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

ALGORITMO PARALELO PARA VACÍOS POLIGONALES EN TRIANGULACIONES DE DELAUNAY

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

JOSÉ BENJAMÍN OJEDA CHONG

PROFESORA GUÍA:
NANCY HITSCHFELD KAHLER

MIEMBROS DE LA COMISIÓN:
NANCY HITSCHFELD KAHLER
BENJAMÍN BUSTOS CÁRDENAS
JOHAN FABRY

SANTIAGO DE CHILE
2016

RESUMEN DE LA MEMORIA PARA OPTAR
AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN
POR: JOSÉ BENJAMÍN OJEDA CHONG
FECHA: 2016
PROF. GUÍA: NANCY HITSCHFELD KAHLER

ALGORITMO PARALELO PARA VACÍOS POLIGONALES EN TRIANGULACIONES DE DELAUNAY

El universo visto a gran escala presenta estructuras llamadas murallas cósmicas y vacíos cósmicos. Las murallas poseen una alta densidad galáctica, mientras que los vacíos presentan una notable baja densidad.

En Astronomía son objeto de estudio y para el efecto es necesario analizar grandes volúmenes de datos, lo cual no es factible hacerlo manualmente, siendo necesarios algoritmos que identifiquen automáticamente ambos tipos de zonas cósmicas. Actualmente existen muy pocos algoritmos diseñados para esa tarea.

Uno de los más recientes fue presentado en el año 2014 por Hervías et al.[3] y permite determinar bajas densidades en "rebanadas" bidimensionales de datos volumétricos, pudiéndose extender naturalmente la forma en que trabaja al caso tridimensional. Recibe como entrada un conjunto de puntos, cada uno siendo la abstracción geométrica de una galaxia.

El algoritmo tiene dos fases: triangulación y clasificación de triángulos. Se basa en la identificación de aristas largas en la triangulación de Delaunay asociada a los n puntos de entrada, para determinar bajas densidades galácticas. En su segunda fase determina en forma secuencial los triángulos que pertenecen a vacíos con una complejidad $O(n \log(n))$.

A mediados de este año, Alonso[9] presenta una extensión del algoritmo, agregándole una tercera fase de fusión de vacíos adyacentes. Logra una implementación que clasifica con complejidad $O(n \log(n))$ en forma experimental.

En esta memoria se hace un razonamiento profundo sobre las propiedades de los conjuntos de triángulos que son identificados como vacíos en una triangulación. La investigación teórica realizada permite presentar tres nuevas formas de clasificación que proporcionan los mismos resultados que la original: una $O(n)$ secuencial, una $O(n)$ paralela y una $O(\log(n)^2)$ paralela.

Se implementan las $O(n)$ desarrolladas, resultando ser experimentalmente más eficientes que la clasificación de la implementación de Alonso. En particular, la $O(n)$ paralela exhibe un mejor desempeño que la $O(n)$ secuencial. Con lo anterior se logra el objetivo de esta memoria.

La $O(\log(n)^2)$ paralela se deja propuesta para su implementación a futuro y tal vez conectarse a una triangulación de Delaunay paralela.

A mi familia y amistades.

Tabla de contenido

1. Introducción **1**
- 1.1. Vacíos cósmicos . . . 1
- 1.2. Algoritmos de identificación . . . 2
- 1.3. Objetivo y resultados . . . 3
- 1.4. Contenido de esta memoria . . . 4

2. Marco teórico y herramientas **5**
- 2.1. Triangulaciones de Delaunay . . . 5
  - 2.1.1. Grafo de adyacencia de triangulación de Delaunay . . . 7
  - 2.1.2. qdelaunay . . . 9
- 2.2. Conceptos y técnicas en algoritmos paralelos . . . 10
  - 2.2.1. Modelo PRAM . . . 10
  - 2.2.2. Técnica de Pointer jumping . . . 11
  - 2.2.3. Técnica de Computación de prefijos . . . 15
  - 2.2.4. Técnica de Computación segmentada de prefijos . . . 16
  - 2.2.5. Ordenamiento paralelo . . . 17
- 2.3. Implementación de algoritmos paralelos . . . 18
  - 2.3.1. CPU y GPU . . . 18
  - 2.3.2. OpenCL . . . 19

3. Clasificación en $O(n \log(n))$ **21**
- 3.1. Algoritmo de Hervías et al. . . . 21
  - 3.1.1. Complejidad temporal . . . 24
- 3.2. DELFIN extendido . . . 24

4. Caracterización de vacíos poligonales **26**
- 4.1. Nomenclatura . . . 26
  - 4.1.1. Zonas, vacíos y murallas . . . 26
  - 4.1.2. Aristas de borde, internas y semilla . . . 27
  - 4.1.3. Tipos de aristas internas a la triangulación . . . 27
- 4.2. Sobre aristas máx-máx . . . 28
- 4.3. Sobre aristas nomáx-nomáx . . . 29
- 4.4. Sobre zonas . . . 31

5. Clasificación en $O(n)$ **33**
- 5.1. Algoritmos . . . 33

5.1.1. Secuencial 35
5.1.2. Paralelización 37
5.2. Implementación 39
5.3. Tests 40
5.4. Experimentación y resultados 41
5.5. Comentarios 44

6. Clasificación en $O(\log(n)^2)$ **45**
6.1. Algoritmo 45
6.1.1. Complejidad temporal 49

7. Conclusiones y trabajo futuro **51**
7.1. Conclusiones 51
7.2. Trabajo futuro 51

Bibliografía **53**

Anexos **54**
A. Diagrama de Voronoi 54
B. Comparación asintótica entre $\log(n)$, $\log(n)^2$ y n 54
B.1. Demostración de $\log(n)^2 \in o(n)$ 55
B.2. Demostración de $\log(n) \in O(\log(n)^2)$ 56
C. Ejemplo de herramienta de visualización SVG 56

Capítulo 1

Introducción

1.1. Vacíos cósmicos

Dentro de los diversos temas de investigación en Astronomía está el estudio de la estructura del universo a grandes escalas.

Las primeras publicaciones científicas relevantes respecto del asunto aparecieron a fines de los años 70, dando cuenta de la existencia de colosales volúmenes en que hay una ausencia notable de galaxias. Aquéllas estructuras con baja densidad galáctica son denominadas **vacíos cósmicos** y están separadas entre sí por filamentos de alta densidad galáctica, llamados **murallas cósmicas**.

Además de dar luces sobre cómo es el universo, ambos tipos de zonas cósmicas son de gran importancia porque su detección y descripción son elementos clave para develar los misterios de la energía oscura – algo hipotético que provocaría una expansión acelerada del cosmos – y el ajuste de los modelos de evolución cosmológica actuales.

Uno de los proyectos pioneros en intentar mapear a macroescala al universo fue el *CfA Redshift Survey*, a cargo del *Harvard-Smithsonian Center for Astrophysics*, que comenzó el año 1977 y completó su adquisición de datos en 1982. Una segunda toma de datos se realizó desde 1985 hasta el año 1995.

La segunda recolección permitió a fines de la década de los 80 el descubrimiento de la Gran muralla[1], una de las superestructuras del universo observable más grandes conocidas hasta el momento. Corresponde a un monumental filamento de alta densidad galáctica, rodeado por un volumen caracterizado por una evidente baja densidad.

En la actualidad está en curso el proyecto *Sloan Digital Sky Survey*, más conocido por sus siglas SDSS. Comenzó el año 2000 y actualmente se encuentra su cuarta fase de toma de datos, estimándose su término en el año 2020. Un ejemplo visual de los datos que ha tomado se aprecia en la figura 1.1, cada punto en ella correspondiendo a una galaxia.

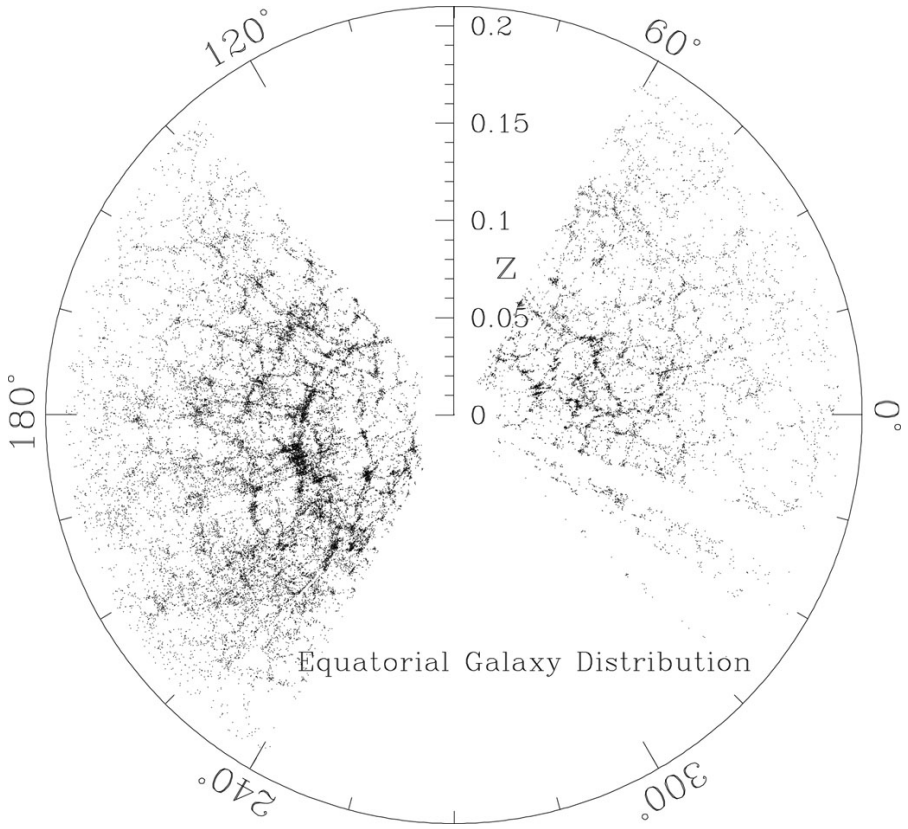


Figura 1.1: *Una "rebanada" del universo observado por el proyecto SDSS, en que aparecen cerca de 60.000 galaxias. Se distinguen vacíos cósmicos y murallas filamentosas.*

Los datos obtenidos deben ser procesados y analizados para poder obtener las zonas cósmicas que corresponden a vacíos y murallas. Dado el gigantesco volumen de información, es inviable realizar un análisis manual, siendo totalmente necesario emplear algoritmos que las identifiquen automáticamente.

1.2. Algoritmos de identificación

Actualmente no existe una definición exacta – o de consenso – de vacío cósmico, por lo que cada algoritmo propuesto para encontrarlos debe probarse experimentalmente para ver su tasa de aciertos.

Hasta ahora se han presentado algunas formas para identificar vacíos, la mayoría siendo susceptible de ser clasificada según en qué se basan para arrojar sus resultados:

- Características geométricas de zonas con ausencia de galaxias.
- Identificación de fugas cósmicas de materia.
- Empleo de densidades galácticas locales.

Dentro de los algoritmos de caracterización geométrica se encuentra ZOBOV[2] – acrónimo de *Zones bordering on voidness* – que utiliza un diagrama de Voronoi* a fin de estimar densidades. Éstas últimas son empleadas finalmente para determinar vacíos. Fue publicado por M. Neyrinck en el año 2008, haciéndose énfasis en que no necesita parámetros para hacer su trabajo.

Un algoritmo más reciente fue introducido en una publicación de Hervías et al.[3] en el año 2014. Recibe como entrada un conjunto de puntos, cada uno correspondiendo a la idealización geométrica de una galaxia. Encuentra vacíos en datos bidimensionales, lo que permite el análisis de "rebanadas" de datos volumétricos, y su comportamiento se puede extender naturalmente al tratamiento de puntos tridimensionales.

Aquél se basa en la identificación de aristas largas en la triangulación de Delaunay asociada al conjunto de n puntos de entrada, para hallar conjuntos de triángulos adyacentes cuyos vértices representan una baja densidad. Su determinación de vacíos sobre la triangulación ya calculada es secuencial y de complejidad computacional $O(n\log(n))$.

Tiene un parámetro libre, que es el área mínima para que un conjunto de triángulos adyacentes candidato a vacío se identifique como tal. Lo anterior le permite ofrecer mayor flexibilidad y versatilidad, dado que permite elegir las características de los conjuntos de triángulos requeridos.

Dada la novedad de este algoritmo para encontrar vacíos cósmicos, su fase de clasificación de triángulos – en la que se determinan los vacíos – aún no posee una versión paralela.

1.3. Objetivo y resultados

El objetivo de esta memoria fue precisamente proponer e implementar una fase de clasificación paralela que entregue los mismos resultados que la secuencial original y que lograra un mejor desempeño que la implementación actual, que se presenta en la sección 3.2.

Para enfrentar el desafío se decidió analizar a fondo cómo funciona la clasificación de Hervías et al. y de esa forma caracterizar desde un punto de vista lógico y formal los conjuntos de triángulos que entrega como resultado final.

Con el conocimiento adquirido se pudieron concebir tres nuevas formas de clasificar: una secuencial y dos paralelas, que aprovechan las características encontradas para lograr complejidades computacionales sustancialmente inferiores a la $O(n\log(n))$ original.

*Véase anexo A para definición.

1.4. Contenido de esta memoria

En el capítulo 2 se entregan los rudimentos necesarios para comprender el resto del texto, a saber: triangulaciones de Delaunay, conceptos y técnicas para algoritmos paralelos e implementación de aquéllos.

Se introduce el algoritmo de Hervías et al. en el capítulo 3, presentándose las características encontradas de los conjuntos de triángulos que arroja en el capítulo 4.

Dentro del capítulo 5 se proponen dos nuevas fases de clasificación que utilizan algunas de las características encontradas: una secuencial $O(n)$ y una paralela $O(n)$ creada a partir de la primera. Se muestran los resultados de las implementaciones de ambas, alcanzándose el objetivo de esta memoria con ellas.

No obstante lo anterior, una solución paralela de menor complejidad aún se presenta en el capítulo 6 y utiliza la totalidad de las características encontradas. Es una clasificación $O(\log(n)^2)$ que se deja propuesta teóricamente.

Capítulo 2

Marco teórico y herramientas

En este capítulo se entregarán conceptos y técnicas necesarias para comprender esta memoria.

En la sección 2.1 se introducirá a grandes rasgos qué es una triangulación de Delaunay, la que es base en el algoritmo de Hervías et al.. También en ella se presentará qdelaunay, que es un software que permite generarlas.

Más adelante en la sección 2.2, se describen conceptos y técnicas básicas para algoritmos paralelos; en la sección 2.3 se hace referencia a cómo implementarlos. Ambas secciones son necesarias para entender las clasificaciones paralelas desarrolladas.

2.1. Triangulaciones de Delaunay

Una triangulación de un conjunto de puntos $P \subseteq \mathbb{R}^2$ es la descomposición de su envoltura convexa en un conjunto de triángulos, de modo sólo pueden intersecarse en sus aristas y todo $p \in P$ es vértice de al menos uno de ellos.

No existe una triangulación única para un mismo conjunto de puntos. Según la figura 2.1, dado el conjunto de puntos mostrado, hay al menos dos triangulaciones posibles.

Una de las formas más conocidas de triangular conjuntos de puntos es mediante una **triangulación de Delaunay**. Tiene la propiedad de que para cada trío de puntos pertenecientes a un triángulo, la circunferencia circunscrita a él no tiene en su interior a ningún otro punto.

Lo propiedad se denomina **condición de Delaunay** y permite asegurar que los ángulos interiores de los triángulos sean lo mayor posible, maximizando el ángulo más cerrado de la triangulación y así evitándose – cuando es posible – la formación de triángulos muy delgados.

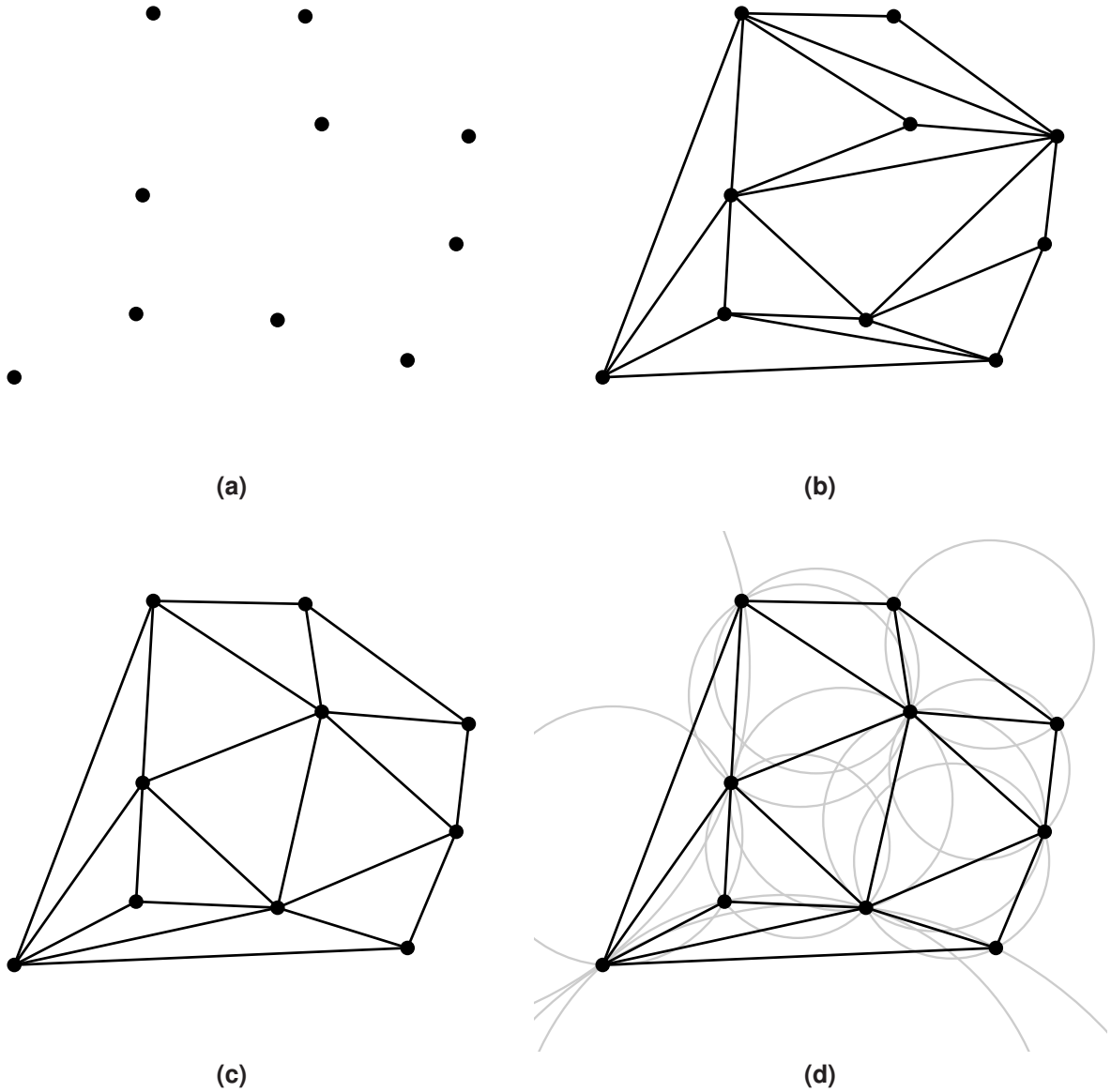


Figura 2.1: *Ejemplos de triangulaciones.* ***(a)*** *Un conjunto de puntos $P \subseteq \mathbb{R}^2$ de ejemplo.* ***(b)*** *Una triangulación para P. Se notan ángulos muy cerrados.* ***(c)*** *Otra triangulación para P. Se trata de su* ***triangulación de Delaunay*** *y muestra ángulos menos agudos.* ***(d)*** *La misma triangulación anterior, pero con las circunferencias circunscritas a cada triángulo de ella. Ninguna tiene a un punto de P dentro de sí.*

Un ejemplo de este tipo de triangulación se exhibe en la figura 2.1c, a la vez que la característica mencionada previamente se muestra en la figura 2.1d.

Existen varias aplicaciones para este tipo de triangulación, tales como: modelado de superficies en aplicaciones gráficas de tres dimensiones, interpolación de funciones y discretización para encontrar soluciones aproximadas a ecuaciones diferenciales parciales, entre otras.

El proceso de triangulación genera triángulos y aristas. Con n siendo el número de puntos a tratar, la cantidad de triángulos que resultan es $O(n)$.

2.1.1. Grafo de adyacencia de triangulación de Delaunay

Una forma de representar de manera abstracta a una triangulación de Delaunay T es a través de un grafo que refleja las vecindades de los triángulos que la componen. Es decir, mediante un grafo de adyacencia $G = (V, E)$ en que por cada triángulo $t \in T$ hay un nodo* v en el conjunto V.

Las relaciones de adyacencia quedan descritas en el conjunto de arcos E, como sigue: considérense dos triángulos $t, u \in T$ y sus correspondientes nodos asociados $v, w \in V$ en G. Existirá el arco dirigido desde v a w, denotado por $(v, w) \in E$, sí y sólo si t es adyacente a u en la triangulación. Como la relación binaria de adyacencia es simétrica, si $(v, w) \in E$ entonces también $(w, v) \in E$.

En el ejemplo mostrado en la figura 2.2, el triángulo 1 es adyacente al 2, por lo que en el G asociado aparecen efectivamente arcos entre los nodos correspondientes. Pero no hay conexión directa entre el nodo del triángulo 1 y los asociados a los triángulos 5 y 3, porque el primero no es adyacente a los últimos dos.

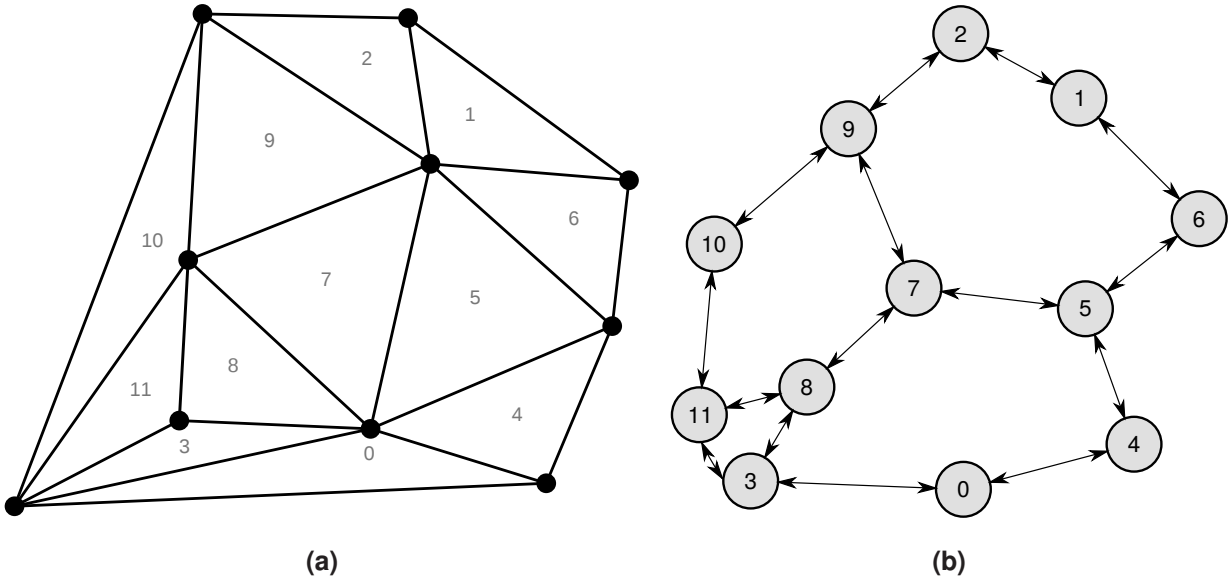


Figura 2.2: ***(a)*** *La misma triangulación de Delaunay de la figura 2.1, ahora con sus triángulos etiquetados.* ***(b)*** *El grafo de adyacencia asociado. Dada la simetría de la relación, se pueden usar arcos bidireccionales para simplificarlo.*

Por otra parte, considérese el concepto de grado de salida de un nodo $v \in V$, denotado por $g^+(v)$ y el de grado de entrada denotado por $g^-(v)$. En grafos para este tipo de triangulación se cumple lo siguiente para ambos grados:

*Para prevenir confusión, se dirá que los triángulos tienen aristas y vértices, mientras que los grafos tienen arcos y nodos.

Lema 2.1. *Sea $G = (V, E)$ un grafo de adyacencia de una triangulación de Delaunay. Entonces para todo $v \in V$, $g^+(v) \leq 3$ y $g^-(v) \leq 3$.*

Demostración. En forma directa.

En una triangulación de Delaunay no hay más de tres triángulos adyacentes a uno dado, por lo que cada nodo del grafo de adyacencia asociado no puede tener más de tres arcos dirigidos que emerjan de él. Por similar argumentación, tampoco puede tener más de tres arcos entrantes. ■

De lo anterior se infiere que:

Corolario 2.2. *Sea $G = (V, E)$ un grafo de adyacencia de una triangulación de Delaunay. Entonces $|E| \in O(|V|)$.*

Demostración. En forma directa.

Por el lema 2.1 se tiene que por cada nodo $v \in V$ no hay más de 6 arcos dirigidos que lo involucran: no hay más de 3 que salgan de él y no hay más de 3 que lleguen a él. Así se tiene la relación $|E| \leq 6|V|$, por lo que $|E| \in O(|V|)$. ■

Recorrido del grafo de adyacencia

Los grafos en general pueden ser recorridos en varias formas, estando entre ellas la **búsqueda en profundidad**. También se conoce como **DFS**[*] y desde un nodo dado visita sus nodos hijos antes que sus nodos hermanos. Se detalla su forma de exploración en el algoritmo 2.1.

```
DFS(G=(V,E) : Grafo):
  Para cada nodo v ∈ V:
    Si v no ha sido visitado:
      visitaDFS(G, v)
```

Algoritmo 2.1: *Algoritmo de búsqueda en profundidad en un grafo $G = (V, E)$. La función auxiliar `visitaDFS` se define en el algoritmo 2.2.*

```
visitaDFS(G=(V,E) : Grafo, v : Nodo a visitar):
  Si v ha sido visitado:
    Retornar
  Sino:
    Marcar a v como visitado
    Por cada nodo u en la vecindad de v:
      visitaDFS(G, u)
```

Algoritmo 2.2: *Función de visita necesaria para `DFS` del algoritmo 2.1. Recibe un grafo $G = (V, E)$ y el nodo $v \in V$ por el cual debe comenzar.*

[*] Siglas de *Depth-first search*.

Como se puede apreciar, el algoritmo de exploración presentado es bastante simple y para un grafo general $G = (V, E)$ la complejidad es de $O(|V| + |E|)$[4].

En el caso de un grafo completo $K = (V, E)$ de tamaño $|V|$ se tiene $|E| = \binom{|V|}{2}$, por lo que la complejidad de un DFS ahí sería $O(|V|^2)$, mostrándose que no siempre es lineal en el número de nodos y depende del tipo de grafo. Para las triangulaciones de Delaunay afortunadamente se tienen los siguientes resultados:

Lema 2.3. *Un DFS en un grafo de adyacencia $G = (V, E)$ de una triangulación de Delaunay, tiene una complejidad de $O(|V|)$.*

Demostración. En forma directa.

En el caso de las triangulaciones de Delaunay, utilizando el corolario 2.2 se tiene que $|E| \in O(|V|)$, por lo que en tal grafo de adyacencia asociado la complejidad de un DFS ahí es $O(|V| + |E|) = O(|V| + O(|V|)) = O(|V|)$. ■

Corolario 2.4. *Un DFS en un grafo de adyacencia $G = (V, E)$ de una triangulación de Delaunay para n puntos, tiene una complejidad de $O(n)$.*

Demostración. En forma directa.

Tomando el lema 2.3, el DFS en G tiene complejidad $O(|V|)$. Como cada $v \in V$ representa a un triángulo de la triangulación y hay $O(n)$ de ellos, $|V| \in O(n)$. Se concluye que la complejidad en cuestión es $O(n)$. ■

El DFS no sólo es lineal en el número de nodos del grafo de adyacencia, sino que también en el número de puntos que dan origen a la triangulación.

2.1.2. qdelaunay

Existen varios paquetes de software que permiten generar triangulaciones de Delaunay. Uno de ellos es **qhull**[5], que ofrece adicionalmente calcular envolventes convexas y diagramas de Voronoi. Está escrito en C++, es de código abierto y se encuentra disponible en la web*.

Uno de los programas que trae se llama **qdelaunay**. Dado un conjunto de puntos leídos desde su entrada estándar, arroja en su salida estándar los datos de la triangulación generada. Entregándole los parámetros apropiados, es posible obtener para cada triángulo: los puntos que lo conforman, los triángulos adyacentes y su área.

Según la documentación de qhull, la triangulación de Delaunay es calculada con una complejidad temporal esperada de $O(n \log(n))$, dado un conjunto de n puntos.

*http://www.qhull.org

2.2. Conceptos y técnicas en algoritmos paralelos

2.2.1. Modelo PRAM

El modelo PRAM* es uno de los modelos teóricos más populares para presentar algoritmos paralelos. Formas para trabajar con arreglos, listas, árboles y grafos, pueden ser fácilmente descritas en él. A pesar de que ignora muchos aspectos importantes de las máquinas paralelas del mundo real, las características esenciales de los algoritmos presentados, trascienden el contexto en que fueron planteados.

La máquina PRAM idealizada posee p procesadores $P_1, \ldots, P_p$ que tienen una memoria global compartida por todos, tal como se esquematiza en la figura 2.3.

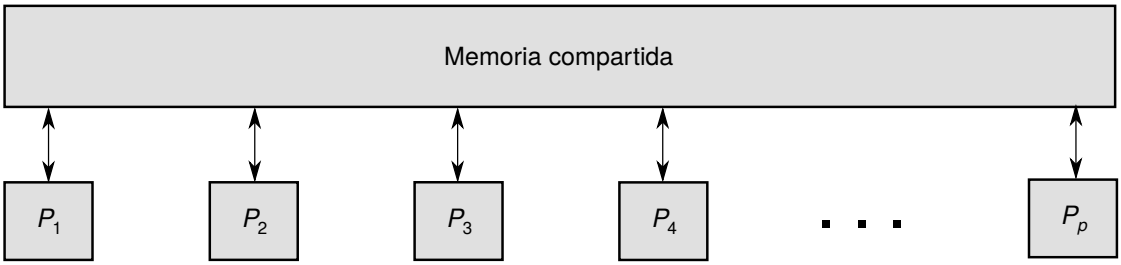


Figura 2.3: *Esquema de la arquitectura básica de una máquina PRAM: hay p procesadores, cada uno de los cuales tiene acceso a una memoria compartida por todos ellos.*

Todos los procesadores tienen la posibilidad de leerla o escribirla en forma simultánea. También pueden realizar operaciones matemáticas en paralelo.

En relación a la tipología de algoritmos en este modelo, uno de **lectura concurrente** es un algoritmo PRAM en cuya ejecución al menos una celda de memoria es leída en forma simultánea por más de un procesador. Por el contrario, uno de **lectura exclusiva** es aquél en que no hay celda de memoria leída por varios procesadores a la vez.

Una clasificación análoga se da respecto de si hay escrituras simultáneas en una misma celda de memoria, teniéndose algoritmos PRAM de **escritura concurrente** y de **escritura exclusiva**.

Los tipos lecturas y escrituras dan origen a cuatro categorías de algoritmos PRAM:

- **EREW**: siglas de *Exclusive read and exclusive write*, que es un algoritmo que hace lecturas y escrituras exclusivas.
- **CREW**: siglas de *Concurrent read and exclusive write*, que hace lecturas concurrentes y escrituras exclusivas.
- **ERCW**: siglas de *Exclusive read and concurrent write*, que realiza lecturas exclusivas y escrituras concurrentes.
- **CRCW**: siglas de *Concurrent read and concurrent write*, que realiza lecturas y escrituras concurrentes.

*Siglas de *Parallel random-access machine*.

Cabe destacar que la complejidad de la resolución de un problema en paralelo puede depender de la disponibilidad o no de concurrencia.

2.2.2. Técnica de Pointer jumping

Esta técnica es fundamental en algoritmos paralelos y consiste simplemente en hacer "saltar" los punteros a siguiente de cada nodo de una lista o árbol. Tiene diversas aplicaciones, como se verá a continuación.

Para esta técnica y sus subsecuentes aplicaciones, supóngase que se tienen n nodos y n procesadores en una PRAM, con cada procesador haciéndose cargo de un nodo.

En listas

La primera aplicación de *pointer jumping* es la comunicación a todos los nodos de una lista el valor que tiene el nodo cola de la misma.

Una forma eficiente de hacerlo se describe en el algoritmo 2.3. Se explica después cómo procede en detalle y el por qué del número de pasos paralelos que realiza.

```
Leer arreglo x de valores de nodos
Leer arreglo proximo de punteros próximos de nodos
Repetir ⌈log2(n)⌉ veces:
  Para cada procesador i, en paralelo:
    Si proximo[i] ≠ Null:
      Si proximo[proximo[i]] = Null:
        x[i] ← x[proximo[i]]
      proximo[i] ← proximo[proximo[i]]
```

Algoritmo 2.3: *Aplicación de pointer jumping a una lista enlazada, la que se representa con dos arreglos:* `x` *que contiene los valores de sus nodos y* `proximo` *que contiene los punteros a siguiente. El nodo de valor* `x[i]` *tiene por siguiente a* `proximo[i]`*. El nodo cola de la lista original se caracteriza por tener un puntero a siguiente nulo.*

En cada paso paralelo, cada procesador ve si el nodo siguiente al suyo, posee un puntero `proximo` nulo. Si es así, debe ser el caso que efectivamente posee el valor que se desea comunicar, copiándolo a sí mismo y colocando su propio puntero `proximo` en nulo. Si encuentra que su siguiente no posee un puntero `proximo` nulo, entonces simplemente lo salta a la espera de que en el siguiente paso en paralelo sí se encuentre con uno que cumpla con eso.

Un esquema paso a paso de cómo va progresando la forma de trabajo descrita, se ofrece en la figura 2.4.

Así, luego de $\lceil \log_2(n) \rceil$ iteraciones en paralelo, se tiene que los n nodos ya poseen el valor que tiene el nodo final de la lista.

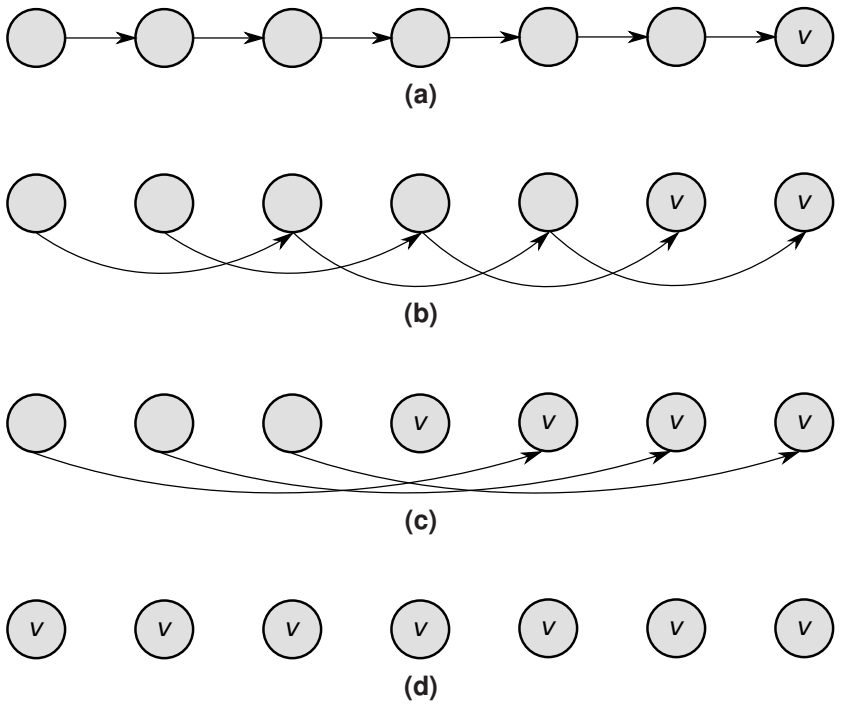


Figura 2.4: *Ejemplo de pointer jumping en una lista de siete elementos.* ***(a)*** *Lista original. El último nodo tiene el valor v a comunicar y tiene un* `proximo` *nulo.* ***(b)*** *Resultado del primer saltado. Se forman dos listas independientes y dos nodos tienen el valor* `v`*.* ***(c)*** *Después del segundo saltado, se generan cuatro listas independientes y cuatro ya poseen el valor* `v`*.* ***(d)*** *Saltado final. Hay siete listas independientes, todas con un sólo nodo, y el valor v se propagó a todos los nodos. Se requirieron no más de* $\lceil \log_2(7) \rceil = 3$ *saltados.*

La observación clave para justificar lo anterior, es que cada paso de *pointer jumping* transforma cada lista en dos listas intercaladas: una constituida por nodos en las posiciones pares y otra por nodos en posiciones impares. Luego, cada saltado dobla el número de listas y reduce a la mitad sus longitudes. Así, luego de $\lceil \log_2(n) \rceil$ pasos[6] en paralelo, toda lista pasa a tener sólo un objeto.

En esta aplicación de *pointer jumping* se tiene un algoritmo EREW, pues sólo hay lecturas y escrituras exclusivas. Es $O(\log(n))$.

Nótese que el proceso destruye la estructura original de la lista. Eso se puede remediar haciendo una copia de respaldo de los punteros a próximo, previamente.

En árboles

Otra aplicación de *pointer jumping* es comunicar a todos los nodos un puntero hacia la raíz de su árbol. La raíz asume el rol de “nodo cola” mencionado en la aplicación anterior y un puntero hacia sí misma es el “valor” a comunicar al resto.

Esta aplicación es particularmente útil cuando la raíz de un árbol almacena datos necesarios para sus descendientes. Luego del saltado de punteros, cada vez que la raíz tenga una nueva información disponible, simplemente basta con que los nodos lean concurren-

temente de ella para obtenerla.

Se utiliza exactamente el mismo algoritmo 2.3 y se ilustra en la figura 2.5 cómo progresa en un árbol.

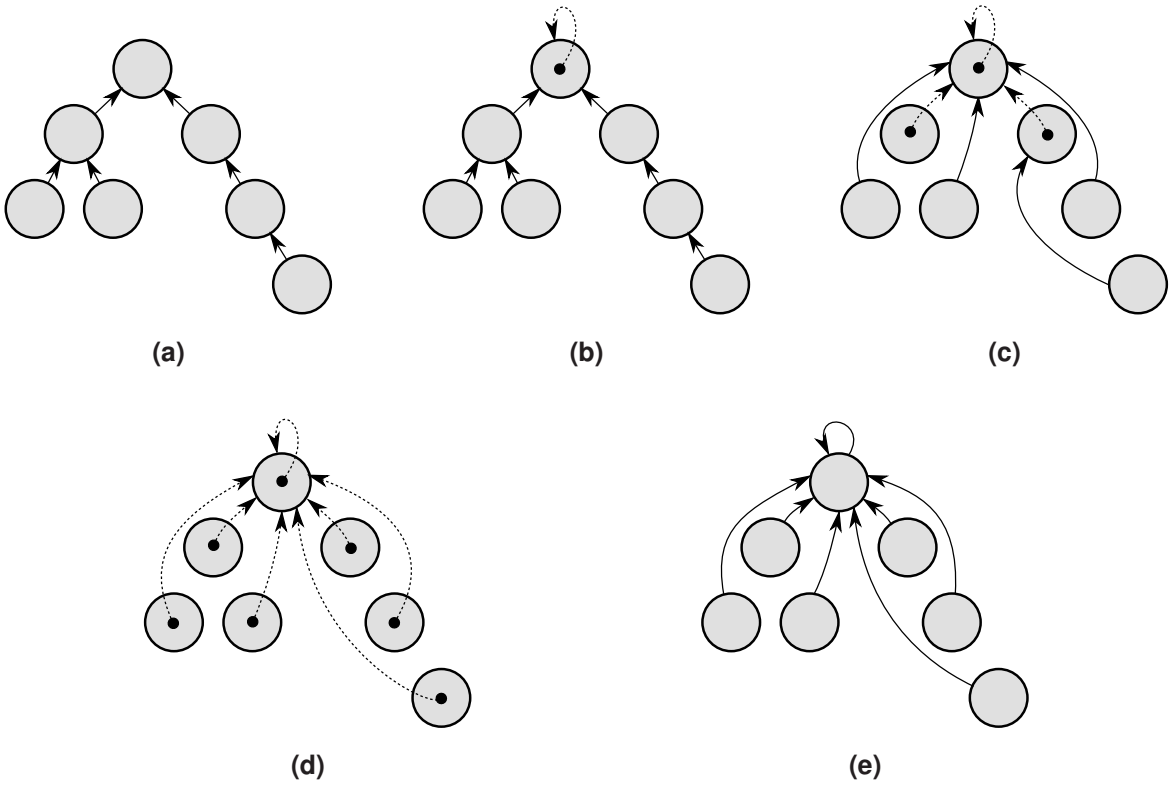


Figura 2.5: ***(a)*** *Árbol original.* ***(b)*** *En su raíz se almacena un puntero hacia sí misma, representado con una curva segmentada.* ***(c)*** *Resultado del primer paso de saltado punteros.* ***(d)*** *Resultado del segundo paso. El puntero a la raíz ya se propagó completamente a todo el árbol. Se hará un paso más para llegar a los* $\lceil \log_2(7) \rceil = 3$ *pasos, quedando todo exactamente igual.* ***(e)*** *Finalmente cada nodo copia el valor que obtuvo a su puntero a próximo, consiguiéndose así la conexión directa de todos los nodos a la raíz.*

En general no se sabe *a priori* la altura del árbol, en el peor caso teniéndose uno que corresponde a una lista enlazada. Tomando aquéllo en consideración, si se hacen $\lceil \log_2(n) \rceil$ saltados se asegura que todos los nodos del árbol obtendrán el puntero hacia a la raíz.

En este caso ocurren lecturas concurrentes, puesto que durante el proceso casi siempre habrán varios nodos apuntando a un padre en común que ya tiene la información que necesitan, por lo que deberán leer de él concurrentemente. Esta aplicación de *pointer jumping* resulta en un algoritmo CREW y también es $O(\log(n))$.

También se puede emplear lo mismo para bosques de árboles. Se hacen $\lceil \log_2(n) \rceil$ pasos de saltado porque no se sabe de antemano la altura del árbol más alto. Se consigue que todos los nodos de cada árbol original tengan información de la raíz que les corresponde.

Luego del *pointer jumping* todos los punteros `proximo` quedan en nulo. Se pueden copiar los valores obtenidos a los punteros `proximo` para dejar finalmente los nodos conectados a sus raíces, tal como se ilustra para un solo árbol en las figuras 2.5d y 2.5e.

En grafos con grados de salida $\Theta(1)$ y árboles

Se puede lograr la misma conexión de los nodos de árboles con sus raíces, en grafos que contienen además otros tipos de componentes conexas. Téngase un grafo cuyos nodos tienen grados de salida acotados por una constante $m \in \Theta(1)$. Un ejemplo es el grafo de adyacencia de una triangulación de Delaunay, cuyos nodos no tienen más de tres arcos dirigidos emergentes (punteros `proximo`), por lo que $m = 3$.

En este contexto, los nodos pertenecientes a árboles se caracterizarán por tener a lo más uno de sus m punteros `proximo` distinto de nulo: en nodos raíz todos serán nulos y en los interiores sólo uno será distinto de nulo. Utilizando la propiedad descrita para extender el comportamiento del *pointer jumping* original al tipo de grafo propuesto, se puede asegurar que se logrará el efecto deseado en aquellas componentes que son árboles, según se presenta en el algoritmo 2.4 y su análisis subsecuente.

```
Leer arreglo x de valores de nodos
Leer arreglo proximo de punteros próximos de nodos
Repetir ⌈log2(n)⌉ veces:
  Para cada procesador i, en paralelo:
    Si existe un único j ∈ {1, ..., m} tal que proximo[i][j] ≠ Null:
      Si existe un único k ∈ {1, ..., m} tal que proximo[proximo[i][j]][k] ≠
         Null:
        proximo[i][j] ← proximo[proximo[i][j]][k]
      Sino, si para todo k ∈ {1, ..., m} se tiene proximo[proximo[i][j]][k] =
         Null:
        x[i] ← x[proximo[i][j]]
        proximo[i][j] ← Null
Para cada procesador i, en paralelo:
  proximo[i][1] ← x[i]
  Para j ∈ {2, ..., m}:
    proximo[i][j] ← Null
```

Algoritmo 2.4: *Extensión del comportamiento del pointer jumping a un grafo con grados de salida* $\Theta(1)$ *y árboles. La celda* `proximo[i][j]` *indica el* j*-ésimo puntero a próximo del* i*-ésimo nodo del grafo. El arreglo* `x` *de valores ya viene preparado: en las celdas correspondientes a nodos raíz de árbol tienen punteros hacia ellas y para el resto de los nodos, valores nulos.*

Cada uno de los $\lceil \log_2(n) \rceil$ pasos paralelos de saltado es $\Theta(1)$, pues en ellos cada procesador consulta los m punteros `proximo` de su nodo a cargo y luego, de haber uno sólo distinto de nulo, revisa los m del único siguiente. Todo resulta en $\Theta(1)$, pues $m \in \Theta(1)$ y luego el total de pasos en $O(\log(n))$.

Esta forma de *pointer jumping* chequea si hay un único puntero `proximo` distinto de nulo, en cada nodo. De cumplirse lo anterior, examina aquél nodo siguiente y ve si es un nodo "cola" (es decir, si todos sus punteros `proximo` son nulos). Si no lo es y tiene un único puntero `proximo` distinto de nulo, entonces hay un salto. Si es uno cola, de manera análoga a la versión original, aquél ya debe tener el valor buscado; el procesador lo copia hacia sí y coloca el puntero correspondiente en nulo, lo que deja a todos los punteros `proximo` de su nodo en nulo, finalmente convirtiéndolo en uno cola.

Todo lo anterior asegura que el *pointer jumping* se hará correctamente en los árboles del

grafo, por la caracterización de los nodos realizada con antelación. Eventualmente también podrían haber algunos saltos de punteros en otros tipos de componentes conexas. Sin embargo, no se realizarán saltos en nodos que no puedan pertenecer a árboles, es decir, los que posean dos o más punteros `proximo` distintos de nulo.

Nuevamente se pueden dejar en las raíces de los árboles punteros hacia sí mismas como valores a comunicar. En el resto de los nodos, déjense valores nulos. Luego de los $\lceil \log_2(n) \rceil$ pasos de saltado, los punteros a las raíces ya se habrán propagado a sus correspondientes descendientes, pero la estructura del grafo original – dada por sus punteros `proximo` – habrá quedado parcial o totalmente destruida.

A fin de conectar finalmente los nodos de los árboles con sus raíces, se tiene la última fase paralela del algoritmo 2.4. En ella, se copia el valor que cada nodo ha obtenido hacia su primer puntero `proximo` y luego se dejan en nulo el resto de ellos. En las componentes que no son árboles, los valores de sus nodos son nulos, por lo que quedarán finalmente como nodos aislados. En las que sí son árboles, los descendientes de sus raíces lograrán conectarse como se deseaba, mediante sus primeros punteros `proximo`.

En la fase de conexión ya descrita, cada procesador hace $m \in \Theta(1)$ pasos en paralelo. Así, el proceso total de esta extensión del *pointer jumping* es CREW y mantiene su complejidad en $O(\log(n))$.

2.2.3. Técnica de Computación de prefijos

Supóngase el problema de generar a partir de una lista de n elementos $\langle x_1, x_2, ..., x_n \rangle$ y una operación binaria asociativa $\oplus$, una nueva lista $\langle x'_1, x'_2, ..., x'_n \rangle$, en que $x'_1 = x_1$ y para cada $i \in \{2, 3, ..., n\}$ se obtiene $x'_i = x'_{i-1} \oplus x_i = \bigoplus_{k=1}^{i} x_k$. Corresponde a la computación de prefijos de los elementos de la lista, teniéndose finalmente en el i-ésimo elemento un término parcial que lo considera a él y todos los valores originales que le precedían.

Una de las aplicaciones de esta computación es resolver eficientemente la suma de n elementos en paralelo. Al final el resultado buscado se encontrará en el nodo cola de la lista procesada.

Utilizando nuevamente la técnica del *pointer jumping* se puede resolver la computación en $\lceil \log_2(n) \rceil$ pasos, según se puede ver en el algoritmo 2.5.

```
Leer arreglo x de valores de nodos
Leer arreglo proximo de punteros próximos de nodos
Repetir ⌈log2(n)⌉ veces:
  Para cada procesador i, en paralelo:
    Si proximo[i] ≠ Null:
      x[proximo[i]] ← x[i] ⊕ x[proximo[i]]
      proximo[i] ← proximo[proximo[i]]
```

Algoritmo 2.5: *Algoritmo de computación de prefijos.*

Un ejemplo que ilustra este tipo de computación se exhibe en la figura 2.6: en cada paso en paralelo las piezas de información necesaria se propagan sistemáticamente hacia la cola de la lista.

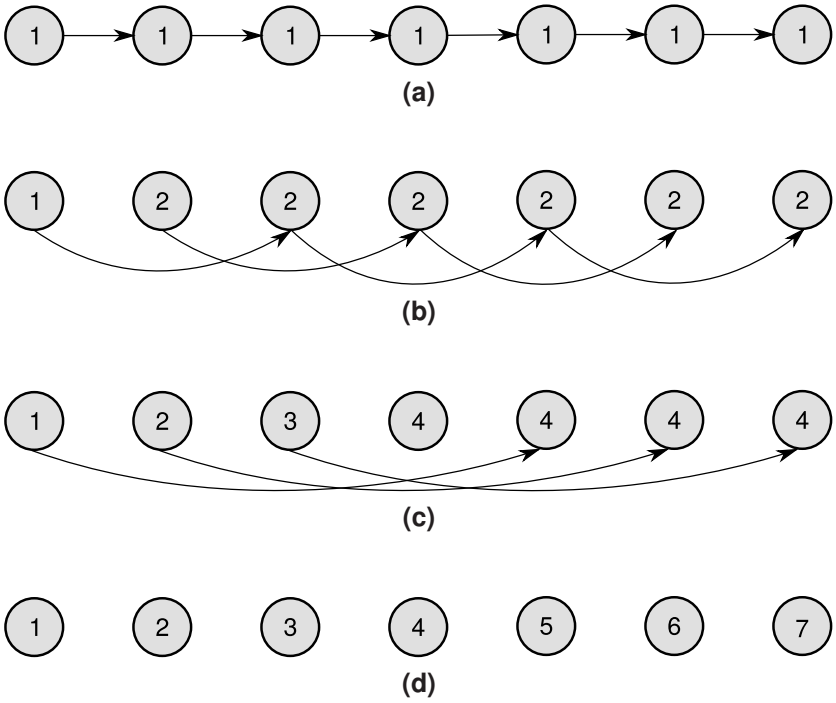


Figura 2.6: *Ejemplo de computación de prefijos en una lista de siete elementos, con la operación de suma común.* ***(a)*** *Lista inicial, con todos sus nodos teniendo por valor 1.* ***(b)*** *Listas resultantes del primer saltado de punteros.* ***(c)*** *Luego del segundo saltado de punteros, se nota que las sumas parciales de los valores se van "acumulando" hacia la derecha.* ***(d)*** *Tercer y último – pues* $\lceil \log_2(7) \rceil = 3$ *– saltado de punteros. En el* i*-ésimo nodo queda la suma de su valor original con los valores originales anteriores.*

Dado el número de pasos logarítmico – cuya justificación es la misma que para la primera aplicación de *pointer jumping* presentada – todo se resuelve con complejidad $O(\log(n))$ y en forma EREW.

2.2.4. Técnica de Computación segmentada de prefijos

La computación de prefijos se realiza sobre una lista enlazada, calculando los prefijos correspondientes a todos los valores de la lista, dada una operación asociativa $\oplus$.

A veces es necesario sólo calcular prefijos para grupos adyacentes de nodos de una lista enlazada. Esto es posible utilizando la misma forma de trabajo de la computación de prefijos normal, pero en este caso debe agregarse un campo extra a cada nodo, llámese s. Aquél será `1` si inicia un segmento y será `0` si no.

Considerando para los nodos el campo x para sus valores, a fin de lograr la computación segmentada se debe utilizar el siguiente operador especial $\tilde{\oplus}$, creado en base a $\oplus$:

$$(s, x) \,\tilde{\oplus}\, (s', x') = \begin{cases} (s, x \oplus x') & ; \text{si } s' = 0. \\ (1, x') & ; \text{si } s' = 1. \end{cases}$$

Claramente este operador es asociativo, por lo que puede usarse sin problemas con la computación de prefijos normal. Un ejemplo de cómo progresa el cálculo puede verse en la figura 2.7.

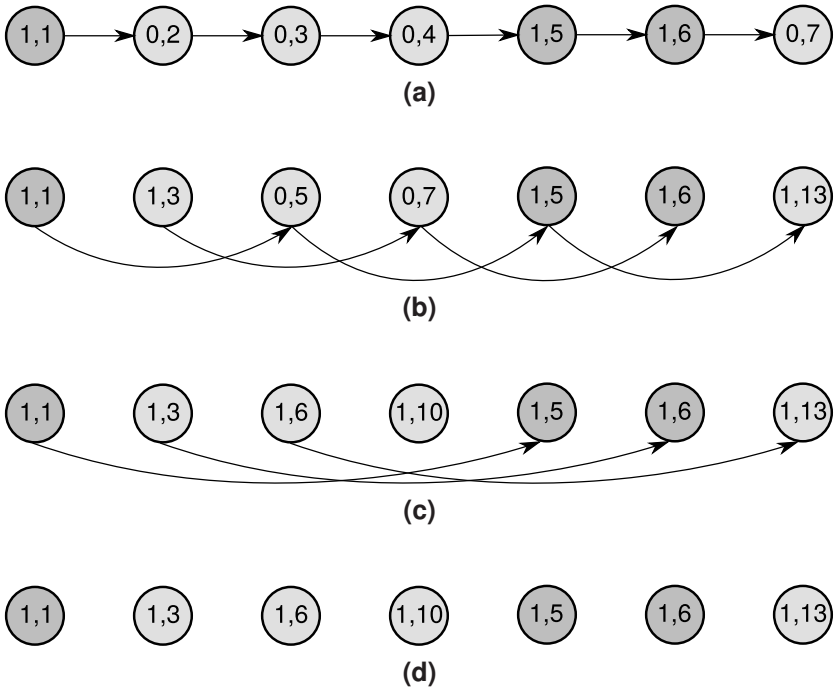


Figura 2.7: *Ejemplo de computación segmentada de prefijos con la operación de suma común.* ***(a)*** *Lista inicial, con el i-ésimo nodo teniendo el valor i. Hay tres segmentos: el primero de nodos con valores 1, 2, 3 y 4; el segundo con sólo el valor 5 y el tercero teniendo los valores 6 y 7. Los nodos gris oscuro indican los inicios de segmento.* ***(b)*** *Listas resultantes del primer saltado de punteros.* ***(c)*** *Luego del segundo saltado de punteros, se observa que las sumas parciales de los valores se van "acumulando" hacia la derecha, pero no afectan a los segmentos colindantes correspondientes.* ***(d)*** *Último saltado de punteros, considerando* $\lceil \log_2(7) \rceil = 3$*. El resultado de la computación deja en el último nodo de cada segmento la suma de todos los pertenecientes a él.*

Dado que la versión segmentada utiliza la computación normal, también necesita hacer $\lceil \log_2(n) \rceil$ pasos y es EREW. En el peor caso hay un único segmento, debiendo realizar obligadamente aquélla cantidad de pasos. Así la computación segmentada de prefijos resulta ser $O(\log(n))$.

2.2.5. Ordenamiento paralelo

Existen diversos algoritmos para ordenar en máquinas paralelas. El que se considerará en esta memoria es el ***Odd-even mergesort***, que fue desarrollado por K. E. Batcher en el año 1968.

Se basa en un algoritmo que mezcla dos mitades ordenadas de una secuencia, produciendo una completamente ordenada. En una máquina paralela de n procesadores, una secuencia de n elementos – con n debiendo ser una potencia de 2 – es ordenada en su totalidad con una complejidad de $O(\log(n)\log(n)) = O(\log(n)^2)$. No necesita de escrituras

concurrentes, así que puede considerarse CREW.

Este algoritmo fue popularizado por el segundo libro GPU Gems[7], como una forma razonablemente rápida de realizar ordenación en aplicaciones de procesamiento de gráficos.

2.3. Implementación de algoritmos paralelos

Respecto a la implementación de algoritmos paralelos, se necesita de un *hardware* apropiado que ofrezca paralelismo y una plataforma de programación que permita accesar y utilizar esa característica.

2.3.1. CPU y GPU

Los dos tipos de procesadores más comunes que ofrecen paralelismo son:

- **Unidad procesamiento central**: Conocida también como **CPU**, por las siglas del inglés *Central processing unit*.
- **Unidad de procesamiento gráfico**: También referida por las siglas **GPU**, del inglés *Graphics processing unit*.

Sólo hoy en día las CPU ofrecen múltiples núcleos de procesamiento, mientras que las GPU fueron desde sus inicios concebidas para tener varios. Ejemplos actuales de CPU son las **Intel Core i5** e **i7**, que ofrecen cuatro núcleos.

Las GPU tienen una memoria propia de distintos niveles y no siempre tienen acceso a la RAM de la máquina en que están, por lo que generalmente es necesario hacer transferencias desde la RAM hacia el dispositivo y viceversa para recuperar los resultados del cómputo.

En el caso de las CPU el manejo de memoria es menos complejo, porque de por sí tienen acceso directo a la RAM, por lo que no son necesarias transferencias de datos. La modalidad de acceso directo es comúnmente llamada **Zero-copy**.

Las CPU multinúcleo ofrecen hilos de ejecución: en el caso de las i5 e i7 están dotadas de SMT*. Ambas familias de CPU se comercializan haciéndose énfasis en que traen la tecnología *Hyperthreading*, que corresponde a la implementación de SMT por parte de Intel.

Los hilos de ejecución pueden entenderse como procesadores en el contexto del modelo PRAM, introducido en la sección 2.2. En adición, las CPU multinúcleo y GPU se pueden considerar como máquinas PRAM que admiten lecturas concurrentes pero no escrituras de ese tipo, pudiendo ejecutar directamente algoritmos de tipo CREW.

*Siglas de *Simultaneous multithreading*.

Por último, mencionar que generalmente los dispositivos de cómputo poseen grupos de hilos, llamados bloques.

2.3.2. OpenCL

Existen varios sistemas de programación para implementar algoritmos paralelos. Dos de los más conocidos hoy en día son CUDA* y **OpenCL**†.

El primero es un conjunto de herramientas desarrolladas por NVIDIA que ofrece una variación del lenguaje de programación C para implementar aplicaciones paralelas. Funciona específicamente para GPUs producidas por aquél fabricante, haciendo que el código programado no sea portable a dispositivos de otro tipo o marca.

Lo anterior sí es posible en OpenCL, que permite hacer computación en sistemas heterogéneos que disponen de distintos dispositivos de cómputo paralelo. Es un estándar abierto de programación paralela, desarrollado y mantenido actualmente por el consorcio Khronos Group.

La portabilidad que ofrece OpenCL no sólo se limita a la marca y tipo de *hardware* (GPU, CPU multinúcleo, etc.) sino que se extiende también a tipos de memoria y sistemas operativos. Para cada dispositivo de cómputo a usar se debe instalar el cargador de ICD‡ apropiado.

En relación a la programación, OpenCL especifica un lenguaje, en cuyo caso se llama **OpenCL C**. Como su nombre lo sugiere, es similar al lenguaje C; en particular a su estándar ISO C99, pero tiene ciertas restricciones (por ejemplo, no soporta recursión) y algunas extensiones (por ejemplo, tipos vectoriales).

Una aplicación de OpenCL se compone de dos partes[8]: uno o más ***kernels*** y un **programa anfitrión**.

Los *kernels* son las unidades básicas a ejecutar en paralelo. Son escritos en OpenCL C y cada instancia puede obtener el identificador del hilo en que se está ejecutando mediante la función `get_global_id`. Es posible hacer una sincronización de distintas instancias paralelas de un *kernel* llamando a la primitiva `barrier`, pero sólo ocurrirá dentro de cada bloque de hilos.

Por otra parte, el programa anfitrión se encarga generalmente de las siguientes tareas:

- Obtención de lista de dispositivos disponibles y selección.
- Lectura de fuentes de *kernels* y su preparación.
- Creación de una cola e introducción de los *kernels* en ella para su ejecución.
- Finalización y liberación de recursos.

*Siglas de *Compute Unified Device Architecture*.
†Acrónimo de *Open Computing Language*.
‡Siglas de *Installable client driver*.

El anfitrión es quién se encarga de indicar en dónde y cómo se ejecutarán los *kernels*. Para la primera tarea mencionada se consultarán los cargadores de ICD instalados para listar los dispositivos disponibles y obtener sus características.

La ejecución es llevada a cabo con una cola, la cual permite establecer el orden de ejecución y tener control de cuando terminan todos los hilos que ejecutan instancias de un *kernel*. Cuando finaliza uno de ellos ocurre una sincronización global que involucra a todos sus hilos asociados, la cual es distinta a la que ofrece `barrier`.

Khronos mantiene una API para los lenguajes C y C++ para escribir los programas anfitriones. Adicionalmente también existen *bindings* para otros lenguajes tales como Java y Python.

Capítulo 3

Clasificación en $O(n\log(n))$

En este capítulo se presenta el algoritmo publicado por Hervías et al. en el año 2014.

La sección 3.1 explica cómo funciona, a la vez que se entrega un ejemplo paso a paso de cómo avanza y se detalla qué complejidad computacional tiene. En la sección 3.2 se presenta una versión extendida del algoritmo – desarrollada en el transcurso del presente año – y su implementación actual.

3.1. Algoritmo de Hervías et al.

Actualmente en la literatura hay muy pocos algoritmos[3] para encontrar espacios de baja densidad en distribuciones de galaxias y no hay un método de consenso para encontrarlos y caracterizarlos. Dado lo anterior la nueva forma propuesta por Hervías et al.[3] para determinarlos es un gran aporte a la discusión sobre el tema.

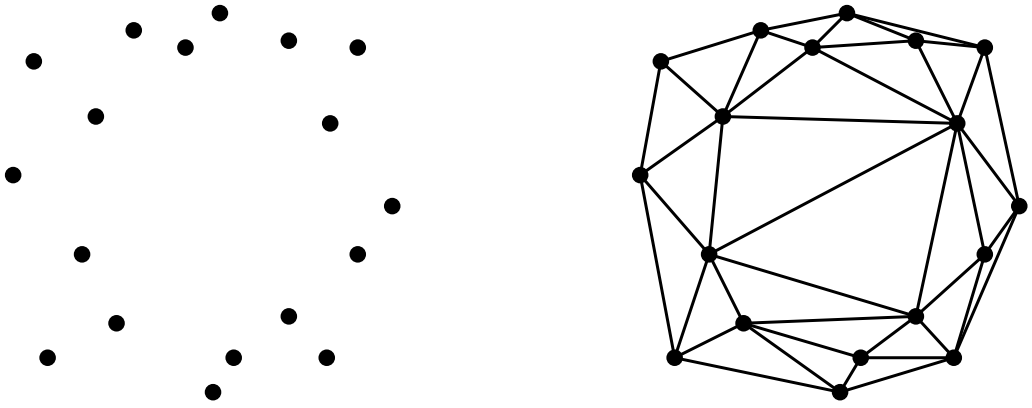
Como ya se mencionó en la parte introductoria, el algoritmo expuesto toma un conjunto de puntos bidimensionales como entrada y su forma de trabajo es naturalmente extensible al caso tridimensional. En el ámbito de aplicación a vacíos cósmicos, cada punto representaría a una galaxia.

A grandes rasgos, se distinguen dos fases principales en su proceso:

- **Triangulación**: Toma los puntos de entrada y calcula su triangulación de Delaunay.
- **Clasificación**: Decide para cada triángulo generado en la fase anterior, si pertenece a un vacío o no. Si es así, además le indica en qué vacío está.

El algoritmo se basa en la heurística de que los conjuntos de triángulos adyacentes cuyos vértices representan una zona de baja densidad de puntos, se caracterizan generalmente por exhibir aristas muy largas. Un ejemplo se muestra en la figura 3.1.

La fase de clasificación de triángulos se detalla en el algoritmo 3.1.



(a) **(b)**

Figura 3.1: *Ejemplo de la heurística.* ***(a)*** *Conjunto de puntos $P \subseteq \mathbb{R}^2$ simulando una zona de baja densidad en el centro.* ***(b)*** *Triangulación de Delaunay del conjunto P. Generalmente las zonas de baja densidad de puntos exhiben aristas muy largas.*

```
Leer triangulación de Delaunay;
Leer threshold_value;
void_list ← φ;
Ordenar triángulos por su arista más larga;
Repetir hasta que no haya triángulo por procesar:
  t1, t2 ← Los dos triángulos aún no usados que comparten arista más larga;
  triangle_set ← {t1, t2};
  Marcar a t1, t2 como usados;
  Para cada triángulo t que es adyacente a triangle_set:
    Si t comparte su arista más larga con un triángulo de triangle_set:
      triangle_set ← triangle_set ∪ {t};
      Marcar a t como usado;
  Si área de triángulos de triangle_set ≥ threshold_value:
    void_list ← void_list ∪ {polygon(triangle_set)};
```

Algoritmo 3.1: *Fase de clasificación de triángulos del algoritmo de Hervías et al.*

Específicamente la fase mencionada primero ordena por arista más larga. Luego intentar tomar el par de triángulos `t1` y `t2` que comparten la arista de largo máximo en la triangulación compuesta por triángulos aún no usados. Ambos conforman un conjunto inicial, llamado `triangle_set`. Luego examina los triángulos adyacentes al conjunto, viendo si alguno comparte su arista más larga con los que ya están dentro y agregándolos a `triangle_set`, de ser así.

La adhesión finaliza cuando ya no hay triángulos que compartan su arista más larga con el conjunto. En aquél momento, el algoritmo calcula el área que cubre el `triangle_set` terminado. Si es mayor o igual que `threshold_value`, se agrega el polígono equivalente a la lista de vacíos `void_list`; si no, se descarta. Así prosigue sistemáticamente la clasificación hasta que no quedan más triángulos no usados por procesar.

En la publicación se indica que la salida del algoritmo es un conjunto de vacíos, cada

uno de ellos siendo un polígono que se define como un conjunto de triángulos. En esta memoria se interpretará a `polygon` como una función que etiqueta a todos los triángulos que recibe, con un mismo identificador único de vacío.

Los conjuntos que arroja el algoritmo son llamados **vacíos poligonales*** y en el contexto de Astronomía, pueden asociarse a vacíos cósmicos.

Un ejemplo de cómo avanza la fase de clasificación sobre una triangulación simple se ofrece paso a paso en la figura 3.2 y se explica el detalle a continuación.

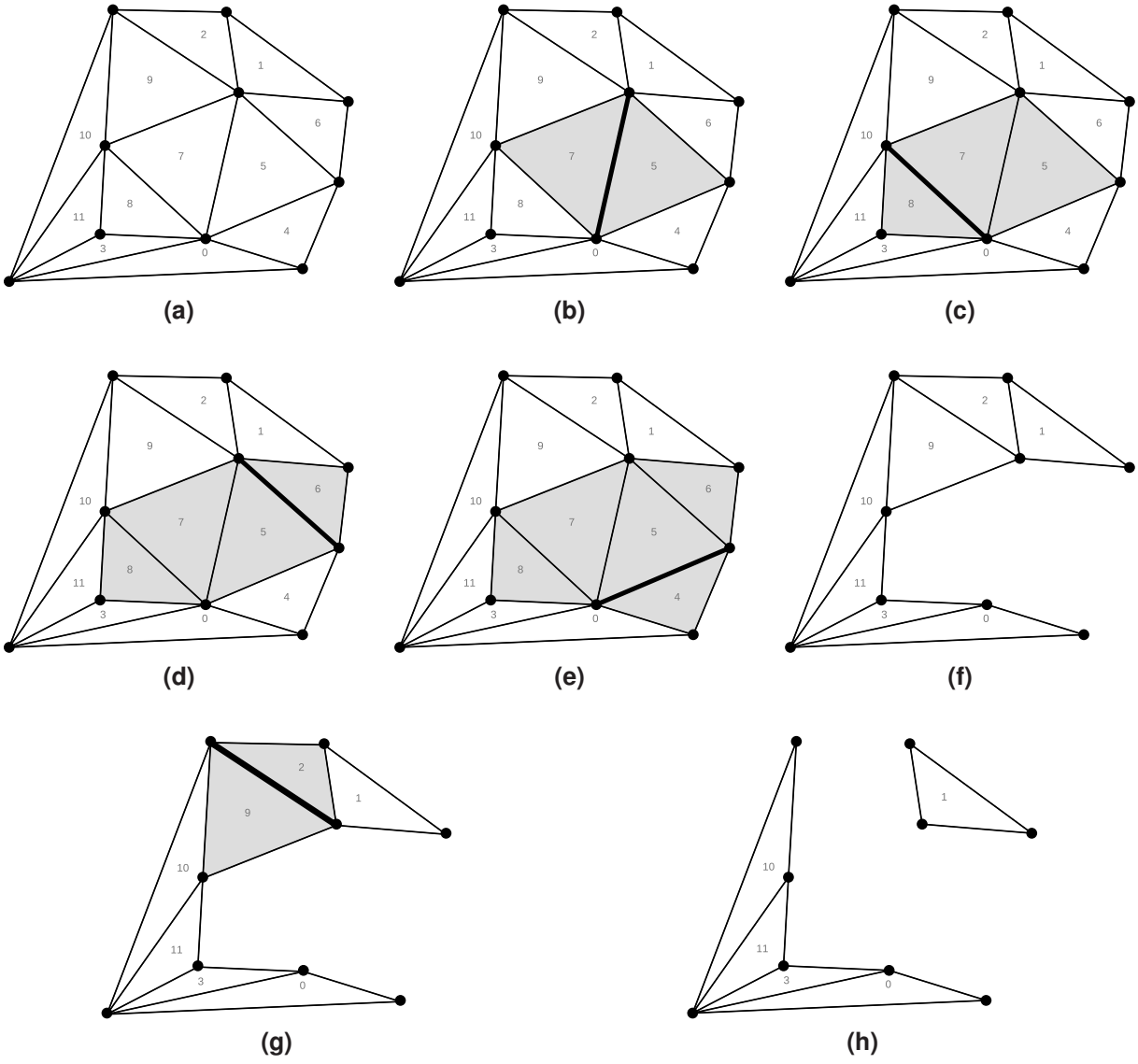


Figura 3.2: *Ejemplo de identificación de vacíos.* ***(a)*** *Misma triangulación de la figura 2.2a.* ***(b)-(h)*** *Se muestra paso a paso el proceso. La arista de mayor grosor es la que se está examinando en el momento correspondiente y los triángulos de color gris claro representan a los que han sido incorporados al* `triangle_set`*.*

Teniéndose la triagulación de la figura 3.2a, se ordenan las aristas, resultando la más

*A partir de aquí se utilizarán indistintamente los nombres “algoritmo de Hervías et al.” y “algoritmo de vacíos poligonales”.

larga la que comparten los triángulos 5 y 7. A continuación, en la figura 3.2b, el algoritmo considera aquélla arista y agrega a ambos triángulos mencionados al conjunto `triangle_set`. El triángulo 8 comparte su arista más larga con el etiquetado con 7, agregándose al conjunto en expansión, como se aprecia en la figura 3.2c. Lo mismo sucede con los triángulos 6 y 4, en las figuras 3.2d y 3.2e respectivamente.

Luego, no quedan más triángulos adyacentes al `triangle_set` que compartan su arista máxima con él, por lo que el algoritmo compara con el `threshold_value` el área que cubre. De acuerdo a esa comparación, se agrega el conjunto de triángulos a `void_list` o no. En la figura 3.2f se hace explícito que los triángulos ya fueron utilizados.

Nuevamente se inicia un `triangle_set` en la figura 3.2g y esta vez los triángulos iniciales son 2 y 9. Luego de su incorporación al conjunto, no hay triángulos que compartan su arista máxima con aquél por lo que nuevamente se calcula el área total y se compara con el `threshold_value` para ver si se agrega como vacío.

Finalmente, el algoritmo no encuentra otra arista tal que sea la más larga de la triangulación restante y que a la vez sea compartida por dos triángulos, terminando así el procesamiento. En la figura 3.2h se muestra la triangulación sin el último `triangle_set` terminado y se nota que no necesariamente se utilizan todos los triángulos.

3.1.1. Complejidad temporal

Como se vio en la subsección 2.1.2, una triangulación de Delaunay para n puntos puede calcularse en $O(n\log(n))$. Así, la primera fase de **triangulación** toma $O(n\log(n))$ y adicionalmente arroja $O(n)$ triángulos.

Respecto a la segunda fase, la publicación indica que el procesamiento de los sucesivos `triangle_set` finalizados toma $O(n)$. Sin embargo, para poder hacer lo anterior se deben identificar las aristas más largas, debiéndose ordenar ellas en $O(n\log(n))$. Por consiguiente resultando la **clasificación** en $O(n\log(n))$.

De esta forma el proceso completo de los n puntos de entrada tiene una complejidad computacional de $O(n\log(n))$.

3.2. DELFIN extendido

A mitad del presente año, Alonso[9] añade una tercera fase que procesa los vacíos poligonales identificados en la etapa de clasificación: consiste en la fusión de vacíos adyacentes*, en función de diferentes criterios a elección, incluyendo la opción de dejarlos sin

*Alonso introduce el concepto de **subvacío** para referirse a los vacíos poligonales inducidos por el algoritmo original y redefine el concepto de "vacío" para nombrar a los conjuntos de triángulos resultantes de la nueva fase de fusión. No se utilizará esa nomenclatura en esta memoria.

modificación (no fusionarlos).

El trabajo realizado por él corresponde a su tesis de Magíster en Ciencias mención Computación, programa de estudio que cursó en esta escuela. En él se denomina como **DELFIN**[*] al algoritmo de Hervías et al. y a la extensión desarrollada por él como **DELFIN extendido**.

Alonso logra una implementación cuya clasificación de forma experimental demuestra ser $O(n \log(n))$. Fue escrita en Python y se apoya en librerías de computación científica para realizar sus cálculos. Para la triangulación utiliza qdelaunay.

Entre los datos que retorna están los conjuntos resultantes de las fusiones de vacíos, que son internos a la triangulación; es decir, aquellos que no tienen triángulos colindantes con la envoltura convexa original. La razón para no incluir los que sí tocan el borde de la triangulación es que podrían estar incompletos, por lo que probablemente no calificarían como vacíos cósmicos propiamente tales, desde un punto de vista de aplicación en Astronomía.

Cuando se utiliza el criterio de no fusión, los resultados del proceso total de DELFIN extendido son idénticos a los vacíos internos a la triangulación que entrega el algoritmo original.

[*] Acrónimo de ***Del****aunay edge void* ***fin****der*.

Capítulo 4

Caracterización de vacíos poligonales

En este capítulo se presentarán las propiedades encontradas para los vacíos que arroja el algoritmo ya presentado en la sección 3.1. Para aquél cometido, previamente se establece una nomenclatura relevante respecto del problema para los tipos de aristas y triángulos que existen en una triangulación, en la sección 4.1.

Luego se presentan las propiedades en las secciones 4.2, 4.3 y 4.4, comentándose el conocimiento que aportan sobre los vacíos.

4.1. Nomenclatura

4.1.1. Zonas, vacíos y murallas

Tomando en consideración la terminología utilizada en el tema de los vacíos cósmicos, se le llamará **zona** a un conjunto de triángulos acumulados en `triangle_set` cuando el algoritmo está listo para comparar el área que cubre respecto del valor base.

Si efectivamente aquélla es mayor o igual que `threshold_value`, la zona pasa a ser un **vacío** y se agrega a `void_list`. Si no, pasa a ser una **muralla** y se descarta.

También podría ser que existieran triángulos que no pertenecen ni a vacíos ni a murallas. En particular aquellos triángulos cuyas sus aristas más largas pertenecen a la envoltura convexa de la triangulación: dada esa característica ninguna zona podrá agregarlos hacia sí. Si un triángulo no pertenece a vacío o muralla alguna, se dirá que está en una **no zona**.

Finalmente, en forma genérica se hará referencia a zonas, murallas, vacíos y no zonas, como **componentes**.

4.1.2. Aristas de borde, internas y semilla

Respecto a las aristas de los triángulos, se dirá que una arista es **de borde de triangulación** si pertenece a la envoltura convexa de la triangulación. En caso de sea compartida por dos triángulos, se llamará arista **interna a la triangulación**.

En relación a las zonas, considérese una y tómese una arista que a través de ella se agregó un triángulo a la zona durante su expansión. Aquélla arista se le denominará **interna de zona**.

Siguiendo con las zonas, también tienen bordes compuestos por aristas. Aquéllas últimas se llamarán **de borde de zona** y se caracterizan porque a través de ellas no se agregó ningún otro triángulo durante la construcción de la zona. Lo anterior lleva a que pudiera darse la situación en que una arista de borde de zona está en el interior de su zona, pero aún así – por la definición entregada – es de borde de zona.

Finalmente, considérese el momento en que el algoritmo extrae un primer par de triángulos para iniciar la construcción de una zona. Aquéllos comparten una arista, la que se denominará **arista semilla**. Por extensión, los triángulos colindantes a ella se llamarán **triángulos semilla** de la zona.

4.1.3. Tipos de aristas internas a la triangulación

Cuando el algoritmo introduce el primer par de triángulos al `triangle_set`, necesariamente ambos deben compartir la arista máxima de toda la triangulación compuesta por los triángulos que aún no han sido marcados como usados. En particular, aquélla arista claramente es máxima para los dos triángulos iniciales. A este tipo de arista se le denominará arista máxima-máxima, o más brevemente **arista máx-máx**.

Luego de que se toma el primer par para `triangle_set`, se van agregando otros triángulos. Una propiedad que se mantiene a lo largo de la construcción del conjunto, es que las aristas que lo bordean, no son máximas para los triángulos de `triangle_set` a las que pertenecen.

Siguiendo con la expansión del `triangle_set`, cada triángulo que se incorpora debe compartir su arista máxima con el conjunto. Pero, como éste último tiene por borde a aristas no máximas para los triángulos internos adyacentes a él, cada arista involucrada en una adhesión es máxima para el triángulo a incorporarse pero no máxima para el triángulo adyacente interno correspondiente. Aquí se configura una arista que se llamará “arista máxima-no máxima”. En forma más breve, **arista máx-nomáx**.

Finalmente, cuando el conjunto `triangle_set` termina de expandirse, es porque ya no quedan triángulos que le compartan su arista máxima. En ese momento, las aristas internas a la triangulación que bordean al conjunto son no máximas tanto para los triángulos internos que están en el borde, como para los triángulos adyacentes externos al conjunto.

A aquél tipo de aristas – como podría esperarse – se denominará “arista no máxima-no nomáxima”. Más convenientemente, como **arista nomáx-nomáx**.

Una muestra de cómo se ven los distintos tipos de aristas en una triangulación, se ilustra en la figura 4.1.

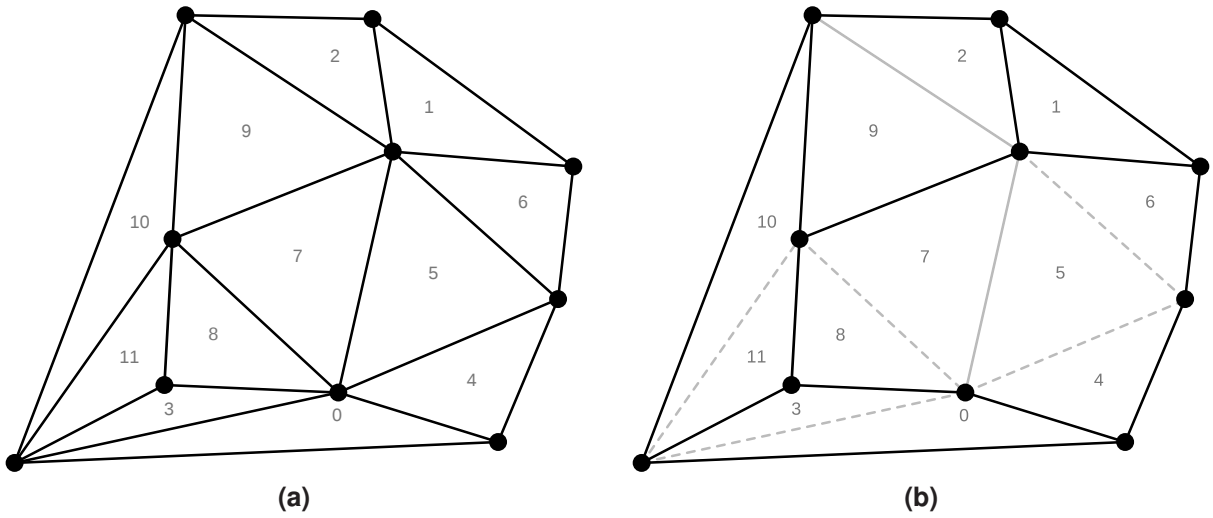


Figura 4.1: ***(a)*** *La misma triangulación de la figura 2.2.* ***(b)*** *Los tipos de aristas que posee. Aristas negras son de borde de triangulación o nomáx-nomáx, las grises sólidas son máx-máx y las grises segmentadas son máx-nomáx.*

Con la nomenclatura construida aquí, ya es posible presentar las propiedades encontradas para los vacíos.

4.2. Sobre aristas máx-máx

Lema 4.1. *Sea x una arista interna a la triangulación. x es máx-máx $\Longrightarrow$ x es elegida como semilla por el algoritmo para crear zona.*

Demostración. En forma directa.

Suponer que x es máx-máx. Nótese que los triángulos T y U que comparten a la arista no pueden ser agregados a otra zona porque no ofrecen arista máxima alguna hacia el exterior.

El algoritmo va usando triángulos hasta que la arista máxima sobre los aún no utilizados es x. Por la forma en que trabaja, debería tomar a T y U como semillas para iniciar una zona. ■

Lema 4.2. *Sea x una arista interna a la triangulación. x es elegida como semilla por el algoritmo para crear zona $\Longrightarrow$ x es máx-máx.*

Demostración. En forma directa.

Suponer que x fue elegida como semilla. Por la forma en que las elige el algoritmo, debe ser la arista máxima sobre la triangulación aún no usada. Por tanto es la más larga para los triángulos T y U que la comparten. Finalmente x debe ser máxima para T y U. ■

Teorema 4.3. *Sea x una arista interna a la triangulación. x es máx-máx $\iff$ x es elegida como semilla por el algoritmo para crear zona.*

Demostración. Por inferencia lógica.

En base a los lemas 4.1 y 4.2. ■

Los resultados anteriores indican que las aristas máx-máx sólo pueden pertenecer al interior de zonas. Es algo propio de ellas, por lo que las identifica como tales. Existe una única por cada zona.

4.3. Sobre aristas nomáx-nomáx

Teorema 4.4. *Sea x una arista interna a la triangulación. x es de borde de zona $\implies$ x es nomáx-nomáx.*

Demostración. Por contradicción.

Suponer que x es de borde de zona pero que no es nomáx-nomáx. Entonces debe ser máx-máx o máx-nomáx.

Sean T y U los triángulos que comparten x. Suponer que el algoritmo agrega primero a T para luego examinar su vecindad.

- **Si x es máx-máx**: entonces T es el primer triángulo de una zona en generación. Como U comparte su arista máxima con T, entonces el algoritmo agregará a U a través de x.
- **Si x es máx-nomáx**: entonces T debe compartir su arista máxima con la zona en generación, por lo que comparte obligatoriamente una de sus aristas no máximas con U. Entonces éste último debe compartir su arista máxima con T, para que se configure una arista máx-nomáx.

  En este escenario, al agregar T el algoritmo a la zona en generación, se encuentra con que U comparte su arista máxima con él. Luego, debería agregar a U a la zona, a través de x.

Se concluye que en ambos casos, x se utiliza para agregar un nuevo triángulo, por lo que no puede ser una arista de borde de zona, llegándose a una contradicción. ■

Teorema 4.5. *Sea x una arista interna a la triangulación. x es nomáx-nomáx* $\Longrightarrow$ *x es de borde de zona* $\vee$ *pertenece al exterior de toda zona.*

Demostración. Por contradicción.

Suponer que *x* es nomáx-nomáx y que no es de borde de zona ni está en el exterior de toda zona. Eso significa que debiera ser una arista interior a una zona.

Sean *T* y *U* los triángulos a los que es adyacente la arista *x*. Supóngase que *T* fue agregado primero durante la construcción de la zona. Para que *x* fuese interna a la zona, a través de ella debió haberse agregado a *U*. Sin embargo, eso no es posible porque *x* es nomáx-nomáx. ■

Así una zona tiene por borde interno a la triangulación a aristas nomáx-nomáx; pero no todo conjunto que tenga esa característica corresponde a una zona. Un ejemplo de aquello se aprecia en la figura 4.2.

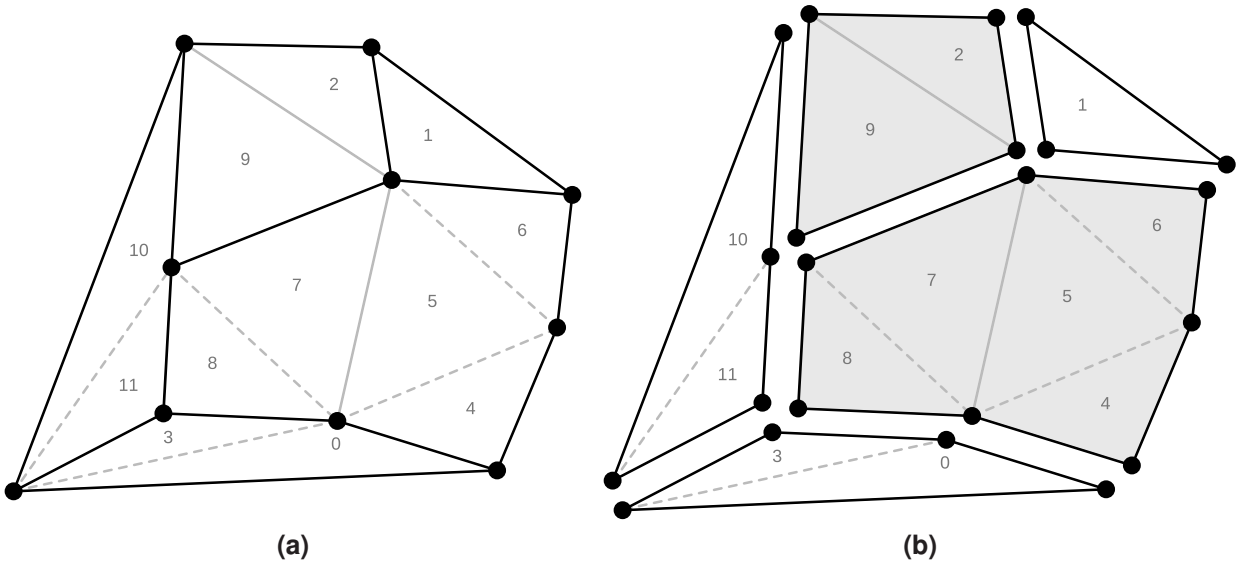


Figura 4.2: ***(a)*** *Misma triangulación con sus tipos de aristas resaltados de la figura 4.1.* ***(b)*** *Se particiona la triangulación, separando los triángulos que comparten aristas nomáx-nomáx. El algoritmo indica que los conjuntos de color gris claro son zonas.*

Los cinco conjuntos tienen por aristas de borde internas a la triangulación a aristas nomáx-nomáx y sólo dos de ellos son zonas. No es coincidencia que sólo ellos contengan una arista máx-máx en su interior (aristas grises sólidas), dado lo inferido en la sección 4.2.

4.4. Sobre zonas

Teorema 4.6. *Sea Z una zona de triángulos y $G = (V, E)$ su grafo de adyacencia. Existe un árbol dirigido $A = (V, E')$ que se puede construir en base a G.*

Demostración. Por inducción matemática sobre el número k de triángulos pertenecientes a una zona Z en construcción.

Caso base $k = 2$: En este caso hay una zona Z compuesta por dos triángulos R y S adyacentes. Se esquematiza la situación en la figura 4.3.

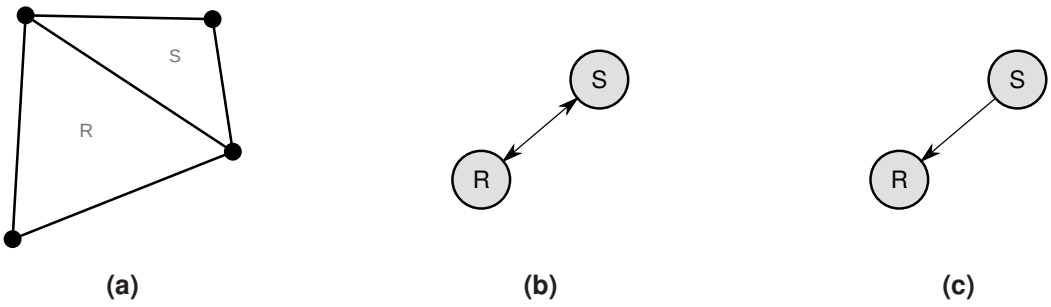


Figura 4.3: ***(a)*** *Esquema de configuración de una zona para $k = 2$.* ***(b)*** *El grafo de adyacencia $G = (V, E)$ de la zona.* ***(c)*** *De G se puede deducir un árbol A, escogiéndose a R como raíz.*

En este caso, el grafo de adyacencia se compone de dos nodos y dos arcos dirigidos. Simplemente se puede remover uno de los arcos. Lo anterior designa al nodo en que termina el arco restante como raíz.

Finalmente se tiene un árbol dirigido A, pues el grafo no dirigido subyacente es acíclico y conexo. Adicionalmente, los arcos presentes se dirigen hacia el nodo elegido como raíz.

Paso inductivo: Suponer que se tiene una zona Z en construcción de k triángulos y su árbol dirigido A de k nodos. Sea G el grafo de adyacencia asociado a la zona.

En cada paso en que se agrega un triángulo U, éste comparte su arista máxima con algún triángulo T que ya está adentro. Sea Z' la nueva zona de $k + 1$ triángulos que incluye a U y sea G' su grafo de adyacencia.

Claramente $G \subset G'$ y adicionalmente, por hipótesis inductiva, de G se puede deducir un árbol A. Si se hace esa transformación en G', queda un grafo intermedio en que está A junto con el nuevo nodo para U y todos los arcos que lo relacionan con otros triángulos que están en Z.

Se puede generar un grafo A' removiendo todos los arcos mencionados a excepción del que apunta desde U a T. El efecto neto de eso es simplemente “colgar” un hijo a un nodo de A, tal como se ve en la figura 4.4.

Así queda A más el nodo para U y un arco dirigido desde él hacia T. Por la forma en que se conectó el nuevo nodo, no puede formarse un ciclo y el nuevo arco se dirige hacia la

raíz de *A*.

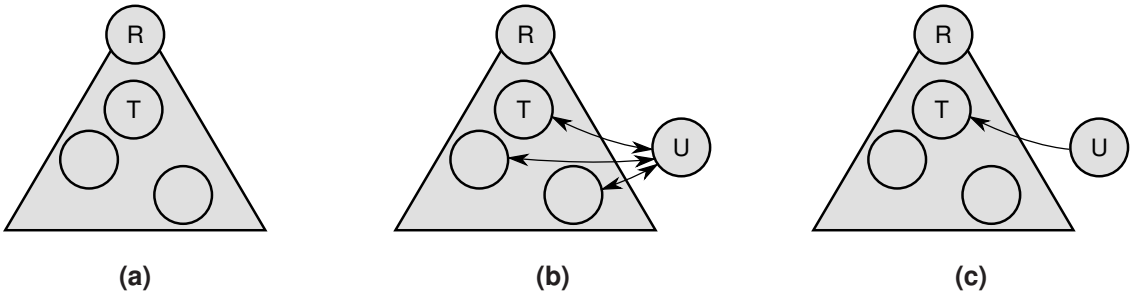


Figura 4.4: ***(a)*** *Árbol A obtenible desde el grafo de adyacencia G de la zona Z de k triángulos. En su interior se encuentra el nodo asociado a T. A modo de ejemplo hay otros dos nodos cuyos triángulos asociados también son adyacentes a U.* ***(b)*** *Grafo intermedio resultante de transformar en A a la porción G contenida en G′ de la zona Z′ de k* + 1 *triángulos.* ***(c)*** *Deducción de A′ removiendo las nuevas aristas a excepción de la dirigida desde U a T.*

Así *A′* posee sólo arcos que apuntan hacia su raíz y su grafo no dirigido subayacente es conexo y acíclico. Por tanto es un árbol dirigido válido. ■

Nótese en el caso base que la zona tiene dos triángulos y por tanto sólo una arista interna. Ella debe ser obligatoriamente máx-máx pues toda zona debe tener una de ese tipo, según lo visto en la sección 4.2.

Respecto al paso inductivo obsérvese que el único arco remanente de los eliminados al pasar del árbol *A* al *A′*, siempre corresponde a uno asociado a una arista máx-nomáx; en particular, el que se dirige desde el triángulo para el que la arista es máxima hacia el triángulo para el que no es máxima. Por otra parte, un arco siempre eliminado es el asociado a esa arista máx-nomáx pero que se dirige en forma contraria a la descrita anteriormente. El resto de los removidos – de haber – son arcos asociados a aristas nomáx-nomáx.

El árbol deducido en la forma expuesta, se le denominará **árbol de zona**.

Capítulo 5

Clasificación en $O(n)$

Del conocimiento adquirido en el capítulo anterior es posible desarrollar nuevas formas de clasificación, que llegan exactamente a los mismos resultados que el algoritmo de vacíos poligonales original (algoritmo 3.1).

En este capítulo se presentan dos clasificaciones $O(n)$: una secuencial y una paralela, utilizando las propiedades para las aristas máx-máx y nomáx-nomáx encontradas.

5.1. Algoritmos

Las nuevas clasificaciones a presentar tienen por pilares fundamentales las siguientes dos ideas:

- Sólo las zonas pueden tener aristas máx-máx, cada zona teniendo una sola arista de ese tipo (teorema 4.3).
- Conjuntos de triángulos adyacentes bordeados por aristas nomáx-nomáx no necesariamente son zonas. Sin embargo, una zona sí se caracteriza por tener un borde así (teoremas 4.4 y 4.5).

Lo expuesto conduce a que la triangulación se puede particionar tal como se mostró en la figura 4.2b y viendo si los conjuntos resultantes poseen o no una arista máx-máx, se puede saber inmediatamente si son zonas o no.

Desde un punto de vista de grafo de adyacencia, se pueden eliminar los arcos asociados a aristas nomáx-nomáx y quedarán varias componentes conexas disjuntas. Un ejemplo de cómo es esa remoción de arcos se ve en la figura 5.1.

Si alguna componente posee arcos asociados a aristas máx-máx, entonces aquella corresponde a una zona. Decidir si una componente es zona también puede llevarse a cabo viendo si alguno de sus nodos está asociado a un triángulo que posee una arista máx-máx.

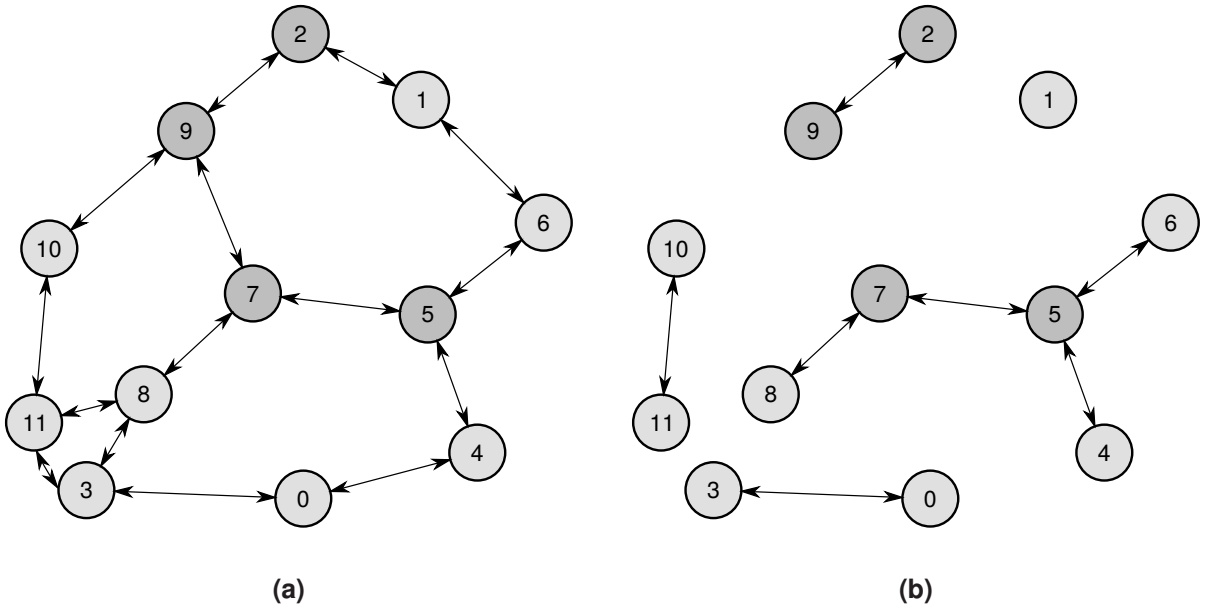


Figura 5.1: ***(a)*** *Grafo de adyacencia G de la triangulación mostrada en la figura 4.2a. Se resaltan con gris oscuro los nodos asociados a triángulos que comparten aristas máx-máx.* ***(b)*** *Remoción de G de los arcos asociados a aristas nomáx-nomáx. La partición que produce en términos de triangulación, es la mostrada en la figura 4.2b. El grafo G pasa a ser una colección de componentes conexas disjuntas, cada una representando a un conjunto de triángulos de los mostrados en la figura mencionada.*

Se puede hacer un DFS (búsqueda por amplitud) sobre el grafo. Luego de cada visita a cada componente conexa es posible saber si tenía algún nodo asociado a un triángulo con arista máx-máx, así pudiéndose decidir si la componente correspondiente es zona. En este capítulo, se dirá que un nodo es **raíz** de su componente, si fue el primero de ella en ser visitado.

Más aún se puede aprovechar el DFS para obtener el área total de los triángulos asociados a cada componente examinada, sin aumentar la complejidad computacional de él. Si es zona, se puede comparar su área con el `threshold_value` para resolver si es vacío o muralla.

De la misma forma, para ampliar las posibles aplicaciones de los vacíos poligonales, se puede saber si una zona es un vacío que toca el borde de la triangulación o está completamente dentro de ella. Basta con consultar durante cada visita si hay algún nodo cuyo triángulo asociado que toque la envolvente convexa original.

Toda la información de las componentes recabada por el DFS se puede guardar en las raíces correspondientes. Sin embargo, a fin de clasificar a todos los triángulos, es necesario comunicarla desde las raíces hacia sus nodos. Nuevamente aprovechando las visitas del DFS, se puede propagar un puntero hacia la raíz de cada componente, quedando el grafo particionado como se muestra en la figura 5.2.

Con todos los nodos apuntando hacia su raíz correspondiente, pueden copiar hacia sí toda la información resultante de las visitas.

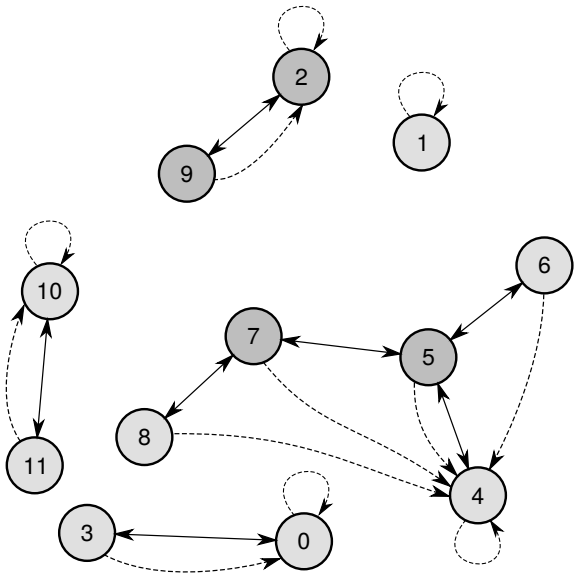


Figura 5.2: *El grafo de adyacencia particionado de la figura 5.1b inmediatamente después del DFS. Se muestran con curvas segmentadas los punteros que deja en todos los nodos hacia las raíces de sus respectivas componentes. La forma en que se pueden elegir las raíces no es única.*

5.1.1. Secuencial

Con todo lo expuesto ya es posible ofrecer un nuevo algoritmo de clasificación secuencial. Toma el grafo de adyacencia $G = (V, E)$ resultante de la triangulación de Delaunay previa y un valor para el `threshold_value`.

En primera instancia marca las aristas máximas de cada triángulo, con lo cual es posible identificar a las aristas nomáx-nomáx para su remoción de G. Así, quedan las componentes conexas mencionadas con antelación.

Luego, procediendo como lo hace un DFS, por cada $v \in V$ pregunta si ha sido visitado o no, visitándolo si no. Cada recorrido a una componente conexa retorna:

- Suma total de las áreas de los triángulos asociados a la componente.
- Si toca el borde de la triangulación (envolvente convexa).
- Si tiene triángulo que posea arista máx-máx.

Con esa información es posible determinar si la componente es zona o no zona. En caso de que sea zona, se puede indicar si es muralla, vacío de borde o vacío interno. El tipo resultante se escribe en la raíz de cada componente.

El DFS también se aprovecha para propagar un puntero hacia la raíz de las componentes. Como se mencionó previamente, aquéllo permite que después todos los nodos puedan leer de su raíz si pertenecen a una no zona, muralla, etc..

El nodo raíz actúa como el representante de su componente y como todos los nodos conocen la raíz de su componente conexa, pueden copiar hacia sí el identificador del triángulo asociado a ella, el cual se puede suponer único. Así cada componente queda

con sus triángulos etiquetados con un mismo identificador para todos, pudiéndose saber *a posteriori* a qué componente pertenecen.

Finalmente completándose la clasificación: todos los triángulos "saben" a qué componente pertenecen y el tipo de ella. Todo se resume en el algoritmo 5.1.

```
clasificacion(G=(V,E) : Grafo de adyacencia, threshold : Threshold value)
  Para cada nodo v ∈ V:
    Marcar arista máxima de triángulo asociado a v
  Para cada nodo v ∈ V:
    Para cada e ∈ E que sale de v:
      Si e está asociado a una arista nomáx-nomáx:
        Eliminar e de E
  Para cada nodo v ∈ V:
    Si v no ha sido visitado:
      toca_borde, area_total, tiene_maxmax ← visitar(G, v, v)
      Si toca_borde ∧ area_total ≥ threshold ∧ tiene_maxmax:
        Tipo de v ← VACIO_BORDE
      Sino, si !toca_borde ∧ area_total ≥ threshold ∧ tiene_maxmax:
        Tipo de v ← VACIO_INTERNO
      Sino, si area_total < threshold ∧ tiene_maxmax:
        Tipo de v ← MURALLA
      Sino:
        Tipo de v ← NO_ZONA
  Para cada nodo v ∈ V:
    Tipo de v ← Tipo de la raíz de v
    Id. de componente de v ← Id. de triángulo de raíz de v
  Retornar G

visitar(G=(V,E) : Grafo de adyacencia, v : Nodo a explorar, r : Nodo raíz)
  Marcar v como visitado
  Raíz de v ← r
  toca_borde ← Triángulo asociado a v está en borde de triangulación?
  area_total ← Área de triángulo asociado a v
  tiene_maxmax ← Triángulo asociado a v tiene arista máx-máx?
  Para cada vecino u de v:
    Si u no ha sido visitado:
      u_toca_borde, u_area_total, u_tiene_maxmax ← visitar(G, u, r)
      toca_borde ← toca_borde ∨ u_toca_borde
      area_total ← area_total + u_area_total
      tiene_maxmax ← tiene_maxmax ∨ u_tiene_maxmax
  Retornar toca_borde, area_total, tiene_maxmax
```

Algoritmo 5.1: *La nueva fase de clasificación secuencial propuesta. Se invoca primero a* `clasificacion`.

De la forma en que trabaja el DFS, la rutina mostrada requiere de una auxiliar para hacer visitas recursivas desde las raíces que encuentra. Aquélla exhibe la misma estructura – y por tanto complejidad computacional – que el DFS mostrado en la sección 2.1.

Complejidad temporal

Sea n el número de puntos de la triangulación de Delaunay, con lo que se tienen $O(n)$ triángulos. Considérese una representación del grafo mediante una **matriz de adyacen-**

cia, que permite consultas en $\Theta(1)$ e implementada correspondería a un arreglo de $O(n)$ celdas, pues cada triángulo no puede tener más de tres adyacencias.

Adicionalmente considérese un arreglo que contiene a todos los nodos del grafo. Un ejemplo se muestra en la figura 5.3. El tamaño del arreglo es $O(n)$, pues se tiene un nodo para cada triángulo.

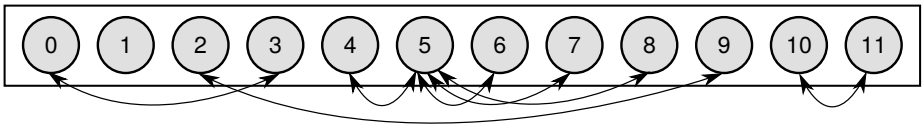


Figura 5.3: *Ejemplo de arreglo de nodos. Se muestran los nodos del grafo de adyacencia particionado de la figura 5.1b.*

Con la representación expuesta, se tiene el siguiente resultado:

Teorema 5.1. *Esta clasificación secuencial es $O(n)$.*

Demostración. En forma directa.

Las etapas de marcado de aristas máximas, eliminación de aristas nomáx-nomáx y obtención de tipo e identificador de componente, pueden ser realizadas con sólo escanear el arreglo de nodos y haciendo una cantidad constante de preguntas en cada paso, por lo que cada una es $O(n)$.

Dado el corolario 2.4, el DFS también es $O(n)$.

Finalmente, la complejidad total del proceso es $O(n)$. ■

A partir de esta nueva forma de clasificar, se puede obtener una versión paralela, presentada a continuación.

5.1.2. Paralelización

Es posible paralelizar partes para mejorar la eficiencia de la clasificación propuesta con antelación, pero no es una mejora en cuanto a complejidad.

Supónganse una máquina PRAM de una cantidad de procesadores igual al número de triángulos, es decir, $O(n)$ procesadores. En adición, permítase concurrencia sólo para lecturas.

Se puede mejorar el desempeño de lo propuesto en el algoritmo 5.1, paralelizando todo a excepción de la búsqueda por amplitud, según se presenta en el algoritmo 5.2 y su subsecuente análisis.

```
clasificacion(G=(V,E) : Grafo de adyacencia, threshold : Threshold value)
  Para cada procesador i, en paralelo:
    Marcar arista máxima de triángulo asociado al i-ésimo nodo v ∈ V
  Para cada procesador i, en paralelo:
    Para cada e ∈ E que sale del i-ésimo nodo v ∈ V:
      Si e está asociado a una arista nomáx-nomáx:
        Eliminar e de E
  Para cada nodo v ∈ V:
    Si v no ha sido visitado:
      toca_borde, area_total, tiene_maxmax ← visitar(G, v, v)
      Si toca_borde ∧ area_total ≥ threshold ∧ tiene_maxmax:
        Tipo de v ← VACIO_BORDE
      Sino, si !toca_borde ∧ area_total ≥ threshold ∧ tiene_maxmax:
        Tipo de v ← VACIO_INTERNO
      Sino, si area_total < threshold ∧ tiene_maxmax:
        Tipo de v ← MURALLA
      Sino:
        Tipo de v ← NO_ZONA
  Para cada procesador i, en paralelo:
    v ← El i-ésimo nodo de V
    Tipo de v ← Tipo de la raíz de v
    Id. de componente de v ← Id. de triángulo de raíz de v
  Retornar G
```

Algoritmo 5.2: *Fase de clasificación paralela propuesta. Se invoca inicialmente a* `clasificacion`*. La función* `visitar` *se define en el algoritmo 5.1.*

Complejidad temporal

Para este análisis considérese la misma representación de matriz de adyacencia, junto a un arreglo que contiene a los nodos que representan a los triángulos de la triangulación de n puntos.

Teorema 5.2. *Esta clasificación paralela, en CREW, es* $O(n)$.

Demostración. En forma directa.

El marcado de aristas máximas se hace en $\Theta(1)$. Cada procesador lee información del triángulo asociado a su nodo y resuelve en un número constante de comparaciones la arista más larga.

En la etapa de determinación de aristas nomáx-nomáx y eliminación de los arcos asociados a ellas, cada procesador puede hacer uso de lecturas concurrentes para obtener las aristas máximas de los triángulos adyacentes. Luego, puede decidir el tipo de cada arista de su triángulo correspondiente y eliminar arcos. A lo más hay tres adyacencias y tres arcos salientes, por lo que la etapa es $\Theta(1)$.

De igual forma a la versión secuencial ya discutida, el DFS es $O(n)$.

Durante la etapa paralela final, cada procesador obtiene desde el nodo raíz asociado a su propio nodo: información sobre el tipo de componente y el identificador de la misma.

Son dos lecturas concurrentes, resultando la fase en $\Theta(1)$.

Luego, la complejidad es la suma de términos $\Theta(1)$ y $O(n)$, resultando todo en $O(n)$. ■

El impacto de la paralelización de las etapas mencionadas puede parecer menor, pero se verá en la sección 5.4 que resultan en una notable mejora de eficiencia respecto de la clasificación $O(n)$ secuencial.

5.2. Implementación

Se escribieron dos programas en el **lenguaje C** los cuales utilizan `popen` para comunicarse con el binario de qdelaunay necesario para la fase de triangulación. La utilización de la comunicación interprocesos mencionada es para evitar lecturas y escrituras en disco de archivos temporales, que mermen innecesariamente el desempeño global.

Ambos reciben un archivo de texto en que están las coordenadas de los puntos a procesar y el `threshold_value`. Arrojan en su salida cuatro archivos de texto, uno por cada tipo de componente listado a continuación:

- Vacíos que tocan borde de triangulación
- Vacíos que no tocan borde de triangulación (internos a ella)
- Murallas
- No zonas

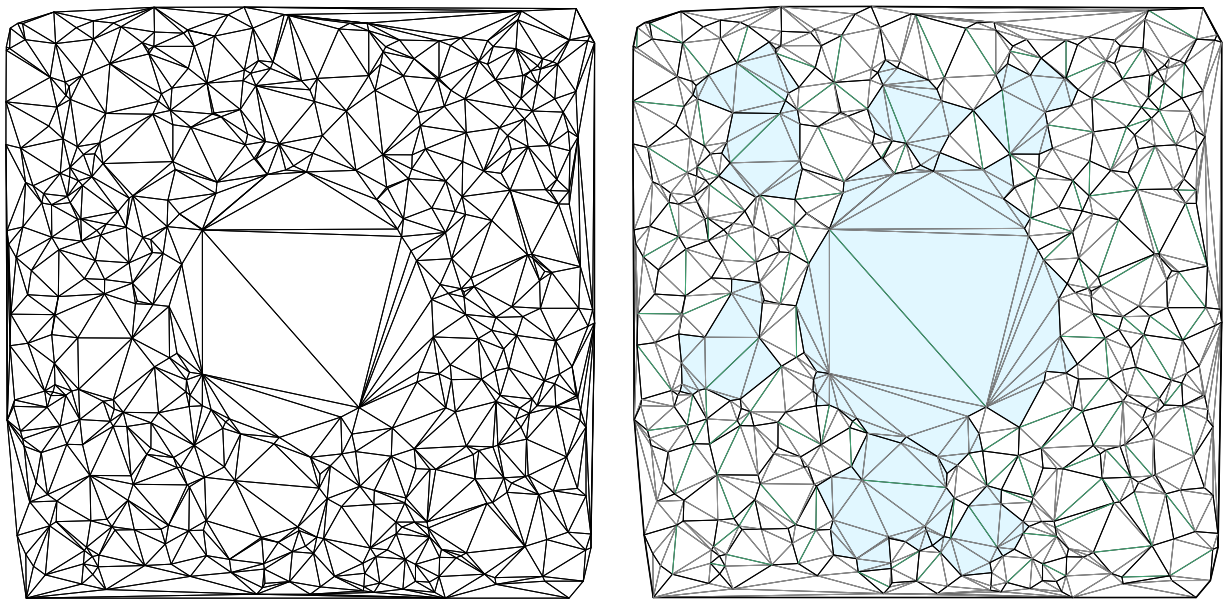
En cada archivo se detallan identificadores de triángulo y por cada uno de ellos el identificador de componente a la que pertenece. Al término de sus ejecuciones, muestran estadísticas acerca del número de componentes de cada tipo que se encontraron.

Producto de la llamada a qdelaunay los programas obtienen una serie de arreglos con información de la triangulación de Delaunay calculada, en particular un arreglo de adyacencias de triángulos.

En el caso del programa de la clasificación paralela, se creó un archivo adicional con los *kernels* escritos en el lenguaje OpenCL C, diseñados para el efecto. Se hacen las llamadas a la API de OpenCL adecuadas para hacer los preparativos y encolarlos para ejecución en paralelo. Respecto al acceso a datos se utilizó la modalidad Zero-copy.

Adicionalmente se creó una pequeña herramienta de visualización que toma los resultados de los programas anteriores y arroja imágenes SVG* con las triangulaciones y sus murallas, vacíos de borde, etc. resaltados, según se necesite. Un ejemplo de los gráficos que entrega se ve en la figura 5.4.

*Acrónimo de *Scalable Vector Graphics*.



(a) **(b)**

Figura 5.4: *Ejemplos de herramienta de visualización SVG desarrollada.* ***(a)*** *La triangulación de Delaunay de uno de los conjuntos de puntos de prueba que se utilizaron para DELFIN extendido. En el centro se simula una gran región de baja densidad.* ***(b)*** *Con un threshold value apropiado se encuentran 8 vacíos internos a la triangulación, que en suma totalizan 161 triángulos, exhibidos con color celeste. Las aristas negras son de borde de triangulación o nomáx-nomáx, mientras que las verdes son máx-máx y las grises son máx-nomáx. Para mayor detalle, se incluye en el anexo C una versión de mayor tamaño.*

5.3. Tests

Para verificar las soluciones que arrojan las implementaciones desarrolladas se hicieron diversas pruebas. En particular, con casos representativos de pocos puntos (por ejemplo, triangulaciones: sin zonas, en que completas constituían una sola zona, con combinaciones de zonas y no zonas con diferentes valores de `threshold_value`) y aleatorias con muchos puntos.

Para las últimas, era impracticable examinar manualmente si las clasificaciones de los triángulos arrojadas eran iguales a las calculadas por DELFIN extendido. Se debió idear una forma automatizada para compararlas.

Las tres implementaciones entregan archivos que contienen tuplas del tipo $(t, c) \in \mathbb{N}^2$, en que el primer valor es el identificador de triángulo y el segundo es el identificador de la componente a la que pertenece.

Para hacer los archivos comparables es necesario estandarizar los identificadores de las componentes: puede elegirse al mínimo identificador de triángulo dentro de los que pertenecen a la misma.

Ya teniendo la forma estándar para identificar a las componentes y como las implementaciones trabajan con los mismos identificadores de triángulo que arroja qdelaunay, se

puede hacer lo siguiente para cada archivo con resultados:

- Ordenar* tuplas por identificador original de componente.
- Cada bloque de tuplas con iguales identificadores originales de componente, ordenarlo por identificador de triángulo.
- En cada primera tupla de bloque quedará el menor identificador de triángulo; copiar a su identificador de componente. Propagar ese identificador hasta que termine el bloque. Los nuevos identificadores de componente son los estandarizados.
- Ordenar tuplas por identificador estandarizado de componente.
- Cada bloque de tuplas con iguales identificadores estandarizados, ordenarlo por identificador de triángulo.

Al pasar los archivos por este proceso, quedan comparables entre sí: sólo basta con comparar sus *i*-ésimas líneas. Si no hay diferencias entre ellas, entonces las clasificaciones de triángulos son iguales.

Todos los tests realizados en esta etapa de verificación **resultaron exitosos**: en cada uno de ellos, al compararse los vacíos que no tocaban el borde de la triangulación que arrojaban DELFIN extendido (sin fusiones) y las dos implementaciones $O(n)$, resultaban iguales.

5.4. Experimentación y resultados

Para ver el comportamiento experimental de las clasificaciones $O(n)$ se generaron conjuntos de puntos aleatorios sobre los cuales se hicieron repetidas ejecuciones, probándose con $2^{14}, 2^{15}, 2^{16}, \ldots, 2^{24}$.

Se entregó un `threshold_value` de cero, no teniendo esa decisión impacto en el tiempo de ejecución porque de igual forma deben calcularse las áreas totales de las componentes encontradas para decidir de qué tipo son.

Para la experimentación se utilizó un equipo con una **CPU multinúcleo** Intel Core i5-5200U de 2,20 GHz, que disponía de 16 GiB de memoria RAM y tenía Debian como sistema operativo.

En el caso de las clasificaciones $O(n)$ se experimentó hasta con 2^{24} puntos porque para el siguiente paso qdelaunay utilizaba más de 16 GiB. En el caso de DELFIN extendido, sólo hasta 2^{21} puntos porque para una cantidad mayor también excedía la capacidad de memoria.

En la figura 5.5 se aprecian los resultados para las clasificaciones desarrolladas. Los tiempos corresponden sólo a la fase de clasificación, sin la triangulación de Delaunay previa necesaria ni la salida de resultados.

*Supóngase un orden de menor a mayor.

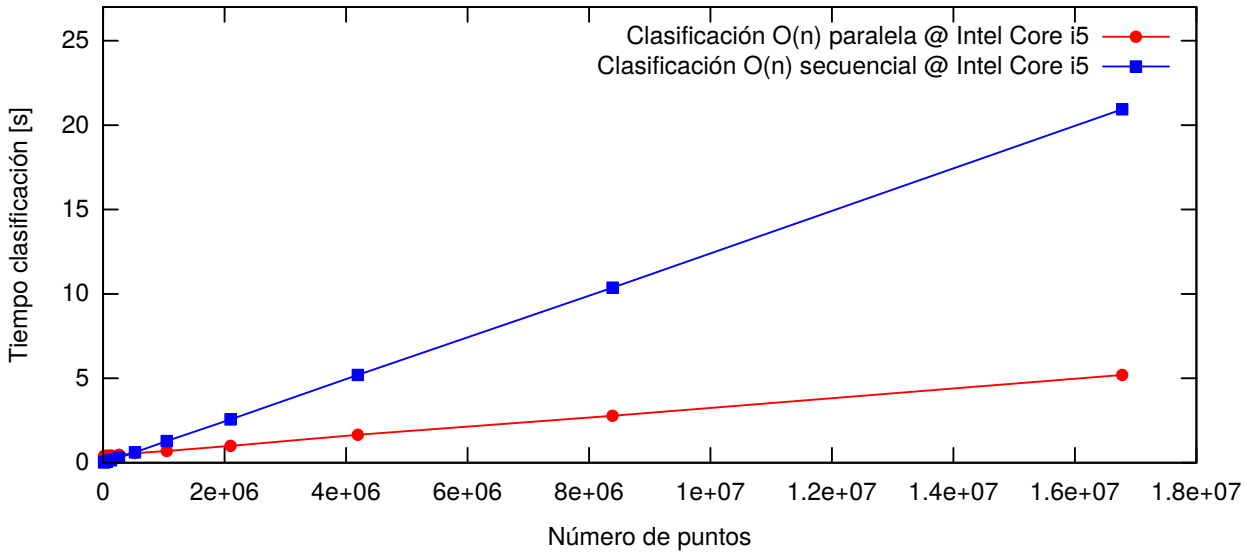


Figura 5.5: *Tiempos de ejecución experimentales de las clasificaciones $O(n)$ desarrolladas. A grandes volúmenes de datos la versión paralela resulta ser la más eficiente.*

Se destaca la inequívoca **tendencia lineal** para ambas curvas mostradas, lo que coincide con la complejidad computacional $O(n)$ prevista. Adicionalmente, se ve que la clasificación paralela es experimentalmente más eficiente que la secuencial y la brecha entre ambas curvas aumenta a medida que lo hace la cantidad de puntos de entrada.

El detalle del gráfico para pequeñas cantidades de puntos, se exhibe en la figura 5.6.

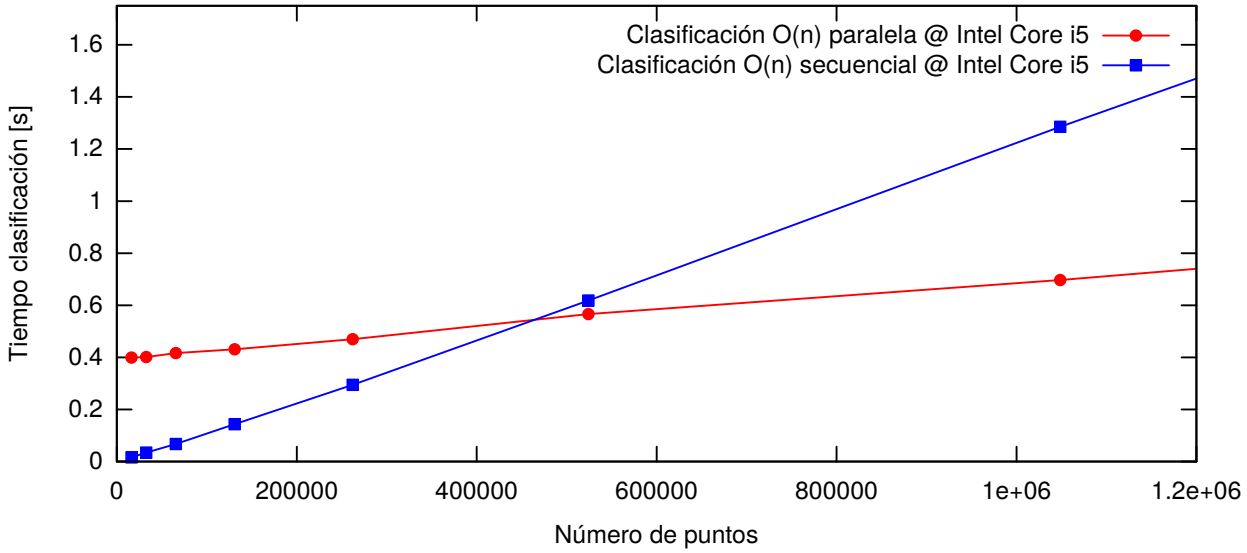


Figura 5.6: *Detalle de la figura 5.5 para examinar tiempos de ejecución para menos de 2^{19} puntos.*

Se advierte que no siempre la clasificación paralela tiene mejor desempeño: para menos de 2^{19} (= 524288) puntos la clasificación lineal secuencial es más rápida. El fenómeno se debe a los preparativos que necesita OpenCL para iniciarse, encolar y ejecutar los *kernels*. Afortunadamente a grandes volúmenes de datos, aquél costo se ve totalmente compensado por la eficiencia de las etapas paralelas.

Para examinar en detalle cuán mejor es la clasificación paralela, se ofrece un gráfico del cociente entre el tiempo de ejecución experimental de la clasificación secuencial y la paralela, en la figura 5.7.

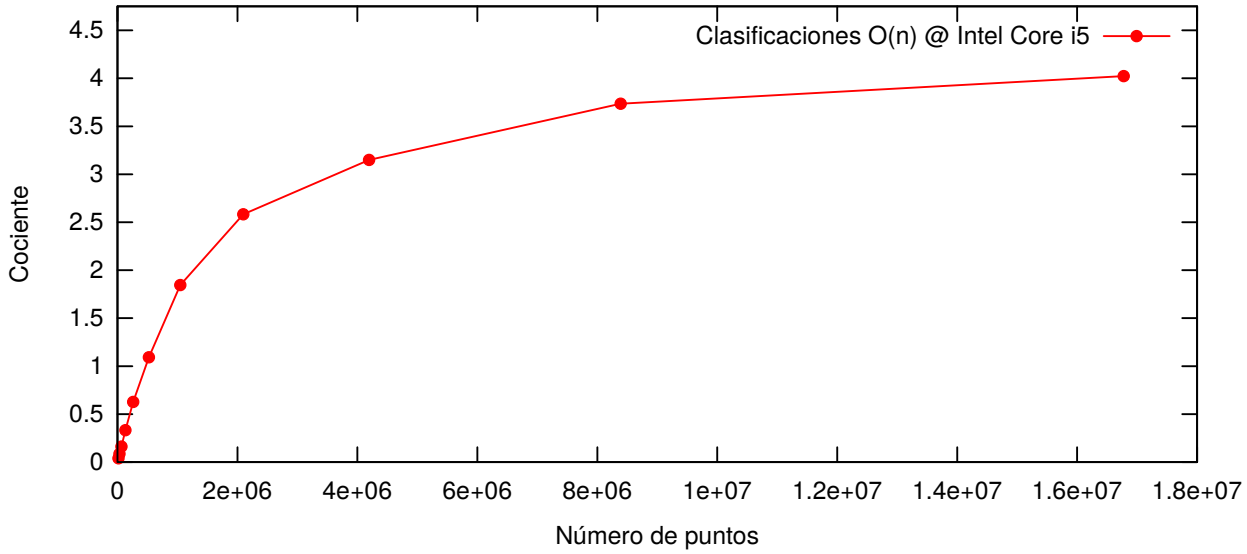


Figura 5.7: *Cociente entre el tiempo de ejecución experimental de la clasificación secuencial y la paralela.*

El gráfico revela que para una cantidad de puntos menor a 2^{19} el cociente es menor que 1, significando que la clasificación secuencial calcula más rápido. Para más de 2^{19} se da la tendencia contraria y para decenas de millones de puntos se acerca a 4, significando que la clasificación paralela **calcula cuatro veces más rápido** en ese orden de magnitud.

Por último, comparando la implementación de las clasificaciones desarrolladas con la de DELFIN Extendido (sin última fase de fusión de vacíos), se tiene la figura 5.8.

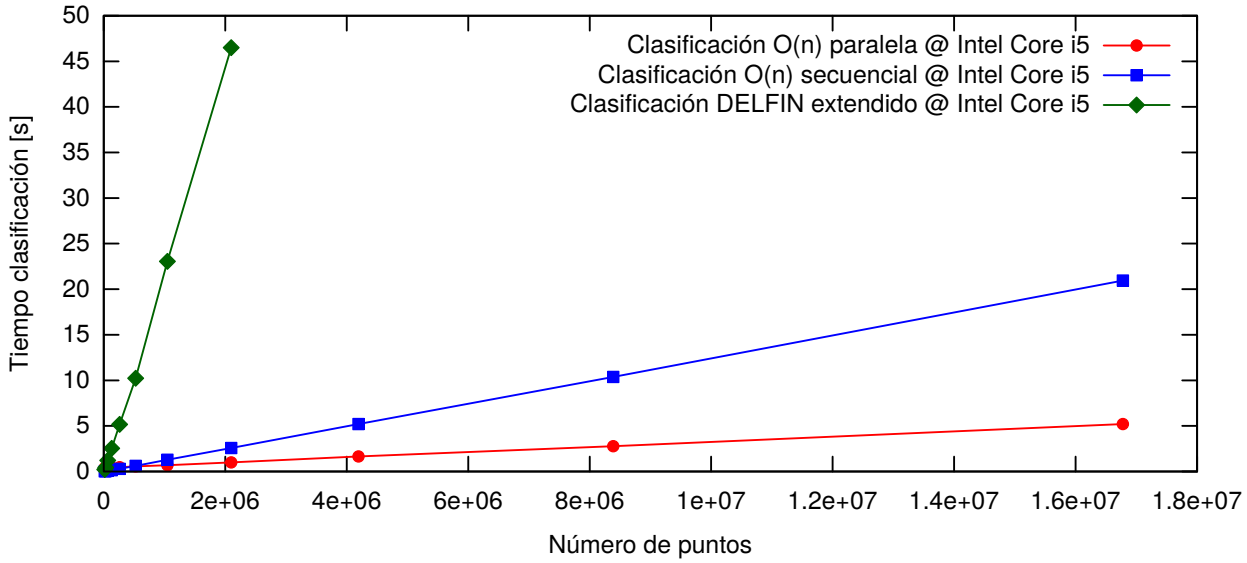


Figura 5.8: *Tiempos de ejecución experimentales de las clasificaciones $O(n)$ y de DELFIN extendido.*

Con aquél gráfico se hace explícito que las clasificaciones $O(n)$ representan una mejora

importante en desempeño respecto de la clasificación de DELFIN Extendido.

Sólo para ilustrar la magnitud del progreso, con 2^{21} (aproximadamente $2 \cdot 10^6$) puntos la clasificación paralela toma $\approx$ 1 segundo y la de DELFIN extendido demora $\approx$ 46 segundos en la máquina en que se hicieron las pruebas.

5.5. Comentarios

Con lo expuesto hasta aquí, se cumple con el objetivo de la memoria: se propone una forma paralela para la fase de clasificación del algoritmo de vacíos poligonales de Hervías et al., adicionalmente teniendo una complejidad computacional inferior a la original y siendo experimentalmente más eficiente que la implementación actual.

Capítulo 6

Clasificación en $O(\log(n)^2)$

Aquí se presentará una clasificación paralela de menor complejidad aún, la cual aprovecha todas las características encontradas en el capítulo 4 para los vacíos poligonales que encuentra el algoritmo original. Se hará uso de los teoremas 4.3, 4.4, 4.5 y 4.6.

6.1. Algoritmo

A las dos ideas fundamentales en que se basa todo el capítulo previo, se añade la existencia de un árbol para cada zona (teorema 4.6).

Tal como se vio, cuando a un grafo de adyacencia se le eliminan los arcos asociados a aristas nomáx-nomáx, quedan componentes conexas disjuntas, pero no todas necesariamente son zonas. Sí lo son aquellas componentes que posean un nodo asociado a un triángulo que tenga arista máx-máx.

Con el grafo ya particionado se puede aplicar la forma en que trabaja el teorema 4.6 para que las componentes que corresponden a zonas se conviertan en árboles de zona. Para ese efecto, según lo discutido, hay que:

- Para cada par de nodos correspondientes a triángulos semilla, eliminar uno de los arcos asociados a la arista máx-máx que comparten (sólo se podrá hacer en zonas).
- Eliminar los arcos asociados a aristas máx-nomáx que apuntan desde el triángulo para el que es no máxima hacia al que es máxima.

Al trabajar en paralelo, se necesita una forma de romper la simetría para poder eliminar uno de los arcos asociados a aristas máx-máx. Suponiendo que los triángulos (y por lo tanto los nodos) tienen un identificador único y comparable con el resto, se puede elegir en base a aquéllos cuál arco eliminar. Así, por ejemplo, se puede decidir eliminar el arco que va desde el triángulo de identificador mayor hacia el de menor, lo cual designaría al último como raíz.

También se mencionó luego de la demostración de la existencia de los árboles de zona que – de haber – debieran removerse arcos asociados a aristas nomáx-nomáx. En esta etapa, cuando el grafo de adyacencia original ya estaba particionado, ya no hay arcos de ese tipo por lo que no es necesario preocuparse por ellos.

La transformación a árbol aplicada sobre las componentes conexas resultantes de la partición mostrada en la figura 5.1b, se muestra en la figura 6.1a.

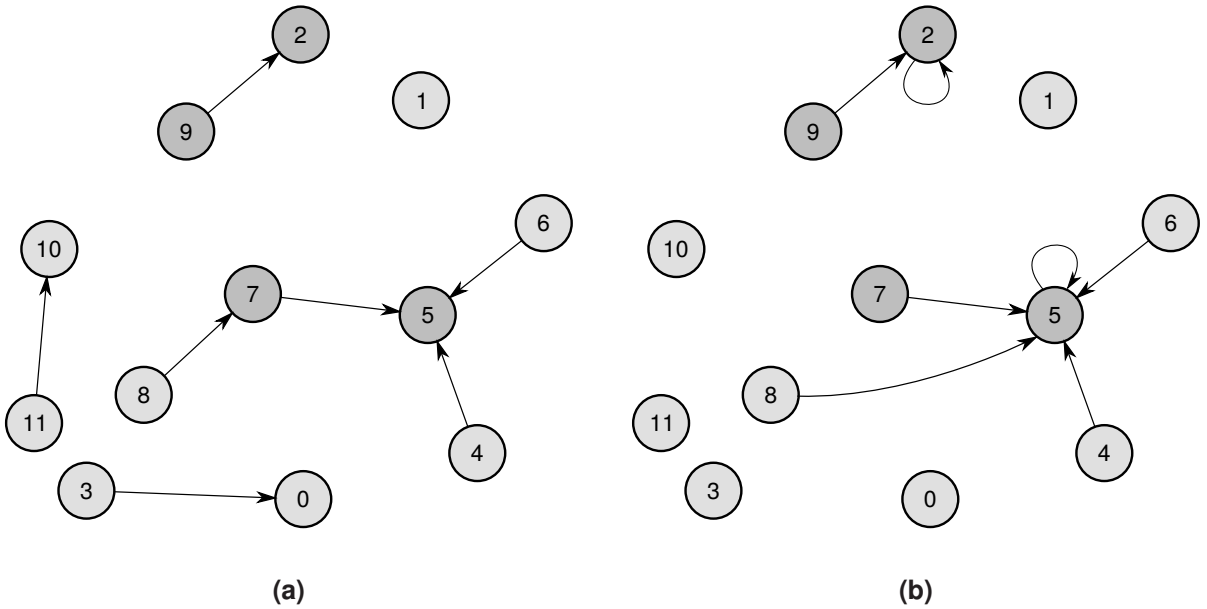


Figura 6.1: ***(a)*** *Transformación a árbol de zona aplicada al grafo mostrado en la figura 5.1b, en que arcos asociados a aristas nomáx-nomáx ya habían sido removidos. Los nodos asociados a triángulos con aristas máx-máx se resaltan con gris oscuro y para ellos se decidió eliminar el arco que iba desde el triángulo de menor identificador al de mayor.* ***(b)*** *Resultado de hacer pointer jumping. En todos los árboles de zona, se asegura que todos sus nodos terminarán apuntando a sus raíces correspondientes.*

La posibilidad de hacer dicha transformación permite prescindir del DFS del capítulo anterior y en su reemplazo hacer *pointer jumping* para conectar los nodos de las componentes que resultan en árboles de zona, a sus respectivas raíces (recuérdese la aplicación mostrada en la subsección 2.2.2, para grafos con grados de salida $\Theta(1)$ y árboles). Previo a la transformación, en las raíces de aquéllos árboles se debe dejar un puntero hacia ellas mismas como "valor" a comunicar. En el resto de los nodos y componentes, se deja como valor a un puntero nulo.

El resultado se ve en la figura 6.1b, en que se aprecia que sólo las componentes que correspondían a zonas resultan con todos sus nodos conectados a sus raíces, mientras que el resto se convierten en nodos aislados. Los nodos pertenecientes a no zonas pasarán a caracterizarse por no tener arcos dirigidos que salgan de ellos, mientras que los que sí están en zonas, tendrán uno solo emergente.

Respecto a la identificación de las componentes, se puede establecer para cada nodo un campo para el efecto. En el caso de las zonas, cada nodo puede copiar el identificador de triángulo desde su respectiva raíz, quedando como su identificador de componente. Para los nodos que están en no zonas se puede establecer un valor de identificación especial.

Por ejemplo, si los triángulos tienen índices enteros iguales o mayores que cero (como la triangulación que genera qdelaunay), se les puedes asignar `-1`.

Hasta el momento se sabe cuáles nodos pertenecen a zonas y cuáles a no zonas. Falta por decidir si las zonas son murallas o vacíos, de acuerdo a la comparación del área que cubren con el `threshold_value`.

Con ese fin, se puede disponer de un arreglo en que están todos los nodos y hacer una ordenación paralela en él, sobre el campo de identificador de componente. De esa forma, todos los nodos que pertenecen a una cierta componente quedarán adyacentes en el arreglo, tal como se ve en la figura 6.2.

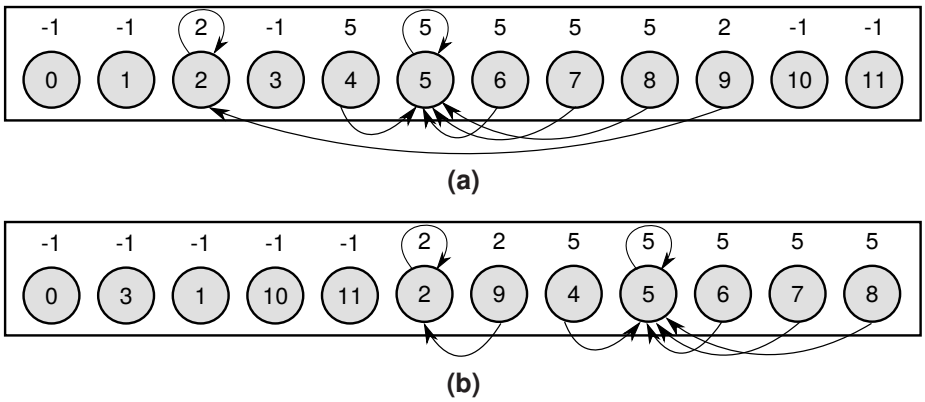


Figura 6.2: *Ordenamiento de arreglo con los nodos del grafo de la figura 6.1b. Encima de cada nodo está el identificador de su componente.* ***(a)*** *Arreglo original a ordenar.* ***(b)*** *Arreglo ordenado por identificadores de componente. Los nodos que comparten un mismo identificador quedan adyacentes, generando segmentos.*

El ordenamiento permite hacer computación segmentada de prefijos para calcular el área total de cada zona, pues se pueden identificar los inicios de segmentos: la primera celda del arreglo siempre iniciará un segmento y el resto de las celdas iniciarán un segmento si el identificador de componente de la celda anterior es distinto.

Según la subsección 2.2.4, la computación se basa en *pointer jumping*, por lo que se necesita de un arreglo de punteros adicional: se crea de modo que su i-ésima celda tenga un puntero hacia el nodo de la $(i + 1)$-ésima celda del arreglo de nodos y la última almacene un puntero nulo. Así se consigue la lista enlazada temporal necesaria.

El cálculo deja las sumas de los segmentos al final de los mismos, los cuales también se pueden identificar: la última celda del arreglo siempre termina un segmento y las otras marcan el fin de uno sólo si el identificador de componente de la celda siguiente es distinto. Así, en cada componente sólo un nodo contendrá el área total de la misma, tal como se ve en la figura 6.3a. Luego, cada uno de los nodos mencionados puede copiar hacia su raíz correspondiente el valor como se ilustra en la figura 6.3b.

De forma similar al capítulo anterior, con el área total en las raíces de las componentes que son zonas, se puede decidir si son vacíos o murallas y escribir en ellas mismas el resultado de la decisión. Finalmente todos los nodos copian hacia sí el tipo de la componente a la que pertenecen, esta vez leyendo concurrentemente desde sus raíces. Todos

los triángulos pertenecientes a zonas quedan exitosamente clasificados.

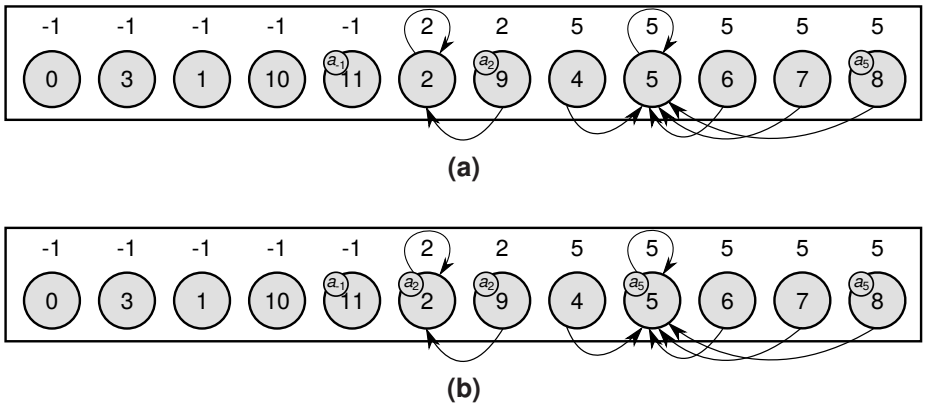


Figura 6.3: ***(a)*** *Arreglo de nodos de la figura 6.2b, luego de la computación. Las sumas requeridas están al final de cada segmento, con a_i indicando el área total de la componente etiquetada con i.* ***(b)*** *Sólo los nodos que terminan un segmento copian hacia sus respectivas raíces su valor de suma de áreas.*

Los pasos descritos se resumen en el algoritmo presentado en el listado 6.1.

```
clasificacion(G=(V,E) : Grafo de adyacencia, threshold : Threshold value)
  Para cada procesador i, en paralelo:
    Marcar arista máxima de triángulo asociado al i-ésimo nodo v ∈ V
  Para cada procesador i, en paralelo:
    Si hay u ∈ V tal que i-ésimo nodo v ∈ V comparte arista máx-máx ∧
      El id. de triángulo de v es menor que el de u:
      Marcar a v como raíz
  Para cada procesador i, en paralelo:
    Para cada e = (v,u) ∈ E que sale del i-ésimo nodo v ∈ V:
      Si e está asociado a arista nomáx-nomáx ∨
        e está asociado a arista máx-nomáx que es no máxima para v ∨
        e está asociado a arista máx-máx con v siendo raíz:
        Eliminar e de E
  Pointer jumping en G
  Para cada procesador i, en paralelo:
    Si hay un sólo arco (v,u) ∈ E que sale del i-ésimo nodo v ∈ V:
      Id. de componente de v ← Id. de triángulo del nodo u
    Sino, si no hay arco que salga de v:
      Id. de componente de v ← -1
  Ordenar en paralelo arreglo de nodos de G por id. de componente
  Para cada procesador i, en paralelo:
    v ← El i-ésimo nodo de V
    Marcar si v inicia un segmento de ids. de componente en arreglo
    Marcar si v termina un segmento de ids. de componente en arreglo
  Computar segmentadamente prefijos en arreglo
  Para cada procesador i, en paralelo:
    Si i-ésimo nodo v ∈ V termina segmento ∧
      Hay un sólo arco (v,u) ∈ E que sale de v:
      Copiar área almacenada en v al nodo u
  Para cada procesador i, en paralelo:
    Si i-ésimo nodo v ∈ V es raíz:
      Si área almacenada en v ≥ threshold:
        Tipo de v ← VACIO
      Sino:
        Tipo de v ← MURALLA
  Para cada procesador i, en paralelo:
    Si hay un sólo arco (v,u) ∈ E que sale del i-ésimo nodo v ∈ V:
      Tipo de v ← Tipo del nodo u
    Sino, si no hay arco que salga de v:
      Tipo de v ← NO_ZONA
  Retornar G
```

Algoritmo 6.1: *Fase de clasificación paralela propuesta en este capítulo.*

6.1.1. Complejidad temporal

Nuevamente téngase una máquina PRAM de $O(n)$ procesadores, uno por cada nodo del grafo que representa a la triangulación de n puntos, y permítanse lecturas concurrentes. Adicionalmente considérese la misma representación de los algoritmos anteriores, con arreglo de nodos y matriz de adyacencia.

Teorema 6.1. *Esta clasificación paralela, en CREW, es* $O(\log(n)^2)$.

Demostración. En forma directa.

De forma similar a lo presentado en el listado 5.1, la etapa de marcado de aristas máximas es $\Theta(1)$.

También es $\Theta(1)$ la etapa marcado de nodos raíz: con lecturas concurrentes cada procesador consulta a los nodos adyacentes al suyo, las aristas máximas y los identificadores de los triángulos asociados. Con esa información decide si hay arista máx-máx y, de haber, usando a los identificadores involucrados el procesador determina si su nodo es raíz.

De igual complejidad es la siguiente fase paralela en que se aplica la transformación a árbol de zona a todo el grafo. Cada procesador lee concurrentemente los identificadores de triángulo y sus respectivas aristas máximas, desde los nodos adyacentes al que tiene a cargo. Teniendo esos datos a disposición cada procesador decide qué arcos eliminar.

Según lo expuesto en la subsección 2.2.2, el *pointer jumping* sobre los $O(n)$ nodos tiene complejidad $O(\log(O(n))) = O(\log(n))$. Para hacerlo, previamente se pueden dejar en las raíces de los árboles de zona punteros hacia sí mismas, en $\Theta(1)$: cada procesador consulta si su nodo es raíz y si lo es, deja el puntero como "valor" a comunicar.

La fase de identificación de componentes a las que pertenecen los triángulos es $\Theta(1)$, pues cada procesador asociado a un nodo de triángulo en zona lee de forma concurrente el identificador desde la raíz y luego lo escribe en su nodo. Cada procesador a cargo de un nodo de triángulo que está en una no zona, simplemente escribe el valor `-1` en su campo de identificador de componente.

Recordando la subsección 2.2.5, el ordenamiento para $O(n)$ elementos se puede lograr con complejidad $O(\log(O(n))^2) = O(\log(n)^2)$.

Pasando a la etapa de marcado de inicios y fines de segmentos, también se hace en $\Theta(1)$. Cada procesador lee en forma concurrente los identificadores de componente desde las celdas adyacentes a la suya. Comparando los identificadores con el propio, determina ambas marcas.

Respecto a la computación segmentada de prefijos para $O(n)$ elementos, según lo presentado en la subsección 2.2.4, es $O(\log(O(n))) = O(\log(n))$. El arreglo temporal de punteros para conformar la lista enlazada necesaria para la operación, se puede construir en $\Theta(1)$: el procesador i-ésimo lee el nodo de la celda $(i + 1)$-ésima del arreglo de nodos, colocando luego un puntero hacia él en la celda i-ésima del nuevo arreglo. El último

procesador sólo coloca un puntero nulo en la última celda de aquél arreglo.

La comunicación del área total de las zonas a sus nodos raíz es $\Theta(1)$, pues cada procesador que posee la información necesaria sólo realiza una escritura. Una vez que los nodos raíz poseen sus áreas totales, la decisión de los tipos de zona es de tiempo constante: se comparan con el *threshold value* y los procesadores a cargo de nodos raíces escriben en sus propios nodos si son vacíos o murallas.

La etapa final nuevamente es $\Theta(1)$ porque cada procesador a cargo de un nodo en zona lee concurrentemente de su raíz para obtener el tipo de zona, mientras que los que están a cargo de nodos en no zonas los marcan finalmente como tales.

La complejidad total de todas las etapas paralelas mencionadas es la suma de términos $\Theta(1)$, $O(\log(n))$ y $O(\log(n)^2)$. Según la comparación asintótica entre las funciones $\log(n)$ y $\log(n)^2$ detallada en el anexo B, se concluye que este algoritmo es $O(\log(n)^2)$. ■

Con el mismo anexo B se tiene que la clasificación presentada en este capítulo tiene una complejidad computacional inferior a la $O(n)$ paralela desarrollada en el anterior, pues muestra que $\log(n)^2 \in o(n)$.

Capítulo 7

Conclusiones y trabajo futuro

7.1. Conclusiones

En esta memoria se realizó un trabajo teórico para encontrar las características de los vacíos poligonales que induce el algoritmo de Hervías et al. sobre una triangulación de Delaunay para n puntos. Lo anterior permitió desarrollar tres nuevas formas de clasificación de triángulos: una secuencial $O(n)$, una paralela $O(n)$ y otra paralela $O(\log(n)^2)$.

Los algoritmos de clasificación $O(n)$ se implementaron exitosamente, resultando ser experimentalmente más eficientes que la clasificación $O(n \log(n))$ de DELFIN extendido, así lográndose a cabalidad el objetivo propuesto para esta memoria.

Los resultados entregados son de relevancia porque son clasificaciones de menor complejidad computacional que la propuesta en el algoritmo original. En caso de que los vacíos poligonales sean la forma de consenso para encontrar vacíos cósmicos, ya existirán formas eficientes para determinarlos.

Parte de lo investigado sobre los vacíos poligonales ha sido incluido en el artículo titulado "*Delaunay based algorithm for finding polygonal voids in planar point sets*", recientemente enviado a la revista científica *Computational Geometry: Theory and Applications* publicada por Elsevier.

7.2. Trabajo futuro

La clasificación $O(\log(n)^2)$ no se implementó debido al tiempo acotado para el desarrollo de una memoria. Se deja propuesta teóricamente con la esperanza de que a futuro se concrete; sería interesante ver su desempeño respecto de los $O(n)$ diseñados y conectarla con algún algoritmo paralelo de triangulación de Delaunay.

También se mencionó que el algoritmo de Hervías et al. es naturalmente extensible a datos volumétricos. Otro posible tema a indagar a futuro sería determinar la viabilidad de extender el comportamiento de las clasificaciones desarrolladas al tratamiento de puntos tridimensionales y con qué complejidad computacional se podría hacer.

Bibliografía

[1] Margaret J. Geller, John P. Huchra. Mapping the universe. *Science*, 246:897–903, Noviembre 1989.

[2] Mark C. Neyrinck. Zobov: a parameter-free void-finding algorithm. *Monthly Notices of the Royal Astronomical Society*, 386:2101–2109, 2008.

[3] Carlos Hervías, Nancy Hitschfeld-Kahler, Luis E. Campusano, Giselle Font. On finding large polygonal voids using Delaunay triangulation: The case of planar point sets. In *Proceedings of the 22nd International Meshing Roundtable*, pages 275–292. Springer, 2014.

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. *Introduction to algorithms*. MIT Press, 2nd edition, 2001.

[5] C. Bradford Barber, David P. Dobkin, Hannu Huhdanpaa. The quickhull algorithm for convex hulls. *ACM Transactions on Mathematical Software*, 22:469–483, 1996.

[6] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest. *Introduction to algorithms*. MIT Press, 1st edition, 1991.

[7] Matt Pharr. *GPU Gems 2*. Pearson Education, 1st edition, 2005.

[8] Janusz Kowalik, Tadeusz Punźniakowski. *Using OpenCL - Programming Massively Parallel Computers*. IOS Press, 1st edition, 2012.

[9] Rodrigo I. Alonso. A Delaunay tessellations based void finder algorithm. Tesis de Magíster en Ciencias mención Computación, Facultad de Ciencias Físicas y Matemáticas, Universidad de Chile, Junio 2016.

Anexos

A. Diagrama de Voronoi

Se construye a partir de un conjunto de puntos en el espacio – llamados “sitios” – y a cada uno se le asocia una región denominada “celda de Voronoi”, en el caso de puntos en el plano.

A partir de cada par de puntos más cercanos entre sí se dibuja un segmento de recta que es equidistante a ambos tal que es perpendicular a la línea que los conecta. En la figura 7.1 se ve un ejemplo.

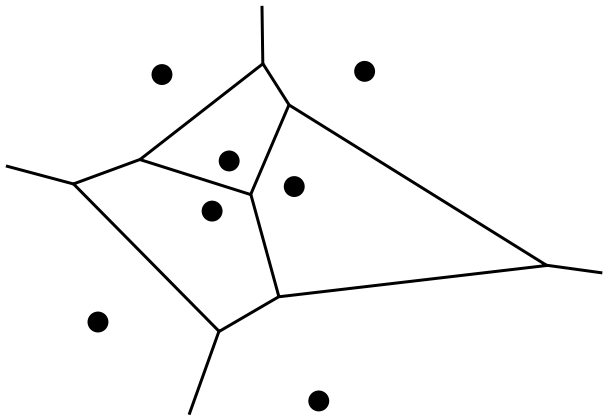


Figura 7.1: *Ejemplo de diagrama de Voronoi para un conjunto de siete puntos.*

Los puntos terminales en que coinciden segmentos se llaman “vértices de Voronoi” y son equidistantes a tres sitios.

B. Comparación asintótica entre $\log(n)$, $\log(n)^2$ y n

El objetivo de este apartado es mostrar la relación que existe entre las complejidades computacionales $\log(n)$, $\log(n)^2$ y n. Para el efecto se utilizará la regla de L’Hôpital y la notación o, las que se explican brevemente a continuación.

La **regla de L'Hôpital** considera dos funciones $f(n)$ y $g(n)$ tal que cumplen:

$$\lim_{n\to+\infty} f(n) = +\infty \wedge \lim_{n\to+\infty} g(n) = +\infty$$

De ser así, el cálculo del límite del cociente $\frac{f(n)}{g(n)}$ con $n \to +\infty$, pasa a ser función de las derivadas correspondientes:

$$\lim_{n\to+\infty} \frac{f(n)}{g(n)} = \lim_{n\to+\infty} \frac{f'(n)}{g'(n)}$$

La nueva expresión para el límite, facilita en muchos casos su cálculo.

Respecto a la **notación *o***, corresponde a una cota superior "estricta". El hecho de que $f(n)$ está estrictamente superiormente acotada por $g(n)$ se denotará como $f(n) \in o(g(n))$. Este tipo de cota se puede demostrar calculando el siguiente límite:

$$\lim_{n\to+\infty} \frac{f(n)}{g(n)} = 0 \Longrightarrow f(n) \in o(g(n))$$

Esta notación será útil a continuación, porque $f(n) \in o(g(n)) \Longrightarrow f(n) \in O(g(n))$.

B.1. Demostración de $\log(n)^2 \in o(n)$

De forma más general, para todo $\alpha, k > 0$ se mostrará que $\ln(n)^k \in o(n^\alpha)$:

$$\begin{aligned}
\lim_{n\to+\infty} \frac{\ln(n)^k}{n^\alpha} &= \left(\lim_{n\to+\infty} \frac{\ln(n)}{n^{\alpha/k}}\right)^k \\
&\overset{L'H}{=} \left(\lim_{n\to+\infty} \frac{1/n}{(\alpha/k)n^{\alpha/k-1}}\right)^k \\
&= \left(\lim_{n\to+\infty} \frac{1}{(\alpha/k)n^{\alpha/k}}\right)^k \\
&= 0^k \\
&= 0
\end{aligned}$$

Lo cual significa que $\ln(n)^k \in o(n^\alpha)$. En particular, asignando $k = 2$ y $\alpha = 1$ se tiene que la función polilogarítmica $\ln(n)^2 \in o(n)$. Como $\ln(n)$ y $\log(n)$ se diferencian sólo por una constante se tiene que $\log(n)^2 \in o(n)$.

B.2. Demostración de $\log(n) \in O(\log(n)^2)$

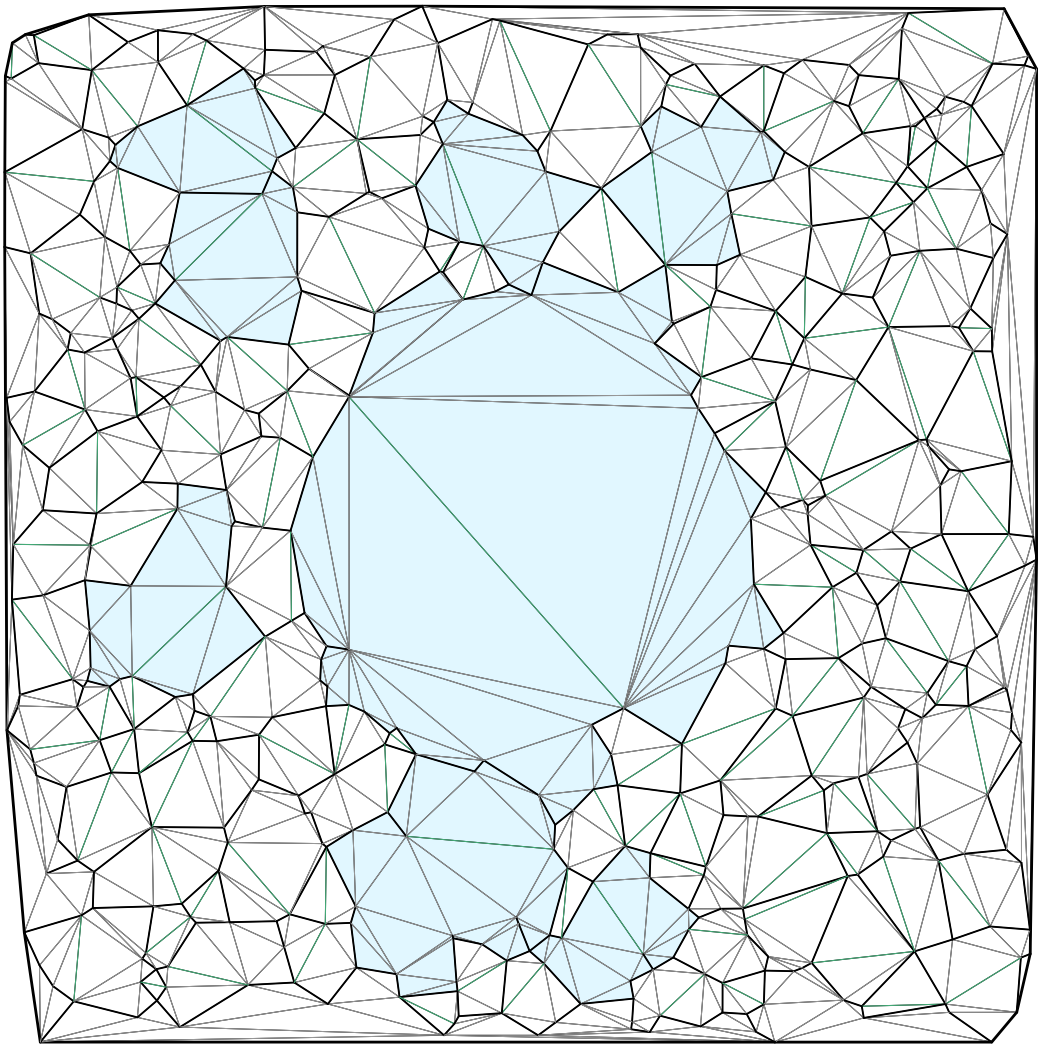
Se mostrará que la función $\log(n)$ está acotada superiormente en forma estricta por la función $\log(n)^2$.

$$\lim_{n \to +\infty} \frac{\log(n)}{\log(n)^2} = \lim_{n \to +\infty} \frac{1}{\log(n)} = 0$$

Pues $\log(n)$ tiende a $+\infty$ cuando n tiende a $+\infty$. Se muestra que $\log(n) \in o(\log(n)^2)$. Finalmente se infiere que $\log(n) \in O(\log(n)^2)$.

C. Ejemplo de herramienta de visualización SVG

Aquí se ofrece una versión de mayor tamaño de la figura 5.4b, para su examen detallado.



Se aprecia cómo los vacíos están rodeados por aristas nomáx-nomáx (de color negro) y cada uno tiene una y sólo una arista máx-máx (de color verde).