Universidad Nacional Autónoma
de México

Facultad de Ciencias

GALERÍAS DE ARTE: POLÍGONOS ORTOGONALES Y
UNA IMPLEMENTACIÓN PARA ILUMINAR EL PLANO

TESIS

QUE PARA OBTENER EL TÍTULO DE:
LICENCIADO EN CIENCIAS DE LA COMPUTACIÓN

PRESENTA:
ALEJANDRO HERNÁNDEZ MORA

DIRECTORES DE TESIS:
DRA. ADRIANA RAMÍREZ VIGUERAS
DR. JORGE URRUTIA GALICIA

Ciudad Universitaria, Cd. Mx., 2017

1. Datos del alumno

Hernández
Mora
Alejandro
63 66 44 71
Universidad Nacional Autónoma de México
Facultad de Ciencias
Ciencias de la Computación
308551392

2. Datos del tutor

Dra.
Adriana
Ramírez
Vigueras

3. Datos del tutor

Dr.
Jorge
Urrutia
Galicia

4. Datos del sinodal 1

Dr.
José David
Florez
Peñaloza

5. Datos del sinodal 2

Dr.
Carlos Bruno
Velarde
Velázquez

6. Datos del sinodal 3

Dr.
Canek
Peláez
Valdés

7. Datos del trabajo escrito

Galerías de arte: Polígonos ortogonales y una implementación para iluminar el plano.
98p
2017

Agradecimientos

Antes que todo, quiero dedicarle este trabajo a mis abuelos Ángel y Cira, que físicamente no me pudieron acompañar hasta el final de esta etapa, pero sé que están conmigo, día a día en mis recuerdos y mis pensamientos. Y también a mis abuelos Esperanza y Temo, que junto con sus respectivas parejas, me tienen consentido desde el día en que llegué a este mundo y eso es algo que no todas las personas tienen la dicha de recordar.

Mamá, papá: Muchas gracias por apoyarme hasta en los momentos más difíciles. Sin ustedes este camino, simplemente no hubiera sido posible. Gracias papá por el esfuerzo que hiciste a lo largo de tantos años y apoyarme en hasta el más mínimo detalle que te fuera posible, siempre serás mi modelo a seguir. Mamá, gracias por el día a día que vivimos durante tanto tiempo, tú eres el sostén de mi vida, en todos los aspectos; escolar, personal y cotidiana; sin ti simplemente nuestro hogar sería inhabitable en casi todos los sentidos.

Dann: Qué triste, que no todos los hermanos tengan la suerte de tener a alguien como tú por hermano, es un sentimiento indescriptible. Eres mi compañero de casa, de cuarto, de banda, de FIFA, de conciertos y millones de cosas más. No puedo imaginarme mi vida universitaria y cotidiana sin ti.

A mi asesor, el Dr. Jorge Urrutia, a quien le tengo una admiración y respeto muy altos. Además de enseñarme algoritmos, que es lo que más me gusta, me enseño un poco más de cerca el mundo de la investigación, motivándome a explorar un poco más este camino.

A Adriana. Aunque el papel diga que es mi cotutora, fue quien me llevo de la mano a lo largo de todo este proyecto; me jaló las orejas, me dio los mejores consejos, asesorías, tips, correcciones, y más cosas inumerables. Además de ser una excelente tutora, es una gran persona y motiva a todos sus estudiantes a dar lo mejor. Nos enseñaste que la formalidad no está peleada con un ambiente más amigable. Eres un ejemplo a seguir.

Muchas gracias a el resto de mis sinodales, los doctores Canek, David y Carlos, que dedicaron tiempo y esfuerzo a revisar mi trabajo, por sus correcciones y su paciencia.

Al club de los “Forever tesistas”, que durante un año tuvimos nuestras reuniones de tesis, que nos sirvieron a todos para aprender a escuchar, analizar y aconsejar a un compañero. Una equipo formado gracias a Adriana, y es una experiencia no podré olvidar.

A todos mis compañeros y amigos de carrera, que hicieron de estos años una etapa inolvidable, les tengo un cariño y un aprecio incomparable. Hicimos de nuestro grupo la mejor generación de computólogos que puede haber.

Adri, jamás me imaginé tener la fortuna de encontrar en nuestra hermosa facultad a alguien como tú, con quien compartir tantas cosas. Gracias por obligarme a trabajar en mi tesis cuando más flojera tenía. Eres la mejor, te quiero muchísimo y te admiro como no tienes idea.

A todas y cada una de las personas que integran y han pasado por “Malix”; nuestra música, producción y ejecución; son todo un proyecto de vida, compartir fines de semana con ustedes ha sido increíble. En especial quiero agradecer a Gerardo, chavo. No sólo eres el mejor amigo que uno puede pedir, sino un compañero de trabajo, una excelente persona y un ser admirable.

No podían faltar mi agradecimientos a la “Clika”: Mario, Gerardo, Paloma, Juan, Adri, Mike, Sandra y Roy. Con ustedes he pasado los momentos más inolvidables de mi vida, y aunque no nos veamos como quisiéramos, son los mejores amigos del mundo, sin nuestras reuniones y cervezas, el estrés de estos años hubiera sido incontenible.

Muchas gracias a CONACyT, que me apoyó con una beca en la primer etapa del desarrollo de este trabajo.

Por último, gracias a la UNAM, la máxima casa de estudios y en particular a la Facultad de Ciencias, por darme la formación académica que tengo. Gracias a todos los profesores y ayudantes que tuve a lo largo de la carrera, sin ustedes ésta institución sería nada. No sólo tuve la fortuna de tener a grandes académicos en frente, sino también a grandes ejemplos a seguir, desde mi iniciación en la licenciatura.

Gracias a todos ustedes, este camino no sólo fue más fácil, sino divertido y ameno también.

Índice general

Índice de figuras

Índice de tablas

Introducción

Imagine que usted está a cargo de la seguridad de una prisión con un amplio patio que es adyacente a algunas calles, esto hace del patio un escenario perfecto para una fuga. Su labor es mantener vigilado el patio las 24 horas de manera permanente. Para evitar una posible fuga usted cuenta con un conjunto de cámaras inmóviles y de postes fijos en el patio donde puede colocar las cámaras. La pregunta es ¿de qué manera puede colocar dichas cámaras en los postes, de tal forma que se pueda monitorear el patio completo?

Lo anterior es en realidad un problema de Geometría Computacional mejor conocido como el ***problema de la iluminación***. En este problema, contamos con un conjunto de reflectores con posiciones fijas, y queremos decidir si el plano puede ser iluminado utilizando este conjunto de reflectores. En este trabajo se expone una solución eficiente al problema de la iluminación que se calcula en un tiempo de $O(n\log(n))$; además se enuncia también un resultado fuerte para el conjunto de algoritmos para particionar conjuntos de puntos.

Un tema clásico de la Geometría Computacional es el ***problema de la Galería de Arte***, que consiste en iluminar un polígono simple mediante de un conjunto de lámparas. Este conjunto de problemas de iluminación en el plano, nos da como resultado muchas aplicaciones en el campo de la Geometría Computacional, extendiéndose así hasta otros campos como el espacio (es decir en $\mathbb{R}^3$).

En los capítulos 4, 5 y 6; la organización se divide en dos partes. En la primera parte se enuncia un fragmento del problema de manera teórica, con las herramientas que se irán desarrollando. Posteriormente se explica de manera breve una propuesta de implementación para cada una de estas herramientas y sus respectivos algoritmos, haciendo énfasis especial en el modelo práctico que se propone y también en la complejidad de los algoritmos y de las pruebas de eficiencia que cada uno requiera.

Se incluye un breve capítulo introductorio, donde se hacen las definiciones necesarias para enunciar correctamente los problemas estudiados a lo largo de este trabajo; también se da una breve introducción al análisis de algoritmos, lo cual es necesario para poder explicar de manera adecuada la eficiencia de la solución final calculada a partir de cada una de las partes que la forman.

En el capítulo 2 se da un contexto histórico de los problemas de *galerías de arte*, además de proporcionar al lector múltiples modificaciones del mismo, abriendo la puerta a otros problemas similares. En la segunda parte se habla sobre la iluminación de polígonos ortogonales, dando un algoritmo y calculando una cota para el conjunto de reflectores utilizados. También se explora una solución diferente que mejora en cierto modo la apertura utilizada por los reflectores.

En el capítulo 3 se explica la necesidad de dividir al problema de la iluminación en dos etapas, en la primera se explica cómo lograr la iluminación de una cuña en el plano. En la segunda etapa se da un algoritmo para partir en tres una nube de puntos en el plano, por lo que los capítulos 4 y 5 estarán dedicados a sus respectivos problemas.

En el capítulo 6 se explica la conjunción de ambos resultados para lograr así la solución al problema de la iluminación, en un tiempo eficiente. Posteriormente se hace una muestra breve de la instalación y uso de la aplicación propuesta en los capítulos 4 y 5.

De esta manera se hace notar que éste es un trabajo teórico-práctico, proporcionando además de este trabajo escrito una aplicación que resuelve el problema de la iluminación antes mencionado, partiendo desde la solución teórica propuesta en 1997 en el artículo "El problema de la iluminación (The Floodlight Problem)" [1].

Capítulo 1

Conceptos básicos

En este capítulo se introducirán los conceptos más importantes que se utilizarán a lo largo de este trabajo, es en términos de estas definiciones que se enuncian todos los problemas estudiados. El capítulo empieza con todo lo relacionado con el análisis de algoritmos, de esta manera podremos ir explicando las complejidades de los algoritmos que llevamos a cabo, al mismo tiempo que vamos proponiendo el modelo para resolver el problema de la iluminación. Esto último nos permite ir construyendo el modelo teórico y el práctico, explicando los algoritmos relacionados con éstos.

1.1. Análisis de algoritmos

Como se menciona en la introducción, este trabajo no solamente muestra una forma de resolver el problema de la iluminación, sino que además es una solución óptima, pero ¿qué quiere decir que una solución sea óptima? Abusando un poco del lenguaje, significa que no hay una solución más rápida que ésta, sin embargo esto no quiere decir que sea una solución única. Lo que significa, es que podemos encontrar muchas soluciones óptimas y como no hay una más rápida que otra, entonces concluimos que todas tardan el mismo tiempo en ser calculadas. Decimos que es un abuso de lenguaje decir que una solución es óptima, porque en realidad la solución por sí misma no es rápida, lenta, mejor o peor. La solución simplemente es el resultado al que queremos llegar. Lo que es en realidad óptimo o decimos que tarda algún tiempo, es el ***algoritmo*** que calcula esa solución.

La definición de algoritmo es un tanto simple, lo interesante es que estos algoritmos tengan un propósito o un problema a resolver.

Definición 1.1. Un ***algoritmo*** es una serie *finita y ordenada* de pasos, que siempre terminan. Un algoritmo trabaja con un conjunto de valores iniciales, a los que les llamamos **entrada**, y también genera un conjunto de valores al terminar, al los que llamamos **salida**. A la cantidad de datos que tiene la entrada, le llamamos ***tamaño de la entrada***. Tanto entrada como salida del algoritmo, son no acotadas, lo cual quiere decir que pueden ser tan grandes o pequeñas como sea necesario.

En Ciencias de la Computación nos interesan muchas cosas cuando resolvemos un problema. Lo más importante es hacer que quede resuelto correctamente, sin embargo eso no es todo. Muchas veces nos interesa que el algoritmo que calcula esa solución lo haga de la forma más rápida posible, pues cabe la posibilidad de que en un futuro cercano la solución que obtenemos ya se haya vuelto obsoleta, por ejemplo: Salimos de casa a un lugar pero no conocemos el camino, utilizamos una aplicación en un dispositivo móvil que nos ayude a encontrar el camino más rápido según el tránsito. Si la aplicación se tarda el tiempo suficiente como para que el tránsito cambie considerablemente, entonces la solución que nos da como resultado ya no es útil, pues el camino que nos trazó contempla el tránsito en el momento que solicitamos la ruta más rápida, haciendo obsoleta nuestra solución obtenida.

Además de preocuparnos por que el problema se resuelva de manera eficiente, hay otros factores a tomar en cuenta, por ejemplo minimizar el espacio de memoria que ocupamos. Es una idea que en principio no nos molesta, pues hoy en día las computadoras con las que incluso contamos en casa, tienen una capacidad suficiente como para desperdiciar un poco de memoria extra. Ésto no es siempre lo mejor, pues un teléfono celular o algún otro dispositivo móvil cuentan con una cantidad de memoria considerablemente limitada y quisiéramos ahorrar la mayor cantidad posible de ésta.

El **análisis de algoritmos** es justamente la rama de las ciencias de la computación que se encarga de estudiar la forma en que encontramos una solución, para poder hacer eficiente el uso del tiempo de cómputo y memoria utilizada. Muchas personas en el campo de las ciencias de la computación consideran que esto es lo más importante, pues hoy en día las exigencias de los sistemas son cada vez mayores por la cantidad de datos con las que se trabajan y porque cada vez es más difícil fabricar componentes aún más pequeños y con capacidad mayor. Por esto nos fijamos en aprovechar el espacio y tiempo de cómputo y así lograr el mejor desempeño posible.

Ya conocemos la motivación de estudiar análisis de algoritmos y hasta cierto punto es fácil imaginar, que algo así es necesario en el desarrollo de software y sistemas, pero ¿cómo medimos el tiempo que se tarda un algoritmo en terminar? o ¿cómo decidimos si un algoritmo es más rápido que otro? Ésta es una pregunta que podríamos responder de manera intuitiva y casi automática, podríamos pensar en poner una computadora a correr los dos algoritmos, tomar el tiempo que la computadora se tarde y ver cuál fue más rápido. También podríamos poner dos computadoras a correr los diferentes algoritmos y ver cuál termina primero. Ambas ideas son intuitivamente buenas, pero incorrectas.

Para la primera idea, el fallo está en que no importa cuántas veces ejecutemos un programa en una computadora, el desempeño siempre va a ser distinto. Esto sucede por varios motivos y la causa de algunos puede ser el software. Si corremos el programa con ninguna otra aplicación abierta el desempeño es uno, pero si abrimos muchas aplicaciones es altamente probable que el desempeño se modifique, pues la computadora debe de repartir la memoria disponible en todas las tareas que está llevando a cabo.

Otra cosa a tomar en cuenta es que el sistema operativo está en constante cambio y tiene la prioridad más alta en una computadora, dando menor prioridad a nuestra tarea. También podría ser por razones físicas, como cuestiones de la temperatura del CPU.

Incluso suponiendo que logramos igualar las condiciones para ejecutar un programa en una computadora, existen otros factores más importantes a tomar en cuenta. Es bastante común que para un mismo algoritmo se encuentren desempeños diferentes, ¿cómo es esto posible si el algoritmo no cambia?

Un ejemplo muy común para ejemplificar lo anterior es el algoritmo de ordenamiento ***quick sort***. Lo que este algoritmo hace es tomar un elemento arbitrario llamado pivote, a partir de éste comienza su ejecución. El desempeño del algoritmo depende mucho del pivote que tomemos, de manera que si tenemos un buen pivote el desempeño será mejor, pero de lo contrario puede ser muy malo. Lo más grave de este ejemplo en particular es que si quisiéramos ordenar un conjunto de elementos que ya está ordenado (pero nosotros ignoramos esto), el desempeño es mucho peor de lo esperado.

A diferencia del *quick sort*, podemos tener otros algoritmos de ordenamiento como el ***merge sort***, que sin importar la entrada el desempeño no varía. Claro que muchas veces tenemos algún costo a cambio de la seguridad del desempeño constante, y la mayoría de las veces requerimos de un poco más de memoria.

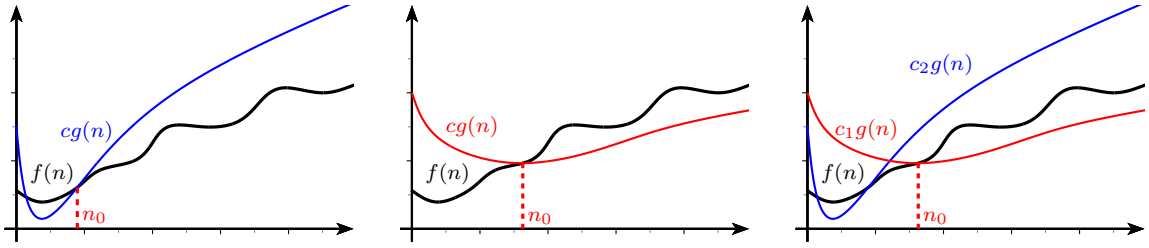
Lo anterior lejos de resolver nuestra duda, la hace aún más grande. ¿Cómo comparar el tiempo que se tardan dos algoritmos? La respuesta la podemos encontrar en la definición de un algoritmo. Si un un algoritmo es un conjunto ordenado finito de pasos, lo más simple es contar el número de pasos que utilizan ambos algoritmos y compararlos, así sabremos cuál es el más rápido. Esta última idea parece bastante razonable, pues la manera de hacer diferencia entre ambos algoritmos no depende de ninguna manera de una computadora. Para no caer en problemas como el que comentamos sobre el algoritmo ***quick sort***, vamos a tomar siempre el peor desempeño posible del algoritmo, de esta forma nuestra comparación no depende de casos particulares.

Ya sabemos que el tiempo que se tarda un algoritmo lo mediremos en función del número de pasos que realiza, pero hay otro factor importante a tomar en cuenta y esto es la entrada del algoritmo, en particular nos interesa el tamaño de la entrada.

Hay diferentes maneras de medir el tiempo que tarda un algoritmo en ser ejecutado, todas ellas toman como parámetro la entrada del algoritmo, así que al final el desempeño medido está en función del tamaño de la entrada. Dependiendo de el tiempo que se tarde el algoritmo en ser ejecutado, vamos a etiquetar el desempeño en diferentes órdenes a los que llamaremos ***órdenes de complejidad*** o simplemente nos referiremos al tiempo que tarda el algoritmo como la ***complejidad de tiempo*** del algoritmo.

Los órdenes de complejidad dependerán de las diferentes maneras de medir la complejidad de un algoritmo. A todas estas maneras les relacionamos una función que va a servir como punto de comparación del tiempo que tarda el algoritmo con relación a la entrada, en algunas ocasiones dicha función sirve como una cota.

Para diferenciar la forma en que medimos la complejidad y también dar algunos de los diferentes órdenes de complejidad, enunciamos las siguientes notaciones para calcular la complejidad de un algoritmo, todas están tomadas del libro "Introduction to Algorithms" [2].



(a) Notación O. **(b)** Notación Ω. **(c)** Notación Θ.

Figura 1.1: En este ejemplo la función para las notaciones Ω y O resultaron también ser una función para la notación Θ

Definición 1.2. La ***notación O grande*** (o ***cota superior asintótica***), mide la complejidad de un algoritmo acotando por arriba a la función que representa el rendimiento del algoritmo. Sean $f(x)$ la función que representa el rendimiento del algoritmo en cuestión, $g(x)$ otra función. Decimos que $f(x) \in O(g(x))$, si existen $n_0 \in \mathbb{N}$ y $c \in \mathbb{R}$, tales que para toda $n \geq n_0$, se cumple que:

$$0 \leq f(n) \leq cg(n), \text{ con } 0 < c$$

Esta notación básicamente nos va a dar el desempeño del algoritmo en el peor caso según la entrada del algoritmo. También es la notación más utilizada en el área del análisis de algoritmos, a partir de ésta es que vamos a definir los órdenes de complejidad de los algoritmos a lo largo de este trabajo.

La notación O nos ayuda a expresar la complejidad de un algoritmo, en términos del proceso más grande. Por ejemplo: Supongamos que tenemos un algoritmo que tiene tres subrutinas de órdenes $O(n)$, $O(1)$ y $O(\log(n))$, entonces el tiempo de ejecución sería de $O(n) + O(1) + O(\log(n))$, sin embargo la definición de O nos indica que la complejidad es de $O(n)$. Lo anterior tiene sentido, porque si pensamos que nuestro algoritmo tiene diferentes procesos, el que va a retardar la ejecución del algoritmo completo es el proceso que tarde más.

Definición 1.3. La ***notación*** Ω (o ***cota inferior asintótica***) mide la complejidad de un algoritmo acotando por abajo a la función que representa el rendimiento del algoritmo. Sean $f(x)$ la función que representa el rendimiento del algoritmo en cuestión, $g(x)$ otra función. Decimos que $f(x) \in \Omega(g(x))$, si existen $n_0 \in \mathbb{N}$ y $c \in \mathbb{R}$, tales que para toda $n \geq n_0$, se cumple que:

$$0 \leq cg(n) \leq f(n), \text{ con } 0 < c$$

Esta notación es poco común para comparar algoritmos, sin embargo es bastante práctica, pues de esta manera conoceremos siempre el tiempo mínimo que requiere un algoritmo para terminar.

Definición 1.4. La ***notación*** Θ (o ***cota asintóticamente justa***) mide la complejidad de un algoritmo acotando por arriba y por abajo a la función que representa el rendimiento del algoritmo. Sean $f(x)$ la función que representa el rendimiento del algoritmo en cuestión, $g(x)$ otra función, $c_1 \neq 0$, $c_2 \neq 0$ y n_0 valores constantes. Decimos que $f(x) \in \Theta(g(x))$, si existen $n_0 \in \mathbb{N}$ y $c_1, c_2 \in \mathbb{R}$, tales que para toda $n \geq n_0$, se cumple que:

$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n), \text{ con } 0 < c_1 \text{ y } 0 < c_2$$

En otras palabras, la notación Θ nos dice cuánto es el tiempo mínimo requerido para que el algoritmo termine y el tiempo máximo que se puede tomar, por esa razón es que decimos que es una cota justa. Formalmente podemos decir que si $f(n) \in O(g(n))$ y además $f(n) \in \Omega((g(n))$, entonces $f(n) \in \Theta(g(n))$.

Existen más formas de medir la complejidad de un algoritmo, pero estas tres son las más comunes y son suficientes para el desarrollo de este trabajo. De hecho únicamente utilizaremos la notación O grande.

Si ponemos especial atención a la definición de la notación O grande, podemos ver que una función $f(n) \in O(g(n))$, si existe una constante c tal que multiplicada por $g(n)$, hace que la función f siempre quede por abajo, esto implica que referirnos a $O(g(n))$ es lo mismo que decir $O(cg(n))$. En otras palabras las constantes que multiplican a g desaparecen, pues ambas funciones están representadas en el mismo orden de complejidad. Por esto mismo, decidimos que no tomaremos en cuenta a las constantes para la división de estos órdenes, así $O(3n)$ será lo mismo que referirnos a $O(n)$, por lo que no se escribiría la constante 3, o cualquier otra

Como bien dijimos antes, los órdenes estarán calculados en función de la entrada del algoritmo, por lo que el argumento que reciben las funciones f y g, son el tamaño de la entrada del algoritmo que estamos analizando. Lo que en el fondo estamos diciendo es que la complejidad (en cualquier notación) nos dice *por cada elemento en la entrada del algoritmo, cuantos pasos se van a ejecutar hasta calcular la solución final.*

Lo que buscamos en análisis de algoritmos es siempre tardarnos el menor tiempo posible. Esto se logra cuando el número de pasos que requiere el algoritmo, no depende de la entrada y siempre tarde lo mismo, es decir un número ***constante*** de pasos. Es así como le llamamos a la complejidad más baja posible.

Un ejemplo, considere un algoritmo cuya entrada es de tamaño variable, sin embargo el algoritmo asegura que siempre se tarda 50 cálculos en terminar, eso es un número constante de pasos, pues no depende de la entrada. Todos desearíamos siempre tener un número constante de pasos para calcular la solución a cualquier problema, independientemente del tamaño de la entrada, desafortunadamente eso no siempre es posible.

Cuando tenemos un algoritmo que se toma un número constante de pasos por cada elemento en la entrada, decimos que la complejidad es ***lineal*** con respecto a la entrada. Por la manera en que está definida la notación O grande, en esencia es como si nos tomáramos un paso por cada elemento en la entrada, pues debemos recordar que $O(cg(x)) \in O(g(x))$.

Hay una infinidad de posibles complejidades para cualquier algoritmo, en la **Tabla** 1.1 podemos ver las complejidades más comunes en el análisis de algoritmos.

Complejidad	Nombre
$O(1)$	Constante
$O(\log(n))$	Logarítmica
$O(\sqrt{n})$	Sublineal
$O(n)$	Lineal
$O(n\log(n))$	Lineal logarítmica
$O(n^2)$	Cuadrática
$O(n^2\log(n))$	Cuadrática logarítmica
$O(n^3)$	Cúbica
$O(n!)$	Factorial

Tabla 1.1: Tabla con complejidades ordenadas de manera ascendente.

1.2. Preliminares geométricos

Las siguientes definiciones serán trabajadas en $\mathbb{R}^2$, fueron tomadas de los libros de Geometría Computacional [3] y [4]. En caso de necesitar algún concepto menos recurrente se definirá en el momento que sea necesario. Además de las definiciones formales, se propone un modelo de manera que cada una se pueda modelar mediante un programa o aplicación.

Definición 1.5. Definimos un ***punto*** como lo hacemos de manera clásica en geometría, es una ubicación en el espacio ($\mathbb{R}^2$) con coordenadas x e y.

Definición 1.6. Una ***nube de puntos*** es un conjunto de puntos en el plano.

Definición 1.7. Decimos que tres puntos diferentes son ***colineales*** si existe una recta que pasa por los tres puntos.

Definición 1.8. Decimos que una nube de puntos está en ***posición general***, si para todos tres puntos de la nube, no son colineales.[1]

Definición 1.9. Sean $\boldsymbol{x}$, $\boldsymbol{y}$ y $\boldsymbol{z}$ puntos en el espacio. Si $\boldsymbol{z}$ se encuentra a la *izquierda (derecha)* de la recta dirigida de $\boldsymbol{x}$ a $\boldsymbol{y}$, al camino que empieza en $\boldsymbol{x}$ va en línea recta a $\boldsymbol{y}$ y de ahí en línea recta a $\boldsymbol{z}$ le llamamos ***vuelta a la izquierda (derecha)***, se muestra un ejemplo en las **Figuras** 1.2 y 1.3 respectivamente.

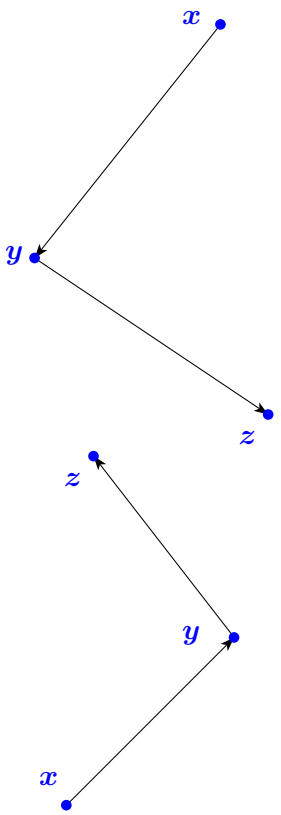


Figura 1.2: El camino que va de x, a y y termina en z forma una vuelta a la izquierda.

Definición 1.10. Una ***cuña*** es el área del plano que está acotada por dos rayos que emanan de un mismo punto llamado ápice u origen.

Definición 1.11. El ***ángulo de una cuña*** es el ángulo que se forma entre los rayos que la forman.

Definición 1.12. Dada una cuña, decimos que su ***cuña complementaria*** es la cuña que resulta de rotar 180° los rayos que la forman, cabe destacar que la cuña complementaria tiene el mismo ángulo que la cuña original.

[1] En los libros citados, la ***posición general*** se define como una configuración en la nube de puntos que evite el uso de casos especiales (en los que se pueda presentar algún problema con el algoritmo). A nosotros nos interesa que no haya elementos colineales por razones que se explicarán más adelante, así que esta es la definición para este concepto que utilizaremos el resto del trabajo.

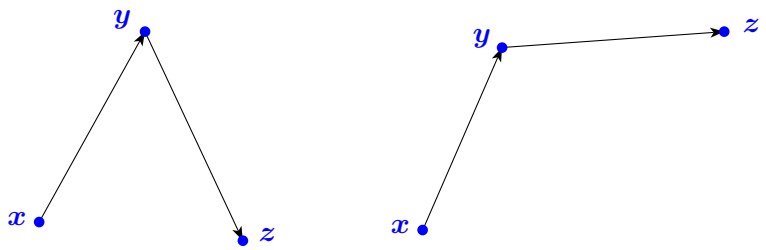


Figura 1.3: El camino que va de x, a y y termina en z forma una vuelta a la derecha.

Definición 1.13. Un ***reflector*** es una cuña que ilumina el área que está entre los dos rayos que la forman, decimos entonces que el reflector ilumina un ángulo de α, donde α es el ángulo de la cuña. Durante el desarrollo de este trabajo vamos a utilizar reflectores pequeños, es decir cuyos ángulos son menores a π radianes, más adelante se explicará el motivo. Podemos encontrar el ejemplo de un reflector en la **Figura** 1.4.

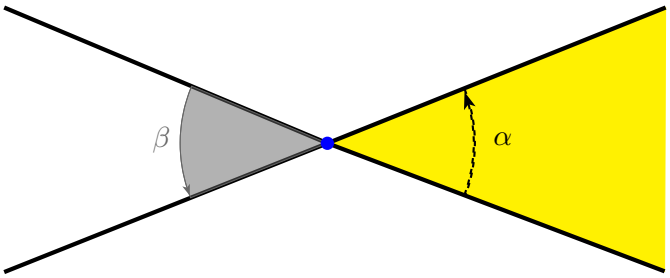
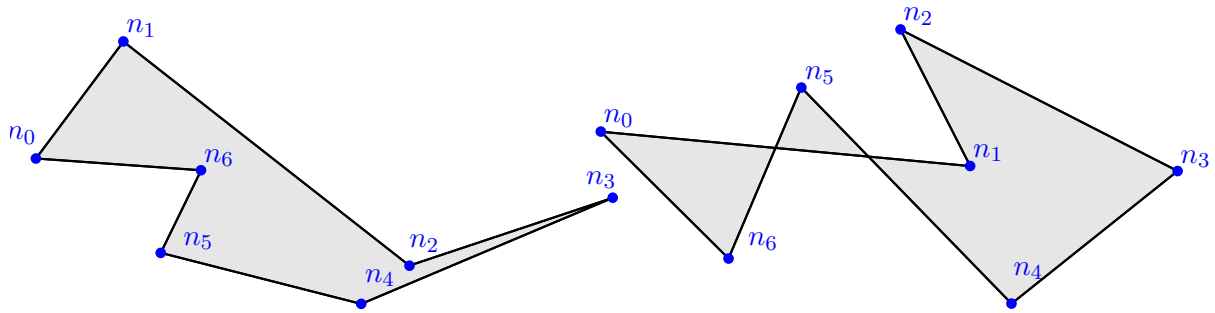


Figura 1.4: Ejemplo de un reflector que está formado por una cuña. El ángulo α corresponde a la cuña original, el ángulo β a su cuña complementaria. Por la definición misma es claro que $\alpha = \beta$. En la imagen el área amarilla es el área que ilumina.

Éste es el conjunto de definiciones fundamentales que vamos a utilizar, sobre todo cuando trabajemos en el *problema de la iluminación*. Como podemos observar, no son muchas definiciones, pero sí importantes, pues en términos de ellas es que se define el mismo. Las siguientes definiciones también se utilizarán a lo largo de este trabajo, pero sobre todo en los primeros capítulos introductorios a los problemas de *galerías de arte* y en la iluminación de polígonos ortogonales.

Definición 1.14. Sea n un número natural tal que $n \geq 3$. Un ***polígono*** es un conjunto $V = \{v_1, v_2, ..., v_n\}$ de n puntos; un conjunto $E = \{(v_1, v_2), (v_2, v_3), ..., (v_{n-1}, v_n), (v_n, v_1)\}$ de n segmentos de recta, las cuales unen a dos puntos de P. Al conjunto de segmentos de recta les llamaremos aristas del polígono y al conjunto de puntos les llamaremos vértices de polígono. Las aristas unen a un vértice con su sucesor y al último con el primero. **Figura** 1.5.

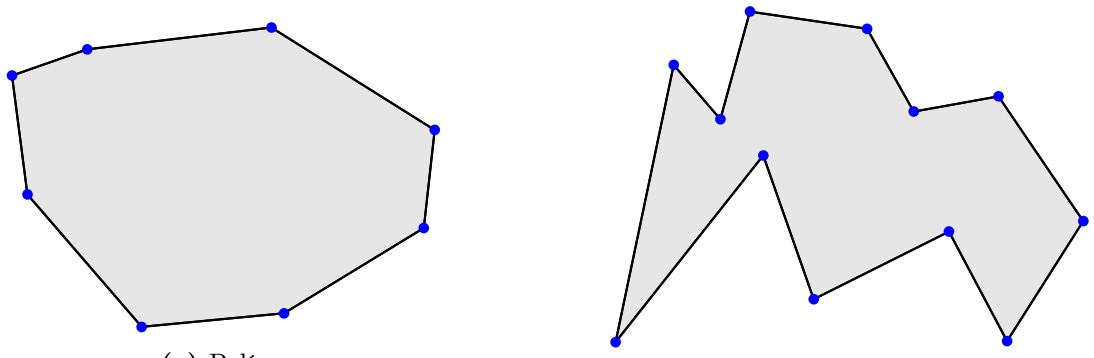
Definición 1.15. Un ***polígono simple*** es un polígono en el que no hay ninguna intersección entre sus aristas (salvo los vértices en los que inciden ellas). Podemos encontrar un ejemplo en la **Figura** 1.5(a).



(a) Polígono simple. **(b)** Polígono no simple.

Figura 1.5: Ejemplos de polígono simple y no simple.

Definición 1.16. Un ***polígono convexo***, es un polígono simple que tiene todos sus ángulos internos menores a π. **Figura** 1.6(a).

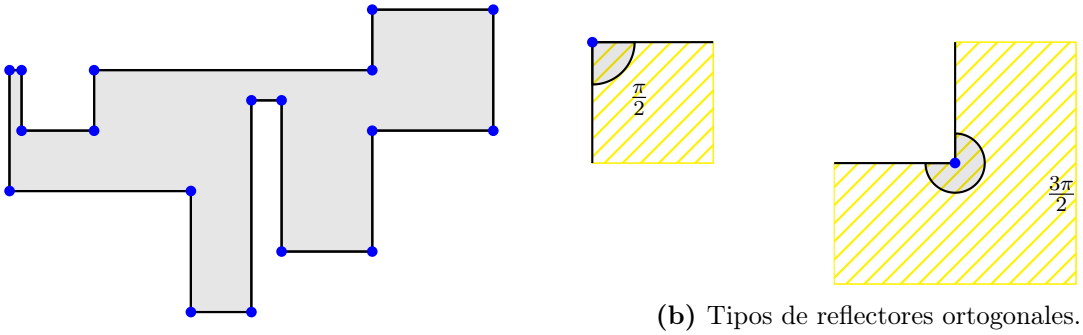


(a) Polígono convexo. **(b)** Polígono no convexo.

Figura 1.6: Ejemplos de polígono convexo y no convexo.

Definición 1.17. Un ***polígono ortogonal*** es un polígono en el que dadas dos aristas incidentes en cualquier vértice del polígono, el ángulo interior que forman tiene un valor de $\frac{\pi}{2}$ o de $\frac{3\pi}{2}$. Podemos ver un ejemplo en la **Figura** 1.7(a).

Definición 1.18. Un ***reflector ortogonal*** es un reflector que tiene un ángulo de iluminación de $\frac{\pi}{2}$ o de $\frac{3\pi}{2}$. Se muestra un ejemplo en la **Figura** 1.7(b).



(a) Polígono Ortogonal.

(b) Tipos de reflectores ortogonales.

Figura 1.7: Ejemplo de polígonos y reflectores ortogonales.

Definición 1.19. Sea P un polígono. Un ***hoyo*** de P es un polígono simple que está al interior de P.

Definición 1.20. Sea P un polígono. Una ***diagonal*** de P es una arista al interior de P que conecta a dos vértices de P.

Definición 1.21. Sea P un polígono. Una ***triangulación*** T de P es $P \cup D$, donde D es el conjunto máximo de diagonales que puede tener P, tales que ninguna de ellas se intersecta con alguna otra diagonal. T no es única, pues el conjunto D no es único. Podemos ver un ejemplo en la **Figura** 1.8.

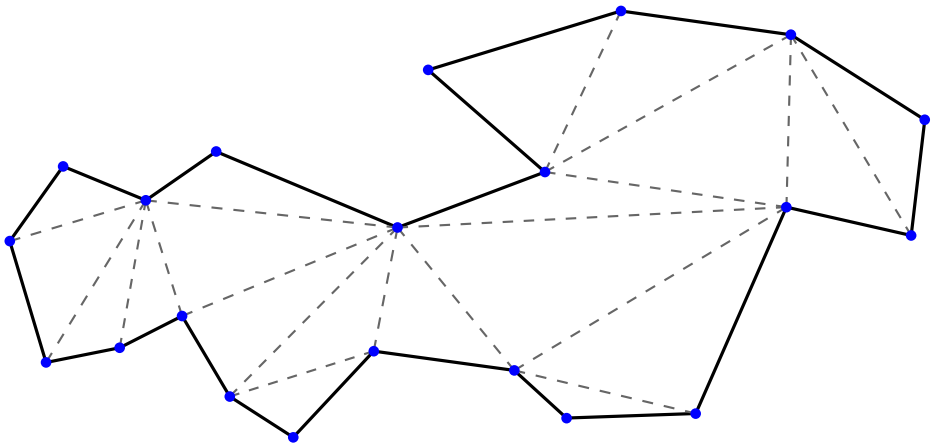


Figura 1.8: Una triangulación del polígono.

Definición 1.22. Una ***gráfica*** G es un conjunto finito de vértices V, y un conjunto de aristas E. E está formado por subconjuntos de V de tamaño dos. La representación de una gráfica es mediante nodos que serán los vértices y segmentos de recta entre dos vértices que representan a las aristas. Denotaremos a G como $G = (V, E)$. La notación que utilizaremos para una arista será $e = uv$, donde $\{u, v\} \subset V$, decimos que u y v son *adyacentes* o *vecinos*.

Definición 1.23. Sea T una triangulación. La ***gráfica dual de*** **T** es una gráfica que tiene un vértice por cada uno de los triángulos de T. Para v un vértice de la gráfica, representaremos a su triángulo relacionado en T como t_v. Los vértices u y v de la gráfica van a estar unidos por una arista, si dos triángulos t_v y t_u comparten una arista en T. Representaremos a la *gráfica dual de* T como $G(T)$. Es importante notar que t_u y t_v comparten a lo más una arista. En la **Figura** 1.9 podemos ver la gráfica dual de la triangulación en la **Figura** 1.8.

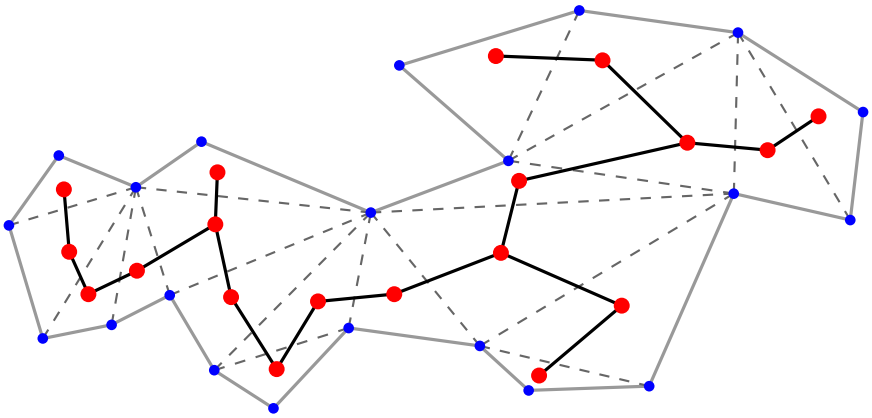


Figura 1.9: Ejemplo de una gráfica dual de una Triangulación. Podemos ver que los vértices de la gráfica son de color rojo, las líneas grises son las aristas del polígono, las líneas punteadas son las aristas al interior del polígono que forman a la triangulación T, las aristas negras son parte de la gráfica dual de T. Así $G(T)$ está formada por los vértices rojos y las aristas negras.

Definición 1.24. Sea G una gráfica. Una ***k-coloración*** de G es una función c, que va de los vértices de G a un subconjunto K de los números naturales de tamaño k, donde a un vértice le asignamos un número en K si y sólo si ninguno de sus vértices adyacentes tiene el mismo número asignado. Se le llama *k-coloración* porque a lo más se utilizan k elementos, los números hacen alusión a un color y cada uno de los vértices de G tiene un color asignado.

1.3. Implementación

Como dijimos al inicio del capítulo, se proponen unas definiciones prácticas de manera que el lector podría modelar el problema de la iluminación en el lenguaje de programación de su preferencia. El modelo que a continuación se muestra, es completamente propuesto por el autor de este trabajo. Por la naturaleza del problema, se decidió utilizar el paradigma orientado a objetos pues habrá algunas funciones que deben poder modelar el comportamiento de cada objeto geométrico definido en la sección anterior.

Para modelar un punto tenemos la clase ***Punto***, la cual simplemente tiene como atributos las coordenadas de un punto, de esta manera una nube de puntos la podemos modelar con un arreglo o lista del tipo de este objeto.

```
1 public class Point{
2   private final double x, y;
3   /**
4     * Class constants for turn directions.
5     */
6   public static final int LEFT =1, RIGHT =0, COLINEARS =-1;
7 }
```

Para crear una nube de puntos en posición general, generamos un par de puntos de manera aleatoria. Posteriormente se genera un nuevo punto p y se verifica que no sea colineal con cada una de las parejas posibles del resto de puntos. Si resulta que son colineales se genera un nuevo punto hasta que se logra generar uno que no sea colineal con alguna pareja.

Vamos a analizar el tiempo que toma el algoritmo 1.1 en ser ejecutado. De la línea 1 a la línea 5, se inicializa una estructura de datos lineal y se agregan 2 puntos de manera aleatoria, el tiempo que toma este proceso es constante (puede ser lineal si la estructura es lineal). En la línea 6 estamos comenzando un proceso de tamaño lineal, es decir que está en términos del tamaño de la entrada. Vamos a hacer $n-2$ ejecuciones de algún proceso que nos toma un tiempo extra, hasta aquí la complejidad es mínimo de $O(1)+O(n)$ sin embargo falta ver cuanto tiempo nos va a tomar cada ejecución del ciclo de la línea 6.

De las líneas 7 a 9 lo que hacemos es generar un punto aleatorio y verificar que esté en posición general. Nuestro algoritmo toma cada par de puntos posible de la colección y verifica que el punto recién creado no sea colineal. En el peor de los casos los puntos que son colineales con el nuevo punto, son los últimos en nuestra estructura llamada *puntos*, por lo que nos tomaría n^2 pasos darnos cuenta de que no están en posición general. Esto hace que el proceso tenga una complejidad de $\Omega(n^2)$, puesto que podríamos repetir este paso más veces, hasta que estén en posición general.

Algoritmo 1.1: Algoritmo para crear una nube de puntos en posición general

```
1  Crear un arreglo(o una lista) llamado puntos de tamaño n;
2  punto = crearPuntoAleatorio();
3  puntos.agregar(punto);
4  punto = crearPuntoAleatorio();
5  puntos.agregar(punto);
6  for i := 1 to n do
7  |   repeat
8  |   |   punto = puntoAleatorio();
9  |   until punto.estaEnPosicionGeneral(puntos);
10 |   puntos.agregar(punto);
11 end
12 return puntos;
```

La verificación de la posición general está dentro de un proceso lineal, por lo que la complejidad real de el proceso del ciclo completo es $\Omega(n^3)$. La complejidad es amortizada de $O(n^3)$, pues es muy poco probable generar muchas veces puntos que no sean colineales, por esta misma razón, hay una probabilidad igual de baja de que el algoritmo no termine.

Utilizaremos la clase ***Vector*** para modelar vectores en $\mathbb{R}^2$, de esto no hablamos antes porque no es necesario tener esta definición en el modelo teórico, pero en el práctico es más que necesaria. Esta clase contará con dos objetos de la clase punto. El primer punto indica el inicio del vector, a este punto le llamaremos ***origen*** o ***inicio***. El segundo punto será para indicar dónde termina y le llamaremos ***destino*** o ***fin***, cabe destacar que es fundamental hacer distinción en estos 2 elementos, pues no es lo mismo dirigir el vector de un punto a otro que en el orden inverso.

```
1 public class Vector{
2   private final Point origin, generator;
3   private final double magnitude;
4 }
```

La clase ***Cuña*** está definida por dos rayos que emanan del mismo punto. Un rayo se define a partir de un vector director, por esto es que una cuña estará definida a partir de dos objetos de tipo vector, ambos tienen como origen al mismo punto. Esta definición nos lleva a deducir que con tres puntos distintos es más que suficiente para definir una cuña, el primero para el origen y los siguientes para el destino de ambos vectores.

```
1 public class Wedge{
2    protected final Point origin;
3    protected Vector v1, v2;
4    protected final double theta;
5 }
```

Es importante darnos cuenta de que hay dos posibles cuñas a formar a partir de estos dos vectores. Esto es dependiendo de cual ángulo es el que vamos a tomar como el ángulo de la cuña, ya que si dos segmentos de recta inician en el mismo punto, podemos hacer que el ángulo que forma, sea a partir de donde está un vector y en favor a las manecillas del reloj hasta que encontramos el otro vector, o del mismo modo pero girando en contra de las manecillas del reloj, así como se muestra en la **Figura** 1.10. Para liberarnos de esta ambigüedad, tomaremos el menor ángulo que se forme, esto es muy cómodo para el programador porque es una forma automática de generar cuñas aleatorias evitando así la confusión de cuál es la que se creó; esto va de la mano con el hecho de que trabajaremos siempre con cuñas cuyo ángulo sea menor o igual a π radianes, de otro modo tendría que ser posible generar cuñas de mayor apertura. Esto es exclusivamente porque la aplicación que se desarrolló en este trabajo, no utiliza vectores con apertura mayor, de lo contrario esto sería un grave error.

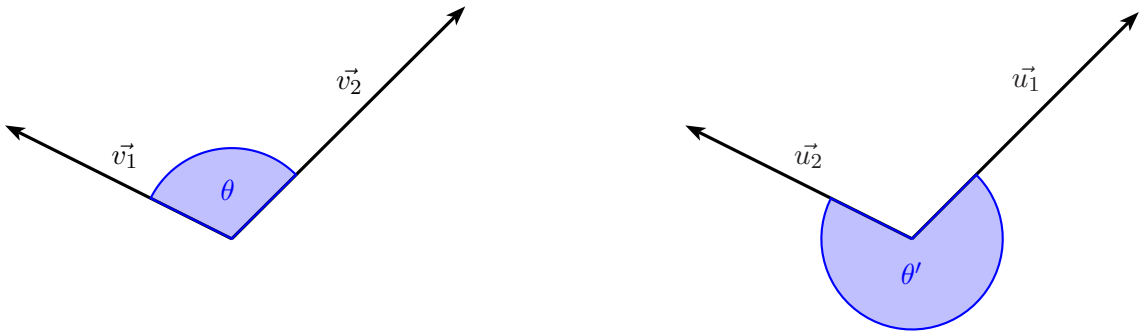


Figura 1.10: Aquí vemos ilustradas las dos posibles maneras de definir una cuña a partir de dos vectores con el mismo origen. Nosotros tomaremos siempre la cuña del ángulo θ. Notemos que el ángulo θ' es lo mismo que el ángulo $2\pi - \theta$.

Otra convención para nuestra definición de cuñas es que los vectores que la forman los llamaremos $\vec{v_1}$ y $\vec{v_2}$ respectivamente en orden a favor de las manecillas del reloj. Podemos ver en la **Figura** 1.10 el nombre de los vectores y la manera en que se forma la cuña con dos vectores, como dijimos antes la cuña formada por los vectores $\vec{u_1}$ y $\vec{u_2}$ no la podríamos formar con nuestro modelo, pues su ángulo es mayor a π.

La siguiente definición es una operación básica que utilizaremos a lo largo de todo este trabajo, nos referimos a la implementación de las ***vueltas a la izquierda y a la derecha***. Podríamos decir que ésta es la operación geométrica básica más recurrente en esta aplicación, pues es a partir de ésta que tomaremos las decisiones más importantes en la mayoría de los algoritmos que se utilizan en cada solución.

Supongamos que tenemos tres puntos, p, q y r. Queremos verificar si el camino que va de p a q y termina en r, forma una vuelta a la derecha, a la izquierda o no hay vuelta. Lo que haremos será tomar a los vectores dirigidos $\vec{pq}$ y $\vec{pr}$, ambos vectores tienen origen en el punto p, si giramos a alguno de los dos en sentido a favor de las manecillas del reloj, eventualmente al haber rotado un ángulo menor o igual a 180º encontraremos al otro vector. Para la siguiente descripción, el lector puede apoyarse de la **Figura** 1.11.

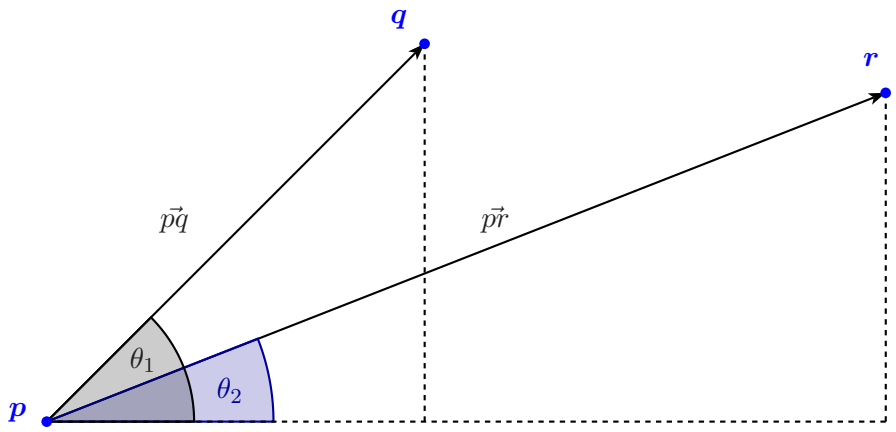


Figura 1.11: Triángulos que se forman con p, q y r. También se forman los ángulos θ_1 y θ_2.

Si rotamos $\vec{pq}$ en favor a las manecillas del reloj y menos de 180º, encontramos al vector $\vec{pr}$, decimos que es una vuelta a la derecha; si rotamos exactamente 180º, entonces decimos que son colineales; el caso restante es en el que rotamos en contra de las manecillas del reloj menos de 180º y encontramos a $\vec{pr}$, en este caso se trata de una vuelta a la izquierda.

Para poder verificar el sentido que tiene el giro entre los vectores, basta con calcular un determinante. El determinante está dado por la fórmula $det(\vec{pq}, \vec{pr}) = x_1y_2 - x_2y_1$, suponiendo que p está en el origen, $q = (x_1, y_1)$ y $r = (x_2, y_2)$. Para el caso en el que p está fuera del origen, el determinante está dado por la fórmula: $det(\vec{pq}, \vec{pr}) = (x_1 - x_0)(y_2 - y_0) - (x_2 - x_0)(y_1 - y_0)$, donde simplemente estamos trasladando los vectores al origen. La complejidad de este proceso es constante, pues es un cálculo que requiere de siete operaciones aritméticas, cinco sumas y dos multiplicaciones.

Teorema 1.25. Sean p, q y r puntos en el plano con coordenadas (x_0, y_0),(x_1, y_1) y (x_2, y_2) respectivamente. Si el determinante de los vectores $\vec{pq}$ y $\vec{pr}$ es positivo, entonces hay una vuelta a la izquierda; si el determinante es negativo, entonces es una vuelta a la derecha; si el determinante es igual a cero, entonces p, q, y r son colineales.

Demostración. Sean p, q y r puntos con coordenadas (x_0, y_0), (x_1, y_1) y (x_2, y_2) respectivamente. Considérense los dos triángulos rectángulos formados por los vectores $\vec{pq}$ y $\vec{pr}$, donde la hipotenusa de cada triángulo es la magnitud de su vector correspondiente.

Recordemos que $det(\vec{pq}, \vec{pr}) = (x_1 - x_0)(y_2 - y_0) - (x_2 - x_0)(y_1 - y_0)$, cuando el determinante de dos vectores es igual a cero, nos indica que éstos son paralelos o antiparalelos, donde eso quiere decir que apuntan en direcciones contrarias sobre la misma recta.

Apoyándonos de la ***Figura*** 1.11, podemos observar que si la pendiente m_1 de la recta que pasa por el vector $\vec{pq}$, es mayor a la pendiente m_2 de la recta que pasa por el vector $\vec{pr}$, se trata de una vuelta a la derecha, es decir que θ_1 es mayor que θ_2. Observemos que lo anterior se cumple independientemente de la magnitud de ambos vectores. Análogamente se trata de una vuelta a la izquierda, si se cumple que m_2 es mayor a m_1 y esto nos indica que θ_1 es menor que θ_2.

Por definición $m_1 = \frac{y_1 - y_0}{x_1 - x_0}$ y $m_2 = \frac{y_2 - y_0}{x_2 - x_0}$. Sin pérdida de generalidad, supongamos que m_1 es mayor a m_2, esto implica que:

$$\frac{y_1 - y_0}{x_1 - x_0} > \frac{y_2 - y_0}{x_2 - x_0} \implies 0 > (x_1 - x_0)(y_2 - y_0) - (x_2 - x_0)(y_1 - y_0) = det(\vec{pq}, \vec{pr})$$

Del razonamiento anterior, podemos concluir que si $m_1 > m_2$, se trata de una vuelta a la derecha y además $det < 0$. Análogamente si $m_1 < m_2$, se trata de una vuelta a la izquierda y $det > 0$. También vimos que si $det = 0$, entonces los puntos son colineales. ■

Notemos que en ningún momento es necesario conocer alguno de los dos ángulos, esta es una de las ventajas que tiene el uso del determinante, pues nos ahorramos el costo de operaciones trigonométricas y raíces cuadradas, además de que el uso de las funciones trigonométricas, podrían afectar la precisión del resultado y considerando que es una operación muy utilizada, preferimos evitar este tipo de errores.

El código para calcular la dirección de una vuelta queda de la siguiente forma.

```
1 public static int turnDirection(Point p, Point q, Point r) {
2         double determinant = (q.x - p.x) * (r.y - p.y) - (r.x - p.x) * (q.y - p.y);
3         if (determinant < 0) {
4             return RIGHT;
5         } else if (determinant > 0) {
6             return LEFT;
7         }
8         return COLINEARS;
```

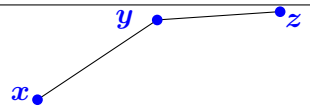
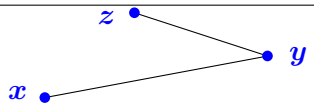
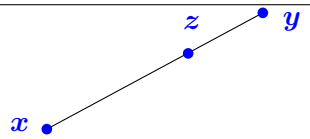
Valor	Dirección	Ejemplo
$det < 0$	Derecha	
$det > 0$	Izquierda	
$det = 0$	Colineales	

Tabla 1.2: Dirección de una vuelta según el determinante

Con ésto terminamos el conjunto de herramientas básicas que vamos a necesitar.

Capítulo 2

Galerías de arte y polígonos ortogonales

Los problemas de iluminación, son un conjunto de problemas que han generado gran interés en el campo de la geometría computacional. En particular los problemas de *galerías de arte*, que consisten en iluminar un objeto geométrico con algún tipo de guardia que por lo general ilumina con un ángulo de 360^o, podemos pensar dichos problemas tanto en el plano como en el espacio. Este trabajo está enfocado en diferentes problemas de *galerías de arte*, los dos principales son el *problema de la iluminación* y el otro tiene que ver con la iluminación de polígonos ortogonales. En este capítulo contamos el origen de este peculiar conjunto de problemas, introducimos algunas variantes y sobre todo observamos las diferentes técnicas que se han utilizado desde el origen del problema original.

El problema original de la *galería de arte* surge en el año de 1973 por Victor Klee, cuando se pregunta: *¿Cuál es el mínimo número de guardias necesarios para iluminar un polígono simple con n aristas?* [5]. Donde un guardia es un punto que se dice que guarda a otro punto dentro del polígono, si y sólo si podemos trazar un segmento de recta del guardia al punto, sin que el segmento toque alguna arista del polígono (también podemos pensar que es un punto que ilumina 360^o).

Esta pregunta fue resuelta por Vasek Chvátal, quien demostró un teorema que dice que $\lfloor \frac{n}{3} \rfloor$ guardias son ocasionalmente necesarios, pero siempre suficientes para iluminar un polígono simple. Este teorema también es conocido como el *Teorema de la Galería de Arte de Chvátal* o el *Teorema del vigilante*, el cuál se formaliza de la siguiente manera:

Teorema 2.1 (De la galería de arte de Chvátal)**.** Sea P un polígono simple de n vértices. Siempre son suficientes y ocasionalmente necesarios $\lfloor \frac{n}{3} \rfloor$ guardias para iluminarlo.

En breve haremos la demostración del *Teorema de Chvátal*, pero antes vamos a enunciar otro teorema, que utilizaremos para dicha demostración. Podemos encontrar lo anterior en [6].

Teorema 2.2. Todo polígono simple de n vértices admite una triangulación T, además T tiene exactamente $n-2$ triángulos.

Demostración del ***Teorema 2.2.*** Sea P un polígono simple de n vértices en posición general.

Inducción.

Caso Base: $n = 3$

P tiene tres vértices, por lo que P es un triángulo.

Hipótesis de Inducción: $n = k$

Suponemos que todo polígono simple P de k vértices admite una triangulación T, que tiene exactamente $k-2$ triángulos.

Paso Inductivo: $n = k+1$

Sea P un polígono simple de $k+1$ vértices. Demostraremos que P admite una triangulación T con $(k-2)+1$ triángulos.

Tomemos arbitrariamente a un vértice v_e de P y eliminaremos sus aristas incidentes, posteriormente agregaremos la arista $e = (v_i, v_{i+1})$, donde v_i y v_{i+1} son los vértices que eran adyacentes a v_e , llamaremos P' al polígono que se forma. Por la hipótesis de inducción, P' tiene una triangulación T' con $k-2$ triángulos. Agregaremos las aristas que eliminamos considerando los siguientes casos y a la triangulación T'.

- Caso 1: El vértice v_e está fuera de P', este caso es el más sencillo.
  Agregamos la arista $e = (v_i, vi+1)$ nuevamente. Se forma un triángulo t nuevo con v_i, v_e y v_{i+1}, aseguramos que $T = T' \cup \{t\}$ es una triangulación, pues simplemente se formó un triángulo nuevo en el límite de T', además T tiene exactamente $(k-2)+1$ triángulos, por el triángulo que agregamos.

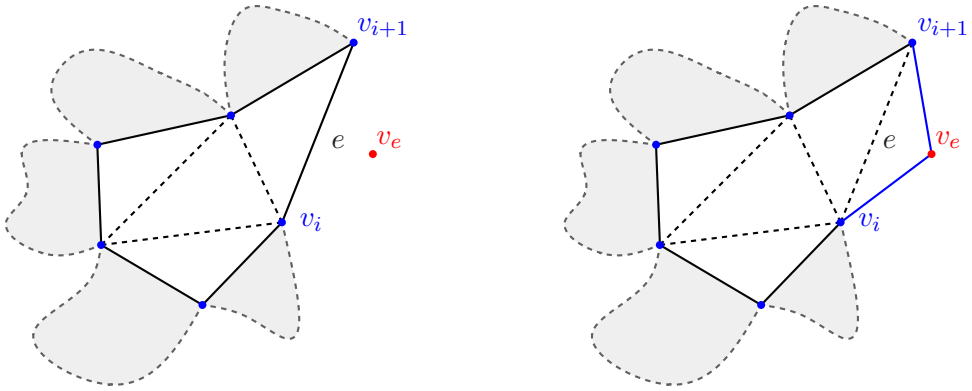


Figura 2.1: Caso 1.

- Caso 2: El vértice v_e está dentro de P' y está dentro de un triángulo que tiene como uno de sus lados una arista de P', la llamaremos e y a sus vértices v_i y v_{i+1}.

  En este caso uniremos v_e a los vértices adyacentes a e, con las aristas (v_i, v_e) y (v_e, v_{i+1}) y eliminamos a e. Esto hace que tengamos $(k-2)-1$ triángulos.

  Al agregar a v_e y dos nuevas aristas, queda un polígono no convexo de cuatro lados vacío, por lo que tenemos que agregar la diagonal de v_e al tercer vértice del triángulo original, para completar la triangulación. Al agregar la diagonal, dividimos el polígono de cuatro lados que se formó, en dos nuevos triángulos, con esto obtenemos una nueva triangulación T. Considerando a los $(k-2)-1$ triángulos que teníamos anteriormente, y a los dos nuevos, obtenemos que T tiene un total de $(k-2)+1$ triángulos.

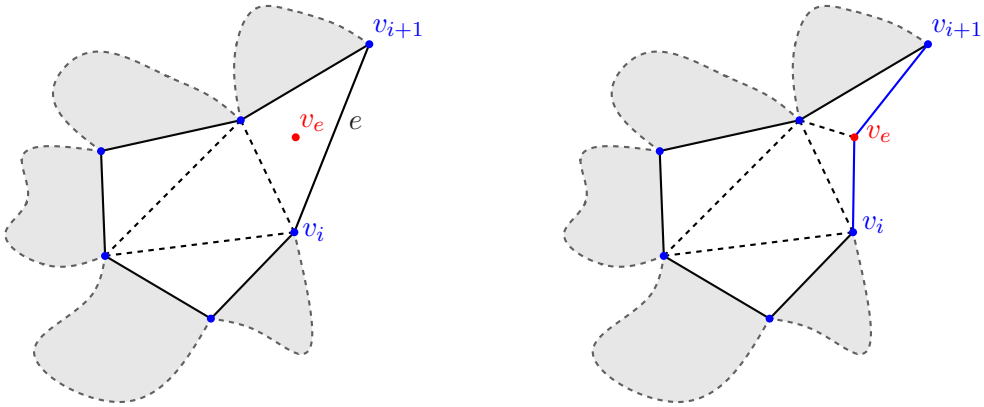


Figura 2.2: Caso 2.

- Caso 3: El vértice v_e está dentro de P' y está dentro de un triángulo cuyos lados son sólo diagonales de T', es decir que ningún lado es una arista de P'.

  Los tres lados del triángulo que contiene a v_e son diagonales, por lo que vamos a quitar una de ellas. Sin pérdida de generalidad le llamaremos d a dicha diagonal. Al quitarla, también eliminamos dos triángulos, por lo que nos quedamos con $(k-2)-2$ triángulos. Originalmente d es adyacente a dos vértices, que llamaremos v_{d1} y v_{d2}, éstos vértices forman un triángulo con un tercer punto al que llamaremos v_{d3}, observemos que d no es adyacente a v_{d3}.

También quitaremos la arista incidente en v_{d3} del exterior del polígono, sin pérdida de generalidad pensaremos en que era la arista (v_{d2}, v_{d3}); esta acción no tiene efectos en el conteo de triángulos, pues dicha arista no formaba parte de ningún triángulo, desde que quitamos a d.

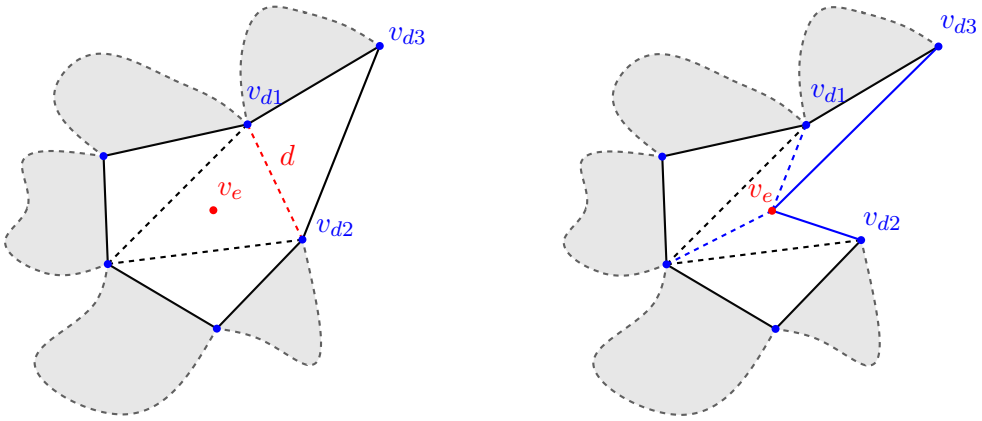


Figura 2.3: Caso 3.

Agregamos la arista (v_e, v_{d3}), es importante mencionar que esta es la arista que originalmente formaba parte de P, por lo que siempre es posible agregarla. Después agregamos las aristas (v_e, v_{d1}) y (v_e, v_{d2}). Al hacer esto, se forma un nuevo triángulo y un cuadrilátero no convexo, por lo que debemos agregar la diagonal de v_e al vértice del cuadrilátero al cuál no es adyacente, formando así dos nuevos triángulos. Con este procedimiento, obtenemos una nueva triangulación T de P, en la que agregamos tres nuevos triángulos, obteniendo así $(k-2)-2+3 = (k-2)+1$ triángulos.

En los tres casos agregamos un punto de manera que $P' \cup \{v_e\} = P$, además construimos una nueva triangulación T con exactamente $(k-2)+1$ triángulos. Por lo que podemos concluir que todo polígono simple de n vértices admite una triangulación de exactamente $n-2$ triángulos. ■

Lo interesante de esta prueba, es que es que puede darnos un pequeño panorama a un algoritmo incremental para construir polígonos simples. Y que una vez triangulado, podemos ir agregando vértices y triangular únicamente con cambios locales.

Ya sabemos que cualquier polígono simple es triangulable, más aún que su triangulación tendrá $n-2$ triángulos, o en otras palabras, tiene un número lineal de triángulos. Utilizaremos el teorema anterior para demostrar el teorema de la galería de arte de Chvátal. Intuitivamente colocaremos guardias en algunos triángulos para asegurar que iluminamos completamente cualquier polígono, si colocáramos un guardia en cada vértice del triángulo, es fácil ver que iluminamos el polígono completo, pero la idea es minimizar el número de guardias. A continuación escribimos la demostración formal.

Demostración del ***Teorema 2.1****:* Sea P un polígono simple de n vértices, al igual que en la demostración anterior, cualquier polígono mencionado durante esta demostración es simple a menos que se indique lo contrario.

Por el **Teorema** 2.2, sabemos que P tiene una triangulación T de $n-2$ triángulos. Sea T una triangulación de P con $n-2$ triángulos.

La idea es colocar un guardia en cada triángulo para iluminar el polígono completo. Recordemos que los guardias iluminan 360º, si colocamos un guardia en cada vértice, tendríamos n guardias y se iluminaría todo el polígono, sin embargo utilizaríamos guardias de más, pues sólo necesitamos un guardia por cada triángulo ya que iluminan 360º. Así que la idea es por cada triángulo colocar un guardia en exactamente uno de sus vértices y no más.

Sea C una tres coloración de los vértices de T. Aseguramos que los tres colores se repiten a lo más $\lceil \frac{n}{3} \rceil$ veces. Para asignar los guardias tomamos al color que tenga el menor número de repeticiones y colocaremos un guardia en cada vértice de ese color. Se utiliza un total de $\lfloor \frac{n}{3} \rfloor$ guardias.

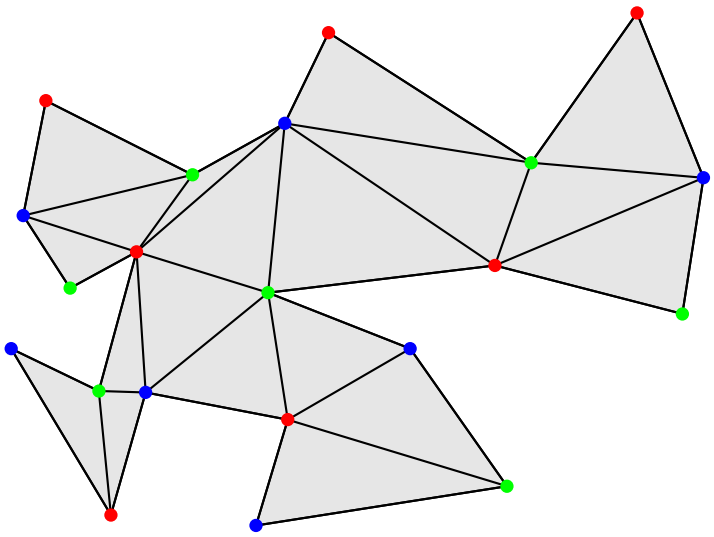


Figura 2.4: Ejemplo de la asignación de guardias en un polígono simple.

Con esta manera de asignar guardias a vértices se ilumina todo el polígono, podemos ver un claro ejemplo en la ***Figura*** 2.4, pues cada uno de los triángulos cuenta con exactamente un vértice de cada color, haciendo que cada triángulo tenga un guardia que ilumina por completo el interior del triángulo.

Por lo tanto, se necesitan exactamente $\lfloor \frac{n}{3} \rfloor$ guardias para iluminar el interior de cualquier polígono simple. ■

Los problemas de *galerías de arte* han sido importantes a partir del problema original, porque con hacer pequeños cambios a la pregunta original de Victor Klee, se han logrado problemas de la misma naturaleza pero sustancialmente diferentes. A continuación enunciamos algunos de estos problemas, que también se mencionan en [6, 7].

Una práctica muy común en los problemas de *galerías de arte* es cambiar el tipo de guardia con el que se cuenta, poner restricciones a éste, por ejemplo:

- Vértices: Esto es cuando estamos restringidos a poner nuestros guardias únicamente en los vértices del polígono. Un ejemplo de este tipo de guardias es el problema de la *galería de arte de Chvátal*, enunciado en el **Teorema 2.1**.

- Puntos: Cuando nuestros guardias son puntos, somos libres de colocar nuestros guardias en cualquier lugar del polígono, puede ser al interior o en el perímetro. En la ***Subsección*** 2.1.3, podemos encontrar un problema en el que iluminaremos polígonos ortogonales con reflectores ortogonales, los reflectores estarán colocados en el perímetro del polígono.

  Otro ejemplo de este tipo de guardias es [8], en donde se estudia el problema de iluminar el exterior de conjuntos de triángulos y rectángulos. En este artículo se demuestra que para iluminar el exterior de los n rectángulos para cierto tipo de configuración, se necesitan $\lfloor \frac{4n+4}{3} \rfloor$ en general. Y para iluminar el exterior de cualquier conjunto de n triángulos se necesitan $\lfloor \frac{4n+4}{3} \rfloor$. Ambas cotas son mejoradas en casos particulares.

- Aristas: Esta idea fue introducida en 1981 por Godfried Toussaint, pensando en que el guardia pudiera recorrer el largo de la arista, otra forma de plantearlo es que la arista ilumina por si misma.

  Podemos encontrar un ejemplo de este problema en [9]; este artículo nos muestra una manera eficiente de saber si desde una arista en específico podemos vigilar todo un polígono de n vértices, en un tiempo de $O(n)$.

  Además de estudiar esta variante en el plano, se pueden encontrar muchos problemas en el espacio iluminando poliedros, un ejemplo de esto es encontrar el mínimo número de aristas en un poliedro ortogonal con el que podemos iluminar el mismo, este problema fue resuelto en [10], donde demuestran que siempre son suficientes, $\frac{11e}{72}$ guardias-arista, donde e es el número de aristas de poliedro.

- Guardias en movimiento: Posteriormente O'Rourke propuso esta variante en que permitimos que los guardias se muevan a lo largo de segmentos del polígono, esto surge a partir de la idea de los guardias en las aristas antes mencionada.

Además de los guardias en aristas, O'Rourke propone en [4] que los guardias se muevan a través de las diagonales del polígono (segmentos de recta al interior del polígono que no se intersectan con ninguna arista del polígono salvo en los extremos, que resultan de trazar el segmento mismo desde un vértice del polígono a otro).

Una aplicación de una sugerencia como la de O'Rourke, puede ser encontrada en [11], en el que se propone una heurística para colocar cámaras o rutas de guardias en movimiento dentro de un videojuego.

- Reflectores: Como dijimos en el capítulo anterior, en la ***Definición 1.13***, son guardias que iluminan cierto ángulo. Al igual que con los guardias de 360°, haremos la distinción entre ponerlos sobre puntos y ponerlos sobre vértices.

  En 1990 durante un taller sobre iluminación; Jorge Urrutia propuso utilizar reflectores, que iluminan cierto ángulo (como con los que estaremos trabajando en el problema de la iluminación). La motivación para que surgiera esto es que muchos dispositivos de vigilancia en tiempo real, cuentan con un ángulo de visión limitado. Un ejemplo es [1] que es uno de los problemas principales de este trabajo y que más adelante vamos a estudiar con mucho detalle en los ***Capítulos 3,4,5 y 6***. Otro buen ejemplo es el *Problema de la iluminación con tres reflectores* que también estudiamos más adelante en la sección 3.1.

  Hay problemas que pertenecen a más de una categoría de las antes mencionadas y un buen ejemplo de esto lo podemos encontrar en [12], en el cuál se ilumina el interior de triángulos o cuadriláteros a partir de reflectores colocados en los vértices de las figuras geométricas. En este artículo se demuestra que 3 reflectores con un ángulo de iluminación de $\frac{\pi}{6}$ son suficientes para iluminar el interior de cualquier triángulo. Así mismo se demuestra que cualquier cuadrilátero puede ser iluminado con 3 reflectores con un ángulo que ilumina $\frac{\pi}{4}$ vértices. Este problema entra en la categoría de los "reflectores" y también en la de los "vértices".

Como podemos imaginar, a partir de estos pequeños cambios ya surgen un sinfín de posibles problemas, que forman parte de este particular conjunto de problemas de *galerías de arte*. Cabe destacar que las antes mencionadas, no son todas las variantes existentes y probablemente continúen surgiendo más. Es interesante mencionar algunos de los problemas que están relacionados para entender el tipo de estrategias utilizados para llegar a la solución del *problema de la iluminación*.

También es muy común iluminar polígonos con alguna característica especial y ver cuál es el mínimo número de guardias necesarios. En particular hay un problema que vamos a estudiar un poco más a detalle y es *el problema de la fortaleza*, que consiste en cubrir el exterior de un polígono utilizando guardias en los vértices. De manera muy breve vamos a describir un poco este problema como preludio para el *problema de la iluminación*.

2.0.1. El problema de la fortaleza

Considere un polígono simple de n vértices. ¿Cuántos guardias en los vértices se necesitan para iluminar el exterior de un polígono? A este problema se le conoce como *Problema de la fortaleza*. Un punto exterior y del polígono es iluminado por un guardia en el vértice x si y sólo si el segmento xy (el segmento de recta que une a x con y) no intersecta al interior del polígono. En 1983 O'Rourke y Wood prueban que se necesitan a lo más $\lceil \frac{n}{2} \rceil$ guardias.

A simple vista podemos pensar en una solución y un algoritmo un tanto intuitivos en caso de que el polígono sea convexo, sin embargo el problema se vuelve interesante cuando el polígono no es convexo, a continuación veremos el análisis de ambos casos dentro de la demostración del teorema enunciado a continuación.

Para la demostración de este teorema se utiliza un poco de Teoría de Gráficas y de triangulaciones de polígonos. La idea de la prueba para el caso general es generar una triangulación del polígono y elegir una 3-coloración, exactamente igual que en la prueba del ***Teorema 2.1 (de Chvátal)*** que vimos en la sección anterior de este mismo capítulo. A continuación la definición formal del ***Problema de la Fortaleza*** y su demostración.

Teorema 2.3. Sea P un polígono simple de n vértices. Siempre son suficientes y en algunas ocasiones necesarios $\lceil \frac{n}{2} \rceil$ guardias para iluminar el exterior de P.

Demostración. Sea P un polígono simple de n vértices.

- Caso 1: P es convexo.
  Aseguramos que, para poder iluminar completamente el exterior de P, es necesario colocar un guardia cada dos vértices, esto es debido a que P es convexo, por lo que la apertura del ángulo exterior entre cada pareja de vértices adyacentes es necesariamente mayor a 180^{o}, esto se sigue fácilmente de la ***Definición*** 1.16.

  Cuando tenemos un número impar de vértices habrá dos guardias en vértices consecutivos, pues al colocar guardias cada dos vértices, quedarán en vértices adyacentes. En la **Figura** 2.5 los guardias son representados por vértices de color rojo, si el vértice G fuera de color azul, no habría forma alguna de iluminar el segmento de recta que va de F a G, pues ese segmento no es iluminado por algún vértice siguiendo la definición anterior.

  Con esta técnica se asegura que el exterior de P queda iluminado completamente y además vemos que se utilizan exactamente $\lceil \frac{n}{2} \rceil$ guardias, precisamente por el caso en el que tenemos un número impar de vértices.

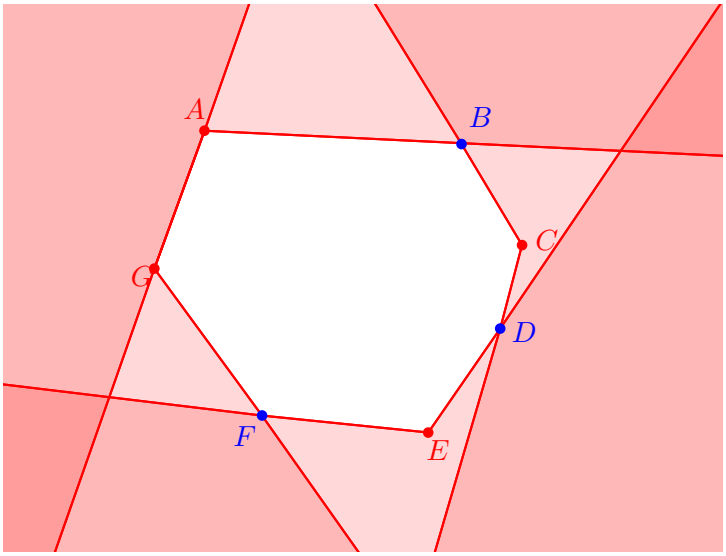
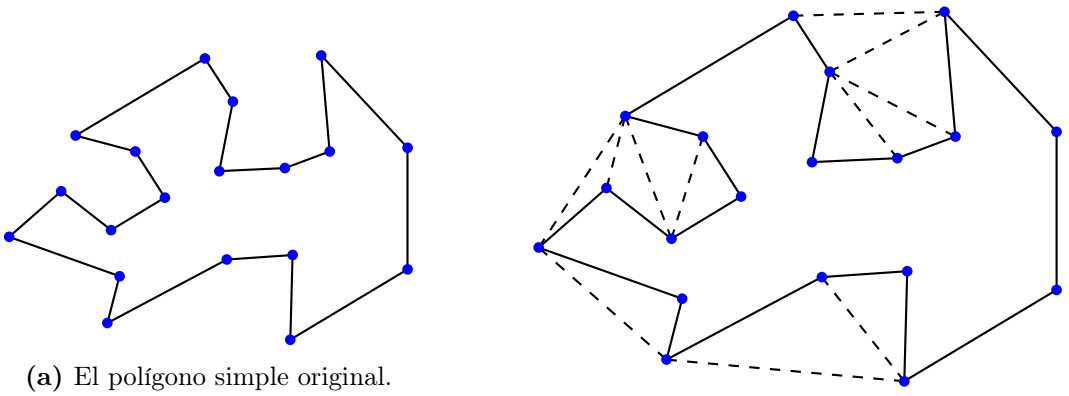


Figura 2.5: Ejemplo del caso 1: El exterior de un polígono convexo iluminado. Podemos observar que las partes de color rojo más tenues son las que únicamente están resguardadas por un vértice y las más oscuras por dos o más.

- Caso 2: P no es convexo.
  Para el caso general, calculamos el cierre convexo del polígono y triangulamos las partes que queden dentro del cierre convexo pero fuera del polígono, en la ***Figura*** 2.6 podemos ver un ejemplo. En este momento tenemos una triangulación que no es propiamente de P, sino del exterior de éste. La idea será encontrar una 3-coloración para dicha triangulación.



(a) El polígono simple original.

(b) El exterior del polígono triangulado.

Figura 2.6: Ejemplo del caso 2: Triangulamos el exterior del polígono.

Posteriormente agregamos un vértice *"al infinito"* y lo haremos adyacente a todos los vértices que pertenezcan al cierre convexo, a este vértice le llamaremos v_∞, en la ***Figura*** 2.7 las líneas rojas punteadas representan a las aristas que agregamos desde v_∞.

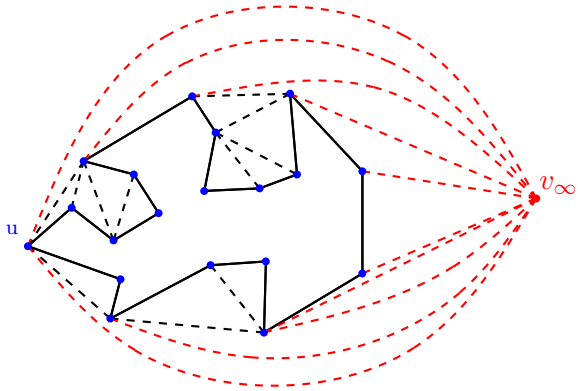


Figura 2.7: El exterior del polígono triangulado. El vértice u es al que vamos a separar.

Después elegimos un vértice u que pertenezca al cierre convexo y lo separamos en dos vértices u' y u'', de manera que juntos preserven todas las adyacencias que originalmente tenía el vértice u agregando una arista desde v_∞ a cada uno, en la ***Figura*** 2.8 podemos ver un ejemplo de esto. La idea es tomar la triangulación que acabamos de construir y asignarle una 3-coloración, no importa qué vértice u elijamos, pues con un poco de imaginación podríamos tomar la triangulación y doblarla de manera que el interior del polígono P original quede como el nuevo perímetro del polígono.

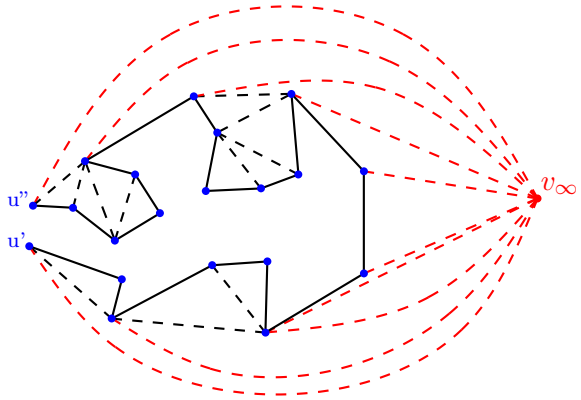


Figura 2.8: Podemos ver que una vez dividido el vértice u en u' y u'' y la tiangulación final.

La gráfica obtenida hasta el momento es plana, más aún es una triangulación, por lo que es 3-coloreable. Tomamos una 3-coloración para la triangulación obtenida y asignamos un guar-

dia a cada vértice en la segunda clase cromática con menos apariciones en la triangulación. Suponiendo que el vértice v_∞ forma parte de la clase cromática de menor tamaño, esta clase cromática dejaría de ser elegible, pues v_∞ no existe.

Utilizando el ***Teorema 2.1***, encontramos una 3-coloración para iluminar la triangulación con a lo más $\lfloor \frac{n+2}{3} \rfloor$ guardias.

Tomando en cuenta el desarrollo de ambos casos y comparando el número de guardias que se necesita en ambos casos, notamos que el máximo número de guardias es el que ocupa el caso en el que el polígono es convexo. Por lo tanto siempre son suficientes y en algunas ocasiones necesarios $\lceil \frac{n}{2} \rceil$ guardias para iluminar el exterior de un polígono simple de n vértices.

■

La prueba anterior la podemos encontrar en [6, p.146].

2.1. Polígonos ortogonales

Al inicio de este capítulo platicamos sobre las modificaciones a los guardias en los problemas de *galerías de arte*. Sin embargo no son el único tipo de modificaciones que se le han hecho al problema original, también se puede estudiar la iluminación de distintos tipos de objetos geométricos. En esta sección estudiaremos la iluminación de *Polígonos Ortogonales*. Además de estudiar la iluminación de polígonos simples, surge la idea de estudiar la misma idea con polígonos ortogonales, después de todo, la mayoría de las estructuras de construcciones y edificaciones suelen tener este tipo de distribución, además de que nos permite obtener otro tipo de resultados y nos provee de otro tipo de estrategias para estudiar un problema de iluminación. A partir de este tipo de aportaciones es que muchas veces surgen las ideas para resolver o proponer un problema diferente.

Las *galerías de arte ortogonales* nacen cuando Kahn, Klawe y Kleithman; proponen y demuestran en 1983 el teorema que asegura que para iluminar una galería de arte ortogonal con n vértices, siempre son suficientes y algunas veces necesarios $\lfloor \frac{n}{4} \rfloor$ guardias de 360^o [13]. Existen muchas pruebas conocidas para este resultado, la que propusieron los mismos autores se vale de descomponer el polígono en piezas mediante aristas auxiliares que se agregan. En un principio, podemos notar que para el problema original de *galerías de arte*, se utilizan $\lfloor \frac{n}{3} \rfloor$ guardias y con este teorema, se reduce de manera significativa el número de guardias utilizados, es por este tipo de resultados que vale mucho la pena estudiar *galerías de arte ortogonales*.

En este capítulo estudiamos una variante al problema original de la galería de arte. Vamos a restringir a que el polígono que estamos iluminando sea un *Polígono Ortogonal*, además de que iluminaremos un polígono Ortogonal, también vamos a iluminarlo con reflectores ortogonales exclusivamente colocados sobre vértices, éstas ya son tres restricciones al problema original, una para el polígono y dos para los guardias.

En esta sección daremos un número suficiente de reflectores ortogonales para iluminar un polígono ortogonal, además de que se dará un algoritmo para encontrar una configuración para éstos. Los resultados que se describen detalladamente en este capítulo son de [14]. En particular el algoritmo que se provee para iluminar un polígono tiene la ventaja de tener una complejidad de tiempo de $O(n)$ y no es difícil de seguir, pues algunos algoritmos que resuelven este mismo problema requieren de fragmentar el polígono en cierto tipo de partes con alguna forma especial y este algoritmo es mucho más simple.

Sea P un polígono ortogonal, simple y que además está conformado por n vértices, sea R un conjunto de k reflectores ortogonales que tienen una apertura en el intervalo $\alpha \in [\frac{\pi}{2}, \frac{3\pi}{2}]$, podemos ver un ejemplo de estos reflectores en la ***Figura*** 2.9. Considere una galería de arte determinada por el polígono P. El problema consiste en determinar si es posible colocar los k reflectores en vértices distintos de P de manera que iluminemos su interior. La otra pregunta interesante para este problema es *¿Cuál es el valor más pequeño que puede tomar k?* o en otras palabras, hay que buscar una cota mínima para k.

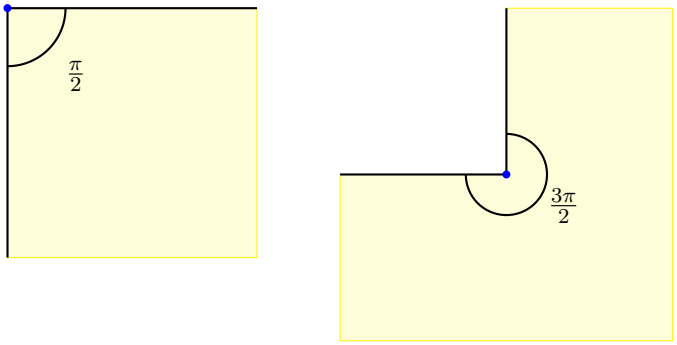


Figura 2.9: Tipos de reflector ortogonal.

En este capítulo damos un algoritmo de tiempo $O(n)$ para iluminar el interior de un polígono ortogonal, donde utilizamos reflectores que iluminan un ángulo de $\frac{\pi}{2}$. Posteriormente daremos una cota mínima para k. Cabe destacar que como estamos trabajando con un *Polígono Ortogonal*, todos los vértices tienen un ángulo interno de $\frac{\pi}{2}$ o de $\frac{3\pi}{2}$.

2.1.1. Iluminación de un polígono ortogonal

En esta sección daremos un algoritmo para iluminar el polígono ortogonal, utilizando reflectores cuya apertura es exactamente de $\frac{\pi}{2}$. Antes de dar el algoritmo necesitamos hacer distinción entre los diferentes tipos de vértices y aristas que podemos tener en el polígono. Tendremos cuatro etiquetas para las aristas y también cuatro etiquetas para los vértices, pero al mismo tiempo podemos encontrarnos dos versiones para cada una de esas cuatro etiquetas de vértices, haciendo un total de ocho diferentes vértices posibles.

Las etiquetas son muy intuitivas y fáciles de recordar, las etiquetas de las aristas serán ***Norte, Sur, Este*** y ***Oeste***. Las aristas *Norte* serán las aristas horizontales tales que el interior del polígono está debajo de ellas, análogamente las aristas *Sur* tienen el interior del polígono sobre ellas. Las aristas *Este* son las aristas verticales que tienen el interior del polígono a su izquierda y análogamente para las aristas *Oeste* que tienen el interior del polígono a su derecha. Podemos ver un ejemplo del etiquetado en la **Figura** 2.10(a).

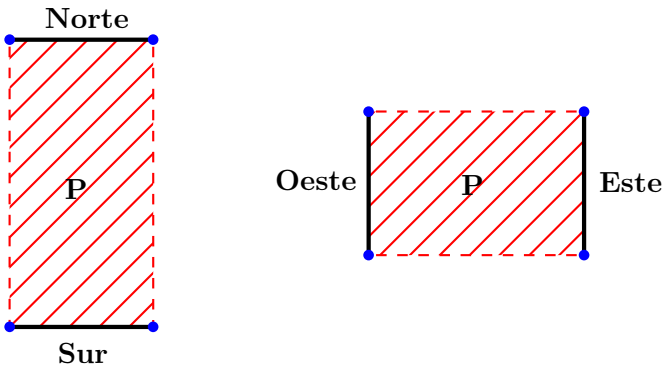
El tipo de vértices que podemos tener sale del tipo de aristas que hay, de manera que el vértice será llamado según las aristas que incidan en él; así un vértice **Noreste (NE)** es aquel en el que inciden una arista *Norte* y una *Este*, un vértice **Noroeste (NO)** es en el que inciden una arista *Norte* y una *Oeste*, un vértice **Sureste (SE)** es en el que inciden una arista *Sur* y una *Este*, por último un vértice **Suroeste (SO)** es en el que inciden una arista *Sur* y una *Oeste*.

Como dijimos antes, además de este etiquetado, hay dos tipos para cada una de estas etiquetas, esto depende del ángulo que es formado por las dos aristas. Si el ángulo es menor a π, entonces es un vértice convexo, pero si el ángulo es mayor a π, entonces es un vértice cóncavo. Podemos encontrar un ejemplo de las etiquetas en la **Figura** 2.10(b). El algoritmo para iluminar el polígono ortogonal es muy simple, y está asociado con una regla muy sencilla con la que podremos iluminar todo el polígono.

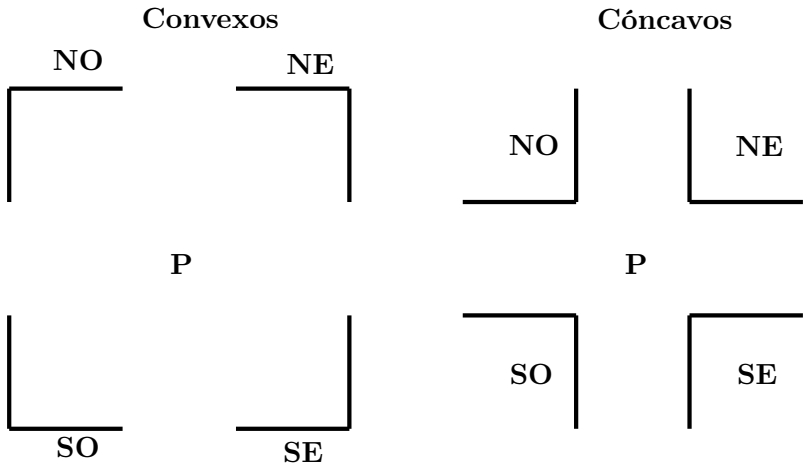
Definición 2.4. Regla de la asignación Noreste:

- Para cada arista **norte** del polígono, colocar un reflector alineado con la arista en su vértice **este**.
- Para cada arista **este** del polígono, colocar un reflector alineado con la arista en su vértice **norte**.

Teorema 2.5. La regla de asignación Noreste genera una configuración que ilumina todo el polígono.



(a) Tipos de aristas en un polígono ortogonal.



(b) Tipos de vértices en un polígono ortogonal.

Figura 2.10: Los diferentes tipos de vértices y aristas en un polígono ortogonal.

Esta regla inmediatamente nos hace formular el teorema anterior, que a continuación será demostrado. La idea es tomar un punto al interior del polígono y ver que es iluminado por algún vértice, que fue asignado por la regla de la asignación Noreste. Al igual que esta regla podríamos definir la misma para los casos **Sureste**, **Suroeste** y **Noroeste**.

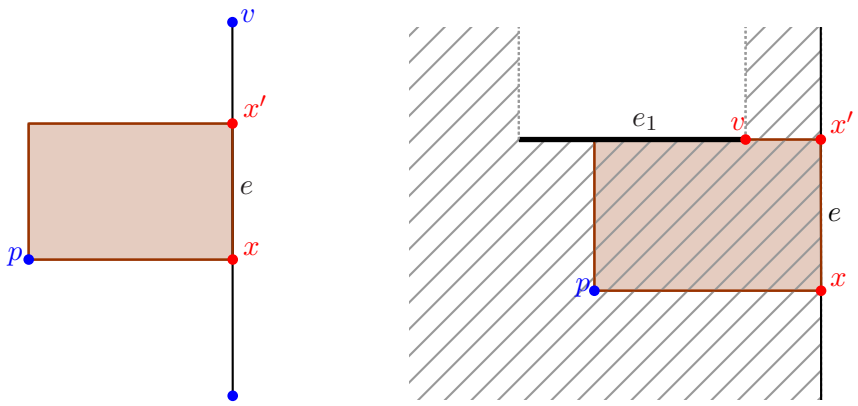
Demostración del ***Teorema 2.5****:* Sea p un puntoso al interior del polígono. Supongamos que le fueron asignados los reflectores suficientes al polígono y fueron colocados de acuerdo con la *regla de asignación Noreste*. Sea x el primer punto visible en el perímetro del polígono, si trazamos un rayo desde p hacia el este, x está en una arista del polígono, le llamaremos e a dicha arista.

Sea x' un punto sobre e que se encuentra al norte de x. Siempre es posible formar un rectángulo definido por x' y p, si x' está lo suficientemente cerca de x, entonces el rectángulo está contenido en el polígono, pero si alejamos demasiado a x' de x en algún momento el rectángulo no estará completamente contenido en el polígono, y esto puede suceder por dos causas.

- Caso 1: x' alcanzó y rebasó el vértice norte de la arista e, llamemos v a este vértice.
  Esto quiere decir que lo más alejados que x y x' pueden estar, es cuando x' está en v. El vértice v está al norte de una arista este, por lo que v es un *vértice norte* y hay un reflector alineado con e, pues la *regla de asignación Noreste* fue seguida. En ese caso, podemos observar que p es iluminado por el reflector en v, pues el rectángulo es vacío y todo lo que está en él es iluminado.

  El punto p es arbitrario, por lo que el área del rectángulo es iluminada completamente sin importar cuánto movamos a p en dirección al *Suroeste* siempre y cuando sigamos dentro del polígono y sin cruzar ninguna arista. Podemos ver una representación gráfica de este caso en la **Figura** 2.11(a).

- Caso 2: El rectángulo se sale del polígono,
  Esto pasa porque en algún punto nos encontramos una arista norte que coincidía con la arista norte del rectángulo y seguimos subiendo a x'; llamaremos e_1 a esta arista. En este caso, lo más alejados que pueden estar x y x' es cuando x' se encuentra sobre e a la altura de e_1 con la que chocó el rectángulo. También aseguramos que p es iluminado, pues e_1 tiene un *vértice este* que la define al que llamaremos v, y según la regla en ese vértice también colocamos un reflector alineado con e_1. Podemos ver una representación gráfica de este caso en la **Figura** 2.11(b)



(a) Caso 1 **(b)** Caso 2

Figura 2.11: Cualquier punto p al interior del polígono es iluminado con la regla de asignación Noreste.

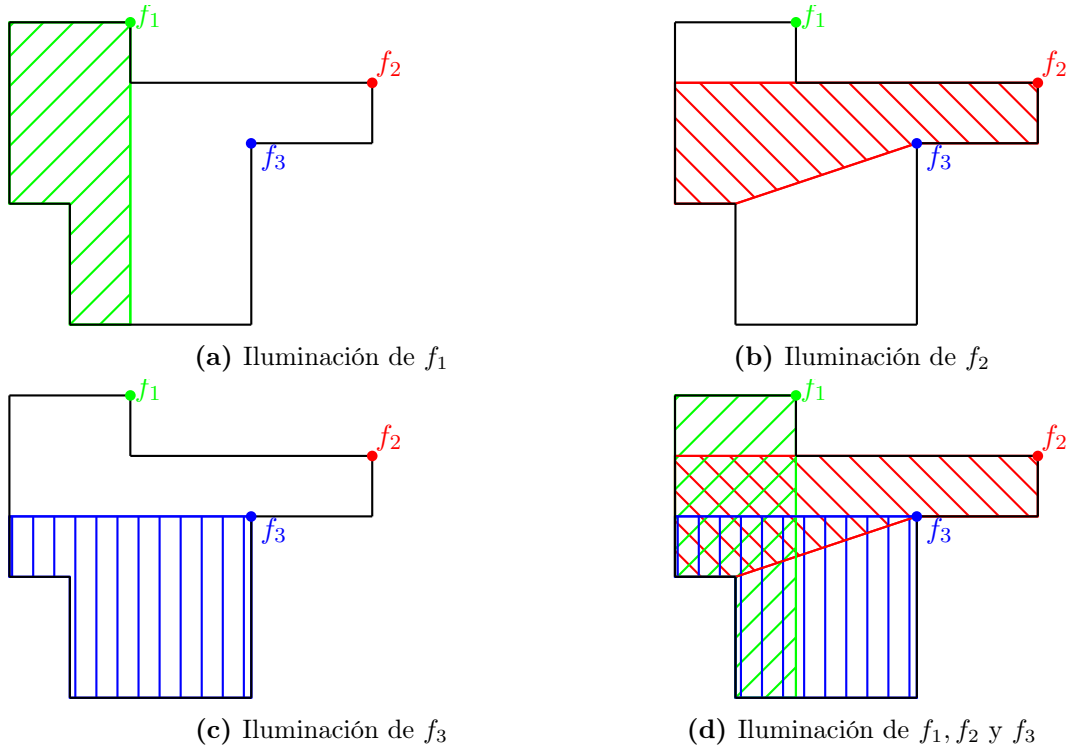
En ambos casos aseguramos que el punto p es iluminado. El punto p es un punto arbitrario dentro del polígono, por lo que la arista este sobre la que tomamos a x siempre existe. Por lo tanto cualquier punto dentro del polígono es iluminado, si utilizamos la regla de asignación noreste. Por lo tanto el polígono se ilumina completamente. ■

Con esto queda demostrado que siempre es posible iluminar cualquier polígono ortogonal con reflectores de apertura de $\frac{\pi}{2}$. Además de demostrar lo anterior, con la *Regla de la asignación noreste* obtenemos de manera directa un algoritmo de tiempo lineal para encontrar dicha configuración, la idea es recorrer todo el polígono por sus vértices y cuando nos encontremos con un vértice en el que debe de haber un reflector, basta con alinearlo con la arista que diga la regla y continuar con el siguiente vértice adyacente, hasta llegar al vértice donde comenzamos.

Podemos encontrar lo antes descrito en el algoritmo 2.1, que también aseguramos que es de tiempo lineal, porque simplemente recorremos los n vértices del polígono y de ser necesario se coloca un reflector cuando éste cumple la regla de asignación, operación que toma tiempo constante, por lo que la complejidad es de $O(n)$.

Algoritmo 2.1: Algoritmo de tiempo $O(n)$ para iluminar un polígono ortogonal

```
1  vertice = primerVertice();
2  verticesVisitados = false;
3  do
4  |  if vertive.cumpleReglaDeAsignacion() then
5  |  |  colocarReflector(vertice);
6  |  end
7  |  vertice = siguienteVerticeAdyacente();
8  |  if vertice.fueVisitado() then
9  |  |  verticesVisitados = true;
10 |  end
11 while !verticesVisitados;
```



(a) Iluminación de f_1

(b) Iluminación de f_2

(c) Iluminación de f_3

(d) Iluminación de f_1, f_2 y f_3

Figura 2.12: Polígono Ortogonal completamente iluminado.

En la **Figura** 2.12 podemos ver un ejemplo de un polígono ortogonal completamente iluminado por la *Regla de asignación noreste* 2.5. También podríamos haber utilizando otra regla y se usarían el mismo número de vértices para. Es fácil hacer el ejercicio mental de iluminar este mismo polígono utilizando alguna de las otras reglas de asignación (Noroeste, Suroeste o Sureste).

Con esto terminamos esta sección que se limita a mostrar un algoritmo para la iluminación de un polígono ortogonal. La siguiente sección es para contar el número de reflectores utilizados en dicho algoritmo.

2.1.2. Contando reflectores

En esta sección contaremos el número mínimo de reflectores que necesitamos para iluminar el polígono como lo describimos en la sección anterior. El algoritmo descrito antes contempla cuando los reflectores iluminan un ángulo de $\frac{\pi}{2}$, este algoritmo es producido por la regla del ***Teorema*** 2.5, a partir de la cual vamos a contar el número de reflectores utilizados.

Teorema 2.6. Si los reflectores tienen apertura de $\frac{\pi}{2}$, entonces son suficientes $\lfloor \frac{3(n-1)}{8} \rfloor$ reflectores para iluminar completamente el interior de un polígono ortogonal.

La prueba sale de la regla de la asignación *Noreste* y sus otras 3 reglas similares. La idea es contar en cuántos vértices vamos a poner reflectores según cada regla y quitar los repetidos, para no contarlos doble. Antes de hacer esta demostración necesitamos un resultado obtenido en [4], el teorema es el siguiente.

Teorema 2.7. En un **Polígono Ortogonal** de n vértices, si hay r vértices cóncavos (vértices cuyo ángulo interno es de $\frac{3\pi}{2}$), entonces se cumple que $n = 2r + 4$.

Demostración del Teorema 2.7. Sea c el número de vértices convexos (vértices cuyo ángulo interno es de $\frac{\pi}{2}$). Podemos ver que $n = c + r$, pues todos los vértices del polígono son cóncavos o convexos. Como la suma de los ángulos internos de un polígono convexo son $(n-2)\pi$ y como el ángulo de los vértices cóncavos es de $\frac{3\pi}{2}$, entonces $(n-2)\pi = c(\frac{\pi}{2}) + r(\frac{3\pi}{2})$. Despejando a c, obtenemos que $c = 2n - 4 - 3r$ y sustituyendo en la ecuación $n = c + r$ obtenemos que $n = 4 + 2r$. Por lo tanto, en un polígono ortogonal de n vértices, siempre se cumple la igualdad $n = 2r + 4$. ■

Ya tenemos una proporción de vértices cóncavos y por lo tanto convexos que hay en un polígono de n vértices. Utilizando este resultado, veremos cuántos reflectores son necesarios para iluminar un polígono ortogonal. Podemos encontrar un ejemplo en el que se cumple esta relación en la **Figura** 2.13. Utilizando este teorema podemos demostrar el **Teorema** 2.6.

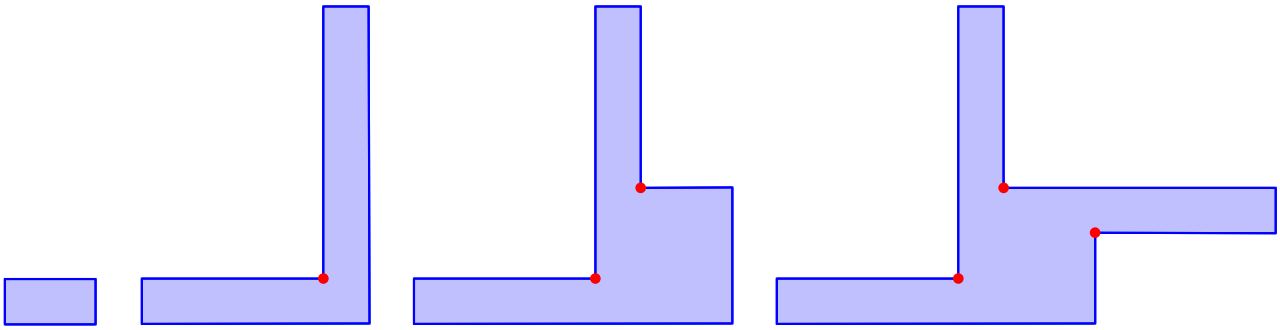


Figura 2.13: Observemos la relación descrita entre el número de vértices cóncavos con respecto al número total de vértices del polígono.

*Demostración del **Teorema** 2.6.* Supongamos que iluminamos el polígono utilizando las 4 reglas de asignación (**noreste, noroeste, sureste y suroeste**). Representaremos el número de vértices utilizados para cada regla como $||X||$ donde X es la regla correspondiente, por ejemplo $||NE||$ es el número de reflectores utilizados al aplicar la regla de asignación noreste.

Estamos utilizando las cuatro reglas al mismo tiempo, por lo que cada arista recibe a lo más dos reflectores, cada uno de reglas diferentes, por ejemplo si tenemos una arista *Norte*, las reglas que asignan un reflector a algún vértice son las reglas *Noreste y Noroeste*, lo mismo para los otros tres tipos de arista. Por lo anterior deducimos que por cada arista, a lo más le asignamos dos reflectores. Pero también es importante ver que los vértices convexos reciben exactamente un reflector, pues recordemos que sólo podemos poner un reflector por vértice. Por lo tanto el número de reflectores que se utilizan con la regla noreste está dado por la ecuación:

$$||NE|| = ||N||_e + ||E||_e - ||NE||_c$$

En esta ecuación $||N||_e$ representa el número de aristas norte que hay en el polígono, $||E||_e$ representa el número de aristas este y $||NE||_c$ el número de vértices convexos en el polígono. Por lo tanto el número total de reflectores está dado por la suma de esta cuenta para cada una de las reglas.

Lo que haremos es contar el número de aristas que estamos tomando en cuenta y también los vértices, como el polígono es ortogonal, simple y sin hoyos el número de aristas es igual al número de vértices por lo que haremos la cuenta y lo pondremos en términos de n, que es el número de vértices. El número de reflectores está representado por f, y c es el número de vértices convexos. Podemos ver el desarrollo de esta cuenta en la ecuación 2.1.

$$\begin{aligned} f &= ||NE|| + ||NO|| + ||SE|| + ||SO|| \\ &= 2||N||_e + 2||S||_e + 2||E||_e + 2||O||_e - ||NE||_c - ||NO||_c - ||SE||_c - ||SO||_c \\ &= 2(||N||_e + ||S||_e + ||E||_e + ||O||_e) - (||NE||_c + ||NO||_c + ||SE||_c + ||SO||_c) \\ &= 2n - (||NE||_c + ||NO||_c + ||SE||_c + ||SO||_c) \\ &= 2n - c \end{aligned} \tag{2.1}$$

Despejando a r del **Teorema** 2.7, y sustituyendo en la ecuación $n = c + r$, llegamos a que $c = \frac{(n+4)}{2}$. Ya sabemos cuántos vértices convexos hay en el polígono, por lo que ya podemos hacer la sustitución en la ecuación 2.1, y llegamos a que $f = \frac{3n-4}{2}$. Sin embargo estamos tomando en cuenta las cuatro reglas, que nos describen conjuntos de vértices completamente ajenos a excepción de los convexos que ya quitamos, por lo que tenemos que dividir esta cuenta entre el número de reglas.

Por lo tanto, para iluminar un polígono ortogonal simple, siempre son suficientes $\lfloor \frac{3n-4}{8} \rfloor$ reflectores de apertura $\frac{\pi}{2}$. ■

Con esta prueba podemos concluir esta sección, mencionando que siempre es posible iluminar *Polígonos Ortogonales* con a lo más $\lfloor \frac{3n-4}{8} \rfloor$ reflectores ortogonales que iluminan un ángulo de $\frac{\pi}{2}$. Además de haber probado que siempre es posible la iluminación y de dar un número suficiente de reflectores, vemos que es un algoritmo de tiempo $O(n)$ bastante fácil de seguir con una idea muy intuitiva, a diferencia de otros algoritmos que requieren de dividir el polígono en cuadriláteros y procesos un poco menos intuitivos, podemos encontrar estas ideas en [14] y en [6].

Algunas observaciones importantes que hay que hacer sobre el ***Teorema*** 2.6 son las siguientes:

1. Se utilizan más reflectores que en la solución al problema original de la iluminación de polígonos ortogonales. Como dijimos en la introducción de este capítulo, en el problema original Kahn, Klawe y Kleithman demostraron que son necesarios exactamente $\frac{n}{4}$ reflectores (que iluminan 360^o para iluminar cualquier polígono ortogonal, con el ***Algoritmo*** 2.1 descrito anteriormente utilizamos $\frac{3n-4}{8}$ reflectores, lo cual es más que $\frac{n}{4}$, así que aparentemente no es un resultado muy útil; sin embargo tiene algunas ventajas que veremos en los siguientes puntos.

2. Utilizamos una apertura de ángulos mucho menor a lo utilizado en la solución del problema original. En la solución que construimos en este capítulo sumando todos los ángulos de los reflectores utilizados, utilizamos una apertura total de $\frac{3\pi n}{16}$, lo cuál es mucho menor a los $n2\pi$ utilizados en la otra solución.

3. Tenemos un algoritmo intuitivo de tiempo $O(n)$ para la solución, evitando algoritmos de descomposición en cuadriláteros o "piezas en forma de L", mientras que en [6][pag. 73] se da un Algoritmo de tiempo $O(n\log(n))$ para dicha descomposición.

4. En realidad no siempre son necesarios tantos reflectores, considerando el caso de la ***Figura*** 2.14 que se ilumina con un solo reflector ortogonal, podemos ver que en el caso general normalmente se utilizarán muchos menos reflectores.

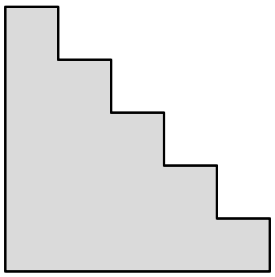


Figura 2.14: Escalera Ortogonal.

2.1.3. Iluminando un Polígono Ortogonal con $\frac{n}{4}$ Reflectores Ortogonales

Con el algoritmo de partir el polígono en "piezas en forma de L" que propone Joseph O'Rourke en [6][pag. 67], como resultado podemos iluminar el polígono con reflectores en el interior de éste, cosa que no es del todo agradable pues nos gustaría colocar los reflectores en las paredes de la galería de arte. Otra de las características que nos gustaría tener es que el algoritmo tenga una complejidad de tiempo menor, sobre todo porque el algoritmo que dimos en la sección anterior es de tiempo lineal y como lo mencionamos, el algoritmo de O'Rourke toma tiempo $O(n\log(n))$.

En esta pequeña sección daremos un algoritmo de tiempo $O(n)$ en el que utilizaremos $\frac{n}{4}$ reflectores ortogonales para iluminar un polígono ortogonal y que además coloca los reflectores en el perímetro del polígono, como deseamos.

La idea del algoritmo que vamos a construir es dividir al polígono en piezas, las cuales tendrán una estructura que nos va a permitir, colocar un conjunto pequeño de reflectores ortogonales y convexos en vértices cóncavos, es importante recordar que un reflector ortogonal convexo ilumina exactamente $\frac{\pi}{2}$. Para lograr esto necesitamos algunas definiciones primero.

Definición 2.8. En un polígono ortogonal P, un ***corte horizontal***, es la extensión de una arista horizontal que incide en al menos un vértice cóncavo. La extensión hacia el interior de P va desde un extremo de la arista hasta el perímetro de P. Los ***Cortes Horizontales*** nos van a servir para dividir a P en dos piezas, en la ***Figura*** 2.16 podemos encontrar ejemplos de cortes horizontales.

Definición 2.9. Un *corte horizontal* es un ***corte horizontal impar***, si una de sus mitades contiene un número impar de vértices cóncavos. En la ***Figura*** 2.15 podemos ver un ejemplo gráficamente.

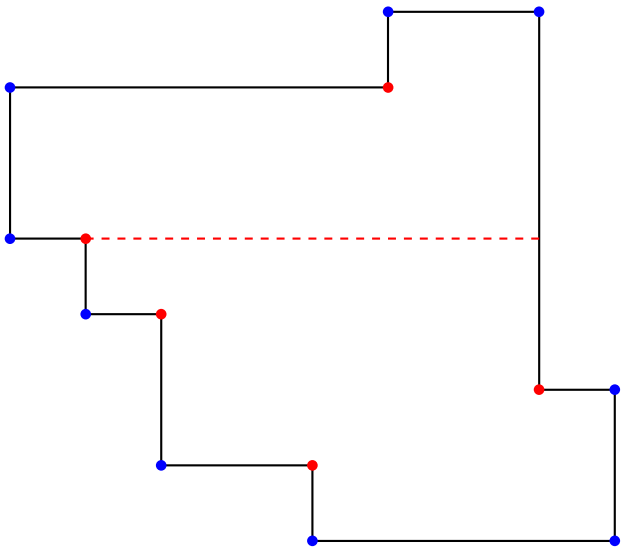


Figura 2.15: Este es un ejemplo de un corte impar, pues a ambos lados del corte representado por la línea punteada de color rojo, tenemos un número impar de vértices cóncavos también de color rojo.

Definición 2.10. Un vértice cóncavo es ***H-aislado***, si el otro vértice en el extremo de su arista horizontal es convexo. En la ***Figura*** 2.16(a) podemos encontrar un ejemplo.

Definición 2.11. Un vértice cóncavo es ***H-par***, si el otro vértice en el extremo de su arista horizontal también es cóncavo. En la ***Figura*** 2.16(b) podemos encontrar un ejemplo.

(a) Vértice H-Aislado.

(b) Vértice H-Par.

Figura 2.16: Ejemplo de Cortes Horizontales y tipos de vértices H.

Definición 2.12. Sea P un polígono ortogonal. Considere agregar todos los *cortes horizontales* para todos los vértices *H-Par* P y todas las piezas generadas a partir de éstos. La ***gráfica de cortes horizontales (Gráfica-H)*** de P, es la gráfica que resulta de asignarle a cada una de dichas piezas un vértice, donde un vértice relacionado con la pieza A tiene una arista dirigida que va al vértice relacionado a la pieza B si y sólo si se cumplen las siguientes condiciones:

1. A y B son adyacentes y están separadas por un *Corte Horizontal*.
2. Los dos vértices *H-par* correspondientes al *Corte Horizontal* están en el perímetro de A.

Podemos encontrar el ejemplo de una ***Gráfica-H*** en la ***Figura*** 2.17.

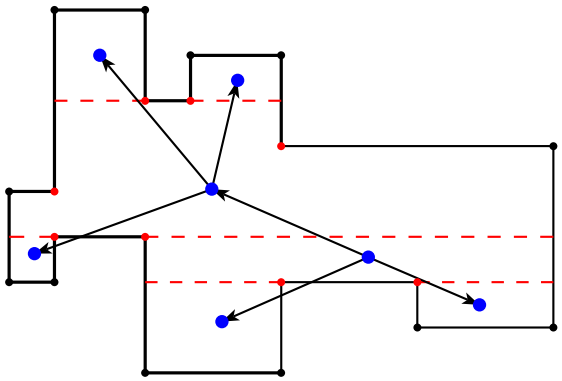


Figura 2.17: Ejemplo de una Gráfica-H. Podemos ver que los cortes horizontales y los vértices cóncavos son de color rojo. También podemos ver que los vértices de la gráfica son de color azul, a cada pieza formada por un corte le corresponde uno.

Aseguramos que la ***Gráfica-H*** de todo polígono P es un árbol, es decir que es una gráfica conexa y sin ciclos, es fácil ver que se cumplen ambas condiciones. Es conexa, pues partimos de un polígono en una sola pieza y cada partición es adyacente a otra por medio de cortes horizontales, impidiendo así desconexiones entre ellas. Así mismo no tiene ciclos, pues no es posible que dos particiones se unan formando algún ciclo, gracias a que todos los cortes son horizontales y si dos piezas estuvieran horizontalmente alineadas, implicaría un espacio fuera del polígono entre ellas o un hoyo.

A continuación demostraremos que para iluminar cualquier polígono ortogonal necesitamos a lo más $\lfloor \frac{n}{4} \rfloor$ reflectores ortogonales, pero antes necesitamos un resultado que nos va a permitir dividir el polígono en partes para iluminarlas y así lograr la iluminación total del polígono original.

Teorema 2.13. Sea P un polígono Ortogonal dividido en $P_1, P_2, ..., P_t$ particiones, cada partición se forma al dibujar todos los *cortes horizontales impares* y todos los *cortes horizontales* que sean los rayos de visibilidad de dos vértices cóncavos. Siempre es posible iluminar cada una de las piezas con $\lfloor \frac{r_i}{2} \rfloor + 1$ reflectores, donde r_i es el número de reflectores en P_i.

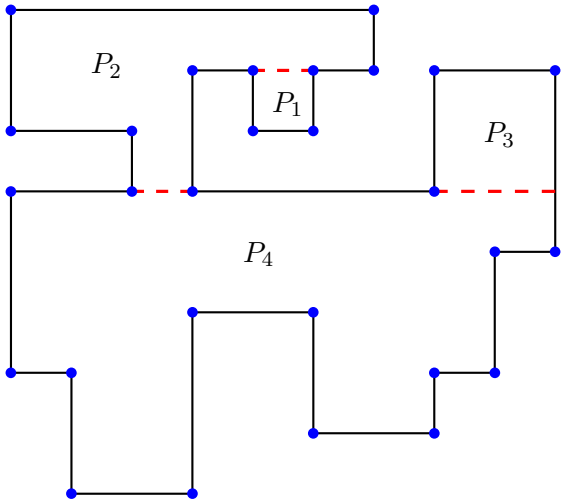


Figura 2.18: Ejemplo de la división de un polígono en particiones según el ***Teorema*** 2.13

Demostración del **Teorema** *2.13.* Decimos que un *Corte Horizontal* "resuelve" un vértice cóncavo, pues en el momento que se traza el corte, el vértice deja de ser cóncavo en cualquiera de las piezas.

Consideremos a la *Gráfica-H* de P. sabemos que es un árbol, así que aprovecharemos su estructura para aplicar una transformación que va eliminando ramas del árbol y colocará reflectores ortogonales que iluminan $\frac{\pi}{2}$.

Sea T una transformación que toma a la ***Gráfica-H de*** P, tal que le aplica las siguientes acciones a la gráfica:

1. Tomamos un nodo interno de la Gráfica-H, es decir un nodo que no es hoja. Esta pieza debe tener un reflector cóncavo asociado, le llamaremos d. La transformación coloca un reflector ortogonal en sobre d y alineado con su arisita vertical.

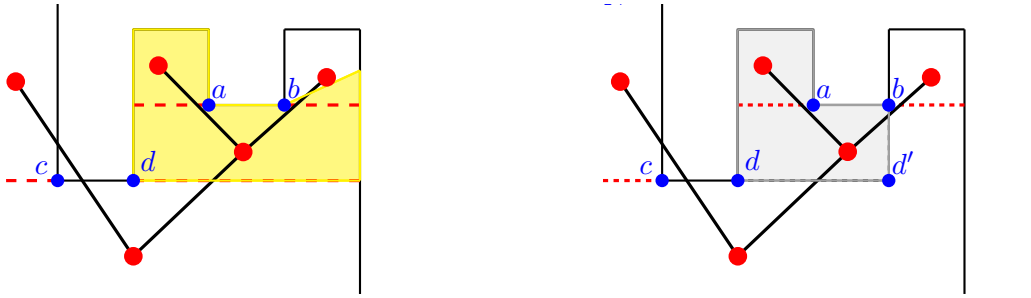
2. La pieza no es una hoja, por lo que tiene dos hijos y éstos tienen un vértice cóncavo asociado cada uno, más aún ambos vértices son $H - Par$ unidos por una arista de P, les llamaremos a y b.
   La transformación elimina al vértice d y coloca a un vértice d' con la coordenada x de b y la coordenada y de d.

3. Elimina a dos los vértices a y b que son cóncavos, recortando a P por el vértice d'.

4. Crea una partición P_i' nueva con dos vértices cóncavos menos, sin crear nuevos vértices cóncavos y tal que todas las aristas verticales de regiones en la *Gráfica-H* no son hojas, son aristas del polígono original.

5. Reemplaza las dos hojas y al nodo interno que tomamos del árbol por una hoja.

Podemos ver los primeros tres puntos de de la transformación T ejemplificados en la ***Figura*** 2.19. En la ***Figura*** 2.19(a) vemos la colocación de un reflector sobre un vértice cóncavo, la zona amarilla es la iluminada por dicho reflector. En la ***Figura*** 2.19(b) vemos cómo se agrega el vértice d', que nos va a servir para "eliminar" a la pieza dibujada en gris del polígono, pues ya fue iluminada.



(a) Coloca un reflector sobre d.

(b) Elimina a d y agrega a d'.

Figura 2.19: Representación gráfica de T.
En esta parte ilumina y prepara todo para reducir el polígono y el árbol

El resultado de aplicarle la transformación T a una parte de P_i en la ***Figura*** 2.19 se puede ver en la ***Figura*** 2.20. Por cada pareja de vértices *H-Par* T coloca un reflector, iluminando el área donde se encuentran los dos vértices cóncavos, así que estos dos vértices son virtualmente "eliminados" del polígono, pues al haber sido iluminados, ya no los tomaremos en cuenta.

Por cada reflector colocado también agregamos un vértice d', pero este vértice simplemente es un auxiliar que en realidad está al interior del polígono, por lo que no podemos poner un reflector sobre él. Con esta estrategia estamos utilizando esencialmente $\frac{r_i}{2}$ vértices en cada partición. Aún nos hace falta tomar en contemplar al resto de la figura después de la última vez que la estructura de la gráfica permite a T colocar reflectores.

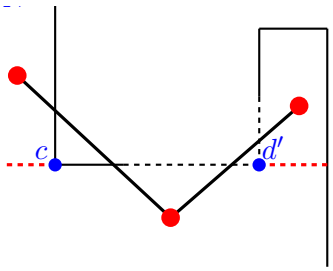


Figura 2.20: Elimina a la región iluminada y recorta al árbol.
Quitó dos vértices cóncavos y agregó uno auxiliar donde no se colocará ningún reflector.

Por hipótesis en todas las particiones, ningún corte horizontal es el rayo de visibilidad de dos vértices *H-Par*, porque ese tipo de cortes definen cada partición, así mismo en todas las particiones sabemos que a lo más hay un vértice cóncavo que no es *H-Par*, pues de haber dos o más habrían estado en particiones diferentes, pues se generaría un *Corte Horizontal Impar*, como consecuencia al podar el árbol no podríamos llegar a un caso como el que se muestra en la ***Figura*** 2.21.

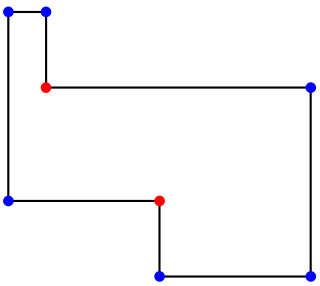


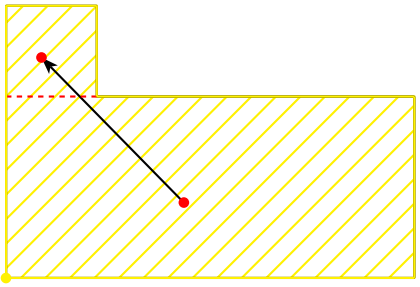
Figura 2.21: No podríamos tener una figura así, pues ambos vértices cóncavos generarían un corte que produce una nueva partición de P.

Al ir recortando a P_i, en cada paso eliminamos a una pareja adyacente de vértices *H-Par* y siempre vamos eliminando a ambos hijos de una rama, dejándolos como una nueva hoja, hijo de otra rama que igualmente puede proceder de una pareja de vértices *H-Par*. Esto se cumple siempre hasta que ya redujimos tanto a P_i que no existen más nodos internos además de uno sólo, que puede tener entre una y cuatro hojas adyacentes.

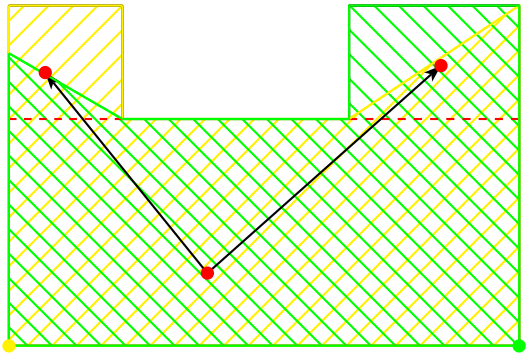
Cada adyacencia representa la presencia de un vértice cóncavo y vamos a iluminar al polígono siempre colocando a los vértices alineados con una arista vertical exterior, aseguramos que estas aristas siempre son parte del polígono original aunque no sean hojas, pues siempre que agregamos segmentos de recta vertical fue al interior del polígono.

Si a la parte que originalmente no era una hoja y tenía varias adyacencias sólo le queda una adyacencia, entonces podemos iluminar el resto del polígono con un reflector ortogonal colocándolo alineado con la base de ésta y con la arista vertical que comparte con la otra pieza, podemos ver el ejemplo de esto en la ***Figura*** 2.22(a).

Si tenemos dos adyacencias, podemos iluminar el polígono con dos reflectores ortogonales. Colocaremos ambos reflectores alineados con las aristas verticales y la base, cada uno es para iluminar a ambas hojas del árbol. El ejemplo está en la ***Figura*** 2.22(b). Como mencionamos anteriormente, no es posible que las hojas estén en bases distintas de P_i porque daría lugar a dos posibles *Cortes Horizontales impares*, dejando ambos vértices cóncavos en particiones diferentes. Los vértices donde se encuentran los reflectores son de color verde y amarillo respectivamente.



(a) Queda sólo un vértice cóncavo. El reflector está sobre el vértice de color amarillo, con un sólo reflector basta.

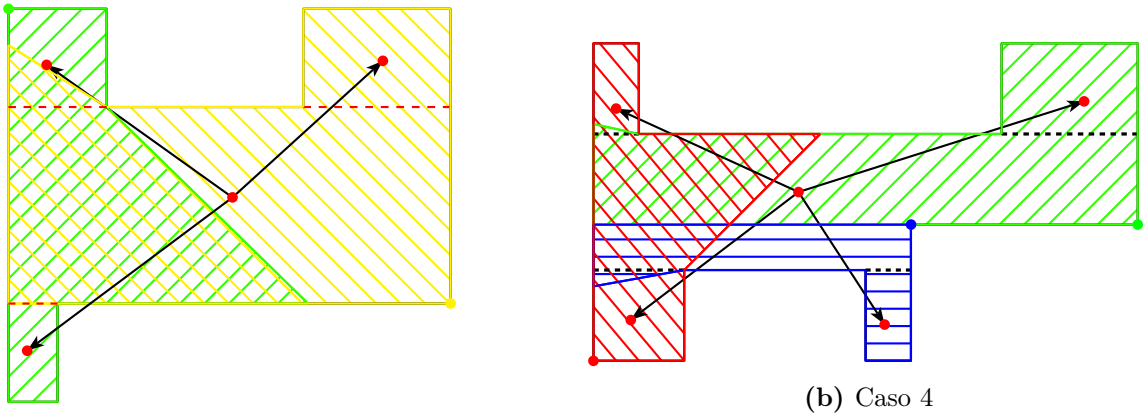


(b) Hay dos vértices cóncavos. Los dos reflectores están sobre los vértices de color amarillo y rojo.

Figura 2.22: Pieza ortogonal con uno, o dos vértices cóncavos.

Si tenemos tres adyacencias, también podemos iluminar el polígono con sólo dos reflectores ortogonales. Aseguramos que al haber tres hojas, hay dos reflectores alineados con la misma arista vertical, pues si eso no se cumpliera habría un vértice cóncavo extra, admitiendo dos posibles *Cortes Horizontales impares*, separando a P_i en dos piezas distintas desde el principio. Podemos ver una ilustración de esto en la ***Figura*** 2.23(a).

Y por último, si tenemos cuatro adyacencias podemos iluminar el polígono con tres reflectores ortogonales, teniendo dos casos. Analizaremos primero el caso en el que tenemos un vértice cóncavo extra en una de las aristas verticales, nos apoyaremos en la ***Figura*** 2.23(b) el resto de este párrafo. Igual que en el caso anterior tendremos dos piezas alineadas por una arista vertical, así que iluminaremos con un reflector ambas piezas, en la figura podemos ver a dicho vértice de color rojo. Posteriormente en el vértice cóncavo extra colocaremos un reflector y lo alinearemos con la arista vertical que incide en éste para que de esta forma ilumine a la pieza que le corresponde, en la figura dicho vértice es de color azul. Y por último colocaremos a un reflector en el vértice convexo adyacente de manera horizontal al vértice azul, de esta manera se ilumina la última pieza de P_i, éste vértice se encuentra de color verde. Entre los vértices azul y verde podemos observar que se ilumina perfectamente la parte de en medio.



(a) Caso 3

(b) Caso 4

Figura 2.23: Pieza ortogonal con tres, o cuatro vértices cóncavos.

Por otro lado, si el vértice azul no existiera, entonces tendríamos que cada una de las hojas estaría alineada con otra pieza frente a ella, el polígono tendría una forma similar a la letra H. Este caso es lo equivalente a tener dos casos como el vértice de color rojo, y en caso de que ambas áreas iluminadas estén muy separadas (es decir que el polígono sea muy ancho y angosto de en medio), necesitaremos un reflector alineado a con la pieza del centro para iluminar esta parte.

Estamos iluminando el polígono por completo, en el análisis inicial vimos que en esencia utilizamos $\lfloor \frac{P_i}{2} \rfloor$ reflectores en cada una de las t piezas y la última iteración también usamos una cantidad igual a la mitad de los vértices restantes más uno, en resumen, estamos utilizando exactamente $\sum_{i=1}^{t} \lfloor \frac{r_i}{2} \rfloor + t$, es decir que iluminamos cada partición con $\lfloor \frac{P_i}{2} \rfloor$ reflectores.

■

Teorema 2.14. Sea P un *Polígono Ortogonal* de n vértices. Para iluminar completamente a P, son siempre suficientes y algunas veces necesarios $\lfloor \frac{n}{4} \rfloor$ reflectores ortogonales en el perímetro de P. Además existe un algoritmo de tiempo $O(n)$ para calcular la configuración de los reflectores ortogonales.

Demostración. Por el ***Teorema*** 2.7 sabemos que en un polígono ortogonal de n vértices se cumple que $n = 2r + 4c$, donde r es el número de vértices cóncavos y c el número de vértices convexos. Si escribimos a r en términos de sólo n, podemos aproximar el valor de r de la siguiente forma:

$$\begin{aligned} r &= \frac{n-4c}{2} \leq \frac{n}{2} - 2c \leq \frac{n}{2} \\ r &\leq \frac{n}{2} \end{aligned} \tag{2.2}$$

Utilizaremos el ***Teorema*** 2.13 para dividir al polígono en particiones y así iluminar cada una, es importante recalcar que cada partición es iluminada con exactamente $\lfloor \frac{r_i}{2} \rfloor + 1$ reflectores, donde r_i es el número de vértices cóncavos en la partición. Sin embargo debemos recordar que al hacer los cortes para hacer las t particiones para dividir a P estamos resolviendo al menos un vértice cóncavo por cada una, por lo que con los cortes estamos resolviendo por los menos t vértices cóncavos y así llegamos a que $\sum_{i=1}^{t} \lfloor \frac{r_i}{2} \rfloor + t \leq \lfloor \frac{r}{2} \rfloor + t$, por lo tanto se cumple que:

$$\sum_{i=1}^{t} \lfloor \frac{r_i}{2} \rfloor \leq \lfloor \frac{r}{2} \rfloor \tag{2.3}$$

Si en el ***Resultado*** 2.3 escribimos a r en términos de n según el ***Resultado*** 2.2, llegamos a que utilizamos $\lfloor \frac{n}{4} \rfloor$ reflectores, en otras palabras acabamos de demostrar que a lo más en la mitad de los vértices cóncavos pondremos un reflector.

Por lo tanto son necesarios y suficientes $\lfloor \frac{n}{4} \rfloor$ reflectores para iluminar a P. ■

Con esto concluimos este capítulo que nos cuenta sobre el problema original de la *galería de arte* y algunas de las modificaciones que se le han hecho para llegar a nuevos problemas y resultados en esta área tan particular de la *Geometría Computacional*. Hay una cantidad impresionante de resultados en los problemas de *galerías de arte* tanto en el plano como en el espacio, como el ejemplo que mencionamos antes en [9]. El propósito de este capítulo es dar un preámbulo al problema de la iluminación, por lo que no se hace un énfasis profundo en otros problemas de esta misma área, pero si se ha generado interés en el lector, pueden consultar [5, 6] donde hay un compendio más importante a través de la línea del tiempo en cuanto a *galerías de arte* y particularmente en cuanto a *galerías de arte ortogonales* se refiere.

Capítulo 3

El problema de la iluminación

Como mencionamos en la introducción, el *problema de la iluminación* consiste en lograr la iluminación del plano a partir de un conjunto de reflectores que iluminan con un ángulo fijo. Sin embargo debemos de cumplir con un conjunto de restricciones, las cuales nos impiden idear una solución intuitiva al problema.

En este capítulo damos un contexto histórico sobre el *problema de la iluminación*, contaremos un poco sobre su origen y de los problemas relacionados con éste. También se define de manera formal el problema de la iluminación y se explica la necesidad de dividirlo en dos problemas para lograr su solución.

3.1. El problema de la iluminación con tres reflectores

Como mencionamos en el capítulo anterior, los reflectores son un concepto parecido a un guardia, la diferencia es que el guardia puede ver (o iluminar como lo hemos estado planteando) un ángulo 360^o y los reflectores tienen un ángulo de menor apertura. Al igual que los guardias, los reflectores pueden estar en puntos o en vértices de polígonos u otros objetos geométricos.

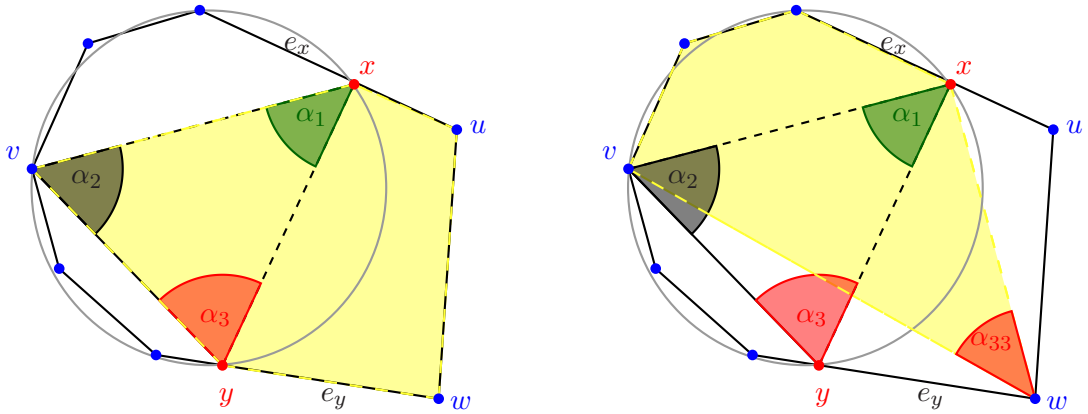
Sean α_1, α_2 y α_3 ángulos positivos que cumplen $\alpha_1 + \alpha_2 + \alpha_3 = \pi$; y sea P un polígono convexo cualquiera. *¿Habrá forma de colocar tres reflectores con aperturas a lo más de α_1, α_2 y α_3, en tres vértices distintos de P de manera que quede completamente iluminado?*. A esta pregunta se le conoce como el problema de la iluminación con tres reflectores, podemos encontrarlo en [5].

Sin pérdida de generalidad vamos a suponer que $\alpha_1 \leq \alpha_2 \leq \alpha_3$. Sabemos que $\alpha_2 \leq \frac{\pi}{2}$, pues la suma de cada ángulo es de π y los tres ángulos son positivos. Si los ángulos son del mismo tamaño, los 3 son menores a $\frac{\pi}{2}$; y si hay uno mayor a $\frac{\pi}{2}$, entonces debe haber uno menor. P es convexo, entonces existe un vértice v cuyo ángulo interior es de al menos $\frac{\pi}{2}$.

Lo que vamos a hacer es formar un triángulo, cuyos ángulos internos sean α_1, α_2 y α_3; colocaremos el vértice con apertura α_2 del triángulo sobre el vértice v; a los otros dos vértices los colocaremos en una arista de P, de manera que el triángulo quede al interior de P. A los dos vértices restantes les llamaremos x y y en sentido a favor de las manecillas del reloj.

Llamaremos f_1,f_2 y f_3 a los reflectores con ángulos menores o iguales a α_1, α_2 y α_3 respectivamente. Colocaremos al reflector f_2 sobre el vértice v de manera que ilumine completamente el triángulo que formamos, además ilumina al resto de P que queda después de la arista que va de x a y.

Lo siguiente es hacer un círculo C que pase por x, y y v; x e y se encuentran en dos aristas del polígono, a esas aristas les llamaremos e_x y e_y respectivamente. C va a cortar a e_x y a e_y de manera que al menos uno de los vértices de cada arista queda fuera del círculo, a esos vértices les llamaremos u y w. En la **Figura** 3.1(a) ilustramos a P, C y a f_2 iluminando x e y.



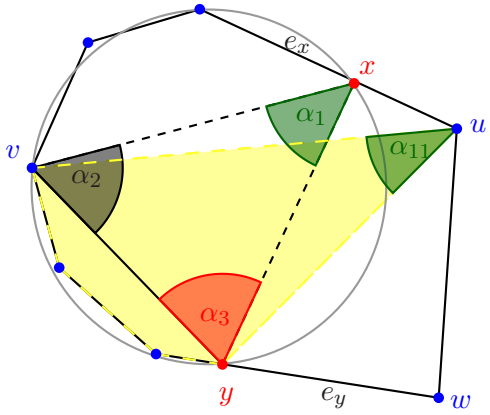
(a) f_2 iluminando el triángulo que formamos usando a x, y y v.

(b) f_3 iluminando el polígono que formamos usando a w, v y x

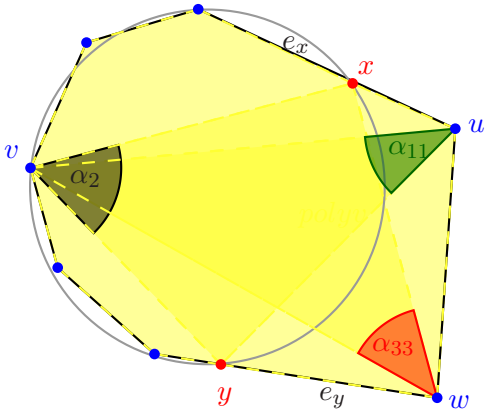
Figura 3.1: Podemos ver un ejemplo de lo descrito en el texto, las etiquetas de los vértices están configuradas como en la descripción en el texto, el círculo C está dibujado de color gris.

Lo último que debemos hacer es colocar a f_1 y a f_3 en los vértices u y w. En u colocaremos a f_1 de manera que ilumine el triángulo formado por los vértices u, y y v (**Figura** 3.1(b)); en w colocaremos a f_3 de manera que ilumine el triángulo formado por los vértices w, x y v (**Figura** 3.2(a)).

Aseguramos que se ilumina el polígono completo, pues si tomamos al triángulo formado por x, y y v y tomamos como base la arista vy y pudiéramos deformar el triángulo moviendo a x hasta donde se encuentra u el ángulo interno que se forma en u es menor o igual al ángulo interno que formaba el vértice x. Lo anterior se cumple gracias a que el vértice u está fuera del círculo C, podemos ver la iluminación completa del polígono en la **Figura** 3.2(b). Análogamente hacemos lo mismo con el vértice w.



(a) f_1 iluminando el polígono que formamos usando a u, y y v



(b) El polígono P totalmente iluminado por f_1, f_2 y f_3.

Figura 3.2: En el ejemplo los vértices azules son parte del polígono y los vértices rojos son del triángulo que formamos con ángulos internos α_1, α_2 y α_3. Podemos ver que los ángulos descritos antes cumplen que $\alpha_{11} \leq \alpha_1$ y $\alpha_{33} \leq \alpha_3$.

Con esto queda resuelto el *problema de la iluminación con tres reflectores*. Este problema es excelente como introducción a los problemas de *galerías de arte* que iluminan utilizando reflectores, pues ayuda a darse cuenta de que en este tipo de variante es importante fijarnos en los ángulos con los que trabajamos. En particular esto último va a ser fundamental en el desarrollo del algoritmo que resuelve *El problema de la iluminación*, el cuál se aborda en la siguiente sección y continúa hasta el final de este trabajo.

3.2. El problema de la iluminación

Al inicio del capítulo, comentamos que el *Problema de la Iluminación* es uno de los derivados del problema de la *galería de arte* y como tal tiene algunas de las restricciones que comentamos; en realidad estas restricciones son muy simples. En esta ocasión no contamos con un polígono a iluminar, el objetivo de este problema es iluminar el plano. Los ángulos deben estar sobre un punto ya fijo en el plano, es decir que no podemos mover esos puntos. Regresando un poco a la introducción, podemos pensar en reflectores o cámaras que queremos poner en postes que ya están en la superficie que queremos iluminar y lógicamente no podemos cambiar su ubicación. Otra restricción es que una vez que colocamos los ángulos, no podemos rotarlos, es decir que el lugar donde iluminan es fijo y no podemos cambiar su orientación en ningún momento. Por último los reflectores deben de ser pequeños, es decir que no tengan un ángulo mayor a 180^o.

La definición del problema es la siguiente. *Dados un conjunto de n puntos y un conjunto de n reflectores, cuyo ángulo es a lo más de 180º. Colocar cada reflector en un punto de manera que se ilumine todo el plano.* A partir de esta definición es que vamos buscar dos resultados, que al final se complementan para lograr el objetivo principal, que es la iluminación del plano. El primero es sobre la iluminación de una parte del espacio y el otro sobre una manera de particionar puntos. De modo que el teorema que vamos a demostrar se formaliza de la siguiente manera.

Teorema 3.1. Sea P una nube de puntos de tamaño n en *posición general*; sea R un conjunto de n reflectores cuyos ángulos cumplen que $\alpha_i \leq \pi$ y $\alpha_1, \alpha_2, ..., \alpha_n$, tales que $\sum_{i=1}^{n} \alpha_i \geq 2\pi$. Existe una configuración en la que colocamos cada reflector en un punto y podemos iluminar el plano completo, más aún podemos encontrar dicha configuración en un tiempo de $O(n \log(n))$.

A partir de este punto, nuestro objetivo principal no es otro más que demostrar el **Teorema** 3.1 y encontrar un algoritmo con complejidad de tiempo de a lo más $O(n \log(n))$ que encuentre una solución.

Es importante hacer notar la restricción que hace que los reflectores tengan un ángulo menor a 180º, pues en [1] nos provee de un contraejemplo, cuando los ángulos son 15º, 15º y 330º; y los puntos de la nube son los de un triángulo equilátero, podemos ver una ilustración en la **Figura** 3.3.

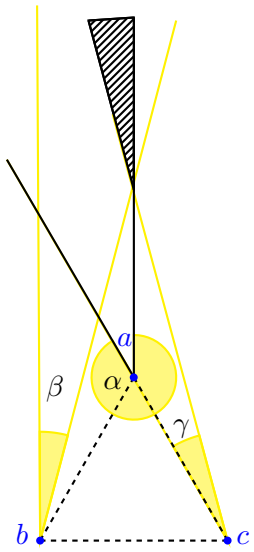


Figura 3.3: Podemos ver el contraejemplo antes descrito. Colocamos los reflectores con ángulos de $\alpha = 330^{\circ}$, $\beta, \gamma = 15^{\circ}$ en los puntos de un triángulo equilátero.

Podemos ver como en el punto a colocamos el reflector α, los otros dos reflectores están colocados sobre los puntos b y c. Para mostrar que es imposible iluminar el plano completo, alineamos al reflector en el punto c con la cuña de apertura de 30^o, que el reflector α no puede iluminar. Con esto cubriremos sólo una parte de esa área, pues podemos observar en la ***Figura*** 3.3 que hay una nueva cuña sombreada de negro. Esta cuña mide 15^o y no podrá ser totalmente iluminada; ya que al intentar cubrir a ésta con la luz del reflector β, que está sobre el punto b; en algún punto se intersectarán dos rayos, uno de la pequeña cuña a iluminar y otro de β; formando así otra nueva cuña que tendríamos que cubrir, sin embargo ya nos quedamos sin puntos y por consecuencia sin reflectores también.

En el contraejemplo anterior, hay que recordar que la suma de los reflectores anteriores es de 360^o, por lo que cumplimos una hipótesis del **Teorema** 3.1, sin embargo no cumplimos la hipótesis que pide que todos los reflectores iluminen a lo más 180^o. En [1] se menciona que los autores creen que al quitar esta hipótesis el problema podría convertirse en un *Problema NP*; no se menciona la razón por la cuál esta idea es sugerida, pero es probable que sea porque depende de qué punto elegimos para colocar el reflector mayor a 180^o, y la orientación del mismo. Para el resto de los reflectores, podemos elegir cualquiera de los puntos restantes, haciendo que la configuración final dependa de todas las posibles combinaciones de los n reflectores en los n puntos.

Para poder iluminar el plano completo, necesitaremos dos resultados que también se trabajan en [1]. El primer resultado se enfoca a la iluminación de un subespacio del plano, en concreto nos referimos a la iluminación de una cuña. El segundo resultado es para lograr hacer una partición de la nube de puntos, a manera de dejar el espacio dividido en tres cuñas. De esta manera podemos juntar ambos resultados para iluminar el plano completo, primero separando el plano en cuñas y posteriormente iluminándolas de manera separada, al final obtenemos el espacio iluminado completamente. En los siguientes dos capítulos se describirá de manera amplia cada uno de los resultados, además de hacer énfasis en la implementación de cada uno de los resultados y sus respectivas complejidades.

Capítulo 4

Iluminación de una cuña

Como dijimos en el capítulo anterior, el problema de la iluminación se divide en dos problemas más pequeños, en este capítulo vamos a hacer énfasis en uno de ellos. Si logramos segmentar el plano en subespacios y posteriormente iluminar cada uno de ellos, entonces lograremos la iluminación total. En este capítulo vamos a hablar de la iluminación de un subespacio, en este caso los subespacios de los que hablamos son cuñas.

También proporcionamos una implementación para iluminar el complemento de una cuña con k puntos al interior, con k reflectores disponibles para colocarlos en los puntos, todo ésto en un tiempo de $O(k\log(k))$.

4.1. Iluminación de una Cuña

Supongamos que tenemos una cuña y dentro de ella tenemos k puntos, nos gustaría tomar k reflectores y colocarlos en los puntos de manera que pudiéramos iluminar el complemento de la cuña. Esto último no es posible, a menos que la suma de los ángulos de los reflectores, sea mayor o igual al ángulo de la cuña que vamos a iluminar. Lo cuál nos lleva al siguiente resultado.

Teorema 4.1. Sean W una cuña cuyo ángulo de apertura es $\theta \leq \pi$. P una nube de puntos en *posición general* dentro de W. $\alpha_1, \alpha_2, ..., \alpha_k$ ángulos de un conjunto de reflectores, tales que $\theta \leq \sum_{i=1}^{k} \alpha_i$, con $k \geq 1$ y cada $\alpha_i \leq \pi$. Entonces, cada reflector puede ser colocado en un punto de P, de tal forma que juntos, iluminan la cuña complementaria de W. Más aún, existe un algoritmo de tiempo $O(k\log(k))$ que encuentra dicha configuración.

La prueba de este teorema es por inducción.

Demostración del Teorema 4.1. Sea W una cuña de ángulo $\theta \leq \pi$, formada por los rayos r y s que emergen de un mismo punto. Sea P una nube de puntos en posición general dentro de W.

Inducción sobre n.

Caso Base: $n = 1$
Tenemos un punto p dentro de W. Queremos iluminar completamente la cuña complementaria de W (denotada por W^c), con un reflector de ángulo α que esté sobre p, pues son el único punto y el único reflector.

Lo que tenemos que hacer es rotar a α sobre p, de manera que el $rayo_1$ que define a α sea paralelo al primer rayo que define a W^c. Notemos que el $rayo1$ y r tienen la misma pendiente, pero con signos contrarios, podríamos decir que son *anti-paralelos*. Por hipótesis $\theta \leq \alpha$, por lo que W^c está completamente iluminada, pues el ángulo $\alpha \geq \theta$. Se muestra un ejemplo gráfico en la **Figura** 4.1.

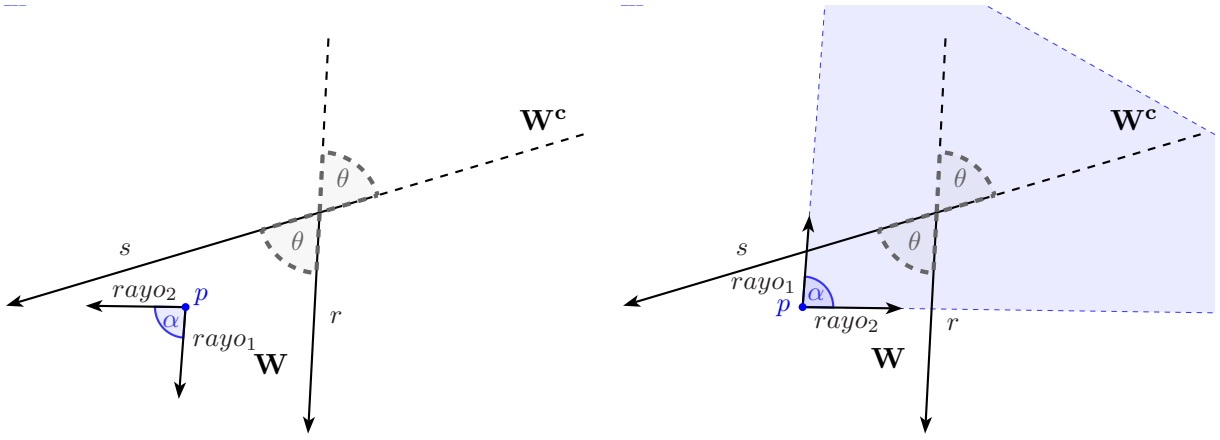


Figura 4.1: Caso base para la iluminación de una cuña.

Hipótesis de Inducción: $n = k$
Sea W'' la cuña de ángulo θ formada por los rayos s y s', tal que contiene k puntos. Supondremos que existe una configuración, para los k reflectores, colocados sobre los k puntos, tal que los reflectores iluminan completamente a W''^c.

Paso Inductivo: $n = k + 1$
Sea W una cuña formada por los rayos r y s, tal que contiene $k+1$ puntos. Tomemos a un conjunto de $k+1$ reflectores, tales que $\alpha_1 + \alpha_2 + ... + \alpha_k + \alpha = \theta + \alpha$, donde α es el ángulo de alguno de los reflectores.

Sea l una recta paralela a alguno de los rayos que forman a W, por ejemplo r. Rotamos a l, un ángulo de α con dirección a s. Sea L la familia de rectas paralelas a l después de dicha rotación. Sea $l_1 \in L$, tal que divide a P, dejando k puntos de un lado y a un punto del otro. Llamaremos s' al rayo que se produce de la intersección de l_1 con s. Notemos que se forman dos cuñas que dividen a W; a la primera le llamaremos W', y está formada por s' y r; a la segunda le llamaremos W'', y está formada por s' y s. W' tiene un ángulo de α, pues su ángulo está dado por la rotación de l, y W'' tiene un ángulo de θ que es la resta del ángulo original de W, menos α.

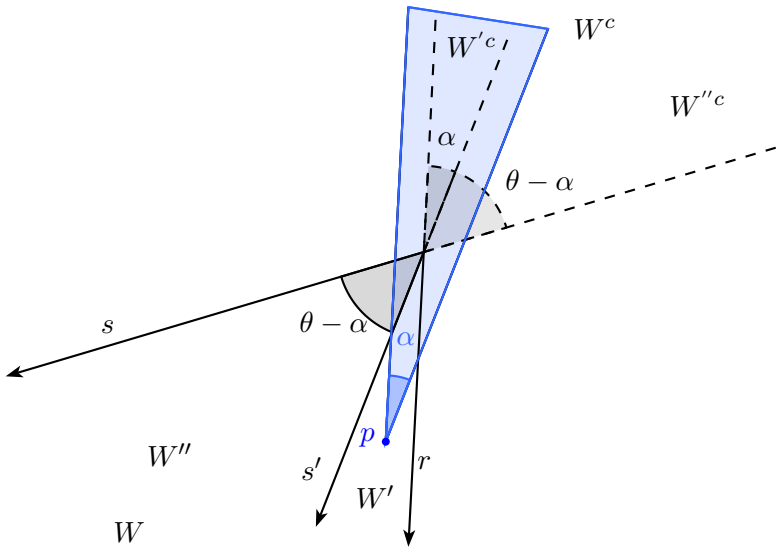


Figura 4.2: Paso inductivo de la iluminación de una cuña

W'' cumple con la hipótesis de Inducción, por lo que existe una configuración en la que los primeros k reflectores iluminan completamente a W''^c. Por otro lado, tenemos un punto dentro de W', y nos sobra un reflector con ángulo mayor o igual α. Si colocamos a este reflector sobre p, exactamente de la misma manera que en el caso base de la inducción, iluminamos a la cuña complementaria de W'.

Con esto queda iluminada completamente W^c. Podemos ver el ejemplo en la **Figura** 4.2. ■

4.2. Implementación

Al inicio del capítulo mencionamos que también proporcionaríamos un algoritmo de tiempo $O(k\log(k))$ para iluminar una cuña, donde k es el número de reflectores dentro de la cuña, con los que vamos a iluminar el complemento de la misma. A continuación ponemos el pseudocódigo del algoritmo que implementamos y que más adelante vamos a describir.

Algoritmo 4.1: Algoritmo de tiempo $O(k\log(k))$ para iluminar una cuña

```
1  if angulos.longitud == 1 then
2  |  iluminarCuña(W, angulos, puntos[0]);
3  else
4  |  angulos1 = 0;
5  |  angulos2 = 0;
6  |  for i := 1 to n do
7  |  |  if i < ⌊n/2⌋ then
8  |  |  |  mitad1.agrega(angulos[i]);
9  |  |  |  puntos1.agrega(puntos[i]);
10 |  |  else
11 |  |  |  mitad2.agrega(angulos[i]);
12 |  |  |  puntos2.agrega(puntos[i]);
13 |  |  end
14 |  end
15 |  l = rectaQueDivideCuña(mitad1.getSuma(), puntos1, W);
16 |  W1, W2 = divideCuña(l, W);
17 |  iluminarCuña(W1, mitad1, puntos1);
18 |  iluminarCuña(W2, mitad2, puntos2);
19 end
```

Podemos ver que el algoritmo 4.1 es recursivo, también es cierto que es un algoritmo divide y vencerás. Vamos a verificar que el algoritmo cumple con el tiempo requerido.

Primero verifica que sea el caso base, si es el caso base, lo que hace es exactamente el procedimiento del caso base de la demostración, por lo que ilumina a la cuña con un sólo reflector. Esto toma tiempo constante, pues simplemente es hacer algunas operaciones aritméticas y algunos cálculos trigonométricos. Si no es el caso base, entonces procede a hacer la división de nuestros conjuntos por la mitad, para que podamos hacer la llamada recursiva. Esta división es con cuidado, pues debemos de tomar en cuenta que queden exactamente la mitad de los puntos de un lado de la recta que se forma y la otra mitad del lado contrario.

La división de la que hablamos es a través de una recta, tenemos que verificar que esa recta forme el ángulo de la suma de la mitad de los ángulos para que se cumplan las hipótesis del teorema y podamos iluminar el complemento de manera satisfactoria. Esta división es posible gracias a un teorema muy recurrente en Geometría Computacional y es el ***teorema de la equipartición*** (en la literatura en inglés podemos encontrarlo como ***Ham Sandwich Theorem*** [15]) y se enuncia a continuación.

Teorema 4.2 (De la equipartición)**.** Dada una nube de puntos en posición general en el plano, y una pendiente p. Es posible encontrar una recta l de pendiente p tal que la mitad de la nube de puntos quede a la izquierda de l y la otra mitad a su derecha.

Con el teorema anterior, podemos fácilmente deducir que es posible construir una recta, con una pendiente que forme exactamente el ángulo necesitado para dividir la cuña y al mismo tiempo dividir la nube de puntos a la mitad. La complejidad de este algoritmo es de $O(n\log(n))$, pues el procedimiento que se sigue en el programa es de proyectar los puntos sobre la recta, y ordenarlos con respecto a la pendiente y como sabemos, el ordenamiento tiene una complejidad de tiempo $(O(n\log(n))$.

En cuanto a la complejidad del algoritmo, si dividimos por la mitad los conjuntos y la cuña cada vez que hacemos la llamada recursiva, entonces hacemos un número logarítmico de divisiones. En el algoritmo original cada una de esas divisiones toman tiempo lineal en ser ejecutadas, pues utilizamos un algoritmo conocido para encontrar la mediana de la familia de rectas, sin embargo en la implementación esto se traduce a un ordenamiento con respecto a una recta, por lo que toma un tiempo de $(O(k\log(k))$, donde k es el número total de puntos en la nube. La complejidad del algoritmo completo es de $O(k\log(k))$, sin embargo con el ordenamiento mencionado antes en la implementación, la complejidad de tiempo aumenta un poco a $O(k\log^2(k))$.

Con la demostración anterior, pseudocódigo y la explicación anteriores, podemos concluir que siempre es posible iluminar una cuña, en tiempo de $O(k\log(k))$. El pseudocódigo del **Algoritmo** 4.1 es prácticamente igual al código definitivo de la aplicación, sin embargo hay cosas en las que haremos un énfasis especial.

Cuando construimos la recta que divide a la mitad de los puntos y nos forma el ángulo exacto de la suma de la mitad de los ángulos, en la parte práctica hay algunos detalles de implementación que vale la pena mencionar.

Cuando dividimos a nuestro conjunto de puntos con una recta, utilizando el *Teorema de la equipartición* (**Teorema** 4.2), el orden depende directamente de la pendiente. El orden obtenido para el conjunto de puntos, se invierte si la recta cambia de signo, es decir que si nuestra pendiente p tiene signo positivo obtendremos un orden, pero si p tiene signo negativo, tendremos el orden inverso al anterior. Este es un detalle mayor a la hora de implementar la división de una cuña en dos, pues el número de puntos puede deseado, puede quedar del lado correcto de la recta o del lado contrario. Si el orden es el inverso al necesario, habrá que invertir las cuñas. Podríamos pensar que están de cabeza y al voltearlas hacemos la asignación correcta de los puntos con la cuña correspondiente.

Al haber demostrado el teorema principal de este capítulo, dar un algoritmo y explicar el código utilizado para la iluminación de una cuña, damos por terminado el primer objetivo para la solución final. Continuamos en el capítulo siguiente, donde se dará un algoritmo para lograr la segmentación de la nube de puntos en el plano.

Capítulo 5

Tripartición del plano

En este capítulo vamos a demostrar un teorema que nos permite tripartir el plano en cuñas, de esta forma podremos utilizarlo con el resultado del capítulo anterior y así iluminar todo el plano.

El teorema que vamos a demostrar pertenece a una familia de resultados muy importantes para la Geometría Computacional, y esa familia de resultados es la de particiones de objetos geométricos, en particular nosotros nos enfocaremos a nubes de puntos. En [1], se menciona que ya hay un resultado publicado en [16]; para particionar una nube de puntos en tres conjuntos de cardinalidad igual, en un tiempo esperado de $O(n)$; sin embargo el resultado que estudiamos en este trabajo deja libre la cardinalidad de dos conjuntos, haciendo de éste un resultado muy fuerte.

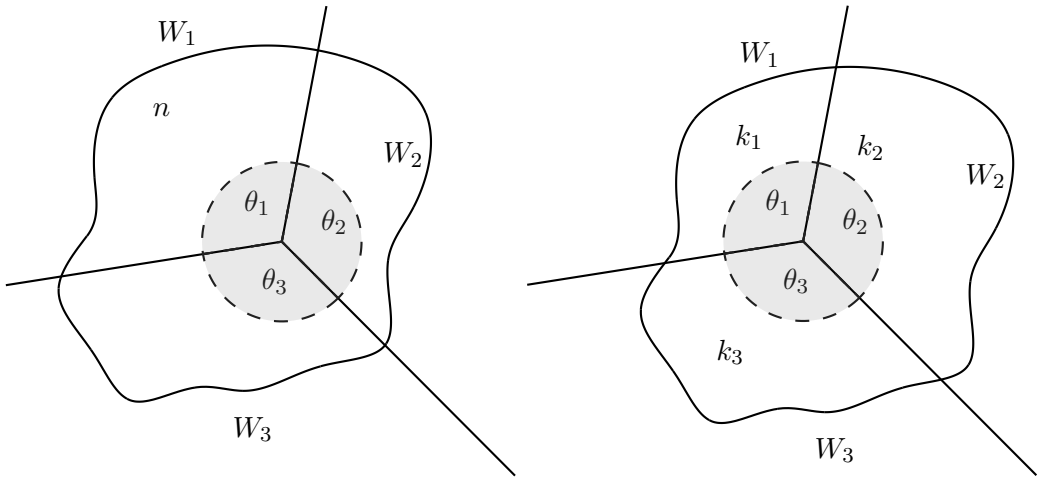
Ya sabemos iluminar el complemento de una cuña que contiene una cantidad de puntos, siempre y cuando la suma de los ángulos de los reflectores sea de al menos la apertura del ángulo de la cuña. Ahora, si logramos dividir la nube de puntos de tal manera que tengamos tres cuñas que emergen de un mismo punto en común, entonces podremos iluminar el espacio completo, pues por hipótesis, la suma del ángulo de los reflectores es de al menos 2π.

Otra parte importante del resultado es que las cuñas que formamos van a tener ángulos θ_i que nosotros vamos a dar, y estos ángulos podemos elegirlos de manera cuidadosa, de tal forma que el ángulo de cada cuña sea igual a la suma de un subconjunto de los ángulos de los reflectores. Así construimos una cuña para elegir un conjunto de reflectores, e iluminar el complemento de la cuña que contiene a los puntos sobre los que pondremos los reflectores.

5.1. Tripartición del plano

A continuación se dará el enunciado formal para la tripartición de una nube de puntos. Una observación interesante es que en el enunciado no se restringe que los ángulos de las cuñas sea menor a π, pues también es posible unir dos cuñas y así bisectar la nube de puntos y el proceso inverso también.

Teorema 5.1. Sean θ_1, θ_2 y θ_3 ángulos mayores a cero tales que su suma es igual a 2π. Sea P un conjunto de n puntos en el plano, y sean $k_1 + k_2 + k_3 = n$ una partición de n. Entonces existe un punto x en el plano y tres cuñas ajenas W_1, W_2 y W_3 que emergen de x, tal que para cada $i = 1, 2, 3$, la cuña W_i tiene ángulo θ_i y contiene k_i puntos de P. Más aún existe un algoritmo para encontrar x y a las W_i en tiempo $O(n \log(n))$. De manera gráfica podemos representar el teorema con la ***Figura*** 5.1.

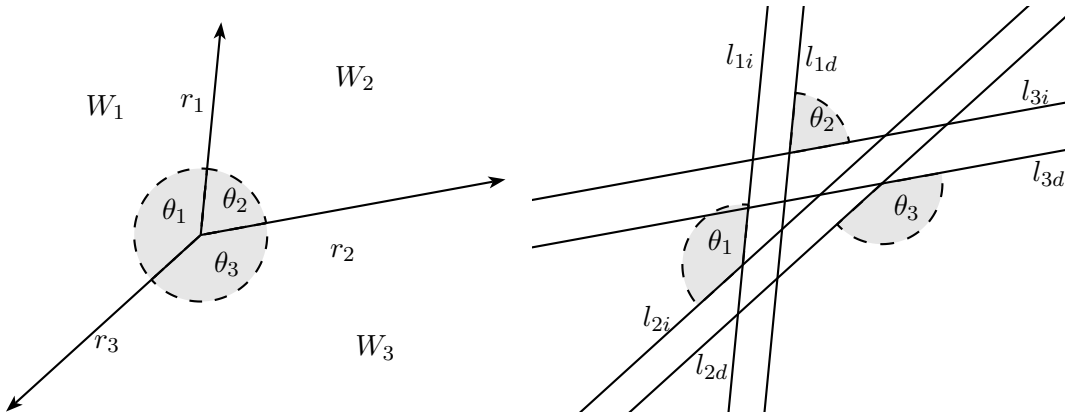


(a) Configuración inicial **(b)** Configuración final

Figura 5.1: Representación gráfica del **Teorema** 5.1

La prueba de este teorema es bastante gráfica y fácil de seguir, sin embargo hay algunas partes importantes que no son fáciles de notar en una lectura rápida, esto los vamos a abordar con todo detalle en la parte de la implementación. La prueba dada en [1] se divide en 2 partes, la primera es dar una orientación inicial a nuestras cuñas de manera que tengan los ángulos deseados y que emerjan del mismo punto, posteriormente iremos midiendo la cardinalidad de los puntos en cada cuña para ir modificando esta configuración hasta encontrar la configuración deseada.

Demostración del Teorema 5.1. Sean r_1, r_2 y r_3 rayos ordenados a favor de las manecillas del reloj que emergen del mismo punto. Estos rayos van a delimitar a las cuñas W_1,W_2 y W_3 respectivamente, eso quiere decir que entre esos rayos hay una ángulo de θ_1 que corresponde a la apertura de la cuña W_1. Esto quiere decir que entre r_1 y r_2 se forma un ángulo de θ_2; entre r_2 y r_3 un ángulo de θ_3; y entre r_3 y r_1 un ángulo de θ_1, en la **Figura** 5.2(a) podemos encontrar un ejemplo de configuración inicial. Una vez que tenemos nuestra configuración inicial, vamos a proseguir a mover dicha configuración hasta encontrar la deseada, es decir que cada W_i contenga exactamente k_i puntos.



(a) Orientación inicial **(b)** Límites de los rayos

Figura 5.2: Podemos ver un ejemplo de orientación inicial con los límites de cada uno de sus rayos.

El siguiente paso es encontrar el punto x del cual emergen las cuñas, el procedimiento será ir moviendo a los rayos manteniendo los ángulos que forman entre ellos, por esto mismo es importante notar que a partir del momento en el que elegimos una orientación inicial, los rayos no van a cambiar el ángulo entre ellos, lo cual significa que las cuñas van a tener exactamente la misma configuración que la inicial a excepción de punto del que emergen.

Consideremos a L_1 la familia de rectas paralelas a r_1. Lo siguiente es encontrar la ubicación correcta para r_1, donde la ubicación correcta contempla que se cumpla el número de puntos en las cuñas W_1 y W_2 que son las cuñas a las que en parte define r_1. Sea $l_1 \in L_1$ tal que hay al menos k_1 puntos a su izquierda y al menos k_2 puntos a su derecha, notemos que esto nos define dos límites para la ubicación de l_1, uno de esos límites nos dice lo más a la izquierda que puede estar y el otro lo más a la derecha, estos límites están acotados por los puntos que dejan al menos k_1 y k_2 puntos a la izquierda y derecha respectivamente. A estos límites les llamaremos l_{1i} para el izquierdo y l_{1d} para el derecho, cabe destacar que cuando l_1 se encuentra lo más a la izquierda posible, hay exactamente k_1 puntosa su izquierda, lo mismo para cuando está lo más a la derecha posible y hay k_2 puntos a su derecha. Análogamente las rectas l_2 y l_3 con los rayos r_2 y r_3, de manera que cada rayo tiene sus propios límites izquierdo y derecho. Podemos ver un ejemplo en la **Figura** 5.2(b).

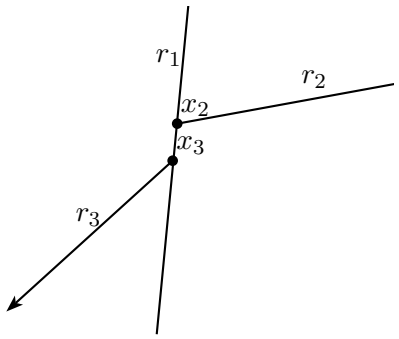


Figura 5.3: Moviendo a r_1. Notemos que al mover a r_1 a la izquierda se alejan x_2 y x_3, sin embargo si la movemos a la derecha, x_2 y x_3 se acercan.

La idea es encontrar un lugar para l_1 para mover a r_1 sobre la recta, pero al mover l_1 hacemos que la configuración se modifique, de forma que los rayos de cada cuña no emergen de un mismo punto. Llamaremos x_2 al punto sobre l_1 del que emerge r_2, y x_3 al punto del que emerge r_3, si coincide que x_2 y x_3 son el mismo punto y ya se cumplen las cardinalidades, entonces ya terminamos, de lo contrario habrá que ir moviendo la configuración hasta que x_2 y x_3 coincidan. En la **Figura** 5.3 podemos ver un ejemplo de cuando $x_2 \neq x_3$. También podemos ver como cuando l_1 se acerca a l_{1i}, es probable que x_3 se encuentre arriba de x_2 quien podría estar muy lejos en la dirección opuesta sobre l_1, análogamente cuando l_1 está cerca de l_{1d}, pero en el orden inverso. Por lo que en algún punto entre los dos límites, debe de existir un punto en el que x_2 y x_3 coincidan en el x buscado. ■

Con esto queda demostrado el teorema 5.1, pero sin entrar mucho a los detalles que mencionamos al inicio del capítulo, esto es porque es más fácil abordar esos detalles desde el punto de vista de la implementación, por lo que la próxima sección hablará más sobre la prueba anterior.

5.2. Implementación

La prueba dada para el **Teorema 5.1** es casi algorítmica, el problema es que no podemos sólo decirle al lenguaje de programación que busque el lugar ideal para l_1, la prueba se limita a decir que x existe y que en realidad $x = x_2 = x_3$. En esta sección vamos a ver la manera de encontrar a x sin perderse ningún detalle, además de dar un algoritmo de tiempo $O(n \log(n))$ en el que tripartimos el espacio de la manera descrita en el **Teorema** 5.1.

La implementación contempla dos partes, la primera es como en la prueba. Debemos elegir una configuración inicial para las cuñas en las que dividiremos el espacio, pero antes de eso tenemos que tomar en cuenta que las cuñas aún no están dadas, es decir que nosotros como programadores, debemos de contemplar crear los objetos geométricos que se suponen en las hipótesis, por lo que vamos a generar tres cuñas de manera específica.

Con algo que siempre contamos independientemente del algoritmo y de todo lo demás, es con un conjunto de reflectores tales que la suma de sus ángulos es de al menos 2π, sabiendo esto vamos a construir nuestras cuñas de tal forma que todo quede perfecto para la iluminación.

Usaremos el algoritmo 5.1 para crear las tres cuñas y ponerlas según la configuración inicial. Este algoritmo es de tiempo lineal, simplemente porque en esencia recorremos toda la estructura donde guardamos los ángulos de nuestras cuñas, para hacer el cálculo del ángulo de cada cuña.

Algoritmo 5.1: Algoritmo de tiempo $O(n)$ para crear las cuñas que dividirán al espacio.

```
1  angulo1 = 0 ;
2  angulo2 = 0 ;
3  angulo3 = 0 ;
4  rand = random(0, π);
5  i = 0;
6  while angulo1 < rand do
7  |  angulo1+ = angulos[i++];
8  end
9  rand = random(π − angulo1, π);
10 while angulo2 < rand do
11 |  angulo2+ = angulos[i++];
12 end
13 while angulo1 + angulo2 + angulo3 < π do
14 |  angulo3 = angulos[i++];
15 end
16 w1 = crearCuña(angulo1);
17 w2 = crearCuña(angulo2);
18 w3 = crearCuña(angulo3);
19 colocarCuñasEnPunto(w1, w2, w3, puntoAleatorio());
20 configuraconInicial(w1, w2, w3, puntos);
21 return w1, w2, w3;
```

Lo que nos dice el algoritmo, es que vamos a generar tres cuñas de apertura aleatoria, siempre que el ángulo de cada una de ellas sea menor a 2π. Hacemos esto con cuidado, porque después de generar la primer apertura aleatoria, la siguiente debe de ser de al menos $\pi - angulo_1$, para que la segunda cuña tenga la apertura mínima tal que la suma de ambos ángulos sea de al menos π, y con el último ángulo vamos a completar la suma de 2π.

Por último hacemos que las tres cuñas tengan como origen al mismo punto generado de forma aleatoria, esto lo hacemos en la línea 19 y nos toma tiempo $O(1)$, pues simplemente es hacer unos cálculos trigonométricos para hacer la rotación del ángulo de diferencia entre cada cuña. También vamos a hacer que queden una seguida de la otra como lo describimos en la sección anterior como configuración inicial.

Vamos a hacer unas modificaciones a esta configuración para comenzar con el algoritmo principal para segmentar la nube de puntos. Esto se logra usando el algoritmo 5.2.Este algoritmo toma tiempo $O(n\log(n))$ porque las líneas 4, 5 y 6 están ordenando los puntos con respecto a las rectas l_1,l_2 y l_3, en breve describimos ese proceso en el que verificamos el mínimo de puntos en el lado deseado.

Algoritmo 5.2: Algoritmo de tiempo $O(n\log(n))$ para calcular la configuración inicial de las cuñas

```
1  l1 = crearRecta(w1.vector1);
2  l2 = crearRecta(w2.vector1);
3  l3 = crearRecta(w3.vector1);
4  l1 = l1.minimoDePuntosIzq(k1);
5  l2 = l2.minimoDePuntosIzq(k2);
6  l3 = l3.minimoDePuntosIzq(k3);
7  w1 = colocarEn(l1);
8  w2 = colocarEn(l2.interseccion(l1));
9  w3 = colocarEn(l3.interseccion(l1));
10 return l1, l2, l3;
```

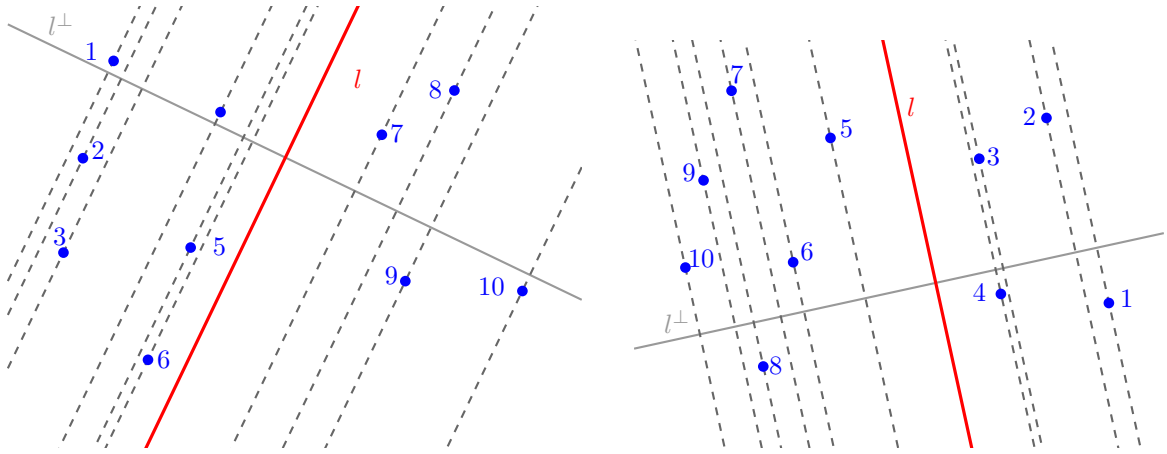
En la implementación para dejar k puntos de un lado de la recta, vamos a hacer uso del teorema de la equipartición (**Teorema 4.2**) pero un poco modificado. Si podemos encontrar una recta que divida al conjunto de puntos en dos partes de la misma cardinalidad, entonces también podemos encontrar una recta lo divida en conjuntos ajenos de cardinalidades distintas, gracias a esto es que podemos aplicar el teorema de la equipartición modificado para encontrar una recta que deje al menos k_i puntos de algún lado y al resto del otro.

El algoritmo utilizado para este teorema modificado es el descrito en el pseudocódigo 5.3. Este algoritmo tiene complejidad de $O(n\log(n))$ porque ya habíamos visto que el ordenamiento que usamos en la penúltima línea toma al menos ese mismo tiempo y es realmente el paso que toma más tiempo en ser ejecutado, ya que el ciclo for tiene una complejidad de $O(n)$ al recorrer los n puntos, después del orden basta con trasladar la recta entre el $k-ésimo$ punto y su siguiente en ese nuevo orden.

Algoritmo 5.3: Algoritmo de tiempo $O(n \log(n))$ para calcular el orden perpendicular a una recta

```
1 nuevosPuntos;
2 for Punto p : puntos do
3 |   nuevosPuntos.agregar(p.rotarEnContra(pendiente));
4 end
5 nuevosPuntos.ordenLexicoGrafico();
6 return l1.trasladar(nuevosPuntos[k].puntoMedio(nuevosPuntos[k + 1]);
```

Podemos encontrar un ejemplo de ordenamiento con respecto a una recta en la **Figura** 5.4.



(a) Recta con pendiente $m > 0$ **(b)** Recta con pendiente $m < 0$

Figura 5.4: Ordenamiento perpendicular a una recta según su pendiente, en este ejemplo ordenamos la misma nube de puntos con respecto a dos rectas diferentes.

Después de todo esto que hemos hecho, lo que nos queda es verificar que se cumpla que cada una de las cuñas tengan el número de puntos, si en ese momento cada una de las cuñas W_i tienen exactamente k_i puntos, entonces ya terminamos y habremos llegado a la configuración deseada, de lo contrario tendremos que ir moviendo a l_1 hasta encontrar la configuración deseada.

En algoritmo 5.4 hacemos una búsqueda binaria para encontrar el lugar ideal de l_i según lo descrito en la sección teórica. Lo primero que hacemos es ordenar de manera perpendicular los puntos con respecto a las 3 rectas, después calcula los límites izquierdo y derecho para l_1. Por último el método *haySolucion*() verifica que cada cuña W_i contenga k_i puntos, pero hacer eso tal cual viendo punto por punto tiene un costo lineal, por lo que describiremos la manera en que el programa lo hace para evitar eso.

Mientras el método *haySolucion*() devuelva falso, vamos a tener trasladar las cuñas a la nueva ubicación que encontremos con el método *contenerKPuntos*(), que recibe como parámetros las dos rectas que definen a los puntos y también la k que representa el número de puntos que debe de contener, posteriormente se calculará la intersección de l_2 con l_3, pues dependiendo de si ese punto se encuentra a la izquierda o derecha es que vamos a mover a l_1 a la mitad del rango que corresponda, según se encuentre el punto a la izquierda o a la derecha y suponiendo que aún no encontramos solución.

Algoritmo 5.4: Algoritmo de tiempo $O(n\log(n))$ para encontrar la configuración deseada de las cuñas

```
1  puntosl1 = ordenPerpendicular(puntos, l1);
2  puntosl2 = ordenPerpendicular(puntos, l2);
3  puntosl3 = ordenPerpendicular(puntos, l3);
4  l1L = l1.trasladar(puntosl1[k1]);
5  l1D = l1.trasladar(puntosl1[k1 + k3]);
6  while !haySolucion() do
7  |  l2 = l2.trasladar(w2.contenerKPuntos(l1, l2, k2, puntosl2));
8  |  l3 = l3.trasladar(w3.contenerKPuntos(l1, l3, k3, puntosl3));
9  |  w1 = w1.trasladar(interseccionl1_l3);
10 |  w2 = w2.trasladar(interseccionl1_l2);
11 |  w3 = w3.trasladar(interseccionl2_l3);
12 |  if l2.interseccion(l3).aLaIzquierda(l1) then
13 |  |  l1D = moverALaMitad;
14 |  else
15 |  |  l1L = moverALaMitad;
16 |  end
17 end
```

Desglosaremos parte por parte este algoritmo para ver sus complejidades y también los detalles que se omiten en la parte teórica. Lo primero que hace este algoritmo es ordenar de manera perpendicular con respecto a las tres rectas, eso ya fue explicado anteriormente y vimos que tiene una complejidad de $O(n\log(n))$. Después se calculan los límites izquierdo y derecho de l_1, notemos que en esta parte estamos utilizando el orden de los puntos de manera perpendicular a l_1, esto es para que podamos definir estos límites en tiempo $O(1)$, esta es la razón por la cual antes de comenzar el algoritmo hacemos los tres órdenes de los puntos, de esta forma no tendremos que ordenar los puntos cuando lo necesitemos.

Lo siguiente a analizar son los métodos *haySolucion*() y *contenerKPuntos*(). Comenzaremos con el segundo método, pues por comodidad dejaremos al final la comprobación de la solución. El pseudocódigo 5.5 describe al algoritmo *contenerKPuntos*().

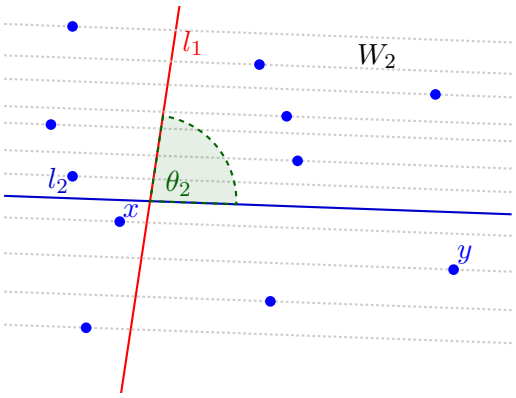
Este algoritmo va contando el número de puntos que quedan dentro de la cuña, toma tiempo lineal porque recorremos siempre una fracción lineal de puntos sobre la estructura de puntos ya ordenados de manera perpendicular a la recta l_i que será l_3 o l_2, dependiendo de la cuña que estemos configurando.

Algoritmo 5.5: Algoritmo de tiempo $O(n)$ que ubica a una cuña con exactamente k_i puntos

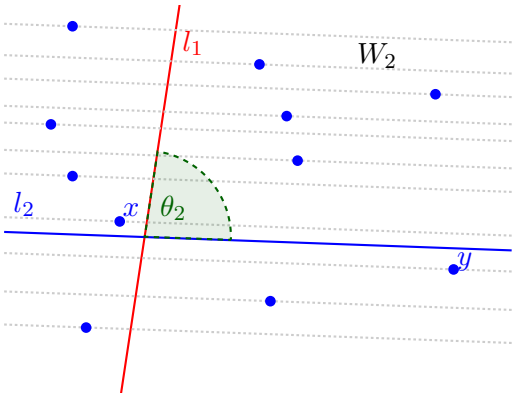
```
1  puntosEnCuña = 0;
2  for i = 0; i < puntos.length − 2; i++ do
3  |   medio = p.puntoMedio(puntos[i + 1]);
4  |   if !p.estaDelLadoCorrecto(l_1) then
5  |   |   continue;
6  |   else
7  |   |   puntosEnCuña++;
8  |   end
9  |   if puntosEnCuña == k then
10 |   |   l_i.trasladar(medio);
11 |   |   return l1.interseccion(l_i);
12 |   end
13 end
```

Es importante notar que este método utiliza muy fuertemente que los puntos ya están ordenados, pues en caso de que movamos una recta al siguiente lugar entre los puntos, vemos si el punto queda del lado correcto de la cuña, eso depende totalmente de la ubicación actual de l_1, pues si el punto está del lado de la cuña W_i, lo tomamos en cuenta, de lo contrario lo omitimos. En la **Figura** 5.5 podemos ver que si avanzamos a l_2 al siguiente intervalo entre x e y, no cambiamos el número de puntos que hay en W_2 por lo que ese punto es omitido con la instrucción *continue* de la línea 5 del pseudocódigo. Por otro lado, si continuamos al siguiente intervalo, agregamos un punto a W_2. Esta es la idea que seguiremos hasta que W_2 contenga k_2 puntos y análogamente para W_3.

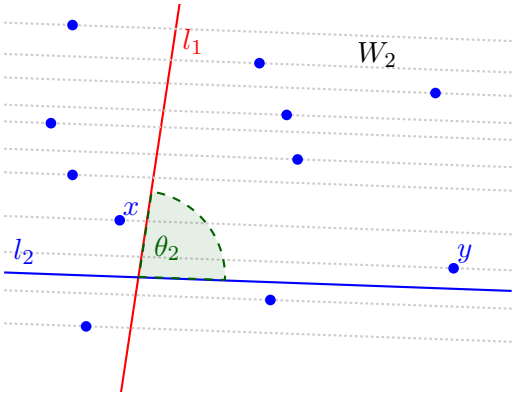
Recordemos que no estamos situando exactamente la ubicación de un punto para decidir colocar las rectas sobre éste, sino que tomamos los intervalos entre los puntos ordenados, para esto calculamos el punto medio y ahí ubicamos a la recta. Esto es porque es posible que la solución única se encuentre entre dos puntos y si fijamos las rectas sobre los puntos, podríamos no encontrar la solución nunca. Los casos especiales a tomar en cuenta son el primer y el último puntos en el arreglo, lo que se hace ahí es simular que hay un punto antes y colocamos la recta en ese intervalo. Se repite el mismo procedimiento con el último, pero simulando que hay un punto después.



(a) W_2 contiene 4 puntos



(b) A pesar de pasar un punto en el orden, no se agrega a W_2, pues está del lado incorrecto de l_1.

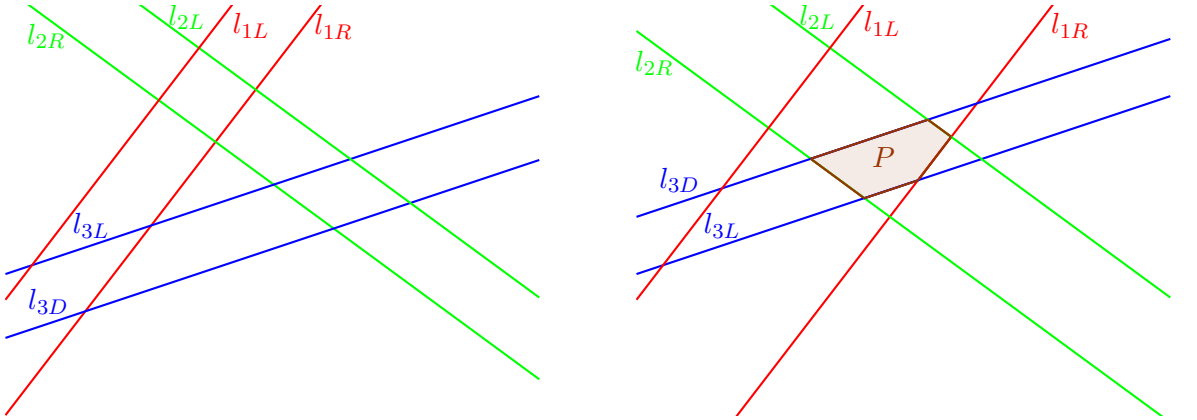


(c) En este paso si agregamos un punto a W_2, ahora tiene 5 puntos.

Figura 5.5: Ejemplo de cómo se acomoda W_2 de tal manera que contenga k_2 puntos, así podemos ir contando cuántos puntos agregamos a la cuña W_2.

Por último tenemos que revisar si ya encontramos una solución, retomando la idea que explicamos en la parte teórica, los límites de las tres rectas nos dirán si encontramos o no una posible solución. La solución va a estar determinada por dos cosas, que las intersecciones de las tres secciones limitadas sea no vacía y por que cada cuña W_i ya contenga exactamente k_i puntos. Lo último ya lo hacemos en cada repetición del ciclo, ahora debemos verificar que la intersección de los límites no sea vacía, en esa intersección podremos colocar al x del que emergen nuestras tres cuñas.

Una vez que ya calculamos el espacio solución P, ya podemos elegir cualquier lugar dentro de él para colocar a x. Estamos intersectando subespacios determinados por rectas, por lo que la solución nos va a determinar siempre un polígono convexo, así que lo que haremos es calcular el centroide del polígono y ahí pondremos a x. El centroide se calcula fácilmente sumando las coordenadas x y calculando el promedio; lo mismo para y.



(a) La intersección de los límites es vacía. No hay solución.

(b) La intersección de los límites es no vacía, el polígono P es un espacio solución

Figura 5.6: Evaluación del espacio solución.

Capítulo 6

La iluminación del plano y su implementación

En este capítulo vamos a combinar los resultados obtenidos en los dos capítulos anteriores para así lograr la iluminación total del plano y demostrar el ***Teorema*** 3.1. También vamos a hacer un pequeño recorrido por la aplicación resultante de este trabajo, ayudándonos con una pequeña guía introductoria al uso de ésta, así como un pequeño manual de instalación y la forma de acceder a este proyecto.

6.1. Iluminación del plano: El teorema de la iluminación

Ya tenemos los dos resultados que necesitábamos para poder lograr la iluminación del plano completo. El **Teorema** 5.1 nos dice cómo podemos segmentar el plano en tres cuñas ajenas, con exactamente k_i puntos arbitrarios dentro de cada cuña, además la unión de las tres cuñas, es el plano completo. Por otro lado, el **Teorema** 4.1 nos dice de qué manera podemos iluminar el complemento de una cuña con k puntos con k reflectores que irán sobre estos puntos. Juntando estos dos teoremas, podemos hacer la demostración del **Teorema** 3.1.La demostración es la siguiente.

Demostración del Teorema 3.1. Sea P una nube de n puntos en *posición general*. Sea R un conjunto de n reflectores cuyos ángulos cumplen que $\alpha_i \leq \pi$.

Usamos el **Algoritmo** 5.1 para crear tres cuñas a partir de los n reflectores con aperturas θ_1,θ_2 y θ_3 respectivamente, donde $\theta_i \leq \pi$ y depende de la suma del ángulo de subconjuntos de R aleatoriamente tomados y ajenos. Es decir que $\theta_1 = \alpha_1 + \alpha_2 + ... + \alpha_{k_1}$, $\theta_2 = \alpha_{k_1+1} + ... + \alpha_{k_2}$ y $\theta_3 = \alpha_{k_2+1} + ... + \alpha_n$. Llamaremos k_1 a la cardinalidad de los elementos en la primera suma, y análogamente llamaremos así a k_2 y a k_3 para la segunda y tercer suma.

Notemos que ahora tenemos los ángulos θ_1, θ_2 y θ_3; y tres enteros k_1, k_2 y k_3 tales que cumplen que $k_1 + k_2 + k_3 = n$. Con esto estamos cumpliendo las hipótesis del **Teorema** 5.1, por lo que podemos utilizarlo para dividir el plano en 3 secciones formadas por tres cuñas W_1,W_2 y W_3 que contienen k_1, k_2 y k_3 puntos respectivamente.

Cada cuña contiene k_i puntos, pero además tenemos un subconjunto de k_i reflectores de R para cada cuña, que cumple que la suma del ángulo de los reflectores es igual al ángulo de la cuña que contiene a los puntos, pues así construimos cada una de las cuñas. Esto hace que podamos utilizar el **Teorema** 4.1 para cada una de las cuñas e iluminar su complemento. La unión de las tres cuñas cubre a todo el plano, entonces la unión de sus complementos también cubre a todo el plano, haciendo que el espacio quede completamente iluminado.

Por último basta decir que el algoritmo que divide al espacio tiene una complejidad de $O(n\log(n))$, y la iluminación de cada cuña tiene una complejidad de $O(k\log(k))$, pero k es una fracción de n, por lo que su complejidad real es de $O(n\log(n))$ y lo hacemos tres veces. Son procesos ajenos, es decir que se aplican de manera independiente, por lo que el total del tiempo es: $O(n\log(n)) + O(3n\log(n)) = O(n\log(n))$. ■

La demostración anterior prueba que logramos la iluminación del plano por completo, y así cumplimos el objetivo proupuesto al final del ***Capítulo 3***: demostramos el ***Teorema*** 3.1, resolviendo así el *Problema de la Iluminación*.

6.2. La aplicación y sus usos

En esta sección hablaremos sobre la aplicación desarrollada a lo largo de este trabajo. Veremos un poco de los detalles técnicos, requerimientos de instalación, cómo instalarla y también de su uso.

El programa fue desarrollado en dos lenguajes de programación, *Java* y *Processing*.
Processing es un lenguaje de programación ejecutado sobre la máquina virtual de *Java*. Su objetivo principal es proveer de un lenguaje de programación fácil de utilizar para el desarrollo de aplicaciones en el ámbito de las artes visuales, haciendo de él una excelente herramienta para la realización de tareas como graficar, dibujar y animar. Si el lector desea más información sobre este lenguaje puede consultar la página: `https://processing.org/`

Como *Processing* es ejecutado sobre la máquina virtual de *Java*, sus desarrolladores diseñaron el lenguaje de forma que fuera posible (y muy fácil de hacer) que *Java* pudiera ejecutar código de processing dentro de sus programas. Esta aplicación se vale de estas facilidades, de manera que todo lo relacionado con el funcionamiento del programa trabaja con *Java*, excepto la parte de graficación, tarea que realiza *Processing*.

6.2.1. Requerimientos e instalación

Para poder ejecutar la aplicación es necesario tener los siguientes requerimientos:

1. La versión 1.8.0_51 de *Java*.
   Para descargar e instalar *Java*, podemos seguir las instrucciones en la página: `https://www.java.com/es/download/`

2. La aplicación ***The Floodlight Problem***: Podemos descargar la aplicación de la siguiente liga: `https://github.com/ahmora/FlooflightProblem`

Instalación:
Para poder instalar la aplicación basta con instalar correctamente *Java*. Una vez descargada la aplicación, debemos descomprimirla. La carpeta comprimida contiene algunas cosas:

1. *javadoc:*
   Esta carpeta contiene la documentación de la biblioteca desarrollada exclusivamente para este trabajo. Podemos acceder a la página principal a través del archivo *Index.html*

2. *lib:*
   Es una carpeta que contiene las bibliotecas necesarias y adicionales a las de *Java*. Encontraremos dentro dos archivos: Uno llamado ***beansbinding.jar*** que contiene las configuraciones administrativas para poder ejecutar el proyecto. El otro archivo se llama ***core.jar*** que es toda la biblioteca de *Processing*.

3. *FloodlightProblem.jar*
   Este archivo es el ejecutable de la aplicación.

Después de descomprimir la carpeta basta con ir ejecutar el archivo ***TheFloodlightProblem.jar*** como cualquier aplicación realizada en *Java*. En linux, esto se traduce a que en la terminal ejecutemos el comando ***java -jar FloodlightProblem.jar***. Después de esto, el programa mostrará su ventana principal, donde el usuario podrá comenzar a utilizarlo.

En la siguiente sección haremos un pequeño tour por la aplicación.

6.2.2. Uso general de la aplicación

En esta sección aprenderemos un uso básico de la aplicación desarrollada para la realización de este trabajo. Comenzaremos con un vistazo a la ventana principal en la ***Figura*** 6.1, esta ventana está dividida en seis partes con una tarea específica para cada una. Podemos ver que están numeradas, a continuación hay una explicación más detallada de cada una de éstas.

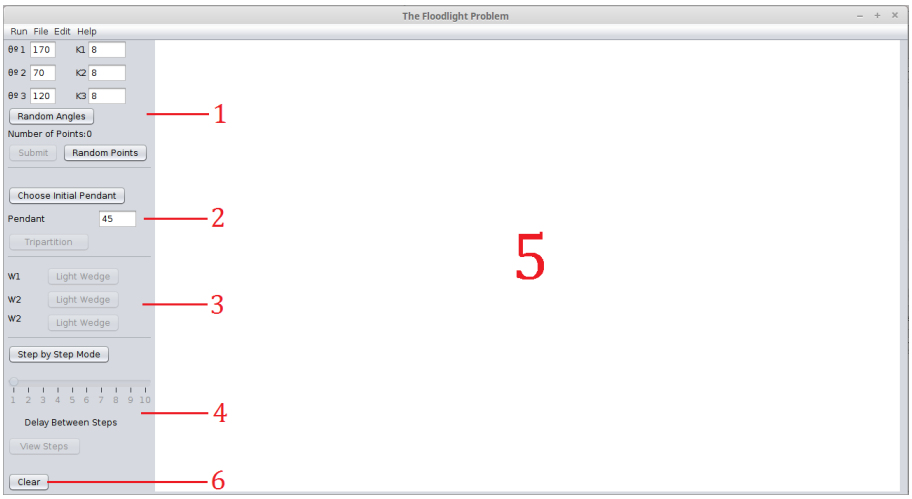


Figura 6.1: Vista general de la aplicación.

1. ***Panel de Configuración:***

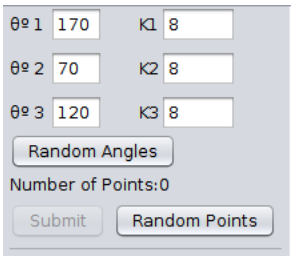


Figura 6.2: Panel de Configuración.

El panel de configuración sirve para determinar los parámetros iniciales de nuestro programa. Es aquí donde nosotros vamos a introducir todos los valores iniciales de las tres cuñas con las que vamos a trabajar. Podemos apoyarnos de la ***Figura*** 6.7 para entender mejor la descripción hecha en el texto.

En la parte superior izquierda del panel vemos tres etiquetas y tres campos de texto, es en estos campos donde vamos a introducir los valores para los ángulos θ_1, θ_2 y θ_3 correspondientes a las tres cuñas W_1, W_2 y W_3 respectivamente que vamos a generar. Inmediatamente a la derecha de esta parte encontramos algo muy parecido con las etiquetas k_1, k_2 y k_3, estos valores corresponden a la cardinalidad de los puntos que van a contener W_1, W_2 y W_3. Estos campos son validados para que el usuario siempre ponga valores positivos y dentro de los rangos necesarios.

Posteriormente encontramos el botón ***"Random Angles"*** que sirve para generar los valores de θ_1, θ_2 y θ_3 de manera aleatoria. Estos valores siempre serán menores a 180 y además su suma es de 360.

Después de botón, está un indicador que nos dice cuántos puntos podemos ver en el lienzo (explicado en el punto 5), éste indicador se actualiza en tiempo real. Después del indicador encontramos el botón ***"Random Points"*** que sirve para generar de manera aleatoria nuestro conjunto de puntos. Dicho conjunto contiene $k_1+k_2+k_3$ puntos. Junto a este botón se encuentra el botón ***"Submit"*** que sirve para avisar al programa que ya hemos colocado manualmente a nuestro conjunto de puntos en el lienzo, este botón no estará activo hasta que sea un conjunto de puntos cuya cardinalidad sea de $k_1 + k_2 + k_3$.

Una vez asignados los valores a los parámetros iniciales del programa, podemos comenzar.

2. ***Panel de Tripartición:***

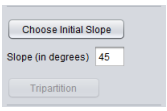


Figura 6.3: Panel de Tripartición.

Este es el panel más sencillo de todos, hay un botón que se enciende y apaga según esté seleccionado o no, dicho botón nos va a servir para decirle al programa si queremos seleccionar la pendiente inicial para hacer la tripartición de los puntos, de manera que podemos elegir la orientación de una de las rectas que forma a las tres cuñas con origen en el mismo punto. Podemos ver dicho panel en la ***Figura*** 6.3.

Si el botón ***"Choose Initial Pendant"*** está seleccionado, entonces la pendiente de la recta l_1 que divide a las cuñas W_1 y W_3 será igual al valor dentro del campo de texto debajo de éste junto a la etiqueta ***"Pendant"***.

Por último vemos el botón ***"Tripartition"*** que acciona el algoritmo que genera a las cuñas que triparten al conjunto de puntos. Este botón no será seleccionable hasta que no se tenga el número de puntos según los valores k_1, k_2 y k_3.

3. ***Panel de Iluminación:***

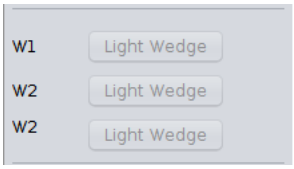


Figura 6.4: Panel de Iluminación.

En este panel decidimos si queremos encender o apagar los reflectores ya colocados de manera que iluminen al complemento de cada una de las cuñas. Podemos ver tres botones seguidos de tres etiquetas, en la que cada etiqueta representa a los reflectores dentro de la cuña W_i según la etiqueta, si el botón ***"Light Wedge"*** junto a la etiqueta W_i está seleccionado, entonces se iluminará en el lienzo el complemento de la cuña W_i.

Los botones no serán seleccionables hasta que se haya aplicado el algoritmo de la tripartición. La primera vez que queremos iluminar alguna de las tres cuñas, al accionar cualquiera de los tres botones, se calculará la solución para la iluminación de las tres cuñas, por lo que la primera vez, puede no ser inmediato el cálculo y sobre todo el pintado en el lienzo. Este panel está ilustrado en la ***Figura*** 6.4.

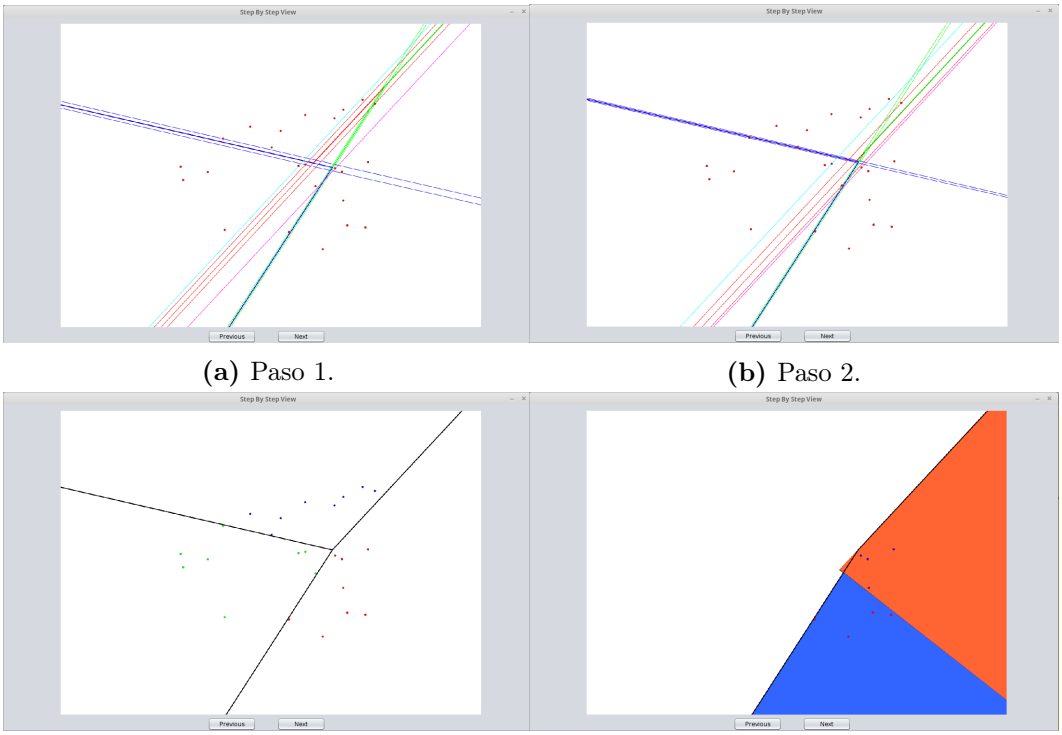
4. ***Panel de Vista:***

Figura 6.5: Panel de Vista.

En este panel vamos a decidir en qué modo va a operar nuestro programa. Hay dos formas de hacer que el programa se ejecute. La primera forma es que el programa ejecute los algoritmos y muestre simplemente el resultado, dibujando directamente a las tres cuñas después de aplicar el proceso de tripartición y posteriormente que encienda y apague a los reflectores dependiendo de la configuración del *Panel de Iluminación*.

Apoyándonos en la ***Figura*** 6.5 podemos ver que hay un botón que se puede encender o apagar, este botón es seleccionable siempre. Al seleccionar el botón ***"Step by Step Mode"*** le estamos diciendo al programa que queremos ver cada uno de los pasos durante la ejecución de ambos algoritmos, lo que significa que en la tripartición veremos paso por paso cada una de las iteraciones del algoritmo, posteriormente veremos como es que se llega a la configuración final de cada uno de los reflectores dentro de las cuñas.

Para proveer al usuario el tiempo que él necesite y a gusto del mismo, tendrá la opción de elegir el tiempo que quiere que dure la visualización de cada uno de sus pasos, de manera que contamos con un deslizador con valores del 1 al 10, cada uno de estos niveles representa el tiempo en segundos, así que si seleccionamos por ejemplo el número 3, el programa mostrará todos los pasos y los mostrará durante 3 segundos cada uno.



(a) Paso 1. **(b)** Paso 2.

(c) Paso 3. **(d)** Paso 4.

Figura 6.6: Pasos ejecutados por el algoritmo.

Si elegimos este modo de ejecución, al final de ver todos los pasos generados por los algoritmos, se habilitará el botón ***"View Steps"***, el cuál nos abrirá una ventana con todos los pasos que vimos del procedimiento y con la ventaja de poder elegir cuál ver y por cuánto tiempo, cosa que puede resultar muy conveniente para el usuario. Este botón ***no*** se habilitará si no está activado el modo de ver paso por paso.

Esta ventana es de un uso bastante sencillo. La interfaz simplemente tiene dos botones, ***Previous*** y ***Next***, los cuales servirán para movernos entre los pasos que nos va a mostrar. Además de poder usar los botones también podremos usar las teclas: *Av. Pag.*, *Re. Pag.*, flecha a la derecha y flecha a la izquierda; para movernos entre los pasos realizados por el algoritmo.

5. ***Lienzo:***
   Es la parte blanca donde se dibuja todo lo antes descrito. Siempre que modifiquemos alguno de los valores para k_i tendremos que cumplir con ese número de puntos, cuando lo hagamos automáticamente el programa ya no nos permitirá dibujar más puntos, y también el botón ***"Tripartition"*** será seleccionable para poder continuar con los algoritmos como se describió antes.

6. ***Botón de limpieza:*** Este botón nos va a ayudar a quitar todos los puntos y cuñas del lienzo

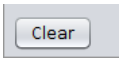


Figura 6.7: Botón de limpieza.

y regresar a una configuración predeterminada. En ésta, ninguna de las opciones seleccionables están activadas, las tres cuñas tienen la misma apertura y cardinalidad de puntos, simulando la apertura del programa desde el inicio.

Con esto terminamos el recorrido por la aplicación que resuelve el *Problema de la Iluminación*, dejando pendiente únicamente las conclusiones de este trabajo, además de los aportes que puede tener la aplicación.

Capítulo 7

Conclusiones y comentarios finales

Este trabajo nos muestra un poco de la historia sobre los problemas de *Galerías de Arte*. Además de enunciar algunos problemas y enseñarnos diferentes técnicas de Iluminación, con las que podemos trabajar con cada uno, dependiendo de su naturaleza. Se estudiaron más a detalle dos: la iluminación del plano y también la iluminación de *Polígonos Ortogonales*.

Iluminando Polígonos Ortogonales:
En el caso de la iluminación de *Polígonos Ortogonales*, podemos darnos cuenta de que cuando nos dedicamos a buscar "la mejor solución", siempre depende de lo que deseamos y del problema mismo. Un ejemplo muy claro de esto es el ***Capítulo*** 2.

En las ***Secciones 2.1.1 y 2.1.2*** se estudia un resultado que minimiza el número de guardias de 360º para iluminar un polígono ortogonal; sin embargo en la ***Sección*** 2.1.3
encontramos una nueva solución, en la que se utilizan más guardias que en la anterior, pero de una naturaleza diferente, con lo que optimizamos de la apertura total utilizada.

En la segunda solución, los guardias eran reflectores ortogonales que iluminan 90º. A pesar de que se utilizan más guardias, por lo anterior disminuye considerablemente la suma de las aperturas de los ángulos utilizados, ahorrando ángulo de visión total. Esta solución es "mejor" suponiendo que los guardias son dispositivos cuyo ángulo de visión es más limitado. Dependiendo de qué tipo de solución se necesite es que podemos usar un resultado o el otro. De modificarse un poco el problema (guardias u objeto geométrico), posiblemente podríamos buscar una manera óptima de resolver el problema con esta nueva modificación.

Trabajo a futuro:
Tomando como referencia el trabajo sobre polígonos ortogonales podemos realizarnos varias preguntas que quedan como incógnitas después de conocer ambas soluciones dadas:

- ¿Podremos reducir el número de vértices si la apertura de los reflectores ortogonales es de $\frac{3\pi}{2}$, y qué tanto?
- ¿Qué pasa con las cotas si se utiliza otro tipo de polígono? Ortogonal o no ortogonal.
- ¿Será posible iluminar el polígono Ortogonal utilizando reflectores no ortogonales, sin permitir dos reflectores en un mismo vértice?

El Problema de la Iluminación:
Por otro lado, también vemos que muchas veces es necesario dividir el problema que queremos resolver, en dos o más problemas relativamente más sencillos. Esta observación se encuentra un poco más marcada en el desarrollo del *Problema de la Iluminación*, ya que para poder resolver correctamente este problema, necesitamos el resultado de la trisección del conjunto de puntos, para poder lograr la tripartición del plano enunciada en el ***Teorema*** 5.1.

Una observación interesante sobre este problema, es que en la mayoría de los problemas de iluminación queremos iluminar objetos geométricos, y normalmente tenemos conflictos con obstáculos para poder lograr esto. En este caso aunque no tenemos paredes u obstáculos, la solución no es para nada trivial y más aún, si alguno de los reflectores tiene un ángulo mayor a 180^o, no sabemos si hay solución y mucho menos cómo calcularla de manera eficiente.

Trabajo a futuro:

- ¿Será cierto que encontrar una solución optima para *El problema de la Iluminación* utilizando algún reflector mayor a 180^o es un problema *NP*?

Sobre la implementación:
Cuando se está desarrollando un proyecto teórico-práctico, es importante saber que hay una gran diferencia entre ambos enfoques. A continuación se enuncian algunos de los puntos que más relevancia cobraron, en el momento de pasar de la parte teórica a la implementación.

Cuando estamos trabajando en el modelo teórico, muchas veces es más simple dar y seguir instrucciones, que geométricamente son fáciles de visualizar, comprobar y ejecutar, sin embargo no siempre es tan simple en la práctica. En el caso particular de este proyecto, tenemos ejemplos como: tomar una recta con cierta pendiente y moverla hacia alguna dirección, hasta que hayamos pasado cierto número de puntos; dividir una cuña, que tenga cierto ángulo de rotación con alguna de las rectas que la forman, dejando a la mitad de puntos en una u otra mitad de cuña; encontrar eventualmente un punto en el que se intersectan tres rectas, si las movemos en alguna dirección en particular.

Todos estos ejemplos de objetos que fácilmente podríamos mover en la imaginación o un dibujo, implican muchas otras cosas detrás del programa que el usuario jamás imagina. Otro ejemplo es la pendiente de las rectas que se utilizan, en particular en el momento de decidir cuál lado es la izquierda y cuál la derecha, los cuales se invierten dependiendo del signo del valor de la pendiente.

Otra de las principales cosas que debemos tomar en cuenta a la hora de implementar un programa geométrico, es que desde el inicio cuando le mostramos al usuario una recta, o algún objeto que se extiende al infinito, en realidad nosotros no estamos pintando tal cosa. Por ejemplo; para pintar una recta es necesario calcular la intersección con los bordes de la ventana y dibujar el segmento de recta que va de uno a otro borde de la ventana, así mismo con las cuñas, que al momento de dibujar una no es que vaya al infinito, debemos calcular las intersecciones con los bordes y dibujar el polígono formado; de esta manera el usuario tiene la ilusión de que las figuras se extienden al infinito.

La aplicación que desarrollada en este proyecto, sirve como material de apoyo para algún curso de *Análisis de Algoritmos*, *Geometría Computacional* o cualquier materia a fin con estas dos. No debemos olvidar que el trabajo para realizar la aplicación implica el desarrollo de una pequeña biblioteca de objetos geométricos, que por el momento está un tanto incompleta y con alguna que otra mejora por hacer. De manera que con el paso del tiempo, podría ir creciendo hasta formar una biblioteca de *Geometría* suficientemente robusta. Esto podría permitir el desarrollo de algún proyecto importante, que pueda desarrollar la ***Facultad de Ciencias de la UNAM*** o cualquier otra institución que desee hacer desarrollo en el área de la ***Geometría Computacional***.

Bibliografía

[1] Prosenjit Bose, Leonidas Guibas, Anna Lubiw, Mark Overmars, Diane Souvaine, and Jorge Urrutia. The floodlight problem. *International Journal in Computational Geometry*, 7:153–163, 1997.

[2] Thomas Cormen, Charles Erick Leiserson, Ronald Linn Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, third edition, 2009.

[3] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry Algorithms and Applications*. Springer, third edition, 2008.

[4] Satyan Linus Devadoss and Joseph O'Rourke. *Discrete and Computational Geometry*. Princeton and Oxford University Press, 2011.

[5] Jorge Urrutia, editor. *Art Gallery And Illumination Problems*. Universidad Nacional Autónoma De México, 2004.

[6] Joseph O'Rourke. *Art Gallery Theroems and Algorithms*. Oxford University Press, 1987.

[7] Jörg-Rüdiger Sack and Jorge Urrutia, editors. *Handbook of Computational Geometry - Chapter 22: Art Gallery And Illumination Problems*. North-Holland, 2000.

[8] Jurek Czyzowicz, Euardo Rivera-Campo, and Jorge Urrutia. Illuminating rectangles and triangles in the plane. *Journal of Combinatorial Theory*, 57:1–17, 1993.

[9] Davd Avis and Godfried Toussaint. An optimal algorithm for determining the visibility of a polygon from an edge. *IEEE Transactions on Computers*, pages 910–914, 1981.

[10] Nadia Benbernou, Erik Demaine, Martin Demaine, Anastasia Kurdia, Joseph O'Rourke, Godfried Toussaint, Jorge Urrutia, and Giovanni Viglietta. Edge-guarding orthogonal polyhedra. *Proceedings of the Twenty Third Canadian Conference on Computational Geometry*, 2011.

[11] Qihan Xu, Jonathan Tremblay, and Clark Verbrugge. Generative methods for guard and camera placement in stealth games. 2014.

[12] Felipe Contreras, Jurek Czyzowicz, Nicolas Fraiji, and Jorge Urrutia. Illuminating triangles and quadrilaterals with vertex floodlights. *Proceedings of 10th Canadian Conference on Computational Geometry (CCCG98)*, 1998.

[13] Jeff Kahn, Maria Klawe, and Daniel Kleitman. Traditional galleries require fewer watchmen. *SIAM Journal on Algebraic Discrete Methods*, 4:194–206, 1983.

[14] Vladimir Estivill-Castro and Jorge Urrutia. Optimal floodlight illumination of orthogonal art galleries. *Proceedings of the Sixth Canadian Conference in Computational Geometry*, pages 81–86, 1994.

[15] Herbert Edelsbrunner. *Algorithms in Combinatorial Geometry*. Springer-Velarg, 1987.

[16] Takeshi Tokuyama and Jun Nakano. Geometric algorithms for a minimum cost assignment problem. pages 262–271, 1991.