Análisis de Algoritmos
Tarea 1

Nombres de los alumnos del grupo:
Alonso Bustos, Gabriela Eweld, Constanza Cristinich

21 de abril 2025

Repositorio con archivos: https://github.com/Ado-do/tarea1-ADA

1 Fuerza Bruta (Alonso y Gabriela)

Implemente un algoritmo `brute force` calculando las $n(n-1)$ distancias de manera a seleccionar tal distancia mínima. Realice un análisis de correctitud del algoritmo. Defina y demuestre su complejidad computacional.

1.1 Implementación

Este es el extracto de la implementación en código cpp (`brute_force.cpp`) del algoritmo brute force que calcula las n(n - 1) distancias en un conjunto de puntos:

```
ClosestPair brute_force(vector<Point2D> &points) {
#ifdef DEBUG
    printf("Calculando con fuerza bruta las distancias entre todos los puntos:\n");
#endif

    ClosestPair result;

    size_t n = points.size();
    for (size_t i = 0; i < n - 1; i++) {
        for (size_t j = i + 1; j < n; j++) {
            double dist = euclidean_distance(points[i], points[j]);

#ifdef DEBUG
            printf("\tdist(p%zu, p%zu) = %6.2lf\n", i, j, dist);
#endif

            if (dist < result.distance) {
                result.distance = dist;
                result.pair = {points[i], points[j]};
            }
        }
    }
    return result;
}
```

1.2 Correctitud

Objetivo del algoritmo:

El objetivo principal del algoritmo es encontrar la distancia euclidiana mínima entre todos los pares de puntos contenidos en un vector de estructuras `Point2D`.

Invariante del ciclo:

Definición de la invariante: Al inicio de cada iteración del ciclo externo (índice i), la variable `min_dist` contiene la distancia mínima encontrada entre todos los pares de puntos ya revisados, es decir, aquellos donde $0 \leq a < b \leq i + k$, siendo k el número de iteraciones completas del ciclo interno (índice j).

Verificación de la invariante:

Verificamos en cada momento del loop:

- **Antes del ciclo**
  La variable `min_dist` se inicializa con un valor suficientemente grande (la distancia entre los puntos (0,0) y (100,100). En este punto, aún no se ha realizado ninguna comparación, por lo que la invariante se cumple trivialmente.
- **Durante el ciclo**
  En cada iteración del ciclo doble, se compara la distancia entre el punto i y todos los puntos j tales que $j > i$. Si la distancia calculada es menor que `min_dist`, esta se actualiza con el nuevo valor. En caso contrario, `min_dist` permanece inalterada. De este modo, al final de cada iteración, `min_dist` sigue conteniendo la distancia mínima entre todos los pares evaluados hasta ese momento, cumpliendo así con la invariante.
- **Después del ciclo**
  Al finalizar la ejecución de ambos ciclos, se ha recorrido todo el conjunto de pares posibles (i, j) donde $i < j$, garantizando que todos han sido evaluados una sola vez. Como resultado, la variable `min_dist` contiene efectivamente la distancia mínima global entre todos los pares de puntos del vector. Por lo tanto, se cumple también la condición de salida del algoritmo.

Conclusión:

Dado que la invariante se cumple inicialmente, se mantiene durante cada iteración, y al finalizar garantiza la postcondición esperada, se concluye que el algoritmo es **correcto** con respecto al problema que pretende resolver.

1.3 Complejidad Computacional

- **Teórica**: $\mathcal{O}(n^2)$ por el doble bucle anidado
- **Empírica**: Ver sección de análisis experimental

2 Dividir para Vencer (Constanza)

Diseñe o busque en Internet (cite su fuente) un algoritmo `divide_and_conquer` usando dividir para vencer para encontrar tal distancia en tiempo en $o(n^2)$. Realice un análisis de correctitud del algoritmo. Defina y demuestre su complejidad computacional.

2.1 Diseño

Se escoge el algoritmo explicado en el libro Introduction to Algorithms de Cormen et al. (Sección 33.4), el cual utiliza el paradigma de Dividir para Vencer para resolver de forma eficiente el problema del par de puntos más cercanos. El algoritmo que fue implementado para las siguientes secciones se basa en:

Previo a la recursion: Se crean dos arreglos, X e Y, que contienen todos los puntos del conjunto original. El arreglo X se ordena de forma creciente según la coordenada x, mientras que el arreglo Y se ordena de forma creciente según la coordenada y.

Dividir para Conquistar: Se divide en tres secciones:

1. **Caso base:** Al comenzar se revisa si los puntos existentes en el array a trabajar son menores o iguales a 3, de ser así se devuelve el resultado de utilizar `brute_force`.
2. **Dividir:**
   a) Se calcula el punto medio del arreglo y se divide en 2 arreglos X_i y X_d que corresponden a la mitad de la izquierda y de la derecha respectivamente.
   b) De manera similar se crean los arreglos Y_i e Y_d, que guardan los puntos ordenados de manera creciente según la coordenada y del lado izquierdo y derecho.
   c) Se aplica `divide_and_conquer` a cada lado: en X_i y X_d.
   d) Esto da dos distancias mínimas: δ_i y δ_d, que se comparan y escoge el mínimo entre las dos δ.
3. **Combinar:**
   a) Se forma una franja vertical de ancho 2δ centrada en la línea divisoria.
   b) Se crea un arreglo Y' que guarda aquellos puntos que existen a una distancia menor a δ de la linea divisoria, ordenados de manera creciente según la coordenada y.
   c) Se comparan cada uno de los puntos del nuevo arreglo entre a lo más 6 o 7 puntos siguientes del mismo. Esta propiedad geométrica se traduce a: comparar cada punto del arreglo con aquellos puntos con los que su distancia euclidiana no sobrepase la distancia mínima actual δ.
   d) Si se encuentra una distancia menor a δ dentro de esta franja, se actualiza el resultado.

2.2 Correctitud

Se aborda el análisis de la correctitud separando esta misma en parcial y total.

2.2.1 ***Correctitud Parcial***

Se basa en que cada paso cumple su propósito:

- En el caso base, si hay 3 o menos puntos, el algoritmo recurre al método por fuerza bruta, el cual es correcto por definición.
- En la fase de división, los puntos se separan correctamente en dos subconjuntos disjuntos (V_i, X_d) y sus equivalentes ordenados por y), asegurando que se evalúe la distancia mínima en cada mitad.
- En la combinación, se considera correctamente la franja central, y se comparan sólo los puntos relevantes mediante una propiedad geométrica fundamentada: en un rectángulo de dimensiones $\delta \times 2\delta$ no puede haber más de 8 puntos, separados al menos por δ, lo que justifica comparar solo con los 6 o 7 puntos siguientes en Y'.

2.2.2 ***Correctitud Total***

Se garantiza porque:

- La recursión siempre se aproxima al caso base, disminuyendo el tamaño del problema en cada llamada.
- En cada nivel, se obtiene una solución válida (correctitud parcial), y se combina correctamente con la solución del subproblema opuesto.
- Al combinar los resultados locales de cada subproblema y la franja, se garantiza que el algoritmo retorna globalmente la distancia mínima.

2.3 Complejidad Computacional

Defina y demuestre su complejidad computacional. El algoritmo divide recursivamente un conjunto de n puntos en dos subconjuntos de tamaño $n/2$, y en cada nivel realiza trabajo adicional lineal: construcción de arreglos auxiliares ordenados (Y_i, Y_d e Y') y evaluación de pares en la franja central. Gracias al ordenamiento inicial de los puntos en orden creciente según su coordenada x e y, estas operaciones pueden implementarse en tiempo $\mathcal{O}(n)$ por nivel.

El tiempo de ejecución se puede modelar con la siguiente recurrencia:

$$T(n) = 2T(n/2) + \mathcal{O}(n)$$

Esta recurrencia puede resolverse mediante el Teorema Maestro, que aplica a recurrencias de la forma:

$$T(n) = aT(n/b) + f(n)$$

Donde:

- a: Número de subproblemas
- b: Factor por el cual se divide el tamaño del problema.
- $f(n)$: Costo del trabajo fuera de las llamadas recursivas (división y combinación).

Se tiene que $a = 2$ y $b = 2$, entonces:

$$\log_b a = \log_2 2 = 1$$

y dado que:

$$f(n) = \mathcal{O}(n) = \mathcal{O}(n^{\log_b a})$$

estamos en el caso del Teorema Maestro, en donde:

$$f(n) = \mathcal{O}(n^{\log_b a}) \quad \Rightarrow \quad T(n) = \mathcal{O}(n^{\log_b a \log n}) = \mathcal{O}(n \log n)$$

Por lo tanto, la complejidad temporal del algoritmo es:

$$T(n) = \mathcal{O}(n \log n)$$

3 Implementación (Constanza)

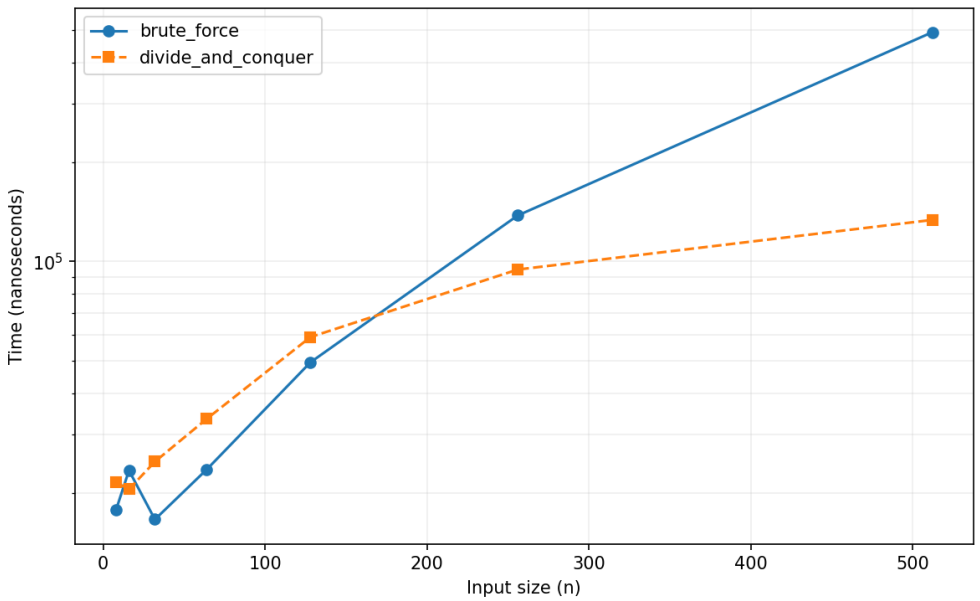
Implemente su algoritmo `divide_and_conquer`, comparando sus respuestas con las de su implementación de `brute_force`.

3.1 Código de implementación:

```
/*
Función DividirParaConquistar recibe un vector de puntos cargados, crea 2 vectores de
punteros, llama a la función que OrdenarPuntosX e OrdenaPuntosY y llama a la función
recursiva para inicializar el algoritmo.
*/
ClosestPair DividirParaConquistar(vector<Point2D> &P);

/*
Función distanciaMinRecursiva recibe los vectores de punteros a puntos X e Y, junto a la
posición en el vector del inicio y del final.
*/
ClosestPair distanciaMinRecursiva(vector<Point2D *> &X, vector<Point2D *> &Y, int inicio,
                                  int final);

/*
Función DistanciaMinFranja recibe un vector "Franja" de punteros a puntos que incluye
solo aquellos puntos cuya coordenada x se encuentra a una distancia menor a "d" de
la línea divisoria vertical del espacio de búsqueda.

La franja esta ordenada en forma creciente según la coordenada Y desde la función
"DividirParaConquistar".

La función hace que cada punto "i" se compara solo con los puntos "j" tales que la
diferencia en la coordenada "y" entre i y j sea menor que d. Aunque y según la teoría
descrita en el libro de de "Introduction to Algorithms" ya mencionado solo es
necesario revisar 7 puntos "j" por punto i en la franja.
Finalmente devuelve la distancia mínima.
*/
ClosestPair DistanciaMinFranja(const vector<Point2D *> &franja,
                               const ClosestPair &current_closest);
```

Más detalles de la implementación en el archivo del código (`src/divide_and_conquer.cpp`).

3.2 Comparación:



4 Análisis Experimental (Alonso y Gabriela)

Diseñe y realice un análisis experimental calculando los tiempos de ejecución en nanosegundos de sus dos soluciones para un conjunto de n puntos cuyas coordenadas enteras están elegidas al azar en un cuadrado de 100×100, variando la cantidad n de puntos entre las potencias de dos de $2^3 = 8$ a $2^9 = 512$.

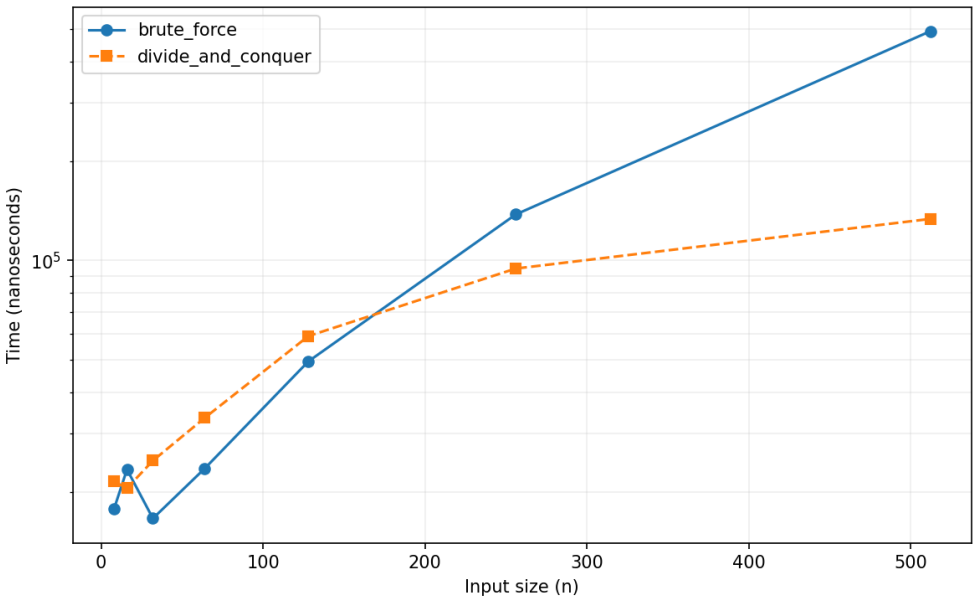
4.1 Diseño

Se diseño un script en bash `benchmark.sh` que realiza lo siguiente:

1. **Preparación de entorno:** se ejecuta el comando `make`, que utiliza un archivo `Makefile` previamente configurado para compilar todo el código implicado en el proyecto.
2. **Ejecución:** se ejecutan pruebas con n = 8, 16, 32, ..., 512 puntos, generados aleatoriamente con el programa `get_random_points`, en todos los algoritmos implementados.
3. **Parsing:** se utiliza el comando linux `sed`, de manera que se encuentra el tiempo de ejecución en el output de los programas ejecutados y se guardan en un archivo csv.
4. **Análisis y Visualización:** se arma el gráfico con ayuda de código en python que recibe el archivo csv y muestra los resultados para su análisis.

4.2 Realización

Gráfico de ejemplo de resultados esperados: Como podemos comprobar a simple vista, efecti-



vamente `divide_and_conquer` presenta un menor complejidad temporal que se pronuncia con una mayor cantidad de puntos.

5 Mejora (Constanza)

Observando el tiempo de ejecución para valores grandes de n, proponga y evalúe una versión trivialmente mejorada de los dos algoritmos.

5.1 Propuesta

Al realizar el análisis experimental con una gran cantidad de puntos, nos percatamos que muchas veces los puntos se duplicaban y resultaba en distancias mínimas cero, debido a la pequeño plano en que se realizan las pruebas (100×100).

Por lo tanto, agregamos la comprobación del caso en que los puntos se llegaban a repetir. Aquí un ejemplo de la comprobación implementada en el código, en `super_brute_force.cpp`:

```
bool detectorDuplicas(const vector<Point2D> &points) {
    set<Point2D> puntosUnicos;
    for (const auto &p : points) {
        if (!puntosUnicos.insert(p).second) {
            closest_points = {p, p};
            return true;
        }
    }
    return false;
}

double brute_force(vector<Point2D> &points) {
    bool duplicaExiste = detectorDuplicas(points);

    if (duplicaExiste == true) {
        return 0.0;
    }
```

Ambos códigos mejorados se encuentran en `src/super_brute_force.cpp` y `src/super_divide_and_conquer.cpp`

6 Gráfico (Alonso y Gabriela)

Construya un gráfico que muestre cómo varían los tiempos de ejecución de sus cuatros soluciones en nanosegundos, variando la cantidad n de puntos entre las potencias de dos de $2^3 = 8$ a $2^9 = 512$.