Universidad de Huelva

Ingeniería Informática

Asignatura:

Algorítmica y Modelos de computación

3º Curso

Integrantes:

Marina Vizcaino Bayo

Miriam Rodríguez Franco

Israel Fargas Asquith

Fecha: 13/11/2023

Práctica 1: Algorítmica y modelos de computación

1. Análisis de algoritmos exhaustivos y Divide y Vencerás

1.1 Planteamiento del problema

Dado un conjunto de puntos situados en un plano$P = \{(x1, y1);\ (x2, y2);\ \ldots; (xn, yn)\}$ este problema consiste en encontrar (teniendo en cuenta que la distancia entre dos puntos i y j es $\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$ el punto pi = (xi,yi) más cercano a otro pj = (xj,yj) tal que distancia ((xi,yi), (xj,yj)) sea mínima, es decir, se debe encontrar de entre todos los puntos posibles cuál es el que está más cerca de otro punto (encontrar la pareja de puntos con la menor distancia euclídea entre ellos).

1.2 Búsqueda Exhaustiva

1.2.1 Explicación:

La primera aproximación que hemos usado para resolver el problema del punto más cercano es la más fácil de codificar, consiste en una búsqueda de *fuerza bruta* realizando todos los cálculos para todos los pares de puntos. A continuación mostraremos un pseudocódigo sencillo y fácil de codificar en cualquier lenguaje pero con un gran coste temporal y computacional por la gran cantidad de operaciones que realiza. Sigue siendo un algoritmo válido para tallas *pequeñas,* pero resulta una práctica completamente fútil en tallas mayores.

1.2.2 Pseudocódigo:

[1] **procedimiento** distanciaEntrePuntos(Punto a[2], Punto b): Real
Empieza
Real distancia =: 0
distancia =: $\sqrt{(x_a - x_b)^2 + (y_a - y_b)^2}$
devuelve distancia

[1] ***Este mismo método se usa para calcular la distancia entre los puntos en todos los algoritmos.***
[2] ***Suponemos que los puntos están formados por dos Reales x e y***

Fin

El conteo de operaciones de la función anterior puede ser expresada como 9OE, pero se podría ser más exactos calculando cuantas operaciones elementales realizan la función que implemente la raíz cuadrada y la potencia. Para más información consulte la primera entrada de la bibliografía.

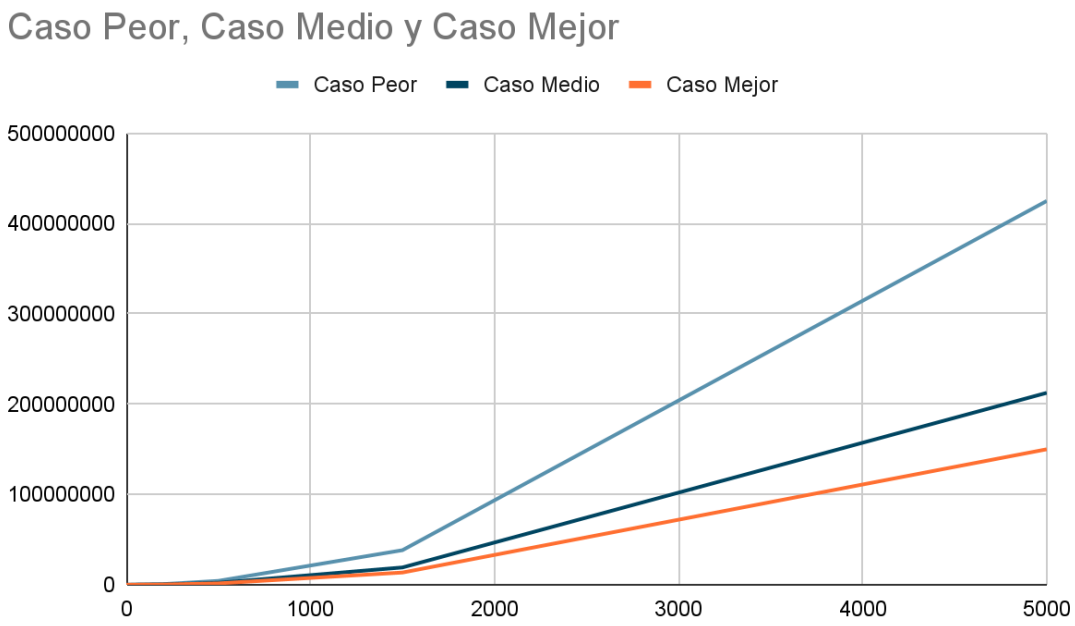
```
procedimiento búsquedaExhaustiva(lista[1...n]:Puntos): DosPuntos
Empieza
    Real distanciaMinima =: +∞
    Real aux =: 0
    DosPuntos puntosMasCercanos
    Punto p1
    Punto p2

    Para i=: 1 hasta  n hacer
      Para j=: i+1 hasta n hacer
       aux =: distanciaEntrePuntos(lista[i],lista[j])
            Si aux < distanciaMinima entonces
                distanciaMinima =: aux
                p1 =: lista[i]
                p2 =: lista[j]
             fSi
      fPara
    fPara
    puntosMasCercanos =:p1 y p2
    devuelve puntosMasCercanos
Fin
```

El conteo resultante es de 6 declaraciones, 3 asignaciones y 1 devolución de datos. Para el cálculo de los bucles anidados vemos que en el caso peor se dará cuando el par de puntos menor sean los últimos de la lista, en cuyo caso siempre se cumple que ***aux*** sea menor que ***distanciaMinima***.

$T(n) = \sum_{i=1}^{n} (\sum_{j=i+1}^{n} 1 + 9 + 1 + 6) + 10 = \sum_{i=1}^{n} (\sum_{j=i+1}^{n} 17) + 10 = 17n^2 + 10 = O(n^2)$

En el caso medio del algoritmo exhaustivo suponemos que es equiprobable que el par de puntos más cercano esté en cualquier lugar de la lista a analizar. Por lo que si ejecutamos un número suficiente de veces el algoritmo sólo entrará en la el la operación **Si** para la mitad de la cantidad de puntos. Por lo que la complejidad del caso medio $\Theta(17n^2/2 + 10) = \Theta(n^2)$

Para el caso mejor el primer par de puntos encontrados será el menor, por lo que no realizará más operaciones de comparación, solo las necesarias del bucle $\Omega(6n^2 + 10) = \Omega(n^2)$



1.3 Exhaustivo con poda

1.3.1 Explicación

La siguiente aproximación lógica consiste en eliminar los puntos que se encuentren más alejados, para ello los ordenaremos en la coordenada X y cuando un punto sea menor que otro abordaremos esa iteración del bucle ahorrando en comparaciones y asignaciones.

1.3.2 Pseudocódigo

procedimiento Ordenaquicksort[3](lista[1…n]: puntos, eje:Cadena)lista[1…n]:puntos{
Empieza
    Quicksort(puntos, 0,n-1, eje)
    devuelve puntos
Fin

procedimiento ExhaustivoConPoda(lista[1…n]: puntos): ParPuntos

[3] Hemos implementado el algoritmo QuickSort cuya complejidad es de nlogn

```
Empieza
Real distanciaMinima =: +∞;
Real aux =: 0
p =: Ordenaquicksort(lista[1…n], "x")
ParDePuntos ParMasCercano
Punto p1
Punto p2
Real  distanciaX
 Para i=:1 hasta n
      Para j=:i+1 hasta n
               distanciaX =:  lista[j].X  - lista[i].X
               Si distanciaX >= distanciaMinima entonces
                        InterrumpirBucle
              fSi

         aux =: distanciaEntrePuntos(lista[i],lista[j])
         Si aux < distanciaMinima entonces
            distanciaMinima =: aux
            p1 =: lista[i]
            p2 =: lista[j]
         fSi

      fPara
   fPara


   ParMasCercano = p1 y p2
   devuelve ParMasCercano
Fin
```

El conteo resultante es de 7 declaraciones, 3 asignaciones, 1 devolución de datos y el procesamiento del ordenamiento de los puntos en el eje X, el coste del algoritmo ***QuickSort*** es de $O(nlog(n))$. Para el cálculo de los bucles anidados vemos que en el peor caso coincide con el exhaustivo tradicional, en cuyo caso siempre se cumple que ***aux*** sea menor que ***distanciaMinima*** y que ***distanciaX*** también sea siempre menor, este caso es muy improbable en nuestra aproximación, ya que las coordenadas están generadas de forma aleatoria.

$T(n) = \sum_{i=1}^{n} (\sum_{j=i+1}^{n} 2+7+3+6) + nlog(n) + 11 = \sum_{i=1}^{n} (\sum_{j=i+1}^{n} 18) + nlog(n) + 11 =$

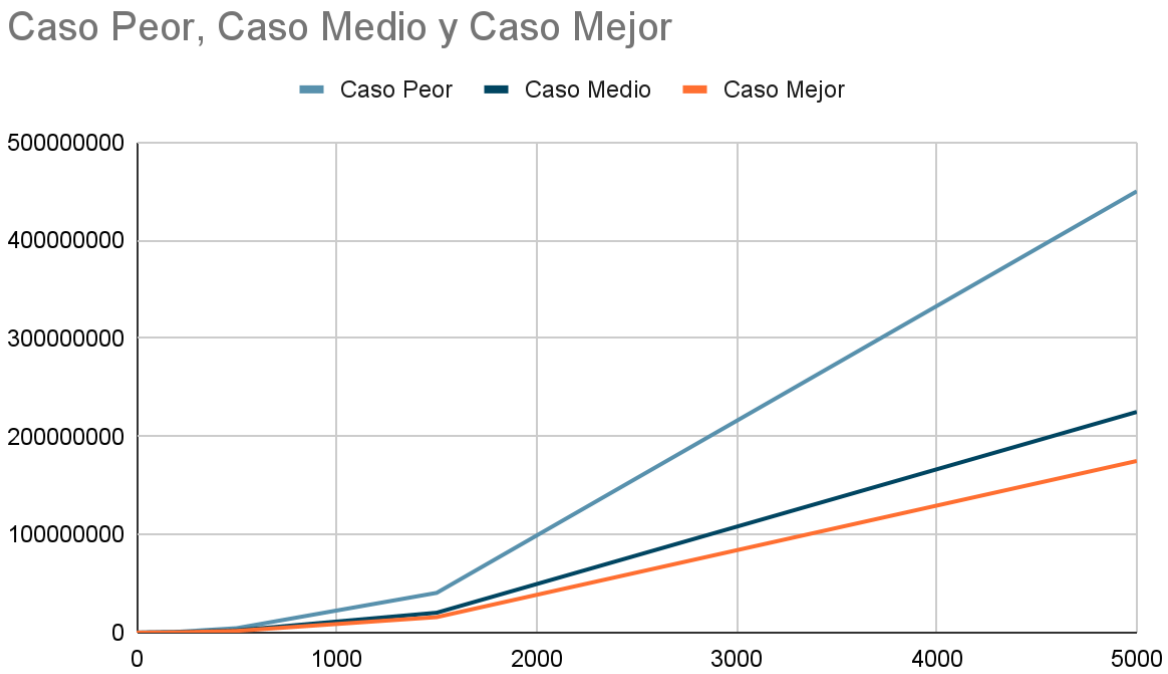
$$= 18n^2 + nlog(n) + 11 = O(n^2)$$

En el caso medio del algoritmo exhaustivo suponemos que es equiprobable que el par de puntos más cercano esté en cualquier lugar de la lista a analizar. Por lo que si ejecutamos un número suficiente de veces el algoritmo abortará el segundo bucle la mitad de veces. Por lo que la complejidad del caso medio

$\Theta(18n^2/2 + nlog(n) + 11) = \Theta(n^2/2)$

Para el caso mejor el primer par de puntos encontrados será el menor, por lo que siempre abortará el segundo bucle

$$\Omega(7n^2 \quad + nlog(n) + 10) \;=\; \Omega(n^2)$$



1. 4 Divide y vencerás sin mejora

1.4.1 Explicación

Para una aproximación más eficiente usaremos la técnica de Divide y Vencerás, en la cuál ordenaremos los puntos por una coordenada, en nuestro caso hemos optado por la X. Dividiremos el espacio de búsqueda …. poner alguna tontería más

1.4.2 Pseudocódigo

```
procedimiento ParDePuntos DyV(Punto P[], int izq, int der)
    si (der – izq + 1 <= 3) entonces
        devolver Exhaustivo(P);
    fsi

    var: mitad <= (der + izq)/2
    ParDePuntos pIzq= DyV(P, izq, mitad)
    ParDePuntos pDer= DyV(P, mitad+1, der)

    var distI = distanciaEntrePuntos(pIzq.getA(), pIzq.getB())
    var distD = distanciaEntrePuntos(pDer.getA(), pDer.getB())

    ParDePuntos pDistMin

    si (distI <= distD) entonces
        pDistMin<= pIzq
    sino
        pDistMin<= pDer
    fsi

    para a <= mitad + 1 hasta der hacer
        si (P[a].GetX – P[mitad + 1] > distanciaEntrePuntos(pDistMin.getA(),
    pDistMin.getB())) entonces
            break
        fsi
    fpara


    para b = izq hasta mitad hacer
        si (P[mitad].GetX – P[b].GetX > distanciaEntrePuntos(pDistMin.getA(),
        pDistMin.getB())) entonces
            break
        fsi
    fpara
```

```
        ParDePuntos taux
        para c <= b+1 hasta mitad hacer
                para d <= mitad+1 hasta a-1 hacer
                        taux = new ParDePuntos(P[c], P[d]])
                        si (distanciaEntrePuntos(P.get(c),
                        P.get(d))<=distanciaEntrePuntos(pDistMin.getA(), pDistMin.getB()))
                        entonces
                                pDistMin<= taux
                        fsi
                fpara
        fpara


        para c <= b+1 hasta mitad hacer
                para d <= mitad+1 hasta a-1 hacer
                        taux = new ParDePuntos(P[c], P[d])
                                si (distanciaEntrePuntos(p.get(c), p.get(d)) <=
                                distanciaEntrePuntos(pDistMin.getA(),
                                pDistMin.getB()))entonces
                                        pDistMin<= taux
                                fsi
                        fpara
                fpara
        fpara
        devolver pDistMin
fprocedimiento
```

Nuestro algoritmo está implementado para recibir un array ordenado(de menor a mayor según la coordenada x), para ello hemos utilizado el algoritmo Quicksort, que ordenará nuestro array en un tiempo promedio de n*log(n). Cuando el tamaño del array del subconjunto es pequeño (3 puntos o menos) utilizamos el algoritmo exhaustivo para encontrar la distancia mínima entre los puntos. Esto tiene un costo constante, ya que el número de comparaciones necesarias es fijo y pequeño. En términos de notación de complejidad del tiempo, podríamos decir que el costo del caso base es de O(1). Cuando el conjunto se divide en dos mitades y se realizan llamadas recursivas en cada mitad. Cada llamada recursiva tiene un costo de T(n/2), donde n es el tamaño del conjunto original. La combinación de resultados implica comparar las distancias mínimas encontradas en las mitades izquierda y derecha del conjunto. Esta operación es de O(1) en términos de operaciones elementales.

Por último, la comprobación de puntos en el medio, implica la iteración de dos bucles, comprobando la franja media, que contribuye con un costo proporcional al tamaño del conjunto de entrada, y su complejidad es O(n) en términos de operaciones elementales.

La expresión en función de tiempo T(n) sería la siguiente:

$$T(n) = 2T(n/2) + nk + c$$

El $2T(n/2)$ representa el tiempo de ejecución de las dos llamadas recursivas en subproblemas pequeños. El nk es el costo asociado con la comprobación de la franja media. c representa el costo constante asociado con otras operaciones como combinación o caso base.

$$T(n) = \begin{cases} 1 & \text{si } n<=3 \\ 2T(n/2) + n\cdot k + c & \text{si } n>3 \end{cases}$$

$T(n) = 2T(n/2) + n$

$T(n) - 2T(n/2) = n$

$t(2^k) - 2t(2^{k-1}) = 2^k$

$t_{k-0} - 2t_{k-1} = 2^k$

$(x-2)\ (x-2)$

Raíces: $r=2$ | $m=2$

$t_k = C_1\cdot 2^k\cdot k^0 + C_2\cdot 2^k\cdot k^1$

$t_k = C_1\cdot k^0 + C_2\cdot 2^k\cdot k^1$

cambio de variable

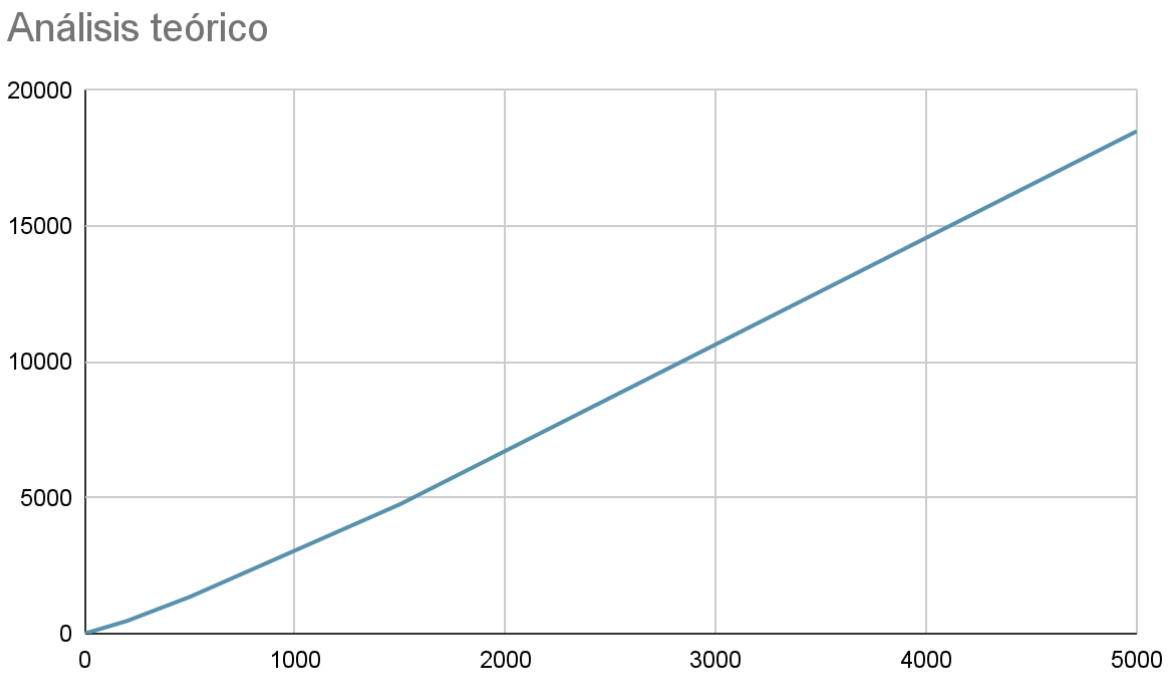
$n=2^k$

$k=\log_2 n$

$T(n) = C_1\cdot n + C_2\cdot n\cdot \log_2 n$ → $O(n\log_2 n)$

En conclusión el algoritmo DyV tendrá un coste aproximado de O(n*log(n)).

El caso mejor se dará cuando no haya que hacer comparaciones en la zona delimitada por ***dmin***.cY en el caso peor todos los puntos están localizados en una zona muy pequeña del eje x, lo que resulta en un gran número de comparaciones en la zona ***dmin***.


Análisis teórico
20000
15000
10000
5000
0
0
1000
2000
3000
4000
5000

1.5 Divide y vencerás con mejora

1.5.1 Explicación

El enfoque de Divide y Vencerás puede perder eficacia cuando los puntos no están distribuidos uniformemente,especialmente en el peor de los casos,donde todos los puntos tienen la misma coordenada X, es decir, están alineados verticalmente.En esta situación, todos los puntos quedarían dentro de la franja, por lo que realizar una búsqueda exhaustiva dentro de la franja supondría un tiempo de ejecución de O(n^2).

Para abordar este escenario,optimizamos la búsqueda en la franja central. Ordenaremos los puntos de la franja media por la coordenada Y, y sólo comprobaremos aquellos puntos que estén a menos de 12 posiciones en la lista ordenada.

1.5.2 Pseudocódigo

```
procedimiento ParDePuntos DyVMejorado(Punto P[], int izq, int der)

    si (der – izq + 1 <= 3) entonces
        devolver Exhaustivo(T);
    fsi

    var: mitad <= (der + izq)/2
    ParDePuntos pIzq= DyVMejorado(P, izq, mitad)
    ParDePuntos pDer= DyVMejorado(P, mitad+1, der)

    var distI = distanciaEntrePuntos(pIzq.getA(), pIzq.getB())
    var distD = distanciaEntrePuntos(pDer.getA(), pDer.getB())

    ParDePuntos pDistMin

    si (distI <= distD) entonces
        pDistMin<= pIzq
    sino
        pDistMin<= pDer
    fsi


    //Ordenamos puntos franja media
    Punto franjaMedia[]
    ParDePuntos taux

    para a <= mitad + 1 hasta der hacer
        si (P[a].GetX – P[mitad] > distanciaEntrePuntos(pDistMin.getA(),
        pDistMin.getB())) entonces
```

```
                    break
            sino
                    franjaMedia <= P[a]
            fsi
    fpara


    para b = izq hasta mitad hacer
            si (P[mitad].GetX – P[b].GetX > distanciaEntrePuntos(pDistMin.getA(),
            pDistMin.getB())) entonces
                    break
            sino
                    franjaMedia <= P[b]

            fsi
    fpara

    franjaMedia <= Ordenaquicksort(franjaMedia, "y");


    para i <= 0 hasta Longitud(franjaMedia) hacer
            para j <= i+1 hasta Minimo(Longitud(franjaMedia),i+12) hacer
                    Var dis <= distanciaEntrePuntos(franjaMedia[i], franjaMedia[j])
                    si (dist < distanciaEntrePuntos(pDistMin.getA(),
                    pDistMin.getB()))entonces
                            pDistMin <= ParDePuntos(franjaMedia[i], franjaMedia[j])
                    fsi
            fpara
    fpara



    devolver pDistMin
fprocedimiento
```

En este algoritmo, se introduce una optimización en la búsqueda de puntos en la franja media.En lugar de realizar una búsqueda exhaustiva para todos los puntos en la franja central, se realizará una búsqueda limitada en los puntos ordenados en la franja media. La mejora radica en la optimización de la búsqueda en la franja central al limitar el número de comparaciones, lo cual puede reducir significativamente el tiempo de ejecución en situaciones específicas.

La fase de ordenación de la franja central tiene un coste de O(n log(n)) debido al uso del algoritmo Quicksort.La búsqueda exhaustiva posterior tiene un coste de O(n). Aunque la mejora reduce el número de comparaciones en ciertos casos, la complejidad total del algoritmo sigue siendo O(nlog(n)). Esto se debe a que la fase de ordenación Quicksort y la resolución recursiva dominan la complejidad del algoritmo.

2. Análisis de algoritmos Voraces

2.1 Planteamiento del problema

El problema del viajante es un problema NP-Completo, muy fácil de formular pero difícil de encontrar una solución aceptable. Se dan n nodos que llamaremos ciudades de las cuales se pueden partir a cualquiera de las n-1 nodos restantes, buscaremos cuál es el camino más corto sin recorrer ninguna ciudad dos veces, las ciudades estarán representadas por puntos en la pantalla y la distancia se calculará con la distancia real entre esos píxeles. Se nos propone afrontar el problema desde dos enfoques de algoritmos voraces, uno dirigido unidireccional y otro dirigido bidireccional.

El primer algoritmo elige un nodo aleatorio y busca el nodo más cercano y no visitado. Repite este proceso hasta que no queda por visitar ninguno y vuelve al nodo origen.

El segundo algoritmo elige un nodo aleatorio y elige el más cercano y el siguiente más cercano, a partir de ahí cada uno va recorriendo los más cercanos entre sí hasta que no queda ninguno sin visitar.

2.2 Comparativa de Voraces sobre los datasets proporcionados

TABLA COMPARATIVA DE COSTES. berlin52.tsp

UNIDIRECCIONAL	BIDIRECCIONAL
9014.993861074247	9293.386945100217

TABLA COMPARATIVA DE COSTES. ch150.tsp

UNIDIRECCIONAL	BIDIRECCIONAL
7526.666159747981	7578.598862454384

TABLA COMPARATIVA DE COSTES. ch130.tsp

UNIDIRECCIONAL	BIDIRECCIONAL
7853.3670156637745	7537.168825221916

TABLA COMPARATIVA DE COSTES. d493.tsp

UNIDIRECCIONAL	BIDIRECCIONAL
43393.3702156001	44356.10722913072

TABLA COMPARATIVA DE COSTES. d657.tsp

UNIDIRECCIONAL	BIDIRECCIONAL
66363.6761246027	64621.67836417514

2.2.2 Comparativa de Voraces por tallas

TABLA COMPARATIVA DE COSTE

TALLA	UNIDIRECCIONAL	BIDIRECCIONAL
100	9585.877067200516	10012.148789575622
200	12363.155382666995	12637.690565445522
300	16618.12266640924	16784.719627568476
400	17444.527720086295	17323.026734783765
500	20378.20332932175	21020.911135289793
600	23122.061356930284	22526.71175669436
700	24723.969437415715	24056.959741523777
800	26084.74488982342	26936.949951845876
900	27484.33626334255	27580.95613822342
1000	29008.049073347425	29418.500349165948

3 Bibliografía

Complejidad de O(log(n)) en potencia

Complejidad de sqrt O(log(n))

Formato de generación de archivos TSP

Travelling salesman description