ESCUELA POLITÉCNICA NACIONAL

FACULTAD DE CIENCIAS

APLICACIONES DE ESTRUCTURAS DE DATOS A PROBLEMAS MATEMÁTICOS. IMPLEMENTACIÓN DEL PROBLEMA DE SELECCIONAR DE UN CONJUNTO DE PUNTOS DADO, LOS SUBCONJUNTOS DE CUATRO PUNTOS COLINEALES

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

JOSHUA ISRAEL CHULDE ORTIZ

joshua.chulde@epn.edu.ec

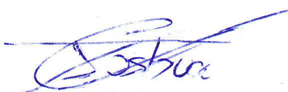
DIRECTOR: MARIA FERNANDA SALAZAR MONTENEGRO

fernanda.salazar@epn.edu.ec

DMQ, MARZO 2023

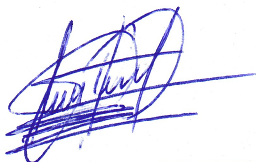
CERTIFICACIONES

Yo, JOSHUA ISRAEL CHULDE ORTIZ, 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.



JOSHUA ISRAEL CHULDE ORTIZ

Certifico que el presente trabajo de integración curricular fue desarrollado por JOSHUA ISRAEL CHULDE ORTIZ, 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.

JOSHUA ISRAEL CHULDE ORTIZ

MARIA FERNANDA SALAZAR MONTENEGRO

DEDICATORIA

Con todo mi corazón a mis padres Patricia Ortiz Calle y Byron Chulde Chicaiza quienes siempre creyeron en mí, de quienes tuve amor y apoyo incondicionales y forjaron dentro mi alma unos valiosos valores, que hoy me hacen ser quien soy.

A todos mis compañeros, amigos, familiares y conocidos que me motivaron y me animaron a culminar mis estudios.

Finalmente, a todos mis profesores, quienes en estos años me hicieron crecer como profesional. Y especialmente a la Dra. Maria Fernanda Salazar por su gran apoyo y guía en la realización del presente trabajo de titulación.

RESUMEN

La geometría computacional a lo largo de los años ha planteado varios problemas de interés, a partir de los cuales se ha desplegado una gran cantidad de herramientas que ayudan a la resolución de estos precisos problemas, y a su paso, proporcionan nuevas herramientas para abordar de manera más práctica y eficiente tanto nuevos como antiguos problemas. Una de estas cuestiones es el problema de puntos colineales. El problema de puntos colineales propone encontrar y reportar dentro de un conjunto de puntos dado, todos los posibles subconjuntos de puntos colineales contenidos en el conjunto.

El presente trabajo aborda dos posibles soluciones a este problema y plantea realizar una comparación en la eficiencia del rendimiento de las mismas. El primero de los algoritmos escogidos, denominado de fuerza bruta, utiliza el principio de analizar todas las posibles combinaciones de puntos para determinar colinealidad, siendo éste, como su nombre lo indica, el algoritmo que no emplea herramientas eficientes para llegar a su solución. El segundo algoritmo, denominado inteligente, aprovechará herramientas de estructuras de datos como los árboles binarios y algoritmos eficientes de ordenación de datos para poder llegar a la misma solución que el primer algoritmo, con la diferencia de manejar de mejor manera el arreglo inicial y resolver el problema en menor tiempo.

Los algoritmos serán implementados en el lenguaje de programación C++ siendo puestos a prueba con instancias de distintos tamaños generadas aleatoriamente. Al finalizar, se comparará la eficiencia de los algoritmos.

Palabras clave: puntos colineales, heapsort, estructuras de datos, algoritmo

ABSTRACT

Among years, computational geometry has proposed various problems of interest, from which a large number of tools have been deployed. Those tools have helped to solve these particular problems, also, providing new tools to address more practical and efficient for both new and old problems. One of these questions is the problem of collinear points. The collinear points problem proposes that, given a set of points, find and report all possible subsets of collinear points.

This document addresses two possible solutions to this problem and proposes a comparison of their performance. The first algorithm, labeled as brute force, analyses all possible combinations of points to determine collinearity. The algorithm does not use efficient tools to reach its solution. While second algorithm, addressed as intelligent, will take advantage of data structure tools such as binary trees and efficient data ordering algorithms in order to reach the same solution, but handling the initial set in a better way.

The algorithms will be implemented in the programming language C++. They will be tested with randomly generated instances of different sizes. At the end, the efficiency of the algorithms will be compared.

Keywords: collinear points, heapsort, data structures, algorithm

Índice general

1. Descripción del componente desarrollado **1**

1.1. Objetivo general 2

1.2. Objetivos específicos 2

1.3. Alcance 2

1.4. Marco teórico 3

1.4.1. Definiciones 3

1.4.2. Algoritmo de la burbuja 8

1.4.3. Algoritmo *Quicksort* 10

1.4.4. Algoritmo *Heapsort* 12

1.4.5. Heap 12

2. Metodología **16**

2.1. Introducción 16

2.2. El problema de puntos colineales 16

2.3. Algoritmo de *fuerza bruta* 17

2.3.1. Análisis de complejidad del algoritmo de fuerza bruta 18

2.4. Algoritmo con criterio de ordenamiento 19

2.4.1. Algoritmo primer criterio de comparación 19

2.4.2. Algoritmo generar pendientes 20

2.4.3. Algoritmo Heapsort Modificado 21

2.4.4. Algoritmo Inteligente 23
2.4.5. Análisis de complejidad del algoritmo inteligente . . . 24
2.4.6. Motivación de Heapsort 25
2.4.7. Implementación 26

3. Resultados computacionales, conclusiones y recomendaciones 27
3.1. Descripción de instancias 27
3.1.1. Resultados 28
3.2. Conclusiones y recomendaciones 34

Bibliografía 36

Índice de figuras

1.1. Árbol binario 13

3.1. Gráfica de línea de comparación de tiempos de ejecución promedio entre ambos algoritmos 31

3.2. Gráfica de los puntos empleados para la primera instancia (50 puntos) 32

3.3. Gráfica de la solución por algoritmo inteligente para la primera instancia (50 puntos) 32

Capítulo 1

Descripción del componente desarrollado

El presente trabajo de integración curricular aborda el siguiente problema: dado un conjunto de puntos en el espacio $\mathbb{R}^2$, se desea encontrar todos los subconjuntos de cuatro puntos, tales que sean colineales entre sí.

Para la resolución de este problema se implementarán dos algoritmos en el lenguaje de programación C++, mismos que abordan el problema de manera distinta. El primer algoritmo resolverá el problema de la forma más 'simple' posible, esto es, analizando cada combinación de cuatro puntos posibles del conjunto para determinar si son colineales o no. Este algoritmo resulta relativamente sencillo de implementar, sin embargo, su costo viene dado en términos computacionales, ya que, si bien el análisis individual de un conjunto de puntos puede resultar rápido, este método puede crear cientos, miles e incluso millones de posibles combinaciones dependiendo del número de puntos, lo que resulta en un algoritmo que puede tomar un tiempo inmensurable para devolver una solución. Es a este algoritmo al que denominaremos como el de 'fuerza bruta'.

El segundo algoritmo a implementar, a diferencia del primero, incluirá métodos de búsqueda más eficientes que permitan encontrar la colinealidad sin recurrir a análisis de conjuntos individuales. Se planea mapear cada uno de los puntos pertenecientes al conjunto, y establecer 2 criterios de ordenamiento, uno para los puntos en el espacio y otro para las

pendientes que se generen entre ellos, de forma que permitan identificar aquellos que pertenecen a la misma recta y con ello su colinealidad.

Se realizará pruebas computacionales para cada uno de los algoritmos y una comparación entre los resultados obtenidos para verificar la eficiencia de los mismos.

1.1. Objetivo general

Desarrollar una solución al problema de encontrar puntos colineales en un conjunto de puntos cualquiera mediante la implementación de dos algoritmos, cuyos resultados serán comparados para identificar la eficiencia de los algoritmos.

1.2. Objetivos específicos

1. Implementar un algoritmo que permita resolver el problema de puntos colineales por medio de 'fuerza bruta', es decir, identificando todas las posibles combinaciones de puntos para determinar la colinealidad.

2. Implementar un algoritmo inteligente que use métodos de búsqueda ayudados por estructuras de datos dinámicas.

3. Encontrar el orden de complejidad de los dos algoritmos implementados y compararlos.

4. Realizar pruebas computacionales usando instancias aleatorias en ambos algoritmos para realizar una comparación de resultados.

1.3. Alcance

El presente trabajo de integración curricular estudia el problema de hallar los puntos colineales en un conjunto dado de puntos, brindando una solución más conveniente que considere las limitaciones computacionales, así como el tamaño de la entrada del problema.

La solución por fuerza bruta tiene una sencilla implementación; sin embargo, debido a su gran orden computacional, la solución a este problema se dará encontrando conjuntos de 4 elementos de puntos colineales, puesto que la implementación con conjuntos con un número de elementos mayor precisamente ha mostrado la limitación computacional que existe en este algoritmo para encontrar soluciones de grado mayor.

El segundo algoritmo, que emplea técnicas inteligentes, puede encontrar conjuntos de puntos colineales con 4 o más elementos. Ambos algoritmos serán explicados teóricamente y se implementarán en el lenguaje C++.

Las instancias con las que se probarán los algoritmos serán generadas aleatoriamente, manteniendo cierta consistencia para que los resultados mostrados en los diferentes algoritmos puedan ser sujetos a comparación.

Además, se comprobará la eficiencia de los algoritmos usando instancias de diferentes tamaños. Con ello se podrá dar un panorama claro sobre la ventaja del algoritmo inteligente en instancias de diferentes tamaños sobre el algoritmo de 'fuerza bruta'.

1.4. Marco teórico

1.4.1. Definiciones

Estructura de datos

Las estructuras de datos son utilizadas en la implementación de algoritmos con el objetivo de manejar de forma eficiente los recursos tecnológicos a disposición [1]. Se refieren a una forma concreta de organizar un conjunto de datos.

Nodo

En [2], un nodo es un elemento particular que conforma la estructura de datos. Es la unidad más pequeña de una estructura.

Árbol

Es un conjunto de nodos con la siguiente estructura: un único nodo es llamado raíz, los nodos restantes forman $n \geq 0$ conjuntos disjuntos, los cuales son en sí mismo un árbol, que se denominan subárboles de la raíz [3].

Árbol binario

Es un árbol especial en el cual cada nodo tiene máximo dos subárboles (izquierdo y derecho) [3].

Algoritmo

Un algoritmo es una serie de instrucciones ordenadas cuyo fin es el de resolver un problema dado. Según [4] un algoritmo necesita de una entrada que son los 'datos del problema' que al ser manipulados según las instrucciones del algoritmo aportan con una 'solución'. Si es visto de esta forma, el algoritmo es una correspondencia de la forma

$$f : D \longrightarrow S$$

donde f es el algoritmo que transforma los datos D en las soluciones S. Es decir, resolver un problema es evaluar a f.

En [5] se explica que cualquier algoritmo debe contener un número finito de pasos para poder resolver el problema.

Los algoritmos enfocados en las ciencias computacionales que usan lenguajes de programación usan 'pseudocódigos' para expresar algoritmos complejos. Un 'pseudocódigo' es una técnica para describir un programa computacional usando un lenguaje general y palabras claves de programación más que una sintaxis específica de algún lenguaje de programación en particular, con el fin de que sea fácil de entender y pueda ser transformado a cualquier lenguaje [6], [7].

La entrada D del algoritmo es un parámetro importante respecto a la eficiencia que presentará el algoritmo en la resolución del problema. Generalmente, los algoritmos se desarrollan primero teniendo en mente

la resolución del problema, a partir de allí, se comprueba su eficiencia con los datos D. Un algoritmo se evalúa no solamente respecto a si resuelve el problema en cuestión o no, sino también en cuántos recursos usa para solucionarlo [8]. Es allí donde el tamaño de los datos D puede dejar totalmente inservible a un algoritmo que funciona aparentemente bien con instancias pequeñas. Nace de allí la necesidad de buscar técnicas que permitan a los algoritmos usar de forma más eficiente los recursos de los que se disponen para obtener la solución.

Relación de orden

Según [9] dado un conjunto $\boldsymbol{X}$ una relación de orden se puede definir de la siguiente forma.

Una relación de orden en un conjunto X es una relación $\sigma = \{X, C\}$ que tiene las siguientes propiedades.
a) Si $x \in X$ entonces $(x, x) \in C$.
b) Si $(x, y) \in C, (y, x) \in C$ entonces $x = y$.
c) Si $(x, y) \in C, (y, z) \in C$ entonces $(x, z) \in C$.

Se utiliza el símbolo $\leqslant$ en la relación de orden, y se escribe $x \leqslant y$ en vez de $(x, y) \in \boldsymbol{C}$. Al usar esta notación, las definiciones anteriores se convierten en:
a) Si $x \in X$ entonces $x \leqslant x$ (reflexiva).
b) Si $x \leqslant y$ e $y \leqslant x$ entonces $y = x$ (antisimétrica).
c) Si $x \leqslant y$ e $y \leqslant z$ entonces $x \leqslant z$ (transitiva).
A veces se escribe $y \geqslant x$ en vez de $x \leqslant y$.

Por tanto, un conjunto donde se ha definido una relación de orden se lo denomina como un conjunto ordenado.

Complejidad de un algoritmo

En 1844 G. Lamé es el primero en mostrar la complejidad teórica de un algoritmo al estimar cuántas divisiones son necesarias para poder ejecutar el algoritmo de Euclides en $\mathbb{Z}$. Sin embargo, la relevancia de la complejidad computacional se atribuye a E. Galois [4]; él propone un método

para decidir si una ecuación de grado 5 o mayor puede ser resoluble por radicales, sin embargo, insiste en que no puede realizar los cálculos con lápiz y papel. Galois descubrió un algoritmo de complejidad exponencial en tiempo de ejecución, que incluso hoy en día sería impracticable para ordenadores cuando el grado es superior a 100.

Con este antecedente, es entendible la necesidad de un algoritmo que cumpla con su objetivo de forma eficiente. En este caso, la eficiencia hace referencia a cuántos recursos computacionales se usa en la ejecución del algoritmo, particularmente el tiempo y memoria del ordenador [10].

En el presente trabajo, se considerará la complejidad en términos temporales para medir la eficiencia de un algoritmo contra otro, que como se intuye, es la medida de tiempo que tardará un ordenador en ejecutar un algoritmo dado ciertos datos D. La complejidad temporal usa el número de operaciones básicas que debe ejecutar el algoritmo usando la entrada D para llegar a la solución.

Operaciones básicas

Hacen referencia a las operaciones básicas que realiza el ordenar dentro el algoritmo. En [11] se considera como operación básica a:

- Operaciones elementales aritméticas (suma, resta, división y multiplicación)
- Comparaciones lógicas
- Asignación de variables
- Saltos (llamado funciones y procedimientos externos)
- Acceso a estructuras indexadas (vectores y matrices)

Para determinar medidas estándar que determinen la complejidad, [12] introduce las siguientes notaciones.

Notación	Orden
$\mathcal{O}(1)$	Constante
$\mathcal{O}(\log(n))$	Logarítmica
$\mathcal{O}(n)$	Lineal
$\mathcal{O}(n \cdot log(n))$	Log-lineal
$\mathcal{O}(n^2)$	Cuadrada
$\mathcal{O}(n^c)$	Polinomial
$\mathcal{O}(c^n)$	exponencial
$\mathcal{O}(n!)$	factorial

Cuadro 1.1: Funciones y su complejidad en orden ascendente según su tiempo de complejidad

Notación Big $\mathcal{O}$

Sean dos funciones $f(x)$ y $g(x)$ con x la entrada de la función. Decimos que la función $f(x)$ es $\mathcal{O}(g(x))$ si existen constantes C y k tales que:

$$|f(x)| \leq C \cdot |g(x)| \quad \forall x > k$$

Notación Big Ω

Sean dos funciones $f(x)$ y $g(x)$ con x la entrada de la función. Decimos que la función $f(x)$ es $\Omega(g(x))$ si existen constantes $C > 0$ y k tales que:

$$|f(x)| \geq C \cdot |g(x)| \quad \forall x > k$$

Notación Big Θ

Sean dos funciones $f(x)$ y $g(x)$ con x la entrada de la función. Decimos que la función $f(x)$ es $\Theta(g(x))$ si $f(x)$ es $\Omega(g(x))$ y $\mathcal{O}(g(x))$

La tabla 1.1 indica el número de operaciones que debe realizar el algoritmo para ser ejecutado completamente dependiendo del tamaño de la entrada n. Por ejemplo, un algoritmo de orden constante, $\mathcal{O}(1)$, indica que el tiempo que le lleva ejecutarse completamente es el mismo para cualquier tamaño de n. Por otro lado, algoritmos $\log(n)$ o $\mathcal{O}(n^c)$ aumentan su complejidad de acuerdo al tamaño de n sin embargo la complejidad del algoritmo logarítmico aumenta más lentamente que un algoritmo polinomial con base en el tamaño de n. Supongamos un valor de

$n = 10$, una función log-lineal da un tiempo de ejecución de $\mathcal{O}(nlog(n)) = 10$ mientras que una función de orden factorial da un tiempo de ejecución $\mathcal{O}(n!) = 3628800$. Por ello, para problemas con tamaños de n grandes es preferible encontrar y ejecutar algoritmos logarítmicos que trabajan relativamente bien sin afectar el rendimiento.

Algoritmo de *fuerza bruta*

Se reconocerá como algoritmo de 'fuerza bruta' en el presente trabajo de integración curricular a aquella función que trata de resolver un problema en particular, usando como método la comprobación de cada una de las posibles combinaciones de elementos o posibilidades que pueden cumplir el resultado sin tratar de introducir ninguna estructura de datos para optimizar el almacenamiento, búsqueda, recuperación o modificación de los datos [13], lo cual suele llevar generalmente a algoritmos ineficientes en ejecución con un orden polinomial como mínimo.

Para los algoritmos que se implementarán posteriormente, se usarán algunas actividades básicas en estructuras de datos, como son la búsqueda, modificación de datos, recuperación de datos, etc. En particular, la ordenación de datos será de relevancia en este contexto.

La clasificación u ordenación de datos parte de un concepto simple. Supóngase que se tiene una lista con n observaciones de cierto tipo de datos y la misión es organizarlos de manera alfabética o numérica. A pesar de lo sencillo que suena, en teoría esta es una actividad que ha conducido al desarrollo de múltiples algoritmos, que al igual que lo que se plantea en el presente trabajo, algunos resuelven la cuestión mediante 'fuerza bruta' mientras que otros aprovechan los recursos a disposición de mejor manera. A continuación se presentan dos de estos algoritmos.

1.4.2. Algoritmo de la burbuja

Este es un algoritmo popular pero bastante ineficiente de ordenación. La forma en la que funciona, como lo explica [14] es la siguiente: Se toma los dos primeros elementos de la lista a ordenar y los compara. Si tienen el orden correcto, los deja tal y como están, si no, entonces los intercam-

bia. Luego toma el segundo y tercer elemento de la lista y los compara, si tienen el orden correcto, los deja tal y como están, si no, entonces los intercambia. El algoritmo continúa de esta forma con el tercer y cuarto elemento, luego el cuarto y quinto y así sucesivamente. Cuando se haya terminado de organizar todos los elementos de la lista, el algoritmo realiza todo el proceso nuevamente hasta que no existan más pares en el orden incorrecto. En pocas palabras, el algoritmo trabaja repetidamente intercambiando elementos adyacentes que están en el orden incorrecto.

El pseudocódigo de este algoritmo puede ser encontrado en [8] de la siguiente forma:

Algoritmo burbuja($Lista\ A$)
for $i = 1$ to $Lista\ A$.longitud -1
  for $j = Lista\ A$.longitud to $i + 1$
    Si $A[j] < A[j-1]$
      Intercambiar $A[j]$ con $A[j-1]$

Cuánto tiempo le tomará al algoritmo ordenar las observaciones depende de los datos de entrada; sin embargo, suponiendo el peor de los casos para este algoritmo, que es cuando los datos se encuentran ordenados de forma descendente (o ascendente en caso de que se requiera ordenar de forma descendente) se puede mostrar que se necesita máximo $n - 1$ revisiones por toda la lista, lo que en tiempo de ejecución se traduce a un tiempo de $\mathcal{O}(n^2)$ [8], tanto en el peor de los casos como en los casos promedio. Este es uno de los algoritmos de ordenación más lentos, por lo que en la práctica nunca debería ser usado si se tienen listas con valores grandes de n.

Existen mejores algoritmos en la práctica, que pueden ser ejecutados en un tiempo $\mathcal{O}(n \cdot \log(n))$ como el *Merge Sort* o el *Quicksort* del cual se profundizará a continuación.

1.4.3. Algoritmo *Quicksort*

El algoritmo *Quicksort* es a menudo la mejor opción práctica para la ordenación de datos, puesto que, a pesar de que su tiempo de ejecución en los peores casos es de $\Theta(n^2)$, el tiempo de ejecución promedio es bastante bueno, $\Theta(n \cdot \log(n))$ [8].

Este método utiliza el paradigma de 'divide y vencerás', que es dividir el problema general en varios subproblemas que son similares al original pero de menor tamaño, resolverlos recursivamente y luego combinarlos en una solución para el problema original [15]. Con un arreglo tipo $A[p..r]$ las tres partes de este paradigma serían:

Dividir: Se particiona el arreglo inicial en dos subarreglos: $A[p..q-1]$ y $A[q+1..r]$, tal que

- Todo elemento de $A[p..q-1]$ es menor o igual que $A[q]$

- Todo elemento de $A[q+1..r]$ es mayor o igual que $A[q]$

Vencer: Se ordenan los siguientes dos subarreglos $A[p..q-1]$ y $A[q+1..r]$ de forma recursiva usando el paso de *dividir*.

Combinar: Al finalizar, los subarreglos ya estarán ordenados dentro de ellos y entre ellos, por lo que $A[p..r]$ estará ordenado.

Ahora, lo que aún no se ha mencionado es cómo calcular el valor del índice q. Esto se realiza por medio del procedimiento de *Partición*, que reordena el arreglo $A[p..r]$. La forma en la que lo hace se incluye en [8] con el siguiente pseudocódigo:

```
PARTICION(A, p, r)
x = A[r]
i = p − 1
for j = p to r − 1
    if A[j] ≤ x
        i = i + 1
        Intercambiar A[i] con A[j]
Intercambiar A[i + 1] con A[r]
Return i + 1
```

donde el valor $i+1$ será el primer valor de q que se usará. Tenemos así que el algoritmo $Quicksort$ puede escribirse de la siguiente forma:

```
QUICKSORT (A, p, r)
if p < r
    q =PARTICION(A, p, r)
    QUICKSORT(A, p, q − 1)
    QUICKSORT(A, q + 1, r)
```

Para ordenar cualquier arreglo A se deberá llamar a $QUICKSORT(A, 1, n)$ donde n es la longitud del arreglo A.

Los dos algoritmos presentados muestran la diferencia en eficiencia

que causan las diferentes aproximaciones al mismo problema. Podemos clasificar algoritmos como mejores o peores basados en diferentes criterios, siendo el tiempo de ejecución claramente uno de los principales, sin embargo, para usos prácticos, debe estudiarse el problema de antemano y buscar entre las diferentes opciones cuál es la que se adapta mejor al problema en cuestión. En este sentido, a pesar de que el algoritmo $Quicksort$ sea ampliamente usado en la práctica, para el problema concreto presentado en este documento, se introducirá el algoritmo $Heapsort$, el cual se adapta mejor, como se verá más adelante, a las necesidades del problema, a pesar de compartir un tiempo de ejecución similar con $Quicksort$.

1.4.4. Algoritmo *Heapsort*

El algoritmo *heapsort* tiene un tiempo de ejecución medio de $\mathcal{O}(n \log n)$ [12]. El algoritmo usa una estructura de datos llamada *heap* para manejar la información, el cual usa de manera eficiente el arreglo. A continuación se dará un desglose más a profundidad de cómo funciona este algoritmo basado en la explicación brindada por [8].

1.4.5. Heap

Llamamos $heap$ a una estructura de datos similar a un árbol binario. Cada nodo del árbol se identifica con un elemento del arreglo inicial. El árbol debe ser llenado al completamente, exceptuando el nivel más bajo, el cual deberá ser llenado en la medida de lo posible. Esta estructura cuenta con dos atributos: la altura del árbol, el cual es el número de niveles del mismo, y el tamaño del árbol, que es el número de elementos o nodos en la estructura. De aquí en adelante llamaremos A a la estructura $heap$, e i al índice de un nodo en A, en este sentido, $A[1]$ será la raíz del árbol. La figura 1.1 muestra un arreglo y el respectivo árbol formado a partir de él.

Dado el índice i de cualquiera de los nodos, es bastante fácil encontrar los índices de padre(PARENT) e hijos derecho e izquierdo(LEFT,RIGHT) del mismo:

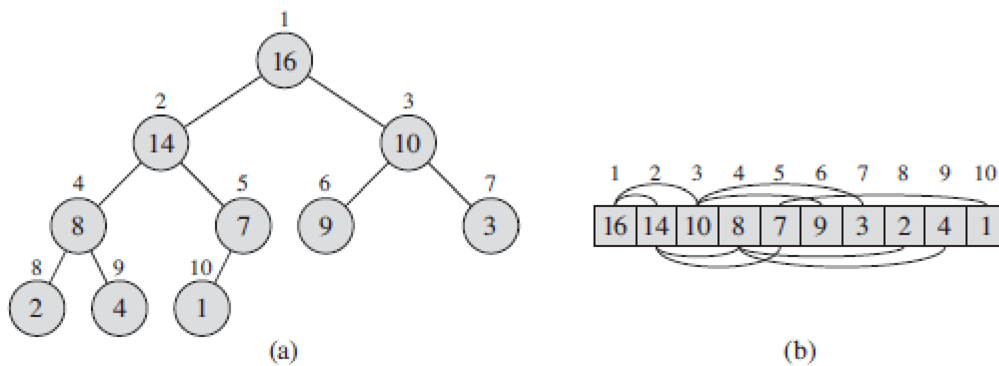


Figura 1.1: Árbol binario
(a) muestra una estructura *heap* como un árbol binario, mientras que (b) es el arreglo con el cual se forma el árbol

PARENT(i)

return $\lfloor i/2 \rfloor$

LEFT(i)

return $\lfloor 2i \rfloor$

RIGHT(i)

return $\lfloor 2i + 1 \rfloor$

El árbol binario $max - heaps$ es aquel que satisface la siguiente propiedad para cada nodo i:

A[PARENT(i)] $\geq A[i]$

Siguiendo esta lógica, el elemento de mayor valor en este tipo de árbol se encuentra en su raíz, y los subárboles de cualquier nodo no tendrán valores mayores a él mismo.

Para mantener la propiedad indicada anteriormente de un árbol $max - heap$, se tiene el procedimiento siguiente:

MAX-HEAPIFY (A, i)

l =LEFT(i)
r =RIGHT(i)
if $l \leq A$.tamaño **and** $A[l] > A[i]$
$largest = l$
else $largest = r$
if $r \geq A$.tamaño **and** $A[r] > A[largest]$
$largest = r$
if $largest \neq i$
intercambiar $A[i]$ con $A[largest]$
MAX-HEAPIFY$(A, largest)$

Este algoritmo toma como entrada un arreglo A y un índice i del arreglo. Para ello, se asume que los árboles binarios que salen del hijo izquierdo y derecho de i son árboles $max - heap$. Esto debido a que si no se cumple la propiedad en el índice i, el algoritmo intercambia los valores con el correspondiente y continúa de forma descendiente realizando el mismo proceso. En [8] se indica cómo el valor del tiempo de ejecución de este algoritmo es $\mathcal{O}(\log n)$.

El algoritmo anterior es usado sobre árboles binarios, por lo que se necesita de un algoritmo que sea capaz de convertir un arreglo cualquiera en un árbol $max - heap$, A. El siguiente algoritmo toma como entrada un arreglo A cualquiera y construye un árbol binario del tipo $max - heap$:

BUILD-MAX-HEAP(A)
A.tamaño= A.altura
for $i = \lfloor A$.altura$/2 \rfloor$ **downto** 1
MAX-HEAPIFY(A, i)

El tiempo de ejecución del algoritmo se determina de la siguiente manera: Cada llamada a MAX-HEAPIFY tiene un costo de $\mathcal{O}(\log n)$ y el algoritmo en sí hace $\mathcal{O}(n)$ de esas llamadas, por lo que el tiempo total de todo el algoritmo será $\mathcal{O}(n \log n)$, sin embargo, en muchas ocasiones el algoritmo puede hallar un árbol A de un arreglo desordenado en un tiempo lineal $\mathcal{O}(n)$.

Finalmente, el algoritmo *heapsort* empieza utilizando el algoritmo BUILD-MAX-HEAP para construir un árbol A con las mencionadas características:

HEAPSORT(A)
for $i = A.$altura **downto** 2
  intercambiar $A[1]$ con $A[i]$
  $A.$tamaño= $A.$tamaño-1
  MAX-HEAPIFY$(A, 1)$

El algoritmo tiene un tiempo de ejecución de $\mathcal{O}(n \log n)$ debido a que llamar al algoritmo BUILD-MAX-HEAP tiene un tiempo de $\mathcal{O}(n)$ y cada una de las $n-1$ llamadas a MAX-HEAPIFY toma un tiempo $\mathcal{O}(\log n)$ [8].

Capítulo 2

Metodología

2.1. Introducción

El presente trabajo aborda el problema de encontrar 4 o más puntos colineales de un conjunto de puntos en $\mathbb{R}^2$. El problema se resuelve mediante dos algoritmos que son analizados usando diversas instancias que permitan verificar y comparar entre ellos la eficiencia de los algoritmos para obtener resultados con diferentes tamaños de entrada. En primer lugar, se presenta una descripción formal del problema a resolver. A continuación, se detalla una descripción del algoritmo de fuerza bruta y del algoritmo con criterio de ordenamiento. En cada método de resolución se presenta la idea de forma breve sobre el funcionamiento de cada uno, para posteriormente presentar en pseudocódigo los algoritmos empleados y finalmente se realiza el análisis de complejidad.

2.2. El problema de puntos colineales

Sea un conjunto de n puntos en $\mathbb{R}^2$ definido de la siguiente manera $D := \{(x_i, y_i) : x_i, y_i \in \mathbb{R}, i = 1, 2, ..., n\}$.

Se define una recta B en el espacio $\mathbb{R}^2$ con geometría euclidiana al conjunto de puntos tales que $B := \{(x, y) \in \mathbb{R}^2 : y = mx + b$ con $m, b \in \mathbb{R}\}$.

Sea A un conjunto de puntos en $\mathbb{R}^2$. Este conjunto se denominará un

conjunto de puntos colineales siempre que $(x, y) \in B, \forall(x, y) \in A$ con B una recta cualquiera.

Dado un conjunto de puntos D, el problema de puntos colineales consiste en encontrar todos los conjuntos de puntos colineales A posibles siempre que su cardinalidad sea $|D| = 4$.

2.3. Algoritmo de *fuerza bruta*

El algoritmo de fuerza bruta propuesto usa cada una de las combinaciones posibles de 4 puntos del total del conjunto. Cada conjunto se somete a un análisis para determinar si los puntos son colineales. Si el conjunto es colineal, se guarda, si no, el algoritmo procede a analizar el siguiente hasta haber estudiado todos los conjuntos posibles. A continuación se desglosa el procedimiento realizado.

Este primer pseudocódigo recibe como entrada dos puntos cualesquiera y determina la pendiente entre ellos.

```
PENDIENTE (punto A(a,b), punto B(x,y))
 if a = x
     if b = y
         return -inf
     return inf
 else if b = y
     return 0
 else
     pendiente = (y − b)/(x − a)
 return pendiente
```

El siguiente pseudocódigo es el que analiza los posibles conjuntos de puntos colineales y devuelve todos los conjuntos que son efectivamente colineales.

```
COLINEAL FUERZA BRUTA (D, |D|, vector − conjuntos)
 for i = 1 to |D| − 3 do
     for j = i + 1 to |D| − 2 do
         u =PENDIENTE(punto j, punto i)
         for k = j + 1 to |D| − 1 do
             t =PENDIENTE(punto n, punto j)
             for l = k + 1 to |D| do
                 colineales= segmento[0]
                 s =PENDIENTE(punto l, punto k)
                 if s = t and t = u
                     colineales= (punto i, punto j, punto k, punto l)
             if |colineales| > 3
                 Agregar colineales a vector − conjuntos
                 Eliminar colineales
 return vector − conjuntos
```

2.3.1. Análisis de complejidad del algoritmo de fuerza bruta

El algoritmo de fuerza bruta presentado trabaja usando lazos anidados. El primer bucle fija, en primera instancia, el primer punto del conjunto. El segundo bucle fija el siguiente punto, el tercero al siguiente. Es en el último bucle que se iterará por el resto de puntos. Cada iteración sobre el último lazo calcula la pendiente entre los diferentes puntos y se comparan entre ellas. Si las pendientes son iguales, el algoritmo determina que son colineales y almacena el conjunto, caso contrario, continúa con la siguiente operación hasta terminar de iterar sobre el último lazo y continuar el proceso en el resto de lazos.

Para determinar el número de operaciones empleadas por el Algoritmo 2, se debe notar que existen n puntos (xi, yi) con i = 1, 2, ..., n. Así, para cada punto del conjunto se fijan otros tres puntos más con los que compararlo.

Vemos que los 4 bucles anidados hacen que el algoritmo tenga un orden $\mathcal{O}(n^4)$ con n el valor de la cardinalidad del conjunto D, es decir, $|D|$.

El algoritmo que se encarga de obtener la pendiente tiene un orden $\mathcal{O}(1)$, por lo que todo el algoritmo se mantiene con un orden $\mathcal{O}(n^4)$.

$$\begin{aligned}
\sum_{i=1}^{N-3}\sum_{j=i+1}^{N-2}\sum_{k=j+1}^{N-1}\sum_{l=k+1}^{N} l &= (N-4)(N-3)(N-2)(N-1) - \sum_{i=1}^{N-4}\sum_{j=i+1}^{N-3}\sum_{k=j+1}^{N-2} k \\
&= N^4 - 10N^3 + 35N^2 - 50N + 24 - \sum_{i=1}^{N-4}\sum_{j=i+1}^{N-3}\sum_{k=j+1}^{N-2} k \\
&\leq= N^4 - 10N^3 + 35N^2 - 50N + 24 + PN^4 \\
&\leq= N^4 + AN^3 + BN^2 + CN + D + PN^4 \\
&\leq= N^4 + A'N^4 + B'N^4 + C'N^4 + D'N^4 + PN^4 \\
&\leq= EN^4
\end{aligned}$$

Donde las constantes $A, B, C, D, A', B', C', D', P$ son valores que acotan superiormente los términos del polinomio y donde E representa la suma de A', B', C', D'. Por lo que existe una constante que acompaña un término de orden cuarto y acota a la suma de las operaciones. Por tanto el orden del algoritmo es de $\mathcal{O}(n^4)$.

2.4. Algoritmo con criterio de ordenamiento

El problema surge al tratar de encontrar un algoritmo que sea sensible respecto a la salida, según las ciencias de la computación, esto es un algoritmo en el que el tiempo de ejecución depende del tamaño de la salida del mismo y no necesariamente de la entrada.[16]

Para resolver este problema se construirá una estrategia que permita reducir las comparaciones entre puntos que no sean necesarias, es decir, aquellas en las que la combinación de punto y pendiente correspondiente al punto y la pendiente analizados, no tengan que volver a ser guardados.

2.4.1. Algoritmo primer criterio de comparación

En este algoritmo se tratará de establecer un primer criterio de ordenamiento, con el fin de poder clasificar los puntos entre sí. Sin embargo,

estudiar si el punto es menor, igual o mayor que otro es una tarea complicada, ya que, debido a las características de $\mathbb{R}^2$ no existe un orden natural que se pueda emplear. Por ello se propone un criterio donde el orden se da en dos partes; dados dos puntos $a, b \in \mathbb{R}^2$, en un primer momento, y como criterio primario, se procederá a comparar la segunda componente de cada punto con el orden estándar propio del espacio $\mathbb{R}$. Esto establecerá el orden existente entre ambas componentes y los puntos heredarán dicha asignación. En caso de ser iguales, se procede a evaluar la primera componente, de igual forma que se hizo previamente, de forma que se consiga un orden completo entre puntos. Si ambas componentes son iguales, ambos puntos estarán en el valor de igualdad.

COMPARAR$(Punto\, A(x, y), Punto\, B(a, b))$
if $y < b$
    retornar -1
if $y > b$
    retornar 1
if $x < a$
    retornar -1
if $x > a$
    retornar 1
else
    retornar 0

2.4.2. Algoritmo generar pendientes

Paralelamente a lo visto anteriormente se desarrolla un algoritmo que reciba un punto y un conjunto de puntos, de forma que calcule y almacene las pendientes creadas por interacción de ambos.

CREAR PENDIENTES$(Punto\, origen, Conjunto\, de\, puntos\, P)$
$pendientes =$null
for $i \in P$
    $pendientes[i] =$pendiente entre origen y el $i-$ésimo punto

return $pendientes$

En la implementación de este algoritmo se hace uso del algoritmo PENDIENTE, como auxiliar para encontrar y asignar el valor correspondiente a cada punto.

2.4.3. Algoritmo Heapsort Modificado

En este tramo la idea es encontrar una forma de determinar si existe una combinación de punto y pendiente que esté siendo analizada y guardada en más de una ocasión, por tanto, es necesario comprender qué implica este concepto. Por definición del problema, para que dados n puntos estos sean colineales, deben caer dentro de la misma recta, es decir, compartir la misma pendiente.

Es aquí que se puede usar la definición de recta a través de un punto y una pendiente para mostrar que es suficiente conocer un punto y un valor para la pendiente a fin determinar la recta buscada.

El desafío consiste en hallar un segundo tipo de ordenamiento, en el que la pendiente juegue un papel fundamental. Con los puntos ya ordenados, será suficiente con fijar un punto, desde el cual se harán los cálculos para establecer las pendientes al resto de puntos y consecuentemente obtener el orden con base en el cual aplicar el criterio adecuado.

Para esto se implementó la siguiente variante del algoritmo Heapsort:

Heapify Modificado Modificado$(Conjunto\, P\, de\, N\, puntos\, ,\, Conjunto\, M\, de\, N\, pendientes)$

$ml = i$
$izq = 2 * i + 1$
$der = 2 * i + 2$

if $(ml < N \quad \& \quad M[izq] > M[ml]) or (ml < N \quad \& \quad M[izq] = M[ml]) \quad \& \quad comparar(A[izq], A[ml]) > 0)$
$ml = izq$

if $(ml < N \;\; \& \;\; M[der] > M[ml]) or (ml < N \;\; \& \;\; M[der] = M[ml]) \;\; \& \;\; comparar(A[der], A[ml]) > 0)$
$ml = der$

if $ml! = i$
Intercambiar $P[i]$ con $P[0]$
Intercambiar $M[i]$ con $M[0]$

Heapify Modificado(P , M , $n\,ml$)

Heapsort Modificado($Conjunto\,P\,de\,N\,puntos$, $Conjunto\,M\,de\,N\,pendientes$)

for i in N /2 -1; i>=0; i- -
Heapify Modificado(P , M , $N\,i$)

for i in N -1; i>=0; i- -
Intercambiar $P[i]$ con $P[0]$
Intercambiar $M[i]$ con $M[0]$

Heapify Modificado(P , M , $i\,0$)

Las diferencias fundamentales en estos algoritmos recae precisamente en los argumentos que recibe y la forma de establecer la manera en que serán ordenados, ya que recibe dos conjuntos de datos. Estos serán el conjunto de D de n puntos sobre el cual se ejecutará la instancia y el conjunto P en $\mathbb{R}$ definido como:

$$P := \{a_i : a_i \in \mathbb{R}, a_i = (y_i - b)/(x_i - a) \forall (x_i, y_i) \in D, i = 1, 2, ..., n, (a, b) \in D, \}$$

El método de ordenamiento para la rutina Heapify hará uso de la primera condición propia del algoritmo para saber si se encuentra en el

brazo izquierdo o derecho del árbol y, en adición, el valor de pendiente correspondiente a ese punto, ordenándolos de manera ascendente. En caso de encontrar dos puntos que compartan la misma pendiente, entrará en juego el algoritmo 2 y así dentro de cada subgrupo de puntos con el mismo valor de pendiente, se seguirán ordenando en sentido creciente.

En las rutinas Heapsort y heapify, se introducen 2 arreglos de igual tamaño, donde los puntos y sus pendientes hacia el punto pivote se guardan en la misma posición, esto es, la posición j-ésima del conjunto de puntos tiene su respectivo valor de pendiente en la posición j-ésima del otro conjunto. Para asegurarse de que conforme los puntos se comienzan a ordenar según la idea pretendida, se procede a intercambiar las posiciones simultáneamente tanto del conjunto de puntos como el de pendientes.

2.4.4. Algoritmo Inteligente

Sea $a \in D$ un punto que hará de pivote, y sea $b \in D$, otro punto cualquiera. El plan es barrer los puntos, calcular las pendientes hacia a , y guardar temporalmente sus valores. Con los valores guardados, se procede a ordenar los puntos basándonos en el algoritmo Hepsort Modificado (Algoritmo). Una vez organizados por ambos criterios, la tarea consiste en comparar cada punto b con los $k \in \mathbb{R}$ puntos siguientes existentes dentro del subgrupo con el que comparten la misma pendiente. El algoritmo permitirá comprobar si existen 4 o más puntos en cada subgrupo y guardar cada uno de ellos en un segmento de recta; sin embargo, al ejecutar esta instancia, no es viable detectar cuál es la longitud de cada subgrupo de elementos, sin aumentar la complejidad del algoritmo, ya que se requeriría de una búsqueda interna añadida. Por ello únicamente se guardará el segmento cuando el punto a sobre que el que se esté realizando la iteración sea menor que el primer elemento guardado por el algoritmo en el segmento. En caso de detectar que se ha salido de un subconjunto de pendientes, se saltará al siguiente hasta mapear todos. Una vez mapeados, el punto pivote avanza y se repite el proceso n veces.

PUNTOS COLINEAL RÁPIDO$(Conjunto\ P\ de\ N\ puntos)$

$segmentos_colineales$ =null

$pendientes$ =null

$puntos$ =null

for i in P

$origen$ =Punto $i-$ésimo en P

$pendientes$ =Valores de pendientes de origen a $b \in P$

$inicio_segmento$ =null;

$colineales$ =null;

for j in P

if (Pendiente de origen a j y $j+1$ iguales)

$contar++$;

If $contar == 2$

Añadir origen y $j-$ésimo punto de P a colineales

$contar++$

$inicio_segmento = j-$ésimo de P

else

Añadir $j-$ésimo de P a colineales

else if $contar = 4$

Añadir $j-$ésimo de P a colineales

if $inicio_segmento$ >origen

Añadir colineales a $segmentos_colineales$

$contar = 1$

else $colineales$ =null

$contar$ =1

return $segmentos_colineales$

2.4.5. Análisis de complejidad del algoritmo inteligente

Se presenta el costo en tiempo sobre cada operación y la cantidad de veces que se ejecuta cada una. Al sumar el tiempo requerido por cada ejecución del algoritmo obtenemos el tiempo total de ejecución. Sea T(N) el tiempo que toma la ejecución del algoritmo inteligente, para calcular su respectivo T(N) se suma el producto de las operaciones llevadas a cabo dentro de los lazos anidados y los tiempos correspondientes a los algoritmos auxiliares:

$$
\begin{aligned}
T(N) &= c_0 N * (c_1 N log(N) + c_2 N) \\
&= c_3 N^2 + c_4 N^2 log(N)) \\
&\leq a N^2 log(N)
\end{aligned}
$$

Donde las constantes $c_i, i = 1, 2, 3, 4$ y a, b, c dependen del costo de las operaciones. De esta forma vemos que el orden del algoritmo con criterio de ordenamiento es del tipo $\mathcal{O}(n^2 log(n))$ un tiempo de ejecución mucho menor al $\mathcal{O}(n^4)$ del algoritmo de fuerza bruta. En adición, el algoritmo inteligente es capaz de detectar segmentos de 4 o más puntos y sin incluir conjuntos ya evaluados. Lo cual es una enorme ventaja respecto al método de fuerza bruta, donde hallar un punto extra incurre en un crecimiento del tiempo de ejecución, si se desean encontrar segmentos de s puntos el orden correspondiente es de $\mathcal{O}(n^s)$, lo que puede volver al algoritmo enormemente lento y caro a nivel computacional.

2.4.6. Motivación de Heapsort

Dado que el algoritmo inteligente requiere de un método de ordenamiento y que el objetivo es reducir el costo computacional del mismo. Debe seleccionarse aquel método que tenga el menor tiempo de ejecución. Para ello existen varios tipos de algoritmos como el *Algoritmo de la Burbuja*, el *Quicksort* o el *Heapsort*.

Las coordenadas de los puntos ingresados para la ejecución del algoritmo serán valores enteros generados aleatoriamente sin que existan puntos repetidos, por lo que no existe mayor control o conocimiento sobre la disposición de los mismos. Por ello, considerar algoritmos como el de la burbuja o *Quicksort* donde el orden de complejidad en el peor de los casos es $\mathcal{O}(n^2)$, a pesar de que en el mejor escenario su tiempo respectivo sea menor, no es una opción óptima, pues el orden del algoritmo seguiría manteniéndose en $\mathcal{O}(n^4)$. Con el *Heapsort*, en cambio, se tiene la certeza de obtener siempre un orden de $\mathcal{O}(n log(n))$, lo cual reduce considerablemente el orden total del algoritmo con criterio de ordenamiento.

2.4.7. Implementación

A fin de obtener un correcto funcionamiento de la implementación del algoritmo inteligente, se usaron archivos de tipo header.h que son cargados desde el fichero principal main.cpp.

El fichero secundario funciones_auxiliares.h es usado para llamar correctamente a los archivos de tipo .txt donde se encuentran los datos a usarse en ambos métodos. Los mismos son los archivos datos_50.txt, datos_100.txt, datos_200.txt, datos_300.txt, datos_400.txt, datos_500.txt, datos_700.txt, datos_800.txt y datos_1000.txt.

Los dos métodos implementados trabajan dentro de una función independiente, respectivamente, dentro de dos archivos .h, llamados Colinear_rapido.h y Fuerza_bruta.h que se nutren de Funciones_auxiliares.h y de dos clases conformadas por estructuras de datos dinámicas, que serán las clases punto.h y segmento.h para:

- Leer los archivos dados en los archivos de tipo .txt.
- Asignar los valores de estos archivos a las coordenadas del punto a crear.
- Calcular la pendiente existente entre 2 pares de puntos.
- Determinar donde comienza y donde termina el segmento que contiene los puntos.
- Ordenar los datos leídos previamente.

Los ficheros de ambos métodos tienen las funciones que se implementaron para:

- Llamar a la ejecución de ambos archivos desde main.cpp
- Presentar los resultados por subgrupo de pendiente en pantalla, así como el número total de segmentos encontrados.

Capítulo 3

Resultados computacionales, conclusiones y recomendaciones

Se presentó en la sección anterior la funcionalidad tanto del algoritmo de fuerza bruta como del inteligente, además de que se mostró los tiempos de ejecución esperados de los mismos. Esta sección pone a prueba la ejecución de los siguientes algoritmos:

- **Algoritmo de fuerza bruta** Resuelve el problema de colinealidad analizando todas las posibles combinaciones de 4 puntos del conjunto inicial y determinando si son colineales o no.
- **Algoritmo inteligente** Propone un orden para los puntos en $\mathbb{R}^2$ usando estructuras de datos y el algoritmo *heapsort* para determinar conjuntos de 4 puntos que sean colineales.

Se reporta el éxito o no las soluciones obtenidas y los tiempos de ejecución de las diferentes pruebas. A partir de ello se compara la eficiencia de los algoritmos y se verifica los resultados teóricos y los obtenidos en práctica.

3.1. Descripción de instancias

Se describen las instancias que se usaron para los experimentos computacionales con los algoritmos de fuerza bruta e inteligente. Los puntos

creados en $\mathbb{R}^2$ de tipo *float* son generados de forma aleatoria, fijando una semilla y usando 2 entradas; n y m. El valor n determina cuántos puntos serán creados y m es el parámetro de la función módulo con la que se crean los números aleatorios.

De esta forma, se generaron 9 instancias aleatorias con números pseudoaleatorios que fueron utilizadas para ejecutar ambos algoritmos de las variantes de la solución al problema de puntos colineales, de fuerza bruta e inteligente, y comparar su tiempo de ejecución con instancias pequeñas, medianas y relativamente grandes. Las 9 instancias se obtuvieron para valores de n iguales a 50, 100, 200, 300, 400, 500, 700, 800 y 1000 puntos.

3.1.1. Resultados

En esta sección se describen los resultados computacionales obtenidos al aplicar los algoritmos de fuerza bruta e inteligente en las 9 instancias detalladas anteriormente. Se realizó 5 pruebas sobre cada una de las instancias. Para realizar la implementación computacional de los algoritmos se utilizó como herramienta el lenguaje de programación C++. Los experimentos fueron llevados a cabo en una computadora con sistema operativo Windows 10 con chip Interl(R) Core(TM) i5-4440 de 3.10 GHz, 4 GB de RAM. No se estableció un tiempo límite de cómputo para que los algoritmos reporten las soluciones encontradas.

Algoritmo de fuerza bruta

A continuación, en las tablas 3.1 y 3.2 se exponen los distintos casos y su eficiencia en términos de tiempo en segundos respecto del algoritmo de fuerza bruta.

Vemos que el algoritmo de fuerza bruta logra encontrar una solución en todas las instancias. Los resultados de las instancias con n entre 50 y 300 han sido obtenidos en un tiempo relativamente corto. Es de la instancia con $n = 300$ a la $n = 400$ donde se tiene el primer salto notable en tiempo de ejecución. A partir de allí los tiempos para la resolución empiezan a tener el predicho comportamiento exponencial.

	N° puntos en la instancia				
Prueba	**50**	**100**	**200**	**300**	**400**
Prueba 1	0.109	0.125	1,000	4.629	14.074
Prueba 2	0.078	0.125	1,000	4.593	14.030
Prueba 3	0.118	0.109	0.988	4.511	14.358
Prueba 4	0.134	0.109	0.984	4.593	14.090
Prueba 5	0.0945	0.109	0.984	4.609	14.030

Cuadro 3.1: Resultados en segundos para las instancias $n = 50, 100, 200, 300, 400$

	N° puntos en la instancia			
Prueba	**500**	**700**	**800**	**1000**
Prueba 1	33.478	145.438	216.40	514.054
Prueba 2	33.437	145.356	215.040	514.054
Prueba 3	33.716	145.438	216.098	514.218
Prueba 4	33.498	145.292	216.130	514.202
Prueba 5	33.513	145.376	216.110	514.218

Cuadro 3.2: Resultados en segundos para las instancias $n = 500, 700, 800, 1000$

Algoritmo inteligente

Las tablas 3.3 y 3.4 indican los distintos casos y su eficiencia en términos de tiempo en segundos respecto del algoritmo inteligente.

	N° puntos en la instancia				
Prueba	**50**	**100**	**200**	**300**	**400**
Prueba 1	0.035	0.042	0.047	0.094	0.125
Prueba 2	0.031	0.031	0.047	0.063	0.109
Prueba 3	0.047	0.047	0.047	0.094	0.109
Prueba 4	0.031	0.047	0.042	0.078	0.109
Prueba 5	0.031	0.031	0.047	0.062	0.125

Cuadro 3.3: Resultados en segundos para las instancias $n = 50, 100, 200, 300, 400$

Vemos que el tiempo de ejecución para todas las instancias se mantiene bastante bajo, es más, las resoluciones para las instancias con $n = 1000$ se realizan en cerca de medio segundo.

	N° puntos en la instancia			
Prueba	**500**	**700**	**800**	**1000**
Prueba 1	0.156	0.266	0.344	0.500
Prueba 2	0.141	0.281	0.328	0.516
Prueba 3	0.156	0.280	0.344	0.513
Prueba 4	0.156	0.266	0.328	0.500
Prueba 5	0.156	0.250	0.328	0.516

Cuadro 3.4: Resultados en segundos para las instancias $n = 500, 700, 800, 1000$

Comparación de resultados entre algoritmos de fuerza bruta e inteligente

El cuadro 3.5 muestra una comparación con los resultados obtenidos en la ejecución entre el algoritmo de fuerza bruta e inteligente. Como se vio anteriormente, cada instancia es probada 5 veces, para realizar la comparación se obtiene un promedio de las pruebas con las 5 instancias.

	Algoritmo	
N° Puntos	**Fuerza bruta**	**Inteligente**
50	0.1067	0.035
100	0.1154	0.0396
200	0.9912	0.046
300	4.587	0.0782
400	14.1164	0.1154
500	33.5284	0.153
700	145.38	0.2664
800	215.9556	0.3344
1000	514.1492	0.509

Cuadro 3.5: Comparación de los promedios de tiempos de ejecución en las 9 instancias

Se puede ver claramente que, como se explicó en la teoría, el algoritmo inteligente tiene un rendimiento notablemente superior que el algoritmo de fuerza bruta, principalmente se nota el resultado a partir de los 300 puntos en la instancia. Vemos que en la instancia $n = 1000$ el tiempo de ejecución del algoritmo de fuerza bruta es mayor al del algoritmo inteligente en cerca de un $1000\,\%$.

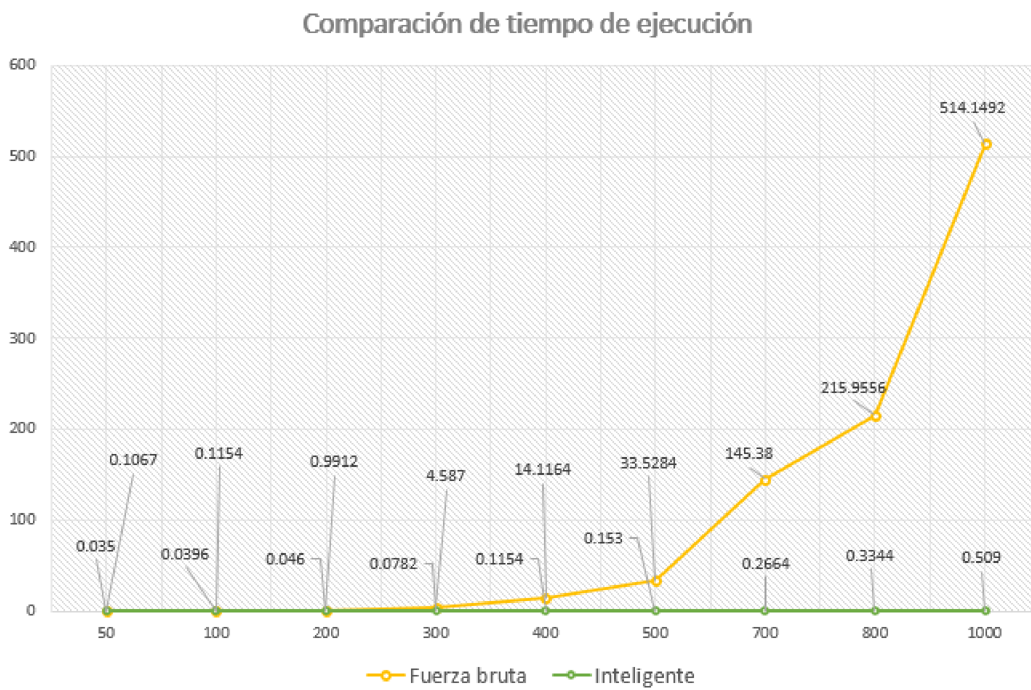


Figura 3.1: Gráfica de línea de comparación de tiempos de ejecución promedio entre ambos algoritmos

En la figura 3.1 se puede ver de forma gráfica la diferencia entre los tiempos de ejecución de los dos algoritmos. Puede verse como la brecha en la eficiencia empieza a hacerse más grande a medida que el valor n de la instancia se hace mayor. Este es el resultado esperado por el análisis teórico.

A continuación, en la figura 3.2 se presentan los puntos empleados para la instancia de 50 puntos . De igual forma se presentan los segmentos hallados tras aplicar el algoritmo inteligente en 3.3:

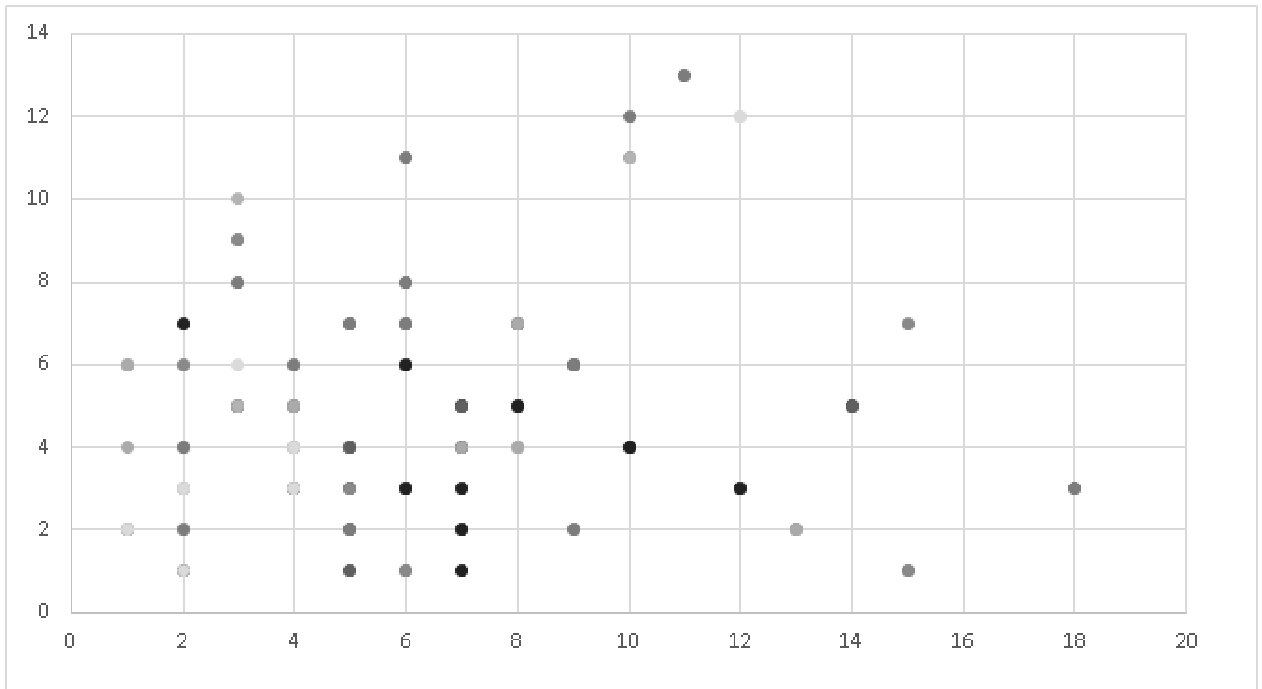


Figura 3.2: Gráfica de los puntos empleados para la primera instancia (50 puntos)

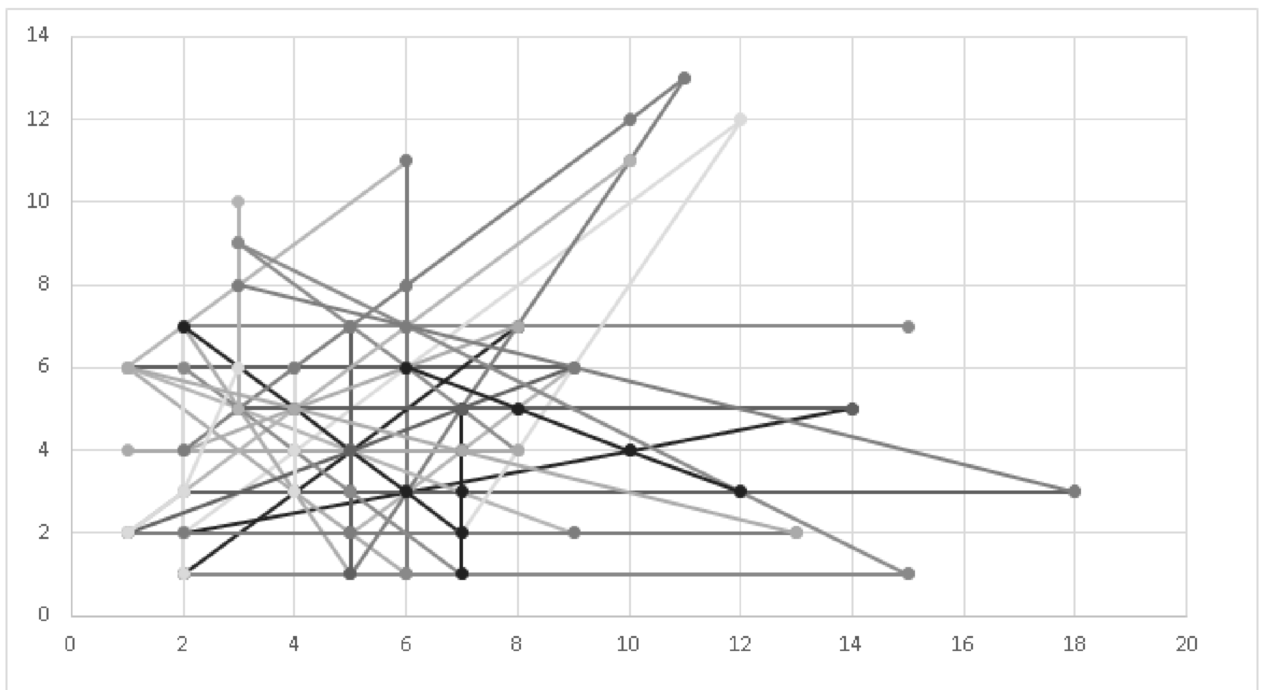


Figura 3.3: Gráfica de la solución por algoritmo inteligente para la primera instancia (50 puntos)

Segmento 1		Segmento 2		Segmento 3		Segmento 4		...		Segmento 35	
x	**y**	**x**	**y**	**x**	**y**	**x**	**y**	**...**	**...**	**x**	**y**
1	6	2	2	5	2	2	1	...	...	18	3
2	6	4	4	6	3	5	1	...	...	9	6
4	6	6	6	7	4	6	1	...	...	6	7
6	6	12	12	8	5	7	1	...	...	3	8
9	6	-	-	9	6	15	1	...	...	-	-

Cuadro 3.6: Puntos y coordenadas solución del algoritmo inteligente para instancia 50

En la tabla 3.6, se muestran parcialmente los segmentos hallados por el método con criterio de ordenamiento. En total son 35 segmentos de puntos colineales de 4 o más puntos, sin repetir ninguna combinación de punto y pendiente. Por el contrario, el algoritmo de fuerza bruta, al barrer todas las posibles combinaciones y estar fijada en encontrar segmentos de exactamente 4 puntos, repite algunas combinaciones hasta encontrar un total de 164 como se muestra en la tabla 3.7.

Segmento 1		Segmento 2		Segmento 3		Segmento 4		Segmento 164	
x	**y**	**x**	**y**	**x**	**y**	**x**	**y**	**x**	**y**
1	6	1	6	1	6	1	6	4	3
4	6	4	6	7	3	5	4	6	3
9	6	2	6	9	2	9	2	12	3
6	6	9	6	3	5	3	5	18	3

Cuadro 3.7: Puntos y coordenadas solución del algoritmo de fuerza bruta para instancia 50

Se puede observar como los segmentos 1 y 2 del algoritmo de fuerza bruta comparten tres puntos, por lo que esos tres puntos están siendo considerados en más de una ocasión pero dentro de la misma recta, repitiendo así, la combinación de punto y pendiente. Lo cual difiere en gran medida con el algoritmo inteligente, en el que se considera una única vez.

Al guardar una única vez los segmentos que se puedan repetir, el algoritmo inteligente encuentra una cantidad mucho menor de segmentos totales, que los que almacena el de fuerza bruta. En este último, se observa en la tabla 3.8 como aumentan considerablemente conforme aumenta el tamaño de la instancia.

	Algoritmo	
N° Puntos	**Fuerza bruta**	**Inteligente**
50	164	35
100	714	112
200	3605	378
300	8588	760
400	16304	1307
500	25649	1956
700	48419	3652
800	63336	4669
1000	47856	7103

Cuadro 3.8: Comparación del número de segmentos encontrados en cada instancia de cada algoritmo

3.2. Conclusiones y recomendaciones

El problema de los puntos colineales ha sido estudiado dentro del área de geometría computacional, siendo varias soluciones propuestas para el problema. Estas soluciones abarcan algoritmos que, obviamente, cumplen con su función principal, pero presentan un rango de eficiencia muy amplio.

Como se mostró en el presente documento, el primer algoritmo propuesto presenta una solución sencilla de comprender y, hasta cierto punto, relativamente fácil de implementar, pero que en su contra presenta un tiempo de ejecución promedio muy poco útil para la práctica, que se acerca a 9 minutos para un conjunto de 1000 puntos. Para aplicaciones sencillas, este tiempo puede parecer aceptable, sin embargo, el aumentar en un $10\,\%$ el tamaño de la instancia incrementa aproximadamente el doble del tiempo de ejecución presentado para la resolución de esa misma instancia. Esto resulta sumamente ineficiente para aplicaciones de mayor envergadura.

En este mismo sentido, se puede observar en la imagen 3.1 el rendimiento del algoritmo inteligente en comparación. En el análisis teórico realizado en el capítulo 2, se anticipa el mejor rendimiento del algoritmo inteligente sobre el otro. El algoritmo inteligente hace uso de varias herramientas de estructuras de datos para poder manejar los recursos a disposición de mejor manera, dando una alternativa de solución frente a

otros.

La geometría computacional cuenta con muchos problemas planteados que han dado paso al desarrollo y creación de nuevas técnicas y estructuras que tratan de aprovechar de la mejor manera su contexto, estas nuevas técnicas a su vez proporcionan nuevas alternativas para crear soluciones eficientes y prácticas. En este sentido, esta solución al problema de puntos colineales puede servir, en sí, como una nueva herramienta para la implementación de aplicaciones en geometría computacional.

Referencias bibliográficas

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

[2] D. E. Knuth, "Structured programming with go to statements," *ACM Computing Surveys (CSUR)*, vol. 6, no. 4, pp. 261–301, 1974.

[3] D. B. West *et al.*, *Introduction to graph theory*, vol. 2. Prentice hall Upper Saddle River, 2001.

[4] L. M. Pardo, "Complejidad computacional." 2016.

[5] A. Karatrantou, C. Panagiotakopoulos, and A. Patras, "Algorithm, pseudo-code and lego mindstorms programming," in *Proceedings of International Conference on Simulation and Programming for Autonomous Robots/Teaching with Robotics: Didactic Approaches and Experiences, Venice, Italy*, pp. 70–79, 2008.

[6] The Linux Information Project, "Algorithms: A very brief introduction." `http://www.linfo.org/algorithm.html`, 2005.

[7] J. Kleinberg, "Introduction to algorithms," 2005.

[8] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, "Introduction to algorithms," 2009.

[9] E. L. Escardó, "Principios de análisis matemático." 1991.

[10] J. Kleinberg and E. Tardos, *Algorithm design.* Pearson Education India, 2006.

[11] A. V. Rosa Guerequeta, “Técnicas de diseño de algoritmos.” 2019.

[12] G. T. Heineman, G. Pollice, and S. Selkow, *Algorithms in a nutshell: A practical guide*. .O’Reilly Media, Inc.", 2016.

[13] S. Rubinstein-Salzedo, “Big o notation and algorithm efficiency,” in *Cryptography*, pp. 75–83, Springer, 2018.

[14] R. Sedgewick and K. Wayne, *Algorithms: Part I*. Addison-Wesley Professional, 2014.

[15] M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, *Data structures and algorithms in Python*. Wiley Hoboken, 2013.

[16] M. Sharir and M. H. Overmars, “A simple output-sensitive algorithm for hidden surface removal. ACM transactions on graphics (TOG), 11(1):1–11.” 1992.