

2.4.2. Multi-threading

Multi-threading es un modelo de programación y ejecución que permite múltiples hilos de ejecución (llamados *threads*) dentro del contexto de un mismo proceso.

Multi-threading está capacitado para resolver los mismos problemas que se resuelven en GPU, pero está limitado por la cantidad de núcleos que existen en la CPU (aproximadamente 8 en comparación a los miles de la GPU). La ventaja de su uso es el manejo de memoria principal compartido, mientras que la desventaja es que se deben utilizar estructuras que soporten accesos concurrentes.

Capítulo 3

Optimización Serial

Teniendo ya analizada la implementación [2], se procede con los distintos cambios para optimizar la versión serial.

Desde este capítulo en adelante, los cambios y mediciones se realizarán bajo las mismas condiciones, con un *Hardware* y *Software* específicos. El *Hardware* utilizado corresponde a un computador con un procesador Intel(R) Core(TM) i7 CPU 3,20GHz x2, memoria RAM de 24GB, sistema operativo Windows 8 de 64 bits, y tarjeta gráfica NVIDIA GeForce GTX 560 con 1024MB de memoria. El *Software* que fue usado en el trabajo consiste en los programas: Blender (herramienta de visualización y animación 3D), OpenFlipper (herramienta para procesar, modelar y visualizar datos geométricos), NetBeans (entorno de desarrollo para implementar, compilar, depurar y ejecutar programas) y Dia (Programa para la creación de diagramas).

La implementación del proyecto fue realizada usando el lenguaje de programación JAVA. Se eligió para mantener el mismo lenguaje de programación de la implementación [2], como también para mantener el uso del mismo lenguaje en las interacciones que tiene el proyecto con otros programas.

Dentro de este capítulo, las mediciones están hechas en base al tiempo de cálculo, cantidad de nodos y segmentos del *skeleton* resultante, y se realizaron en la mallas de la Tabla 3.1, e ilustradas en el Apéndice A.1. Los cambios presentes en la optimización serial incluyen: (3.1) la implementación del algoritmo usando una estructura de representación de mallas distinto (*Halfedge*), (3.2) una optimización en la actualización de los costos, (3.3) una optimización en la construcción de la cola de costos de los arcos, y (3.4) la optimización de estructuras de datos.

Los cambios son acumulativos, por lo que las mediciones son resultado de la mejora respectiva y todas las previas.

Malla	Vértices	Arcos	Triángulos
Neuronas			
cherry1	46.711	140.163	93.442
cherry2	40.385	121.167	80.778
cherry3	26.716	80.154	53.436
neuroest1	52.929	158.931	105.954
neuroest2	79.129	237.657	158.438
neuroest3	25.205	75.687	50.458
Crestas Neuronales			
crestaneuronal	81.966	246.186	164.124
Retículos			
retículo1	65.516	199.104	132.736
retículo2	5.720	17.190	11.460
retículot00	45.182	137.070	91.380
retículot01	44.898	136.248	90.832
retículot02	45.191	137.163	91.442
retículot03	45.162	137.100	91.400
Mallas Sintéticas			
arbol1	2.367	7.125	4.750
arbol2	2.803	8.433	5.622
arbol3	2.626	7.914	5.276
red1	6.588	19.824	13.216
red2	6.630	19.956	13.304
red3	6.939	20.877	13.918

Tabla 3.1: Lista de mallas sobre las que se hacen las mediciones (ver imágenes en A.1).

Los árboles sintéticos son generados iniciando desde un punto raíz. A partir de ese punto se van generando ramas y bifurcaciones con ángulos de crecimiento que siguen una distribución uniforme entre un ángulo máximo y mínimo. Por otra parte, las redes sintéticas se crean en base a un distribución uniformemente aleatoria de puntos. Con esos puntos se construye la malla usando el algoritmo de triangulación de *Delaunay* [5].

Las mallas cherry1, cherry2, cherry3, retículo1, retículo2 y retículotXX fueron gentileza de SCIAN-Lab (cherry $\rightarrow$ Karina Figueroa, retículo1 y retículo2 $\rightarrow$ Omar Ramírez, retículotXX $\rightarrow$ Jorge

Toledo) y neuroestX de Estibaliz Ampuero (Universidad Andres Bello).

3.1. Nueva estructura: Halfedges (Mejora 1)

La implementación [2] funciona en base a una estructura basada en triángulos. Los elementos presentes son vértices, arcos y triángulos. Los arcos tienen referencias a sus vértices y a los triángulos derecho e izquierdo. Los triángulos tienen las referencias a sus vértices y arcos. Además, cada vértice guarda la lista de triángulos, arcos y vértices que lo rodean.

Dada la cantidad de referencias que existen, al momento de colapsar un arco la actualización de las referencias se vuelve una tarea difícil y meticulosa. Además, en la implementación [2] existe un post-procesamiento de revisión llamado “Revisión y Reparación”, en el cual se reparan casos de error que ocurren luego de un colapso (ver Sección 2.3.2). Después de cada colapso el proceso de Revisión y Reparación realiza varios recorridos por la vecindad del sector del colapso para buscar esos errores.

Con el fin de reducir las referencias y prevenir el proceso de Revisión y Reparación, se decidió probar con dos nuevas ideas: una estructura llamada Halfedge y la aceptación de caras solapadas (o repetidas). La idea del cambio de estructura fue probar si esos errores se reducen y/o están relacionados con el tipo de estructura que se ocupe para representar la malla.

Estructura *Halfedge* [19]: La estructura de datos *Halfedge* es una estructura centrada en los arcos, capaz de mantener la información de incidencia de vértices, arcos y caras (ver Figura 3.1). Cada arco está formado por dos *halfedges* con direcciones opuestas. Cada uno de esos *halfedge* guarda su cara y vértice incidente. Así mismo, cada vértice y cara guarda su *halfedge* incidente. Una cara puede ser representada por un conjunto de *halfedges* que forman un ciclo, en donde cada uno de ellos tiene la referencia a su *halfdege* anterior(*prev*) y posterior(*next*). Este ciclo representa la presencia de una cara en la malla.

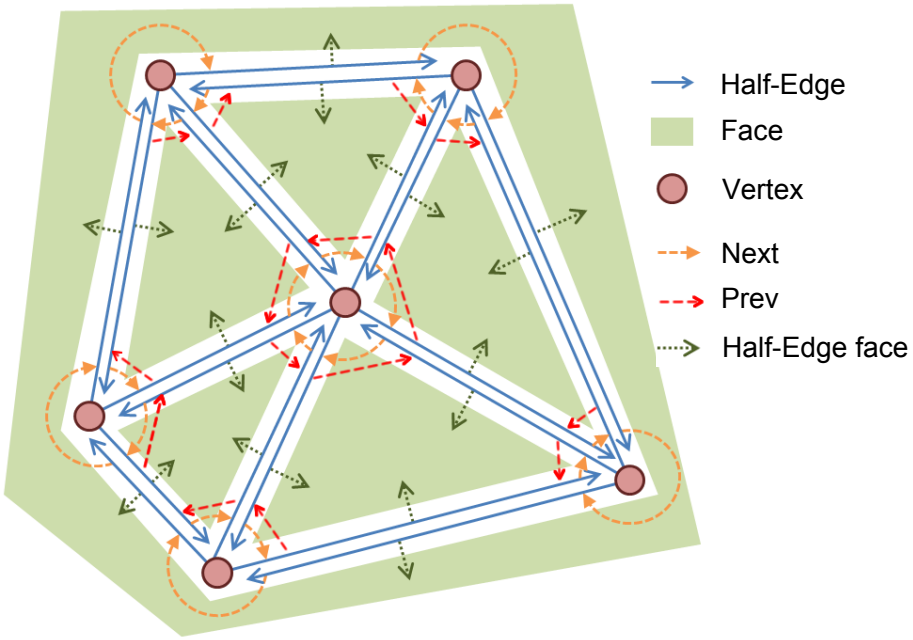


Figura 3.1: Estructura *Halfedge* con sus distintos elementos y referencias [28].

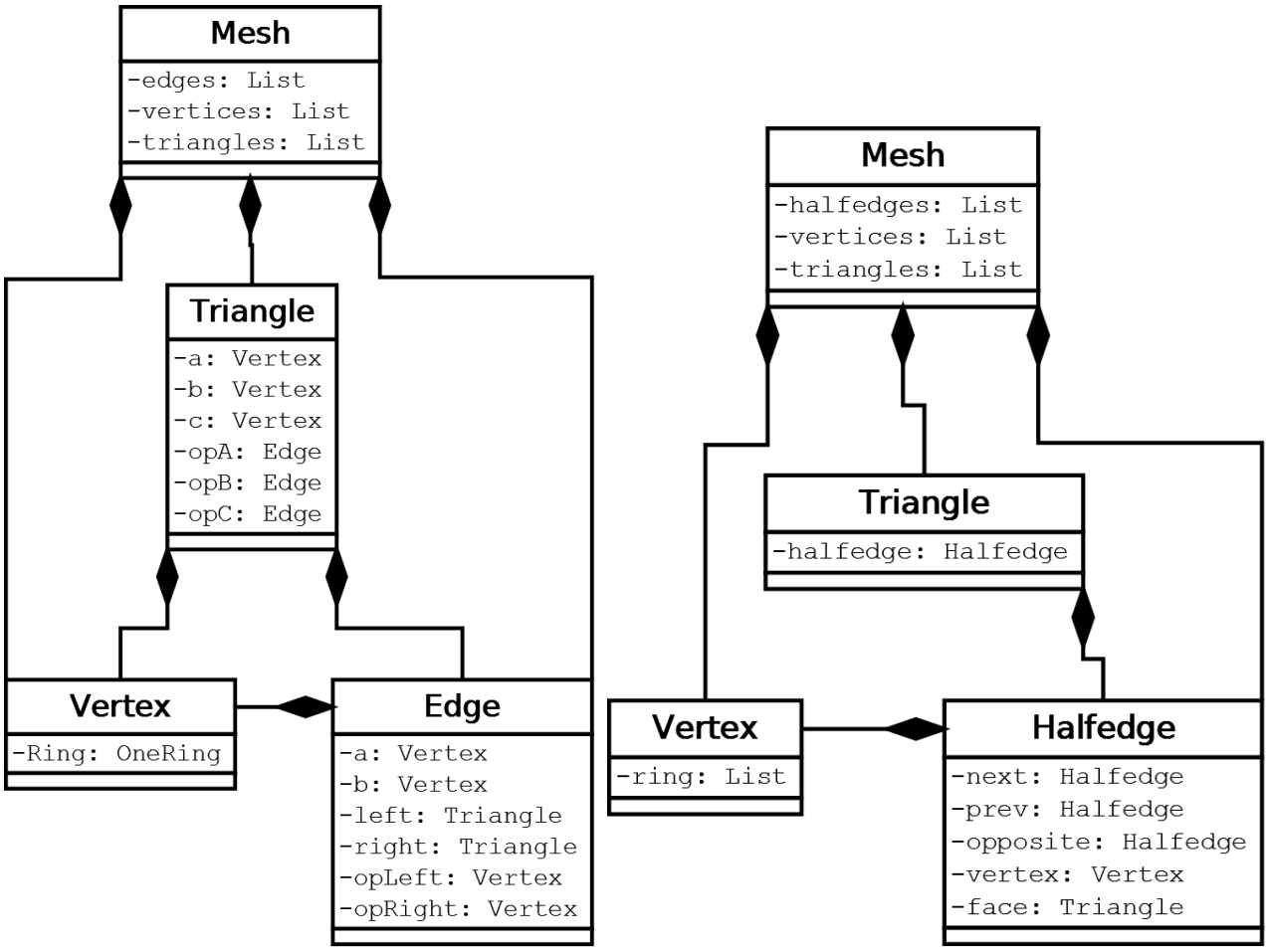
Para poder utilizar la estructura *halfedge* en el algoritmo de *skeleton*, a cada vértice se le agregó un arreglo que contiene a todos los *halfedges* que apuntan hacia él. De esa forma se puede obtener información de los vecinos de forma más fácil.

3.1.1. Estructuras y Referencias

A continuación se hace una comparación entre las estructuras y sus referencias principales de la implementación [2] y la implementación *Halfedge* para representar una malla.

Implementación Alcayaga [2] (Figura 3.2 (a))

- *Mesh*: estructura formada por la lista de vértices, arcos y triángulos.
- *Vertex*: posee una referencia a una estructura llamada *OneRing*. Ésta corresponde a la vecindad del vértice y contiene todo lo que esté a distancia de un arco del vértice. *OneRing* posee la lista de los arcos y triángulos de esa región.
- *Edge*: mantiene la referencia de sus vértices, triángulos adyacentes y sus vértices opuestos.
- *Triangle*: referencia a sus vértices y arcos.



(a) Implementación [2] **(b)** Implementación *Halfedge*

Figura 3.2: Diagramas de las clases. (a) Implementacion [2] , y (b) La nueva estructura de datos *Halfedge*.

Implementación *Halfedge* (Figura 3.2 (b))

- *Mesh*: estructura formada por la lista de vértices, arcos y triángulos.
- *Vertex*: posee una referencia a una lista de los *halfedge* que llegan al vértice.
- *Halfedge*: mantiene la referencia del vértice al que llega, de su triángulo adyacente, el *halfedge* anterior, posterior y el opuesto.
- *Triangle*: referencia a uno de sus *halfedge* adyacentes.

Variables de instancias

Implementación [2]	Implementación *Halfedge*
Mesh	
List edges	List halfedges
List vertices	List vertices
List triangles	List triangles
Vertex	
Ring:	Ring:
List edges	List halfedges
List triangles	
Edge	Halfedge
Vertex a	Halfedge next
Vertex b	Halfedge prev
Triangle left	Halfedge opposite
Triangle rigth	Vertex vertex
Vertex opLeft	Triangle face
Vertex opRight	
Triangle	
Vertex a	Halfedge halfedge
Vertex b	
Vertex c	
Edge opA	
Edge opB	
Edge opC	

Tabla 3.2: Variables de instancias de los elementos de la malla.

La estructura de la implementación [2] tiene una cantidad similar de referencias y variables de instancia que las de la implementación *Halfedge* (ver Figura 3.2 y Tabla 3.2). La diferencia principal se encuentra en los triángulos. En *Halfedge*, el triángulo sólo tiene una referencia, y eso es suficiente para conocer todos sus elementos. En cambio, en implementación [2] el triángulo tiene varias referencias y variables. Estas múltiples referencias resultan de utilidad para acceder de forma inmediata a un elemento del triángulo, pero mientras más referencias se tengan más casos de revisión se deben considerar al momento de eliminar o actualizar estructuras.

3.1.2. Triángulos Solapados

Junto al cambio de estructura de malla, se agregó el hecho de aceptar triángulos solapados (ver Figura 3.3). Mientras que en la implementación [2] se busca repararlos inmediatamente, ahora se buscó aceptarlos y tratarlos sólo cuando se identifique su presencia. Es decir, cuando se detecte que triángulos solapados son afectados por el colapso de un arco.

En la implementación [2], se tiene una restricción para los colapsos:

Condición de colapso: sólo se puede colapsar un arco si es que tiene sus dos triángulos adyacentes existentes (esa condición se anula al final para colapsar triángulos que aún existen).

Los tipos de arcos que no cumplían con la condición de colapso eran los que ya no tenían triángulos o los que pertenecían a un solo triángulo. Estos últimos arcos aparecen cuando se reparan caras solapadas, que luego de la reparación produce un triángulo con algunos lados que pueden ser bordes de una superficie abierta.

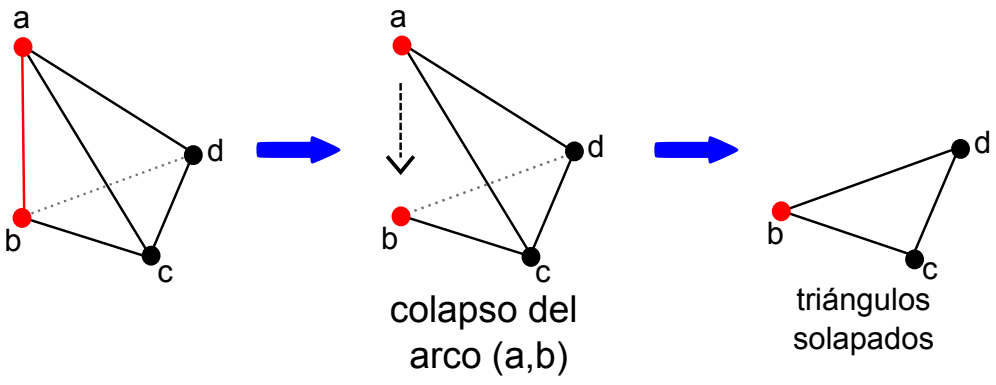


Figura 3.3: Ejemplo de generación de triángulos solapados. Al colapsar el arco (a,b), los triángulos (a,c,d) y (b,c,d) quedan en la misma posición.

En la Figura 3.3, luego del colapso del arco (a,b) y bajo la condición de colapso, todos los arcos del triángulo resultante son no colapsables. En la nueva implementación, para mantener dicha regla, se revisa si el arco a colapsar pertenece o no a un triángulo solapado. Si es así, no se colapsa. Para su detección, se revisa de la siguiente forma:

Detección de borde de triángulo solapado: Para saber si el halfedge a colapsar pertenece al borde de un triángulo solapado, se revisan las coordenadas de los vértices de dos caras: la cara del halfedge y la del halfedge opuesto. Si las coordenadas de sus tres vértices coinciden, entonces se detectó el caso.

Dada la aceptación de los triángulos solapados aparecen casos especiales a tratar al momento de colapsar. Los distintos casos son:

1. Caso normal

El caso normal ocurre cuando el arco es adyacente a dos triángulos no solapados, y sólo dos triángulos son adyacentes a ese arco (Figura 3.4).

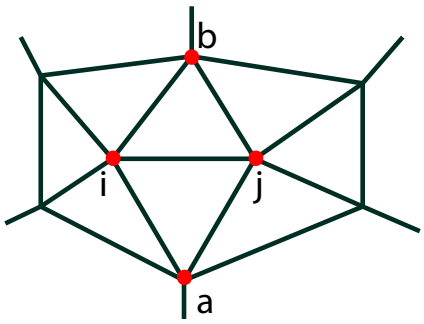


Figura 3.4: Caso normal de colapso. El arco a colpasar es (i,j), el cual posee los triángulos vecinos (i,j,b) y (i,a,j).

2. Caso borde de triángulos solapados

El caso borde de triángulos solapados corresponde al hecho de que el arco elegido a colapsar sea un arco en común entre dos triángulos solapados. Si los *halfedge next* y *prev* también son parte del arco en común para ambos triángulos, quiere decir que el colapso producirá un solo arco sin triángulos adyacentes.

3. Caso múltiples triángulos incidentes al arco

El caso múltiples triángulos incidentes al arco se describe como la existencia de tres o más triángulos adyacentes al arco que se quiere colapsar. Este caso se maneja por cada triángulo adyacente (Figura 3.5).

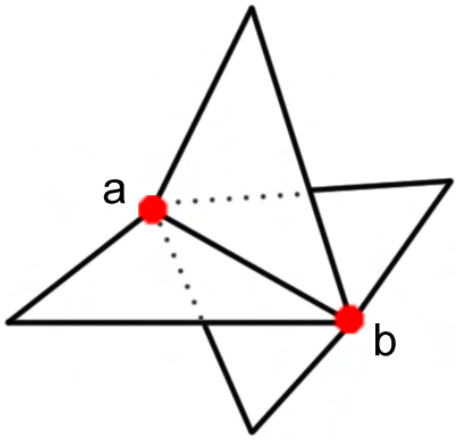


Figura 3.5: Ilustración del caso de múltiples triángulos incidentes al mismo arco.

3.1.3. Resultados

Al usar *Halfedge* y aceptar triángulos solapados, se observan mejoras en el tiempo de cálculo de la etapa Colapso de la Malla de entre 2x-6x (ver Tabla 3.3). El promedio del porcentaje de tiempo reducido fue de 69,5 %$\pm$20,2[1] y el *speed-up* promedio es 4,1$\pm$1,5[2]. Las desviaciones estándar son altas debido a que las mallas son homogéneas en los distintos grupos, teniendo tamaños variables.

Las Figuras 3.6 y 3.7 muestra que en general todos los grupos de mallas tuvieron una mejora en el tiempo sobre el 50 %, excepto el grupo de árboles. En ese grupo es donde se vieron menores cambios, correspondiendo a las mallas más pequeñas (los árboles de las mallas sintéticas). Uno de los resultados de los árboles produce un porcentaje de reducción negativo (árbol0). En el caso de la malla árbol0 el tiempo de cálculo es del orden de centésimas de segundos, situación que ocurre cuando las mallas poseen menos de 10 mil vértices y arcos, como es el caso de los árboles. En estos tiempos de ejecución cortos puede influir el estado del SO, y otros procesos en ejecución, más que la cantidad de elementos que tenga la malla.

[1] Sólo considera los valores significativos (no se contaron los casos con porcentaje negativo)
[2] Sólo considera los valores significativos (no se contaron los casos con *speed-up* menor que 1)

	Tiempos de Cálculo Mejora 1 (hh:mm:ss)			
Malla	Implementación [2]	Mejora 1	Reducción (%)	*Speed-up*
Neuronas				
cherry1	03:52:06	00:59:45	74,25 %	3,88
cherry2	02:57:33	00:44:28	74,95 %	3,99
cherry3	00:51:30	00:16:49	67,32 %	3,06
neuroest1	08:35:08	02:36:38	69,59 %	3,28
neuroest2	27:24:26	06:55:01	74,76 %	3,96
neuroest3	01:29:01	00:29:54	66,41 %	2,97
Cresta Neuronal				
crestaneuronal	17:31:36	03:27:26	80,27 %	5,06
Retículos				
retículo1	15:31:16	04:52:38	68,57 %	3,18
retículo2	00:00:57	00:01:00	-5,38 %	0,94
retículot00	10:39:11	02:06:28	80,21 %	5,05
retículot01	09:42:11	02:05:44	78,40 %	4,63
retículot02	10:32:36	02:08:51	79,62 %	4,90
retículot03	10:16:20	01:52:44	81,70 %	5,46
Mallas Sintéticas				
árbol0	00:00:08	00:00:09	-6,81 %	0,93
árbol1	00:00:16	00:00:13	18,66 %	1,22
árbol2	00:00:14	00:00:12	17,03 %	1,20
red0	00:09:03	00:01:30	83,39 %	6,02
red1	00:09:46	00:01:33	84,07 %	6,27
red2	00:08:31	00:01:33	81,80 %	5,49
Promedios			**69,5 %$\pm$20,2***	**4,1$\pm$1,5***

*Los promedios sólo consideran los valores significativos (no se contaron los casos con porcentaje negativo y los con *speed-up* menor que 1)

Tabla 3.3: Tiempos de cálculo de la etapa Colapso de la Malla para la implementación [2] y la Mejora 1.

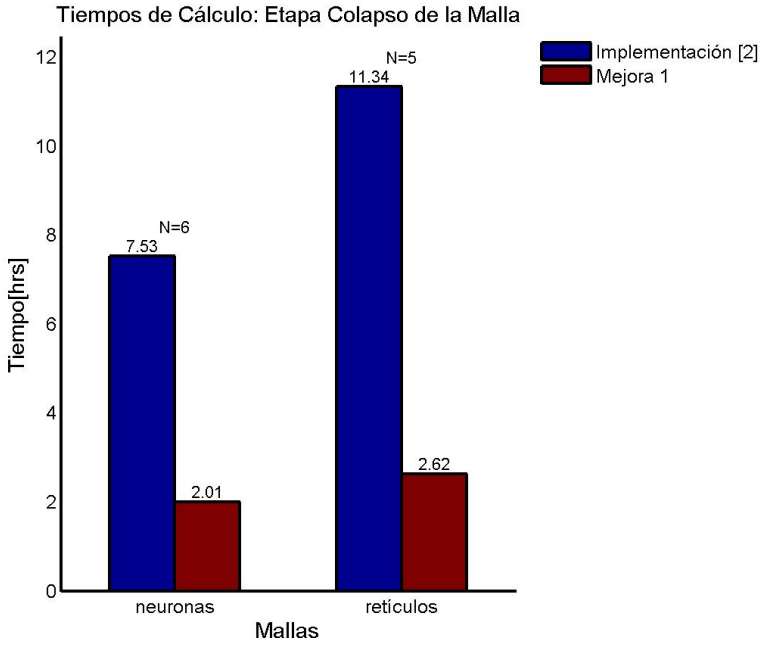


Figura 3.6: Tiempo promedio de cálculo de la etapa Colapso de la Malla al implementar la Mejora 1 con respecto a la imp. [2], para las mallas de neuronas y retículos. *Los promedios sólo consideran los valores significativos (no se contaron los casos negativos).

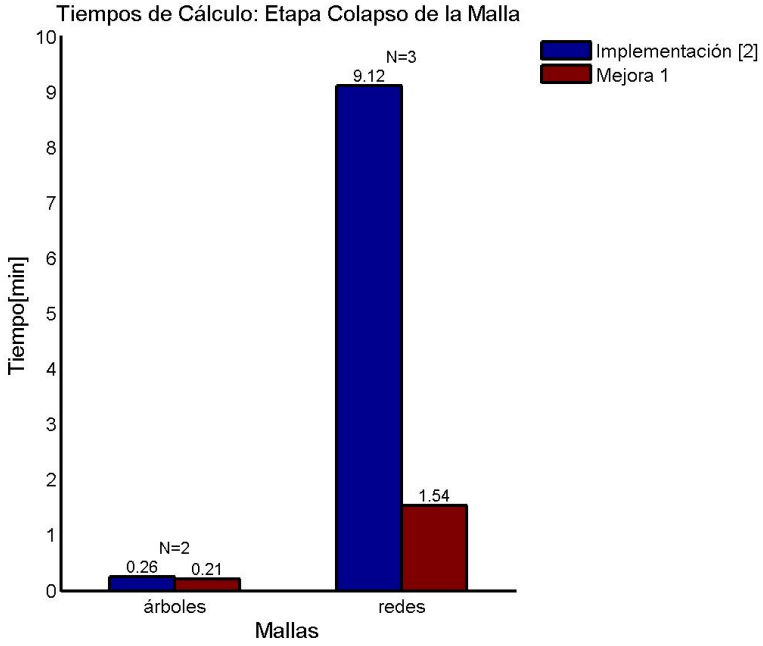


Figura 3.7: Tiempo promedio de cálculo de la etapa Colapso de la Malla al implementar la Mejora 1 con respecto a la imp. [2], para las mallas de árboles y redes. *Los promedios sólo consideran los valores significativos (no se contaron los casos negativos).

Se observa en la Figura 3.8 que existe dos grupos que tienen una gran dispersión en el porcentaje de reducción del tiempo, los retículos y los árboles. La dispersión de los retículos se explica por el resultado obtenido por el retículo2 (porcentaje negativo), caso que se explica de la misma forma que el caso del árbol0 antes descrito. De la misma forma, la dispersión en los árboles se debe al porcentaje negativo del árbol0.

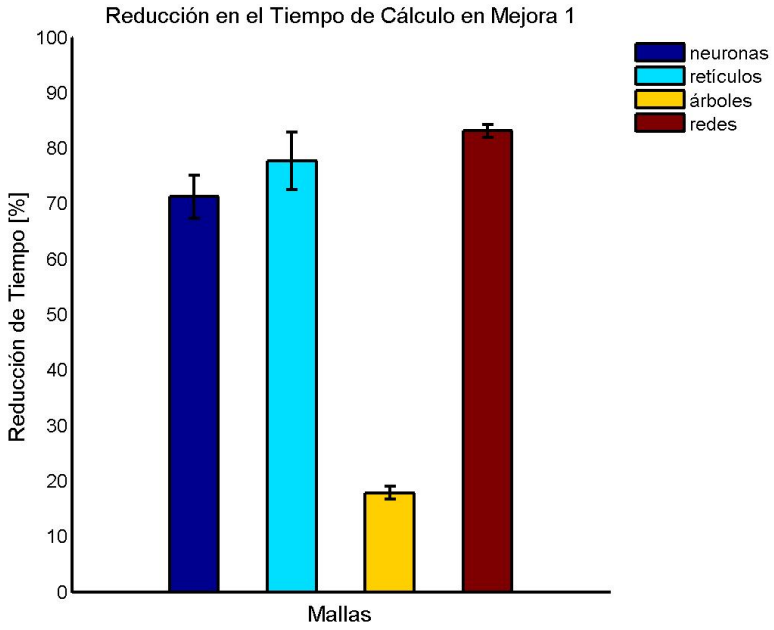


Figura 3.8: Porcentaje promedio de reducción en el tiempo de cálculo de la etapa Colapso de la Malla al implementar la Mejora 1 con respecto a la implementación [2]. *Para el cálculo de los promedios se consideran sólo los valores significativos (porcentajes no negativos).

3.2. Zona de Actualización de Costos (Mejora 2)

3.2.1. Actualización de Costos

Luego del colapso de un arco, viene el proceso de actualización de los costos de los arcos. Eso debido a que sus costos son afectados por el cambio en posiciones de algunos vértices. Para esto, se calcula por cada arco el *shape cost*, el *sampling cost*, y finalmente el costo total (explicados en la Sección 2.3.1).

$$Totalcost(i,j) = w_a \cdot Shapecost(i,j) + w_b \cdot Samplingcost(i,j)$$

En la implementación [2], luego de cada colapso, los costos son actualizados para todos los arcos restantes de la malla. Actualizar el costo de todos los arcos no es la mejor solución, porque al colapsar un arco sólo se cambia la malla localmente.

3.2.2. Zona de Actualización

Los cambios producto del colapso de un arco ocurren solamente dentro de una zona alrededor de donde ocurrió el colapso. Fuera de esa zona, los arcos y su vecindad se mantienen intactos, por lo que sus costos siguen siendo iguales.

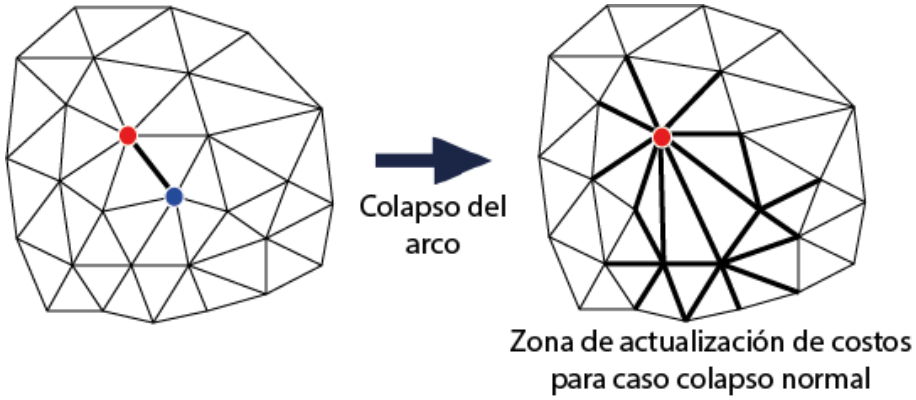


Figura 3.9: Zona de actualización de costos luego de colapsar un arco. Los arcos a actualizar su costo son los marcados.

Como nuevo cambio se define una zona limitada centrada en el vértice restante del arco colapsado. Esa zona corresponde a una vecindad de distancia 4. La distancia 4 resultó de la realización de mediciones, de las que se obtuvo la distancia mínima requerida. Para un colapso normal una vecindad de distancia 2 es suficiente (ver Figura 3.9), pero dado que se aceptan triángulos solapados, se necesita un radio mayor para garantizar un correcto funcionamiento.

3.2.3. Resultados

Luego de la Mejora 2 se observa un mejor desempeño en el tiempo de cálculo de la etapa Colapso de la Malla de entre 2x-10x (ver Tabla 3.4). El promedio del porcentaje de tiempo reducido fue de 80,1 %$\pm$9,3, con baja dispersión en el porcentaje promedio de reducción por grupo (ver Figura 3.12), y el *speed-up* promedio es 5,9$\pm$2,2.

El tiempo de cálculo se redujo fuertemente en todos los grupos de mallas (ver Figuras 3.10

y 3.11). Esta mejora en el tiempo fue gracias a que se pasó de una actualización en todos los m arcos (costo lineal) a sólo actualizar dentro de una región acotada con una cantidad de arcos « m, teniendo un costo aproximadamente constante para la actualización luego de un colapso.

	Tiempos de Cálculo Mejora 2 (hh:mm:ss)			
Malla	Mejora 1	Mejora 1+2	Reducción (%)	*Speed-up*
Neuronas				
cherry1	00:59:45	00:22:27	62.41 %	2,66
cherry2	00:44:28	00:15:26	65.28 %	2,88
cherry3	00:16:49	00:06:20	62.29 %	2,65
neuroest1	02:36:38	00:32:30	79.24 %	4,81
neuroest2	06:55:01	01:18:34	81.06 %	5,28
neuroest3	00:29:54	00:05:19	82.18 %	5,61
Cresta Neuronal				
crestaneuronal	03:27:26	01:13:35	64.52 %	2,81
Retículos				
retículo1	04:52:38	00:50:42	82.67 %	5,77
retículo2	00:01:00	00:00:12	78.81 %	4,71
retículot00	02:06:28	00:18:28	85.39 %	6,84
retículot01	02:05:44	00:20:18	83.85 %	6,19
retículot02	02:08:51	00:20:48	83.85 %	6,19
retículot03	01:52:44	00:17:02	84.88 %	6,61
Mallas Sintéticas				
árbol0	00:00:09	00:00:01	85.36 %	6,83
árbol1	00:00:13	00:00:01	87.52 %	8,01
árbol2	00:00:12	00:00:01	85.43 %	6,86
red0	00:01:30	00:00:08	90.50 %	10,53
red1	00:01:33	00:00:10	88.49 %	8,69
red2	00:01:33	00:00:10	88.69 %	8,84
Promedios			**80,1 %$\pm$9,3**	**5,9$\pm$2,2**

Tabla 3.4: Tiempos de ejecución de la etapa Colapso de la Malla para la Mejora 1 y para la Mejora 1+2. La reducción y el *speed-up* son calculados para la implementación con las mejoras 1 y 2 con respecto a la implementación con sólo la mejora 1.

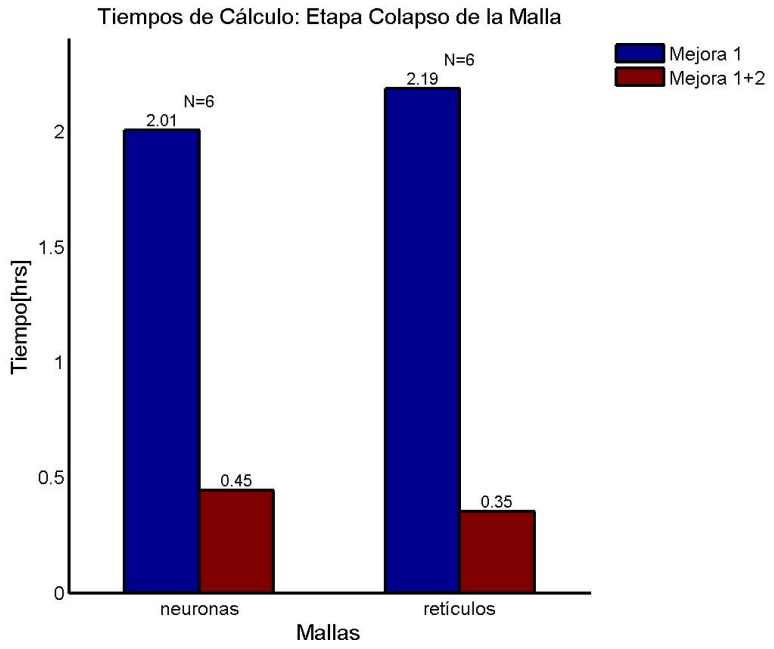


Figura 3.10: Tiempo promedio de cálculo de la etapa Colapso de la Malla al implementar la Mejora 1+2 con respecto a Mejora 1, para las mallas de neuronas y retículos.

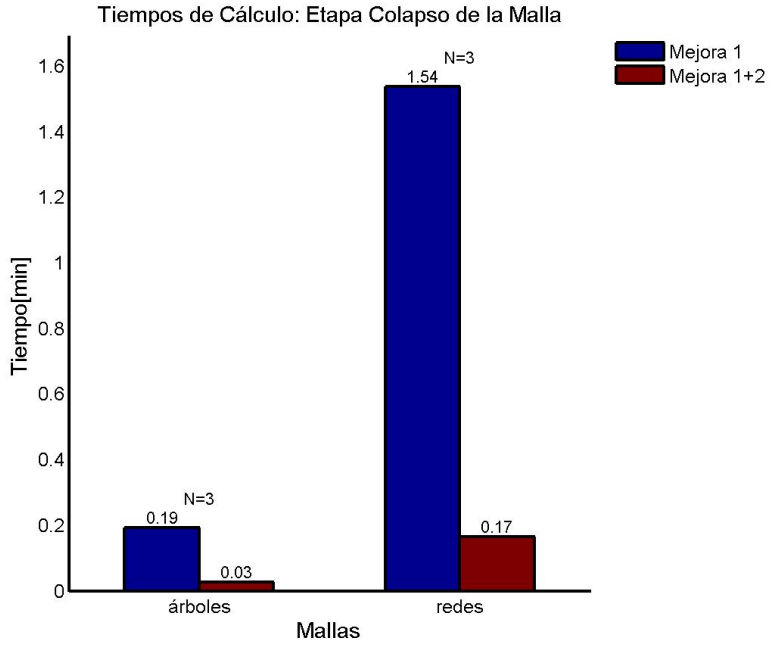


Figura 3.11: Tiempo promedio de cálculo de la etapa Colapso de la Malla al implementar la Mejora 1+2 con respecto a Mejora 1, para las mallas de árboles y redes.

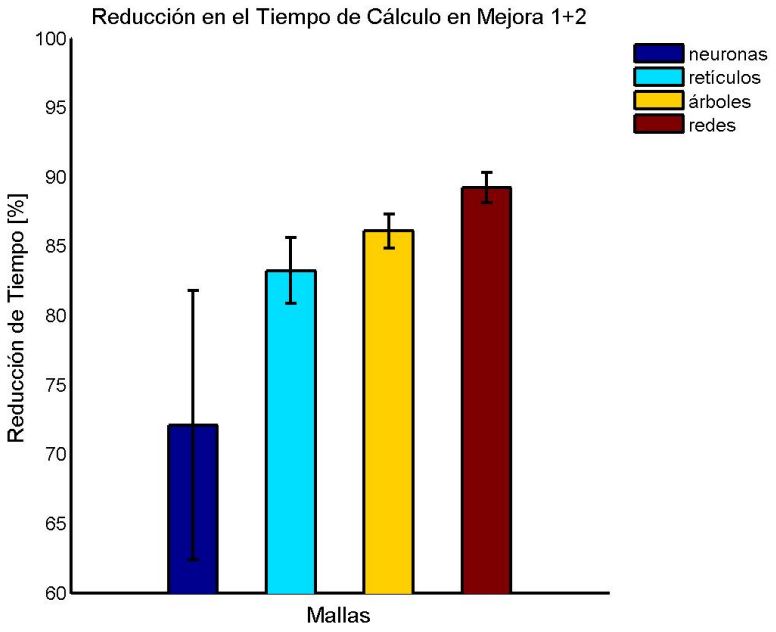


Figura 3.12: Porcentaje promedio de reducción en el tiempo de cálculo de la etapa Colapso de la Malla al implementar la Mejora 1+2 con respecto a las mejoras anteriores.

3.3. Optimización en la Construcción de la Cola de Costos (Mejora 3)

3.3.1. Construcción de la Cola en la Implementación 2011

En la implementación 2011 [2], para la elección del arco con menor costo se ordenaban todos los arcos por ese criterio en una cola. Luego de cada colapso, se recalculaba el costo de todos los arcos y se construía la cola nuevamente.

La construcción reiterada de la cola implica un costo de orden $O(mlog(m))$ para la construcción de la cola y $O(m^2log(m))$ para la etapa Colapso de la Malla (incluyendo todos los colapsos), siendo m la cantidad de arcos.

3.3.2. Nueva Construcción de la Cola de Costos

Para reducir el tiempo y costo, se decidió hacer la construcción de la cola una sola vez al inicio del colapso de la malla. Luego de cada colapso, cada arco eliminado se extrae de la cola de costos. Por otra parte, cada arco cuyo costo haya sido afectado por el colapso, se elimina y reinserta en la cola. Su reinserción garantiza su nueva posición en el ordenamiento dado su nuevo costo.

El solo hecho de no construir nuevamente la cola y reinsertar sólo los arcos que cambian de costo, provoca un importante ahorro en el costo de inserción de los arcos. Se pasa de un costo de orden $mlog(m)$ a $klog(m)$, siendo k la cantidad de arcos reinsertados dentro de la zona de colapso, y donde $k << m$.

3.3.3. Resultados

Luego de la Mejora 3 se observan mejoras en el tiempo de cálculo de la etapa Colapso de la Malla de entre 2x-4x (ver Tabla 3.5). El promedio del porcentaje de tiempo reducido fue de 60,3 %$\pm$17,0, con un baja dispersion del porcentaje promedio de reducción del tiempo por grupo de mallas (ver Figura 3.15). El *speed-up* promedio es 2,9$\pm$0,9.

Como se observa en las Figuras 3.13 y 3.14, los cambios en tiempo se mantienen sobre el 60 % en los grupos de neuronas y retículos, mientras que ese porcentaje es menor para las redes, en comparación a los resultados de las mejoras anteriores.

La mejora en tiempo se debe principalmente a la reducción en el orden, pasando de $O(m^2log(m))$ a $O(mlog(m) + mklog(m))$ (construcción de la cola + colapso de los arcos), con $k < m$. Se mantiene el costo logarítmico debido a que los arcos que se reinsertan en la cola, producto a su cambio en el costo, implican un costo $log(m)$ por operación.

	Tiempos de Cálculo Mejora 3 (hh:mm:ss)			
Malla	Mejora 1+2	Mejora 1+2+3	Reducción (%)	*Speed-up*
Neuronas				
cherry1	00:22:27	00:06:34	70,70 %	3,41
cherry2	00:15:26	00:04:42	69,47 %	3,27
cherry3	00:06:20	00:01:43	72,88 %	3,68
neuroest1	00:32:30	00:07:15	77,66 %	4,47
neuroest2	01:18:34	00:17:43	77,44 %	4,43
neuroest3	00:05:19	00:01:37	69,35 %	3,26
Cresta Neuronal				
crestaneuronal	03:27:26	00:24:15	67,03 %	3,03
Retículos				
retículo1	00:50:42	00:12:56	74,47 %	3,91
retículo2	00:01:00	00:00:05	59,85 %	2,49
retículot00	00:18:28	00:05:35	69,76 %	3,30
retículot01	00:20:18	00:05:56	70,73 %	3,41
retículot02	00:20:48	00:06:12	70,17 %	3,35
retículot03	00:17:02	00:05:47	66,00 %	2,94
Mallas Sintéticas				
árbol0	00:00:01	00:00:00	26,04 %	1,35
árbol1	00:00:01	00:00:01	31,05 %	1,45
árbol2	00:00:01	00:00:01	26,15 %	1,35
red0	00:00:08	00:00:04	51,95 %	2,08
red1	00:00:10	00:00:05	44,16 %	1,79
red2	00:00:10	00:00:05	50,07 %	2,00
Promedios			**60,3 %$\pm$17,0**	**2,9$\pm$0,9**

Tabla 3.5: Tiempos de ejecución de la etapa Colapso de la Malla para la Mejora 1+2 y para la Mejora 1+2+3. La reducción y el *speed-up* son calculados para la implementación con las mejoras 1, 2 y 3 con respecto a la implementación con sólo las mejoras 1 y 2.

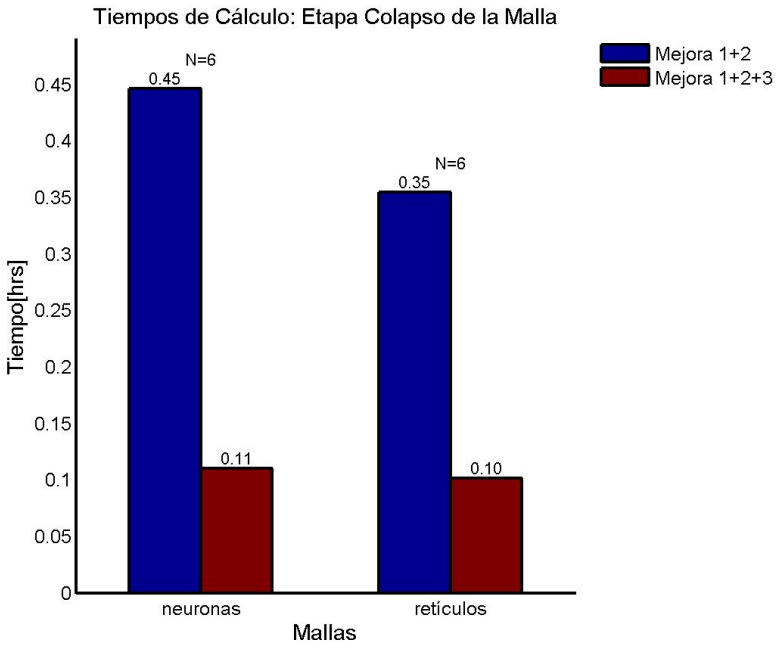


Figura 3.13: Tiempo promedio de cálculo de la etapa Colapso de la Malla al implementar la Mejora 1+2+3 con respecto a Mejora 1+2, para las mallas de neuronas y retículos.

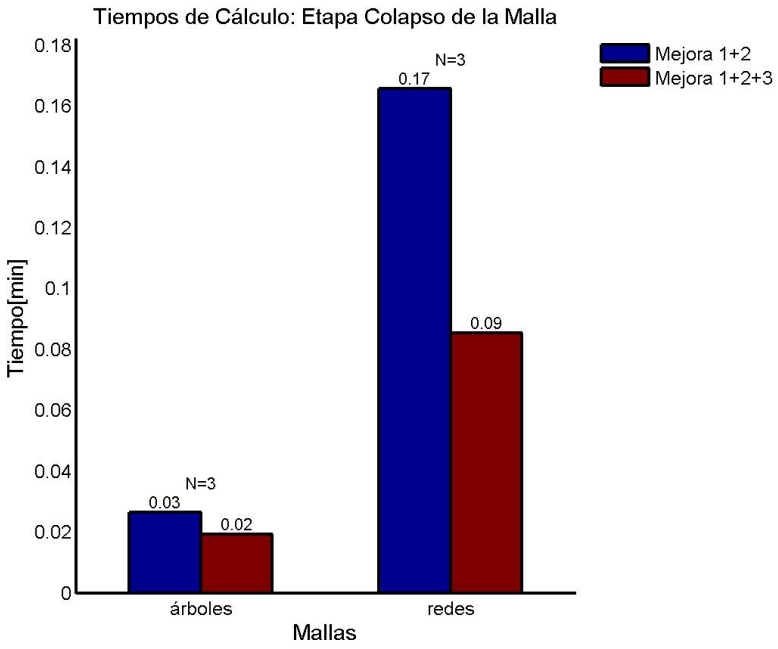


Figura 3.14: Tiempo promedio de cálculo de la etapa Colapso de la Malla al implementar la Mejora 1+2+3 con respecto a Mejora 1+2, para las mallas de árboles y redes.

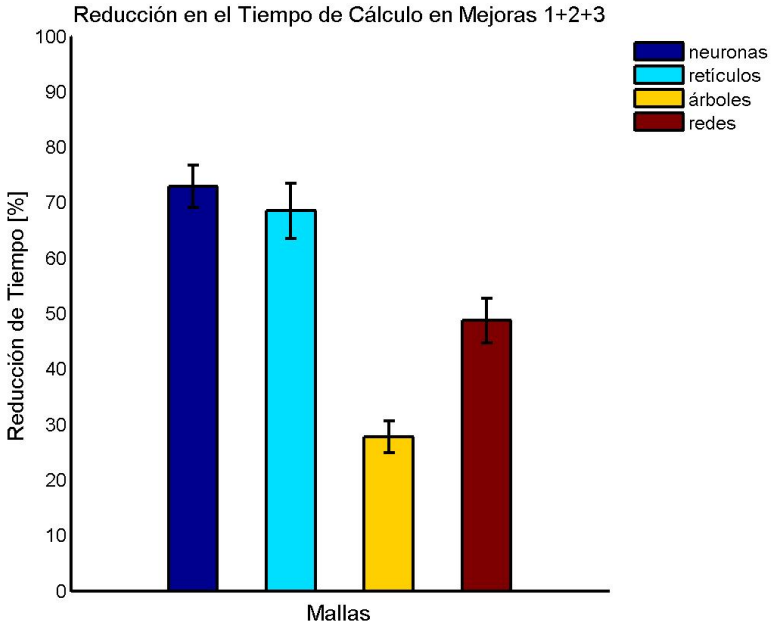


Figura 3.15: Porcentaje promedio de reducción en el tiempo de cálculo de la etapa Colapso de la Malla al implementar la Mejora 1+2+3 con respecto a las mejoras anteriores.

3.4. Optimización de Estructuras de Datos (Mejora 4)

3.4.1. TreeSet

La cola de costos que se usa en la implementación [2] es un objeto *PriorityQueue* [21]. Éste es una cola en que los elementos son ordenados de acuerdo a un orden natural o usando un comparador. Para el caso de los arcos, al comparador se le especifica que los arcos se ordenan por costo.

PriorityQueue provee $O(log(n))$ para la inserción y eliminación del primero (*remove()*). El problema es que la eliminación de un objeto específico (*remove(Object)*) es de orden lineal. Esto afecta en gran medida, dado que por cada colapso se eliminan en promedio 6 *halfedges* de la cola de costos.

Para aligerar el costo de eliminación, se cambió *PriorityQueue* por la estructura *TreeSet* [23]. Esta estructura está basada en la estructura *TreeMap* (árbol Rojo-Negro) y la estructura *Set* (conjunto que no acepta elementos repetidos). También ordena sus elementos usando un comparador.

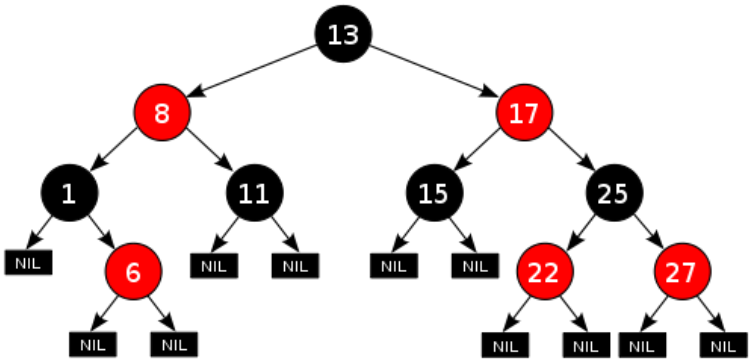


Figura 3.16: Árbol *Red-Black*, en el cual se basa la estructura *TreeSet*.

Se eligió *TreeSet* ya que permite ordenar y provee $O(log(n))$ para sus operaciones básicas, en especial la de remover. Eso se aplica tanto para remover un objeto en específico (*remove(Object)*) o remover el menor (de más a la izquierda en la Figura 3.16).

3.4.2. LinkedHashSet

La mayoría de las listas de objetos se han implementado usando la estructura *ArrayList* [20]. Esa estructura corresponde a un arreglo. *ArrayList* provee la adición de elementos en tiempo constante, y sus otras operaciones en tiempo lineal. Su desventaja es igual que con *PriorityQueue*, y es que la operación remover es de $O(n)$.

La estructura elegida para reemplazar a *ArrayList* fue *LinkedHashSet* [22]. Esta estructura es una lista doblemente enlazada y está basada en la estructura *Set* y *Hash*. *Set* requiere que no hayan elementos repetidos y *Hash* permite operaciones de tiempo constante.

Aunque *HashSet* también provee la creación de un conjunto sin elementos repetidos y una eliminación rápida, la iteración en sus elementos es más costosa. Eso se debe a que cuando existen eliminaciones, el espacio o *bucket* donde estaba el valor aun sigue existiendo pero sin valor. Así, cada iteración costará igual al número de elementos no vacíos más los vacíos.

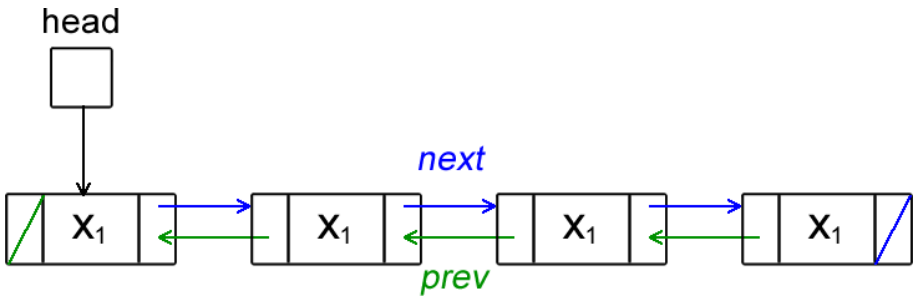


Figura 3.17: Lista doblemente enlazada, en la cual se basa la estructura *LinkedHashSet*.

3.4.3. Resultados

Luego de la Mejora 4 se observa una reducción en el tiempo de cálculo de la etapa Colapso de la Malla de entre 2x-3x (ver Tabla 3.6). El promedio del porcentaje de tiempo reducido fue de 58,3 %$\pm$14,3, con un baja dispersion del porcentaje promedio de reducción de tiempo por grupo de mallas (ver Figura 3.20). El *speed-up* promedio es 2,6$\pm$0,7.

Los cambios en los tiempos continuan siendo favorables y similares para todos los grupos exceptuando el grupo de los árboles (ver Figuras 3.18 y 3.19). Como se explicó en los resultados de la Mejora 1, existen factores (SO u otros procesos de ejecución) que corresponden a un costo fijo y constante (independientes del tamaño de la malla), y que para las mallas con pocos elementos se nota más su influencia.

	Tiempos de Cálculo Mejora 4 (hh:mm:ss)			
Malla	Mejora 1+2+3	Mejora 1+2+3+4	Reducción (%)	*Speed-up*
Neuronas				
cherry1	00:06:34	00:02:26	62,87 %	2,69
cherry2	00:04:42	00:01:47	62,08 %	2,63
cherry3	00:01:43	00:00:31	69,80 %	3,31
neuroest1	00:07:15	00:02:36	64,05 %	2,78
neuroest2	00:17:43	00:08:06	54,22 %	2,18
neuroest3	00:01:37	00:00:28	70,70 %	3,41
Cresta Neuronal				
crestaneuronal	00:24:15	00:11:40	51,90 %	2,07
Retículos				
retículo1	00:12:56	00:03:47	70,63 %	3,40
retículo2	00:00:05	00:00:01	62,99 %	2,70
retículot00	00:05:35	00:01:42	69,53 %	3,28
retículot01	00:05:56	00:01:58	66,62 %	2,99
retículot02	00:06:12	00:02:08	65,38 %	2,88
retículot03	00:05:47	00:01:53	67,41 %	3,06
Mallas Sintéticas				
árbol0	00:00:00	00:00:00	23,64 %	1,30
árbol1	00:00:01	00:00:00	29,28 %	1,41
árbol2	00:00:01	00:00:00	36,64 %	1,57
red0	00:00:04	00:00:01	71,40 %	3,49
red1	00:00:05	00:00:02	50,41 %	2,01
red2	00:00:05	00:00:02	57,60 %	2,35
Promedios			**58,3 %$\pm$14,3**	**2,6$\pm$0,7**

Tabla 3.6: Tiempos de ejecución de la etapa Colapso de la Malla para la Mejora 1+2+3 y Mejora 1+2+3+4. La reducción y el *speed-up* son calculados para la implementación con las mejoras 1, 2, 3 y 4 con respecto a la implementación con sólo las mejoras 1, 2 y 3.

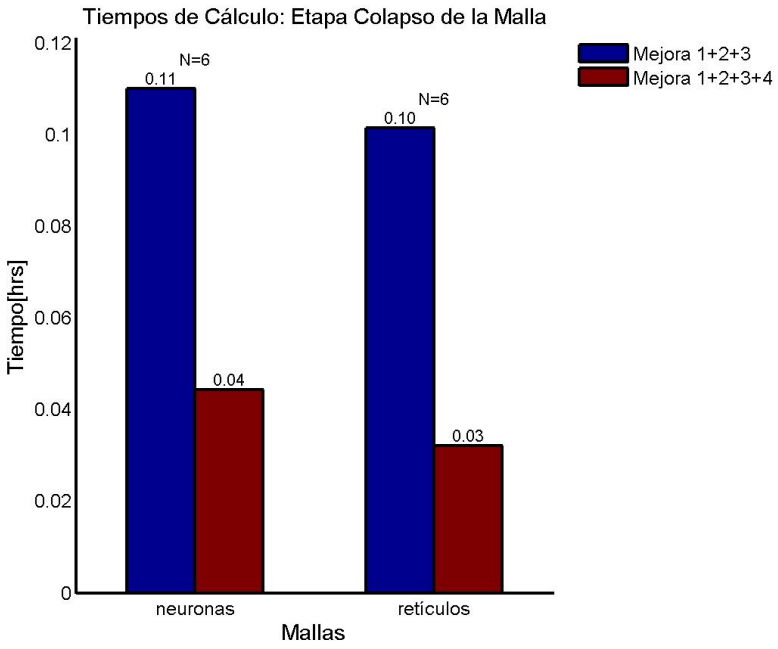


Figura 3.18: Tiempo promedio de cálculo de la etapa Colapso de la Malla al implementar la Mejora 1+2+3+4 con respecto a Mejora 1+2+3, para las mallas de neuronas y retículos.

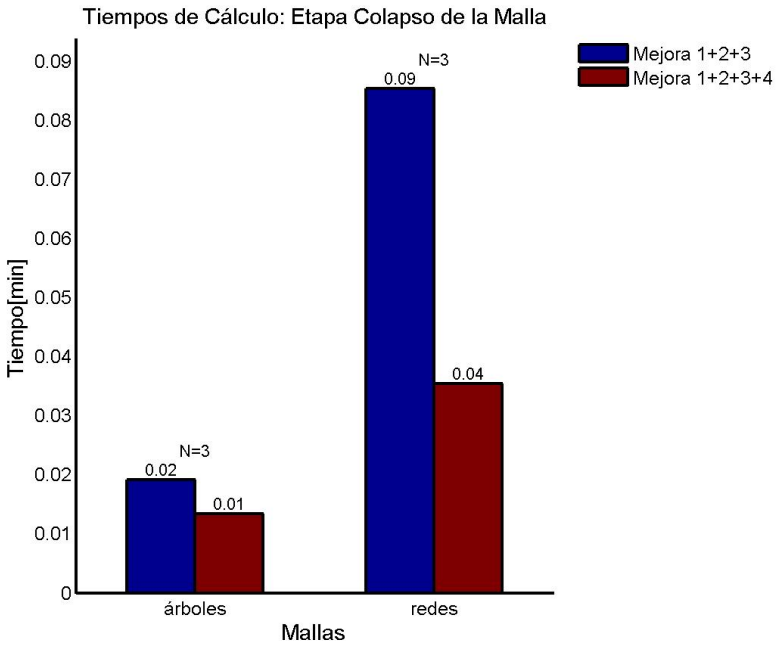


Figura 3.19: Tiempo promedio de cálculo de la etapa Colapso de la Malla al implementar la Mejora 1+2+3+4 con respecto a Mejora 1+2+3, para las mallas de árboles y redes.

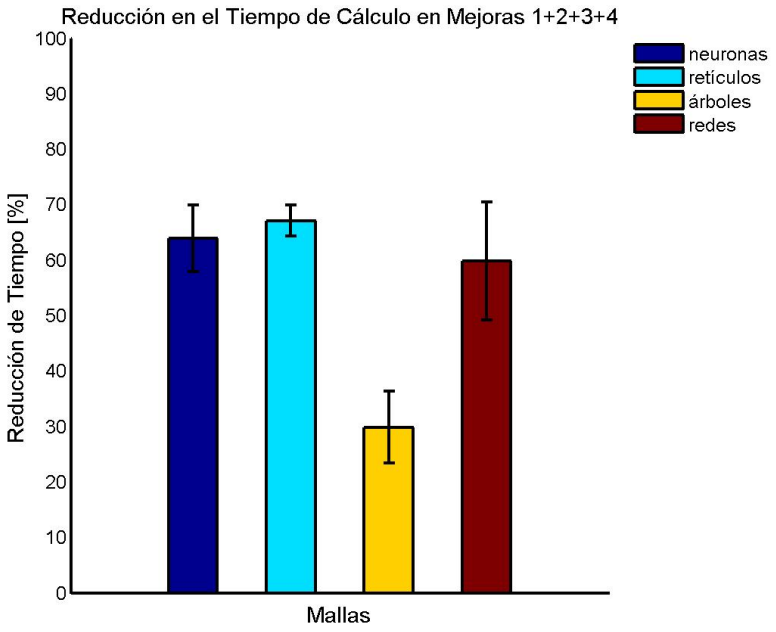


Figura 3.20: Porcentaje de reducción en el tiempo de cálculo de la etapa Colapso de la Malla al implementar la Mejora 1+2+3+4 con respecto a las mejoras anteriores.

3.5. Discusión

Luego de la realización de las distintas optimizaciones en la versión serial del programa, se observa una reducción en el tiempo de cálculo de un 98,1 %$\pm$2,9 en promedio, y un *speed-up* promedio de 182,3$\pm$130,6 en la etapa Colapso de la Malla (Tabla 3.7).

Todas las mejoras poseen desviaciones estándar alta debido a la hetereogeneidad de los tamaños de las distintas mallas. Esa variación también se presenta dentro de las mallas de los mismos grupos, aunque en menor grado.

Tanto los cambios que se observan en las Figuras 3.21 y 3.22, como los porcentajes promedios de reducción del tiempo (ver Figura 3.23) permiten estimar, dada la selección de una malla perteneciente a los mismos grupos de mallas asignados en las mediciones, la obtención de una mejora de más del 95 % si se procesa con la optimización serial implementada.

	Tiempos de Cálculo mejoras 1+2+3+4 (hh:mm:ss)			
Malla	Implementación [2]	Mejoras 1+2+3+4	Reducción (%)	*Speed-up*
Neuronas				
cherry1	00:59:45	00:02:26	98,94 %	95,02
cherry2	02:57:33	00:01:47	98,99 %	99,36
cherry3	00:51:30	00:00:31	98,99 %	99,11
neuroest1	08:35:08	00:02:36	99,49 %	197,36
neuroest2	27:24:26	00:08:06	99,50 %	202,72
neuroest3	01:29:01	00:00:28	99,46 %	186,20
Cresta Neuronal				
crestaneuronal	17:31:36	00:11:40	98.89 %	90,12
Retículos				
retículo1	15:31:16	00:03:47	99,59 %	245,07
retículo2	00:00:57	00:00:01	96,68 %	30,14
retículot00	10:39:11	00:01:42	99,73 %	375,58
retículot01	09:42:11	00:01:58	99,65 %	293,56
retículot02	10:32:36	00:02:08	99,66 %	294,53
retículot03	10:16:20	00:01:53	99,69 %	326,60
Mallas Sintéticas				
árbol0	00:00:08	00:00:00	91,16 %	11,32
árbol1	00:00:16	00:00:00	90,44 %	10,46
árbol2	00:00:14	00:00:00	94,34 %	17,68
red0	00:09:03	00:00:01	99,78 %	461,68
red1	00:09:46	00:00:02	99,49 %	197,11
red2	00:08:31	00:00:02	99,56 %	229,58
Promedios			**98,1 %$\pm$2,9**	**182,3$\pm$130,6**

Tabla 3.7: Tiempos de cálculo de la etapa Colapso de la Malla antes y después de la optimización serial, que incluye las cuatro mejoras.

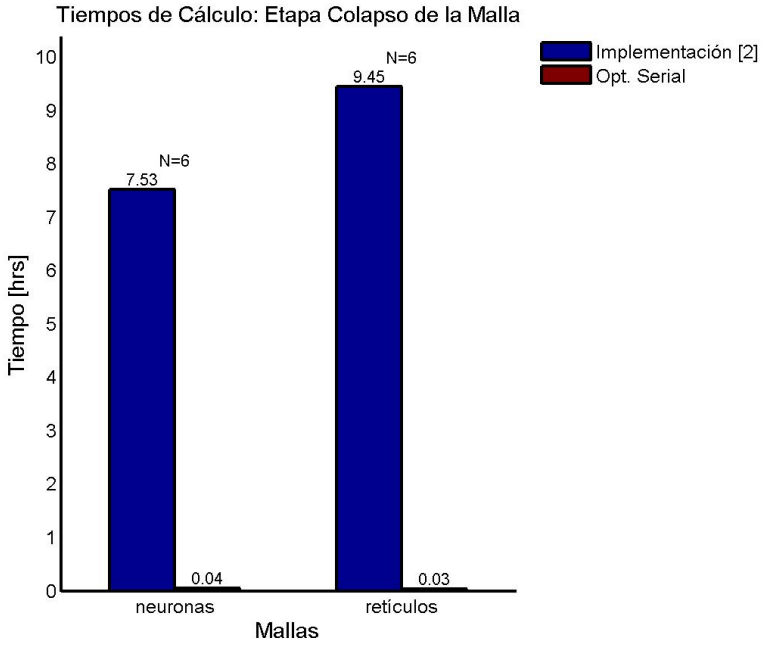


Figura 3.21: Tiempo promedio de cálculo de la etapa Colapso de la Malla al implementar la optimización serial, para las mallas de neuronas y retículos.

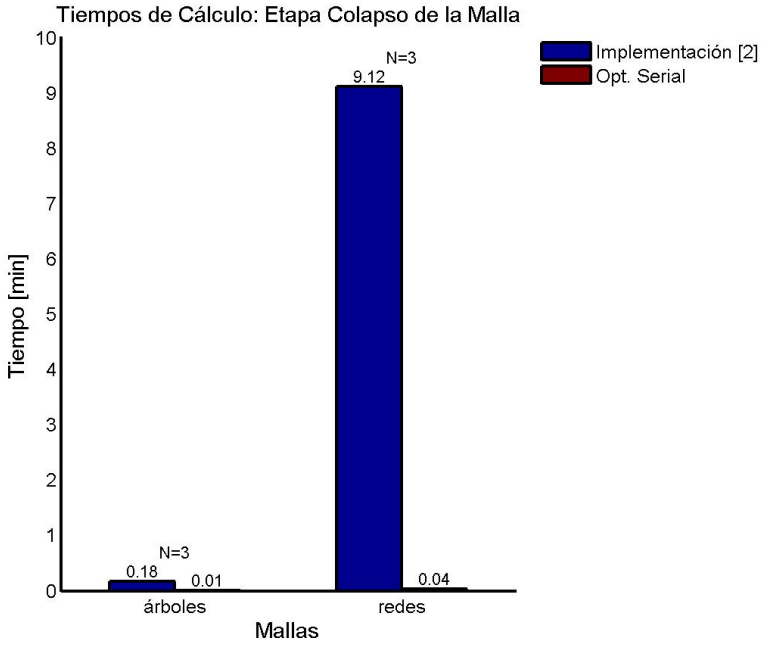


Figura 3.22: Tiempo promedio de cálculo de la etapa Colapso de la Malla al implementar la optimización serial, para las mallas de árboles y redes.

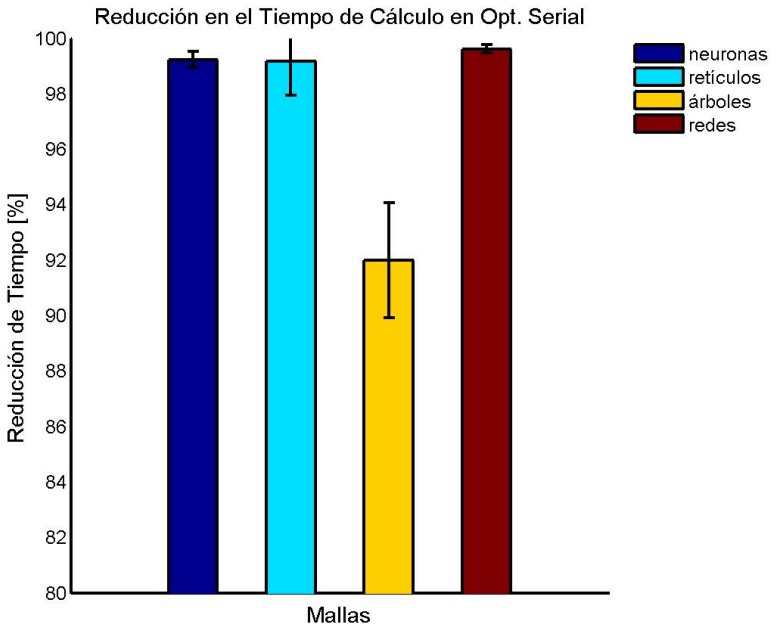


Figura 3.23: Porcentaje promedio de reducción en el tiempo de cálculo de la etapa Colapso de la Malla al implementar la Optimización Serial con respecto a la Impl. [2].

Los *skeletons* generados mantienen las mismas características que los de la implementación [2], tanto en la cantidad de nodos y segmentos, como en el largo total del *skeleton* (ver Tabla 3.8). Sin embargo, existen casos en que la cantidad de esos elementos se diferencia por tener más nodos y segmentos en la implementación *Halfedge* (ver filas marcadas en Tabla 3.8). Las diferencias se produjeron con el primer cambio, la implementación usando *Halfedge*, mostrando que aún existe una diferencia en la forma de tratar los distintos casos de colapsos (en el método o forma de colapso). La diferencia es de 1 nodo o 1 segmento, lo cual se puede deber a diferencias en la precisión de los números con que trabajan las dos implementaciones.

	Comparación de *skeletons* resultantes de la optimización serial					
Malla	Nodos		Segmentos		Largo total	
	Imp. [2]	Mejoras 1+2+3+4	Imp. [2]	Mejoras 1+2+3+4	Imp. [2]	Mejoras 1+2+3+4
Neuronas						
cherry1	603	603	606	606	2.158,43	2.158,43
cherry2	673	673	675	675	1.787,68	1.787,68
cherry3	365	365	366	366	1.146,05	1.146,05
neuroest1	**1.185**	**1.186**	**1.202**	**1.203**	**3.888,39**	**3.897.77**
neuroest2	1.447	1.447	1.471	1.471	5.127,39	5.127.39
neuroest3	811	811	820	820	2.331,16	2.331,16
Cresta Neuronal						
crestaneuronal	**697**	**698**	**734**	**735**	**4.135,93**	**4.130,01**
Retículos						
retículo1	1.041	1.041	**1.285**	**1.286**	**8.261,75**	**8267,477**
retículo2	240	240	244	244	752.59	752.59
retículot00	2.392	2.392	2.646	2.646	7.658,33	7.658,33
retículot01	2.436	2.436	2.694	2.694	7.782,73	7.782,73
retículot02	2.496	2.496	2.760	2.760	7.869,30	7.869,30
retículot03	2.517	2.517	2.786	2.786	7.850,55	7.850,55
Mallas Sintéticas						
árbol0	129	129	133	133	749,87	749.87
árbol1	200	200	204	204	953,97	953.97
árbol2	178	178	182	182	873,96	873.96
red0	1.848	1.848	1.858	1.858	7.654,81	7.654.81
red1	1.290	1.290	1.301	1.301	4.796,93	4.796.93
red2	1.612	1.612	1.622	1.622	5.635,79	5.635.79

Tabla 3.8: Cantidad de nodos y segmentos, y largo total de los *skeleton* resultantes para la implementación [2] y Mejoras 1+2+3+4, incluyendo todos los cambios de la optimización serial.

Finalmente, gracias a todas las optimizaciones realizadas, el orden de la etapa Colapso de la Malla se reduce de $O(m^2 log(m))$ a $O(mlog(m) + mklog(m))$.

Capítulo 4

Paralelización

Teniendo realizada la optimización serial, en este capítulo se procede a paralelizar la primera etapa del algoritmo, la Contracción de la Malla, haciendo uso de la unidad de procesamiento gráfico (GPU).

4.1. Identificación de Secciones Demandantes

Luego de las optimizaciones que se hicieron a la etapa Colapso de la Malla, se volvió a realizar una medición de los tiempos de ejecucion de cada etapa del algoritmo de *skeletonización*. Esto con el fin de determinar cuáles son las nuevas etapas más demandantes.

	Etapas del Algoritmo		
Malla	Contracción de la Malla	Colapso de la Malla	Corrección de la Malla
cherry1	00:00:17 **(10,45 %)**	00:02:26 **(89,48 %)**	00:00:00 **(0,07 %)**
cherry3	00:00:13 **(30,40 %)**	00:00:31 **(69,09 %)**	00:00:00 **(0,51 %)**
neuroest1	00:00:36 **(18,89 %)**	00:02:36 **(81,03 %)**	00:00:00 **(0,08 %)**
crestaneuronal	00:03:35 **(23,53 %)**	00:11:40 **(76,44 %)**	00:00:00 **(0,03 %)**
retículot00	00:00:45 **(32,55 %)**	00:01:42 **(67,17 %)**	00:00:00 **(0,28 %)**
retículot03	00:00:56 **(33,27 %)**	00:01:53 **(66,50 %)**	00:00:00 **(0,23 %)**

Tabla 4.1: Tiempos y porcentajes de las etapas del algoritmo para distintas mallas luego de la optimización serial.

Los resultados obtenidos de las nuevas mediciones mostraron que la etapa Colapso de la Malla sigue siendo una de las que ocupa el mayor tiempo (ver Tabla 4.1). A pesar de eso y a diferencia de las mediciones iniciales, ahora la etapa Contracción de la Malla aparece con un porcentaje de tiempo de cálculo comparable a la etapa Colapso de la Malla. Esto es esperable pues luego de todas las optimizaciones seriales realizadas se logró tener un orden $O(mlog(m) + mklog(m))$ para la etapa Colapso de la Malla.

Teniendo la optimización serial finalizada, se decidió paralelizar la etapa Contracción de la Malla en vez de la etapa Colapso de la Malla. Se eligió esa etapa porque es mucho más simple, y se trata de una operación conocida como paralelizable.

Con el fin de optimizar la etapa Contracción de la Malla, se decidió paralelizarla usando GPU. Esto debido a que esa etapa es una operación algebraica del tipo $Ax_i = x_{i+1}$, que es el caso típico de operación paralelizable con GPU.

4.2. Paralelización de la Etapa Contracción de la Malla

4.2.1. Representación de Operación Matricial

En la etapa Contracción de la malla los vértices se van desplazando hasta que la malla alcance un volumen cercano a 0. El desplazamiento de cada vértice se determina mediante el promedio de las posiciones de los vecinos y la posición del vértice. Ambos valores se ponderan por sus factores correspondientes y se suman, generando la nueva posición. Este proceso está resumido en el Algoritmo 1.

La operación de contracción se define en la ecuación:

$$v_i^{k+1} = v_i^k \cdot 0.5 + \left(\sum_{vecinos} \frac{1}{n_i} v_i^k\right) \cdot 0.5.$$

Matricialmente, esta ecuación se puede escribir como:

$$V^{k+1} = \left(AV^k + V^k\right) \cdot 0.5 = \tilde{A}V^k,$$

en donde:

- $\tilde{A}$ es la matriz de vecindad de tamaño $3n$x$3n$. $3n$ es porque contiene las tres coordenadas de los n vértices. Cada fila corresponde a los vecinos de un vértice. Los valores en la fila son 0

Algoritmo 1 Contracción de la Malla

procedure geometryContraction($mesh, halt\_criterion, displacement\_cirterion$)
  $w1 \leftarrow 0.5$
  $w2 \leftarrow 0.5$
  while $halt\_criterion.isTrue\,(mesh)$ **do**
    for all $vertex\ \epsilon\ mesh$ **do**
      $vector \leftarrow getAverageOfPositionNeighbors\,(vertex)$
      $position_1 \leftarrow vertex.position$
      $vertex.position \leftarrow w1 \cdot vertex.position + w2 \cdot vector.position$
      $position_2 \leftarrow vertex.position$
      $vertex.displacement \leftarrow |position_1 - position_2|$ ▷ desplazamiento del vértice
    end for
    if $displacement\_criterion.isTrue\,(mesh)$ **then**
      $break$
    end if
  end while
  return $mesh$
end procedure

si no es vecino y $\frac{1}{n\_vecinos} \cdot 0.5$ si lo es. Además, si se está en la fila de la coordenada x del vértice i, la columna de la misma coordenada de ese vértice tendrá valor $1 \cdot 0.5$ (que representa $w1 \cdot vertex.position$ en el Algoritmo 1).

$$\tilde{A} = \begin{array}{c} \\ a_x \\ a_y \\ a_z \\ b_x \\ b_y \\ \vdots \end{array} \begin{array}{c} \begin{array}{ccccc} a_x & a_y & a_z & b_x & \dots \end{array} \\ \begin{pmatrix} 1 \cdot 0.5 & 0 & 0 & \frac{1}{n_a} \cdot 0.5 & \dots \\ 0 & 1 \cdot 0.5 & 0 & 0 & \dots \\ 0 & 0 & 1 \cdot 0.5 & 0 & \dots \\ \frac{1}{n_b} \cdot 0.5 & 0 & 0 & 1 \cdot 0.5 & \dots \\ 0 & \frac{1}{n_b} \cdot 0.5 & 0 & 0 & \dots \\ \vdots & \vdots & \vdots & \vdots & \vdots \end{pmatrix} \end{array}$$

- V es la matriz de los valores de las coordenadas de los vértices, siendo su tamaño $3N$x1.

$$V = \begin{bmatrix} v_{a_x} & v_{a_y} & v_{a_z} & v_{b_x} & v_{b_y} & \dots \end{bmatrix}$$

$\tilde{A}$ tiene la particularidad de ser una matriz *sparse*. Una matriz *sparse* es una matriz compuesta mayoritariamente por elementos de valor 0. Tomando en cuenta que cada fila tiene todos los vértices, sólo muy pocos serán distintos de cero (la cantidad de vecinos por vértice es en promedio 6). Todos esos ceros generan muchas multiplicaciones innecesarias, ya que sólo interesan los pocos vecinos de cada vértice.

Otro problema que se presenta es el tamaño de la matriz. Si se tiene 100.000 vértices y los elementos de la matriz son tipo $double$, entonces el tamaño de $\tilde{A}$ sería:

$$3n \cdot 3n \cdot 8B = 72 \cdot n^2 \cdot B = 72 \cdot (10^5)^2 B = 72 \cdot 10^{10} B \approx 720GB$$

Los 720GB resultantes superan a la memoria disponible en las CPU y GPU del laboratorio, las cuales poseen alrededor de 1 o 2 GB de memoria.

Con el fin de manejar sólo los elementos necesarios y reducir el tamaño, se plantea un nuevo formato de la matriz. La idea es guardar, por cada vértice, sólo los elementos asociados a los vecinos distintos de 0 y los índices de los vecinos. Para ello se construye una matriz de 6nxMax, siendo Max el número máximo de vecinos de los vértices.

Si se toma el mismo caso anterior, con n = 100.000 y 8 Bytes por elemento en un arreglo, pero ahora con Max = 10, se logra una matriz de tamaño:

$$6n \cdot Max \cdot 8B = 48n \cdot 10 \cdot B = 480 \cdot 10^5 B = 48 \cdot 10^6 B \approx 48MB$$

Para pasar esa matriz de 2 dimensiones a GPU, hay que guardar en memoria cada fila de la matriz. En resumen, se hace un llamado malloc (asignación en memoria) por cada fila, lo cual es costoso en tiempo. Finalmente, para evitar hacer tantas veces la asignación de memoria, la matriz de vecindad se representó como dos arreglos de 1 dimensión y largo ($3n \cdot Max$). El primer arreglo contiene los índices de los vecinos y el segundo contiene los correspondientes valores distintos de 0.

4.2.2. Ejecución en GPU

Teniendo la matriz construida, lo siguiente es analizar cómo se paraleliza la multiplicación de las matrices. Paralelizar es crear múltiples tareas iguales que corran simultáneamente. En la multiplicación de matrices existe un paso que se repite muchas veces y que siempre tiene el mismo comportamiento, la multiplicación de una fila por una columna. Ese paso es el que genera cada elemento de la matriz resultante y es el objetivo de la paralelización. En CUDA, existe el *kernel*, que es la función que se ejecuta en cada *thread* de la GPU. Con eso, a cada *thread* se le designa la multiplicación de una fila por columna.

En el *kernel* implementado, el índice de cada *thread* corresponde al de un elemento en la matriz V (ver Algoritmo 2). De esa forma, cada nueva coordenada de V será calculada por una tarea independiente.

Algoritmo 2 *Kernel* de Multiplicación de Matrices

```
1: procedure multiply(Max, w1, A_indexs, A_values, V, V2, Displacement)
2:     tid ← getIndex()                                          ▷ Id del thread
3:     sum ← 0
4:     for i ← 0, Max do
5:         index_neighbor ← A_indexs[Max · tid + i]
6:         sum ← sum + A_values[Max · tid + i] · V[index_neighbor]
7:     end for
8:     sum ← sum + w1 · V[tid]
9:     V2_values[tid] ← sum                                      ▷ Nueva posición del vértice
10: end procedure
```

4.2.3. Criterios de detención

Los criterios de detención definidos son umbrales que al ser traspasados indican que se debe detener el proceso de contracción de la malla.

Criterio del Área

El área de la malla se calcula sumando el área de todos sus triángulos. Al contraer la malla, su área se va reduciendo. El umbral de detención, definido en la implementación [2], es el 15 % del área de la malla inicial. Su paralelización consiste en entregarle a cada *thread* una cara para que calcule su área, y hacer la suma total en CPU.

Criterio del Desplazamiento

Por cada modificación en la posición de los vértices durante la contracción, los vértices guardan su último desplazamiento. Por desplazamiento se toma la distancia recorrida entre la posición anterior y la nueva. Teniendo todos los desplazamientos de una contracción, se calcula el promedio de éstos. El umbral definido fue el valor 0,001. Si el desplazamiento promedio es menor que el umbral, quiere decir que la contracción ya no produce un efecto relevante, por lo que se detiene el proceso.

4.2.4. Resultados

Luego de la paralelización se observa un mejor desempeño en el tiempo de cálculo de la etapa Contracción de la Malla de entre 2x-4x (ver Tabla 4.2). El promedio del porcentaje de tiempo reducido fue de 22,7 %$\pm$60,6 y *speed-up* promedio es 2,2$\pm$1,4.

	Tiempo de Cálculo de la Paralelización (hh:mm:ss)			
Malla	Implementación [2]	Paralelización	Reducción (%)	*Speed-up*
Neuronas				
cherry1	00:00:17	00:00:08	51,02 %	2,04
cherry2	00:00:21	00:00:12	43,57 %	1,77
cherry3	00:00:13	00:00:08	35,63 %	1,55
neuroest1	00:00:36	00:00:09	72,75 %	3,67
neuroest2	00:01:17	00:00:19	75,35 %	4,05
neuroest3	00:00:05	00:00:02	59,40 %	2,46
Cresta Neuronal				
crestaneuronal	00:03:35	00:01:35	55,76 %	2,26
Retículos				
retículo1	00:02:12	00:00:31	76,13 %	4,18
retículo2	00:00:00	00:00:00	27,75 %	1,38
retículot00	00:00:45	00:00:12	74,58 %	3,93
retículot01	00:00:51	00:00:13	73,04 %	3,70
retículot02	00:00:56	00:00:14	74,04 %	3,85
retículot03	00:00:56	00:00:14	75,03 %	4,00
Mallas Sintéticas				
árbol0	00:00:00	00:00:00	-42,55 %	0,70
árbol1	00:00:00	00:00:00	-54,37 %	0,64
árbol2	00:00:00	00:00:00	-38,41 %	0,72
red0	00:00:00	00:00:00	-91,48 %	0,52
red1	00:00:00	00:00:00	-65,64 %	0,60
red2	00:00:00	00:00:00	-70,40 %	0,58
Promedios			**22,7 %$\pm$60,6**	**2,2$\pm$1,4**

Tabla 4.2: Tiempos de cálculo de la etapa Contracción de la Malla antes y después de su paralelización.

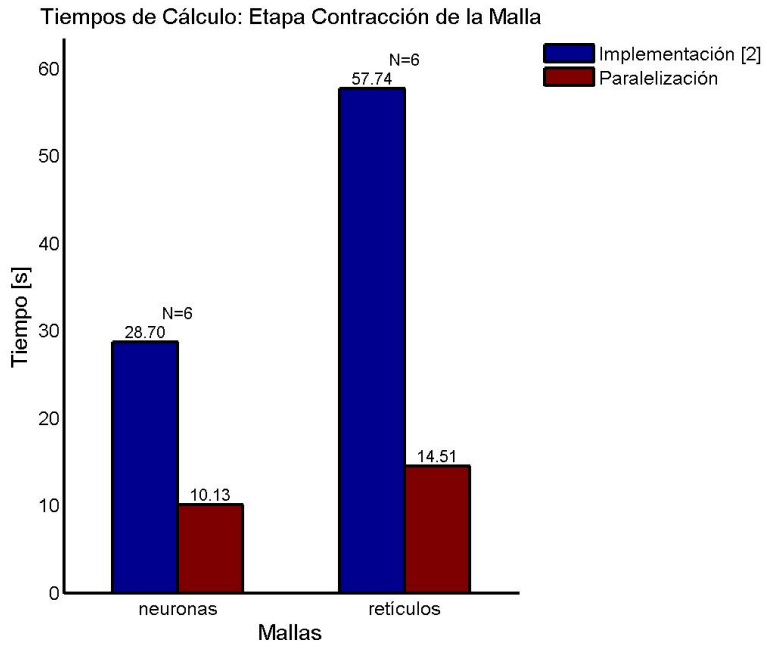


Figura 4.1: Tiempo promedio de cálculo de la etapa Contracción de la Malla al paralelizar, para las mallas de neuronas y retículos.

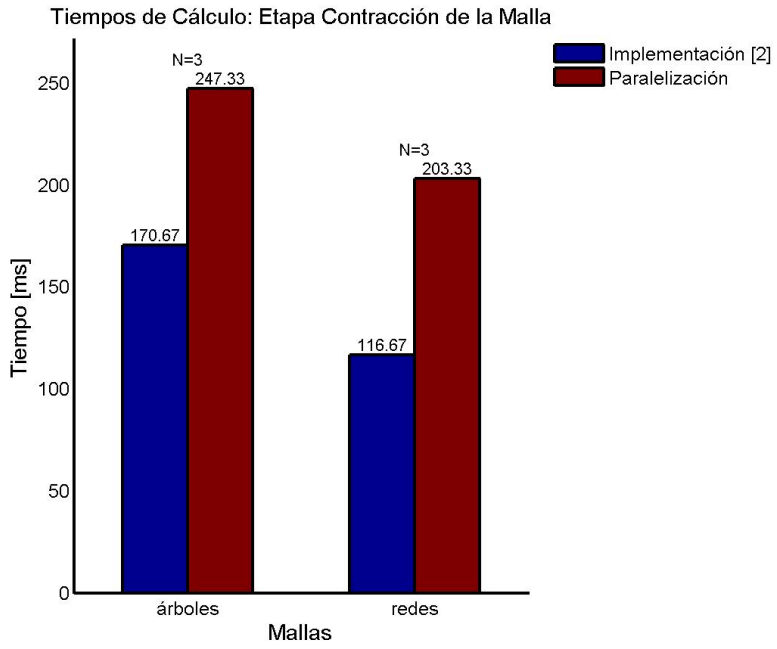


Figura 4.2: Tiempo promedio de cálculo de la etapa Contracción de la Malla al paralelizar, para las mallas de árboles y redes.

4.3. Discusión

Al paralelizar la etapa Contracción de la Malla, se obtiene una mejora de un 22,7 %±60,6 en el tiempo de cálculo y un *speed-up* de 2,2±1,4 (ver Tabla 4.2). Sin embargo, la cantidad de nodos y segmentos, y el largo se ven afectados para todas las mallas (ver Tabla 4.3).

Al igual que en la optimización serial, la paralelización resultó con una desviación estándar alta debido a la hetereogeneidad de los tamaños de las distintas mallas, la cual también se presenta en las mallas de los mismos grupos.

Ciertos resultados presentaron una particularidad: aumentaron el tiempo de cálculo luego de la paralelización (ver Figura 4.2 y 4.3). Eso ocurrió con las mallas sintéticas, siendo estas mallas las que tienen menos elementos comparadas con las demás mallas. Al tener pocos elementos (menos de 10 mil), influye más la implementación más que él número de vértices o arcos. Algunas pasos que en la implementación 2011 no están son la creación de las matrices a usar en la paralelización. Esa parte incurre en un costo que no existía en la implementación [2], así como también el costo de comunicación entre GPU y CPU. Esa comunicación implica traspasar las matrices entre las memorias de CPU y GPU.

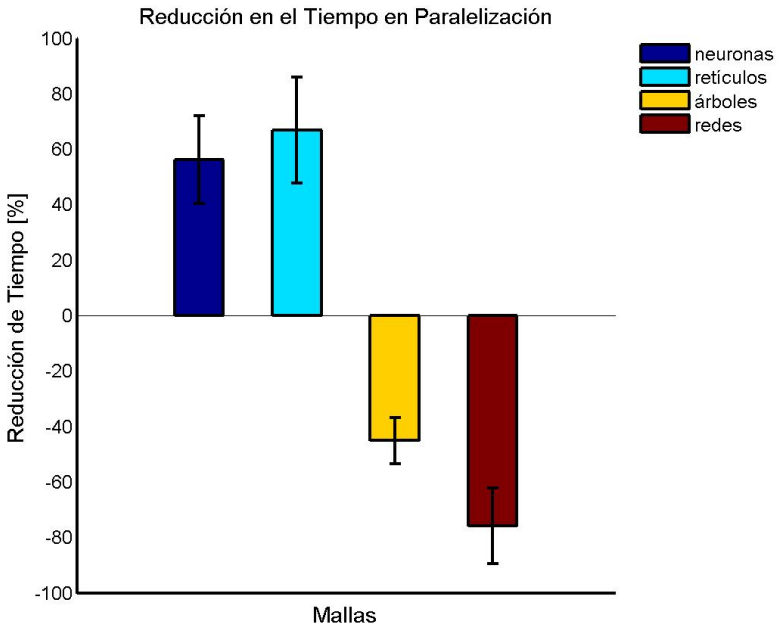


Figura 4.3: Porcentaje promedio de reducción en el tiempo de cálculo de la etapa Contracción de la Malla al implementar la Paralelización con respecto a la Impl. [2].

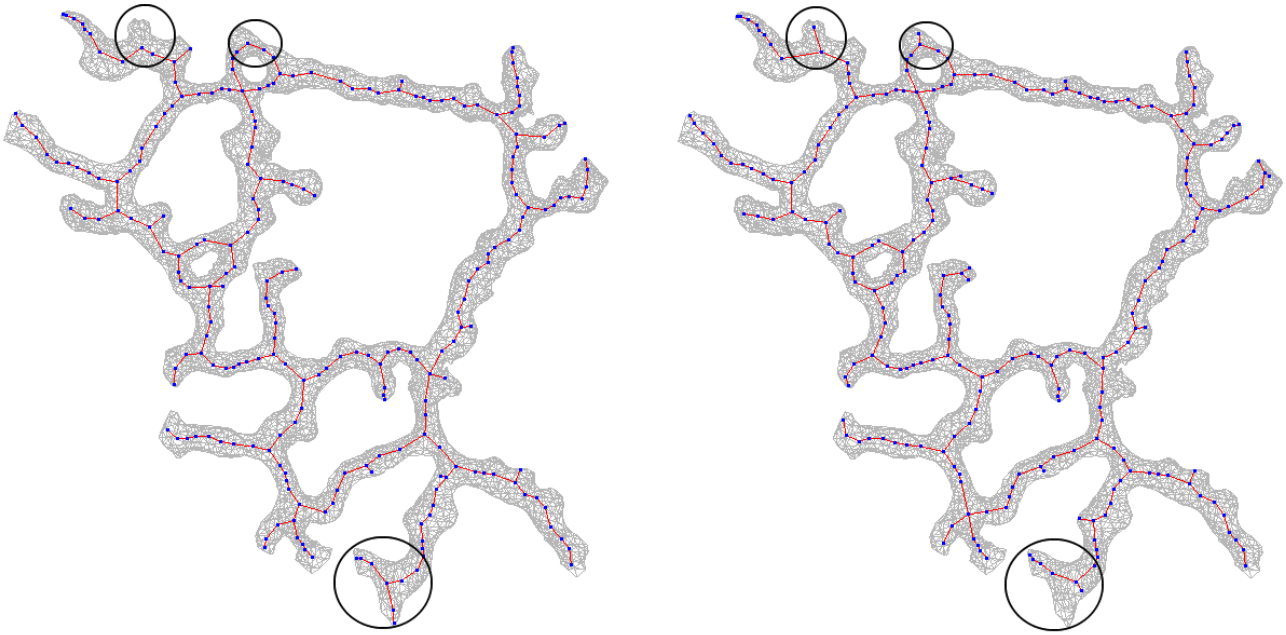
A diferencia de la versión serial de la etapa Contracción de la Malla, la paralelización realiza una

asignación simultánea de las nuevas coordenadas de los vértices. En la versión serial, la asignación de nuevas coordenadas se realiza dentro de un ciclo de manera incremental. Es decir, cuando se calculan las coordenadas de un vértice, se efectua la asignación (se guardan en el objeto *vertex*) y se prosigue con el siguiente vértice. Ese orden provoca que los cálculos de los vértices siguientes se vean afectados por las asignaciones anteriores. En cambio, en la paralelización el cálculo de las nuevas coordenadas se realiza en paralelo, y cuando se tienen todas las nuevas coordenadas se realizan las asignaciones.

La diferencia en el orden de pasos entre la versión serial y la paralelización produce contracciones diferentes para una misma malla, que a su vez resulta en *skeletons* diferentes.

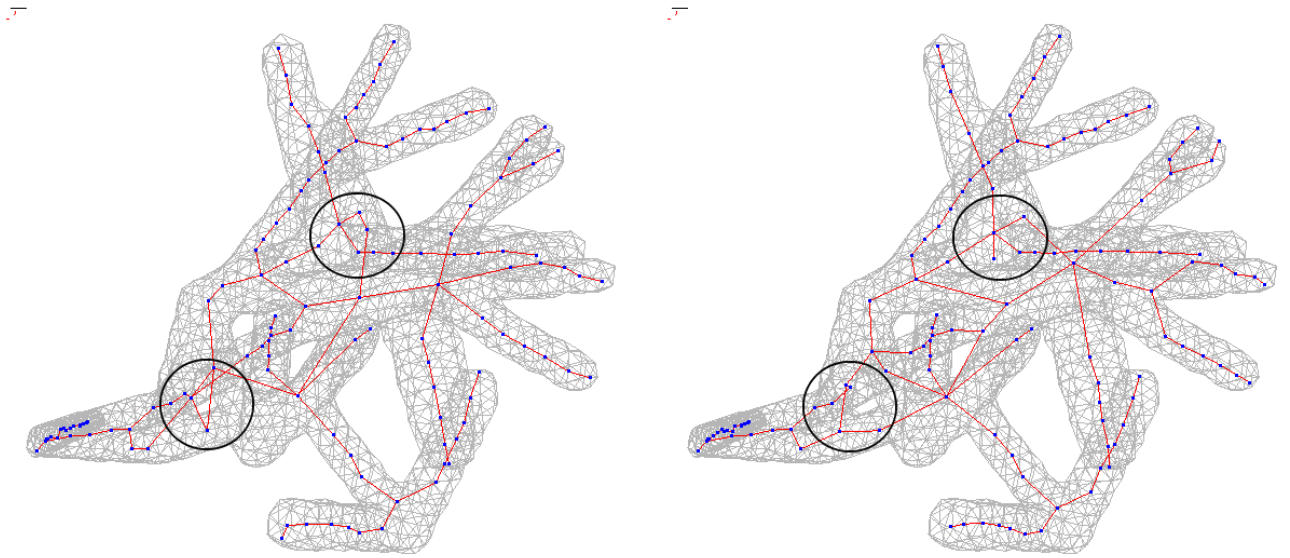
	Comparación de *skeletons* resultantes de la paralelización					
Malla	Nodos		Segmentos		Largo total	
	Imp. [2]	Paralel.	Imp. [2]	Paralel.	Imp. [2]	Paralel.
Neuronas						
cherry1	603	603	606	606	2.158,43	2.110,09
cherry2	673	689	675	691	1.787,68	1.867,88
cherry3	365	355	366	356	1.146,05	1.105,54
neuroest1	1.185	1.189	1.202	1.208	3.888,39	3.938,32
neuroest2	1.447	1.458	1.471	1.482	5.127,39	5.123,57
neuroest3	811	828	820	837	2.331,16	2.291,89
Cresta Neuronal						
crestaneuronal	697	697	734	733	4.135,93	4.144,85
Retículos						
retículo1	1.041	1.049	1.285	1.291	8.261,75	8.244,35
retículo2	240	242	244	246	752.59	752,27
retículot00	2.392	2.405	2.646	2.659	7.658,33	7.705,48
retículot01	2.436	2.422	2.694	2.678	7.782,73	7.752,50
retículot02	2.496	2.512	2.760	2.777	7.869,30	7.890,43
retículot03	2.517	2.508	2.786	2.777	7.850,55	7.833,90
Mallas Sintéticas						
árbol0	129	128	133	132	749,87	750,21
árbol1	200	196	204	200	953,97	950,25
árbol2	178	180	182	184	873,96	872,28
red0	1.848	1.884	1.858	1.894	7.654,81	7.659,23
red1	1.290	1.301	1.301	1.312	4.796,93	4.810,92
red2	1.612	1.615	1.622	1.625	5.635,79	5.630,11

Tabla 4.3: Cantidad de nodos y segmentos, y largo total de los *skeletons* resultantes para la implementación [2] y *Halfedge*, que incluye optimización serial y paralelización.



(a) *Skeleton* sin paralelización (a) *Skeleton* con paralelización

Figura 4.4: *Skeletons* de la malla retículo2 para la versión sin y con paralelizar.



(a) *Skeleton* sin paralelización (a) *Skeleton* con paralelización

Figura 4.5: *Skeletons* de la malla árbol0 para la versión sin y con paralelizar.

En las Figuras 4.4 y 4.5 se observa que las pequeñas diferencias radican en extremidades más ramificadas, y el cambio en la posición de intersecciones.

Capítulo 5

Conclusiones y Trabajo Futuro

5.1. Conclusiones

En este trabajo de título se optimizó la implementación serial y paralelizó la implementación de un algoritmo de generación de *skeletons* realizada en 2011 por Liliana Alcayaga [2]. Como metodología de trabajo se identificaron las secciones más demandantes en tiempo y pasos necesarios, con el fin de focalizar la optimización serial y la paralelización.

En la optimización de la versión serial se realizaron cuatro cambios: (1) implementación del algoritmo utilizando la estructura de datos *Halfedge*, (2) la optimización en la actualización de los costos, (3) la optimización en la construcción de la cola de costos, y (4) la optimización de estructuras de datos. Estas optimizaciones implicaron una reducción en el tiempo de cálculo del 98,1 %$\pm$2,9 en promedio, logrando reducir el tiempo del algoritmo de días y horas a minutos, e incluso segundos. En términos de complejidad desde una implementación $O(m^2log(m))$ se logró una implementacion $O(mlog(m) + mklog(m))$.

En la paralelización se trabajó en la etapa Contracción de la Malla, aprovechando su implementación simple, y su incidencia en hasta un 30 % del tiempo total de cálculo en la implementación serial. La paralelización implicó un *speed-up* de 2,2$\pm$1,4 (o reducción del 22,7 %$\pm$60,6).

Los resultados de este trabajo de título, tanto en la optimización de la implementación serial y la paralelización fueron exitosos, cumpliendo con el objetivo general de mejorar el tiempo de cálculo, acercándolo al tiempo real. Además, los resultados conservan la correctitud de los *skeletons*, pudiendo observar que se obtienen *skeletons* idénticos a los de la implementación 2011 [2], y

similares con la paralelización.

El uso de objetos geométricos construidos a partir de datos biológico ayudó a la corroboración del funcionamiento de la optimización y paralelización, debido a que poseen estructuras más complejas, con múltiples ramificaciones y túneles. Por otra parte, se probó con mallas sintéticas (creadas artificialmente) para tener resultados de estructuras con características aleatorias.

5.2. Trabajo Futuro

5.2.1. Paralelización de la Etapa Colapso de la Malla

Tomando en cuenta los tiempos de ejecución de la etapa Colapso de la Malla y la paralelización de la etapa Contracción de la Malla, un interesante trabajo futuro es la paralelización de la etapa Colapso de la Malla. En el Colapso de la Malla, se van colapsando los arcos según su costo. Los colapsos consecutivos son en general distantes, como se ve en la Figura A.13. Esta observación muestra que se puede dividir la malla en regiones y colapsar de manera paralela.

La paralelización de la etapa Colapso de la Malla se puede hacer dividiendo la malla en regiones, y por cada una de ellas realizar el proceso de colapsar los arcos segun su costo de colapso. Hay distintas formas de dividir la malla. Una de ellas es usando una grilla o cuadrícula (*grid*), o eligiendo vértices (semillas) y a partir de ellos ir eligiendo vértices y creando distintas regiones.

En el algoritmo detallado en el paper [24] se divide la malla en distintas regiones, cada una de ellas con un vértice en el centro. Esas regiones son encontradas buscando los conjuntos independientes maximales (MIS), conjuntos formados por los vértices que no comparten arcos en común. Luego de tener esas regiones, buscan de entre los vértices del MIS aquellos que sean el centro de una región de distancia 3, y que no contenga otros vértices de MIS. Así, usando los vértices elegidos, se definen áreas independientes, en donde no afecte el colapso de arcos de otras áreas. Definadas las áreas, se buscan los arcos de menor costo de entre los adyacentes a los vértices centros de las áreas independientes, y se colapsan. Luego de realizar los colapsos, se vuelven a buscar las áreas independientes y se repite el proceso de colapso.

El algoritmo anterior se implementa en [24] usando GPU, lo cual se puede incorporar aprovechando las características que proveen los computadores del laboratorio y el desarrollo previo usando GPU.

5.2.2. Manejo de Túneles

La implementación 2011 [2] y la actual (*Halfedge*) sufren de un problema: no siempre mantienen la cantidad original de túneles (ver ejemplo Figura A.15). Si bien esto es poco frecuente, en las mallas más complejas, como la cresta neuronal (ver Figura A.14), se observó la pérdida de 1 ó 2 túneles. Esta pérdida ocurre aún en la implementacion de Au et al [3], por lo cual se considera como algo propio del algoritmo.

Con el fin de evitar eliminar los túneles, en el algoritmo de Au et al [3] se propone una condición que también fue probada en el implementacion propuesta en esta memoria. Esta condición es que un arco no se puede colapsar si existen dos arcos más adyacentes tal que los tres juntos formen un triángulo sin caras. Si bien esta condición previene la eliminación de arcos que pertenezcan a un tunel mínimo (con sólo 3 arcos), también encuentra triángulos sin cara en el interior de la superficie. Eso provoca que se llega a un punto en el que ya no quedan arcos que se puedan colapsar, pero aún permanecen algunas caras.

Una forma de poder evitar la eliminación de arcos es conocer de antemano cuáles son los arcos que conforman los túneles de la malla. Existen algoritmos que buscan los ciclos de la superficie de una malla para definir los túneles [25] [26] [27]. Una propuesta a futuro es buscar los túneles antes de colapsar. Teniendo identificado los arcos que pertenecen a un ciclo que representa un túnel, hacerles un seguimiento durante los colapsos, y si llegan a forma un ciclo de 3 arco se prohibe su colapso.

Bibliografía

[1] Cornea, N. D., Silver, D. and Min, P. 2007. Curve-Skeleton Properties, Applications, and Algorithms. IEEE Transactions on Visualization and Computer Graphics, 13 (3), pp. 530-548.

[2] Alcayaga Gallardo, L. 2012. Generación de Skeletons a Partir de Mallas de Superficie. Memoria para optar al título de Ingeniería Civil en Computación, DCC, Universidad de Chile.

[3] Au, O. K., Tai, C., Chu, H., Cohen-Or, D. and Lee, T. 2008. Skeleton Extraction by Mesh Contraction. ACM Trans. Graph., 27 (3), Artículo 44, pp. 1-10.

[4] Laboratory for Scientific Image Analysis. Dirección URL: `http://www.scian.cl/portal/globals.php?COD_SECCION=2988`, última visita: Abril de 2013.

[5] Villarroel, S., Cerda, M., Santibañez, F., Ramirez, O., Palma, K., Concha, M., Hitschfeld, Nancy and Härtel, S. 2012. Defining Metrics to Compare Skeleton Estimation Methods Applied to Biological Structures. XXVI Annual Meeting of the Sociedad de Biologa Celular de Chile. Pto. Varas, Chile. Poster presentation.

[6] CUDA. Dirección URL: `https://developer.nvidia.com/category/zone/cuda-zone`, última visita: Diciembre de 2013.

[7] Sanders, J. and Kandrot, E. 2010. CUDA by Example: An Introduction to General-Purpose GPU Programming (1st ed.). Addison-Wesley Professional.

[8] NVIDIA. Dirección URL: `http://www.nvidia.es/object/gpu-computing-es.html`, última visita: Diciembre de 2013.

[9] OpenCL. Dirección URL: `http://www.khronos.org/opencl/`, última visita: Julio de 2013.

[10] R. Akagi, OpenCL, Universidad Católica - Nuestra Señora de la Asunción, `http://www.jeuazarru.com/docs/OpenCL.pdf`.

[11] Wang, Y. and Lee, T. 2008. Curve-Skeleton Extraction Using Iterative Least Squares Optimization. IEEE Transactions on Visualization and Computer Graphics, 14 (4), pp. 926-936.

[12] Bloomenthal, J. 2002. Medial-based Vertex Deformation, paper presented at ACM, pp. 147-151.

[13] Aylward, S. R., Jomier, J., Weeks, S. and Bullitt, E. 2003. Registration and Analysis of Vascular Images. Int. J. Comput. Vision, 55 (2-3), pp. 123-138.

[14] Brennecke, A. and Isenberg, T. 2004. "3D Shape Matching Using Skeleton Graphs.", paper presented at SCS Publishing House e.V., pp. 299-310.

[15] Eckhardt, U. and Maderlechner, G. 1993. Invariant Thinning. IJPRAI, 7 (5), pp. 1115-1144.

[16] Wan, M., Dachille, F. and Kaufman, A. 2001. Distance-field Based Skeletons for Virtual Navigation, paper presented at IEEE Computer Society, pp. 239-246.

[17] Br, T, J. W. and Algazi, V. R. 1991. Continuous Skeleton Computation by Voronoi Diagram. CVGIP: Image Underst., 55 (3), pp. 329-338.

[18] Chuang, J., Tsai, C. and Ko, M. 2000. Skeletonization of Three-Dimensional Object Using Generalized Potential Field. IEEE Trans. Pattern Anal. Mach. Intell., 22 (11), pp. 1241-1251.

[19] CGAL. Dirección URL: `http://doc.cgal.org/latest/HalfedgeDS/index.html`, última visita: Diciembre de 2013.

[20] ArrayList. Dirección URL: `http://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html`, última visita: Diciembre de 2013.

[21] PriorityQueue. Dirección URL: `http://docs.oracle.com/javase/7/docs/api/java/util/PriorityQueue.html`, última visita: Diciembre de 2013.

[22] LinkedHashSet. Dirección URL: `http://docs.oracle.com/javase/7/docs/api/java/util/LinkedHashSet.html`, última visita: Diciembre de 2013.

[23] TreeSet. Dirección URL: `http://docs.oracle.com/javase/7/docs/api/java/util/TreeSet.html`, última visita: Diciembre de 2013.

[24] Papageorgiou, A. and Platis, N. 2013. Triangular mesh simplification on the GPU.

[25] Dey, T. K., Li, K. and Sun, J. 2007. On Computing Handle and Tunnel Loops, paper presented at IEEE Computer Society, pp. 357-366.

[26] Dey, T. K., Li, K., Sun, J. and Cohen-Steiner, D. 2008. Computing Geometry-aware Handle and Tunnel Loops in 3D Models. ACM Trans. Graph., 27 (3), pp. 451-459.

[27] Dey, T. K., Fan, F. and Wang, Y. 2013. An Efficient Computation of Handle and Tunnel Loops via Reeb Graphs. ACM Trans. Graph., 32 (4), Artículo 32, pp. 1-10.

[28] Halfedge. Dirección URL: `http://leonardofischer.com/dcel-data-structure-c-plus-plus-implementation/`, última visita: Enero de 2014.

Apéndice A

A.1. Mallas iniciales

Figura A.1: Malla de la neurona cherry1.

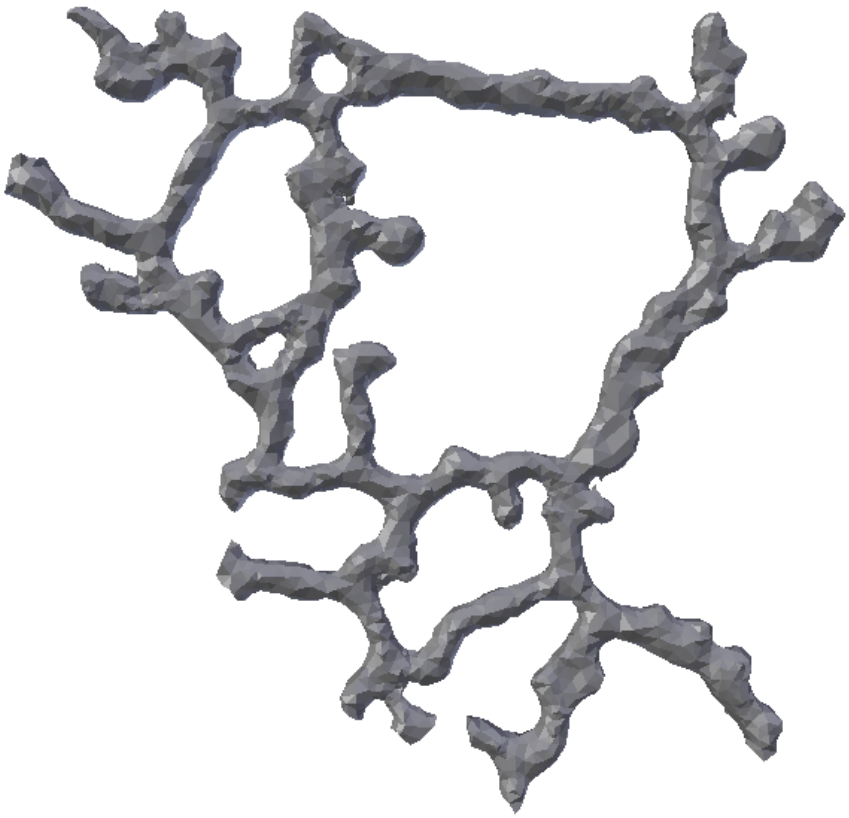


Figura A.2: Malla del retículo retículo2.

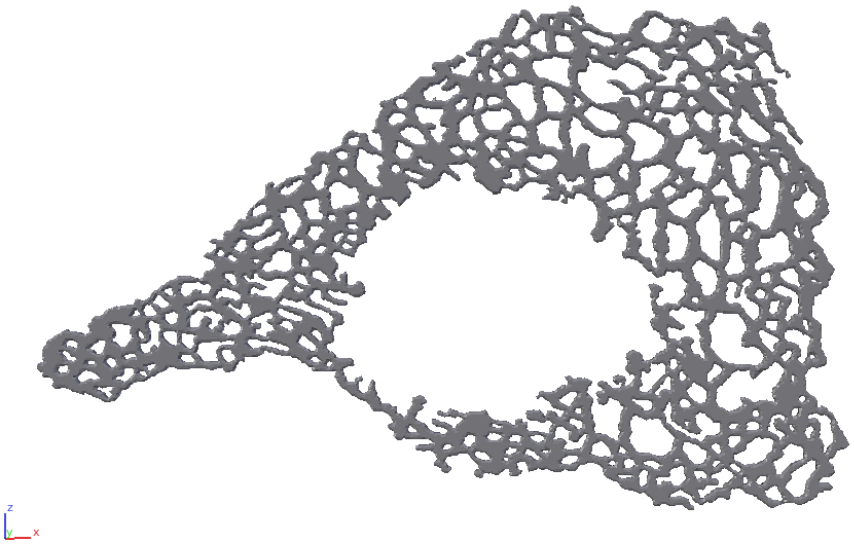


Figura A.3: Malla del retículo retículot00.

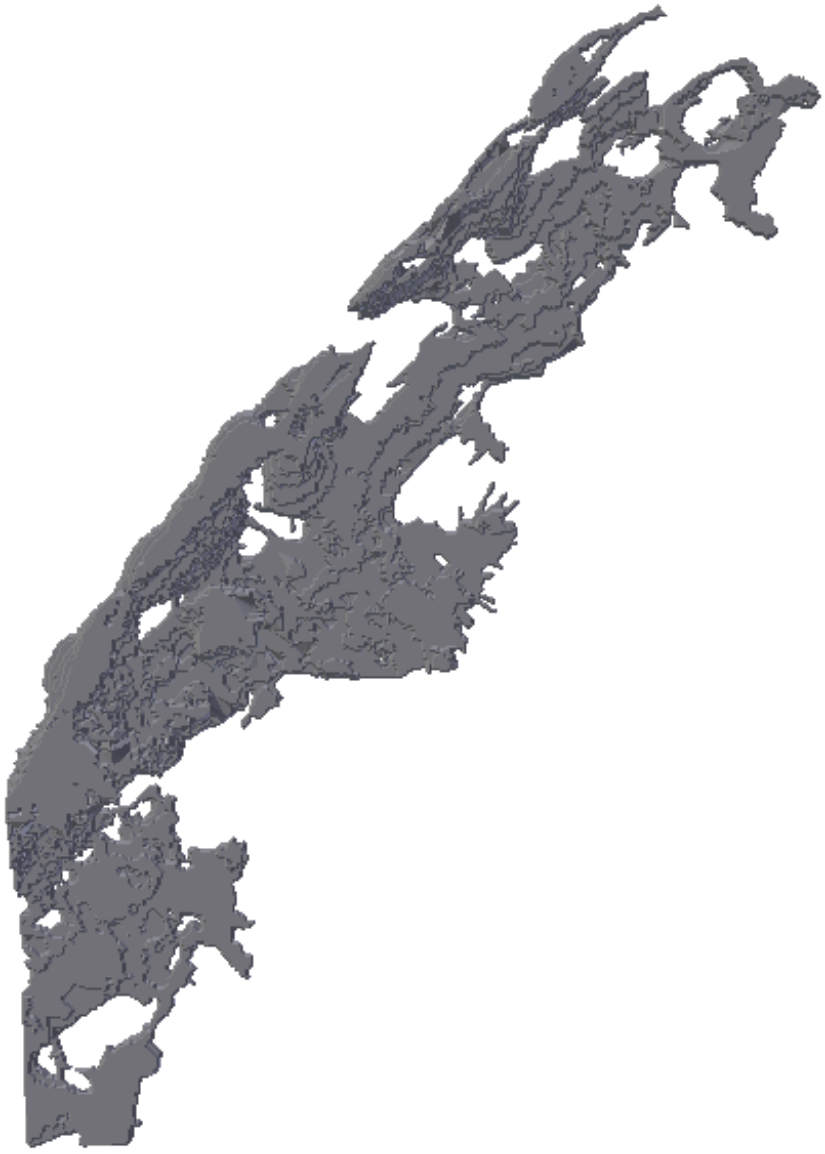


Figura A.4: Malla de la cresta neuronal.

Figura A.5: Malla de la neurona neuroest1.

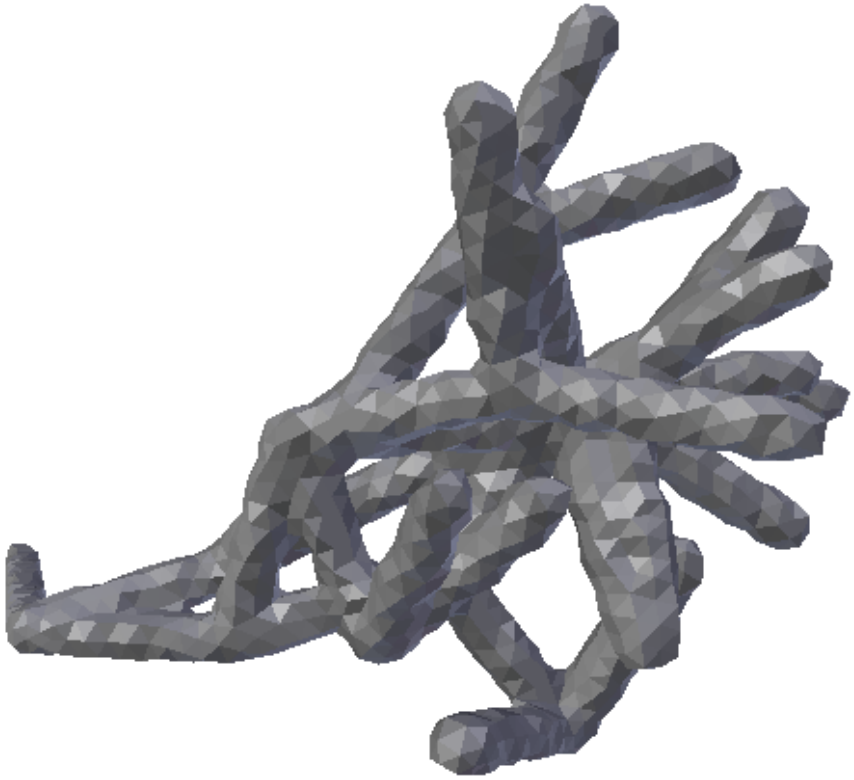


Figura A.6: Malla del árbol árbol0.

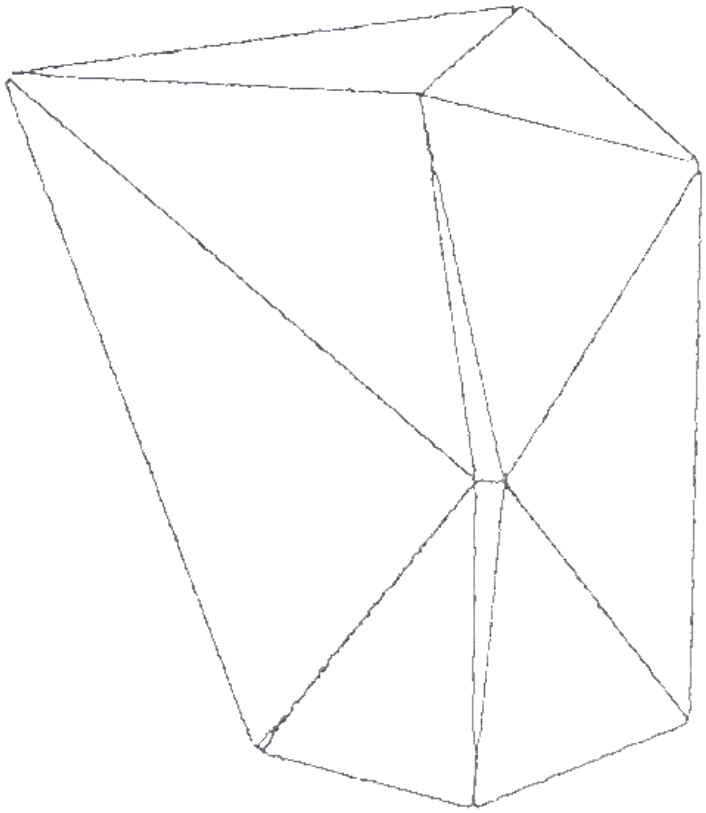


Figura A.7: Malla de la red red0.

A.2. *Skeletons*

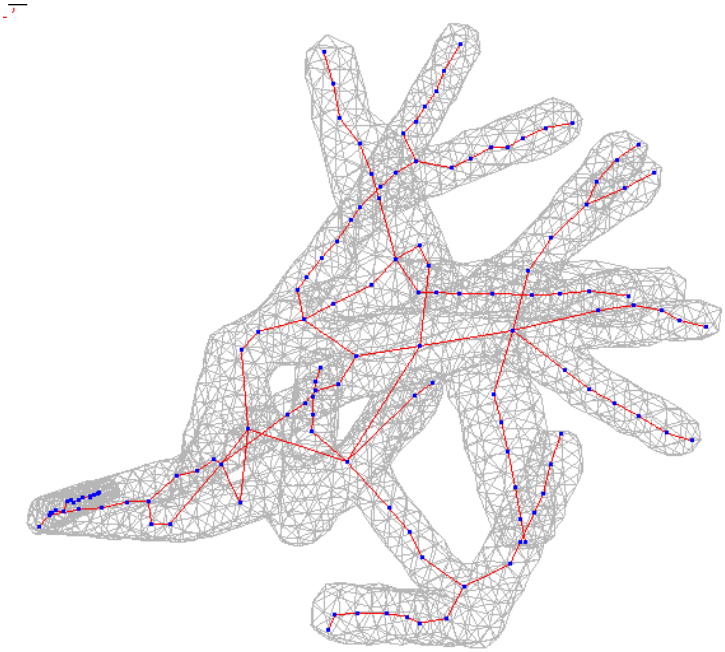


Figura A.8: *Skeleton* de la malla árbol0.

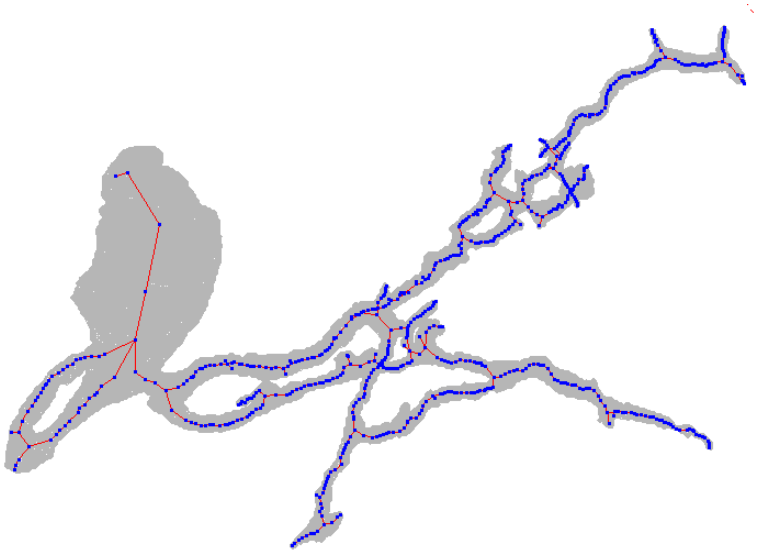


Figura A.9: *Skeleton* de la malla cherry2.

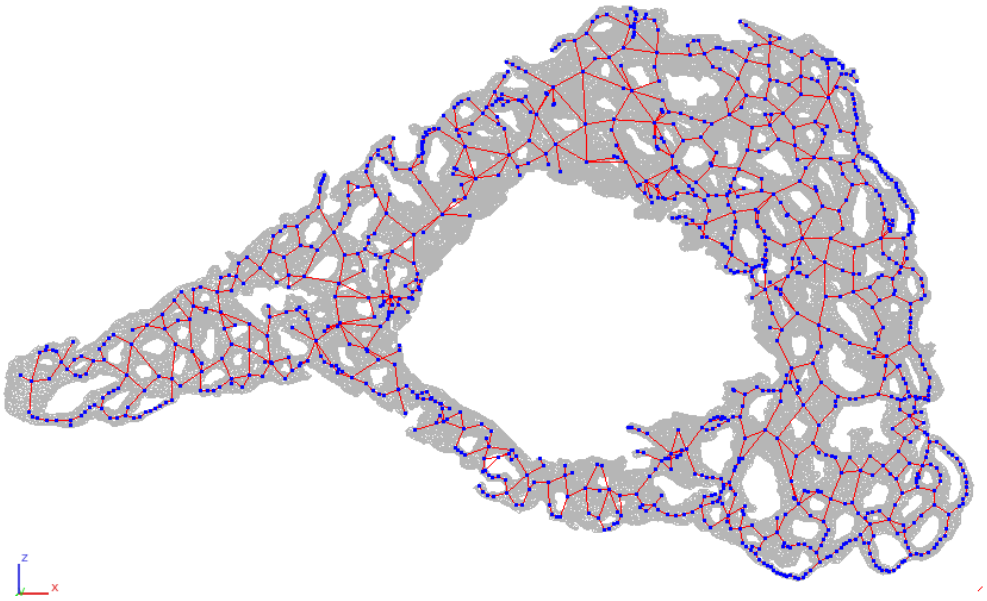


Figura A.10: *Skeleton* de la malla retículo1.

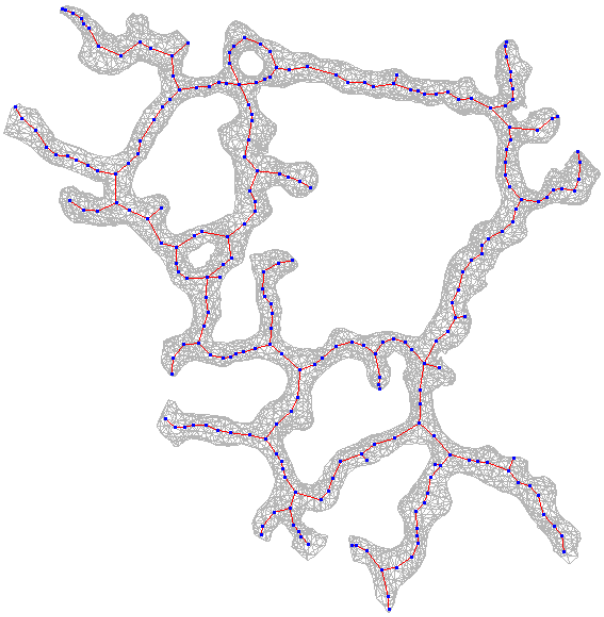


Figura A.11: *Skeleton* de la malla retículo2.

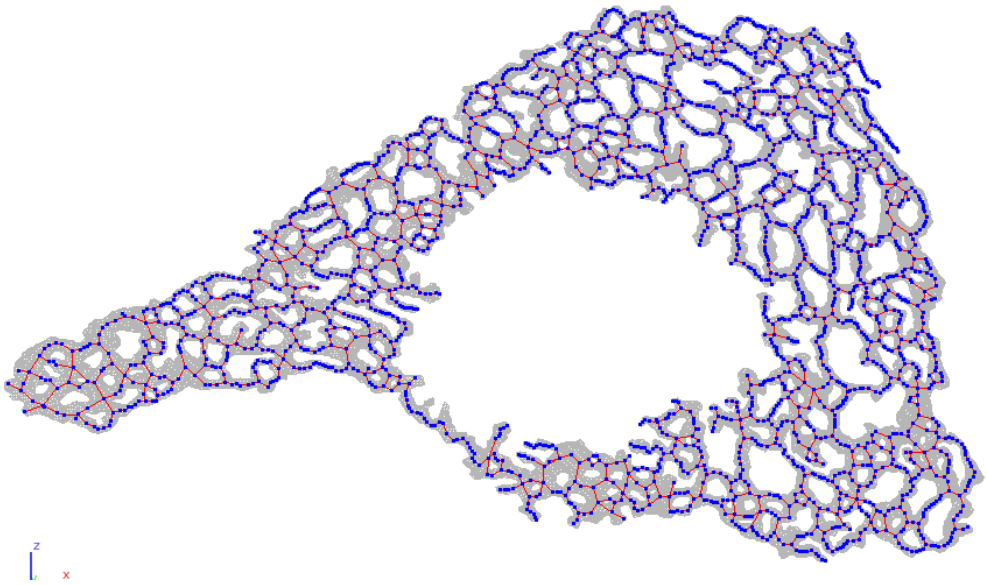


Figura A.12: *Skeleton* de la malla retículot00.

A.3. Trabajo Futuro

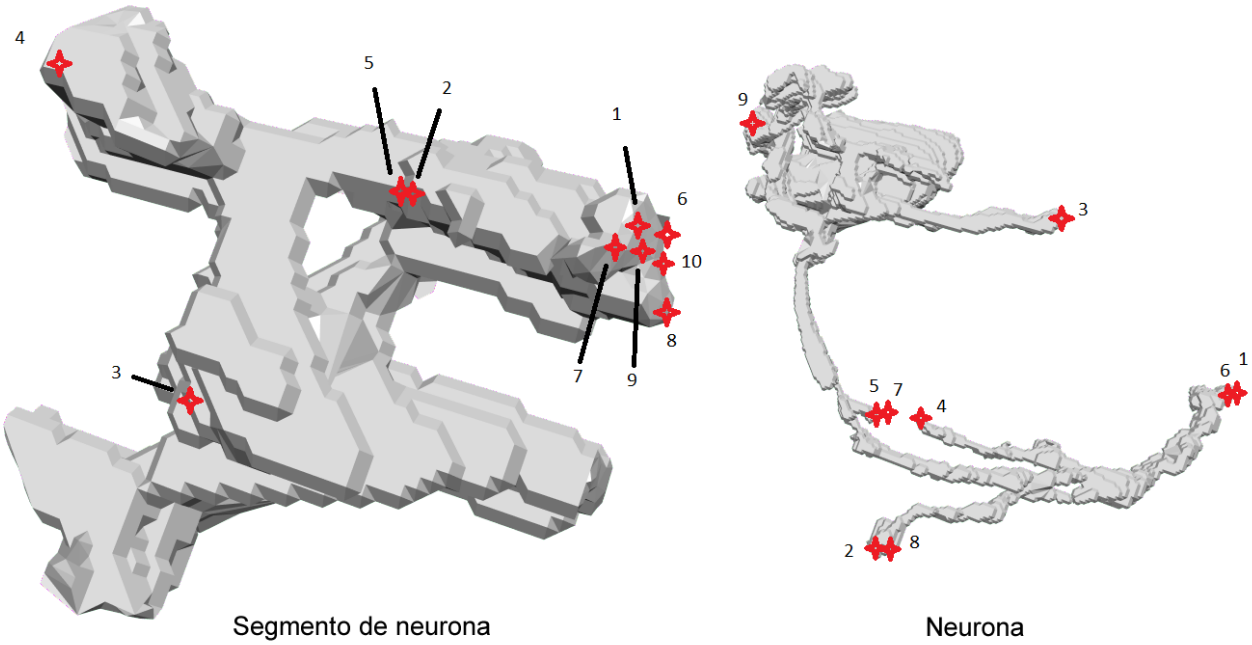


Figura A.13: Orden en que se realizan los colapsos. Colapsos (marcados en rojo) consecutivos en un segmento de neurona y en una neurona.

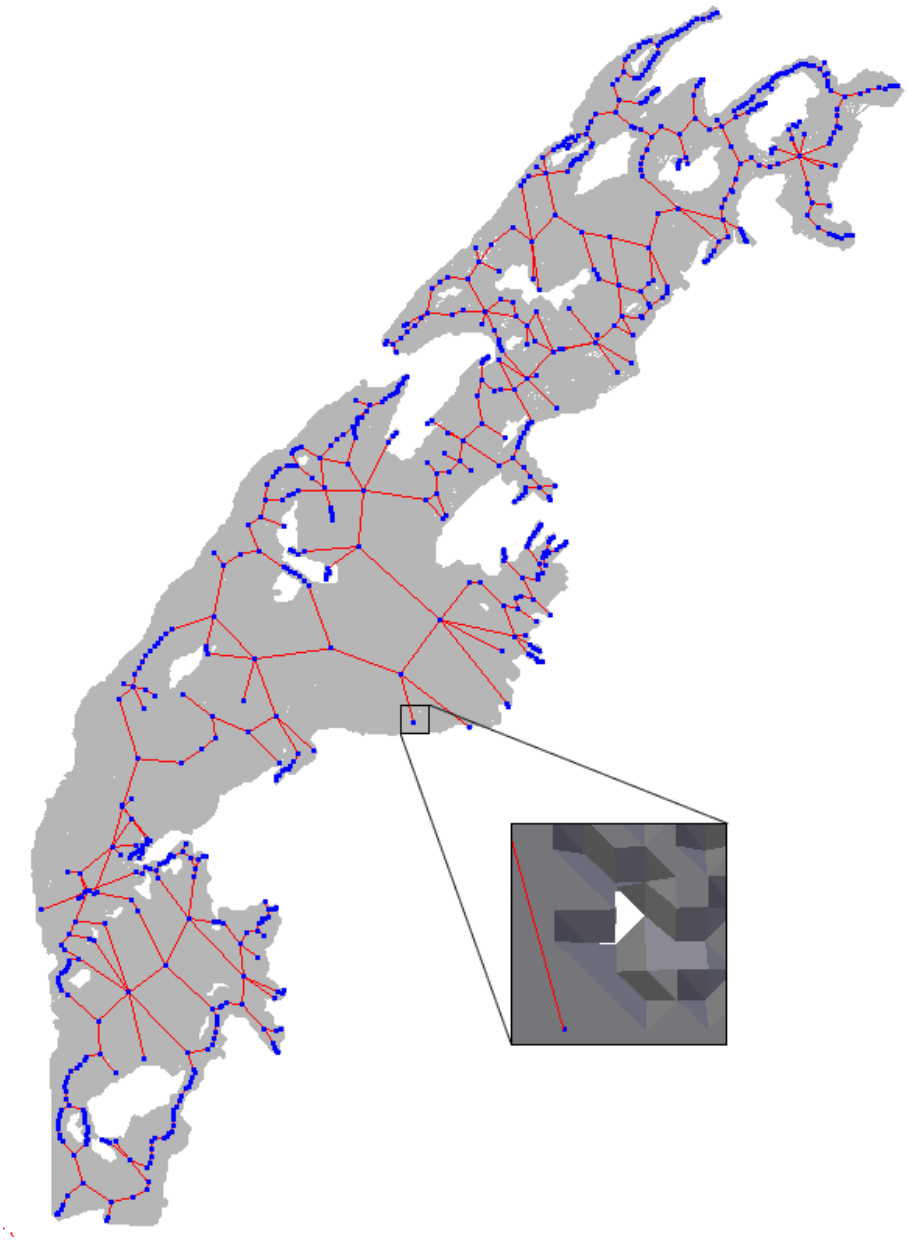


Figura A.14: Pérdida de un túnel en la malla cresta neuronal luego de generar el *skeleton*. Este resultado es obtenido con la optimización serial, pero se mantiene el mismo problema tanto para la implementación 2011 como para la paralelización.

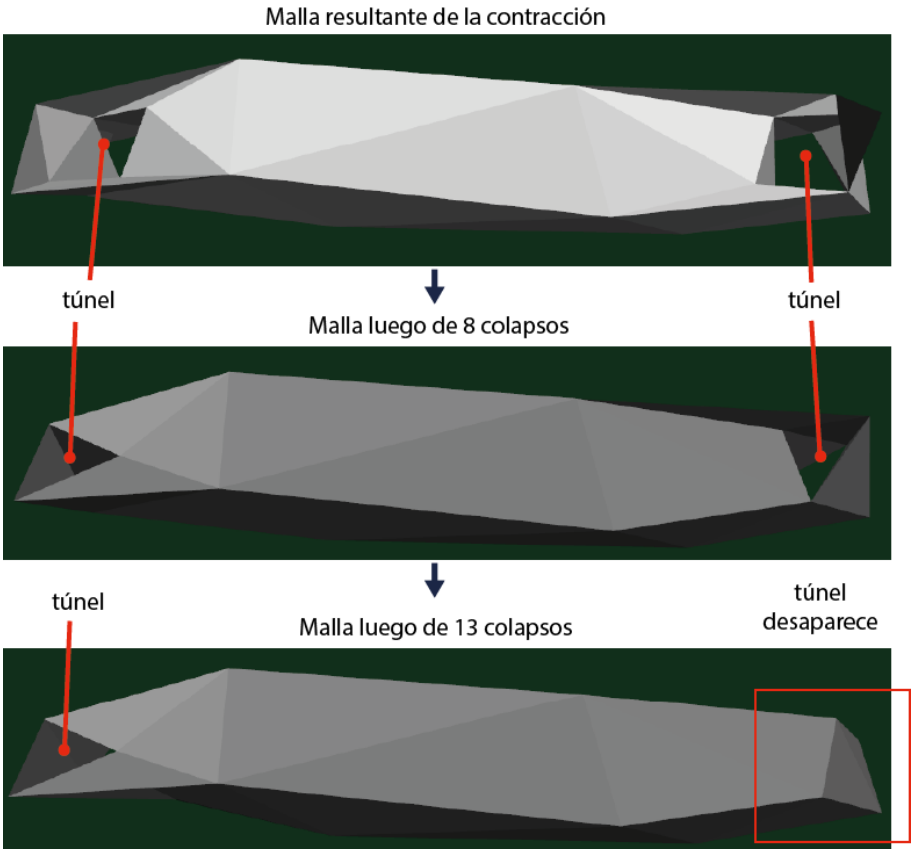


Figura A.15: Distintas iteraciones en el colapso de una malla con túneles. Luego de una cantidad de colapsos, uno de los túneles desaparece. Prueba realizada con la implementación *Halfedge*.