Laboratorio Closest Pair con LinkedList

Alan Daniel Florez Cerro, *Ingenieria de sistemas*

Abstract

El siguiente laboratorio se enfoca, como en el laboratorio anterior, 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. Sin embargo, esta vez la estructura de datos usada para las pruebas será una implementación propia de listas enlazadas. Igual que en el laboratorio anterior, se determinará el par más cercano usando las estrategias de fuerza bruta y "divide and conquer"), observando el tiempo de ejecución de cada una 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. Se reutilizará gran parte de la implementación en java anterior y de los resultados obtenidos vemos que usando listas enlazadas como estructura de datos causa que el tiempo de ejecución de los algoritmos crezca en comparación a usar ArryList

Index Terms

divide and conquer, fuerza bruta. complejidad, LinkedList.

I. Introducción

EL objetivo principal de este informe es determinar si los resultados para la complejidad de ambas estrategias se ve alterado de alguna manera al utilizar una estructura de datos (LinkedList implementada por uno mismo), 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 el uso de una estructura de datos distinta puede alterar los resultados previamente obtenidos. Además, podemos esperar que el numero de iteraciones no se vea muy afectado pero si los tiempos de ejecución pues el usar listas enlazadas no cambia las iteraciones que requiere realizar el algoritmo para calcular el par más cercano, lo que nos permite obtener una nueva perspectiva respecto al informe anterior, cuyos resultados se usarán para comparación.

Es importante resaltar que se hace uso de listas enlazadas pues, como menciona Cairó(2006):

> Este es un tipo de estructura lineal y dinámica de datos. Lineal porque a cada elemento le puede seguir sólo otro elemento; dinámica) porque se puede manejar la memoria de manera flexible, sin necesidad de reservar espacio con antelación. La principal ventaja de manejar un tipo dinámico de datos es que se pueden adquirir posiciones de memoria a medida que se necesitan; éstas se liberan cuando ya no se requieren.
>
> mds
> Noviembre 24, 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. Sin embargo, del informe anterior ya hemos aprendido que usar aproximamientos como divide y vencerás que reduzcan la cantidad de operaciones necesarias en base a lógica permiten obtener mejores resultados.

Ahora bien, aunque es cierto que el como se codifica el algoritmo incide fuertemente en sus resultados, también existen otros factores que pueden alterar su comportamiento, y en el caso de nuestra implementación pondremos a prueba como sustituir ArrayList por una implementación propia de listas enlazadas incide en el comportamiento de nuestro algoritmo.

III. Metodología

Se tomará el tiempo de ejecución del algoritmo de fuerza bruta y de "divide y vencerás" usando una implemenatción propia de listas enlazadas y usando ArrayLists 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 hasta que el tiempo de ejecución sea demasiado alto, en caso de la versión con ArrayList se logró ejecutar hasta con 2^{18} puntos y en el caso de la versión con LinkedList hasta 2^{14} puntos.

Nota: La implementación de la lista enlazada personalizada se puede ver en: https://github.com/adcerro/closest_linked-list_/blob/main/LinkedList.java

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:

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
```

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

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.

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
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
```

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
```

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 en x entre cada punto y el punto en el índice b. Si la distancia es menor a la distancia c entonces añade el punto a un Array o LinkedList 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.

Estos algoritmos fueron implementados en java y se pueden ver en el siguiente repositorio de github : https://github.com/adcerro/closest_linked-list_.git

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

IV. Resultados

Puntos	Iteraciones	Tiempo (ms)
4	6	32200
8	28	43000
16	120	89600
32	496	394100
64	2016	676500
128	8128	1087800
256	32640	4671500
512	130816	6188600
1024	523776	11980800
2048	2096128	14932100
4096	8386560	38897300
8192	33550336	168208400
16384	134209536	783966400
32768	536854528	2206748300
65536	2147450880	9461164400
131072	8589869056	39927323800
262144	34359607296	501522026400

TABLE I
Resultados fuerza bruta con ArrayList

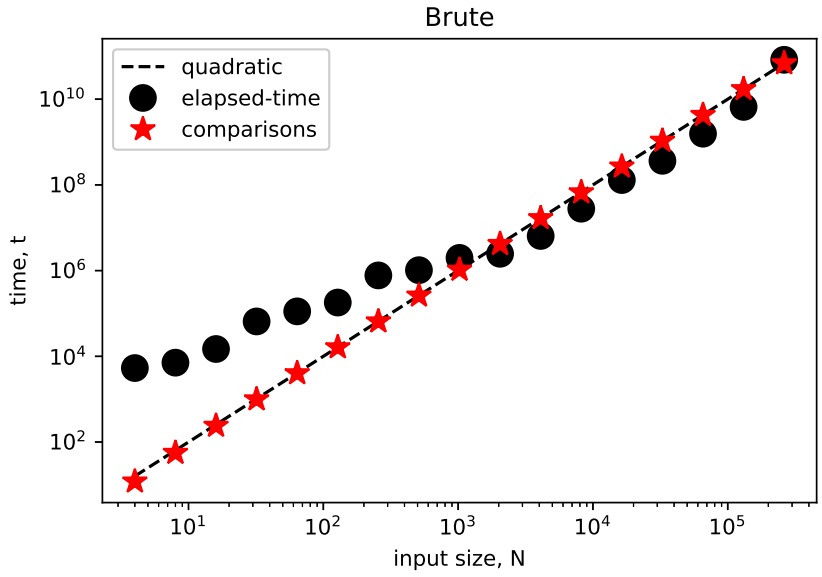

Fig. 1. Resultados algoritmo fuerza bruta con ArrayList

Puntos	Iteraciones	Tiempo (ms)
4	6	32575
8	20	43800
16	40	7003950
32	57	87775
64	94	175200
128	169	142300
256	292	302800
512	488	235725
1024	884	451225
2048	1613	983525
4096	3099	2266750
8192	6187	4112150
16384	12297	5877875
32768	24528	9713350
65536	49266	18014550
131072	98349	47090775
262144	197843	153357475

TABLE II
RESULTADOS DIVIDE Y VENCERÁS CON ARRAYLIST

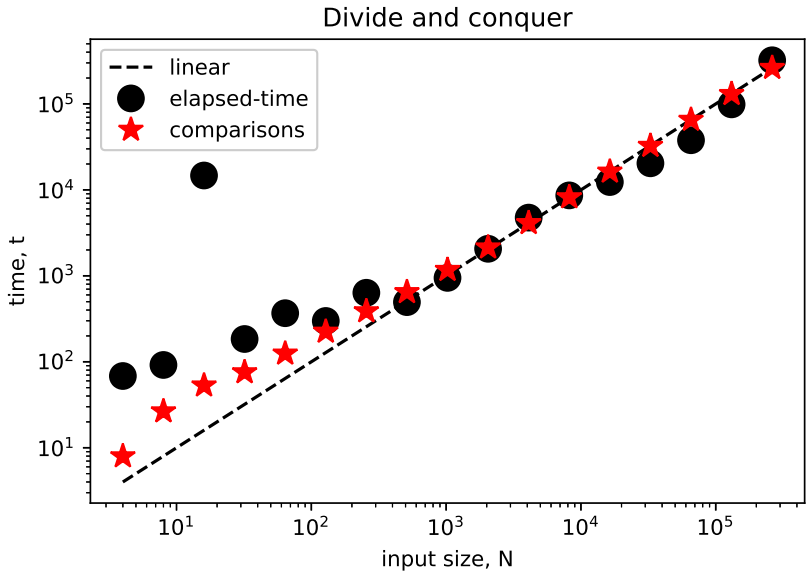

Fig. 2. Resultados algoritmo divide and conquer con ArrayList

Puntos	Iteraciones	Tiempo (ms)
4	6	63700
8	28	161700
16	120	274100
32	496	667400
64	2016	6925600
128	8128	3660000
256	32640	24735900
512	130816	75925300
1024	523776	568953600
2048	2096128	4732181900
4096	8386560	39307846600
8192	33550336	322736597800
16384	134209536	4852096053400

TABLE III
RESULTADOS FUERZA BRUTA CON LINKEDLIST

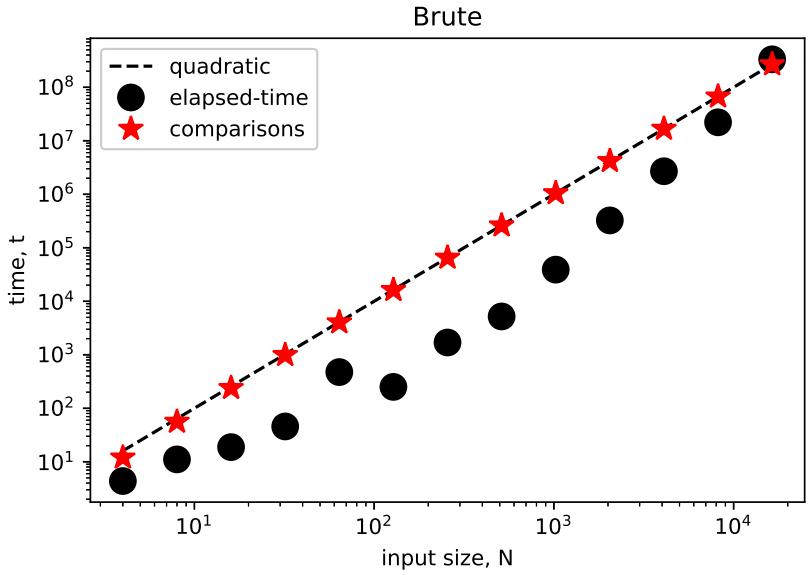

Fig. 3. Resultados algoritmo fuerza bruta con LinkedList

Puntos	Iteraciones	Tiempo (ms)
4	6	4446250
8	24	26500
16	34	35325
32	59	98650
64	103	225175
128	170	254675
256	298	259600
512	504	760150
1024	885	2804800
2048	1608	11440125
4096	3112	50112050
8192	6187	171533850
16384	12311	700195150

TABLE IV
RESULTADOS ALGORITMO DIVIDE AND CONQUER CON LINKEDLIST

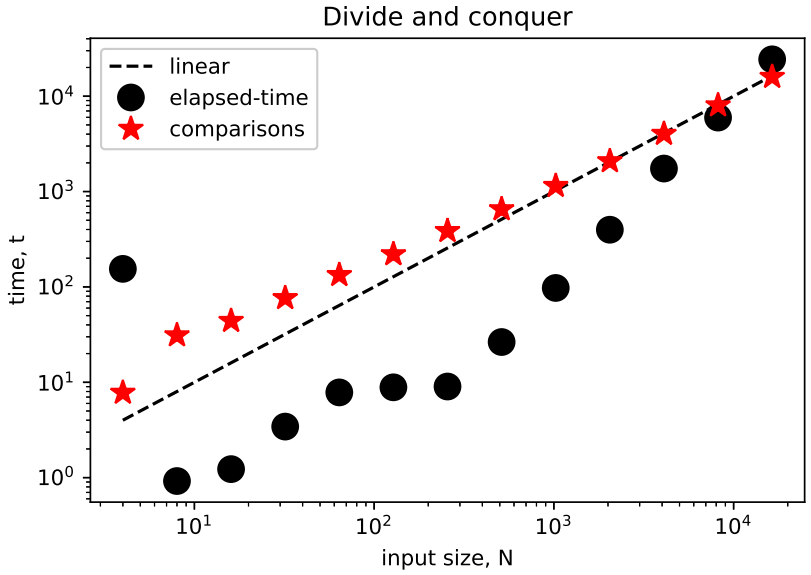

Fig. 4. Resultados algoritmo divide and conquer con LinkedList

V. Discusión

Empezamos comparando las tablas I y III. En las mismas se observa que, como se esperaba, el número de iteraciones no se ve alterado por el uso de ArrayList o LinkedList. También se observa con las gráficas 1 y 3 que la complejidad es la misma para ambas estructuras. Por otra parte, calculando el promedio de los tiempos de ambas usando Excel obtenemos un tiempo promedio de $4,01504 * 10^{11}$ ms usando LinkedList y de 32596966541 ms usando ArrayList, por lo tanto, la implementación de fuerza bruta con LinkedList consume más tiempo en promedio que la que usa ArrayList.

Comparando las tablas II y IV se encuentra que el número de iteraciones ya no es igual para cada cantidad de puntos, siendo en general la implementación con LinkedList la que requiere más iteraciones para cada caso. Sin embargo, dado a que el número de iteraciones usando divide y vencerás es un promedio y la cantidad de iteraciones siempre puede variar para cada prueba, no se puede afirmar concluyentemente que el uso de LinkedList causo este aumento en las iteraciones promedio. Además, si se observan las gráficas 2 y 4 se nota que ambas siguen una tendencia lineal. Finalmente, calculando el promedio de los tiempos de ambas obtenemos que la implementación con LinkedList tarda en promedio 72476330,77 ms y la implementación con ArrayList tarda en promedio 14699517,65 ms, por lo tanto, la implementación de divide y vencerás con LinkedList consume más tiempo en promedio que la que usa ArrayList.

Una explicación para estos resultados es que la implementación personalizada de LinkedList utiliza búsqueda secuencial, lo que causa que al aumentar la cantidad de puntos se requiera una cantidad considerable de iteraciones dentro de la lista para las operaciones que requieren los algoritmos, causando el aumento en el tiempo de ejecución promedio.

VI. Conclusión

Usar una estructura de datos distinta no altera directamente las iteraciones que requiere realizar el algoritmo implementado como tal para encontrar el par de puntos más cercano. Sin embargo, la codificación/optimización de la estructura de datos si incide directamente en el tiempo de ejecución de las implementaciones y el número de iteraciones en general que realiza el programa.

Se puede mejorar la lista enlazada para reducir el tiempo de ejecución de las implementaciones que la utilizan empleando algoritmos de búsqueda mejor optimizados. Aún así, se debe tener en cuenta que el ”manejo” de la información de listas enlazadas difiere del de ArrayList por lo que, a consideración del autor, las implementaciones de fuerza bruta y divide and conquer con listas enlazadas requieren ser pensadas e implementadas de tal forma que aprovechen eficientemente esta estructura de datos.

Reconocimiento

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

REFERENCES

[1] Cairó, O., & Guardati, S. (2006). Estructuras de datos. MC GRAW HILL INTERAMERICANA. Recuperado el 24 de noviembre de 2022, de http://up-rid2.up.ac.pa:8080/xmlui/handle/123456789/1299
[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.