Universidad Nacional Autónoma de México

POSGRADO EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN

Implementación de algoritmos en conjuntos de puntos 3, 4-coloreados

T E S I S

que para optar por el grado de

Maestro en Ciencias de la Computación

PRESENTA:
Chávez Jiménez Rodrigo Guadalupe

Tutor Principal:
Dr. Jorge Urrutia Galicia, IMATE

México,CDMX. (Febrero) 2020

Agradecimientos

Dedico este trabajo a mis padres, muchas gracias por apoyarme en todo momento, sin ustedes este camino hubiera sido imposible. Gracias papá por el esfuerzo que hiciste a lo largo de tanto años para educarme, por tantas enseñanzas y por los consejos de vida. Mamá, gracias por siempre apoyarme en todo, has sido el sostén de mi vida en muchos aspectos, gracias por soportarme y por darme tu cariño. Eres la mejor mamá.

Agradezco a todos mis hermanos por su apoyo incondicional y sincero en todo momento, en especial a Sofía por ser tan buena compañía y fiel cómplice, soy afortunado de ser tu hermano.

Quiero agradecer a Jorge por ser un gran tutor, por el tiempo que me dedicó y por sus enseñanzas tanto dentro como fuera del salón de clases. A los doctores David Flores, Juan José Montellano, Adriana Ramírez y Carlos Velarde por ser mis sinodales, por el tiempo que dedicaron a este trabajo y sus valiosos comentarios.

Al "Team Jorge"por hacer tan amenas las tardes de trabajo en la oficina, tan divertidos como productivos los talleres de investigación y por aquel miércoles de pan. En particular a Alma por ser tan buena roomie, por introducirme al mundo de los viajes académicos y por mostrarme tu increíble actitud ante la vida.

Reconozco la labor excelente y el gran apoyo que me han mostrado siempre Lulú, Cecilia y Amalia, sin ustedes los tramites burocráticos hubieran sido mil veces mas difíciles.

A mis amistades de años, Alejandra, gracias por siempre creer en mí y por acompañarme todos estos años. Jennifer, tus revisiones y correcciones fueron de gran ayuda. Agradezco especialmente a Ferat por todos los consejos tan educados que me has dado.

A CONACYT por ser benefactor de mis estudios de maestría.

Finalmente, gracias a todos los que directa e indirectamente han sido parte de esta historia en la búsqueda de mi maestría.

¡Muchas gracias a todos!
Rodrigo.

Índice general

Índice de figuras

1. Introducción y trabajo previo

1.1. Introducción

La Geometría Computacional es una rama de las Ciencias de la Computación con mayor crecimiento en los últimos años. Su principal objetivo es el diseño y análisis de algoritmos para resolver, de manera óptima, problemas que pueden ser expresados en términos geométricos. Los resultados de la Geometría Computacional son ampliamente usados en áreas como: Sistemas de Información Geográfica, Planeación de Movimiento, Diseño de Circuitos Integrados, Ingeniería Asistida por Computadora, entre otros; por lo que la implementación de los algoritmos desarrollados en esta área son de especial relevancia. Últimamente, la comunidad científica del área ha mostrado interés en problemas relacionados con la existencia de objetos geométricos definidos en conjuntos de puntos coloreados y en algoritmos para encontrar tales objetos. En particular, Arévalo *et al.* [11] diseñaron un algoritmo de complejidad $O(n^2)$ para determinar si en un conjunto de puntos 4-coloreado existe un cuadrilátero heterocromático convexo. La resolución de éste último problema señaló la dirección para el abordamiento de nuevas incógnitas, entre ellas, el diseñar un algoritmo para buscar cuadriláteros heterocromáticos convexos que además cumplan ser vacíos. Sin embargo, atacar esta última línea de investigación, requiere de herramientas que automaticen la comprobación de resultados intermedios, por lo que se necesita poder contar con una implementación del algoritmo propuesto por Arévalo *et al.*. El presente trabajo lleva a cabo esta tarea, mediante la programación de primitivas geométricas, permitiendo así el esclarecimiento de esta interrogante abierta y ayudando en el avance de la investigación en Geometría Computacional.

Nuestro trabajo es de especial importancia si ahora se investigan las variantes donde se buscan cuadriláteros heterocromáticos vacíos ya que nuestra implementación además de reportar la existencia puede ser fácilmente modificada para reportar todos los cuadriláteros que cumplan la condición que busca el algoritmo junto con las regiones donde puedan encontrarse los puntos que generen los convexos vacíos y de está manera acotar el espació de búsqueda así como la cantidad de casos a estudiar.

Es natural preguntarse por estos convexos ahora de área máxima y mínima, de la misma manera que en la variante de convexos vacíos nuestra implementación puede ayudar a acotar regiones de interés. Justificamos esto con una implementación del algoritmo que reporta los triángulos heterocromáticos de área máxima y mínima utilizando las mismas técnicas.

A pesar de que existan ya bibliotecas para la manipulación de objetos geométricos

decidimos nosotros implementar estos para así obtener un control granular el cada paso de los programas. Aunque cabe mencionar que fácilmente se puede traducir nuestra implementación para utilizar alguna de estas bibliotecas.

1.1.1. Estructura del documento

El presente trabajo está enfocado a la implementación de primitivas geométricas relacionadas con polígonos convexos, ordenación angular respecto a un punto, trasformaciones duales sobre conjuntos de puntos coloreados.

El capitulo 2 está enfocado en la descripción de preliminares.

El capitulo 3 habla de las rectas duales de un conjunto de puntos, de la subdivisión del plano que estás rectas inducen y de la representación eficiente de esta subdivisión.

En el capitulo 4 describimos las rutinas para resolver el problema de encontrar el triángulo heterocromático de mayor y menor área en un conjunto de puntos 3-coloreado.

En el capitulo 5 describimos rutinas e implementación para el problema de decidir si un conjunto de puntos 4-coloreado contiene un cuadrilátero heterocromático convexo.

2. Preliminares

2.1. Conjuntos de puntos

Definición 2.1.1. *Decimos que $S \subset \mathbb{R}^2$ es un conjunto convexo si para cada dos elementos $x, y \in S$ el segmento $\overline{xy} \subset S$.*

Definición 2.1.2. *El* cierre convexo *de un conjunto de puntos S, denotado $CH(S)$, es la frontera de la intersección de todas las regiones convexas que contienen a S.*

Definición 2.1.3. *Un conjunto de puntos S está en posición convexa si cada punto de S pertenece al cierre convexo de S.*

2.2. Polígonos

Definición 2.2.1. *Un polígono P es una sucesión ordenada de n puntos $v_1, v_2, \ldots, v_n$ en el plano, con $n \geq 3$, llamados vértices, junto con el conjunto de segmentos de línea que une v_i a v_{i+1} para $i = 1, 2, .., n-1$, y a v_n con v_1, llamado conjunto de aristas de P.*

Definición 2.2.2. *Llamaremos m-ágono a un polígono de m vértices.*

Se dice que P es un polígono *simple* si ningún par de aristas no consecutivas se intersecan. Un polígono simple divide al plano en dos regiones, una región no acotada llamada *exterior* y una región acotada llamada *interior*. La colección de vértices y aristas de P es llamada *frontera* de P, y será denotada como ∂P. Por simplicidad de aquí en adelante usaremos el término polígono para denotar un polígono simple junto con su interior.

Un vértice v de P es llamado *convexo* si el ángulo interno de P en v es menor a π, y *cóncavo* si el ángulo interno de P en v es mayor a π.

Definición 2.2.3. *Un polígono P es* convexo *si para cualquier par de puntos $a, b \in P$ el segmento de línea $\overline{ab}$ está totalmente contenido en P.*

2.3. Gráficas

Definición 2.3.1. *Una* gráfica *G es una tupla $(V(G), E(G))$ que consiste de un conjunto $V(G)$ de* vértices *y un conjunto $E(G)$, ajeno a $V(G)$, de aristas, junto con una* función de incidencia *Φ_G, la cual asocia a cada arista de G con un par* vértices *de G.*

Si en una gráfica G tenemos que una arista e y un par de vértices u, v tales que $\Phi_g(e) = \{u, v\}$, entonces decimos que e *une* a u y v, los vértices u y v son llamados los *extremos* de e. Por simplicidad nos referiremos a la arista que une al par de vértices $\{u, v\}$ como uv.

El número de vértices de una gráfica G es llamado el *orden* de G mientras que el número de aristas es llamado el *tamaño* de G. El número de aristas que inciden en un vértice v es llamado el *grado* de v.

Se dice que los extremos de una arista *inciden* en la arista y viceversa. Dos vértices que inciden en la misma arista, o dos aristas que inciden en el mismo vértice son llamados adyacentes. Dos vértices adyacentes distintos son llamados *vecinos*.

Una gráfica *aplanable* es aquella que puede ser representada en el plano de modo tal que sus aristas (representadas por curvas) solo se encuentren en sus extremos comunes (representados por puntos), a tal representación se le conoce como *encaje plano*. A una gráfica aplanable junto con su encaje plano se le conoce como gráfica *plana*.

2.4. Coloraciones

Definición 2.4.1. *Una* coloración *es una función,* $c : S \to \{c_1, ..., c_k\}$. *A los elementos de la imagen se les conoce como* colores. *La clase cromática* C_i *es la preimagen del color* c_i.

A una coloración con k colores se le conoce como k-coloración, también se dice que el conjunto está k-coloreado. En particular hablaremos de conjuntos k-coloreados de puntos en el plano.

Definición 2.4.2. *Sea* S *un conjunto coloreado de puntos en el plano. Decimos que un subconjunto* $T \subset S$ *es:*

- monocromático *si todos sus elementos son del mismo color.*
- heterocromático *o* arcoíris *si todos sus elementos son de color distinto.*

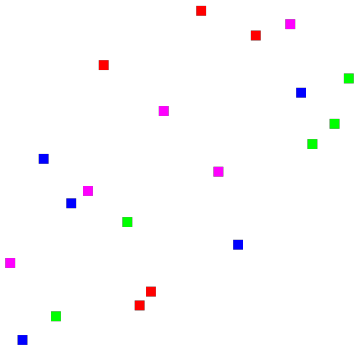


Figura 2.1: Conjunto de puntos 4-coloreado.

2.5. Trabajo previo

Ya hemos mencionado el concepto de convexidad y cuándo un conjunto de puntos está en posición convexa. Sobre esto existe un resultado clásico de 1935, dado por Paul Erdös y George Szekeres[8]. Este resultado al mismo tiempo se base en otro dado por Esther Klein, el cual mostramos en la siguiente proposición.

Para toda configuración de 5 puntos en posición general en el plano, es posible encontrar 4 de ellos que estén en posición convexa.

Así mismo, fue planteada también por Klein la siguiente pregunta en busca de una generalización de este resultado.

Problema 2.5.1. *Dado un entero m ¿Existe un valor $f(m)$ que garantice que todo conjunto de $f(m)$ puntos en el plano contiene m de ellos en posición convexa?*

Erdös y Szekeres mostraron que la respuesta es afirmativa.

Teorema 2.5.1 (Erdös-Szekeres)**.** *Para cada entero positivo $m > 3$, existe un mínimo entero $f(m)$ tal que todo n-conjunto, con $n \geq f(m)$, contiene un m-subconjunto de puntos en posición convexa.*

Notemos que hacemos mención a m puntos en posición convexa, o a un m-ágono convexo de manera indistinta. Variantes coloreadas de estos problemas fueron estudiadas por primera vez por Devillers, Hurtado, Károlyi y Seara en [6] dando inicio a una nueva rama de investigación sobre la existencia de convexos heterecromáticos, monocromáticos y policromáticos en conjuntos de puntos coloreados. Consideremos un n-conjunto, S, de puntos en el plano en posición general, coloreados con 4 colores. Por el resultado de Erdös-Szekeres, si $n \geq 5$, S necesariamente contiene un cuadrilátero convexo; y de la misma manera, si alguna una de las clases cromáticas tiene al menos 5 puntos entonces contendrá un cuadrilátero monocromático. Sin embargo, en [9] mostraron que sin importar el valor de n, S no necesariamente contiene un cuadrilátero convexo heterocromático incluso cuando las clases cromáticas tengan la misma cardinalidad. Por esta razón se vuelve interesante diseñar algoritmos para su detección.

Dado un conjunto S de puntos en el plano, varios artículos han estudiado el problema de encontrar un m-ágono de área extrema cuyos vértices son elementos de S. Chazelle *et al.* [4] y Edelsbrunner *et al.* [7] probaron que un triángulo de área mínima contenido en S puede ser obtenido en tiempo y espacio $O(n^2)$ usando dualidad y arreglos de líneas.

Consideramos el problema de encontrar m-ágonos de área máxima en S. Cordes [5] probó que el m-ágono de área máxima en S está determinado por vértices del cierre convexo de S, siempre que se cumpla que m es a lo más la cantidad de vértices en la frontera del cierre. Así, tiene sentido estudiar el problema de encontrar el m-ágono de área máxima inscrito en un polígono convexo para valores pequeños de m. Recientemente Jin [10] y Kallus [12] presentaron algoritmos correctos de tiempo lineal para encontrar el triángulo de área máxima contenido en un polígono, Rote [16] probó que el cuadrilátero de área máxima contenido en un polígono puede ser encontrado también en tiempo lineal.

Otros problemas en conjuntos de puntos en el plano se relacionan con determinar la existencia de ciertas estructuras en tal conjunto. El teorema de Erdös-Szekeres nos dice que para cada entero positivo $m > 3$ existe otro entero positivo $f(m)$ tal que para todo conjunto de al menos $f(m)$ puntos en el plano, este contiene los vértices de un m-ágono convexo. En [6] Devillers *et al.* estudiaron algunas variantes del problema de Erdös-Szekeres en conjuntos de puntos coloreados. Las variantes estudiadas por Devillers *et al.* incluyen determinar la máxima cantidad de m-ágonos convexos monocromáticos vacíos contenidos en un conjunto de puntos k-coloreado, para algún valor de k y m. Aichholzer *et al.* [1] probaron que cada conjunto de puntos 2-coloreado con n elementos tiene $\Omega(n^{5/4})$ triángulos monocromáticos vacíos. Esta cota fue mejorada a $\Omega(n^{4/3})$ por Pach y Tóth en [14]. Recientemente, Fabila Monroy *et al.* [9] probaron que cualquier conjunto de puntos k-coloreado con al menos k puntos en cada clase cromática contiene $\Theta(k^3)$ triángulos heterocromáticos vacíos. Además dieron un ejemplo de un conjunto de puntos sin cuadriláteros heterocromáticos vacíos no necesariamente convexos. En [11] Arévalo *et al.* dieron un algoritmo de tiempo y espacio $O(n^2)$ para detectar la existencia de un cuadrilátero heterocromático convexo en un conjunto de puntos 4-coloreado. Además utilizando las mismas técnicas diseñaron un algoritmo de tiempo y espacio $O(n^2)$ para encontrar el triángulo heterocromático de mayor y menor área en un conjunto 3-coloreado. En esta tesis abordamos de manera pragmática los resultados en [11] dando una implementación de dicho algoritmo.

2.6. Bibliotecas de algoritmos en geometría computacional

Existen ya bibliotecas para el manejo de objetos geométricos en diversos lenguajes de programación. En el apéndice A mencionamos algunas de ellas, son dos de estas bibliotecas las que sobresalen tanto en popularidad como en el número de algoritmos que implementan, `CGAL` y `LEDA`. Mencionamos estas dos porque cuentan con estructuras de datos y algoritmos que desarrollamos en nuestra implementación, como la representación de arreglos de líneas, polígonos y cierres convexos. A pesar de existir estas bibliotecas optamos por implementar nuestros algoritmos y estructuras de datos para tener un mayor control sobre los cambios que realizamos a algoritmos clásicos como el escaneo de Graham para calcular el cierre convexo. De cualquier forma se puede traducir de manera relativamente sencilla nuestra implementación a una de estas bibliotecas, por ejemplo, CGAL pone a disposición un constructor para puntos en $\mathbb{R}^2$,

```
Point_2 (double x, double y)
```

Nosotros extendemos la funcionalidad de un punto para además guardar su color, su recta dual, y el orden angular de los otros puntos respecto a él.

```
class Point2D {
    float x, y;
    Linea linea;
    Color color;
    ArrayList<Point2D> orden;
}
```

Listado 2.1: Clase que representa un punto en $\mathbb{R}^2$.

Y así como este podríamos dar más ejemplos de como extender las estructuras para acomodarse a bibliotecas ya existentes.

3. DCEL

[3] Antes de que podamos usar algoritmos para calcular intersecciones de líneas y las subdivisiones del plano que estas generan, debemos desarrollar un representación adecuada de la subdivisión. Guardar la subdivisión como una colección de segmentos de línea no es buena idea porque operaciones como reportar la frontera de una región sería complicado. Es mejor incorporar información estructural y topológica: cuáles segmentos acotan a una región dada, cuáles regiones son adyacentes, etc.

Las divisiones que consideramos son *subdivisiones planas* inducidas por encajes planos de gráficas. El encaje de un nodo de una gráfica es llamado *vértice* y el encaje de un *arco* es llamado *arista*. Solamente consideramos encajes en que las aristas son segmentos de línea recta. Una *cara* de la subdivisión es un subconjunto conexo maximal del plano que no contiene un punto sobre alguna arista o vértice. Así una cara es una región poligonal cuya frontera está formada por aristas y vértices de la subdivisión. La *complejidad* de una subdivisión está definida como la suma del número de vértices más el número de aristas y el número de caras del que consiste. Si un vértice es el extremo de una arista, entonces decimos que el vértice y la arista son *incidentes*. Análogamente, una cara y una arista en su frontera o una cara y un vértice en su frontera son incidentes.

Es razonable que queramos caminar al rededor de la frontera de una cara dada, o que accedamos a una cara desde otra cara si nos es dada la arista en común entre ellas. Otra operación que nos será útil es la de visitar todas las aristas al rededor de un vértice dado. La representación que vamos a discutir, introducida y estudiada en [13] y [15], llamada lista de aristas doblemente conexa, *DCEL* (por sus siglas en inglés doubly-connected edge list), soporta estas operaciones.

Una DCEL contiene un registro para cada cara, arista y vértice de la subdivisión. Además de la información geométrica y topológica, cada registro puede contener información adicional como sea necesaria. Por ejemplo, si una subdivisión representa un mapa de vegetación, la DCEL guardaría en su registro de caras la vegetación correspondiente a cada región. La información adicional es también llamada *información de atributo*. La información geométrica y topológica es lo que nos permitirá realizar las operaciones antes mencionadas. Para poder caminar al rededor de un cara en orden antihorario guardamos un apuntador por cada arista a la siguiente en la cara. También podría resultar útil poder caminar en el sentido horario entonces también guardamos un apuntador a la arista previa en la cara. Una arista usualmente acota dos caras, así que necesitamos dos pares de apuntadores para ello. Es conveniente ver a los diferentes lados de una arista como dos distintas *medias aristas* (o *half edges*). Esto significa que cada media arista acota solamente una cara. Las dos medias aristas obtenidas de una

arista se dice que son *gemelas*. Al definir la siguiente media arista de una media arista dada con respecto al recorrido antihorario de una cara, se induce una orientación en las medias aristas: es orientada de tal manera que la cara que acota está a la derecha de un observador caminando sobre la arista. Consecuencia de esta orientación es que podamos hablar de un *origen* y un *destino* de una media arista. Si la arista $\overrightarrow{e}$ tiene a v como su origen y a w como su destino, entonces su gemela $Twin(\overrightarrow{e})$ tiene a w como su origen y a v como su destino. Para alcanzar la frontera de una cara solamente necesitamos guardar un apuntador en el registro de la cara a una media arista arbitraria que la acote. Empezando por esta arista, podemos pasar de la media arista a la siguiente al rededor de la cara acotada por la arista.

En resumen. Una DCEL consiste de tres colecciones de registros: una para los vértices, otra para las caras, y una para las medias aristas. Estos registros guardan la siguiente información geométrica y topológica:

- El registro de vértices guarda de un vértice v sus coordenadas en un campo llamado $Coordenadas(v)$. También guarda un apuntador $AristaAdyacente(v)$ a una media arista arbitraria que tiene a v como su origen.

- El registro de caras guarda de una cara f un apuntador $ComponenteExterna(f)$ a alguna media arista que esté en su frontera exterior y un apuntador $ComponenteInterna(f)$ de alguna media arista que esté en su frontera interior en el caso de que la cara sea una región no acotada.

- El registro de medias aristas guarda de una media arista $\overrightarrow{e}$ un apuntador $Origin(\overrightarrow{e})$ de su vértice de origen, un apuntador $Twin(\overrightarrow{e})$ a su media arista gemela, y un apuntador $IncidentFace(\overrightarrow{e})$ a la cara que acota. Observemos que no hace falta guardar el destino de la media arista porque es igual a $Origin(Twin(\overrightarrow{e}))$. El origen es escogido tal que $IncidentFace(\overrightarrow{e})$ este a la izquierda de $\overrightarrow{e}$ cuando es recorrida de origen a destino. El registro también guarda apuntadores $Next(\overrightarrow{e})$ y $Prev(\overrightarrow{e})$ a las medias aristas siguiente y previa en la frontera de $IncidentFace(\overrightarrow{e})$ recorrida en sentido antihorario. Así $Next(\overrightarrow{e})$ es la única media arista en la frontera de $IncidentFace(\overrightarrow{e})$ que tiene el destino de $\overrightarrow{e}$ como su origen, y $Prev(\overrightarrow{e})$ es una única media arista en la frontera de $IncidentFace(\overrightarrow{e})$ que tiene a $Origin(\overrightarrow{e})$ como su destino.

Una cantidad constante de información es usada por cada vértice, arista y cara, concluimos entonces que la cantidad de espacio requerida es lineal en la complejidad de la subdivisión.

3.1. Implementación de una DCEL

Se implementan como un conjunto de clases con sus distintos apuntadores. Un vértice es representado con flotantes para sus coordenadas (x, y).

```
class Vertex {
    float x;
    float y;
    HalfEdge incident;
}
```

Listado 3.1: Clase que representa un vértice.

Las aristas se representan con dos medias aristas con orientaciones opuestas.

```
class HalfEdge {
    Vertex origin;
    Face face;
    HalfEdge next;
    HalfEdge prev;
    HalfEdge twin;
    Linea line;
}
```

Listado 3.2: Clase que representa una media arista.

Las caras guardan dos apuntadores, uno apunta a una (cualquier) media arista en la frontera interna de la cara y otro apunta una (cualquier) media arista en la frontera externa de la cara.

```
class Face {
    HalfEdge outer;
    HalfEdge inner;
}
```

Listado 3.3: Clase que representa una cara.

3.2. Dualidad y arreglos

3.2.1. Dualidad

En [11] trabajan con la trasformación dual estándar entre puntos y líneas en el plano, así como la subdivisión del plano inducida por un conjunto de líneas. Estos conceptos, que introducimos de manera breve, son estudiados extensivamente en [7] y [4]. Un punto en el plano tiene 2 parámetros: la coordenada x y la coordenada y. Una línea (no vertical) en el plano también tiene dos parámetros: la pendiente y su intersección con el eje y. Por lo tanto podemos mapear uno a uno un conjunto de puntos a un conjunto de líneas y viceversa. Incluso podemos hacer esto de manera que ciertas propiedades del conjunto de puntos se trasladen a otras ciertas propiedades del

conjunto de líneas. Por ejemplo, tres puntos sobre una línea se convierten en tres líneas intersecadas en un mismo punto. Diferentes mapeos que consiguen esto son posibles, estos mapeos son llamados *transformaciones duales*. La imagen de un objeto bajo una transformación dual es llamado el *dual* del objeto. Una simple transformación es la siguiente.
Sea $p := (p_x, p_y)$ un punto en el plano. El dual de p, denotado por p^* es la línea definida como

$$p^* := (y = p_x x - p_y).$$

El dual de la línea $\ell : y = mx + b$ es el punto p tal que $p^* = \ell$. En otras palabras,

$$\ell^* := (m, -b).$$

La transformación dual no está definida para líneas verticales. En la mayoría de los casos las líneas verticales puede ser manejada por separado, así que esto no es problema. Otra solución es rotar el plano tal que no hayan líneas verticales.

Decimos que que la transformación dual mapea objetos del *plano primal* a objetos del *plano dual*. Ciertas propiedades que se cumplen en el plano primal se mantienen en el plano dual:

Observación 3.2.1. *Sea p un punto en el plano y sea ℓ un línea no vertical en el plano. La transformación dual $o \mapsto o^*$ tiene las siguientes propiedades.*

- *Preserva la incidencia: $p \in \ell \iff \ell^* \in p^*$*
- *Preserva el orden: p esta encima de $\ell \iff \ell^*$ encima de p^**

La transformación dual también puede ser aplicada a otros objetos además de puntos y líneas. ¿Qué sería el dual del segmento $s := \overline{pq}$?. Una opción lógica para s^* es la unión de los duales de todos los puntos en s. Lo que obtenemos es un conjunto infinito de líneas. Todos los punto en s son colineales, así que todas las líneas duales pasan sobre un mismo punto. La unión forma una *cuña doble* acotada por los duales de los extremos de s. Las líneas duales de los extremos de s definen dos cuñas dobles, una de izquierda a derecha y otra de arriba a abajo, s^* es la cuña de derecha a izquierda. La figura 3.1 muestra el dual de un segmento s. También muestra una línea ℓ que interseca a s, cuyo dual ℓ^* yace en s^*. Cualquier línea que interseque al segmento s debe tener a p o q arriba y el otro punto por debajo, entonces el dual de tales líneas esta en s^* por la precedencia de orden de la transformación dual.

Cuando pensamos en el dual nos podríamos preguntar cómo este nos puede ser útil. Si podemos resolver un problema en el plano dual, podríamos resolver el problema en el primal también, mediante la solución del problema en el dual imitada en el primal. Después de todo, el problema dual y primal son esencialmente el mismo. Aún así, transformar un problema al plano dual tiene una ventaja importante: provee de una nueva perspectiva. Mirar un problema de un ángulo distinto puede dar la visión necesaria para resolverlo.

Ahora necesitamos una representación para los puntos y sus rectas duales. El punto está definido en la sección 2.6. Las coordenadas (x, y), al igual que los vértices, están

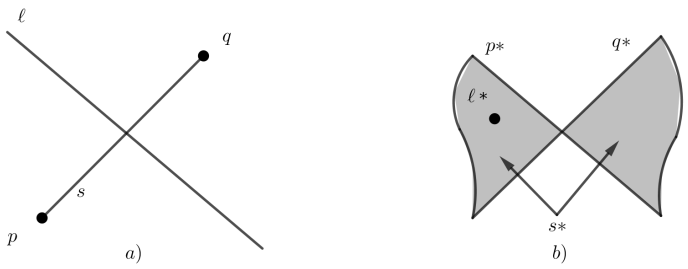


Figura 3.1: Dual de un segmento de recta.

representadas con flotantes, en nuestro problema los puntos son coloreados por lo que la clase Point2D tiene como atributo `Color` y el atributo `Linea` es la representación de su recta dual. Se incluye como atributo una lista de puntos que será útil después para guardar un orden angular del resto de los puntos.

```
class Linea extends HalfEdge{
    HalfEdge primerArista;
    Point2D puntoPrimal;
}
```

Listado 3.4: Clase que representa la recta dual de un punto.

La clase línea hereda los atributos de `HalfEdge` (los métodos para dibujar una línea en pantalla requieren un punto inicial y un punto final) además declara una media arista que será utilizada para representar la arista que interseca por la izquierda a un rectángulo isotético y guarda un apuntador a su punto primal.

```
public Linea calcularDual(Point2D punto) {
    float y1;
    float y2;
    float x1 = rWidth / 2, x2 = -(rWidth / 2);
    //rWidth es el ancho del marco
    y1 = punto.x * x1 - punto.y;
    y2 = punto.x * x2 - punto.y;
    DCEList dcel = new DCEList();
    return dcel.crearLinea(x1, y1, x2, y2);
}
```

Listado 3.5: Dual de un punto.

3.2.2. Arreglos de Línea

Sea L un conjunto de líneas en el plano. El conjunto L induce una subdivisión del plano que consiste de vértices, aristas y caras. Algunas de las aristas y caras son no acotadas. Esta subdivisión es usualmente referida como *arreglo* inducido por L y es denotado por $A(L)$. Un arreglo es llamado *simple* si no existen tres líneas que pasen

sobre un mismo punto y ningún par de líneas son paralelas. La *complejidad* del arreglo es el número total de vértices, aristas y caras del arreglo. Arreglos de líneas y sus contrapartes de dimensiones más altas aparecen repetidamente en geometría computacional. Con frecuencia un problema definido en conjuntos de puntos es dualizado y convertido en un problema en arreglos de líneas. Esto es porque la estructura en un arreglo de líneas es más aparente que en el conjunto de puntos. Por ejemplo una línea sobre un par de puntos en el primal se convierte en un vértice en el arreglo dual. La estructura extra en un arreglo no es gratis: La construcción de un arreglo es costoso tanto en tiempo como en espacio porque la complejidad combinatoria es alta.

Teorema 3.2.1. *Sea L un conjunto de n líneas en el plano y sea $A(L)$ el arreglo inducido por L.*

- *El número de vértices de $A(L)$ es a lo más $n(n-1)/2$.*
- *El número de aristas de $A(L)$ es a lo más n^2.*
- *El número de caras de $A(L)$ es a lo más $n^2/2 + n/2 + 1$.*

La igualdad se cumple si el arreglo es simple.

Entonces el arreglo $A(L)$ inducido por el conjunto L de líneas es una subdivisión planar de a lo más complejidad cuadrática. La DCEL es adecuada para guardar el arreglo, con esta representación podemos eficientemente listar las aristas de un cara dada, pasar de una cara a otra que sea adyacente, etc. Sin embargo una DCEL solamente puede guardar aristas acotadas y un arreglo es de rectas no acotadas. Por lo tanto colocamos una rectángulo que encierre las intersecciones del arreglo, esto es, un *marco* que contiene todos los vértices del arreglo en su interior. La subdivisión definida por el marco, más la parte del arreglo dentro de este, tiene solamente aristas acotadas y puede ser guardada usando una DCEL.

Ahora ¿Cómo podremos construir esta DCEL? Se nos puede ocurrir hacer un barrido de línea, un algoritmo bien conocido para la calcular la intersección de segmentos en el plano. De hecho no es muy difícil adaptar alguno de estos algoritmos para calcular el arreglo $A(L)$. Como el número de puntos de intersección es cuadrático el algoritmo se ejecutaría en tiempo $O(n^2 \log n)$. Que no es malo pero tampoco es óptimo. Mejor se utiliza un algoritmo incremental.

El marco $B(L)$ que contiene todos los vértices de $A(L)$ en su interior puede ser fácilmente computado en tiempo cuadrático: computa todas las intersecciones de pares de líneas y escoge las que tengan mayor y menor coordenada x , y mayor y menor coordenada y. Un cuadrado isotético que contiene estos cuatro puntos contiene todos los vértices del arreglo.

```
public Point2D calcularInterseccionDual(Point2D
   punto1, Point2D punto2, Graphics g) {
    float y1;
    float y2;
    float m1 = punto1.x;
    float m2 = punto2.x;
    float b1 = punto1.y;
    float b2 = punto2.y;
    float x;
    x = (b1 - b2) / (m1 - m2);
    y1 = (m1 * x) - b1;
    y2 = (m2 * x) - b2;
    return new Point2D(x, y1);
}
```

Listado 3.6: Intersección de rectas duales de dos puntos.

El algoritmo incremental agrega las líneas $\ell_1, \ell_2, ..., \ell_n$ una tras otra y actualiza la DCEL después de cada adición. Sea A_i la subdivisión del plano inducida por el marco $B(L)$ y la parte de $A(\{\ell_1, ..., \ell_i\})$ dentro de $B(L)$. Para agregar la línea l_i, debemos partir las caras en A_{i-1} que son intersecadas por ℓ_i. Podemos encontrar estas caras caminando sobre ℓ_i de izquierda a derecha: Supongamos que entramos a una cara f a través de la arista e. Podemos caminar a lo largo de la frontera de f siguiendo los apuntadores $Next()$ en la DCEL hasta encontrar la media arista de la arista e' donde ℓ_i sale de f, entonces damos paso a la siguiente cara usando el apuntador $Twin()$ de esa media arista para llegar a la otra media arista de e'. De esta manera encontramos la siguiente cara en tiempo proporcional a la complejidad de la cara f.

Para encontrar la intersección más a la izquierda de ℓ_i con A_{i-1} sabemos que es una arista de $B(L)$. Simplemente probamos con todas las aristas en $B(L)$ para encontrar donde comenzar el recorrido. La cara dentro de $B(L)$ incidente a esta arista es la primer cara que ℓ_i parte. El arreglo A_{i-1} tiene a lo más $2i + 2$ aristas en $B(L)$ por lo tanto el tiempo necesario para este paso es lineal por cada línea.

Supongamos que tenemos que partir una cara f, y supongamos que la cara intersecada por ℓ_i a su izquierda ya ha sido partida. En particular, suponemos que la arista e donde entramos a f ya ha sido partida. Partir a f se hace como sigue. Primero creamos dos registros uno para la parte de f por debajo de ℓ_i y otro para la parte arriba de ℓ_i. Después partimos a e', la arista donde ℓ_i sale de f y creamos un nuevo vértice para $\ell_i \cap f$. Lo que queda por hacer es inicializar correctamente apuntadores en los registros nuevos de vértice, medias aristas y caras, actualizar apuntadores existentes a estos nuevos registros, además de eliminar los registros de f y de e'. El tiempo total para partir la cara es lineal en la complejidad de f.

El algoritmo para construir el arreglo puede ser resumido en:

Algoritmo 1: ConstruirArreglo(L)

Datos: Un conjunto L de líneas en el plano.
Resultado: La DCEL para la subdivisión inducida por $B(L)$ y la parte de $A(L)$ dentro de $B(L)$, donde $B(L)$ es el marco que contiene a todos los vértices de $A(L)$ en su interior.

```
1  inicio
2      Computa el marco B(L) que contiene todos los vértices de A(L) en su
        interior;
3      Construye la DCEL para la subdivisión inducida por B(L);
4      para i ← to n hacer
5          Encuentra la arista e en B(L) que contiene la intersección más a la
            izquierda de ℓ_i ∩ A_i;
6          f ← la cara acotada por e;
7          mientras f no es la cara no acotada, esto es, la cara fuera de B(L)
            hacer
8              Parte a f;
9              f ← la siguiente cara intersecada;
10         fin
11     fin
12 fin
```

Ahora el análisis del algoritmo. El paso 2 del algoritmo, calcular $B(L)$, puede ser hecho en $O(n^2)$. Paso 3 es de tiempo constante. Encontrar la primer cara que parte ℓ_i toma $O(n)$. Ahora acotamos el tiempo que toma partir las caras intersecadas por ℓ_i.

Primero supongamos que $A(L)$ es simple. En este caso el tiempo que nos toma para partir la cara f y para encontrar la siguiente cara intersecada es lineal en la complejidad de f. Por lo tanto, el tiempo total que necesitamos para insertar la línea ℓ_i es lineal en la suma de las complejidades de las caras de A_{i-1} intersecadas por ℓ_i. Esto nos lleva al concepto de zonas.

La *zona* de una línea ℓ en el arreglo $A(L)$ inducido por el conjunto L de líneas en el plano es el conjunto de caras de $A(L)$ cuya cerradura interseca ℓ. La complejidad de una zona es definida como el total de la complejidad de todas las caras que la componen. El tiempo necesario para insertar la línea ℓ_i es lineal en la complejidad de la zona de ℓ_i en $A(\{\ell_1, ..., \ell_i\})$. El teorema de la zona nos dice que esta cantidad es lineal:

Teorema 3.2.2. *La complejidad de la zona de una línea en un arreglo de m líneas en el plano es $O(m)$ [7].*

Ahora podemos acotar el tiempo de ejecución del algoritmo incremental para la construcción del arreglo. Ya vimos que el tiempo necesario para insertar ℓ_i es lineal en la complejidad de la zona de ℓ_i en $A(\{\ell_1, ..., \ell_{i-1}\})$. Por el teorema de la zona esto es $O(i)$, entonces el tiempo requerido para insertar todas las líneas es

$$\sum_{i=1}^{n} O(i) = O(n^2).$$

Los pasos 2 y 3 juntos toman $O(n^2)$, la suma total del tiempo de ejecución es $O(n^2)$. La complejidad de $A(L)$ es $\Theta(n^2)$ cuando $A(L)$ es simple, por lo tanto el algoritmo es óptimo.

Teorema 3.2.3. *Una DCEL para el arreglo inducido por un conjunto de n líneas en el plano puede ser construido en tiempo* $O(n^2)$.

```
public ArrayList<HalfEdge>
   recorrerFrontera(ArrayList<HalfEdge> arreglo, Face
   unbounded) {
    ArrayList<HalfEdge> faces = new
       ArrayList<HalfEdge>();
    HalfEdge edge = unbounded.inner;
    HalfEdge iterador = edge;
    faces.add(iterador);
    iterador = iterador.next;
    while (!iterador.equals(edge)) {
        faces.add(iterador);
        iterador = iterador.next;
    }
    return faces;
}
```

Listado 3.7: Método para recorrer la frontera.

```
public ArrayList<HalfEdge> recorerCara(HalfEdge edge) {
    ArrayList<HalfEdge> face = new
       ArrayList<HalfEdge>();
    HalfEdge iterador = edge;
    iterador = iterador.next;
    while (!iterador.equals(edge)) {
        face.add(iterador);
        iterador = iterador.next;
    }
    return face;
}
```

Listado 3.8: Método para recorrer una cara.

Veamos algunas subrutinas que podemos realizar una vez obtenida la DCEL y que nos serán de utilidad después.

Algoritmo 2: NextEdge(e)

Datos: Media arista e.
Resultado: La siguiente media arista en la línea de e.

```
1 inicio
2 |  devolver e.next.twin.next
3 fin
```

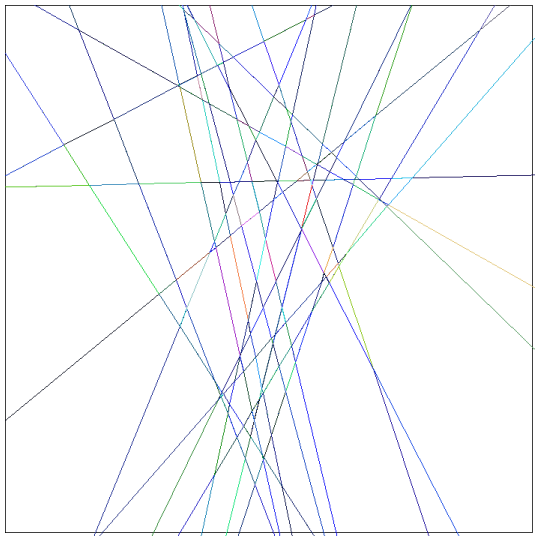


Figura 3.2: Dibujo de una DCEL.

Esta rutina recibe una media arista e y nos regresa la media arista que continua por la derecha sobre la línea de e, al estar trabajando en un arreglo simple (proveniente de un conjunto primal en posición general) podemos suponer correctamente que ningún vértice proviene de la intersección de más de dos líneas.

Algoritmo 3: TraverseLine(e)

Datos: La media arista e que es adyacente por la izquierda al marco.
Resultado: Una lista de medias aristas en la línea de e.

```
1 inicio
2 |  g = e ;
3 |  mientras g .next ∉ marco hacer
4 |  |  g = NextEdge(g) ;
5 |  fin
6 fin
```

La rutina `TraverseLine` recibe la media arista cuyo origen es la intersección de una línea con el marco y regresa una lista de las medias aristas dentro de $A(L)$ sobre la línea a la que pertenece e. Llama a `NextEdge` como subrutina hasta encontrar una media arista que es parte del marco.

4. Triángulo heterocromático de mayor y menor área

Enseguida daremos el planteamiento formal del problema resuelto en [11] y del cual daremos la implementación.

Problema 4.0.1. *Dado un conjunto P de puntos en el plano, en posición general, y coloreados de tres colores, $P = C_1 \cup C_2 \cup C_3$. Encontrar $a \in C_1, b \in C_2$ y $c \in C_3$ tal que se minimiza (maximiza) el área del triángulo que definen.*

En el siguiente capitulo, utilizando técnicas similares, implementaremos también el algoritmo que resuelve:

Problema 4.0.2. *Dado un conjunto P de puntos en el plano, en posición general, y coloreados de cuatro colores, $P = C_1 \cup C_2 \cup C_3 \cup C_4$. Determinar si existen puntos $a \in C_1, b \in C_2, c \in C_3$ y $d \in C_4$, en posición convexa.*

4.1. Definiciones

Definición 4.1.1. *Dado un punto p en el plano, llamaremos $H(p)$ a la recta horizontal que pasa por p.*

Definición 4.1.2. *Dado un punto p en el plano, llamaremos $H^-(p)$ al semiplano inferior determinado por $H(p)$, y $H^+(p)$ al semiplano superior.*

4.2. Algoritmo

En [11] exhiben un algoritmo que en tiempo $O(n^2)$ reporta el triángulo heterocromático de mayor y menor área utilizando técnicas conocidas como arreglo dual, ordenación angular y construcción de cierres convexos dada una sucesión ordenada de puntos. La idea principal del algoritmo es que por cada pareja bicromática hay que encontrar el punto más cercano (lejano) de la tercer clase cromática para calcular el triángulo de área mínima (máxima) que tiene a esa pareja y al tercer punto. Es por esto que a partir de este momento discutiremos únicamente el caso del triángulo de área mínima, el caso de área máxima es completamente análogo.
La entrada del algoritmo es un conjunto S de puntos 3-coloreado en posición general y los pasos, en alto nivel, son:

- Calcular el arreglo dual de S para poder obtener en tiempo lineal la sucesión ordenada angular respecto a un punto.
- Por cada permutación de los 3 colores. Sin perdida de generalidad fijémonos en la permutación de colores (c_1, c_2, c_3):
  - Por cada punto $p \in C_1$.
    - Dado el arreglo dual, ordenar angularmente respecto a p en sentido horario a $(S \setminus \{p\}) \cap H^+(p)$, denotemos a esta sucesión ordenada como $s_1, ..., s_k$.
    - Digamos que C_2' y C_3' son los puntos en $C_2 \cap H^+(p)$ y $C_3 \cap H^+(p)$ respectivamente.
    - Construir el cierre convexo de C_2' dada la sucesión ordenada $s_1, ..., s_k$, construimos este cierre con una modificación del algoritmo de Graham que nos permite desmantelar el cierre convexo punto por punto.
    - Recorrer la sucesión en sentido antihorario, es decir, $s_k, ..., s_1$, eliminando de $C'2$ cada punto recorrido de color c_2, actualizando $CH(C_2')$ y buscando la tangente más cercana a este cierre convexo actualizado que además sea paralela a la línea determinada por p y un punto en C_3' procesado.

4.2.1. Ordenamiento Angular

A manera de mantener la complejidad de $O(n^2)$ ordenaremos todas las sucesiones angulares utilizando el arreglo dual. Como ya hemos visto el arreglo dual se puede construir en tiempo $O(n^2)$ haciendo uso de una DCEL.

Ahora para conseguir la sucesión angular respecto a un punto p hacemos dos recorridos sobre su recta dual de izquierda a derecha auxiliándonos de `TraverseLine`. En el primer recorrido por cada intersección con otra recta cuyo punto primal está a la derecha de p agregamos el punto a una lista, en el segundo recorrido completamos la lista agregando los puntos que están a la izquierda de p. El tiempo total de ejecución de este algoritmo es $O(n^2)$.

4.2.2. Construcción y desmantelamiento de cierre convexo

Sea S un conjunto de puntos en el plano en posición general y una recta l horizontal por debajo de $CH(S)$. Además un punto $p \in l$ y S está ordenado angularmente respecto a p como $\{s_1, s_2, ..., s_n\}$. Entonces podemos calcular el cierre convexo de S de manera incremental en $O(n)$ mediante un barrido angular que visite los puntos en orden. Observación: un punto s_i está fuera del cierre convexo de los puntos $\{s_1, s_2, ..., s_{i-1}\}$. Esto porque el cierre convexo esta contenido en la cuña $s_1 p s_{i-1}$, por el orden angular sabemos que s_i no puede estar en la misma cuña y por transitividad tampoco puede estar en el cierre convexo.

El cierre convexo lo representaremos mediante una lista ligada, cada elemento de la lista será un punto del cierre y guardará una pila de puntos de la cual el ultimó punto

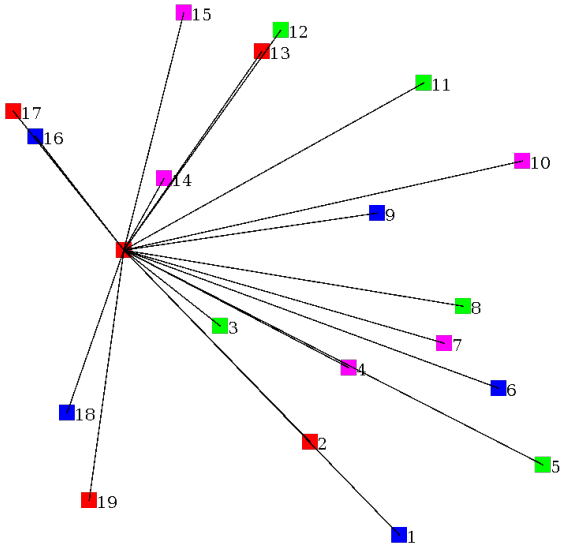


Figura 4.1: Ejemplo de orden angular respecto a un punto.

de la pila corresponde al punto sucesor en el cierre convexo (en sentido antihorario), además guardará un apuntador a su punto anterior (en sentido horario). Con los puntos s_1, s_2, s_3 formamos un triángulo en sentido antihorario. Para insertar s_i, $i > 3$, calculamos las tangentes a s_i con el cierre convexo $CH(s_1, ..., s_{i-1})$. Los puntos de tangencia serán adyacentes a s_i en $CH(s_1, ..., s_i)$. Hacemos esto de la siguiente manera:

- Calcular el giro del antecesor de s_{i-1} con s_{i-1} y s_i. Si el giro es hacia la derecha actualizar s_{i-1} a su antecesor e iterar. Si el giro es izquierdo se cumple la convexidad y el punto s_{i-1} es un punto de tangencia. El punto además debe guardar una pila de apuntadores al siguiente punto del cierre convexo. Agrega el punto de tangencia a su pila, como se muestra en la figura 4.2.

- Para el otro punto de tangencia hacemos el proceso análogo pero ahora la prueba de convexidad es entre s_i, s_{i-1} y el sucesor de s_{i-1}. Si no se cumple la convexidad probar con el sucesor de s_{i-1} y su respectivo sucesor. Iterar hasta encontrar el punto de tangencia.

4.2.3. Actualización de la recta tangente

Una parte importante del algoritmo es calcular la línea paralela a un segmento más cercano que sea tangente al cierre convexo de C_2. Hacemos esto de la siguiente manera:

- En tiempo $O(n)$ encontramos a los puntos con menor coordenada y simplemente recorriendo el convexo, la horizontal en este punto es la tangente inicial. Calculamos el ángulo entre el punto de menor coordenada y con un punto en su horizontal a la derecha y con su siguiente punto en el convexo.

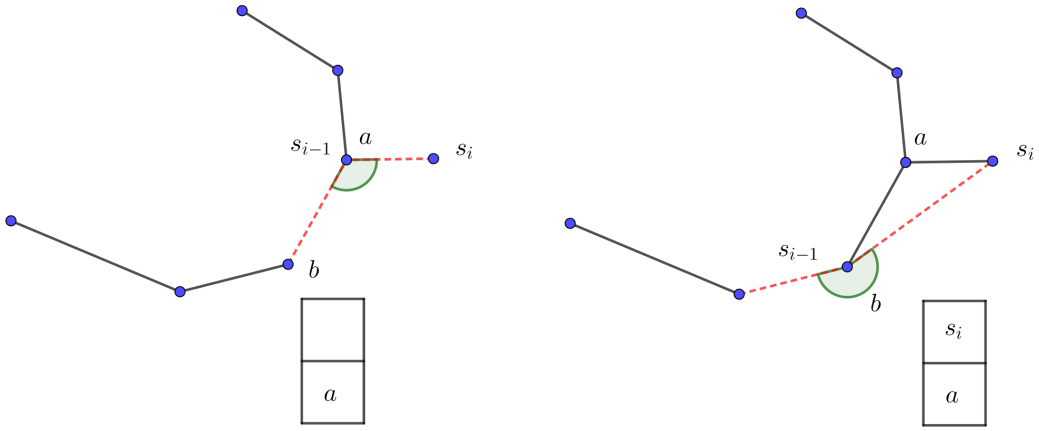


Figura 4.2: Actualización del cierre convexo.

- Recorremos en orden inverso los puntos de color c_3. Cada vez que visitemos un nuevo punto consideramos la recta pc_{3_i} y calculamos el ángulo entre esa recta y el rayo vertical hacia la derecha de p. Si este ángulo es mayor que el formado por la recta tangente y la horizontal entonces buscamos las rectas tangentes paralelas recorriendo en sentido antihorario los puntos del convexo hasta encontrar un punto que forme un ángulo menor con su punto sucesor. Dicho punto es ahora el punto de tangencia. Figura 4.3a.

- Cada que consideremos un punto $c_{3_{i-1}}$ debemos dejar de tomar en cuenta para el cierre convexo C_2 los puntos dentro de la cuña $c_{3_{i-1}}c_n$. Si el punto de tangencia está dentro de esa cuña entonces debemos buscar nuevamente. Solo hace falta buscar a partir del antecesor del punto de tangencia anterior hacia los sucesores en el cierre actualizado, como se muestra en la figura 4.3b.

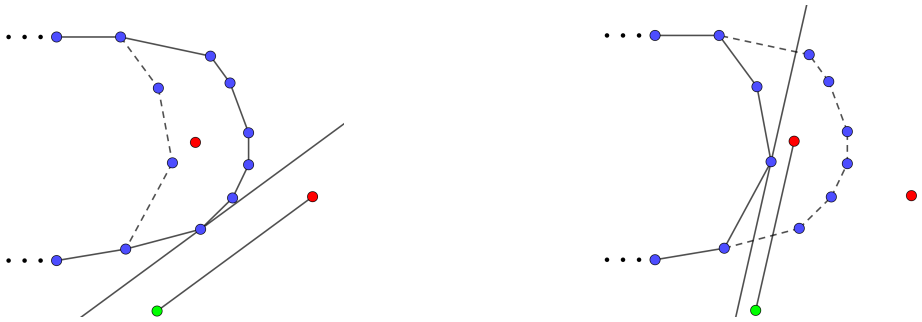
- Por cada segmento pc_{3_i} hemos calculado su punto de tangencia al convexo de color c_2 anterior a c_{3_i} en la sucesión, con estos segmentos y sus respectivos puntos de tangencia calculamos los triángulos definidos. Cada que calculemos uno de estos triángulos guardamos siempre el de menor área.

4.3. Implementación

Ahora listaremos las rutinas básicas que serán necesarias para luego usarlas en la implementación del algoritmo principal.

4.3.1. Orientación de tres puntos

[2] Supongamos que nos son dados una tripleta ordenada (A, B, C) de tres puntos en el plano $\mathbb{R}^2$ y queremos saber su orientación, en otras palabras, queremos saber si estamos girando en sentido horario o antihorario cuando visitamos estos puntos en



(a) (b)

Figura 4.3: Actualización de la recta tangente.

el orden dado. Nos referiremos a estas posibles orientaciones derecha e izquierda respectivamente. Existe una tercera posibilidad, que los puntos A, B y C estén sobre una misma línea recta, consideraremos esta orientación como orientación recta. Si trazamos los puntos en papel, vemos inmediato en cuál de estos casos cae, pero ahora necesitamos una manera de computar la orientación usando solamente la coordenadas dadas $x_A, y_A, x_B, y_B, x_C, y_C$. Definamos los dos vectores **a**= CA y **b**= CB como se muestra en la figura 4.4.

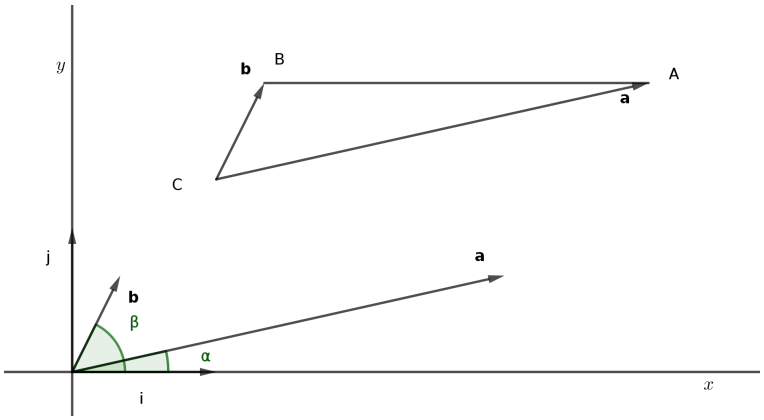


Figura 4.4: El uso de vectores `a` y `b` en lugar de usar las aristas CA y CB de un triángulo.

Los vectores están determinados por su dirección y longitud, no su localización, podemos decir que empiezan en el origen O en vez del punto C. Aunque este problema es esencialmente dos-dimensional y puede ser resuelto usando conceptos en dos dimensiones, es conveniente usar el espacio tres-dimensional. Como es usual, los vectores unitarios **i** ,**j** y **k** tiene direcciones opuestas a los ejes positivos x, y y z. Supongamos que el vector **k** al igual que **i** y **j** empiezan en O, y apunta hacia nosotros. Denotando los extremos de los vectores trasladados **a** y **b**, que empiezan en O, por $(a_1, a_2, 0)$ y $(b_1, b_2, 0)$, tenemos que

$$\mathbf{a} = a_1 i + a_2 j + 0k$$
$$\mathbf{b} = b_1 i + b_2 j + 0k$$

y

$$\mathbf{a} \times \mathbf{b} = \begin{vmatrix} i & j & k \\ a_1 & a_2 & 0 \\ b_1 & b_2 & 0 \end{vmatrix} = \begin{vmatrix} a_1 & a_2 \\ b_1 & b_2 \end{vmatrix} k = (a_1 b_2 - a_2 b_1)k$$

Entonces $\mathbf{a} \times \mathbf{b}$ es un vector perpendicular al plano xy con la misma dirección que $\mathbf{k} = \mathbf{i} \times \mathbf{j}$ o en la dirección opuesta dependiendo del signo de $a_1 b_2 - a_2 b_1$. Si esta expresión es positiva la relación entre $\mathbf{a}$ y $\mathbf{b}$ es similar a la de $\mathbf{i}$ y $\mathbf{j}$: podemos dar un giro menor a 180° en sentido antihorario sobre $\mathbf{a}$ para obtener la dirección de $\mathbf{b}$, de la misma manera en que podemos hacer esto con $\mathbf{i}$ para obtener $\mathbf{j}$. En general, tenemos

$$a_1 b_2 - a_2 b_1 \begin{cases} > 0 & \text{: orientación de A, B y C es giro izquierdo.} \\ = 0 & \text{: A, B y C están sobre la misma línea.} \\ < 0 & \text{: orientación de A, B y C es giro derecho.} \end{cases}$$

4.4. Área de un triángulo

Como vimos en la sección anterior el producto cruz $\mathbf{a} \times \mathbf{b}$ es un vector cuya longitud es igual al área del paralelogramo del cual $\mathbf{a}$ y $\mathbf{b}$ son dos de sus aristas. Ya que este paralelogramo es la suma de dos triángulos de áreas iguales se sigue que

$$2 \cdot \text{Area}(\Delta ABC) = \left|\mathbf{a} \times \mathbf{b}\right| = \begin{vmatrix} a_1 & a_2 \\ b_1 & b_2 \end{vmatrix} = a_1 b_2 - a_2 b_1$$

Notemos que esto es valido solo si A, B y C están etiquetados en sentido antihorario, si este no es el caso tenemos que usar el valor absoluto de $a_1 b_2 - a_2 b_1$. Como $a_1 = (x_A - x_C), a_2 = (y_A - y_C), b_1 = (x_B - x_C), b_2 = (y_b - y_c)$ también podemos escribir

$$2 \cdot \text{Area}(\Delta ABC) = (x_A - x_C)(y_B - y_C) - (y_A - y_C)(x_B - x_C)$$

Usaremos repetidas veces el método area2 mostrado abajo que toma tres argumentos de la clase `Point2D`. Este metodo computa el área del triángulo ABC multiplicado por 2, o, si A, B y C están en sentido antihorario, por -2.

```
public float area2(Point2D a, Point2D b, Point2D c) {
    float area2 = (a.x - c.x) * (b.y - c.y) - (a.y - c.y)
       * (b.x - c.x);
    return area2;
}
```

Listado 4.1: Método area2.

Si estamos interesados solamente en la orientación de los puntos A, B y C, podemos escribir.

```
public boolean esGiroIzquierdo(Point2D a, Point2D b,
   Point2D c) {
    return area2(a, b, c) > 0;
}

public boolean esGiroDerecho(Point2D a, Point2D b, Point2D
   c) {
    return area2(a, b, c) < 0;
}
```

Listado 4.2: Métodos para obtener la orientación de 3 puntos.

Una representación simple de un triángulo:

```
public class Triangulo {

    Point2D a;
    Point2D b;
    Point2D c;
    float area;

    public Triangulo(Point2D a, Point2D b, Point2D c,
       float area) {
        this.a = a;
        this.b = b;
        this.c = c;
        this.area = area;
    }

}
```

Listado 4.3: Representación de un triángulo.

4.5. Rutinas

Algoritmo 4: ObtenerOrdenAngular(e)

Datos: La media arista e que es adyacente por la izquierda al marco.
Resultado: Una lista de puntos angularmente ordenada con respecto a el punto dual de la línea que contiene a e.

```
1  inicio
2  |  orden = lista vacía;
3  |  segmentos = TraverseLine(e);
4  |  para segmento ∈ segmentos hacer
5  |  |  si segmento.next.twin.face ∉ bounding entonces
6  |  |  |  si segmento.next.line.puntPrimal.x > e.line.puntoPrimal.x
   |  |  |   entonces
7  |  |  |  |  orden.add();
8  |  fin
9  |  para segmento ∈ segmentos hacer
10 |  |  si segmento.next.twin.face ∉ bounding entonces
11 |  |  |  si segmento.next.line.puntPrimal.x <= e.line.puntoPrimal.x
   |  |  |   entonces
12 |  |  |  |  orden.add();
13 |  fin
14 |  devolver orden
15 fin
```

La rutina `ObtenerOrdenAngular` recibe la media arista más a la izquierda de una línea adyacente al marco, con ella llama a `TraverseLine` para obtener la lista de medias aristas contenidas dentro del marco, cada una de estas aristas se formó por la intersección de únicamente la línea que contiene a e y otra línea distinta, es por esto que por cada media arista en la lista obtenemos el punto primal de la media arista adyacente y comparamos primero si está a la derecha del punto primal de la media arista e, si esto se cumple se agrega el punto a la lista, en un segundo recorrido análogo ahora agregamos el punto a la lista si el punto primal está la izquierda.

Ahora veamos la modificación del algoritmo de Graham para calcular el cierre convexo. La idea es apoyarnos de que podemos obtener mediante la DCEL la sucesión ordenada angularmente de puntos con respecto a otro punto, dada esa sucesión procesamos punto por punto. El cierre convexo de un polígono convexo y un punto fuera de este puede ser actualizado calculando ambas tangentes, teniendo los puntos de tangencia basta con actualizar estos puntos para que apunten ahora al punto agregado. Para esto necesitamos las siguientes dos subrutinas, `BuscarPrev` y `BuscarNext`.

Algoritmo 5: BuscarPrev(p)

Datos: Un punto p fuera de la región convexa definida por el cierre.
Resultado: El punto previo a p en el cierre convexo actualizado.

```
1 inicio
2     Puntos a, b, c;
3     a = p;
4     b = ultimoInsertado;
5     c = ultimoInsertado.prev;
6     mientras abc sea un giro a la izquierda hacer
7         b = c;
8         c = c.prev;
9     devolver c
```

`Buscar prev` recibe el punto p que va a ser agregado al cierre convexo, además supone que se conoce el último punto agregado y el cierre convexo de los puntos anteriores a p en la sucesión ordenada. Itera sobre la cadena convexa anterior al último punto agregado, recorre las aristas de esta cadena hasta que los puntos que definen la arista, con el punto p hacen un giro a la derecha, de esta manera obtenemos una de las tangentes, así garantizando la convexidad.

Algoritmo 6: BuscarNext(p)

Datos: Un punto p fuera de la región convexa definida por el cierre.
Resultado: El punto sucesor de p en el cierre convexo actualizado.

```
1 inicio
2     Puntos a, b, c;
3     a = p;
4     b = ultimoInsertado;
5     c = ultimoInsertado.next;
6     mientras abc sea un giro a la derecha hacer
7         b = c;
8         c = c.next;
9     devolver c
```

De manera análoga a `BuscarPrev` la subrutina `BuscarNext` recibe el punto p que queremos agregar, supone que se conoce el último punto agregado y el convexo anterior a p en el orden. Ahora itera sobre la cadena convexa posterior al último punto agregado, recorre la cadena hasta que los puntos que definen la arista con el punto p hacen un giro a la izquierda.

Ahora que sabemos como podemos actualizar el cierre convexo describimos como usar las rutinas anteriores para construir el cierre convexo, esta rutina mantiene un apuntador al último punto agregado y una lista de puntos que representa el cierre convexo.

Algoritmo 7: AgregarPunto(p)

Datos: Un punto p fuera de la región convexa definida por el cierre.
Resultado: La información del convexo actualizado.

```
1  inicio
2      Puntos nextAux, prevAux // para no modificar el convexo al buscar
         adyacencia;
3      prevAux = buscarPrev(p);
4      nextAux = buscarNext(p);
5      p.prev = prevAux;
6      p.prev.next = p;
7      p.next = nextAux;
8      ultimoInsertado = p;
9      cardinalidad = cardinalidad + 1;
```

En la rutina `AgregarPuntos` declaramos variables auxiliares al momento de buscar las tangentes, al momento de encontrar una de las tangente no actualizamos de inmediato parte del convexo puesto que interferiría con la búsqueda de la otra tangente. Al momento de ya tener ambos puntos tangenciales podemos actualizar el convexo agregando el punto nuevo y modificando los apuntadores de los puntos tangenciales encontrados.

Ahora que tenemos estas subrutinas es inmediato el algoritmo de cierre convexo dada la sucesión angularmente ordenada respecto a un punto.

Algoritmo 8: CrearArregloConvexo(L)

Datos: Una lista de puntos rojos y azules ordenados angularmente en sentido horario.
Resultado: Una lista que representa al convexo.

```
1  inicio
2      Cardinalidad = 0 ;
3      para cada punto p en L hacer
4          si p es azul entonces
5              si cardinalidad==0 entonces
6                  ultimoAgregado = p ;
7                  cardinalidad = 1 ;
8              si no, si cardinalidad == 1 entonces
9                  ultimoAgregado.next = p;
10                 ultimoAgregado.prev = p;
11                 p.next = ultimoAgregado;
12                 p.prev = ultimoAgregado;
13                 ultimoAgregado = p;
14                 cardinalidad = 2 ;
15             en otro caso
16                 agregarPunto(p);
```

El algoritmo mantiene la información del último punto insertado y la lista que representa al convexo para ser usado por subrutinas.

Algoritmo 9: BuscarTangente(p, s, r)

Datos: Un punto p donde se inicia la búsqueda, en la primera iteración es el punto con menor coordenada y. El punto s que define el orden y el punto rojo r que define la línea $\overline{rs}$ de la cual la tangente debe ser paralela.

Resultado: Un punto q que es de tangencia, además es el punto del convexo donde empieza la búsqueda de la nueva tangente al modificar el cierre convexo.

```
1 inicio
2 |  q = p ;
3 |  mientras ∠q-q.next ≤ ∠s-r hacer
4 |  |  q = q.next ;
5 |  fin
6 |  devolver q
7 fin
```

Esta rutina supone que se conoce la tangente al convexo para el último punto rojo procesado r_{i-1}, este punto de tangencia determina el inicio de la búsqueda del punto tangencial para el punto rojo actual r_i. Sabemos que el ángulo entre un par de puntos p_{i-1}, p_i y el rayo positivo paralelo al eje x con origen en p_{i-1} es menor que el ángulo entre los puntos p_i, p_{i+1} con el rayo positivo x con origen en p_i.
Debemos tener cuidado al eliminar un punto azul ya que este puede ser el punto de tangencia encontrado para el último punto rojo procesado p_k y por tanto es el punto en donde empezaría la búsqueda de la siguiente tangente al procesar un punto rojo p_{k+1}. No nos movemos al siguiente punto en el convexo porque la tangente puede estar en la sección del convexo que es visible al punto azul como se ve en la figura 4.3b. La siguiente rutina nos ayuda a lidiar con este caso.

Algoritmo 10: BorrarPuntoAzul$(p, puntoTangencia)$

Datos: Un punto p en el orden angular, el cierre convexo hasta el punto p. El punto de tangencia anterior.

Resultado: Un punto q que es el punto donde empieza la búsqueda de la nueva tangente.

```
1 inicio
2 |  si p == puntoTangencia entonces
3 |  |  q = p.prev ;
4 |  en otro caso
5 |  |  q = puntoTangencia ;
6 |  fin
7 |  p.prev.pop;
8 |  devolver q
9 fin
```

Ahora ya tenemos todas las herramientas para describir de manera muy compacta y sin perdida de detalles el algoritmo para encontrar el triángulo heterocromático de menor área.

Algoritmo 11: BuscarTriangulo(A, s)

Datos: Un arreglo A de puntos rojos y azules ordenados angularmente respecto a s en sentido antihorario, los puntos rojos almacenan la información de la construcción del cierre convexo.

Resultado: El triángulo heterocromático de menor área que tiene a s como el vértice de menor coordenada y.

```
1  inicio
2      para cada punto p en A hacer
3          si p es azul entonces
4              eliminarPunto(p);
5          si no, si p es rojo entonces
6              si cardinalidad == 0 entonces
7                  //no hay convexo donde buscar;
8              si no, si cardinalidad == 1 entonces
9                  puntoTangente = único punto del convexo;
10             en otro caso
11                 puntoTangente = buscarTangente(p);
12             trianguloMenor = min(trianguloMenor,
                   áreaTriángulo(s, p, puntoTangente));
13     devolver triánguloMenor
```

En aras de comprensión hemos omitido en el texto la implementación en lenguaje java de algunas de las rutinas descritas, la implementación completa puede ser consultada en `https://github.com/RodGpe/Java_graphics/`.

4.6. Análisis de complejidad

La construcción del arreglo de líneas es de complejidad temporal $O(n^2)$ como lo vimos en su respectiva sección. Por cada punto estamos obteniendo la ordenación angular en tiempo $O(n)$, la construcción del cierre convexo nos toma tiempo $O(n)$ amortizado por la simple observación de que por cada punto procesado agregamos exactamente 2 aristas al cierre convexo y las aristas que recorrimos para buscar las tangentes al punto no las volvemos a visitar ya que no son visibles a puntos fuera de la región convexa definida por el cierre. La ventaja de usar pilas para la actualización del cierre convexo es que al eliminar un punto de la pila lo que nos queda es el cierre convexo sin considerar el punto eliminado, sin necesidad de hacer más operaciones, es decir, quitar un punto y actualizar el cierre convexo nos toma tiempo constante.

El proceso de recorrer un convexo para encontrar las tangentes paralelas a las parejas bicromáticas también es de tiempo $O(n)$ amortizado por razones parecidas. Al

procesar los puntos en orden angular nos aseguramos de que los ángulos que se generen con las parejas y un rayo horizontal serán en orden creciente; al caminar sobre el cierre convexo empezando por el punto de menor coordenada y, las ángulos generados entre las aristas y un rayo horizontal también serán en orden creciente; por lo tanto, cada vez que visitemos una arista del cierre no será necesario volver a visitarla.

Esto se repite para cada punto del conjunto y para cada permutación de los 3 colores. La complejidad total entonces es $6n(O(n) + O(n)) + O(n^2) = O(n^2)$.

4.7. Análisis de correctitud

Al ser exhaustivo en las parejas bicromáticas de puntos y por cada pareja encontrando su triángulo de menor área, siempre guardando el que minimice el área entonces se sigue directamente que el algoritmo reporta correctamente el triángulo de menor área.

5. Cuadrilátero heterocromático convexo

5.1. Descripción del problema

Con las mismas técnicas de la sección anterior se resolverá un problema distinto. Dado un conjunto de puntos 4-coloreado en posición general en el plano real $\mathbb{R}^2$ queremos determinar si existe un cuadrilátero heterocromático convexo.

5.2. Algoritmo

Presentamos un algoritmo junto con su implementación para detectar en tiempo $O(n^2)$ si en un conjunto de puntos 4-coloreado existe un cuadrilátero convexo heterocromático.

Definición 5.2.1. *Llamaremos* recta de soporte *de dos conjuntos de puntos, a la tangente exterior a sus correspondientes cierres convexos que separa a los conjuntos y a un punto p.*

Definición 5.2.2. *Dados en el plano dos conjuntos de puntos P_1 y P_2, un punto p por debajo de ambos conjuntos y un punto q, diremos que q está encima de la recta de soporte de P_1 y P_2 si se encuentra en el semiplano definido por esta, opuesto a donde se encuentra p. Figura 5.1.*

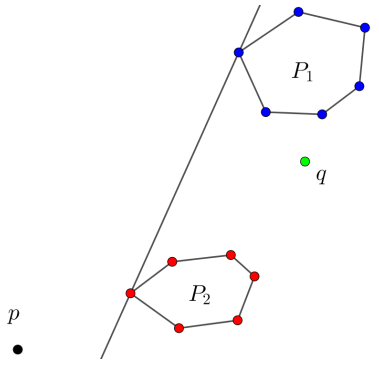


Figura 5.1: Un punto q por encima de la recta soporte de P_1 y P_2.

Definición 5.2.3. *Sea S una sucesión ordenada de puntos coloreados, denotaremos por $s_{i_k}, ..., s_{i_\ell}$ a la subsucesión ordenada en S de los puntos de color i.*

La entrada del algoritmo es un conjunto S de puntos 4-coloreado en posición general y los pasos, en alto nivel, son:

- Calcular el arreglo dual de S.
- Por cada permutación de los 4 colores. Sin perdida de generalidad fijémonos en la permutación (c_1, c_2, c_3, c_4) o $(negro, rojo, azul, verde)$:
  - Por cada punto $p \in C_1$
    - Obtener el orden angular en sentido horario de los puntos por encima de p, denotemos a la sucesión como $s_1, .., s_k$.
    - Construir el cierre convexo de los puntos rojos en sentido antihorario con la modificación del algoritmo de Graham que nos permite desmantelar el cierre convexo punto por punto.
    - Por cada punto $s_i \in s_1, ..., s_k$
      - Si s_i es de color azul re-calculamos el cierre convexo con s_i y los puntos anteriores en el orden.
      - Si s_i es de color rojo re-calculamos el cierre convexo de los puntos rojos ahora sin tomar en cuenta a s_i.
      - Si s_i es de color verde entonces verificamos que esté por encima de la recta de soporte, si esto se cumple entonces reportamos que los puntos que definen la recta de soporte, con el punto s_i y p forman un cuadrilátero heterocromático convexo y el algoritmo termina.

5.3. Estructuras de datos

Representamos a un cuadrilátero con la siguiente clase:

```
public class Cuadrilatero {
    Point2D a;
    Point2D b;
    Point2D c;
    Point2D d;

    public Cuadrilatero(Point2D a, Point2D b, Point2D c,
       Point2D d) {
        this.a = a;
        this.b = b;
        this.c = c;
        this.d = d;
    }
}
```

Listado 5.1: Representación de un cuadrilátero.

Esta es la clase que utilizamos para representar la recta soporte y los puntos que son parte de el cierre convexo:

```
class PuntoInfoConvex extends Point2D {

    Stack<PuntoInfoConvex> prev;
    Stack<PuntoInfoConvex> next;

    public PuntoInfoConvex(Stack<PuntoInfoConvex> prev,
       Stack<PuntoInfoConvex> next, Color color, float x,
       float y) {
        super(x, y, color);
        this.prev = prev;
        this.next = next;
    }

}
```

Listado 5.2: Clase que representa un punto del convexo.

Al momento de actualizar las rectas de soporte necesitamos recorrer el cierre convexo en sentido horario y antihorario, a diferencia del algoritmo para encontrar el triángulo de menor área. Por está razón de un punto necesitamos guardar en pilas tanto los puntos siguientes en el cierre convexo así como los puntos anteriores.

```
class SegmentoSoporte {

    PuntoInfoConvex color1;
    PuntoInfoConvex color2;

    public SegmentoSoporte(PuntoInfoConvex color1,
       PuntoInfoConvex color2) {
        this.color1 = color1;
        this.color2 = color2;
    }

}
```

Listado 5.3: Clase que representa la recta soporte.

5.4. Rutinas

Algoritmo 12: BuscarCuadrilatero(L, p)

Datos: Una lista L de puntos azul, rojo y verdes ordenados angularmente respecto a p en sentido horario.

Resultado: Si existe, regresa un cuadrilátero heterocromático formado por puntos en L.

```
1  inicio
2      CrearConvexoRojo;
3      para cada punto en A hacer
4          si punto es azul entonces
5              agregarPuntoConvexAzul;
6          si no, si punto es rojo entonces
7              eliminarPuntoConvexRojo;
8          si no, si el punto es verde entonces
9              si el punto está por encima de la recta soporte entonces
10                 devolver El cuadrilátero definido por el punto verde, los puntos
                       de la recta de soporte y el punto p
```

`BuscarCuadrilatero` es el algoritmo que determina si existe o no el cuadrilátero convexo, consiste en primero construir en sentido antihorario el cierre convexo de puntos rojos para después recorrer angularmente los puntos en sentido horario. Cuando se procese un punto azul agregamos ese punto al convexo de puntos azules y actualizamos la recta de soporte, si el punto es rojo eliminamos ese punto del convexo de puntos rojos y actualizamos la recta de soporte. Si el punto es verde entonces verificamos que esté en el semiplano opuesto al punto de referencia de ordenación formado por la recta de soporte.

A continuación describimos con mayor detalle las subrutinas usadas cuando agregamos un punto azul, cuando eliminamos un punto rojo y cómo actualizamos la recta de soporte en ambos casos.

Algoritmo 13: agregarPuntoConvexAzul(p)

Datos: Ordenados angularmente en sentido horario: un convexo azul, un punto azul p y un convexo rojo.

Resultado: El cierre convexo de los puntos azules después de agregar el punto p.

```
1  inicio
2      si cardinalidadAzul==0 entonces
3          ultimoInsertado = p;
4          cardinalidadAzul = +1;
5      si no, si cardinalidadAzul==1 entonces
6          ultimoInsertado.next = p;
7          ultimoInsertado.prev = p;
8          p.next = ultimoInsertado;
9          p.prev = ultimoInsertado;
10         ultimoInsertado = p;
11         cardinalidadAzul+ = 1; ;
12     en otro caso
13         prevAux;
14         nextAux;
15         prevAux = buscarPrev(p);
16         nextAux = buscarNext(p);
17         p.prev = prevAux;
18         p.prev.next = p;
19         p.next = nextAux;
20         p.next.prev = p;
21         ultimoInsertado = p ;
22         cardinalidadAzul+ = 1;
23     actualizarRectaSoporteAgregando(p);
```

Al considerar un nuevo punto azul para el cierre convexo usamos las subrutinas `buscarPrev` y `buscarNext` para buscar los puntos anterior y siguiente del punto nuevo dentro del convexo que lo contiene. Separamos los casos en si la cardinalidad del convexo antes de agregar el punto es 0, 1 o mayor que 1, esto lo hacemos porque las funciones `buscarPrev` y `buscarNext` suponen que el convexo de búsqueda tiene al menos dos puntos. Al agregar el punto azul puede ocurrir que este quede debajo de la recta de soporte que estaba definida con los cierres convexos azul y rojo cuando no incluían al nuevo punto azul. Para eso al final de 13 se llama a `actualizarRectaSoporteAgregando` cuya función es actualizar la recta de soporte cuando el cierre convexo azul es modificado.

Algoritmo 14: actualizarRectaSoporteAgregando(p)

Datos: Ordenados angularmente en sentido horario: un convexo azul, un punto azul p y un convexo rojo.
Resultado: Recta de soporte actualizada.

```
1  inicio
2      si soporte.azul, soporte.rojo, p forman giro izquierdo entonces
3          //no se actualiza la recta de soporte
4      en otro caso
5          soporte.azul = p;
6          a = soporte.azul;
7          b = soporte.rojo;
8          c = soporte.rojo.next;
9          mientras abc sean giro derecho hacer
10             b = c;
11             c = c.next;
12         soporte.rojo = b;
```

La manera de actualizar la recta de soporte en el caso cuando se agrega un punto azul es muy directa. Si el punto azul agregado cae por encima de la recta soporte, el punto no modifica a la recta, esto por la definición de recta soporte. En el caso de que el punto caiga por debajo de la recta soporte sabemos que el punto agregado va a definir la nueva recta soporte, entonces lo que tenemos que hacer es recorrer en sentido antihorario el convexo rojo a partir del punto rojo de la recta de soporte, por cada punto recorrido nos fijamos en el giro, izquierdo o derecho, que se forma con el punto, su vecino en el cierre y el punto azul agregado, cuando se cumpla que el giro es giro izquierdo entonces también se cumple que todos los puntos rojos están por encima de la recta soporte.

Algoritmo 15: eliminarPuntoConvexRojo(p)

Datos: Ordenados angularmente en sentido horario: un convexo azul, un punto rojo p y un convexo rojo.
Resultado: El cierre convexo de los puntos rojos después de eliminar el punto p.

```
1  inicio
2      p.prev.next.pop();
3      p.next.prev.pop();
4      si p == soporte.rojo entonces
5          actualizarRectaSoporteEliminando(p);
```

Ahora veamos el caso cuando dejamos de considerar un punto rojo. Para eliminarlo del cierre convexo basta con borrar de sus vecinos en el cierre convexo la referencia al punto que estamos eliminando. Si el punto que eliminamos formaba parte de la recta de

soporte debemos actualizar los nuevos puntos de esta. A diferencia del caso del punto azul la manera de actualizar la recta de soporte no es tan directa, veamos la subrutina `actulizarRectaSoporteEliminando`.

Algoritmo 16: actualizarRectaSoporteEliminando(p)

Datos: Ordenados angularmente en sentido horario: un convexo azul, un punto rojo p y un convexo rojo.
Resultado: Recta de soporte actualizada.

```
1  inicio
2      mientras soporte.azul,soporte.rojo,soporte.rojo.next es giro derecho
3      o soporte.azul, soporte.rojo, soporte.azul.next es giro derecho hacer
4          a = soporte.azul;
5          b = soporte.rojo;
6          c = soporte.rojo.next;
7          mientras a,b,c es giro derecho hacer
8              b = c;
9              c = c.next;
10         soporte.rojo = b;
11         a = soporte.rojo;
12         b = soporte.azul;
13         c = soporte.azul.next;
14         mientras a,b,c es giro izquierdo hacer
15             b = c;
16             c = c.next;
17         soporte.azul = b;
```

Al eliminar un punto del cierre convexo rojo, a diferencia de cuando se agrega un punto azul, no se puede garantizar de manera inmediata cuáles puntos van a pertenecer a la recta de soporte, pero sí podemos acotar las regiones donde pueden estar los nuevos puntos que definan a la nueva recta de soporte. El punto rojo de la nueva recta debe ser visible al punto rojo eliminado, es decir, se puede trazar un segmento de recta entre ambos puntos sin intersecar al polígono. Primero actualizamos el punto rojo de la recta utilizando al punto azul de la recta soporte anterior. Recorremos la región visible al punto rojo eliminado y verificamos que el giro que generan el punto azul, el punto rojo recorrido y su siguiente punto en el convexo en sentido antihorario generen giro izquierdo. Una vez hecho esto, pudo haber pasado que ahora el punto azul no sea un punto de tangencia, por lo tanto actualizamos el punto azul de manera análoga, verificamos que el punto rojo actualizado, el punto azul recorrido y su siguiente punto en el convexo en sentido antihorario generen un giro derecho. Una vez más al hacer esto el punto rojo que ya actualizamos anteriormente puede ahora no ser de tangencia, si este es el caso repetimos las actualizaciones hasta que las condiciones de los giros izquierdo y derecho se cumplan.

Así hemos concluido la descripción de las subrutinas necesarias para la implementación del algoritmo, el código fuente de la implementación se encuentra en el repositorio `https://github.com/RodGpe/Java_graphics/`. En la figura 5.2 se muestran los puntos coloreados que son entrada del programa y el cuadrilátero heterocromático encontrado por nuestra implementación.

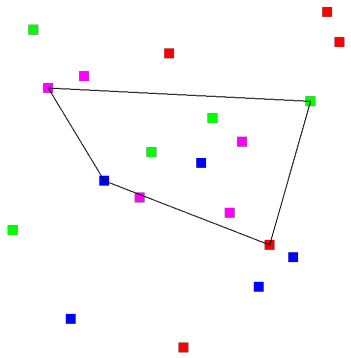


Figura 5.2: Un cuadrilátero heterocromático convexo en un conjunto 4-coloreado.

5.5. Análisis de complejidad

El análisis de complejidad es análogo al del algoritmo para encontrar los triángulos heterocromáticos, la construcción del arreglo de líneas es de complejidad $O(n^2)$, por cada punto obtenemos la sucesión angularmente ordenada en tiempo $O(n)$, la construcción del cierre convexo en $O(n)$, podemos eliminar puntos y actualizar el cierre convexo en tiempo constante gracias a la utilización de pilas. Los recorridos sobre los cierres convexos por cada actualización de la recta de soporte pueden tomar tiempo $O(n)$ cada uno, pero cada vez que visitamos una arista esta no vuelve a ser visitada en actualizaciones posteriores; por lo tanto, todas la actualizaciones de la recta tangente se llevan a cabo en tiempo $O(n)$ amortizado. La verificación de que un punto esté por encima de la recta de soporte se hace en tiempo constante.

Esto se repite para cada punto del conjunto y para cada permutación de los 4 colores. La complejidad total entonces es $24n(O(n) + O(n)) + O(n^2) = O(n^2)$.

5.6. Análisis de correctitud

En aras de llegar a una contradicción supongamos que existe al menos un cuadrilátero convexo heterocromático y que el algoritmo responde de manera negativa. Sea R uno de estos cuadriláteros y a, b, c, d sus vértices en sentido antihorario, siendo a el de menor coordenada y. Al procesar punto por punto llegará el momento en el fijaremos a a en búsqueda del convexo que tenga a a como su punto más bajo. Mientras procesamos a a consideraremos todas las permutaciones de colores, en particular la permutación de colores de los puntos b, c, d. Al procesar c en la sucesión ordenada

angular tenemos el cierre convexo de los puntos del mismo color que d antes que c en la sucesión ordenada y el cierre convexo de los puntos del mismo color que b después que c. Como supusimos que el algoritmo respondió negativamente, entonces c debe estar en el mismo semiplano que a definido por la recta de soporte de los cierres convexos. Por definición de recta de soporte, cualquier otra recta definida por cualquier par de puntos contenidos en esos cierres también está por encima de c, dejando a a y c en el mismo semiplano, lo que contradice que R es un cuadrilátero convexo. Por lo tanto podemos decir que el algoritmo es correcto.

5.7. Conclusiones y trabajo futuro

La implementación de los algoritmos en [11] que describimos en este trabajo pretende servir para continuar la investigación de la existencia de convexos heterocromáticos en conjuntos de puntos coloreados. Por ejemplo en familias de conjuntos como:

- Conjuntos de Horton.
- Cadenas convexas dobles.
- Cadenas ZigZag dobles.
- Conjuntos de puntos en forma de embudo.

Nuestra implementación trabaja con puntos de precisión finita e inexacta. Una manera interesante de mejorar la implementación es bajo el paradigma de computación exacta.

A. Bibliotecas de algoritmos en geometría computacional

- CGAL. Está escrito en el lenguaje C++. implementa algoritmos como
  - Triangulación
  - Diagramas de Voronoi
  - Operaciones booleanas en polígonos
  - Cierre convexo
  - Arreglo de líneas

  Su última versión, 5.1, fue lanzada en septiembre de 2020. Actualmente es la biblioteca con mayor soporte.

- libigl. Biblioteca escrita en C++ para procesamiento geométrico con algoritmos como
  - Triangulación de polígonos cerrados
  - Tetraedralización de superficies cerradas
  - Representación de mallas
  - Operaciones booleanas en mallas
  - Contar componentes conexas de una gráfica
  - Programación cuadrática
  - Deformación de figuras
  - Localización de puntos

  Una de sus características más importantes es la posibilidad de llamar a funciones de esta biblioteca desde el programa MATLAB para descargar a C++ las partes computacionalmente intensas de una aplicación de MATLAB.

- LEDA. Otra biblioteca en C++ que provee algoritmos en campos como gráficas, redes, computación geométrica, optimización combinatoria. Algunos de sus algoritmos son:
  - Ruta más corta en gráficas

  - Flujo máximo
  - Algoritmos para dibujar gráficas

  Su última versión es la 6.6.

- Geometry3Sharp. Biblioteca escrita en C# para computación geométrica y algoritmos en mallas. Su última actualización fue en 2018.

- Wykobi. Biblioteca escrita en C++ orientada a geometría computacional. Algoritmos que implementa:

  - Ciere convexo - Graham, Jarvis, Melkman
  - Triangulación
  - Círculo delimitador de área mínima

  Su última actualización fue lanzada en diciembre de 2018.

- Gosl - Go scientific library. Conjunto de herramientas escritas principalmente en el lenguaje GO para el desarrollo de simulaciones. Su principal enfoque es el desarrollo de métodos numéricos y solucionadores para ecuaciones diferenciables pero también presentan algunas funciones para la trasformada de Fourier, generación de números aleatorios, distribuciones de probabilidad y geometría computacional. Actualmente tiene una comunidad activa que sigue agregando funcionalidades.

- Geode. Biblioteca para geometría computacional escrita en C con una capa de enlace para ser utilizada con el lenguaje Python.

- Habrador's Computational Geometry Unity Library. Unity es un motor gráfico multi plataforma especilizado para crear juegos. La biblioteca escrita en C# por Erik Nordeus implementa algoritmos clásicos de geometría computacional en el motor Unity. Última actualización fue en febrero de 2020.

- Boost es una colección de bibliotecas para el lenguaje C++ que probee soporte para tareas en áreas como algebra lineal, generación de números aleatorios, expresiones regulares entre otras. Boost.Geometry (también conocido como GGL, por sus siglas en inglés, Generic Geometry Library) es parte de la colección Boost, está define conceptos, primitivas y algoritmos para resolver problemas geométricos. Muchos de los fundadores de Boost son parte del comité del estándar para C++. Su útlima actualización fue en agosto de 2020.

- mikhaildubov/Computational-geometry. Repositorio con implementación en Java de algoritmos de geometría computacional. De las librerias existentes esta es de las pocas con mayor aceptación que están escritas en Java. Su última actualización fue en febrero de 2016.

Bibliografía

[1] Oswin Aichholzer, Ruy Fabila-Monroy, David Flores-Penaloza, Thomas Hackl, Clemens Huemer, and Jorge Urrutia. Empty monochromatic triangles. *Computational geometry*, 42(9):934–938, 2009.

[2] Leen Ammeraal and Kang Zhang. *Computer Graphics for Java Programmers*. John Wiley and Sons, Inc., Hoboken, NJ, USA, 2nd edition, 2007.

[3] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag TELOS, Santa Clara, CA, USA, 3rd ed. edition, 2008.

[4] Bernard Chazelle, Leo J Guibas, and Der-Tsai Lee. The power of geometric duality. *BIT Numerical Mathematics*, 25(1):76–90, 1985.

[5] Craig Cordes and KB Reid. Largest polygons with vertices in a given finite set. *Discrete applied mathematics*, 14(3):255–262, 1986.

[6] Olivier Devillers, Ferran Hurtado, Gyula Károlyi, and Carlos Seara. Chromatic variants of the erdos–szekeres theorem on points in convex position. *Computational Geometry*, 26(3):193–208, 2003.

[7] Herbert Edelsbrunner, Joseph O'Rourke, and Raimund Seidel. Constructing arrangements of lines and hyperplanes with applications. *SIAM Journal on Computing*, 15(2):341–363, 1986.

[8] Paul Erdös and George Szekeres. A combinatorial problem in geometry. *Compositio mathematica*, 2:463–470, 1935.

[9] Ruy Fabila-Monroy, Daniel Perz, and Ana Laura Trujillo. Empty rainbow triangles in k-colored point sets. 2020.

[10] Kai Jin. Maximal area triangles in a convex polygon. *arXiv preprint arXiv:1707.04071*, 2017.

[11] Urrutia Jorge, Arevalo Alma, Chávez Rodrigo, Hernández Alejandro, López Ricardo, and Marín Nestaly. On rainbow quadrilaterals in colored point sets. *En preparacion*, 2020.

[12] Yoav Kallus. A linear-time algorithm for the maximum-area inscribed triangle in a convex polygon. *arXiv preprint arXiv:1706.03049*, 2017.

[13] D.E. Muller and F.P. Preparata. Finding the intersection of two convex polyhedra. *Theoretical Computer Science*, 7(2):217 – 236, 1978.

[14] János Pach and Géza Tóth. Monochromatic empty triangles in two-colored point sets. *Discrete Applied Mathematics*, 161(9):1259–1261, 2013.

[15] Franco P. Preparata and Michael I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, Berlin, Heidelberg, 1985.

[16] Günter Rote. The largest contained quadrilateral and the smallest enclosing parallelogram of a convex polygon. *arXiv preprint arXiv:1905.11203*, 2019.