Algoritmos voraces (Greedy)

ALGORÍTMICA

UNIVERSIDAD DE GRANADA

JOAQUÍN ARCILA PÉREZ
LAURA LÁZARO SORALUCE
CRISTÓBAL MERINO SÁEZ
ÁLVARO MOLINA ÁLVAREZ

ÍNDICE

INTRODUCCIÓN

Esta práctica consiste en el uso de algoritmos Greedy para la resolución de dos problemas, uno de contenedores y otro del viajante del comercio.

Ejercicio 1: Este ejercicio tiene dos apartados que tienen común que tenemos un carguero que soporta K kilos y una serie de contenedores, cuya suma total supera estos K kilos. El primero de los apartados nos pide buscar el algoritmo greedy que maximiza el número de contenedores que entran en el barco, mientras que el segundo nos pide el algoritmo que maximiza el número de toneladas que caben en el barco.

Ejercicio 2: Este ejercicio se basa en encontrar el camino más corto, que una un cierto número de ciudades, sin pasar varias veces por la misma (no se pueden formar ciclos). Para ello hemos implementado tres algoritmos distintos: vecino más cercano, inserción y un algoritmo propio; y los hemos probado con tres conjuntos de datos distintos.

Para este ejercicio, lo primero que hemos hecho es crear una clase Punto2D, que guarda las coordenadas x e y de cada punto, y que tiene varios métodos de consulta, así como uno que calcula la distancia hasta otro punto. Este último nos ha sido muy útil para calcular la matriz de adyacencia, que guarda las distancias entre cada dos puntos, por ejemplo la de ulysses16.tsp:

0	5	5	3	10	8	7	0	11	7	25	5	5	5	6	1
5	0	1	4	16	14	13	6	17	13	31	11	10	11	12	6
5	1	0	4	16	13	12	5	16	12	30	10	10	10	12	5
3	4	4	0	12	11	10	2	14	11	28	8	7	8	8	4
10	16	16	12	0	4	5	10	7	8	15	6	6	6	4	10
8	14	13	11	4	0	1	8	4	3	17	3	3	2	2	7
7	13	12	10	5	1	0	7	4	2	18	2	2	2	3	6
0	6	5	2	10	8	7	0	11	8	25	5	5	5	6	2
11	17	16	14	7	4	4	11	0	3	15	6	6	7	7	10
7	13	12	11	8	3	2	8	3	0	18	3	3	4	5	6
25	31	30	28	15	17	18	25	15	18	0	20	20	20	19	24
5	11	10	8	6	3	2	5	6	3	20	0	0	0	3	4
5	10	10	7	6	3	2	5	6	3	20	0	0	0	2	4
5	11	10	8	6	2	2	5	7	4	20	0	0	0	2	4
6	12	12	8	4	2	3	6	7	5	19	3	2	2	0	6
1	6	5	4	10	7	6	2	10	6	24	4	4	4	6	0

DESARROLLO

1.Ejercicio 1

1.1 Ejercicio de los contenedores

```
vector<int> greedy1 (vector<int> pesos, int K){
 sort(pesos.begin(), pesos.end());
 vector<int> out;
 int suma=0;

 for(auto x : pesos){
 if((suma+=x)<=K){
 out.push_back(x);
 }
else
 break;
 }

 return out;
}
```

Como sabemos el algoritmo sort tiene una eficacia de nlog(n).

Un bucle con instrucciones O(1), por lo que la eficacia quedaría como O(n).

Este algoritmo lo que hace es ordenar todos los contenedores por peso de menor a mayor y lo va recorriendo añadiendo pesos mientras comprueba que la suma no se pase del tope, saliendo del bucle cuando llegamos a un valor que ya no se puede meter, ya que a partir de ese ninguno más va a poder insertarse.

Analizando la eficiencia teórica de este algoritmo, vemos que se queda el producto de nlog(n) por al eficiencia del bucle, por tanto: $T(n) \in O(n^2 \cdot \log(n))$.

Para demostrar su optimalidad, empleamos una reducción al absurdo. Digamos que nuestra función nos da que logramos meter K contenedores en el carguero y sean $\{\alpha i\}$ $i \in \{1,..,K\}$ esos contenedores. Supongamos que nuestro algoritmo no es el más óptimo, por lo que existirá un $\rho \in N$ $\rho>=1$, para el cual existirá un conjunto de contenedores $\{\beta i\}$ $i \in \{1,...,K+\rho\}$, para el cual se cumplirá que $\Sigma(i=1,k)(\alpha i) <= \Sigma(i=1,k+\rho)(\beta i)$. Sin perder generalidad, asumiremos que ambos conjuntos están ordenados de forma creciente, es decir:

$m<n \Rightarrow \alpha m <= \alpha n \ \forall n,m \in \{1,....,k\}$

$m´<n´ \Rightarrow \beta m´ <= \beta n´ \ \forall n´,m´ \in \{1,....,k+p\}$

Entonces, por seleccionar nuestro algoritmo los K elementos con el menor peso, tenemos que:

$\Sigma(i=1,k)(\alpha i) <= \Sigma(i=1,k)(\beta i) < \Sigma(i=1,k)(\beta i)+ \Sigma(i=k,k+p)(\beta i) = \Sigma(i=1,k+\rho)(\beta i)$

lo cual es una contradicción, por lo que se demuestra que nuestro algoritmo es el más óptimo posible.

DESARROLLO

1.2 Ejercicio de las cargas

```
vector<int> greedy2(vector<int> pesos, int K){
  int aux=0;          //carga montada
  vector<int> out;
  sort(pesos.begin(), pesos.end());
  for (auto it = pesos.end(); it != pesos.begin(); --it) {
    if((aux + *it)<=K){     //se puede meter en la carga
      aux += *it;
      out.push_back(*it);
    }
  }
  return out;
}
```

Como sabemos el algoritmo sort tiene una eficacia de nlog(n).

Un bucle con instrucciones O(1), por lo que la eficacia quedaría como O(n).

Este algoritmo es parecido al del ejercicio anterior, solo que con la diferencia de que esta vez estamos recorriendo el vector de final a principio, o sea, primero buscando los tamaños de contenedor más grandes. La otra diferencia es que este algoritmo recorre el vector entero, porque que un contenedor no entre no significa que los siguientes no entren tampoco.

Analizando la eficiencia teórica de este algoritmo, vemos que se queda el producto de nlog(n) por al eficiencia del bucle, por tanto: $T(n) \in O(n*\log(n))$.

Podemos observar en este caso que este algoritmo no es óptimo, lo que se puede ver con el siguiente contraejemplo:

Tomamos K=11, y P[6,4,3,2]. Por nuestro algoritmo tomaríamos 6,4 lo cual no es la solución óptima ya que si tomáramos 6,3,2 llenaríamos completamente el carguero.

DESARROLLO

2.Ejercicio 2

2.1 Algoritmo vecino más cercano

```
int VecinoMasCercano (int dim, int **matriz, vector<int> &recorrido) {

 srand(time(0));
 int comienzo=rand()%dim;
 recorrido.push_back(comienzo);
```

Punto por el que comienza el recorrido, escogido de manera aleatoria

```
 int insertar; // Siguiente punto a insertar en el recorrido
 bool cogido=false;
 int suma=0; // Suma total del recorrido

 while (recorrido.size()<dim) {
```

Mientras queden puntos por visitar, comprobamos todos los puntos uno por uno. Este while se recorre n-1 veces, ya que al empezar el vector ya tiene 1 elemento.

```
  int dist_min=10000000;
  for (int i=0; i<dim; i++) {
   cogido=false;
```

Este for se recorre n-1 veces.

```
   for (auto j=recorrido.begin(); j!=recorrido.end() && !cogido; ++j){
    if (i==(*j))
     cogido=true;
   }
```

Si el punto que estamos comprobando ya está en el recorrido, no hacemos nada con él, lo descartamos.
Este for se recorre i veces, siendo i el tamaño del recorrido -1.

```
   if (!cogido && matriz[i][recorrido[recorrido.size()-1]]<dist_min) {
    dist_min=matriz[i][recorrido[recorrido.size()-1]];
    insertar = i;
   }
  }
```

De todos los puntos que no están ya cogidos, nos quedamos con el que esté a menos distancia del último punto por el que se ha pasado, el último del recorrido que llevamos.

```
  suma+=matriz[recorrido[recorrido.size()-1]][insertar];
  recorrido.push_back(insertar);
 }
```

Insertamos el nuevo punto y añadimos la nueva distancia recorrida a la suma total.

```
 suma+=matriz[recorrido[recorrido.size()-1]][comienzo];
 recorrido.push_back(comienzo);
```

Insertamos el primer punto visitado para cerrar el ciclo.

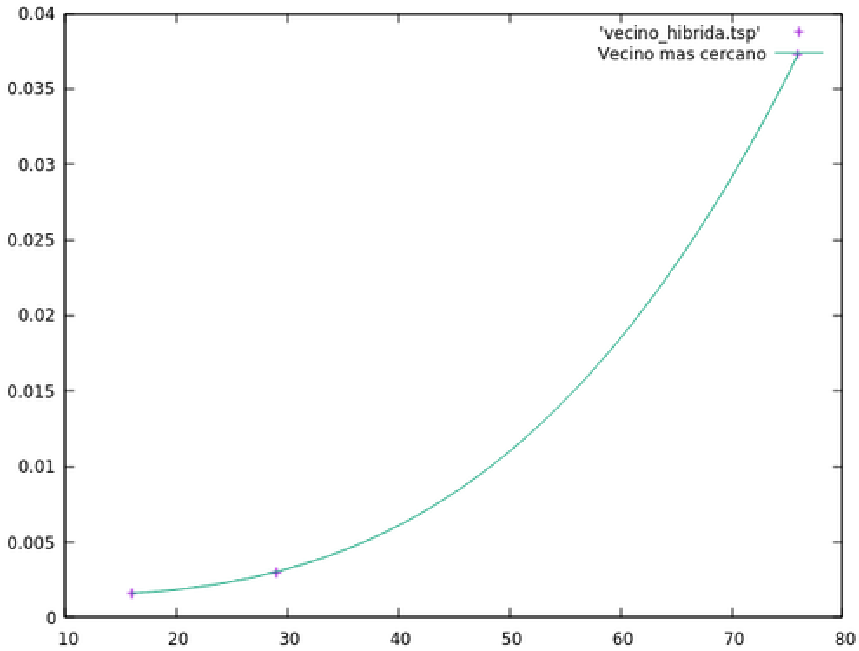
```
 return suma;
};
```

Analizando la eficiencia teórica de este algoritmo, vemos que, resolviendo los sumatorios de los bucles, obtenemos $(n^3-n^2)/2$, por tanto: $T(n) \in O(n^3)$.

DESARROLLO

2.Ejercicio 2

2.1 Algoritmo vecino más cercano



Al estudiar la eficiencia empírica, vemos que la curva que mejor se ajusta a nuestro datos, es una polinomial, de orden 3, que coincide con lo esperable según la eficiencia teórica.

Para la eficiencia híbrida, utilizamos gnuplot para obtener las constantes ocultas de la función: $9.12106e\text{-}08 \cdot n^3 - 7.26387e\text{-}07 \cdot n^2 + 0.00141436$.

DESARROLLO

2.2 Algoritmo inserción

```
void TrianguloInicial (Punto2D *puntos, int dim, int &norte, int
&oeste, int &este) {

 Punto2D p_norte; Punto2D p_este;
 Punto2D p_oeste(1000000, -1);
 oeste=0; este=0; norte=0;

 for (int i=0; i<dim; i++) {
  if (puntos[i].get_x()<p_oeste.get_x()) {
   p_oeste=puntos[i];
   oeste=i;
  }
  if (puntos[i].get_x()>p_este.get_x()) {
   p_este=puntos[i];
   este=i;
  }
  if (puntos[i].get_y()>p_norte.get_y()) {
   p_norte=puntos[i];
   norte=i;
  }
 }

 int oeste_ant, este_ant;
 oeste_ant=oeste;
 este_ant=este;

 if(norte==oeste) {
  p_oeste.set_both(10000,-1);
  for (int i=0; i<dim; i++) {
   if (i!=oeste_ant && puntos[i].get_x()<p_oeste.get_x()) {
    p_oeste=puntos[i];
    oeste=i;
   }
  }
 }
 if(norte==este) {
  p_este.set_both(-1,-1);
  for (int i=0; i<dim; i++) {
   if (i!=este_ant && puntos[i].get_x()>p_este.get_x()) {
    p_este=puntos[i];
    este=i;
   }
  }
 }
};
```

Guardamos los tres puntos que conforman el recorrido inicial, así como su índice.

Comprobamos todos los puntos.
Este for se recorre n-1 veces.

Vamos comparando cada punto con el punto más al norte encontrado hasta el momento. Si el punto que estamos mirando está más arriba, se convierte en el nuevo norte. Hacemos lo mismo con el oeste y el este.

Si el norte coincide con alguno de los otros dos puntos, repetimos el procedimiento, sin tomar el punto norte como opción. Así nos aseguramos de tener un triángulo.
Estos for se recorren n-1 veces.

Resolviendo los sumatorios de este algoritmo, nos da: a(n-1), por lo tanto: $T(n) \in O(n)$.

DESARROLLO

2.2 Algoritmo inserción

```
int Insercion(vector<int> &recorrido, int**matriz, int dim, int norte, int oeste, int este) {

 recorrido.push_back(norte); recorrido.push_back(oeste);
 recorrido.push_back(este);  recorrido.push_back(norte);
 int suma=0;
 suma+=matriz[norte][oeste]+matriz[oeste][este]+matriz[este][norte];
 bool cogido=false; // Indica si el punto visitado ya está en el recorrido
```

Guardamos el triángulo inicial en el recorrido y añadimos sus distancias a la suma.

```
 int copia1_suma; // Suma total si insertásemos el nuevo nodo
 auto it_at=recorrido.begin(); // Iterador al
 auto it_post = recorrido.begin();

 for (int i=0; i<dim; i++) {
  for (auto it=recorrido.begin(); it!=recorrido.end() && !cogido; ++it) {
   if (i==(*it))
    cogido = true;
  }
```

Comprobamos todos los puntos.
Este for se recorre n-1 veces.

Vamos comprobando si ya se ha cogido el elemento. Este for se recorre j veces, siendo j el tamaño del vector recorrido -1.

```
  if (!cogido) {
   auto it=recorrido.begin(); // Iterador al segundo punto entre el que queremos insertar en el recorrido (para comprobar)
   auto anterior=it; // Iterador al primer punto entre el que queremos insertar en el recorrido (para comprobar)
   ++it;
   auto min_it=it; // Iterador al punto donde finalmente vamos a insertar el nuevo punto (será el definitivo)
   int dist_min=suma - matriz[*it][*anterior] + matriz[i][(*it)]+matriz[i][(*anterior)];
   ++it; ++anterior;

   for (it; it!=recorrido.end(); ++it) {
    copia1_suma=suma-matriz[*it][*anterior]+matriz[i][(*it)]+matriz[i][(*anterior)];
    if (copia1_suma <= dist_min) {
     dist_min = copia1_suma;
     min_it=it;
    }
    ++anterior;
   }
```

Comprobamos cuanto aumentaría la suma si insertásemos el punto actual entre cada una de las parejas. Si la suma total con la posibilidad de inserción actual es menor que todas las anteriores, esta pasa a ser la mínima.
Este for se recorre j-2 veces, siendo j el tamaño del vector recorrido -1.

```
   it_at=min_it; --it_at; it_post=min_it; ++it_post;
   suma-=(matriz[*it_at][*min_it]);
   recorrido.insert(min_it, i);
   suma+=(matriz[*min_it][*it_at] + matriz[*min_it][*it_post]);
  }
  cogido=false;
 } return suma; };
```

Cuando ya hemos encontrado la mejor opción de inserción, restamos la distancia entre los dos puntos entre los que vamos a insertar el nuevo punto, lo insertamos, y sumamos las dos nuevas distancias, de los dos puntos anteriores a este nuevo.

DESARROLLO

2.2 Algoritmo inserción

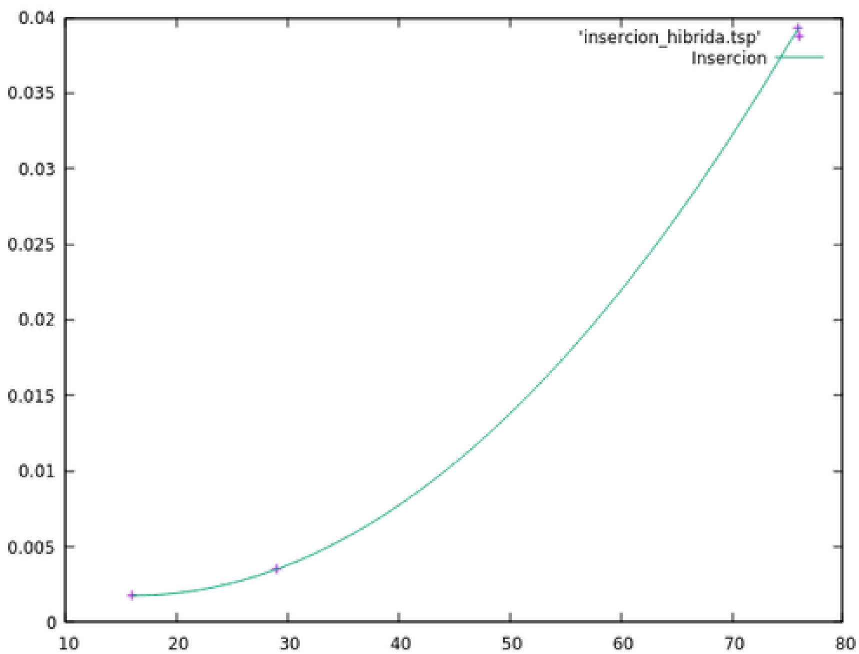
```
for (it; it!=recorrido.end(); ++it) {
    copia1_suma=suma-matriz[*it][*anterior]+matriz[i][(*it)]+matriz[i][(*anterior)];
    if (copia1_suma <= dist_min) {
     dist_min = copia1_suma;
     min_it=it;
    }
    ++anterior;
   }
```

Comprobamos cuanto aumentaría la suma si insertásemos el punto actual entre cada una de las parejas. Si la suma total con la posibilidad de inserción actual es menor que todas las anteriores, esta pasa a ser la mínima.
Este for se recorre j-2 veces, siendo j el tamaño del vector recorrido -1.

```
   it_at=min_it;
   --it_at;
   it_post=min_it;
   ++it_post;

   suma-=(matriz[*it_at][*min_it]);
   recorrido.insert(min_it, i);
   suma+=(matriz[*min_it][*it_at] + matriz[*min_it][*it_post]);
  }
  cogido=false;
 } return suma; };
```

Cuando ya hemos encontrado la mejor opción de inserción, restamos la distancia entre los dos puntos entre los que vamos a insertar el nuevo punto, lo insertamos, y sumamos las dos nuevas distancias, de los dos puntos anteriores a este nuevo.

Analizando la eficiencia teórica de este algoritmo, vemos que, resolviendo los sumatorios de los bucles, obtenemos n^2-n, por tanto: $T(n) \in O(n^2)$.



Al estudiar la eficiencia empírica, vemos que la curva que mejor se ajusta a nuestro datos, es una polinomial, de orden 2, que coincide con lo esperable según la eficiencia teórica.

Para la eficiencia híbrida, utilizamos gnuplot para obtener las constantes ocultas de la función: $1.05089e\text{-}05 \cdot n^2 - 0.000340899 \cdot n + 0.00455511$.

DESARROLLO

2.3 Algoritmo propio

```
vector<int> AlgoritmoPropio (int **matriz, int dim) {

 vector<int> minimo; // Guarda el recorrido mínimo según nuestro algoritmo
 srand(time(0));
 int comienzo=rand()%dim; // Punto por el que comienza el recorrido
 minimo.push_back(comienzo);
 minimo.push_back(comienzo);
```

Añadimos el punto por el que vamos a empezar 2 veces, porque también es el punto final

```
 int dist_min=10000; // Distancia mínima encontrada
 int valor=0;
 int min1;
 bool cogido=false;
 auto iterador=minimo.begin();
 auto menos=minimo.begin();
 auto mas=minimo.begin();
```

Mientras queden puntos por recorrer, comprobamos qué otro punto que NO esté ya en el recorrido, es el más cercano al que estamos analizando.

```
 while (minimo.size()<=dim) {
  int dist_min=10000;

  for (auto it2=minimo.begin(); it2!=minimo.end(); ++it2) {
   valor = *it2;
   for (int i=0; i<dim; i++) {
    for (auto it=minimo.begin(); it!=minimo.end() && !cogido; ++it) {
     if (i==(*it))
      cogido = true;
    }

 if (!cogido && matriz[i][valor]<dist_min) {
  dist_min=matriz[i][valor];
  min1=i; /
```

Vamos guardando la distancia mínima encontrada, así como el punto que más cerca está del actual, que podamos insertar.

```
 iterador=it2; // Iterador que apunta al punto del recorrido que estamos analizando
```

DESARROLLO

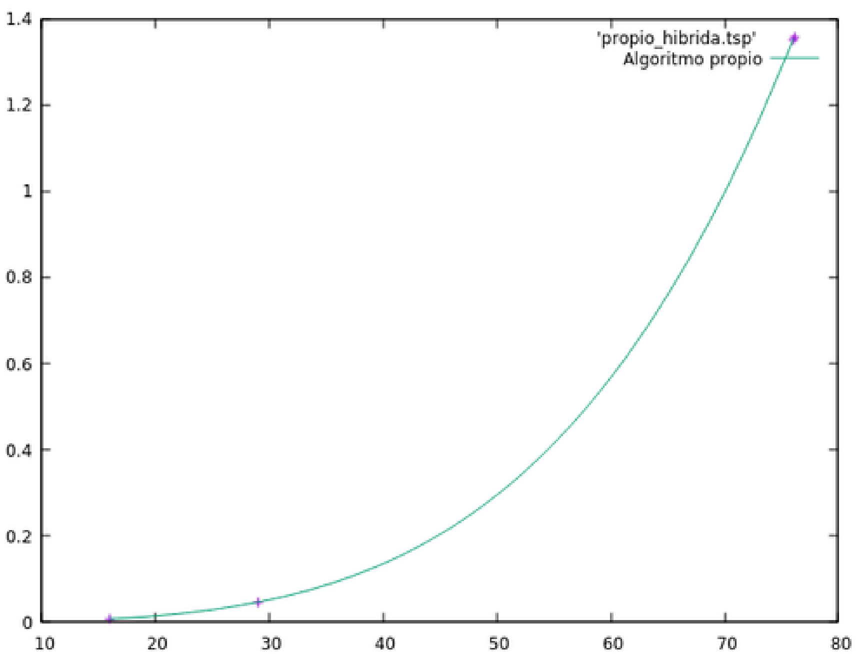
2.3 Algoritmo propio

```
mas=++iterador; // Iterador que apunta al siguiente punto del recorrido al que estamos analizando (para insertar después del que estamos analizando)
iterador=it2;

if (iterador!=minimo.begin()) {
menos=--iterador;

if (matriz[min1][*menos]<matriz[min1][*mas]) /
iterador=it2;
else
iterador=mas; /
}
else
iterador=mas;

}
cogido=false;
}
}
minimo.insert(iterador, min1);
}

return minimo;
}
```

Si el punto que estamos analizando no es el primero del recorrido, miramos si está mas cerca del punto anterior del recorrido al que estamos analizando o del siguiente, para saber donde conviene más insertarlo.

Insertamos el nuevo punto

Resolviendo los sumatorios de los 4 bucles anidados de este algoritmo, obtenemos $(n^4+n^3-n^2)/2$. Por lo tanto, $T(n)\in O(n^4)$.



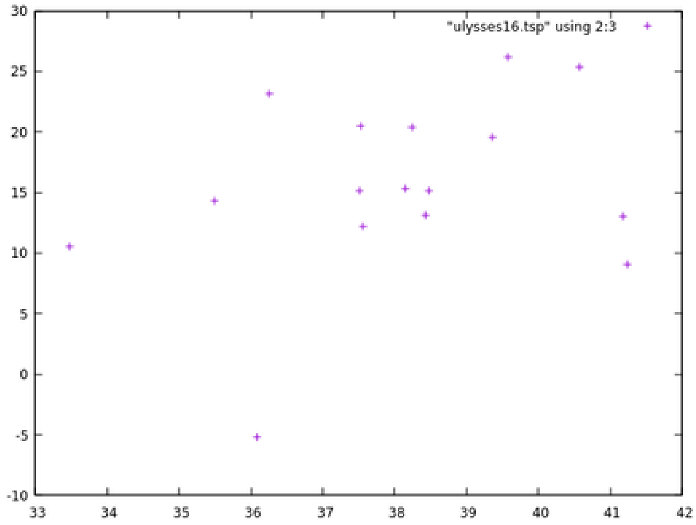
Al estudiar la eficiencia empírica, vemos que la curva que mejor se ajusta a nuestro datos, es una polinomial, de orden 4, que coincide con lo esperable según la eficiencia teórica.

Para la eficiencia híbrida, utilizamos gnuplot para obtener las constantes ocultas de la función: $3.16122e\text{-}08 \cdot n^4 + 4.71899e\text{-}07 \cdot n^3 + 1.54389e\text{-}05 \cdot n^2$.

DESARROLLO

2.4 Comparaciones

Resultados para: ulysses16.tsp

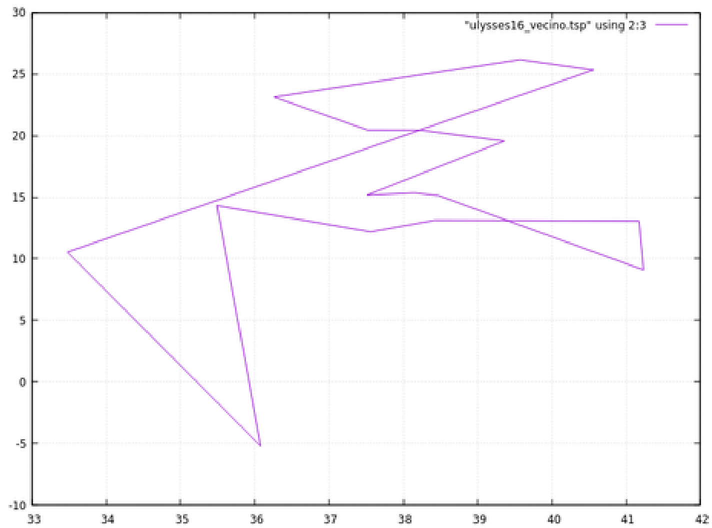


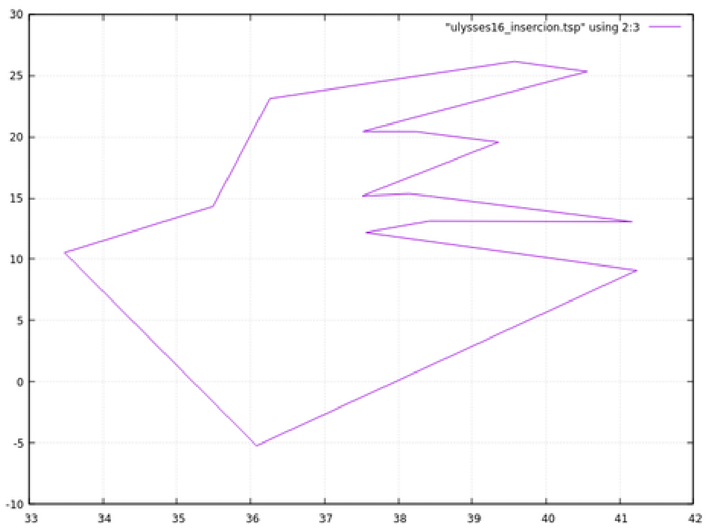
Puntos originales

Vecino más cercano

Recorrido: 15 6 7 10 9 12 13 14 16 1 8 4 2 3 5 11 15

Longitud ruta: 83





Inserción

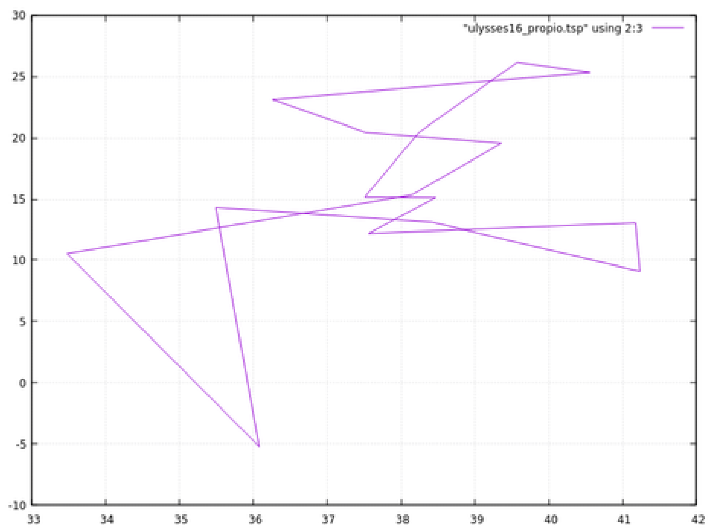
Recorrido: 2 4 15 5 11 9 6 7 10 12 13 14 16 1 8 3 2

Longitud ruta: 67

Algoritmo propio

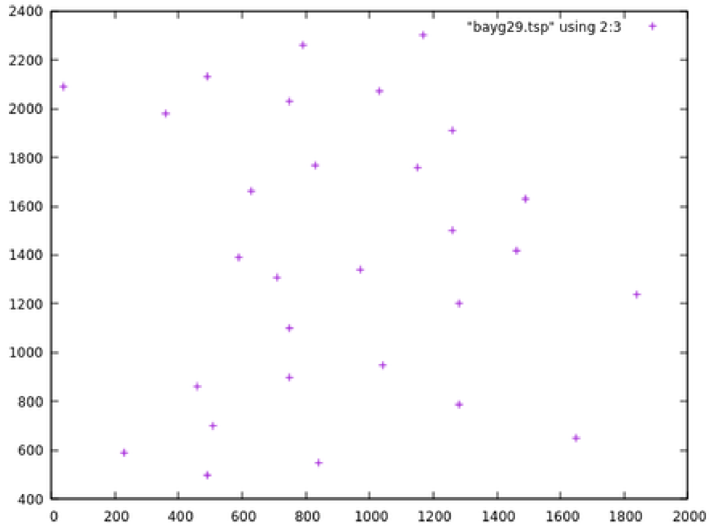
Recorrido: 15 11 5 13 16 8 4 3 2 1 14 12 6 10 9 7 15

Longitud ruta: 72

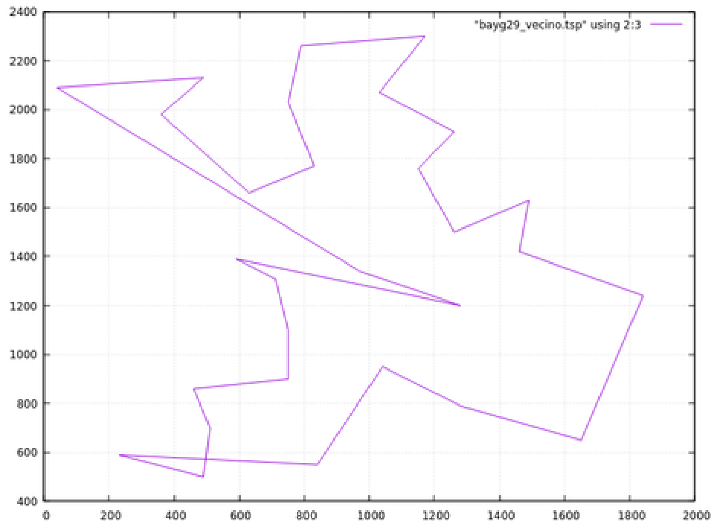


COMPARACIONES

Resultados para: bayg29.tsp



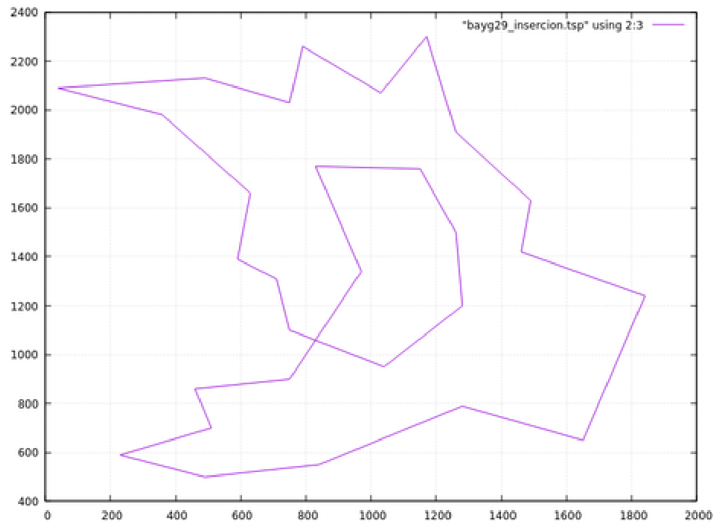
Puntos originales



Vecino más cercano

Recorrido: 22 14 18 15 4 10 20 2 21 5 9 6 12 28 1 24 27 8 16 13 19 25 7 23 11 17 29 26 3 22

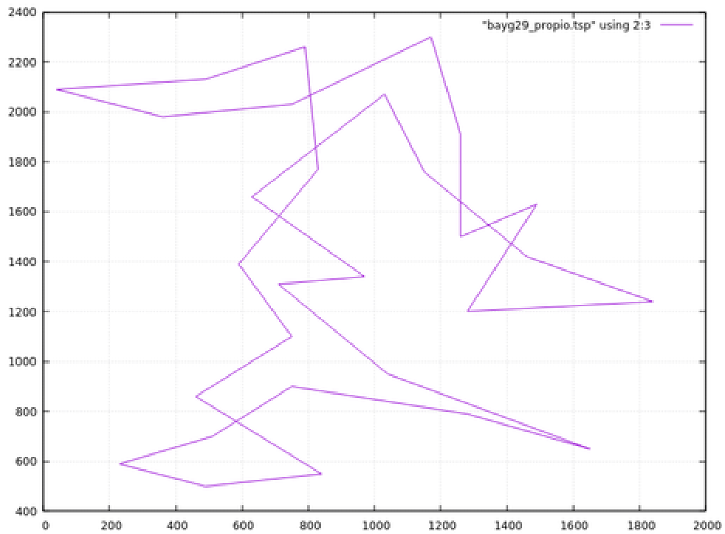
Longitud ruta: 12129



Inserción

Recorrido: 12 6 9 5 26 3 29 2 20 10 4 19 16 24 1 21 13 15 18 14 17 22 11 25 7 23 27 8 28 12

Longitud ruta: 9735



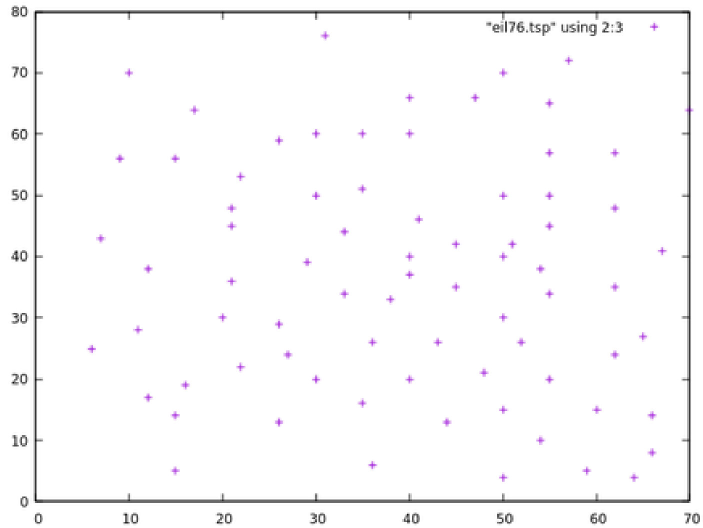
Algoritmo propio

Recorrido: 22 11 17 14 15 25 7 19 10 13 2 6 1 27 23 16 8 24 28 12 5 29 3 26 9 21 20 4 18 22

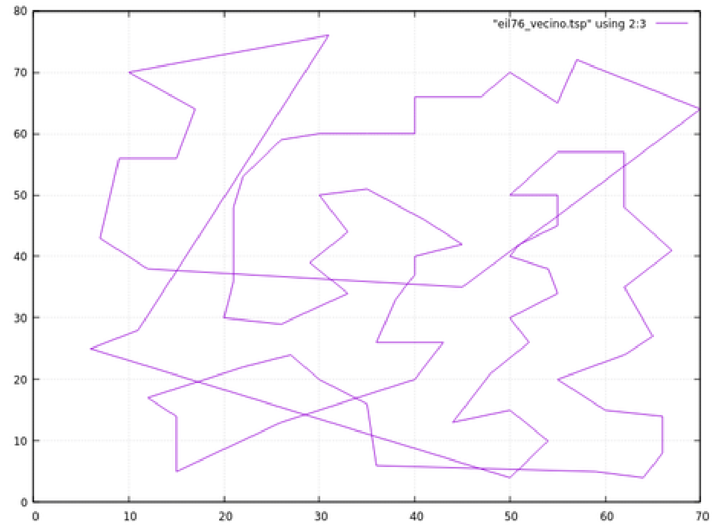
Longitud ruta: 12318

COMPARACIONES

Resultados para: eil76.tsp



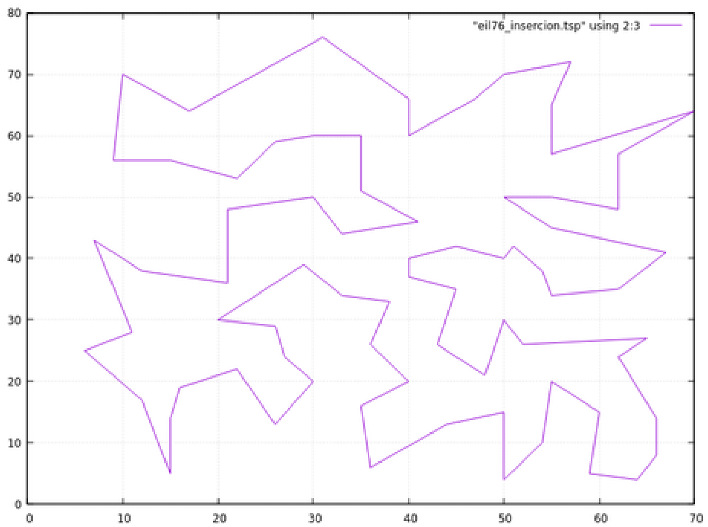
Puntos originales



Vecino más cercano

Recorrido: 69 36 47 21 48 29 45 27 52 34 46 8 35 7 53 14 19 54 13 57 15 5 37 20 70 60 71 61 28 62 73 1 43 41 42 64 22 74 30 2 68 75 76 67 26 12 40 17 51 6 33 63 16 3 44 32 9 39 72 58 10 38 65 11 66 59 4 49 24 18 50 25 55 31 23 56 69

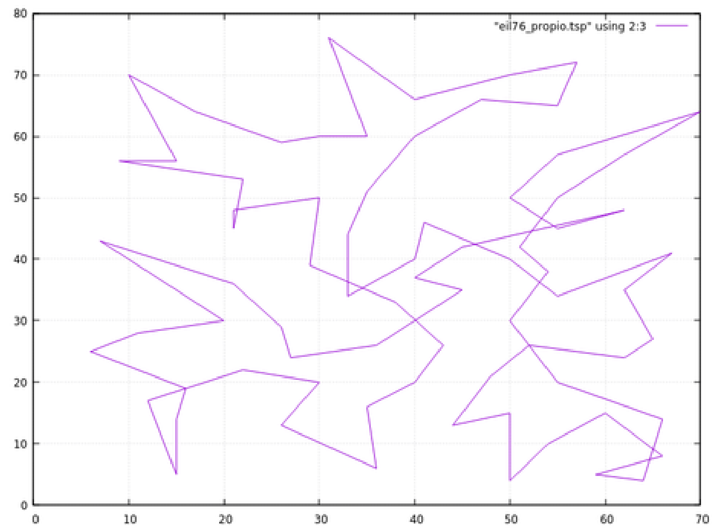
Longitud ruta: 667



Inserción

Recorrido: 31 25 55 18 50 32 9 39 72 12 26 17 40 44 3 16 49 24 23 56 41 64 42 43 1 22 62 73 33 63 51 6 68 2 74 28 61 21 47 69 36 5 37 71 60 70 20 15 57 29 45 48 30 4 75 76 67 34 46 52 27 13 54 8 7 35 19 14 59 53 11 66 65 38 58 10 31

Longitud ruta: 581



Algoritmo propio

Recorrido: 8 19 67 75 4 2 73 33 16 24 49 63 23 56 43 42 64 41 1 62 22 61 28 74 30 68 51 40 44 3 32 18 50 55 25 9 39 72 31 10 65 66 11 38 58 12 17 6 76 26 34 27 54 13 57 15 29 48 21 47 69 36 37 70 71 60 20 5 45 52 46 35 14 59 53 7 8

Longitud ruta: 667

COMPARACIONES

Como vemos en las rutas anteriores y sus longitudes, el algoritmo de inserción es el que genera la ruta más corta en los tres casos. En cuanto a los otros dos algoritmos, depende del número de puntos que haya que ordenar y del punto por el que se empiece, pero podría parecer que para tamaños más pequeños, es el algoritmo propio el que genera la ruta más corta, y para mayores tamaños, están bastante igualados.

Comparación de tiempos (en segundos)

Vecino más cercano:
Ulysses: 0.001602 - Bayg: 0.003028 - Eil: 0.037258

Inserción:
Ulysses: 0.001791 - Bayg: 0.003507 - Eil: 0.039346

Algoritmo propio:
Ulysses: 0.007957 - Bayg: 0.046852 - Eil: 1.35098

Como podemos comprobar, el algoritmo que menos tarda en ordenar las ciudades, es el del vecino más cercano en los tres casos, seguido del de inserción, y del algoritmo propio. Dentro de cada algoritmo, el fichero que más tiempo conlleva es el que ordena un mayor número de puntos, como era de esperar.

CONCLUSIÓN

Para terminar con el trabajo, vamos a describir una breve conclusión.

Para el primer ejercicio hemos podido observar que, incluso para dos problemas aparentemente parecidos como maximizar el número de contenedores y el número de toneladas, el algoritmo greedy no siempre es la solución más viable a la hora de atajar el problema, debido a su naturaleza de encontrar el mejor caso actual sin preocuparse por el futuro.

En el caso del problema del viajante del comercio, podemos observar que los algoritmos Greedy no dan siempre la mejor solución, pues el único algoritmo que ha obtenido el recorrido mínimo ha sido el de inserción. Sin embargo, aportan una solución óptima y son fáciles y rápidos de implementar. Por tanto, es un tipo de algoritmo muy a tener en cuenta para problemas con una dificultad considerable.

BIBLIOGRAFÍA

https://www.geeksforgeeks.org/vector-in-cpp-stl/

https://www.geeksforgeeks.org/graph-and-its-representations/
Matriz de adyacencia

https://www.geeksforgeeks.org/traveling-salesman-problem-tsp-implementation/

https://www.cplusplus.com/reference/vector/vector/reserve/
https://parzibyte.me/blog/2019/06/20/agregar-elemento-arreglo-vector-cpp/