ESCUELA POLITÉCNICA NACIONAL

FACULTAD DE CIENCIAS

IMPLEMENTACIÓN DEL PROBLEMA DE ENCONTRAR TODAS LAS INTERSECCIONES DE N RECTAS HORIZONTALES Y VERTICALES.

TRABAJO DE INTEGRACIÓN CURRICULAR PRESENTADO COMO REQUISITO PARA LA OBTENCIÓN DEL TÍTULO DE INGENIERO MATEMÁTICO

LUIS ENRIQUE PIONCE GALLARDO

luis.pionce@epn.edu.ec

DIRECTOR: MARIA FERNANDA SALAZAR MONTENEGRO

fernanda.salazar@epn.edu.ec

DMQ, SEPTIEMBRE 2022

CERTIFICACIONES

Yo, LUIS ENRIQUE PIONCE GALLARDO, declaro que el trabajo de integración curricular aquí descrito es de mi autoría; que no ha sido previamente presentado para ningún grado o calificación profesional; y, que he consultado las referencias bibliográficas que se incluyen en este documento.

LUIS ENRIQUE PIONCE GALLARDO

Certifico que el presente trabajo de integración curricular fue desarrollado por LUIS ENRIQUE PIONCE GALLARDO, bajo mi supervisión.

MARIA FERNANDA SALAZAR MONTENEGRO

DIRECTOR

DECLARACIÓN DE AUTORÍA

A través de la presente declaración, afirmamos que el trabajo de integración curricular aquí descrito, así como el(los) producto(s) resultante(s) del mismo, es(son) público(s) y estará(n) a disposición de la comunidad a través del repositorio institucional de la Escuela Politécnica Nacional; sin embargo, la titularidad de los derechos patrimoniales nos corresponde a los autores que hemos contribuido en el desarrollo del presente trabajo; observando para el efecto las disposiciones establecidas por el órgano competente en propiedad intelectual, la normativa interna y demás normas.

LUIS ENRIQUE PIONCE GALLARDO

MARIA FERNANDA SALAZAR MONTENEGRO

DEDICATORIA

A Dios por darme la fortaleza y la sabiduría para poder lograr las metas que me propongo y por ser una luz, guía e inspiración en las decisiones más importantes de mi vida.

A mis padres Hugo Alfonso y Edith Marjorie quienes me apoyaron todos estos años brindándome un apoyo incondicional, por los valores y principios que cultivaron en mí, y por impulsarme a ser mejor y a culminar con éxito mi carrera.

A todos mis compañeros de la facultad por su valiosa amistad y por su colaboración fueron una parte importante en mi formación como profesional.

Finalmente, quiero dedicar esta tesis a mis docentes quienes con la enseñanza de sus conocimientos hicieron de mí un mejor profesional, en especial a la Dra. María Fernanda Salazar por su apoyo y dirección en este presente trabajo de titulación.

RESUMEN

En el presente trabajo de integración curricular se implementó un algoritmo propuesto por Jon L. Bentley y Thomas A. Ottmann (1979) [3], para encontrar y reportar todas las intersecciones de un conjunto de segmentos de recta verticales y horizontales. Se comenzó definiendo ciertos elementos de la teoría computacional, además se definieron las estructuras de datos que fueron utilizadas en el algoritmo.

El algoritmo fue implementado en el lenguaje de programación C++. En el desarrollo del algoritmo se realizaron dos variantes del mismo algoritmo, cuya variante consiste en la estructura de datos para almacenar los segmentos, primero se utilizó un árbol binario de búsqueda y luego un árbol binario de búsqueda auto-balanceado, este último permite mejorar los tiempos de ejecución del algoritmo.

Para poner en ejecución los algoritmos desarrollados, se construyeron las instancias simulando segmentos de recta aleatorios. También se implementó un algoritmo de fuerza bruta.

Finalmente, se ejecutaron distintas instancias, poniendo a prueba los algoritmos y comparándolos en términos de eficiencia.

Palabras clave: algoritmo, instancia, complejidad de algoritmos, estructura de datos, árbol binario de búsqueda, cola de prioridad, línea de barrido.

ABSTRACT

In this work, an algorithm proposed by Jon L. Bentley and Thomas A. Ottmann (1979) [3] was implemented with the aim of finding and reporting all the intersections of a set of vertical and horizontal line segments. It began by defining elements of the computational theory; then, the data structures used in the algorithm were defined.

The algorithm was implemented in C++ programming language. In the development of the algorithm, two variants of the same algorithm were carried out. Each variant corresponds to a data structure used to store the segments, first a binary search tree was used and then a self-balancing binary search tree. The latter improve algorithm performance.

To implement the developed algorithms, the instances were built simulating random line segments. A brute force algorithm was also implemented.

Finally, different instances were executed, testing the algorithms and comparing them in terms of efficiency.

Keywords: algorithm, instance, algorithm complexity, data structure, binary search tree, priority queue, sweep line.

Índice general

Índice de figuras

Capítulo 1

Descripción del componente desarrollado

En el presente trabajo de integración curricular se ha resuelto el problema de encontrar todos los puntos de intersección, de un conjunto de N segmentos de rectas horizontales y verticales en el plano.

Se proponen dos métodos de solución para este problema, el primero consiste en un algoritmo de fuerza bruta, cuyo fundamento es muy simple, probar todas las posibles combinaciones, comparar todos los pares de rectas y reportar todas las intersecciones de segmentos; sin embargo, este método es muy caro en términos computacionales.

El segundo método consiste en un algoritmo inteligente en donde se incluye una herramienta de geometría computacional conocida como línea de barrido; esta herramienta en conjunto con estructuras de datos clásicas como colas y árboles binarios de búsqueda, permiten mejorar los tiempos de ejecución.

Finalmente, se realizan pruebas computacionales y se compara el desempeño de los dos métodos de solución propuestos.

1.1. Objetivo general

Implementar un algoritmo inteligente, para encontrar todas las intersecciones de un conjunto de N segmentos de rectas horizontales y verticales dados.

1.2. Objetivos específicos

1. Implementar el algoritmo de fuerza bruta para el problema de estudio.
2. Implementar el algoritmo inteligente para el problema de estudio.
3. Determinar el orden de complejidad del algoritmo inteligente y compararlo con el algoritmo de fuerza bruta.
4. Realizar pruebas computacionales para los dos métodos de solución propuestos.

1.3. Alcance

Las metodologías propuestas serán implementadas en el lenguaje de programación c++.

Mediante el presente desarrollo, se espera poder brindar una alternativa más eficiente al problema de estudio. Además de dar un ejemplo de, cómo ciertas estructuras de datos pueden ayudar a resolver problemas de una manera más inteligente.

La búsqueda por fuerza bruta es sencilla de implementar y, siempre que exista, encuentra una solución; sin embargo, su coste de ejecución es proporcional al número de soluciones candidatas. Por este motivo, se implementará un algoritmo inteligente que permita resolver el problema de una forma más eficiente.

Para construir el algoritmo se considerará la idea simple pero poderosa de una línea de barrido: una línea vertical que se barre conceptualmente a través del plano. Dicha herramienta ayudará a poder identificar las posibles intersecciones de un conjunto de segmentos de recta dados.

Además, se estudiarán estructuras de datos dinámicas, que gracias a las operaciones que éstas permiten ejecutar; ayudarán a la construcción del algoritmo inteligente.

Finalmente se estudiará la eficiencia del algoritmo inteligente, es decir, se determinará el orden de complejidad del algoritmo, y se comparará con

el algoritmo de fuerza bruta.

1.4. Marco teórico

1.4.1. Antecedentes

El problema de encontrar todos los puntos que se intersecan en un conjunto de segmentos de recta horizontales y verticales surge en muchas aplicaciones. En el área de la electrónica, cuando se diseña un circuito integrado, los conductores suelen estar restringidos a líneas horizontales y verticales; detectar todos los cruces de conductores exige encontrar todos los pares de segmentos de recta verticales y horizontales que se cruzan.[3]

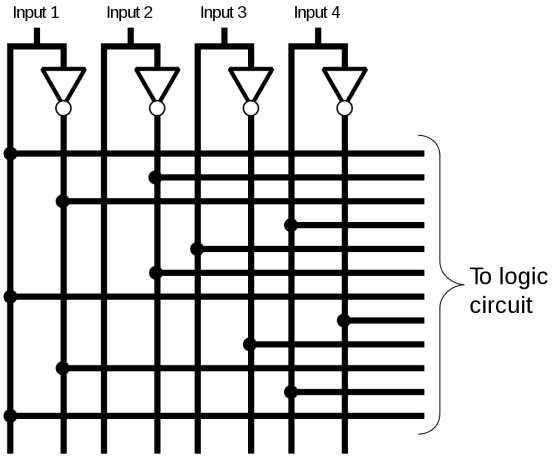

Figura 1.1: Un esquema de cableado típico de Manhattan (se omite el circuito lógico).

Los cuadrados y rectángulos orientados rectilíneamente se construyen a partir de segmentos de recta verticales y horizontales. Por lo tanto, se puede usar un algoritmo para encontrar pares de segmentos de recta verticales y horizontales que se intersecan.

Este problema geométrico también surge en gráficos por computadora y sistemas de información geográfica (SIG): calcular la disposición de un conjunto de n segmentos que se pueden colorear de rojo y azul para que no haya cruces rojo/rojo o azul/azul. Algunos problemas geométricos importantes se pueden abstraer como intersección de segmentos rojo/azul:

recorte de polígonos en gráficos por computadora, operaciones booleanas en diseño 2D asistido por computadora (CAD/CAM) y superposición de mapas en SIG, un objeto que se muestra oscurece a otro si sus proyecciones en el plano de visualización se cruzan.[2]

La importancia de desarrollar algoritmos eficientes para detectar intersecciones se está volviendo evidente a medida que las aplicaciones industriales se vuelven cada vez más ambiciosas. Un solo circuito integrado puede contener decenas de miles de componentes, una imagen gráfica complicada puede involucrar cien mil vectores. En tales casos, incluso los algoritmos que son solo cuadráticos respecto del número de objetos son inaceptables.[11]

1.4.2. Fundamentos y Definiciones

DEFINICIÓN 1. *"Un **algoritmo** es un conjunto de instrucciones ordenadas para resolver un problema, es decir, para obtener un resultado requerido para cualquier instancia legítima en un tiempo finito".[9] Una instancia es un caso especial de problema que contiene todos los parámetros del problema.*

Uno de los parámetros más importantes de una instancia es el tamaño de la entrada N, que denota el número de datos a ser considerados. Más adelante, se verá como este parámetro es absolutamente decisivo para determinar la eficiencia de un algoritmo. Sin embargo, hay muchos algoritmos que se comportan de distintas maneras independientemente de N, si no, que debido a la estructura del algoritmo su comportamiento varía en función de las especificaciones de la instancia. Por ejemplo, en el problema de estudio que corresponde a encontrar intersecciones de segmentos de recta, se verá cómo el algoritmo inteligente que será construido más adelante mejora considerablemente, cuando hay pocas intersecciones respecto del número N de segmentos, a diferencia del algoritmo de fuerza bruta el cual es invariante respecto del número de intersecciones.[1]

Complejidad de algoritmos

Considerando que uno de los objetivos del presente trabajo es comparar los algoritmos, una manera de evaluar la "bondad" de un algoritmo es mediante su *complejidad*, la cual consiste en medir el número de operaciones elementales que realiza un algoritmo hasta llegar a una posible solución. Para poder realizar estas comparaciones, es necesario antes introducir varias definiciones, las cuales ayudarán a tener una base teórica que permita comparar de manera objetiva los algoritmos.

DEFINICIÓN 2. ***Operaciones elementales****: son aquellas que el ordenador realiza en tiempo acotado por una constante. Así, se consideran operaciones elementales las operaciones aritméticas básicas, asignaciones a variables, los saltos (llamadas a funciones y procedimientos, retorno desde ellos, etc.), las comparaciones lógicas y el acceso a estructuras indexadas básicas, como son los vectores y matrices.*[6]

DEFINICIÓN 3. ***Notación Big O****. Sea $f : \mathbb{N} \to [0, \infty)$, se define al conjunto de funciones de orden $O(g(n))$ como:*

$$O(g(n)) = \{f(n) : \textit{existen constantes no negativas } \alpha \textit{ y } n_0 \textit{ tales que } 0 \leq f(n) \leq \alpha g(n), \textit{ para todo } n \geq n_0\}.[5] \tag{1.1}$$

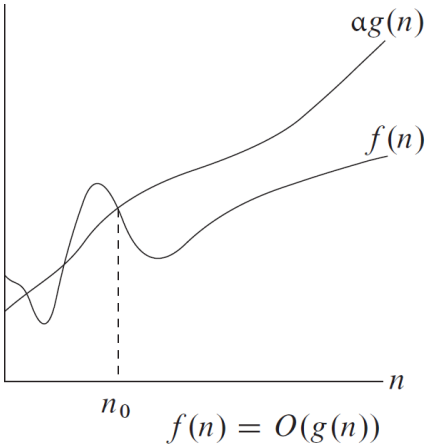

Figura 1.2: Ejemplo de notación Big O: a partir de n_0, la función $f(n)$ es acotada por $\alpha * g(n)$, describiendo el comportamiento límite de f.

Entre las clases de funciones más comunes de acuerdo con su orden de complejidad se encuentran:

Orden	**Notación**
constante	$O(1)$
logarítmico	$O(log(n))$
lineal	$O(n)$
loglineal	$O(nlog(n))$
cuadrático	$O(n^2)$
polinomial o algebraico	$O(n^c)$
exponencial	$O(c^n)$, $c > 1$
factorial	$O(n!)$

Cuadro 1.1: Clases de funciones más comunes en análisis de orden de complejidad de algoritmos.

Las clases de funciones del cuadro 1.1 se ordenan de la siguiente manera:

$$O(1) \subset O(log(n)) \subset O(n) \subset O(nlog(n)) \subset O(n^2) \subset O(n^c) \subset O(c^n) \subset O(n!)$$

Estructuras de datos

Para implementar algoritmos eficientes, se requiere de un manejo óptimo de los recursos del computador. Para ello es importante introducir objetos o estructuras dinámicas que tengan la capacidad de crecer o reducir su tamaño dependiendo de las necesidades del algoritmo.

El algoritmo que se implementará más adelante debe mantener al menos dos estructuras de datos dinámicas, una de las cuales debe admitir tanto operaciones de consulta: *búsqueda*, *sucesor* y *predecesor*, como operaciones de modificación: *inserción* y *eliminación*.

DEFINICIÓN 4. ***Estructura de datos:*** *una estructura de datos es una forma particular de organizar un grupo de datos, generalmente optimizada para un almacenamiento eficiente, búsqueda rápida, recuperación rápida y/o modificación rápida.[8]*

DEFINICIÓN 5. ***Puntero****: Un puntero es un objeto que almacena una dirección de memoria, el cual permite acceder a la información guardada en dicha dirección de memoria.[7]*

DEFINICIÓN 6. ***Nodo****: Un nodo es la unidad básica de una estructura de datos, la cual es a menudo usada para almacenar los atributos de di-*

chas estructuras. Los nodos suelen vincularse entre sí mediante el uso de punteros.[8]

DEFINICIÓN 7. ***Árbol:*** *es un conjunto de uno o más nodos tales que, en primer lugar, hay un solo nodo designado llamado raíz y, en segundo lugar, los nodos restantes se dividen en* $n \geq 0$ *conjuntos disjuntos,* T_1*,* T_2*,...,* T_n*, donde cada uno de estos conjuntos son en sí mismo un árbol, denominados subárboles de la raíz. [4]*

DEFINICIÓN 8. ***Árbol binario****: es un árbol en el cual cada nodo tiene a lo más dos subárboles, llamados subárbol* ***izquierdo*** *y* ***derecho****.[4]*

DEFINICIÓN 9. ***Árbol binario de búsqueda****: El árbol binario de búsqueda (ABB) se organiza, como sugiere su nombre, en un árbol binario, como se muestra en la figura 1.3. Se puede representar dicho árbol mediante una estructura de datos enlazados en la que cada nodo es un objeto. Cada uno de los nodos posee los atributos hijo izquierdo, hijo derecho y padre los cuales son accedidos mediante los punteros* $izquierdo$*,* $derecho$ *y* $padre$ *respectivamente. Además de una clave* (key) *y datos satelitales. Si alguno de los atributos de un nodo no existe, entonces su correspondiente puntero contiene el valor de* $NULL$*. El nodo raíz es el único nodo del árbol cuyo padre es* $NULL$*.[5]*

A diferencia de un árbol binario, para cualquier nodo x*, las claves en el subárbol izquierdo de* x *son menores que la clave del nodo* x *(*$x.key$*), y las claves en el subárbol derecho de* x *son al menos igual a la clave del nodo* x *(*$x.key$*). Diferentes árboles de búsqueda binarios pueden representar el mismo conjunto de valores.*

Debido a las propiedades que posee el árbol binario de búsqueda, es posible realizar operaciones como búsqueda, sucesor o predecesor a partir de un nodo dado, lo cual conduce a que esta estructura de datos pueda tener muchas aplicaciones.

El tiempo de ejecución para la mayoría de las operaciones del árbol de búsqueda es proporcional a la altura del árbol *(véase [1])*. Sin embargo, la altura del árbol depende del orden en el que entran los objetos, como se puede observar en la figura 1.3. Los objetos son los mismos, pero han ingresado en distinto orden; de allí que se tengan distintos casos y por

tanto, sus operaciones tendrán distintos órdenes de complejidad. En el peor de los casos las operaciones son de orden $O(n)$, y en el mejor de los casos se obtiene un orden $O(log(n))$. [5]

Teorema 1. *Las operaciones de consulta sobre un árbol binario de búsqueda son de orden* $O(h)$.

Demostración. Para acceder a los elementos de un *ABB*, se empieza por el nodo raíz, cuya operación puede ser realizada en tiempo constante $O(1)$, y se desciende en los niveles del árbol mediante comparaciones, hasta completar la consulta. Por tanto, el número de comparaciones está acotado por la altura h del árbol, y por la definición 1.1 se concluye que estas operaciones son $O(h)$. □

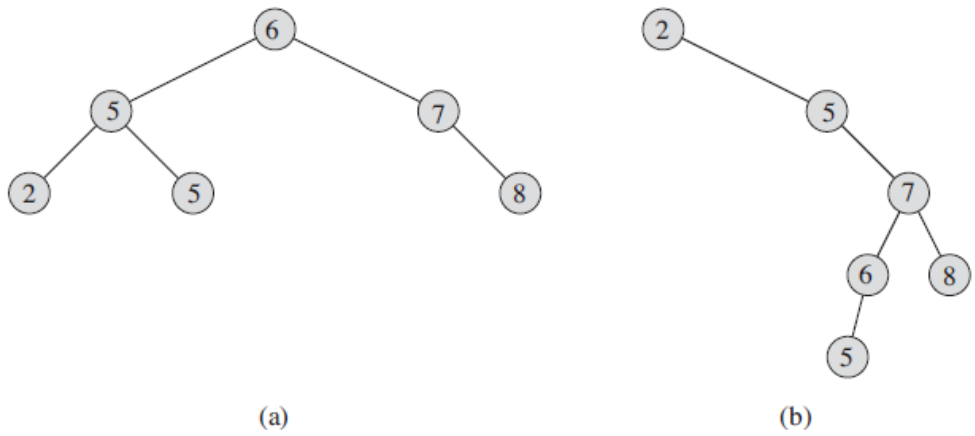

Figura 1.3: Árbol binario de búsqueda: (a) Un árbol binario de búsqueda de 6 nodos con altura 2. (b) Un árbol binario de búsqueda que contiene los mismos nodos, con altura 4, menos eficiente.

DEFINICIÓN 10. ***Cola de prioridad****: una cola de prioridad es una extensión de la cola, otro tipo de estructura de datos, que tiene las siguientes propiedades:*

- *Cada elemento de la cola posee un atributo llamado prioridad.*
- *Un elemento con una mayor prioridad sale de la cola antes que un elemento de menor prioridad.*
- *En el caso de que los elementos tengan una prioridad igual, los elementos serán manejados por la regla FIFO (first in/ first out), es decir, en una cola será eliminado siempre el elemento que ha estado en el conjunto durante más tiempo. En otras palabras, la cola implementa una política de primero en entrar, primero en salir. La propiedad FIFO de una cola hace que opere como una fila de clientes esperando para pagar en un cajero.*

En una cola de prioridad de tamaño n las operaciones de inserción y eliminación, toman un tiempo de $O(log(n))$, por tanto, construir la cola de prioridad, tomará un tiempo total de $O(nlog(n))$.

DEFINICIÓN 11. ***Árbol binario de búsqueda auto-balanceado:*** *Un árbol binario de búsqueda auto-balanceado posee las mismas características que un árbol binario de búsqueda, pero añade una propiedad de balance, la cual garantiza una construcción más eficiente del árbol. Esta propiedad consiste en que para cualquier nodo x, los subárboles izquierdo y derecho de x difieren en altura a lo más en una unidad. El sufijo "auto" hace referencia a que, si el árbol no cumple con la propiedad anteriormente expuesta, se realizará un reequilibrio del árbol para restaurar la propiedad.*

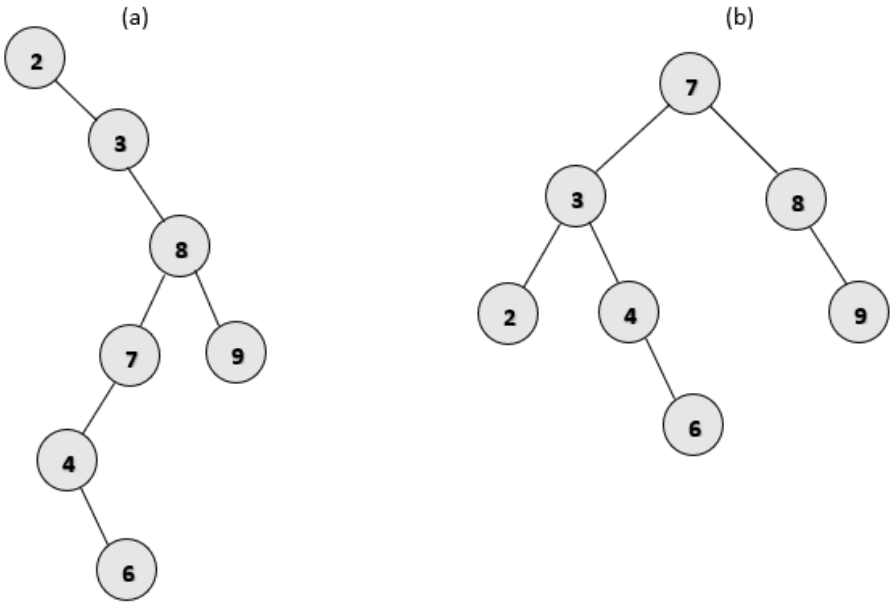

Figura 1.4: Árbol binario de búsqueda auto-balanceado: (a) Un árbol binario de búsqueda de altura 5, poco eficiente. (b) árbol "(a)" balanceado, su altura se reduce a 3.

Esta propiedad de balance garantiza que la altura sea la menor posible en un *ABB*, tal y como se puede observar en la figura 1.4, la altura del árbol izquierdo disminuye en dos unidades después de ser balanceado. Dado que las operaciones en esta estructura son proporcionales a su altura, se garantiza un orden $O(log(n))$.

Teorema 2. *La altura de un árbol binario auto-balanceado de n nodos es $O(log(n))$.*

Demostración. Sea T un árbol binario de búsqueda auto-balanceado de n nodos, sea h su altura, y sea $N(j)$ el número de nodos en un subárbol de T, de altura $j = 0, 1, ..., h$, es decir, para $j = h$ se tiene $N(h) = n$. Por la propiedad de *balance* que posee T, los subárboles izquierdo y derecho de cualquier nodo difieren en altura a lo más en una unidad. Sin pérdida de generalidad, vamos a suponer que para cualquier nodo x su subárbol izquierdo tendrá $N(j-1)$ nodos y su subárbol derecho tendrá $N(j-2)$ nodos. Por tanto, se verifica que

$$N(j) = N(j-1) + N(j-2) + 1$$

Desarrollando esta expresión se tiene:

$$
\begin{aligned}
N(j) &= N(j-2) + N(j-3) + 1 + N(j-2) + 1 \\
N(j) &= 2N(j-2) + N(j-3) + 2 \\
N(j) &> 2N(j-2) \\
N(j) &> 2 * 2N(j-4) \\
&\vdots \\
N(j) &> 2^{j/2} \qquad \text{si } j = h \text{ entonces } N(j) = n \\
n &> 2^{h/2} \\
log_2(n) &> \frac{h}{2} log_2(2) \\
\Leftrightarrow h &< 2log_2(n) \qquad \therefore \quad h \in O(log(n))
\end{aligned}
$$

□

Estructuras de datos populares que implementan este tipo de árbol:

- Árbol AVL (por las iniciales de quienes lo idearon Adelson-Velskii y Landis).
- Árbol rojo-negro.

Para las implementaciones se hará uso del árbol AVL, el cual emplea operaciones de rotación sobre los nodos para lograr el *balance* en el árbol.

Árbol binario de búsqueda construido aleatoriamente

Considerando que la altura de un árbol es de vital relevancia, pues sus operaciones dependen de su altura, es importante mencionar que la altura de un árbol depende de la naturaleza de las claves de los nodos y el orden en que estos sean insertados y eliminados.

Desafortunadamente, se sabe poco sobre la altura promedio de un *ABB* cuando se utilizan tanto la inserción como la eliminación para crearlo. Cuando el árbol se crea solo por inserción, el análisis se vuelve más manejable. Por lo tanto, se define un árbol binario de búsqueda construido aleatoriamente en n nodos como uno que surge de insertar las claves

de los nodos en orden aleatorio en un árbol inicialmente vacío, donde cada una de las $n!$ permutaciones de las claves de entrada es igualmente probable.[5]

Lema 1. *Dado un espacio muestral S y un evento A en el espacio muestral S, sea $X_A = I\{A\}$, donde $I\{A\}$ es la función indicadora del evento A. Entonces $E[X_A] = Pr\{A\}$.[5]*

Demostración. Sea S un espacio muestral y A un evento en el espacio muestral S, se define la variable aleatoria indicadora $I\{A\}$ asociada con el evento A como

$$I\{A\} = \begin{cases} 1 \text{ si el evento } A \text{ ocurre,} \\ 0 \text{ si el evento } A \text{ no ocurre.} \end{cases}$$

así, por la definición A.2 se tiene que,

$$\begin{aligned} E[X_A] &= E[I\{A\}] \\ &= 1 \cdot Pr\{A\} + 0 \cdot Pr\{\overline{A}\} \\ &= Pr\{A\}, \end{aligned}$$

donde $\overline{A}$ denota $S - A$, el complemento de A.[5]

□

Teorema 3. *La altura esperada de un árbol binario de búsqueda construido aleatoriamente con n nodos cuyas claves son distintas es $O(log(n))$.[5]*

Demostración. Se definirán tres variables aleatorias que ayudarán a medir la altura de un árbol binario de búsqueda. Sea X_n la altura de un árbol binario de búsqueda de n nodos, se define la altura exponencial como $Y_n = 2^{X_n}$. Cuando se construye un árbol binario de búsqueda de n nodos, se elige una $clave$ como la de la raíz y sea R_n la variable aleatoria que ocupa el rango de esta $clave$ dentro del conjunto de n claves; es decir, R_n ocupa la posición que ocuparía esta clave si el conjunto de claves estuviera ordenado. El valor de R_n es equiprobable respecto del elemento que tome del conjunto $\{1, 2, ..., n\}$. Si $R_n = i$, entonces el subárbol izquierdo de la raíz es un árbol binario de búsqueda construido aleatoriamente de $i-1$ nodos, y el subárbol derecho es un árbol binario de búsqueda construido

aleatoriamente de $n - i$ nodos. Debido a que la altura de un árbol binario es 1 más que la longitud de las alturas de los dos subárboles de la raíz, la altura exponencial de un árbol binario es dos veces más grande que las alturas exponenciales de los dos subárboles de la raíz. Dado que $R_n = i$, se sigue que

$$Y_n = 2 \cdot max(Y_{i-1}, Y_{n-i}).$$

Como caso base, se tiene que $Y_1 = 1$, pues la altura exponencial de un árbol con 1 nodo es $2^0 = 1$ y, por conveniencia, se define $Y_0 = 0$.

Por otro lado, se definen las variables aletorias $Z_{n,1}, Z_{n,2}, ..., Z_{n,n}$, donde $Z_{n,i} = I\{R_n = i\}$.

Ya que R_n toma cualquier elemento de $\{1, 2, ..., n\}$ de forma equiprobable, se sigue que $Pr\{R_n = i\} = 1/n$ para $i = 1, 2, ..., n$, y por tanto, por el lema 1, se tiene que

$$E[Z_{n,i}] = \frac{1}{n}, \tag{1.2}$$

para i = 1, 2, ..., n. Ya que exactamente un valor de $Z_{n,i}$ es 1 y los otros son 0, también se tiene que

$$Y_n = \sum_{i=1}^{n} Z_{n,i}(2 \cdot max(Y_{i-1}, Y_{n-i})).$$

Se debe mostrar que $E[Y_n]$ es polinomial en n, lo cual finalmente implicará que $E[X_n] = O(log(n))$.

Se afirma que la variable aleatoria $Z_{n,i} = I\{R_n = i\}$ es independiente de los valores de Y_{i-1} y Y_{n-i}. Habiendo escogido $R_n = i$, el subárbol izquierdo (cuya altura exponencial es Y_{i-1}) es construido aleatoriamente con los $i - 1$ nodos cuyos rangos son menores que i. Este subárbol es también un árbol binario de búsqueda construido aleatoriamente con los $i - 1$ nodos. Aparte del número de nodos que contiene, la estructura de este subárbol no se ve afectada por tomar $R_n = i$, y por tanto las variables aleatorias Y_{i-1} y $Z_{n,i}$ son independientes. De la misma manera, el subárbol derecho cuya altura exponencial es Y_{n-i}, es construido aleatoriamente con los $n - i$ nodos cuyos rangos son más grandes que i. Su estructura es independiente del valor de R_n, y así Y_{n-i} y $Z_{n,i}$ son independientes. Por

tanto,

$$
\begin{aligned}
E[Y_n] &= E\left[\sum_{i=1}^{n} Z_{n,i}(2 \cdot max(Y_{i-1}, Y_{n-i}))\right] \\
&= \sum_{i=1}^{n} E[Z_{n,i}(2 \cdot max(Y_{i-1}, Y_{n-i}))] && \text{(por linealidad de la esperanza)} \\
&= \sum_{i=1}^{n} E[Z_{n,i}]E[2 \cdot max(Y_{i-1}, Y_{n-i})] && \text{(por independencia)} \\
&= \sum_{i=1}^{n} \frac{1}{n} E[2 \cdot max(Y_{i-1}, Y_{n-i})] && \text{(por la ecuación (1.2))} \\
&= \frac{2}{n} \sum_{i=1}^{n} E[max(Y_{i-1}, Y_{n-i})] \\
&\leq \frac{2}{n} \sum_{i=1}^{n} E[Y_{i-1}] + E[Y_{n-i}] && (Y_{i-1},\ Y_{n-i} \text{ son no negativas})
\end{aligned}
$$

Ya que cada término $E[Y_0]$, $E[Y_1]$, ..., $E[Y_{n-1}]$ aparece dos veces en la última sumatoria una como $E[Y_{i-1}]$ y otra como $E[Y_{n-i}]$, se tiene la recurrencia

$$E[Y_n] \leq \frac{4}{n} \sum_{i=0}^{n-1} E[Y_i] \tag{1.3}$$

Usando el método de sustitución descrito en el anexo B, se prueba que para todo entero positivo n, la recurrencia 1.3 tiene la solución

$$E[Y_n] \leq \frac{1}{4}\binom{n+3}{3}$$

considerando la identidad

$$\sum_{i=0}^{n-1} \binom{i+3}{3} = \binom{n+3}{4} \tag{1.4}$$

Para los casos bases, nótese que las cotas $0 = Y_0 = E[Y_0] \leq (1/4)\binom{3}{3} = 1/4$ y $1 = Y_1 = E[Y_1] \leq (1/4)\binom{1+3}{3} = 1$ se cumplen. Por el caso inductivo, se

tiene que

$$
\begin{aligned}
E[Y_n] \leq \frac{4}{n}\sum_{i=0}^{n-1} E[Y_i] &\leq \frac{4}{n}\sum_{i=0}^{n-1}\frac{1}{4}\binom{i+3}{3} && \text{(por la hipótesis inductiva)}\\
&= \frac{1}{n}\sum_{i=0}^{n-1}\binom{i+3}{3}\\
&= \frac{1}{n}\binom{n+3}{4} && \text{(por la ecuación 1.4)}\\
&= \frac{1}{n}\frac{(n+3)!}{4!(n-1)!}\\
&= \frac{1}{4}\frac{(n+3)!}{3!(n)!}\\
&= \frac{1}{4}\binom{n+3}{3}
\end{aligned}
$$

Se ha acotado $E[Y_n]$, faltaría por acotar $E[X_n]$. Considere la función convexa $f(x) = 2^x$, por la desigualdad de Jensen [A.3], la cual establece que cuando aplicamos una función convexa a una variable aleatoria X, se tiene que $E[f(X)] \geq f(E[X])$. Por tanto,

$$
\begin{aligned}
2^{E[X_n]} &\leq E[2^{X_n}]\\
&= E[Y_n],
\end{aligned}
$$

así,

$$
\begin{aligned}
2^{E[X_n]} &\leq \frac{1}{4}\binom{n+3}{3}\\
&= \frac{1}{4}\cdot\frac{(n+3)(n+2)(n+1)}{6}\\
&= \frac{n^3+6n^2+11n+6}{24}\\
&\leq \frac{n^3+6n^3+11n^3+6n^3}{24}\\
&= n^3
\end{aligned}
$$

Tomando logaritmos a ambos lados se obtiene que $E[X_n] = O(log(n))$

[5] □

Capítulo 2

Metodología

2.1. Método y Enfoque

La metodología para resolver el problema de encontrar todas las intersecciones de segmentos de rectas horizontales y verticales está basada en el método deductivo, ya que se partió de la idea general de que las estructuras de datos dinámicas permiten almacenar datos y acceder a los mismos en tiempos relativamente óptimos, pues se busca implementar un algoritmo que mejore los tiempos de ejecución cuadráticos del algoritmo de fuerza bruta [2.1.2].

2.1.1. El problema

Se define un segmento de recta horizontal en $\mathbb{R}^2$ como el conjunto $\{(x, b) \in \mathbb{R}^2 : x_1 \leq x \leq x_2, \{x_1,\ x_2,\ b\} \subset \mathbb{R}\}$, el cual de ahora en adelante será denotado como $[(x_1, b); (x_2, b)]$, y el segmento vertical como el conjunto $\{(a, y) \in \mathbb{R}^2 : y_1 \leq y \leq y_2, \{y_1,\ y_2,\ a\} \subset \mathbb{R}\}$, el cual será denotado como $[(a, y_1); (a, y_2)]$. Dado un conjunto de N segmentos de rectas horizontales y verticales en $\mathbb{R}^2$, el problema consiste en reportar eficientemente todos los pares de intersecciones $(a, b) \in \mathbb{R}^2$, mejorando los tiempos de ejecución del algoritmo de fuerza bruta.

2.1.2. Algoritmo de Fuerza Bruta.

Un algoritmo de fuerza bruta para el presente problema compara todos los pares disjuntos de segmentos de recta. Para una instancia de n segmentos de recta existen $\frac{n}{2} * (n-1)$ pares disjuntos y además cada prueba de intersección se puede realizar en tiempo constante $O(1)$, por tanto este algoritmo es de orden $O(n^2)$.

El siguiente algoritmo permite determinar si existe intersección entre dos segmentos de recta, recibe un segmento vertical y otro horizontal y retorna *Verdadero* si existe intersección y *Falso* caso contrario.

Algoritmo 1 EXISTE-INTERSECCIÓN(S_1,S_2)

Entrada: Un segmento de recta vertical $S_1 = [(a, y_1); (a, y_2)]$ y otro horizontal $S_2 = [(x_1, b); (x_2, b)]$.

Salida: Verdadero si existe intersección, Falso caso contrario.

```
1: if S1.a ∈ [min(S2.x1, S2.x2), max(S2.x1, S2.x2)] then
2:     if S2.b ∈ [min(S1.y1, S1.y2), max(S1.y1, S2.y2)] then
3:         return Verdadero
4:     else
5:         return Falso
6:     end if
7: else
8:     return Falso
9: end if
```

El siguiente algoritmo permite encontrar todas las intersecciones de un conjunto de segmentos de recta verticales y horizontales:

Algoritmo 2 FUERZA-BRUTA

Entrada: Conjunto de N segmentos de recta verticales y horizontales.

Salida: Conjunto de puntos de intersección.

```
1: for i desde 1 hasta N − 1 do
2:     for j desde i + 1 hasta N do
3:         S1 = Arreglo[i]
4:         S2 = Arreglo[j]
5:         if S1 es vertical y S2 es horizontal then
6:             if EXISTE-INTERSECCION(S1, S2) then
7:                 Imprimir (S1.a, S2.b)
8:             end if
9:         else if S1 es horizontal y S2 es vertical then
10:            if EXISTE-INTERSECCION(S2, S1) then
11:                Imprimir (S2.a, S1.b)
12:            end if
13:        end if
14:    end for
15: end for
```

Análisis de complejidad del algoritmo de Fuerza Bruta

En primer lugar se tiene que el algoritmo *EXISTE-INTERSECCION* se resuelve en tiempo constante $O(1)$, y por otro lado se tiene que el doble lazo *for* corresponde a un número de operaciones de:

$$\begin{aligned}
\sum_{i=1}^{N-1}\sum_{j=i+1}^{N} j &= (N-1)+(N-2)+\ldots+(N-(N-1)) \\
&= (N-1)N - \sum_{i=1}^{N-1} i \\
&= N^2 - N - \frac{1}{2}(N^2-N) \\
&= \frac{1}{2}(N^2-N) \\
&\leq N^2 \qquad \text{por la definición 1.1, } \textit{FUERZA-BRUTA} \text{ es } O(N^2)
\end{aligned}$$

2.1.3. Línea de barrido

El verdadero desafío consiste en encontrar un algoritmo que sea sensible a la salida, en ciencias de la computación, un **algoritmo sensible a la salida** es un algoritmo cuyo tiempo de ejecución depende del tamaño de la salida, en lugar de, o además del tamaño de la entrada.[12] En el problema de estudio se traduce a que el tiempo de ejecución dependa no solo del número de rectas dadas, sino también del número de intersecciones. Se construirá una estrategia que permita comparar posibles intersecciones únicamente entre aquellos pares de segmentos que se intersecan.

Para lograr este objetivo, se puede empezar por considerar aquellos casos en los que sería inútil realizar una comparación. Sea un segmento de recta vertical $V = [(a, y_1); (a, y_2)]$ y un segmento horizontal $H = [(x_1, b); (x_2, b)]$, tales que $a \notin [x_1, x_2]$ es claro que V y H no se intersecan. En tales escenarios el algoritmo de fuerza bruta realiza un gasto innecesario de recursos. Entonces, si de alguna manera se pudiera mantener un orden sobre las rectas horizontales, tal que, las comparaciones se realicen con los segmentos verticales contenidos en los rangos de los segmentos horizontales; se evita las comparaciones innecesarias. Para abordar este problema, se introduce una herramienta geométrica conocida como **línea de barrido**.

Sea $S = \{s_0, s_1, ..., s_{n-1}\}$ un conjunto de n segmentos horizontales y verticales en $\mathbb{R}^2$, y sea $p_{ij} = s_i \cap s_j$, $i, j \in [0, ..., n-1]$, un punto de intersección entre dos segmentos. Una línea de barrido L es una recta vertical $x = p$, $p \in \mathbb{R}$, la cual se barre conceptualmente a través del eje x, etiquetando a todas los extremos de las rectas horizontales y a las rectas verticales, calculando intersecciones cuando L se posicione sobre un segmento vertical, es suficiente para capturar todos los puntos de intersección. Algunos de estos segmentos verticales de hecho no intersecan, pero veremos que el "esfuerzo malgastado" es reducido.

El plan es barrer L sobre los segmentos, deteniéndose sobre tres tipos de eventos, que permiten discretizar el "barrido":

1. se alcanza el extremo izquierdo de un segmento de recta horizontal,

2. se alcanza el extremo derecho de un segmento de recta horizontal, y
3. se alcanza la coordenada del eje x de un segmento de recta vertical.

Para lograr el orden de los segmentos respecto a la coordenada del eje x es importante utilizar una estructura de datos que facilite barrer la línea de barrido sobre los segmentos.

2.1.4. Motivación para los árboles binarios de búsqueda

Dado que el algoritmo inteligente necesita almacenar segmentos de recta, es importante que las estructuras empleadas sean lo más eficientes posibles. Si bien existen muchas estructuras que admiten operaciones de modificación con tiempo constante para ciertos casos, no pueden proporcionar operaciones eficientes de consulta de predecesor, sucesor y consulta de rango.

Consulta de predecesor

Sea T un árbol binario de búsqueda que almacena segmentos de recta, de n nodos y altura h; la operación de consulta de predecesor recibe un elemento $x \in T$ y retorna, si existe, el elemento más grande $y \in T$ tal que $y.key \leq x.key$. En un ABB esta operación se puede realizar fácilmente, recorriendo el árbol hasta encontrar x y retornando el máximo del subárbol izquierdo de x, así por el teorema [1] esta operación es $O(h)$. La consulta de sucesor es simétrica.

Algoritmo 3 PREDECESOR(x, z)

Entrada: El nodo *raíz* x, y el nodo z del cuál se busca su predecesor.

Salida: El nodo predecesor de z.

```
1: if z < x.key then
2:     return PREDECESOR(x.izquierda, z)
3: else if z > x.key then
4:     return PREDECESOR(x.derecha, z)
5: else
6:     if x.izquierda ≠ NULL then
7:         return maximo(x.izquierda)
8:     end if
9: end if
```

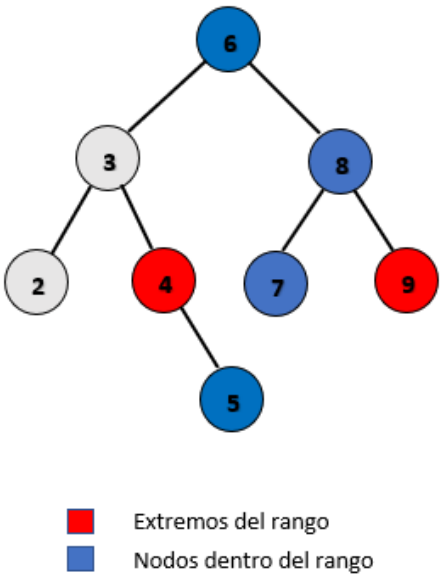

Figura 2.1: Ilustración de la búsqueda de rango sobre un ABB, en el rango $[4; 9]$, retornando todos los nodos coloreados.

Búsqueda de rango

Sea T un árbol binario de búsqueda de n nodos y altura h, y sean los extremos del rango $k_1, k_2 \in \mathbb{R}$. La operación de búsqueda de rango retorna todos los elementos $y \in T$ tales que $k_1 < y.key < k_2$. En un ABB esta función puede ser completada fácilmente realizando una búsqueda del nodo en T cuya clave sea la menor dentro del rango, y seguido de un recorrido sobre el árbol hacia la derecha, esta operación depende de la altura del árbol y retorna k elementos dentro del rango, así se tiene que el orden de esta operación es $O(h + k)$. Si el árbol está balanceado, entonces por el teorema [2], esta operación es $O(log(n) + k)$.

Para encontrar todos los nodos en un árbol, cuyas claves se encuentren en un rango dado $[k_1, k_2]$ se realiza el siguiente procedimiento recursivo .

Algoritmo 4 BUSQUEDA-RANGO(x, k_1, k_2)

Entrada: El nodo *raíz* x, y el rango $[k_1, k_2]$.

Salida: Imprime todas las claves de todos los nodos dentro del rango.

```
1: if k1 < x.key then
2:     BUSQUEDA-RANGO(x.izquierda, k_1, k_2)
3: else if k1 ≤ x.key ≤ k2 then
4:     imprimir(x.key)
5: else
6:     BUSQUEDA-RANGO(x.derecha, k_1, k_2)
7: end if
```

2.2. Validez del Enfoque

Antes de introducir un algoritmo a partir de las ideas expuestas en el precedente capítulo, se debe justificar si el enfoque que se ha tomado es capaz de llegar siempre a la solución correcta, sin importar el conjunto de segmentos de recta que sea recibido. La pregunta a responder es: ¿se encuentran todas las intersecciones?, para ello sean dos segmentos cualesquiera $S_1 = [(a, y_1); (a, y_2)]$ vertical y $S_2 = [(x_1, b); (x_2, b)]$ horizontal, tales que, existe un punto de intersección (a, b) entre ellos; ¿toma siempre la línea de barrido L esta posición en el plano?: En efecto, pues *L* se posiciona particularmente en cada segmento de recta vertical, asegurando eventualmente que *L* siempre tomará el valor $x = a$.

2.2.1. El algoritmo de Bentley-Ottmann

Jon L. Bentley y Thomas A. Ottmann (1979)[3] propusieron el siguiente algoritmo para reportar todas las intersecciones de un conjunto de segmentos de rectas horizontales y verticales en $\mathbb{R}^2$:

1. Inicialice una cola de prioridad Q de eventos. Cada evento posee dos atributos, uno de ellos es la ***prioridad*** que consiste en un punto en $\mathbb{R}^2$ y el otro un ***Segmento***, y ordenado según la coordenada del eje x de la *prioridad*. Así, Q contiene dos eventos para cada extremo de los segmentos horizontales y un evento para los segmentos verticales.

2. Inicialice un árbol binario de búsqueda T para almacenar segmentos horizontales que intersecan la línea de barrido L, ordenados según su coordenada en el eje y. Inicialmente, T está vacío.

3. Siempre que Q no esté vacío, saque un evento E de Q, y determine qué tipo de evento es para aplicar uno de los siguientes casos:

   - Si $E.Segmento$ es horizontal y $E.prioridad$ es un extremo izquierdo, inserte $E.Segmento$ en T.
   - Si $E.Segmento$ es horizontal y $E.prioridad$ es un extremo derecho, elimine $E.Segmento$ de T.
   - Si $E.Segmento$ es vertical entonces, se ejecuta una operación de búsqueda de rango entre los extremos de $E.segmento$.

A continuación, se detalla el funcionamiento del algoritmo sobre una instancia dada.

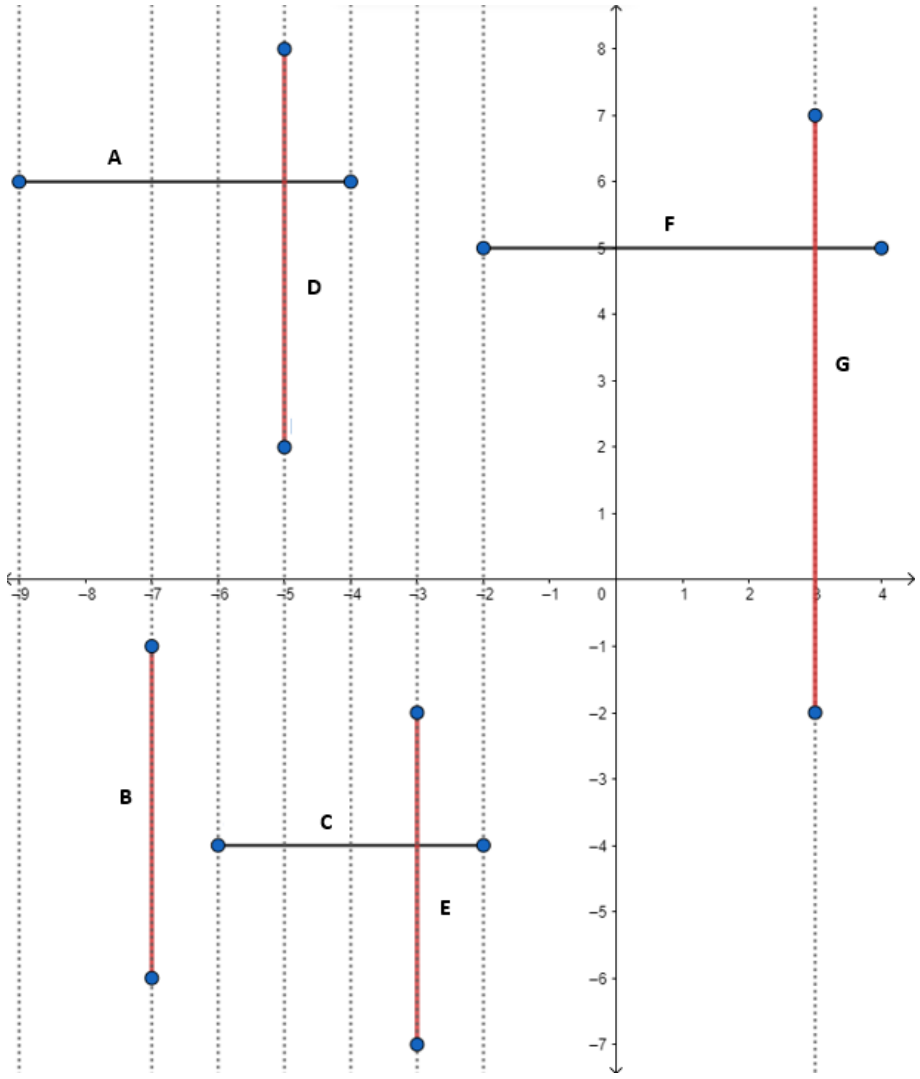

Figura 2.2: Algoritmo de línea de barrido de Bentley-Ottmann: La línea de barrido se muestra como la línea punteada, tomando la posición de todos los extremos de segmentos (puntos azules), y posicionándose eventualmente sobre las intersecciones.

Considere el conjunto de segmentos $S = \{A, B, C, D, E, F, G\}$ mostrado en la figura 2.2. Primero, la cola de eventos se inicializa en Q todos los extremos de los segmentos ordenados de izquierda a derecha. Cuando L alcanza el punto $x = -9$, se almacena el segmento A en la estructura T cuya *clave* dada por su coordenada en el eje y será $key = 6$, pues se trata de un extremo izquierdo de un segmento horizontal; luego L tomará el valor $x = -7$ del segmento B, y como se trata de un segmento vertical ejecuta la búsqueda de rango entre los extremos de B $(-7, -6)$ y $(-7, -1)$, como no existe un segmento horizontal en T que se encuentre en el rango dado, el algoritmo continúa con la siguiente posición. L toma el valor

$x = -6$ e ingresa el segmento C; L toma el valor $x = -5$, y al ser D un segmento vertical ejecuta la búsqueda de rango entre los extremos de D encontrando exitosamente al segmento A en la estructura T y reportando la intersección $(-5, 6)$. Luego la línea de barrido toma la posición $x = -4$ como es un extremo derecho de un segmento horizontal, entonces sale el segmento A de la estructura T. Y así continúa el algoritmo hasta detectar todas las intersecciones.

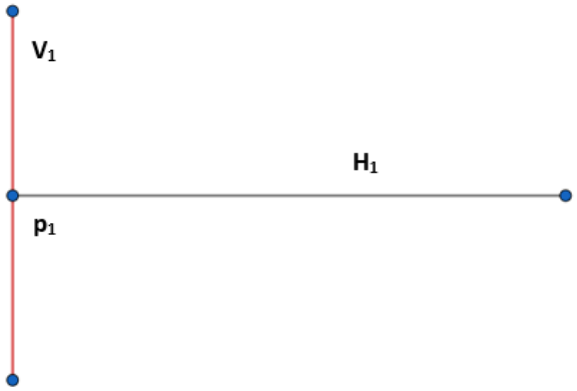

Figura 2.3: Intersección producida entre el extremo izquierdo de un segmento horizontal H_1 y un segmento vertical V_1.

Casos especiales

Es importante mencionar los casos que impiden el correcto funcionamiento del procedimiento. Como se mencionó en el capítulo precedente, si existen eventos que compartan la misma prioridad, la cola de prioridad almacenará los segmentos de acuerdo con la regla *FIFO*, es decir, el orden estará dado por la posición en la que fueron insertados, surgiendo posibles situaciones desfavorables al correcto funcionamiento del algoritmo inteligente.

La primera situación se da si el extremo izquierdo de un segmento horizontal interseca a un segmento vertical, tal como se observa en la figura 2.3; nótese que el problema tiene lugar si en el cuerpo principal del algoritmo primero sale de la cola de prioridad el segmento vertical V_1 antes del horizontal H_1, entonces se ejecuta una búsqueda de rango sobre la estructura T, sin reportar la intersección p_1, pues el segmento H_1 aún no se encuentra almacenado en T.

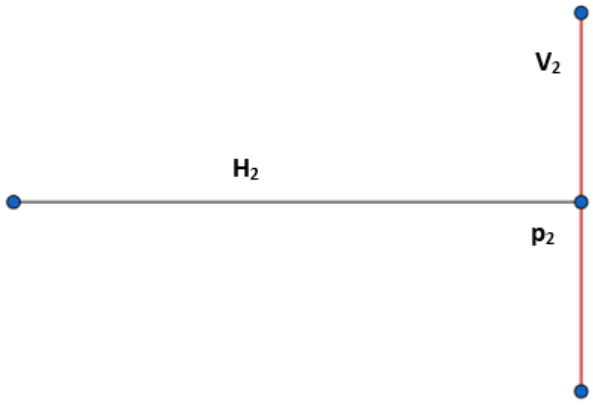

Figura 2.4: Intersección producida entre el extremo derecho de un segmento horizontal H_2 y un segmento vertical V_2.

Por otro lado, si el extremo derecho de un segmento horizontal interseca a un segmento vertical, tal como se observa en la figura 2.4; dado que al compartir la misma prioridad ambos eventos, se puede tener que primero salga de la cola de prioridad el segmento H_2, por tanto, H_2 sale de la estructura T, para luego seguir con el segmento V_2 y ejecutar una búsqueda de rango sobre la estructura T, sin reportar la intersección p_2, pues el segmento H_2 ya no se encuentra almacenado en T.

Estos casos pueden ser manejados, imponiendo un mayor número de operaciones elementales sobre el algoritmo.

2.2.2. Algoritmo Inteligente

Para simplificar la presentación del algoritmo de Bentley & Ottmann, se adaptará el mismo sin considerar los casos especiales y más adelante se lo modificará para admitir los mismos casos. Primero se restringe la entrada al caso donde todos los valores de la coordenada del eje x de los segmentos verticales y los extremos izquierdo y derecho de los segmentos horizontales son distintos por pares. En el problema de interés se recibe N segmentos de recta verticales y horizontales en el plano. Cada segmento de recta vertical V está especificado por su coordenada en el eje x como por sus extremos inferior y superior $bot(V)$ y $top(V)$ respectivamente. Cada segmento de recta horizontal H se especifica de manera similar por su coordenada en el eje y $y(H)$ y por los valores de x de sus extremos iz-

quierdo y derecho $left(H)$ y $right(H)$ respectivamente. Sea M el conjunto de coordenadas x con su respectivo segmento.

$$M = \{x(V);\ \text{A vertical}\} \cup \{left(B);\ \text{H horizontal}\} \cup \{right(H);\ \text{H horizontal}\}.$$

en donde M tiene a lo mucho $2N$ elementos.

El ciclo principal del algoritmo inteligente barre una línea vertical de izquierda a derecha a través del conjunto M. Se usa una estructura de datos dinámica T para almacenar segmentos de recta horizontal ordenadas por la coordenada en el eje y. Inicialmente T está vacío. Cada vez que se observa un extremo izquierdo (respectivamente derecho) de un segmento de recta horizontal H, H se inserta en (respectivamente se elimina de) la estructura T. Cuando se encuentra un segmento vertical durante el barrido, se ejecuta la *búsqueda de rango* entre los extremos del segmento vertical, retornando las intersecciones con los segmentos de recta horizontal en T.

Para el correcto funcionamiento del algoritmo se modificará la función *BUSQUEDA-RANGO*, de tal manera que ésta reciba un segmento de recta vertical $V = [(a, y_1); (a, y_2)]$, e imprima todos los puntos de intersección con V. A continuación, se detalla el procedimiento bajo el nombre *RANGO*.

Algoritmo 5 RANGO(x, k_1, k_2, V)

Entrada: El nodo *raíz* x, el rango $[k_1, k_2]$ y el segmento vertical V.

Salida: Imprime todos los puntos de intersección con el segmento V.

```
1: if k1 < x.key then
2:     RANGO(x.izquierda, k_1, k_2, V)
3: else if k1 ≤ x.key ≤ k2 then
4:     imprimir (V.a, x.key)
5: else
6:     RANGO(x.derecha, k_1, k_2, V)
7: end if
```

Para presentar el algoritmo se supone que se reciben N segmentos de recta, con n segmentos horizontales y m segmentos verticales. Sea $j = 1, 2, ..., m$ se denota t_j como el número de intersecciones que se reportan sobre el segmento vertical j-ésimo. Se describe el algoritmo inteligente en

pseudo-código como “Algoritmo 6 Encontrar intersecciones”.

Algoritmo 6 Encontrar intersecciones

		costo	*veces*
Entrada:	N segmentos de recta, n horizontales y m verticales.		
Salida:	Conjunto de puntos de intersección.		
1:	Q <- se construye la cola de prioridad.	$c_0 N log(N)$	1
2:	T <- se inicializa la estructura dinámica.	c_1	1
3:	**for** E en Q **do**	c_2	$2n + m$
4:	**if** $E.Segmento$ es horizontal **then**	c_3	$2n$
5:	**if** $E.prioridad$ es extremo izquierdo **then**	c_4	$2n$
6:	insertar $E.Segmento$ en T	$c_5 log(n)$	$2n$
7:	**else** $E.prioridad$ es extremo derecho	0	$2n$
8:	eliminar $E.Segmento$ de T	$c_6 log(n)$	$2n$
9:	**end if**		
10:	**else** $E.Segmento$ es vertical	0	m
11:	$A = sucesor(T.raiz, bot(S))$	$c_7 log(n)$	m
12:	$B = predecesor(T.raiz, top(S))$	$c_8 log(n)$	m
13:	RANGO($T.raiz$, $A.key$, $B.key$, $E.Segmento$)	$c(log(n) + t_j)$	m
14:	**end if**		
15:	**end for**		

Es fácil ver que el Algoritmo 6 encuentra correctamente todos los pares de segmentos de recta que se intersecan: Siempre que un segmento de recta vertical $V = [(a, y_1); (a, y_2)]$ se considera, T contiene exactamente los segmentos de recta horizontales que cruzan la recta vertical $x = a$ (ordenados por su coordenada en el eje y). Luego, el algoritmo informa todos los pares $(V.a, H.b)$ para todo segmento de recta horizontal $H \in T$ que interseca a V. La estructura de datos T se puede implementar como un árbol auto-balanceado. Por lo tanto, el tiempo para realizar las operaciones: *insertar*, *eliminar*, *sucesor* y *predecesor* es $O(log(n))$, y la función *RANGO* es $O(log(n) + k)$. El reporte de los pares que se cruzan se puede hacer en un tiempo proporcional a su número si usamos una implementación apropiada de árboles balanceados. [10] La estructura Q se puede implementar como una cola de prioridad. Clasificar los elementos de M y almacenarlos en orden creciente toma tiempo $O(Nlog(N))$.

El costo total de mantener T es $O(nlog(n))$: $n \leq N$ segmentos horizon-

tales insertados y eliminados en $O(log(n))$ cada uno.

Análisis del algoritmo inteligente

El algoritmo es presentado con el *costo* de tiempo de cada operación y el número de *veces* que cada una es ejecutada. El tiempo de ejecución del algoritmo es la suma de los tiempos de ejecución de cada operación ejecutada. Sea $T(N)$ el tiempo de ejecución del algoritmo 6, para calcular $T(N)$ se suma el producto de las columnas *costo* y *veces* del algoritmo 6, obteniendo:

$$\begin{aligned}
T(N) &= c_0 N log(N) + c_1 + c_2 N + c_3 2n + c_4 2n + c_5 2nlog(n) + c_6 2nlog(n) + \\
&\quad c_7 mlog(n) + c_8 mlog(n) + cm(log(n) + t_j) \\
&\leq c_0 N log(N) + c_1 + c_2 N + c_3 N + + c_4 N + c_5 N log(N) + c_6 N log(N) + \\
&\quad c_7 N log(N) + c_8 N log(N) + c(N log(N) + \sum_{j=1}^{m} t_j) \\
&\leq (c_0 + c_5 + c_6 + c_7 + c_8 + c_9 + c) N log(N) + (c_1 + c_2 + c_3 + c_4) N + c \cdot k \\
&\leq a \cdot N log(N) + b \cdot N + c \cdot k
\end{aligned}$$

Donde las constantes a, b y c dependen del costo de las operaciones, y $k = \sum_{j=1}^{m} t_j$ es el número de intersecciones encontradas. Así se tiene que

$$T(N) \leq (a + b) N log(N) + c \cdot k$$

Por la definición 1.1 se tiene que $T(N)$ es $O(N log(N) + k)$.

Para simplificar la presentación del Algoritmo 6 se asumió que ningún par de segmentos de recta comparten puntos finales o se encuentran en la misma recta.

Para poder controlar el caso de los extremos de segmentos horizontales que intersecan segmentos verticales, se tomó la siguiente estrategia para solucionar estos dos problemas:

1. Si el extremo izquierdo de un segmento horizontal interseca a un segmento vertical, se modificó la relación de orden en la cola de prioridad: si la prioridad de un evento corresponde a un extremo iz-

quierdo y la prioridad de un segmento vertical son iguales, se inserta primero el segmento horizontal. Así de esta manera se garantiza que primero salga de la cola de prioridad el evento correspondiente al segmento horizontal.

2. Si el extremo derecho del segmento horizontal interseca a un segmento vertical, se realizó la siguiente modificación del cuerpo principal del algoritmo almacenando en una cola de prioridad auxiliar a los segmentos horizontales antes de ser eliminados de la cola de prioridad principal, luego si el siguiente evento corresponde a un segmento vertical, se compara con la prioridad del último evento insertado en la cola auxiliar y si las prioridades de ambos eventos son iguales, entonces se vuelve a almacenar el segmento horizontal en la estructura T.

Ninguno de los casos especiales que se acaban de presentar alteran el orden del algoritmo $O(NlogN + k)$.

Capítulo 3

Resultados, conclusiones y recomendaciones

3.1. Resultados

En el capítulo anterior se presentó el algoritmo inteligente y se justificó que este logra reportar con éxito la solución al problema de estudio. Además, se demostró que el problema puede ser resuelto en un menor tiempo respecto del algoritmo de fuerza bruta.

En esta sección se presentarán los resultados de las ejecuciones de los siguientes algoritmos:

- **Algoritmo-AVL**: Algoritmo inteligente implementado con el árbol binario de búsqueda auto-balanceado, como estructura de datos para almacenar los segmentos.
- **Algoritmo-ABB**: Algoritmo inteligente implementado con el árbol binario de búsqueda, como estructura de datos para almacenar los segmentos.
- **Fuerza-Bruta**: Algoritmo de fuerza bruta.

Para poner a prueba el algoritmo inteligente se diseñaron varias instancias entre las cuales se consideran los casos especiales previamente mencionados, y también se crearon instancias aleatorias mediante generación de números pseudo-aleatorios. Con estos ejemplos se busca poner

a prueba el algoritmo inteligente en los casos especiales y además ejecutar las dos variantes del algoritmo inteligente con instancias relativamente grandes, para comparar el tiempo de ejecución con el algoritmo de fuerza bruta.

Las pruebas fueron realizadas en un computador con un chip Intel(R) Core(TM) i3-7020U CPU que funciona a 2.30 GHZ usando 8GB de RAM y corriendo windows 10.

3.1.1. Casos especiales

Extremos de segmentos horizontales que intersecan segmentos verticales

Considere el siguiente conjunto de segmentos que incluye los casos especiales mencionados en 2.2.1:

$$
\begin{aligned}
S = \{&s_1 = [(1,1);(1,9)] \\
&s_2 = [(2,1);(2,9)], \\
&s_3 = [(3,1);(3,9)], \\
&s_4 = [(9,1);(9,9)], \\
&s_5 = [(3,2);(9,2)], \\
&s_6 = [(1,3);(9,3)], \\
&s_7 = [(1,4);(9,4)], \\
&s_8 = [(1,5);(9,5)], \\
&s_9 = [(1,6);(9,6)], \\
&s_{10} = [(1,7);(9,7)], \\
&s_{11} = [(2,8);(9,8)]\}
\end{aligned}
$$

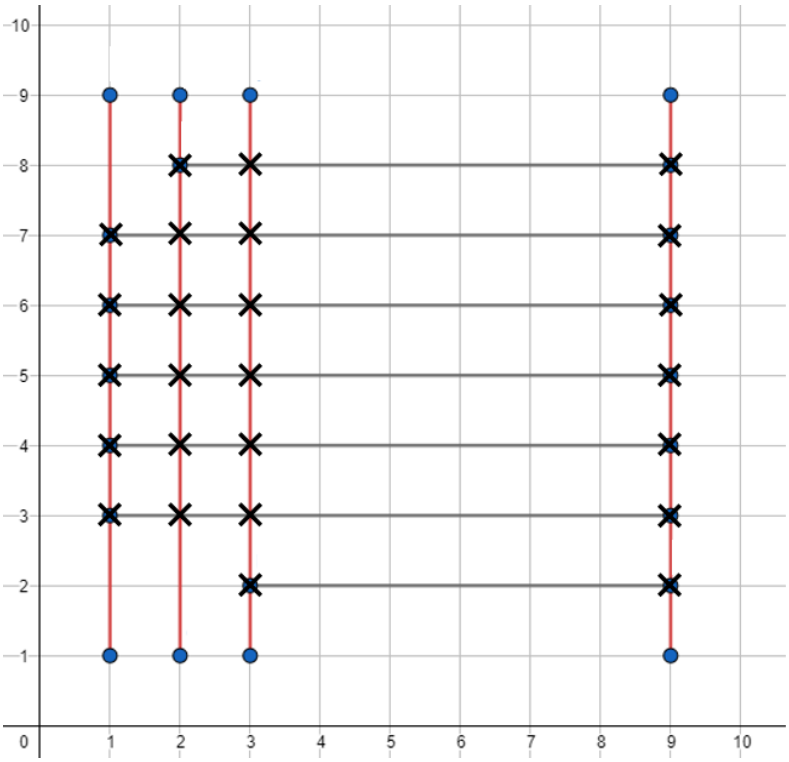

Figura 3.1: Intersecciones entre extremos de segmentos horizontales con segmentos verticales.

La solución que reporta el algoritmo es:

N^o	**X**	**Y**	N^o	**X**	**Y**
1	1	3	14	3	4
2	1	4	15	3	5
3	1	5	16	3	6
4	1	6	17	3	7
5	1	7	18	3	8
6	2	3	19	9	2
7	2	4	20	9	3
8	2	5	21	9	4
9	2	6	22	9	5
10	2	7	23	9	6
11	2	8	24	9	7
12	3	2	25	9	8
13	3	3			

Cuadro 3.1: 25 intersecciones encontradas con el algoritmo inteligente, sobre la instancia de casos especiales. Con un tiempo de ejecución de 0.004 segundos.

Nótese que el algoritmo encuentra exitosamente todas las intersecciones, reportando inclusive los casos especiales en un tiempo razonablemente eficiente.

3.1.2. Instancias Aleatorias

Dado que el orden del algoritmo inteligente es $O(Nlog(N) + k)$ donde N es el número de segmentos y k es el número de intersecciones, se puede deducir que, si en una instancia existen muchas intersecciones entonces, el comportamiento del algoritmo será parecido al de fuerza bruta, pues $k \leq N^2$, caso contrario si existen pocas intersecciones respecto del número de segmentos de recta, se obtendrán tiempos considerablemente más bajos en el algoritmo inteligente.

A continuación, se exponen los distintos casos y su eficiencia en términos de tiempo respecto del algoritmo de fuerza bruta.

Caso 1: pocas intersecciones respecto del número de segmentos de recta

Se generaron segmentos de recta de una longitud de 100 unidades, dentro de un rango de números aleatorios entre -5000 y 5000. Así es menos probable que existan muchas intersecciones.

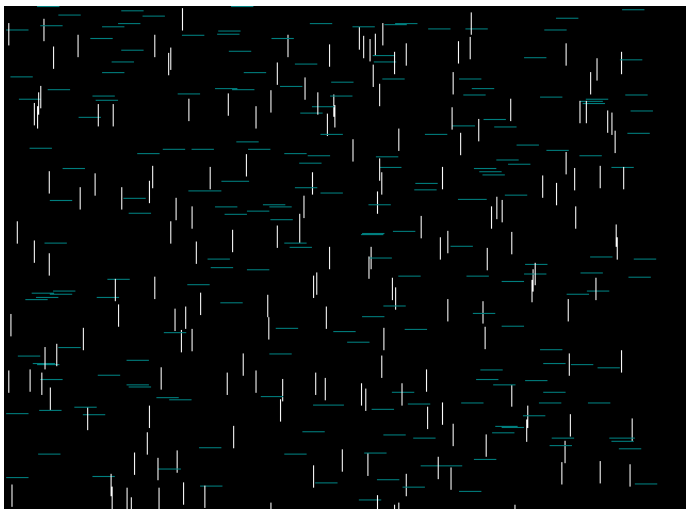

Figura 3.2: Segmentos de recta de longitud fija igual a 100 unidades (pocas intersecciones).

Dada esta instancia se obtuvieron los siguientes resultados:

Datos	Intersecciones	Algoritmo-AVL	Algoritmo-ABB	Fuerza-Bruta
500	6	0.003	0.003	0.012
5000	650	0.034	0.028	0.79
10000	2.491	0.129	0.082	2.881
20000	9.932	0.251	0.219	11.631
30000	22.306	0.511	0.472	26.157
40000	39.366	0.881	0.798	46.459
50000	61.828	1.341	1.16	76.715
60000	89.016	1.937	1.698	104.459

Cuadro 3.2: Resultados de las instancias de segmentos de longitud fija igual a 100 unidades (tiempos en segundos).

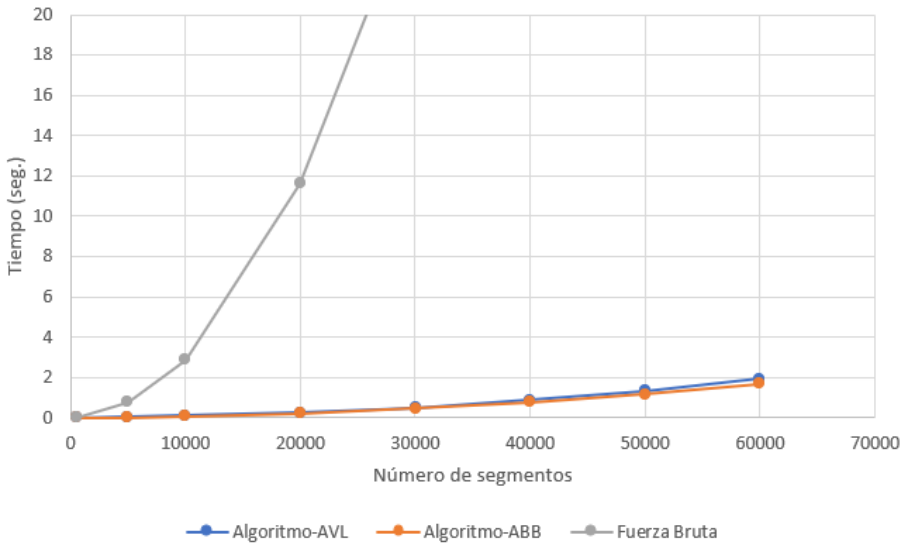

Figura 3.3: Rendimientos de los algoritmos para segmentos de recta de longitud fija (pocas intersecciones).

Como se puede observar en el gráfico [3.1.2] y la tabla [3.2], existe una diferencia en el desempeño entre los algoritmos inteligentes y el algoritmo de fuerza bruta, cuando existen pocas intersecciones. Este resultado era el esperado, pues el algoritmo inteligente es sensible a la salida, es decir, al número de intersecciones en la instancia.

Caso 2: muchas intersecciones respecto del número de segmentos de recta

Se generaron segmentos de recta de una longitud aleatoria, dentro de un rango de números aleatorios entre -5000 y 5000. Así es más probable que existan muchas intersecciones.

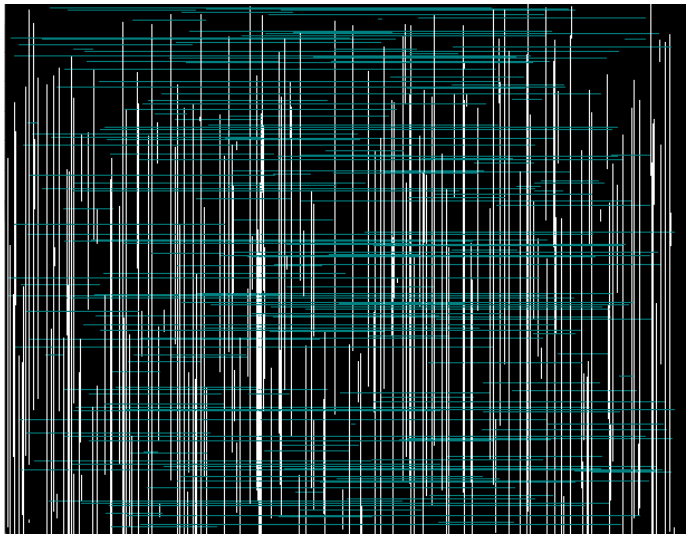

Figura 3.4: Segmentos de recta de longitud aleatoria (muchas intersecciones).

Dada esta instancia se obtuvieron los siguientes resultados:

Datos	Intersecciones	Algoritmo-AVL	Algoritmo-ABB	Fuerza-Bruta
500	7.066	0.128	0.112	0.119
5000	683.246	8.958	9.845	10.634
10000	2'748.238	40.319	43.467	58.969
15000	6'114.716	80.296	80.975	88.354
20000	10'990.768	149.2	144.069	156.286

Cuadro 3.3: Resultados de las instancias de segmentos de longitud aleatoria (tiempos en segundos).

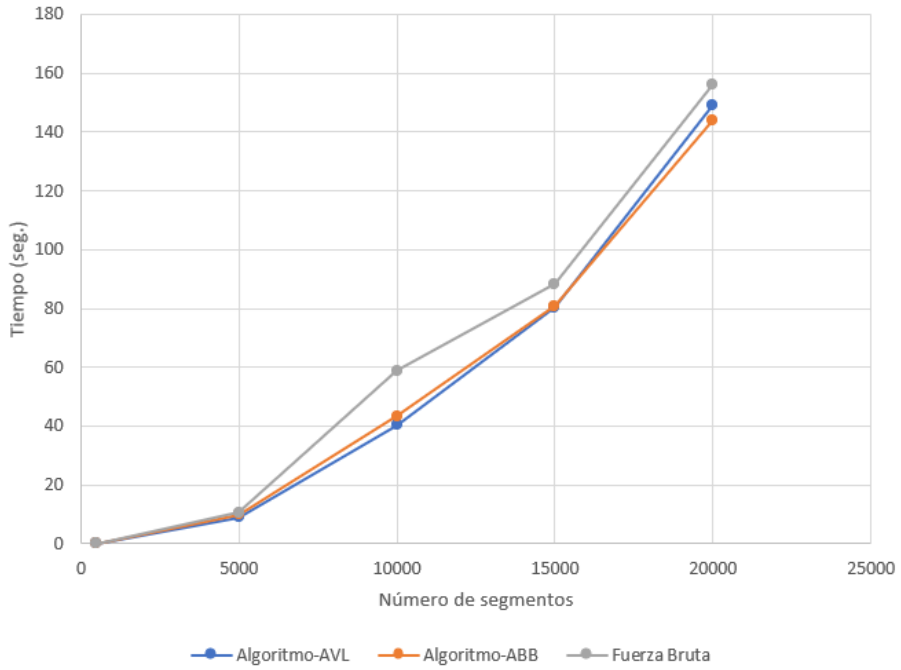

Figura 3.5: Rendimientos de los algoritmos para segmentos de recta de longitud fija (muchas intersecciones).

Nótese en la tabla 3.3 que el número de intersecciones reportadas es considerablemente mayor a la instancia previa, por tanto, se observa que los tiempos son muy parecidos en los tres algoritmos. Era lo esperado pues el algoritmo inteligente es sensible a la salida, es decir, al número de intersecciones reportadas.

3.2. Conclusiones y recomendaciones

El problema de intersección de segmentos es uno de los problemas fundamentales en la geometría computacional. El algoritmo presentado en la anterior sección fue propuesto por Bentley y Ottmann [3] en 1979. Sin embargo, otros científicos ya habían propuesto soluciones al problema de la detección de intersecciones.

Aún cuando Bentley y Ottmann demostraron que su algoritmo resuelve el problema en $O(Nlog(N)+k)$, donde k es el número de intersecciones, y el trabajo de almacenamiento es $O(N)$, el algoritmo en el peor de los casos se asemeja en términos de *orden de complejidad* al algoritmo de fuerza bruta, siempre que k sea suficientemente grande $(k \approx N^2)$.

Se pudo observar en los resultados que en ciertas instancias el algoritmo *Algoritmo-ABB* consigue mejores tiempos que el algoritmo *Algoritmo-AVL*, esto puede explicarse debido a que, por la naturaleza de la instancia presentada se consigue construir un *árbol binario de búsqueda aleatorio*, el cuál comparte el mismo orden de complejidad $O(Nlog(N)+k)$ con el árbol *AVL*. Además, el algoritmo *Algoritmo-AVL* debe de ejecutar un mayor número de operaciones que consisten en rotaciones de los nodos para lograr el *equilibrio* en el árbol, este gasto de recursos explicaría los mejores tiempos logrados por el algoritmo *Algoritmo-ABB*.

Se pudo observar cómo esta herramienta de *línea de barrido* en conjunto con eficientes estructuras de datos dinámicas han resultado en una alternativa de solución más eficiente al problema de estudio frente al enfoque de fuerza bruta.

Esta herramienta tiene muchas otras aplicaciones en problemas geométricos asociados a encontrar intersecciones, entre los cuales se encuentran resolver el problema de triangulación de polígonos, para calcular el llamado diagrama de Voronoi de un conjunto de puntos.

La idea de las líneas de barrido puede ser también adaptada a espacios de más de dos dimensiones, barriendo un hiperplano en el espacio. Tales algoritmos son llamados algoritmos de barrido espacial.

Capítulo A

Probabilidad

La probabilidad es una herramienta esencial para el diseño y análisis de algoritmos probabilísticos y aleatorizados. Esta sección repasa brevemente algunos conceptos de la teoría básica de la probabilidad. Se define la probabilidad en términos de un espacio muestral S.

DEFINICIÓN 12. ***Espacio muestral*** Ω***:*** *un espacio muestral es el conjunto de todos los posibles resultados de un experimento.*

DEFINICIÓN 13. ***Evento:*** *se describe un evento como un subconjunto del espacio muestral* $A \subset \Omega$.

DEFINICIÓN 14. ***distribución de probabilidad*** $Pr\{\}$***:*** *una distribución de probabilidad sobre un espacio muestral* Ω *es una función:*

$$\begin{aligned} Pr :& \Omega \to \mathbb{R} \\ & A \to Pr\{A\} \end{aligned}$$

que satisface las siguientes propiedades:

- $Pr\{A\} \geq 0$ Para cualquier evento $A \in \Omega$.
- $Pr\{\Omega\} = 1$.
- Si A y B son eventos disjuntos en Ω, se tiene $Pr\{A + B\} = Pr\{A\} + Pr\{B\}$.

DEFINICIÓN 15. ***variable aleatoria discreta*** X***:*** *una variable aleatoria discreta es una función de un espacio muestral finito* Ω *sobre el conjunto de los números reales. Esta asocia un número real con cada posible resultado de un experimento.*

Para una variable aleatoria X y un número real x, se define el evento $X = x$ como $\{s \in \Omega : X(s) = x\}$. Por tanto,

$$Pr\{X = x\} = \sum_{s \in \Omega : X(s) = x} Pr\{s\} \tag{A.1}$$

DEFINICIÓN 16. ***Esperanza:*** *la esperanza, el valor esperado, o la media de una variable aleatoria discreta* X *se define como,*

$$E[X] = \sum_{x} x \cdot Pr\{X = x\} \tag{A.2}$$

Desigualdad de Jensen

La desigualdad de Jensen establece que para una función convexa f y una variable aleatoria X,

$$E[f(X)] \leq f(E[X]), \tag{A.3}$$

siempre que las esperanzas existan y sean finitas. (Una función f es convexa si para todo x, y y $0 \leq \lambda \leq 1$ se tiene que $f(\lambda x + (1 - \lambda)y) \leq \lambda f(x) + (1 - \lambda) f(y)$.)

Capítulo B

Método de Sustitución para resolver recurrencias

El método de sustitución para resolver recurrencias consiste en dos pasos:

- Suponer la forma de la solución.
- Usar inducción matemática para encontrar las constantes y mostrar que la solución funciona.

Se sustituye la forma de la solución supuesta por la función cuando se aplica la hipótesis inductiva a valores más pequeños; de ahí el nombre de "método de sustitución". Este método es poderoso, pero se debe ser capaz de suponer la forma de la respuesta para poder aplicarlo. Se puede usar el método de sustitución para establecer límites superiores o inferiores en una recurrencia. Como ejemplo, se determinará un límite superior en la recurrencia

$$T(n) = 2T(\lfloor n \rfloor) + n, \tag{B.1}$$

El cuál es el tiempo de ejecución del famoso algoritmo de ordenamiento *MERGE-SORT*, se supondrá que la solución es $T(n) = O(n\ lg\ n)$. El método

de sustitución requiere que se pruebe que $T(n) \leq cn\ lg\ n$ para una elección apropiada de la constante $c > 0$, se empieza asumiendo que esta cota se cumple para todo número positivo $m < n$ en particular $m = \lfloor n/2 \rfloor$, teniendo $T(\lfloor n/2 \rfloor) \leq c\lfloor n/2 \rfloor\ lg(\lfloor n/2 \rfloor)$. Sustituyendo esto en la ecuación se obtiene,

$$\begin{aligned} T(n) &\leq 2(c\lfloor n/2 \rfloor\ lg(\lfloor n/2 \rfloor)) + n \\ &\leq cn\ lg(n/2) + n \\ &= cn\ lg\ n - cn\ lg\ 2 + n \\ &= cn\ lg\ n - cn\ + n \\ &\leq cn\ lg\ n, \end{aligned}$$

donde el último paso se cumple siempre que $c \geq 1$.

La inducción matemática ahora requiere que se muestre que la solución se cumple para las condiciones de contorno. Por lo general, se lo hace mostrando que las condiciones de contorno son adecuadas como casos base para la prueba inductiva.

Para la recurrencia B.1 se debe probar que se puede elegir la constante c lo suficientemente grande para que la cota $T(n) = cn\ lg\ n$ funcione también para las condiciones de contorno. Este requisito a veces puede dar lugar a problemas. Se asume, en aras del argumento, que $T(1) = 1$ es la única condición de frontera de la recurrencia. Entonces, para $n = 1$, la cota $T(n) = cn\ lg\ n$ verifica $T(1) = c1\ lg\ 1 = 0$, lo cual contradice $T(1) = 1$. En consecuencia, el caso base de la prueba inductiva no se cumple.

Se puede superar este obstáculo probando una hipótesis inductiva para una condición límite específica con solo un poco más de esfuerzo. En la recurrencia B.1, por ejemplo, se toma ventaja de la notación asintótica que exige solo probar que $T(n) \leq cn\ lg\ n$ para $n \geq n_0$ donde n_0 es una constante que puede ser elegida. Se mantiene la problemática condición de contorno $T(1) = 1$ pero no será considerada en la prueba inductiva. Se observa primera que para $n > 3$ la recurrencia no depende de directamente de $T(1)$. Por tanto, puede reemplazarse $T(1)$ por $T(2)$ y $T(3)$ como los casos base en la prueba inductiva, poniendo $n_0 = 2$. Note que se hace una distinción del caso base de la recurrencia $(n = 1)$ y el caso base de la

prueba inductiva ($n = 2$ y $n = 3$). Con $T(1) = 1$, de la recurrencia se tiene que $T(2) = 4$ y $T(3) = 5$. Ahora se puede completar la prueba inductiva $T(n) \leq cn\ lg\ n$ para alguna constante $c \geq 1$ tomando c suficientemente grande tal que $T(2) \leq c2\ lg\ 2$ y $T(3) \leq c3\ lg\ 3$. Resulta que cualquier elección de $c \geq 2$ es suficiente para que se cumplan los casos base de $n = 2$ y $n = 3$.

Referencias bibliográficas

[1] Ravindra K Ahuja, Thomas L Magnanti, and James B Orlin. Network flows. 1988.

[2] DS Andrews, Jack Snoeyink, Jim Boritz, T Chan, Graham Denham, Jenny Harrison, and Chong Zhu. Further comparisons of algorithms for geometric intersection problems. In *Proc. 6th Int'l. Symp. on Spatial Data Handling*, 1994.

[3] Jon Louis Bentley and Thomas A Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Transactions on computers*, 28(09):643–647, 1979.

[4] Andrew Butterfield, Gerard Ekembe Ngondi, and Anne Kerr. *A dictionary of computer science*. Oxford University Press, 2016.

[5] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2022.

[6] Rosa Guerequeta and Antonio Vallecillo. *Técnicas de diseño de algoritmos*. 2019.

[7] Donald E Knuth. Structured programming with go to statements. *ACM Computing Surveys (CSUR)*, 6(4):261–301, 1974.

[8] Phillip A Laplante. *Dictionary of computer science, engineering, and technology*. CRC Press, 2017.

[9] Anany Levitin. Design and analysis of algorithms reconsidered. In *Proceedings of the thirty-first SIGCSE technical symposium on Computer science education*, pages 3–3, 2000.

[10] Thomas Ottmann and SIX HW. Eine neue klasse von ausgeglichenen binaerbaeumen. pages 395–400, 1976.

[11] Michael Ian Shamos. *Computational geometry*. Yale University, 1978.

[12] Micha Sharir and Mark H Overmars. A simple output-sensitive algorithm for hidden surface removal. *ACM Transactions on Graphics (TOG)*, 11(1):1–11, 1992.