

INFORME: SISTEMA DE DETECCIÓN DE COLISIONES AÉREAS PROYECTO FINAL ANALISIS Y DISEÑO DE ALGORITMOS I

**Cómo encontramos aviones en peligro de chocar
usando Divide y Vencer**

Para el curso: Análisis y Diseño de Algoritmos I

Profesor: Ing. Mateo Echeverry Correa

Estudiantes:

RIVERA ARIAS LAURA SOFIA (2380712-3743)

RODRIGUEZ GUTIERREZ JUAN PABLO (2380422-3743)

Fecha: SIETE DE DICIEMBRE 2025

Semestre: 2025-II

Universidad del Valle

TABLA DE CONTENIDO

1. Descripción Formal del Problema
2. Justificación del Enfoque Divide y Vencer
3. Análisis Teórico del Algoritmo
4. Análisis de Complejidad
5. Conclusiones y Aprendizajes

1. DESCRIPCIÓN FORMAL DEL PROBLEMA

En sistemas de control de tráfico aéreo modernos, múltiples aeronaves reportan constantemente sus posiciones geográficas. Estas posiciones son representadas como coordenadas (latitud, longitud) o, en un modelo simplificado, como puntos en un plano cartesiano 2D.

Problema crítico: Identificar en tiempo real pares de aeronaves cuya proximidad representa riesgo de colisión.

Sea P un conjunto de n puntos en \mathbb{R}^2 :

$$P = \{p_1, p_2, \dots, p_n\} \text{ donde } p_i = (x_i, y_i) \in \mathbb{R}^2$$

Cada punto representa una aeronave con coordenadas (x, y) .

Dado un umbral $\delta > 0$ (distancia mínima segura), el objetivo es identificar todos los pares de aeronaves para los cuales la distancia euclidiana es menor o igual a δ .

Formalmente, se define como el conjunto de pares:

$$R = \{ (p_i, p_j) \in P \times P \mid i < j \text{ y } d(p_i, p_j) \leq \delta \},$$

donde la distancia euclidiana $d(p_i, p_j)$ está dada por:

$$d(p_i, p_j) = \sqrt{(x_j - x_i)^2 + (y_j - y_i)^2}.$$

Este problema se interpreta como una extensión del clásico "problema del par más cercano" [Bentley & Shamos, 1979]. Mientras que el problema clásico busca únicamente el par con distancia mínima, nuestra formulación identifica todos los pares cuya distancia sea $\leq \delta$, lo que es crucial para aplicaciones de seguridad aérea donde múltiples pares pueden estar simultáneamente en riesgo.

El algoritmo se basa en la estrategia "Divide y Vencer" descrita por Bentley y Shamos:

1. **Dividir:** Ordenar los puntos por coordenada X y dividir el conjunto en dos mitades aproximadamente iguales.
2. **Vencer:** Resolver recursivamente el problema en cada mitad.
3. **Combinar:** Unir las soluciones y buscar pares que crucen la línea divisoria, restringiendo la búsqueda a una banda de ancho δ .

Bentley, J. L., & Shamos, M. I. (1979). The closest-pair problem. In D. S. Johnson (Ed.), *Computational Geometry: An Introduction* (pp. 153-174). Springer-Verlag.

2. JUSTIFICACIÓN DEL ENFOQUE DIVIDE Y VENCER

Dado:

- Un conjunto P de n puntos
- Un umbral $\delta > 0$ (distancia mínima segura)

Encontrar: $R = \{(p_i, p_j) \in P \times P \mid i < j \text{ y } d(p_i, p_j) \leq \delta\}$

text

Objetivo: Calcular R eficientemente para grandes valores de n .

Un algoritmo por fuerza bruta compararía cada punto con todos los demás:

Número de comparaciones = $C(n, 2) = n(n-1)/2 \in \Theta(n^2)$

Ejemplos de ejecución por fuerza bruta:

- $n = 100 \rightarrow 4,950$ comparaciones
- $n = 1,000 \rightarrow 499,500$ comparaciones
- $n = 10,000 \rightarrow 49,995,000$ comparaciones

Esta complejidad cuadrática es **inacceptable** para aplicaciones en tiempo real.

El enfoque **Divide y Vencer** fue seleccionado porque:

1. **Optimalidad teórica:** $O(n \log n)$ es el mejor límite inferior conocido para este problema.
2. **Estructura recursiva natural:** Se adapta perfectamente a problemas geométricos.
3. **Eficiencia garantizada:** No depende de distribución de datos.
4. **Aplicabilidad comprobada:** Usado en sistemas reales de control aéreo.

Se seleccionó Divide y Vencer porque reduce la complejidad de $O(n^2)$ a $O(n \log n)$. La función principal `encontrar_pares_cercanos()` implementa las tres fases: DIVIDIR, CONQUISTAR, COMBINAR

3. ANÁLISIS TEORICO DEL ALGORITMO

División de los puntos

```
mitad = n // 2 punto_medio = puntos_x[mitad] puntos_izq_x = puntos_x[:mitad] puntos_der_x  
= puntos_x[mitad:]
```

Línea 1: Calcula el índice que divide la lista de puntos por la mitad.

Línea 2: Guarda el punto medio, que es el punto en la mitad de la lista de puntos ordenados por la coordenada x. Este punto es clave para dividir el conjunto de datos.

Líneas 3 y 4: Se dividen los puntos en dos listas, puntos_izq_x y puntos_der_x, que corresponden a la mitad izquierda y derecha de los puntos, respectivamente.

Distribución de los puntos por la Coordenada Y

Después de dividir los puntos en dos mitades, se deben organizar en función de la coordenada y, ya que el algoritmo también necesita comparar los puntos a lo largo de esta dimensión. Porque si la diferencia en altura ya es mayor que el umbral, aunque estuvieran uno encima del otro, igual estarían lejos.

```
puntos_izq_y = []  
puntos_der_y = []  
for p in puntos_y:  
    if p.x < punto_medio.x:  
        puntos_izq_y.append(p)  
    elif p.x > punto_medio.x: puntos_der_y.append(p)  
    else:  
        if p.id < punto_medio.id:  
            puntos_izq_y.append(p)  
        else: puntos_der_y.append(p)
```

Se crean listas vacías para almacenar los puntos de la mitad izquierda (puntos_izq_y) y derecha (puntos_der_y).

En el ciclo, los puntos ordenados por y son distribuidos en las listas puntos_izq_y y puntos_der_y. Si el punto tiene la misma coordenada x que el punto medio, se usa el id para distribuirlo correctamente en una de las dos mitades.

Recursión: Llamada a la Función para Resolver las Mitades

Una vez que los puntos están organizados, el algoritmo llama recursivamente a dividir_y_vencer para resolver cada mitad de manera independiente.

```
pares_izq = dividir_y_vencer(puntos_izq_x, puntos_izq_y)  
pares_der = dividir_y_vencer(puntos_der_x, puntos_der_y)
```

Se llaman recursivamente a la función `dividir_y_vencer` para cada mitad de los puntos (izquierda y derecha). El resultado de estas llamadas serán los pares cercanos en cada mitad.

Combinación de Resultados

Después de resolver las mitades, el algoritmo combina los resultados de ambas mitades y también compara los puntos cercanos que están cerca de la línea divisoria o sea la banda.

```
pares_totales = pares_izq + pares_der
```

Se combinan los resultados de las dos mitades (pares encontrados en la mitad izquierda y derecha).

Comparación en la Banda Central

El siguiente paso es comparar los puntos que están cerca de la línea divisoria. El algoritmo sólo compara los puntos dentro de un rango definido por el umbral, optimizando así el proceso.

```
banda = [p for p in puntos_y if abs(p.x - punto_medio.x) < umbral]
for i in range(len(banda)): for j in range(i + 1, min(i + 8, len(banda))):
    if banda[j].y - banda[i].y > umbral:
        break
    if distancia(banda[i], banda[j]) <= umbral:
        pares_totales.append((banda[i], banda[j]))
```

Se crea una lista `banda` con los puntos que están cerca de la línea divisoria (distancia x menor que el umbral).

Se comparan los puntos dentro de la banda. La optimización ocurre en el `for`, donde se limitan las comparaciones a los puntos cercanos sólo se comparan hasta 7 puntos adelante (El teorema de la banda establece que, para un conjunto de puntos distribuidos en el plano, los puntos que pueden estar cerca unos de otros (según el umbral de distancia) están restringidos a solo 7 puntos en el eje y . Esto se puede demostrar matemáticamente utilizando teoría de la geometría plana y las propiedades de las distancias euclidianas. Por el teorema geométrico: en un rectángulo de $2\delta \times \delta$ centrado en un punto P , no pueden haber más de 8 puntos con distancia mínima $\geq \delta$ entre ellos. Como P ocupa uno, quedan 7) Si la distancia entre dos puntos en y es mayor que el umbral, el ciclo se interrumpe para evitar comparaciones innecesarias.

Llamada Recursiva Final

El algoritmo continúa recursivamente, dividiendo y venciendo hasta que el número de puntos es pequeño suficiente (3 o menos), y en ese caso, se utilizan comparaciones de fuerza bruta.

```
if n <= 3: pares = []
for i in range(n):
    for j in range(i + 1, n):
        if distancia(puntos_x[i], puntos_x[j]) <= umbral:
```

```
pares.append((puntos_x[i], puntos_x[j]))  
return pares
```

Si el número de puntos es pequeño ($n \leq 3$), el algoritmo aplica la fuerza bruta, calculando todas las distancias entre los puntos y añadiendo los pares cercanos a la lista pares.

Resultados

Una vez que el algoritmo ha recorrido todas las recursiones y ha combinado todos los resultados, devuelve los pares de aeronaves que están cerca del umbral de distancia.

```
return dividir_y_vencer(puntos_x, puntos_y)
```

La función devuelve los pares cercanos después de completar la recursión.

4. ANALISIS DE COMPLEJIDAD

Análisis de Complejidad del Algoritmo *Divide y Vencerás*

El algoritmo de *Divide y Vencerás* para la detección de pares cercanos en el plano 2D puede analizarse utilizando una ecuación de recurrencia.

Ecuación de Recurrencia

El algoritmo divide el conjunto de puntos en dos mitades en cada nivel recursivo y luego resuelve cada mitad de manera independiente. Una vez resueltos los subproblemas, el algoritmo compara los puntos en la banda central. La ecuación de recurrencia para el tiempo de ejecución ($T(n)$) es:

$$[T(n) = 2T\left(\frac{n}{2}\right) + O(n)]$$

- **$2T(n/2)$** : Representa la recursión, donde el problema se divide en dos subproblemas de tamaño $(n/2)$.
- **$O(n)$** : Representa el trabajo realizado en cada nivel de recursión, que es la comparación de los puntos en la banda central, tomando un tiempo proporcional al número de puntos, es decir, **$O(n)$** .

Aplicación del Teorema Maestro

La ecuación de recurrencia tiene la forma estándar del Teorema Maestro:

$$[T(n) = aT\left(\frac{n}{b}\right) + O(n^d)]$$

Donde:

- $(a = 2)$ (dos subproblemas),
- $(b = 2)$ (tamaño de los subproblemas reducido a la mitad),
- $(d = 1)$ (el trabajo adicional es **$O(n)$**).

El Teorema Maestro nos indica que:

- Si $(a > b^d)$, la complejidad es **$O(n^{\log_b a})$** .
- Si $(a = b^d)$, la complejidad es **$O(n^d \log n)$** .
- Si $(a < b^d)$, la complejidad es **$O(n^d)$** .

En nuestro caso, tenemos $(a = 2)$, $(b = 2)$, y $(d = 1)$. La relación entre (a) y (b^d) es:

$$[2 = 2^1]$$

Esto corresponde al **caso 2** del Teorema Maestro, lo que nos da:

$$[T(n) = O(n \log n)]$$

Complejidad Temporal

La complejidad temporal total del algoritmo es **$O(n \log n)$** . Esto se debe a que:

- La recursión tiene **$O(\log n)$** niveles.
- El trabajo en cada nivel es **$O(n)$** (comparaciones en la banda central).

Complejidad Espacial

La complejidad espacial es **$O(n)$** , ya que se utilizan listas para almacenar los puntos ordenados y los resultados intermedios.

Resumen de Complejidad

- **Tiempo (Complejidad Temporal): $O(n \log n)$**
- **Espacio (Complejidad Espacial): $O(n)$**

podemos evidenciar entonces como el algoritmo *Divide y Vencerás* es mucho mas eficiente para resolver el problema de los pares más cercanos en comparación con una solución de fuerza bruta **$O(n^2)$** .

5. CONCLUSIONES Y APRENDIZAJES

Conclusiones

El enfoque de *Divide y Vencerás* aplicado al problema de la detección de pares cercanos en un sistema de control aéreo muestra una mejora significativa en términos de eficiencia en comparación con la solución de fuerza bruta. A través de la recursión y la optimización de las comparaciones en la banda central, el algoritmo reduce la complejidad temporal de $O(n^2)$ a $O(n \log n)$, lo que lo convierte en una opción ideal para escenarios en los que se deben manejar grandes volúmenes de datos, como en el caso de la simulación y control de tráfico aéreo.

Las principales conclusiones de la implementación del algoritmo son:

1. **Reducción de la complejidad:** Utilizando *Divide y Vencerás*, logramos una complejidad $O(n \log n)$ frente a la $O(n^2)$ de la fuerza bruta, lo cual es esencial cuando se deben procesar grandes cantidades de aeronaves en tiempo real.
2. **Eficiencia para aplicaciones en tiempo real:** Dado que las aplicaciones de control aéreo requieren respuestas rápidas y precisas, la optimización de las comparaciones en la banda central es una característica clave del algoritmo.
3. **Aplicabilidad práctica:** Este algoritmo es aplicable a situaciones del mundo real, como la detección de colisiones en el control de tráfico aéreo, donde la seguridad y la rapidez de procesamiento son fundamentales.
4. **Importancia de las optimizaciones geométricas:** La optimización que limita las comparaciones en la banda a solo 7 puntos demuestra cómo las propiedades geométricas de los puntos en el plano pueden ser utilizadas para mejorar el rendimiento sin sacrificar la precisión del resultado.

Aprendizajes

Durante la implementación y análisis de este algoritmo, se obtuvieron varios aprendizajes importantes:

1. **Conceptos fundamentales de la geometría computacional:** Aprendimos sobre el **problema del par más cercano** y cómo se puede abordar de manera eficiente utilizando el enfoque *Divide y Vencerás*, en lugar de la fuerza bruta. Este algoritmo es un ejemplo práctico de cómo las técnicas de geometría computacional pueden mejorar la eficiencia en problemas del mundo real.
2. **Uso de la recursión:** La división de los puntos en mitades y la resolución recursiva del problema es una técnica poderosa que no solo mejora la eficiencia del algoritmo, sino que también facilita su implementación de manera natural.
3. **Optimización de comparaciones en la banda:** El concepto de limitar las comparaciones dentro de la banda central a solo 7 puntos fue una de las

optimizaciones más interesantes y útiles. Aprendimos que la teoría matemática detrás de este enfoque es crucial para garantizar que el algoritmo sea eficiente incluso para grandes conjuntos de puntos.

4. **Implementación práctica en un contexto real:** Aunque el problema original es geométrico, la implementación del algoritmo en un sistema de control aéreo nos permitió conectar los conceptos teóricos con aplicaciones prácticas. Este proyecto nos mostró cómo los conceptos de la informática teórica se pueden aplicar a sistemas reales, como el control de tráfico aéreo.