Laboratorio Closest Pair

Alan Daniel Florez Cerro, *Ingenieria de sistemas*

Abstract

El siguiente laboratorio se enfoca en analizar el comportamiento de un algoritmo que compara las distancias de un conjunto de puntos generados aleatoriamente y determina que par de puntos es el más cercano. Se determinara el par más cercano usando dos estrategias (algoritmo de fuerza bruta y algoritmo empleando ”divide and conquer”). Se observará el tiempo de ejecución de cada estrategia en distintos conjuntos de puntos aleatorios, donde la mitad tendrá un x menor a cierto valor y la otra mitad tendrá un x mayor a dicho valor. El algoritmo se implementará en java y de los resultados obtenidos vemos que divide y vencerás reduce el numero de iteraciones y tiempo de ejecución para determinar el par más cercano en conjuntos grandes de puntos.

Index Terms

divide and conquer, fuerza bruta. complejidad.

I. Introducción

EL objetivo principal de este informe es determinar la complejidad de ambas estrategias poniéndolas a prueba con distintos casos para así poder elaborar una gráfica con los resultados obtenidos de los tiempos de ejecución que ilustre la complejidad de cada una.

Esto es importante porque permite ver como la estrategia de ”divide and conquer” es capaz de hacer la solución de problemas complejos de tal forma que el algoritmo sea más eficiente a la hora de resolver problemas que conllevan una gran cantidad de operaciones como, para este caso, el hallar el par de puntos más cercano en un conjunto de puntos generados aleatoriamente. Para hacerlo solo se requiere de cambiar un poco la forma de ver el problema, aplicar algo de lógica y recursividad de forma que se evite hacer procesos innecesarios o muy demandantes.

mds
Noviembre 18, 2022

II. Definición del problema

Cuando codificamos un algoritmo que deba comparar valores en un conjunto de datos usualmente el primer aproximamiento que nos viene a la mente es el de fuerza bruta considerando que es lo más sencillo de implementar y el poder de computo de los equipos actuales les permite hacer muchas operaciones. Sin embargo, si tuviéramos un conjunto de datos de gran tamaño y recursos de computo limitados usar fuerza bruta resultará en fallos o en desperdicio de recursos.

Para poder reducir el uso de recursos incluso cuando el conjunto de datos es grande podemos usar la estrategia de divide y vencerás definida por Programiz(s. f.) como una estrategia que consiste en dividir el problema en problemas más pequeños, resorlverlos y de sus resultados obtener el resultado buscado.

III. Metodología

Se tomará el tiempo de ejecución del algoritmo de fuerza bruta y de ”divide y vencerás” en conjuntos de n puntos generados aleatoriamente y ordenados de forma ascendente respecto a su valor de x donde n aumentará en potencias de 2, desde $2^2 a 2^{18}$.

Para el algoritmo de fuerza bruta se tomará el tiempo y las iteraciones obtenidas un vez ejecutado sobre el conjunto de puntos, por otra parte para el algoritmo de divide y vencerás se ejecutara el mismo cuatro veces sobre conjuntos aleatorios distintos para cada tamaño n y los tiempos e iteraciones obtenidas serán promediados.

El tiempo en nanosegundos se medirá usando la función System.nanotime() de java antes y después de las lineas de código que llaman a cada algoritmo, la resta de ambas permite hallar el tiempo de ejecución del algoritmo.

Para contar las iteraciones se pondrá un contador dentro de los ciclos usados para determinar el par de puntos más cercanos. A continuación se muestran los algoritmos usados:

Nota: distance(a, b) recibe dos puntos (a y b) y calcula su distancia usando la formula de $d^2 = (Y_2 - Y_1)^2 + (X_2 - X1)^2$

Algorithm 1 Algoritmo de fuerza bruta

```
bruteForce(list)
closest[] ← Empty
closest[0] ← distance(list[0], list[1])
closest[1] ← list[0].x
closest[2] ← list[0].y
closest[3] ← list[1].x
closest[4] ← list[1].y
for j from 0 to lenght(list)-2 with increments of 1 do
    for i from 0 to lenght(list)-1 with increments of 1 do
        if distance(list[j], list[i]) < closest[0] then
            closest[0] ← distance(list[j], list[i])
            closest[1] ← list[j].x
            closest[2] ← list[j].y
            closest[3] ← list[i].x
            closest[4] ← list[i].y
        end if
    end for
    iterations ← iterations + 1
end for
```

Algorithm 2 Algoritmo divide y vencerás

```
divideAndConquer(list)
firsthalf[] ← the first half of the points in the list
secondhalf[] ← the second half of the points in the list
first[] ← bruteForce(firsthalf)
second[] ← bruteForce(secondhalf)
if first[0] > second[0] then
    bruteforce(firsthalf, secondhalf, second)
    return second
else
    if first[0] = second[0] then
        bruteforce(firsthalf, secondhalf, second)
        return second
    else
        bruteforce(firsthalf, secondhalf, first)
        return first
    end if
end if
```

El algoritmo de divide y vencerás maneja dos casos. Estos son:

- Los puntos más cercanos del conjunto se encuentran en el mismo subconjunto.
- Los puntos más cercanos del conjunto se encuentran cada uno en un subconjunto distinto.

Si se da el primer caso, el algoritmo anterior simplemente encuentra el par mas cercano al comparar el par de un grupo con el otro y escoger el menor.

Cuando se da el segundo caso se deben revisar los puntos cercanos a donde se realizó la división del grupo de puntos para poder determinar correctamente el par de puntos más cercanos. Para ello se usa una versión que sigue la misma lógica del algoritmo de fuerza bruta, solo que un tanto modificada.

Nota: El método discard(a, b, c) (donde a es una lista, b un índice de la misma y c un array) utilizado en el algoritmo 3 recorre la lista a y calcula la distancia entre cada punto y el punto en el índice b. Si la distancia es menor a la distancia c entonces guarda añade el punto a un array que retorna al final. Esto lo hace de manera recursiva, pues al estar los puntos en orden ascendente se puede saber si se debe seguir revisando en base al resultado obtenido.

Algorithm 3 Algoritmo de fuerza bruta (modificado)

```
bruteForce(listA, listB, dist[])
firstgroup[] ← discard(listA, lenght(listA) − 1, dist[0])
secondgroup[] ← discard(listB, 0, dist[0])
for j from 0 to lenght(listA)-1 with increments of 1 do
    for i from 0 to lenght(listB)-1 with increments of 1 do
        if distance(list[j], list[i]) < dist[0] then
            dist[0] ← distance(list[j], list[i])
            dist[1] ← list[j].x
            dist[2] ← list[j].y
            dist[3] ← list[i].x
            dist[4] ← list[i].y
        end if
    end for
    iterations ← iterations + 1
end for
```

Estos algoritmos fueron implementados en java y se puede ver en el siguiente repositorio de github : https://github.com/adcerro/closest-pair

Luego de obtener los resultados en un archivo de texto se pueden generar las gráficas y tablas con los resultados obtenidos.

IV. Resultados

Puntos	Iteraciones	Tiempo (ms)
4	6	622600
8	28	144300
16	120	129900
32	496	233000
64	2016	1754700
128	8128	1498600
256	32640	7091000
512	130816	15373200
1024	523776	15469900
2048	2096128	21580900
4096	8386560	56996600
8192	33550336	459652300
16384	134209536	1083541700
32768	536854528	4046464200
65536	2147450880	22597086800
131072	8589869056	135702551400
262144	34359607296	802027289500

TABLE I
Resultados fuerza bruta

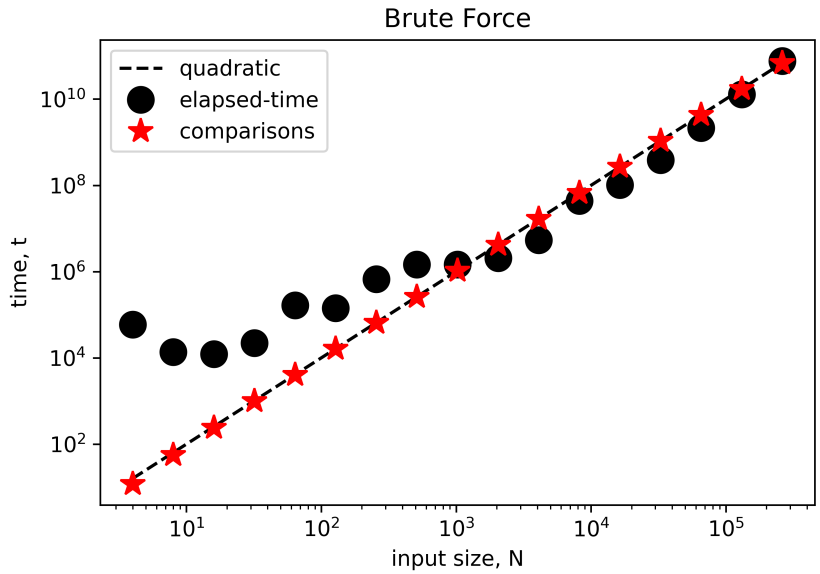

Fig. 1. Resultados algoritmo iterativo

Puntos	Iteraciones	Tiempo (ms)
4	3	10487775
8	16	758975
16	72	661400
32	240	486275
64	992	601150
128	4032	1163550
256	16256	2936900
512	65280	4203275
1024	261632	3493450
2048	1047552	7773700
4096	4192256	25683525
8192	16773120	102718075
16384	67100672	445447950
32768	268419072	1575060025
65536	1073709056	6736732150
131072	4294901760	33748572725
262144	17179738112	270556875925

TABLE II
RESULTADOS DIVIDE AND CONQUER

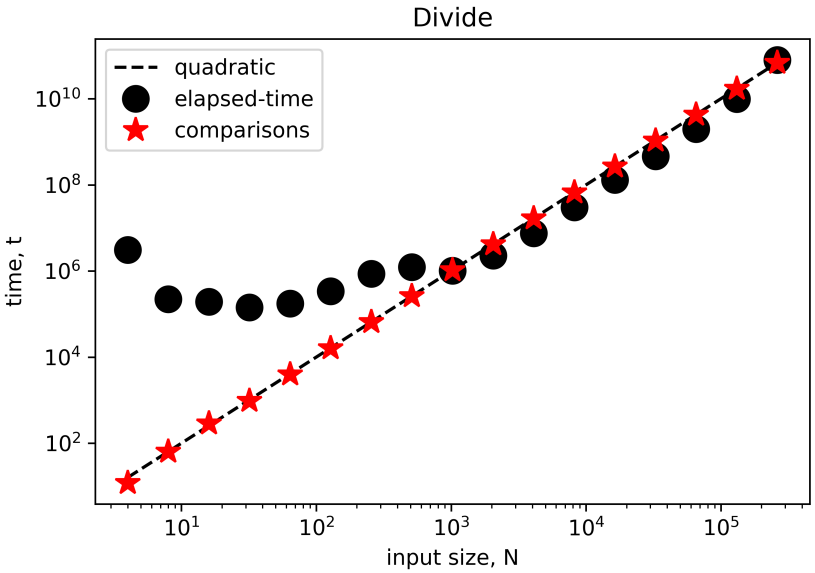

Fig. 2. Resultados algoritmo divide and conquer

V. Discusión

Si bien podemos ver en la tablaII que el número promedio de iteraciones que le tomó al algoritmo usando ”divide y vencerás” es menor al numero de iteraciones que requiere usar el algoritmo de fuerza bruta, las gráficas siguen apuntando a que la implementación usada de dicha estrategia tiene complejidad cuadrática , por lo que, en torno a complejidad, ambos algoritmos son iguales

Se esperaba que la complejidad del algoritmo de fuerza bruta fuera cuadrática, por lo que la implementación cumple la expectativa. Sin embargo, se esperaba que la complejidad del algoritmo de divide y vencerás fuera lineal, por lo que, la implementación no cumplió la expectativa.

En la tablaII se observan tiempos de ejecución promedio bastante altos en comparación a los tiempos en la tabla I para los primeros conjuntos de puntos, luego, mientras aumenta la cantidad de puntos se evidencia que este patrón se invierte, por lo que para conjuntos con muchos puntos divide y vencerás logra hallar el par más cercano más rápido que fuerza bruta.

Entonces, ¿que debe mejorarse de la implementación actual del algoritmo con divide y vencerás?. Pues considero que lo primero a mejorar es que en los test se usó un limite para el valor aleatorio de Y de 1000 el cual puede resultar en puntos demasiado dispersos en el eje y para los n pequeños, lo que contrarresta en parte el beneficio que puede dar usar esta estrategia.

Lo segundo es que al realizar un análisis del algoritmo 3 se tiene:

$$\sum_{j=1}^{N}\sum_{i=1}^{N}1$$

$$\sum_{j=1}^{N}N$$

$$N \cdot N$$

$$N^2$$

Lo que nos deja ver que se debe mejorar la manera como el algoritmo recorre los arrays de puntos cercanos al centro para determinar un par cercano, pues revisa todos los posibles elementos cercanos en vez de poder descartar elementos más distantes en base los resultados que se van obteniendo.

Indagando más respecto a los resultados obtenidos para la implementación de divide y vencerás el profesor Misael resaltó que la implementación (ver algoritmo 2) como tal no usa por completo el aproximamiento, ya que solo divide la lista en dos partes y nada más. Si bien esto beneficia los conjuntos de puntos pequeños, no trae beneficios para los conjuntos más grandes pues se siguen haciendo muchas comparaciones usando fuerza bruta, lo que resulta en la complejidad cuadrática.

VI. Mejoras

Teniendo en cuenta lo analizado en discución considero que se puede mejorar el algoritmos así:

Algorithm 4 Algoritmo divide y vencerás

```
dividAndConquer(list)
firsthalf[] ← the first half of the points in the list
secondhalf[] ← the second half of the points in the list
if list.size() > 6 then
    f[] ← divideAndConquer(firsthalf)
    s[] ← divideAndConquer(secondhalf)
    if f[0] > s[0] then
        bruteForce(firsthalf, secondhalf, s)
        return s
    else
        bruteForce(firsthalf, secondhalf, f)
        return f
    end if
else
    first[] ← bruteForce(firsthalf)
    second[] ← bruteForce(secondhalf)
    if first[0] > second[0] then
        bruteforce(firsthalf, secondhalf, second)
        return second
    else
        if first[0] = second[0] then
            bruteforce(firsthalf, secondhalf, second)
            return second
        else
            bruteforce(firsthalf, secondhalf, first)
            return first
        end if
    end if
end if
```

VII. Conclusión

Usar la implementación de fuerza bruta puede funcionar bien para conjuntos de puntos pequeños, pero, al aumentar la cantidad de puntos se ve afectado el rendimiento y se hace una cantidad de comparaciones bastante alta para determinar el par más cercano.

Si bien la implementación usada para este informe no obtuvo un comportamiento lineal esperado, si muestra que logra reducir considerablemente el número de comparaciones realizadas en conjuntos de puntos cada vez más grandes así como también logra reducir el tiempo requerido para hallar el par más cercano en conjuntos de gran cantidad de puntos.

Las implementaciones de algoritmos con el enfoque divide y vencerás requieren de buenas codificaciones de recursividad para que logren dar al máximo las ventajas que ofrece esta perspectiva.

Reconocimiento

Agradecimientos al profesor Misael por su contribución a la mejora de la implementación del algoritmo para este laboratorio.

References

[1] Programiz. (s. f.). Divide and Conquer Algorithm. Recuperado 26 de octubre de 2022, de https://www.programiz.com/dsa/divide-and-conquer
[2] Diaz Maldonado, M. (8 de Agosto de 2022). loglogPlot.py. Github. Recuperado el 26 de octubre de 2022, de https://github.com/misael-diaz/computer-programming/blob/main/src/io/java/loglogPlot.py
[3] How to increase the Java stack size? (13 de septiembre de 2010). Stack Overflow. Recuperado el 4 de noviembre de 2022, de https://stackoverflow.com/questions/3700459/how-to-increase-the-java-stack-size

Alan Florez Estdudiante de ingenieria de sistemas en la Universidad del norte.