Práctica 3

Pablo Baeyens Fernández
Antonio Checa Molina
Iñaki Madinabeitia Cabrera
José Manuel Muñoz Fuentes
Darío Sierra Martínez

Algorítmica

Índice

1. Contenedores en un barco

El problema consiste en rellenar un buque con carga limitada (K) con un conjunto de contenedores $c_1, \ldots, c_n$ con pesos $p_1, \ldots, p_n$ con un cierto objetivo.

La implementación de los algoritmos es de la forma:

Entrada: Vector de pesos de los contenedores, p y capacidad total K

Salida: Vector con los contenedores elegidos

Utilizaremos también una estructura de datos simple llamada `Cont` que almacena un contenedor:

```
struct Cont{
  int id;
  peso_t peso;

  Cont(int i, peso_t p){
    id = i;
    peso = p;
  }
};
```

1.1. Maximizar el número de contenedores cargados

Para este caso nos basta coger siempre el contenedor de menor peso disponible de entre los que no hemos elegido ya. Para hacer que el algoritmo sea eficiente emparejamos cada elemento del vector de entrada con su posición (el contenedor que representa) y ordenamos este vector en función de los pesos:

```
vector<int> max_num_conts(const vector<peso_t> p, peso_t K){
  vector<Cont> conts = a_cont(p);

  sort(conts.begin(), conts.end(), menor);

  vector<int> elegidos;

  for(int i = 0; i < conts.size() && conts[i].peso <= K; i++){
    elegidos.push_back(conts[i].id);
    K -= conts[i].peso;
  }

  return elegidos;
}
```

La función `menor` nos permite comparar dos contenedores según su peso, indicando cuál es menor.

La eficiencia del algoritmo es $O(n \log(n))$, ya que la operación de ordenado es $O(n \log(n))$ y el bucle posterior es de eficiencia lineal.

1.1.1. Optimalidad de la solución

Este criterio (coger el de menor peso hasta que no quepan más) **siempre obtiene la mejor solución** (la solución con un mayor número de contenedores). Llamemos a nuestra solución $o_1, \ldots, o_k$ y sea $s_1, \ldots, s_m$ otra solución cualquiera. Supongamos que $m > k$. Podemos asumir sin pérdida de generalidad que las soluciones están ordenadas por pesos de menor a mayor.

Como los pesos de nuestra solución son mínimos, es claro que $\forall i < k : o_k \leq s_k$. Por tanto también tendríamos como posible la solución $o_1, \ldots o_k, s_{k+1}, \ldots s_m$. Sea o_{k+1} el contenedor que no está en nuestra solución que tenga el menor peso. Es claro que $o_{k+1} \leq \sum_{i>k} s_i$, por lo que tendríamos que $o_1, \ldots, o_{k+1}$ es solución.

Esto es una contradicción ya que por construcción nuestra solución toma los contenedores de peso mínimo hasta que añadir uno más suponga sobrepasar la capacidad. Por tanto nuestra solución es la solución con el mayor número de contenedores posibles.

1.2. Maximizar el número de toneladas cargadas

En este caso empleamos como estrategia *greedy* coger los contenedores en orden decreciente de peso: empezamos con el contenedor más pesado que quepa en el buque y continuamos ordenadamente añadiendo el contenedor más pesado de entre los que quepan.

El código es muy similar a la solución anterior, salvo que cambiamos la condición que ordena la lista inicialmente para ordenarla de forma decreciente en función del peso y necesitamos realizar, en cada paso, la comprobación de que el contenedor que queremos añadir no exceda la capacidad restante del buque:

```
vector<int> max_peso_greedy(const vector<peso_t> p, peso_t capacidad){
  vector<Cont> conts = a_cont(p);
  sort(conts.begin(), conts.end(), mayor);
  vector<int> elegidos;

  for(int i = 0; i < conts.size() && capacidad > 0; i++){
    if(conts[i].peso <= capacidad){
      elegidos.push_back(conts[i].id);
      capacidad -= conts[i].peso;
    }
  }
  return elegidos;
}
```

Para ver que realmente el algoritmo greedy, aunque parezca muy simple y poco eficaz, es realmente útil, decidimos hacer el algoritmo óptimo. Este otro algoritmo recorre todas las posibles combinaciones de contenedores y se queda con la que más carga meta en el buque. Colocamos a continuación el código:

```
vector<Cont> bruto_sub(const vector<Cont> p, peso_t K){
  if(p.empty() || K == 0)
    return vector<Cont>();

  peso_t max = 0;
  vector<Cont> max_v;
  for(vector<Cont>::const_iterator it = p.cbegin(); it != p.cend() &&
     max < K; ++it){
    vector<Cont>::const_iterator next(it);
    vector<Cont> orig_copy(++next, p.cend());   // Copia todos los
       posteriores
    vector<Cont> v_candidato;
    if(it->peso > K)
      v_candidato = bruto_sub(orig_copy, K);
```

```
    else {
      v_candidato = bruto_sub(orig_copy, K-it->peso);
      v_candidato.push_back(*it);    // Si usamos listas, mejor usar
         push_front
    }
    peso_t suma1 = suma(v_candidato);
    if(suma1 > max && suma1 <= K){
      max = suma1;
      max_v = v_candidato;
    }
  }
  return max_v;
}
```

Como es fácil de ver, este algoritmo es $O(n!)$, ya que tiene que recorrer todas las combinaciones y quedarse con el máximo, y sabemos que hay exactamente $n!$. Por lo tanto, aunque sea un algoritmo que siempre encuentre el óptimo, es claramente lento en problemas un poco grandes.

Por otro lado, uno esperaría que la diferencia en carga de los dos algoritmos fuera apreciable, ya que el greedy es muy bruto, mientras que el algoritmo de fuerza bruta es siempre lo más eficaz que puede ser. Para nuestra sorpresa, cuando hicimos las gráficas no se puede notar mucha diferencia entre las cargas de los dos algoritmos, mientras que en tiempo sí que se observa el contraste:

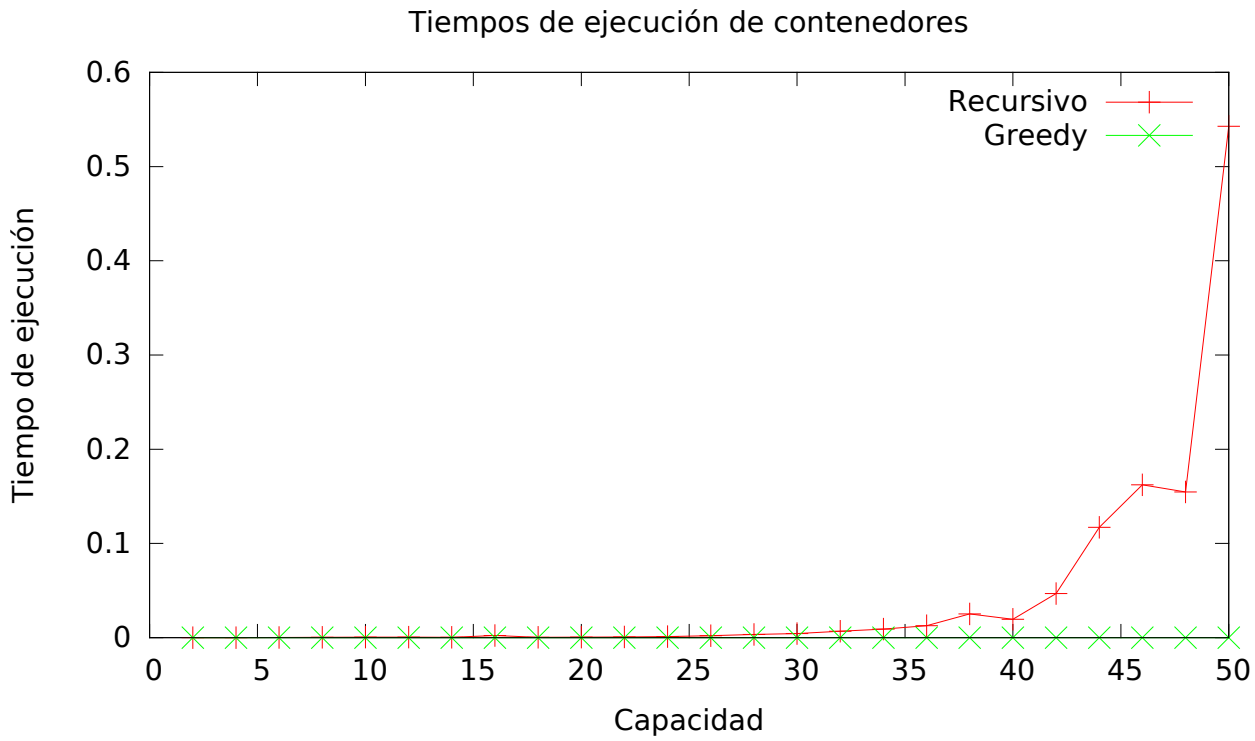

Figura 1: Comparativa de tiempo de los dos algoritmos

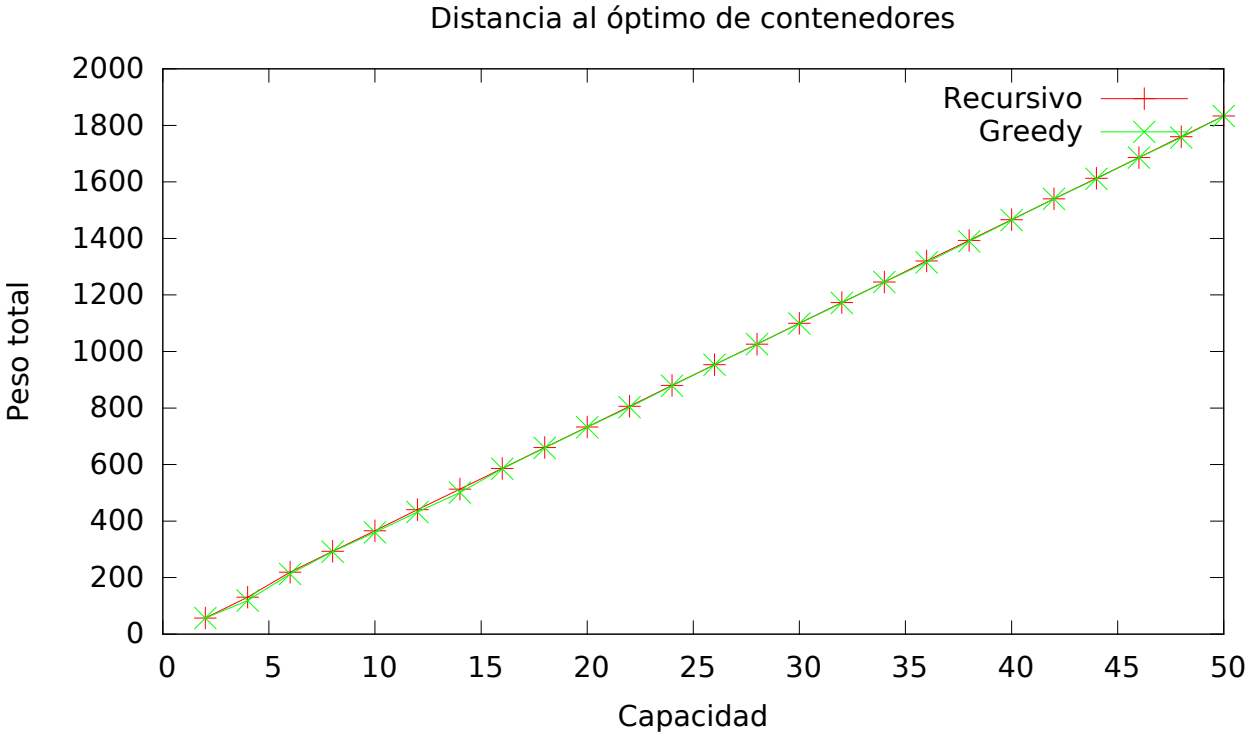

Figura 2: Comparativa de carga total de los dos algoritmos

2. El problema del viajante de comercio

El problema del viajante de comercio consiste en hallar el recorrido con distancia mínima en un conjunto de ciudades que pase por todas las ciudades y regrese al punto inicial.

La implementación de los algoritmos es de la forma:

Entrada: Ficheros con ciudades indicadas como puntos en el plano según sus coordenadas.

Salida: `vector<int>` con el orden en el que se recorren las ciudades.

En la salida nos ahorraremos repetir el primer nodo al final del recorrido.

Para todos los algoritmos utilizaremos una estructura de datos que nos permita manejar el problema: la clase `Grafo` (en el fichero `grafo.h`). Un grafo consta de:

- Una **cantidad de nodos**, almacenada en el atributo `nodos`.
- Una **matriz de pesos**, almacenada en el vector `lados`.

La interfaz nos permite acceder y modificar estos datos con mayor facilidad. Mediante los métodos `setPeso` y `peso` accederemos al peso de un lado del grafo, y el método `pesosDesdeCoordenadas` nos permite inicializar el grafo utilizando el formato de datos en el que aparece el problema: calculando la distancia entre cualesquiera dos ciudades y añadir esta como peso de ese lado:

```
void pesosDesdeCoordenadas(double c[][2]) {
   for (int i = 0; i < nodos-1; i++)
      for (int j = i+1; j < nodos; j++)
         setPeso(i, j, dist(c[i],c[j]));
}
```

Adicionalmente, la función `longitud` calcula la longitud de un camino dado, teniendo en cuenta la distancia del último nodo al primero:

```
peso_t longitud(const std::vector<int>& ids, const Grafo<peso_t>& g) {
  peso_t l = g.peso(ids.back(), ids.front());
  for (int i = 1; i < ids.size(); i++)
    l += g.peso(ids[i], ids[i-1]);
  return l;
}
```

Siendo `peso_t` el tipo de dato con el que se manejen las distancias entre nodos. Según el enunciado del problema (que pide que se redondee la distancia euclídea al entero más próximo), será de tipo `int`.

2.1. Algoritmos

2.1.1. Vecino más cercano

Utilizando la estructura de datos explicada anteriormente, la resolución del problema utilizando la heurística del vecino más cercano quedará:

```
vector<int> tsp_1(const Grafo<peso_t>& g, const double coordenadas[][2]
    = 0) {
  vector<int> trayecto(1, 0);
  list<int> disponibles;
  for (int i = g.numNodos()-1; i > 0; i--)
    disponibles.push_front(i);

  while(!disponibles.empty()) {
    list<int>::iterator it = disponibles.begin(), cercano = it, fin =
       disponibles.end();
    int actual = trayecto.back();
    peso_t d_actual = g.peso(actual, *cercano);
    for (++it; it != fin; ++it) {
      peso_t d_candidato = g.peso(actual, *it);
      if (d_candidato < d_actual) {
        cercano = it;
        d_actual = d_candidato;
      }
    }

    trayecto.push_back(*cercano);
    disponibles.erase(cercano);
  }

  return trayecto;
}
```

Tomamos como ciudad inicial el nodo 0 y almacenamos en la lista `disponibles` las ciudades no visitadas. A continuación, mientras queden ciudades disponibles, recorremos la lista buscando aquella ciudad con distancia mínima a la última, la cual añadimos al trayecto y eliminamos de la lista de disponibles.

2.1.2. Estrategias de inserción

Para las estrategias de inserción realizamos el trabajo en 3 etapas. En primer lugar hallamos las 3 ciudades que conforman el **recorrido inicial**. Para ello basta tomar las ciudades como coordenadas en el plano y tomar las ciudades más al norte, este y oeste (tomando los puntos con menor y mayor coordenada x y mayor coordenada y):

```
vector<int> triangulo_inicial(const double coordenadas[][2], int n) {
  vector<int> iniciales = {0,1,2};
  if (coordenadas[0][1] < coordenadas[1][1])
     swap(iniciales[0], iniciales[1]);
  if (coordenadas[iniciales[0]][1] < coordenadas[2][1])
     swap(iniciales[0], iniciales[2]);

  if (coordenadas[iniciales[1]][0] > coordenadas[iniciales[2]][0])
     swap(iniciales[1], iniciales[2]);
  for (int i = 3; i < n; i++)
     if (coordenadas[i][1] > coordenadas[iniciales[0]][1]) {
        if (coordenadas[iniciales[0]][0] < coordenadas[iniciales
           [1]][0])
           iniciales[1] = iniciales[0];
        else if (coordenadas[iniciales[0]][0] > coordenadas[iniciales
           [2]][0])
           iniciales[2] = iniciales[0];
        iniciales[0] = i;
     }
     else if (coordenadas[i][0] < coordenadas[iniciales[1]][0])
        iniciales[1] = i;
     else if (coordenadas[i][0] > coordenadas[iniciales[2]][0])
        iniciales[2] = i;

  return iniciales;
}
```

En el caso en el que el punto de mayor coordenada y sea también el de mayor (o menor) coordenada x cogeremos el segundo con mayor (o menor) coordenada x.

Una vez conseguido un triángulo en el grafo lo suficientemente grande lo siguiente es añadir el resto de ciudades. Utilizaremos el siguiente procedmiento a la hora de insertar:

- Recorrer la lista de nodos disponibles.
- Para cada nodo seleccionar el índice donde, de insertarse, aumente lo menos posible el peso del grafo.
- Insertamos en dicho índice el nodo que menos aumente el peso total.

```
vector<int> tsp_2(const Grafo<peso_t>& g, const double coordenadas
    [][2]){
  vector<int> triangulo = triangulo_inicial(coordenadas, g.numNodos());
       // Inicializamos el vector con las ciudades que forman el mayor
     triángulo en el grafo.

  list<int> disponibles;
  for (int i = 0; i < g.numNodos(); i++)
    if (i != triangulo[0] && i != triangulo[1] && i != triangulo[2])
       disponibles.push_front(i);

  list<int> trayecto_l(triangulo.begin(), triangulo.end());
  trayecto_l.push_back(trayecto_l.front());
  while (!disponibles.empty()){
    list<int>::iterator minimo = disponibles.begin();
    pair<int,list<int>::iterator> PesoIndiceMin = PesoNuevoCircuito(*
       minimo,trayecto_l,g);

    for(list<int>::iterator it = ++disponibles.begin(); it !=
       disponibles.end(); it++){
      pair<int,list<int>::iterator> PesoIndicetmp = PesoNuevoCircuito(*
         it,trayecto_l,g);
      if(PesoIndiceMin.first > PesoIndicetmp.first){
        minimo = it;
        PesoIndiceMin=PesoIndicetmp;
      }
    }

    trayecto_l.insert(PesoIndiceMin.second,*minimo);
    disponibles.erase(minimo);
  }

  trayecto_l.pop_back();
  vector<int> trayecto(trayecto_l.begin(), trayecto_l.end());
  return trayecto;
}
```

El verdadero corazón del algoritmo reside en saber cómo varía el peso del grafo al insertar un nodo. Consideremos lo siguiente:

Sea y_0 el nodo a insertar entre los nodos n_i y n_{i-1}, que se encuentran seguidos, separados por un segmento $\overline{n_i\ n_{i-1}}$. Al hacer esta inserción el segmento que une los dos nuevos nodos deja de ser relevante en la longitud del recorrido y aparecen dos segmentos nuevos: $\overline{y_0\ n_i}$ y $\overline{y_0\ n_{i-1}}$.

Por tanto, el peso de la inserción de n_0 en el indice i es:

$$\text{PesoInsercion}_i = \text{Peso}(\overline{y_0\ n_i}) + \text{Peso}(\overline{y_0\ n_{i-1}}) - \text{Peso}(\overline{n_i\ n_{i-1}})$$

Primero buscamos cuál es el mínimo incremento de insertar el nodo n_i y lo comparamos con los demás. Insertamos el nodo cuyo peso mínimo añadido sea menor en su índice correspondiente.

2.1.3. Colonia de hormigas

Como solución adicional propuesta por el equipo utilizamos una heurística basado en colonias de hormigas. Este algoritmo se inspira en la comunicación por feromonas de una colonia de hormigas para encontrar el camino óptimo hacia una fuente de comida.

Implementamos la Colonia (en el fichero `colonia.h`) mediante una clase. Una Colonia consta de:

- Un grafo de **distancias** entre las ciudades.
- Un grafo con las **feromonas** de cada camino, inicialmente a un valor arbitrario.
- Una serie de constantes $\alpha, \beta, \rho, \xi, C, P, I$.

Las constantes controlan el comportamiento del algoritmo:

α es el peso que tienen las feromonas a la hora de decantarse por un camino u otro.

β es el peso que tienen la distancias en la circunstancia anterior.

ρ es el coeficiente de evaporación de las feromonas.

ξ es el coeficiente de debilitamiento de las feromonas.

C determina cuánta feromona se añade a un camino.

P es un flotante entre 0 y 1. Durante el progreso de un recorrido, hay una probabilidad de $1 - P$ de que la hormiga opte por elegir el siguiente nodo de forma no aleatoria.

I es la cantidad de feromona inicial en cada nodo.

Este algoritmo genera una serie de ciclos. En cada ciclo se empieza en un nodo al azar y se añaden nodos adicionales, formando un trayecto que termina cerrándose, decidiendo el siguiente nodo en función de la distancia al nodo actual y de la cantidad de feromona depositada en la arista entre ambos nodos.

Se define una fórmula que determina la probabilidad de escoger un nodo. Sea i el nodo actual, j un nodo candidato y K la lista de nodos candidatos, la probabilidad de que se escoja el nodo j a continuación del i es:

$$P_j^i = \frac{(\tau_{i,j})^\alpha \cdot (1/d(i,j))^\beta}{\sum_{k \in K} (\tau_{i,k})^\alpha \cdot (1/d(i,k))^\beta}$$

Donde $\tau_{a,b}$ denota la cantidad de feromona depositada en la arista que une a con b, y $d(a,b)$ es la distancia entre dos nodos a y b.

En cada nodo, hay una probabilidad de P de que se escoja un elemento a partir de la lista de probabilidades acumuladas (generando un flotante entre 0 y 1 y buscando la posición donde es superado), y una probabilidad de $1 - P$ de que se tome directamente el elemento con mayor probabilidad de ser escogido. En la práctica no calcularemos el denominador de la expresión anterior y simplemente normalizaremos el flotante respecto del último elemento de la lista de probabilidades acumuladas.

Cada vez que una serie de hormigas termina su recorrido, se actualizan las feromonas de los nodos por los que han pasado. Para el mejor recorrido obtenido, la feromona de cada arista atravesada se multiplica por $1 - \rho$ y se le suma $\rho C L^{-1}$, siendo L^{-1} la inversa de la longitud del recorrido. En todos los recorridos, la feromona de cada arista recorrida se multiplica por $1 - \xi$ y se le suma $\xi \cdot I$. Con esto se pretende potenciar el mejor recorrido y reducir el interés en las aristas que no han intervenido en ese camino, de forma que la próxima generación de hormigas optará por repetir caminos similares al

óptimo evitando tomar aristas que no dieron resultado.

El algoritmo ejecuta varias generaciones de hormigas. Pueden usarse distintos criterios de parada, como por ejemplo detenerse cuando el mejor recorrido no se haya modificado en las últimas iteraciones, aunque optaremos por fijar el número de generaciones a un múltiplo del número de nodos del grafo.

2.2. Comparativa de los algoritmos

Para ver mejor las diferencias entre los tres algoritmos, hemos decidido ejecutarlos en algunos de los ejemplos propuestos, y hacer gráficos con los resultados. A continuación presentamos los caminos óptimos que deberían dar en cada uno de los ejemplos junto a lo que da cada algoritmo. Se adjunta, además, un diagrama de barras para poder apreciar mejor la eficacia de los resultados.

Cabe destacar que nos hemos encontrado recorridos que decían ser óptimos pero, por diversas razones, resultaban no serlo, o al menos no tomando las distancias euclídeas redondeadas a la baja. En `rd100.opt.tour` el nodo 7 estaba repetido y faltaba el 17. Por su parte, `pa561.opt.tour` usaba para las medidas de las aristas unos números indicados explícitamente, con valores distintos a las distancias entre sus nodos, y `gr666.opt.tour` tenía calculada la ruta óptima sobre una esfera, puesto que `gr666.tsp` contiene coordenadas esféricas y no cartesianas.

2.2.1. Ejemplo 1: Ulysses16

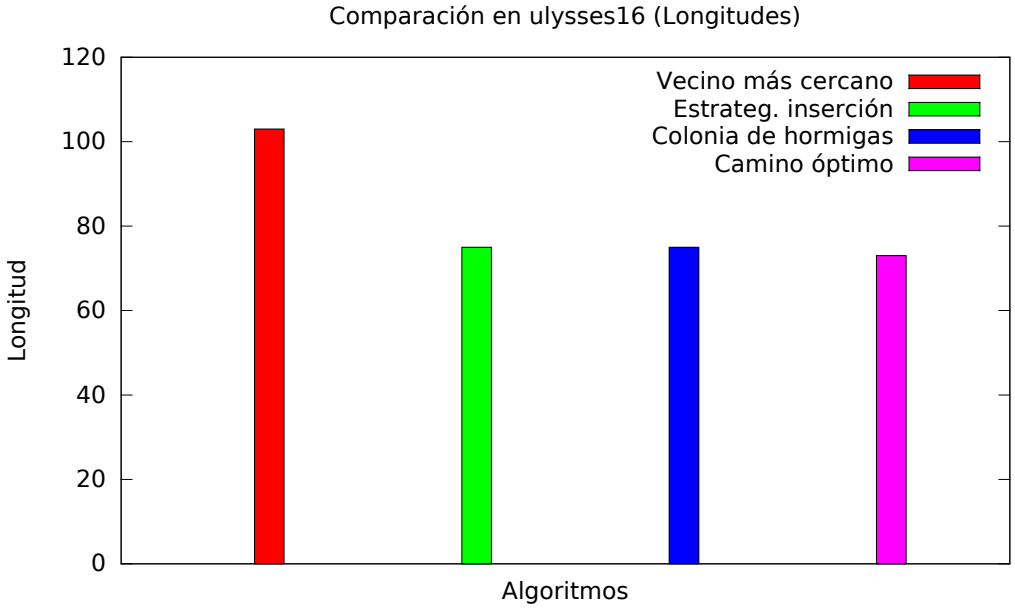

Figura 3: Comparativa de longitud de los caminos de cada algoritmo

En el primer ejemplo tanto inserción como la colonia de hormigas dan un resultado muy bueno, mientras que el vecino más cercano se aleja bastante de la longitud óptima. Colocamos aquí los caminos realizados:

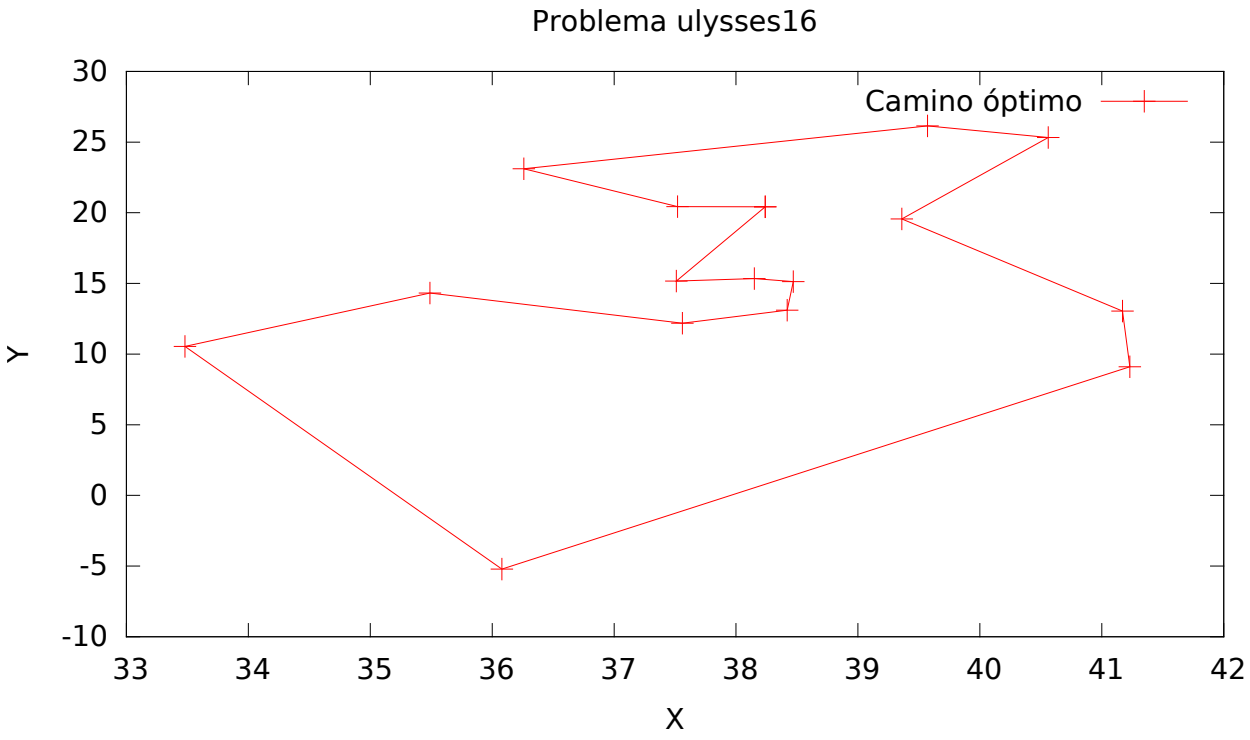

Figura 4: Camino óptimo para el problema Ulysses16

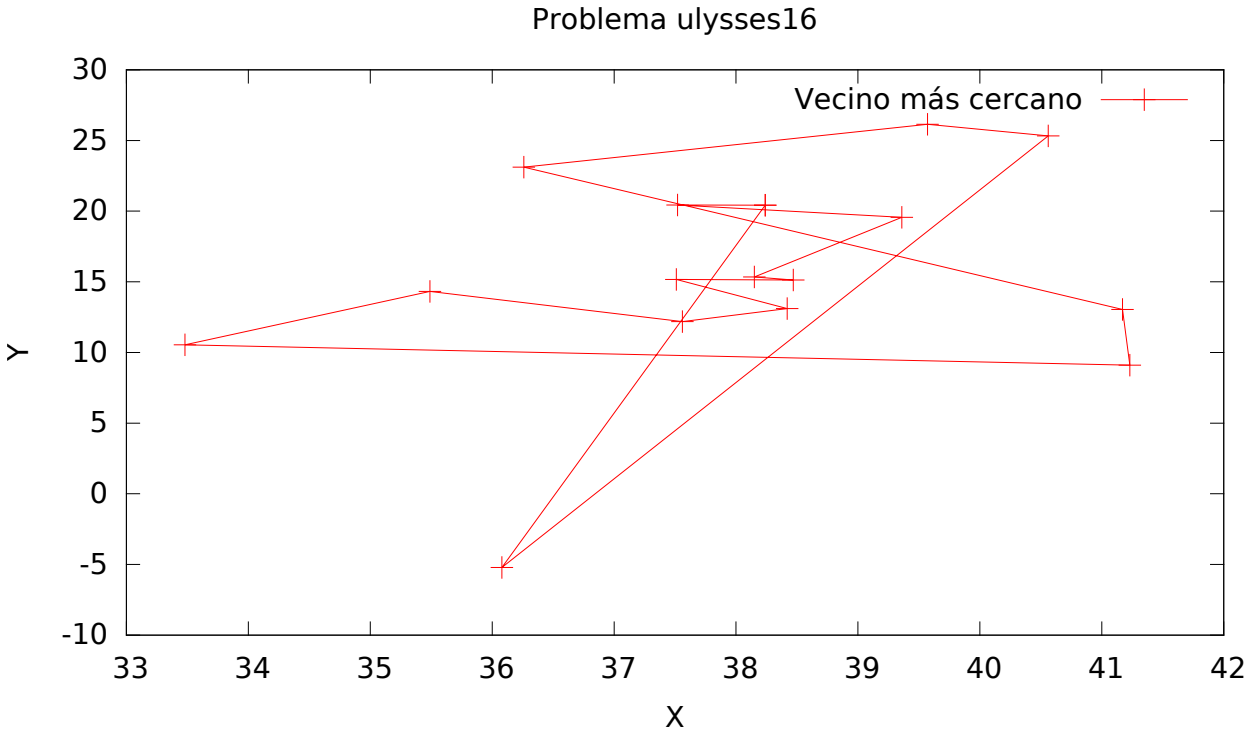

Figura 5: Camino proporcionado por el algoritmo del vecino más cercano

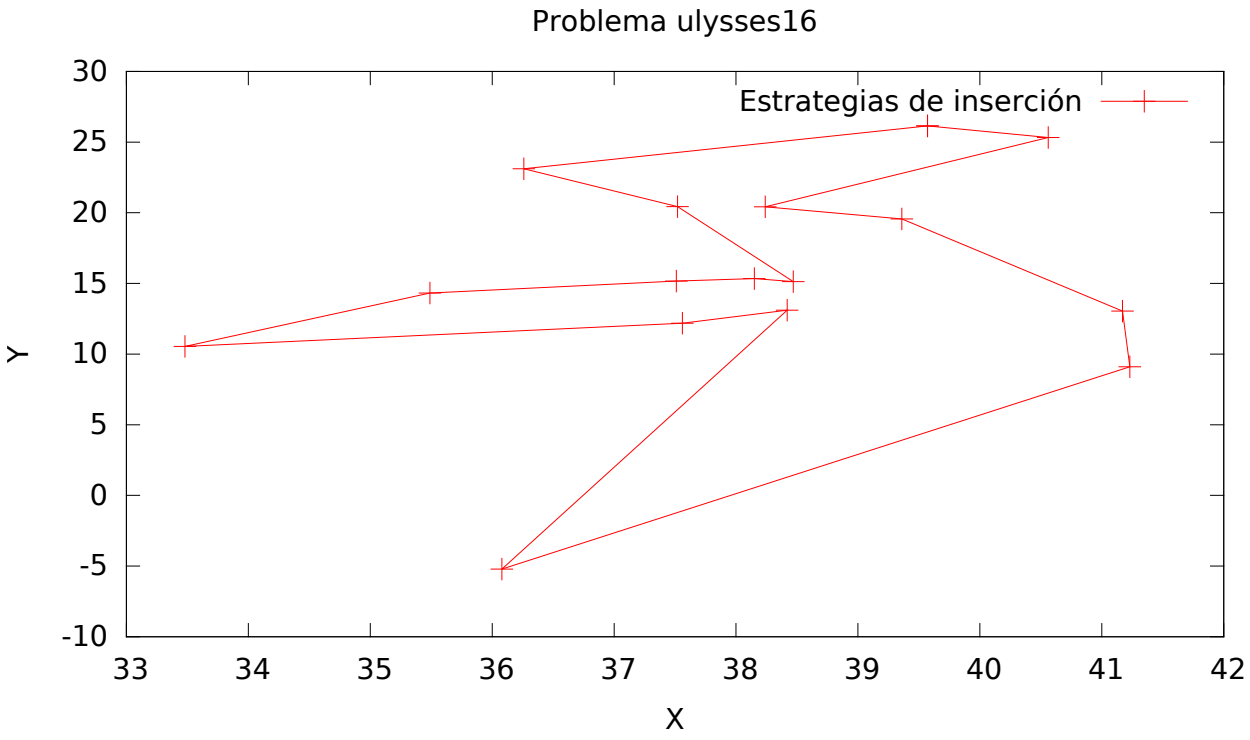

Figura 6: Camino proporcionado por el algoritmo de inserción

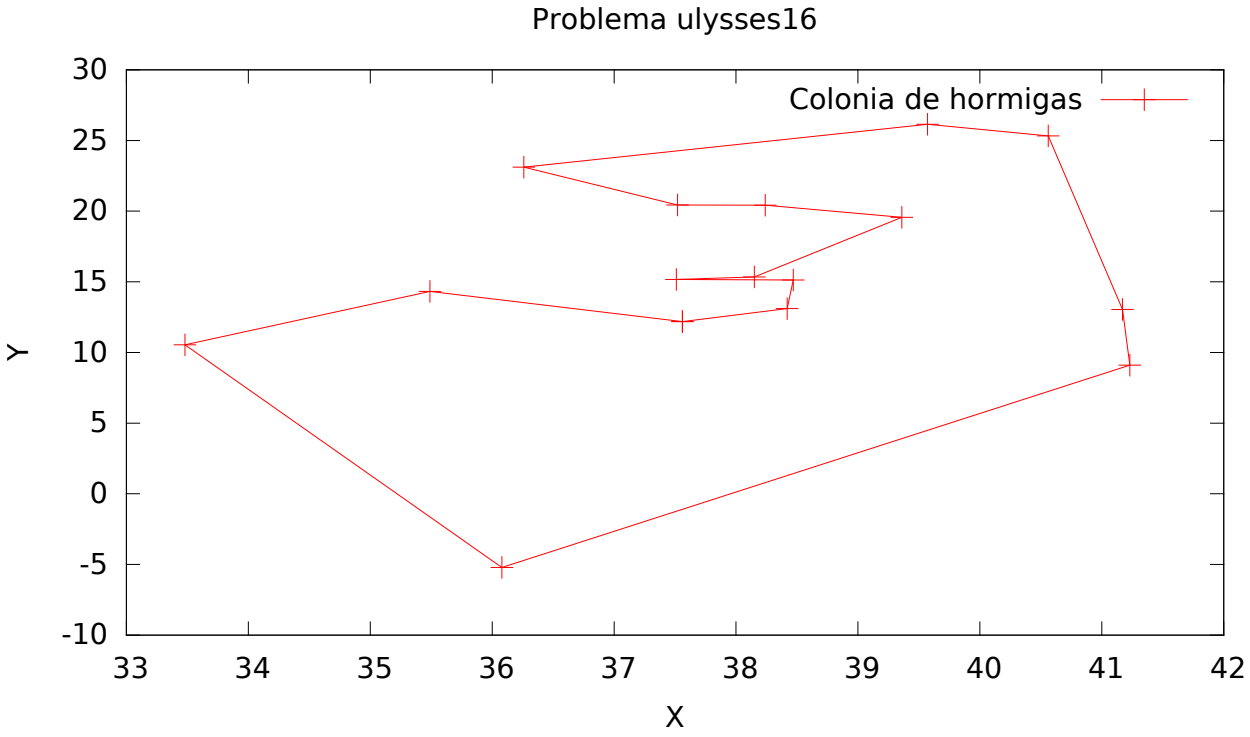

Figura 7: Camino proporcionado por el algoritmo de la colonia de hormigas

2.2.2. Ejemplo 2: Berlin52

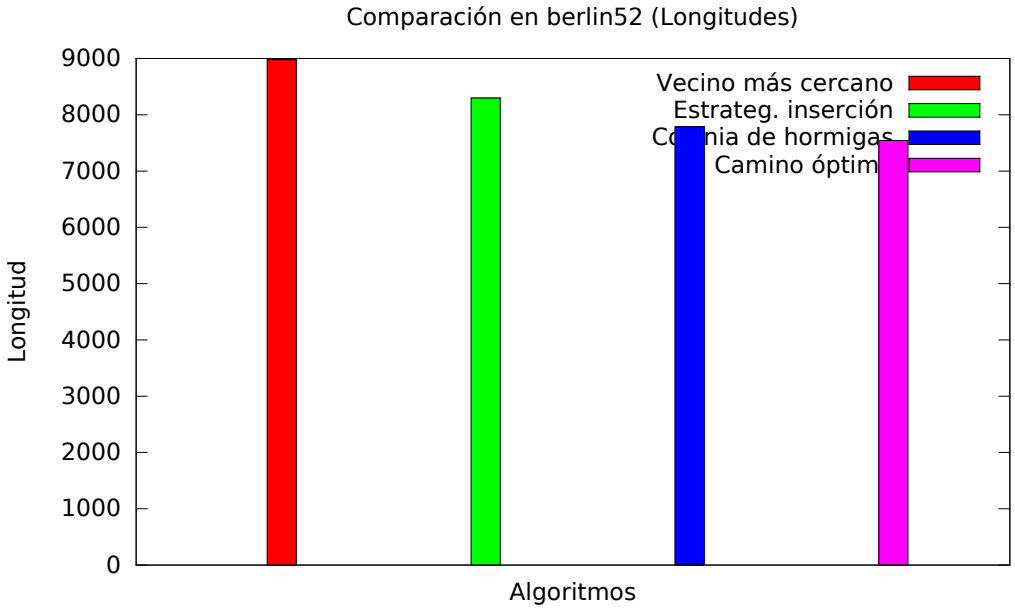

Figura 8: Comparativa de longitud de los caminos de cada algoritmo

En este ejemplo el mejor resultado lo da el tercer algoritmo, la colonia de hormigas. A continuación se ponen los caminos realiados:

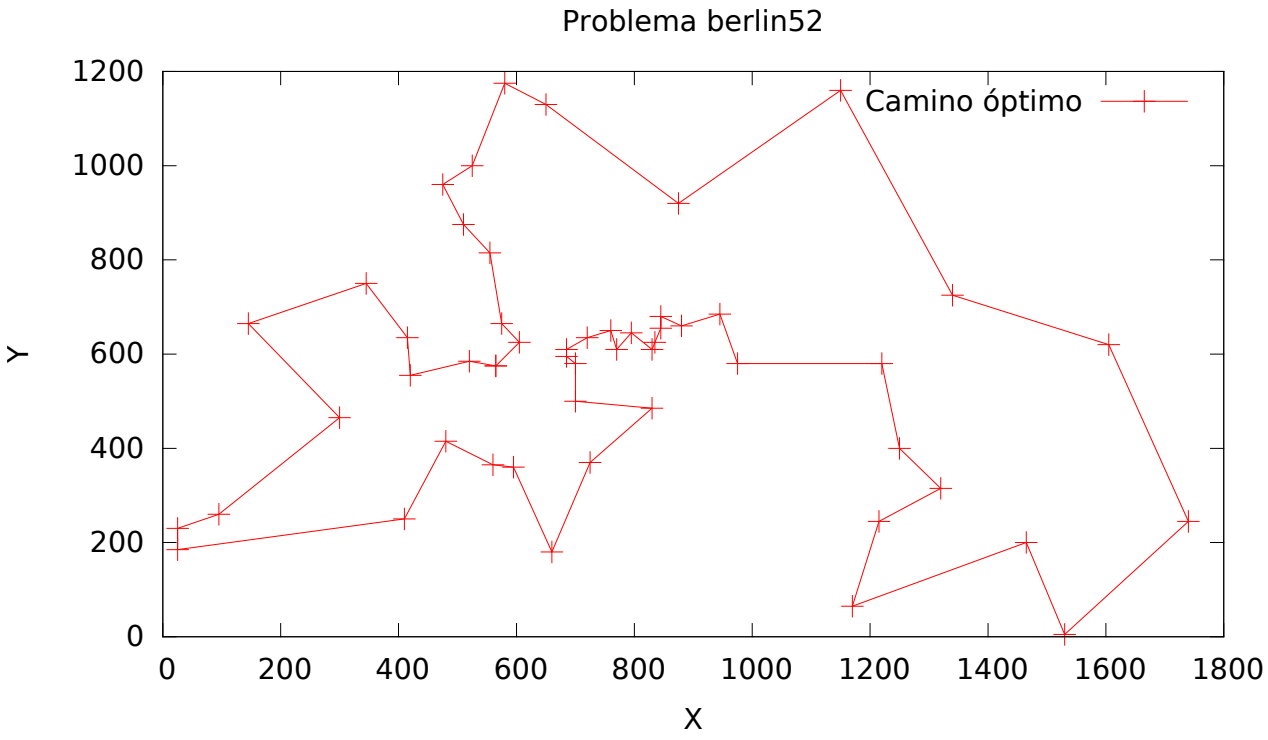

Figura 9: Camino óptimo para el problema Berlin52

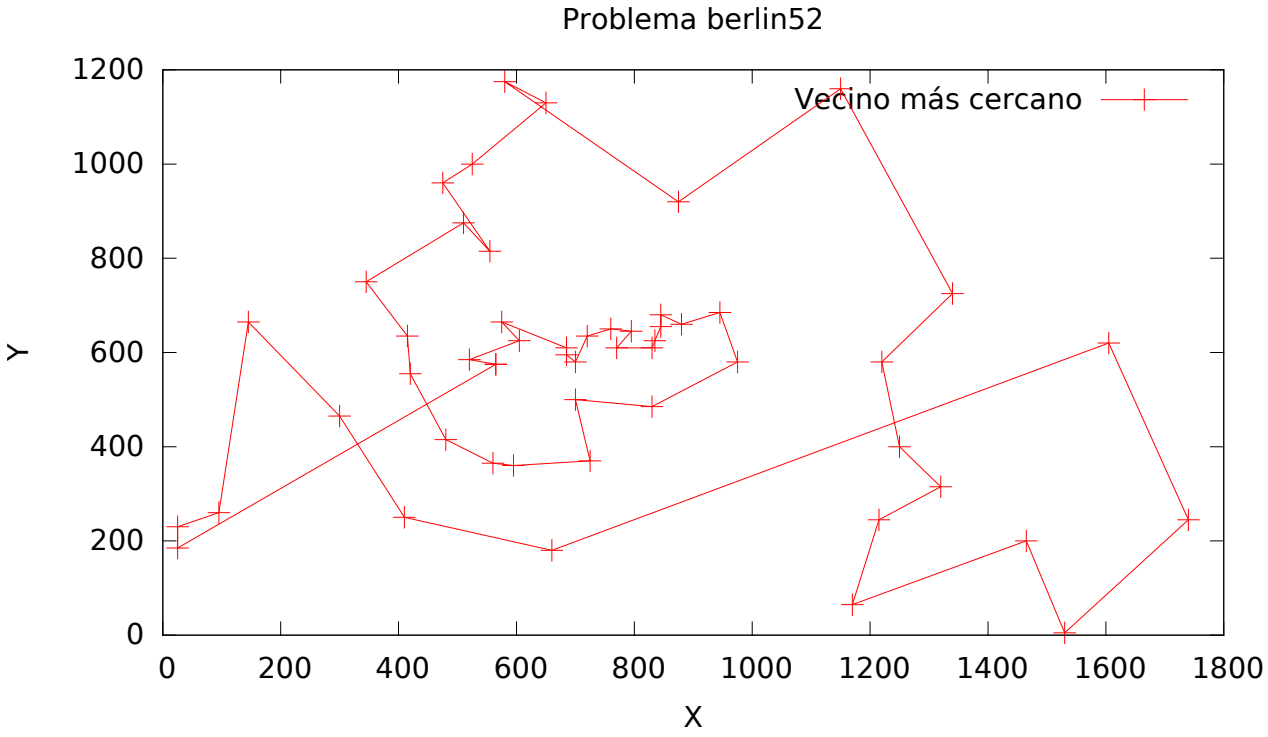

Figura 10: Camino proporcionado por el algoritmo del vecino más cercano

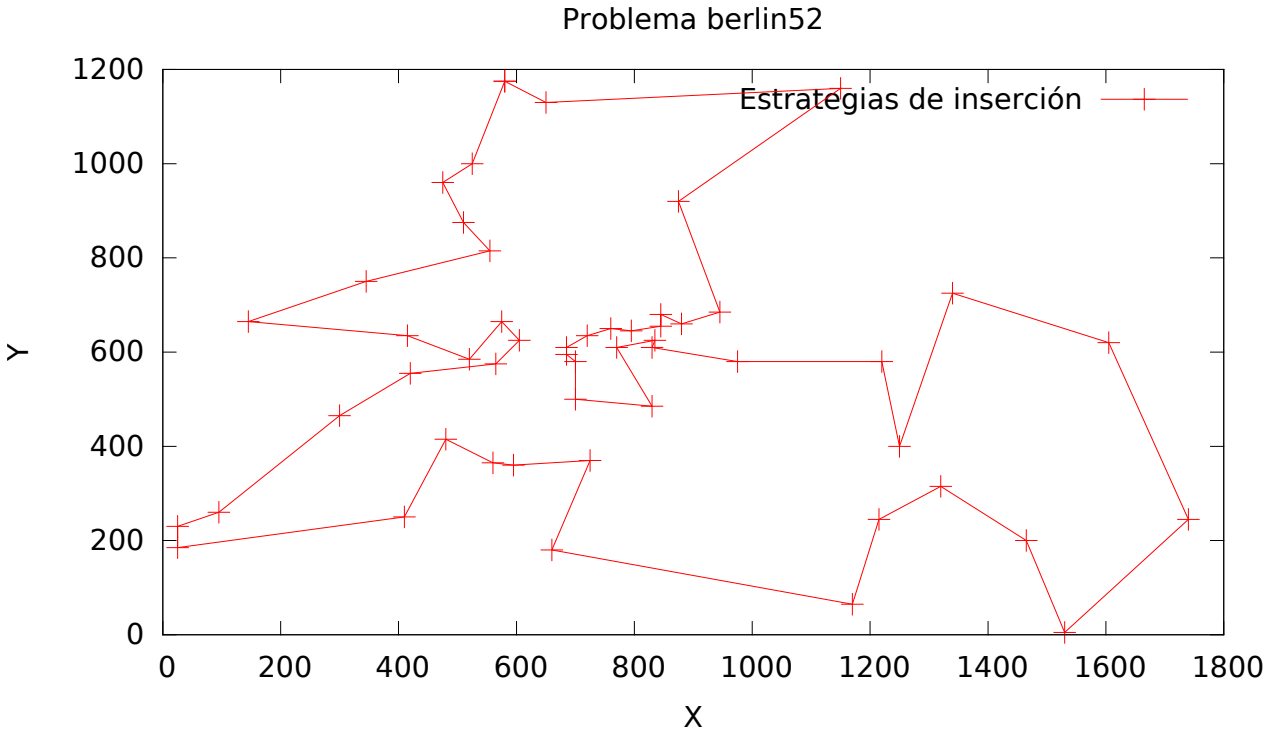

Figura 11: Camino proporcionado por el algoritmo de inserción

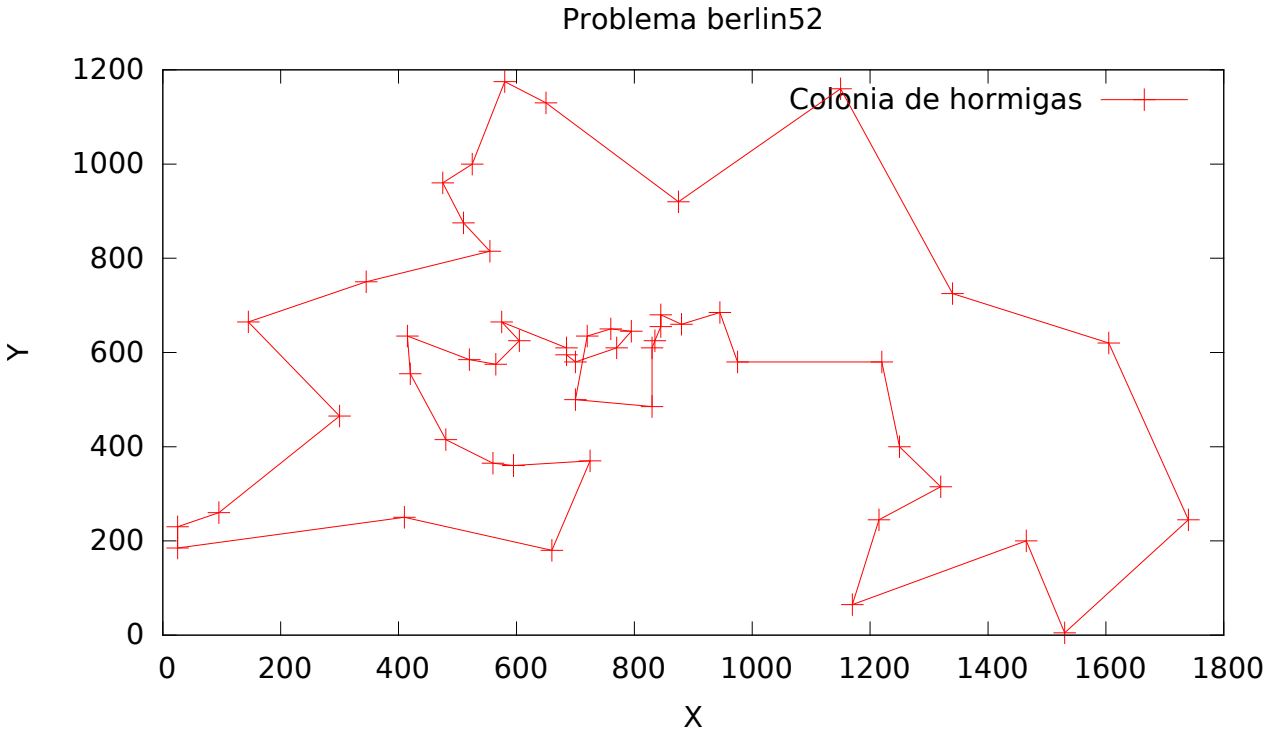

Figura 12: Camino proporcionado por el algoritmo de la colonia de hormigas

2.2.3. Ejemplo 3: Pr76

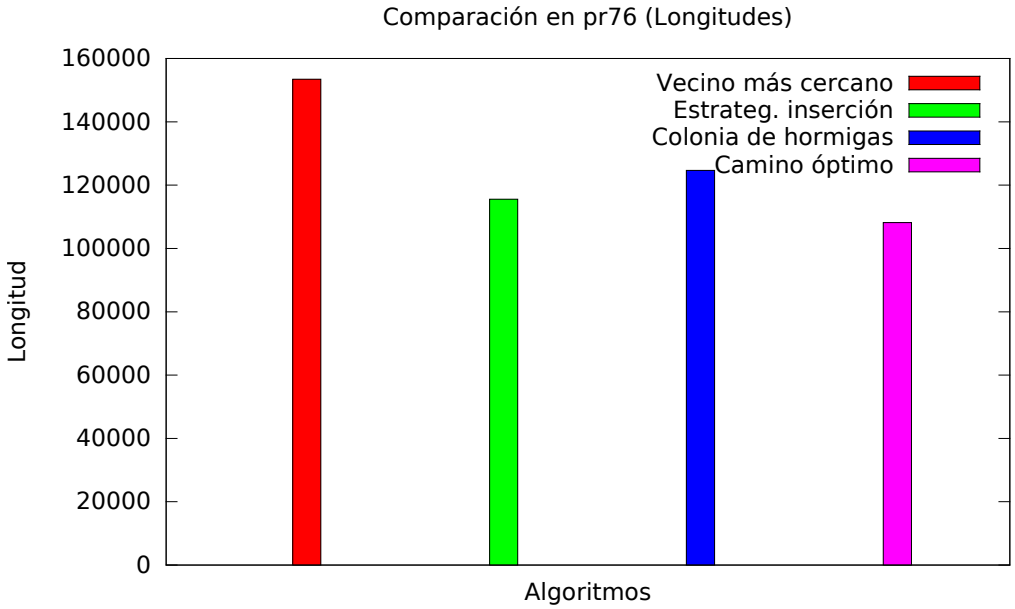

Figura 13: Comparativa de longitud de los caminos de cada algoritmo

Por último, hemos colocado un ejemplo en el que el de inserción es el que mejor resultados da, ganando tanto a la colonia de hormigas como al vecino más cercano. Los caminos realizados son:

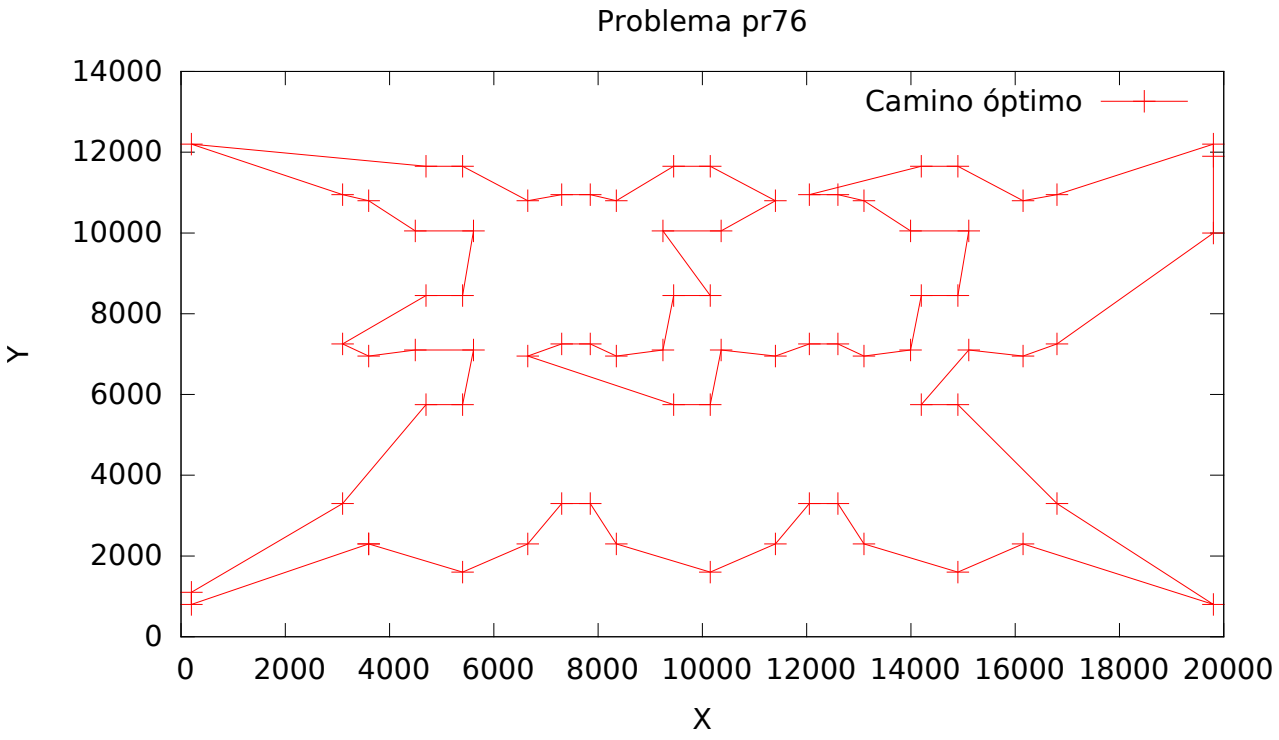

Figura 14: Camino óptimo para el problema Pr76

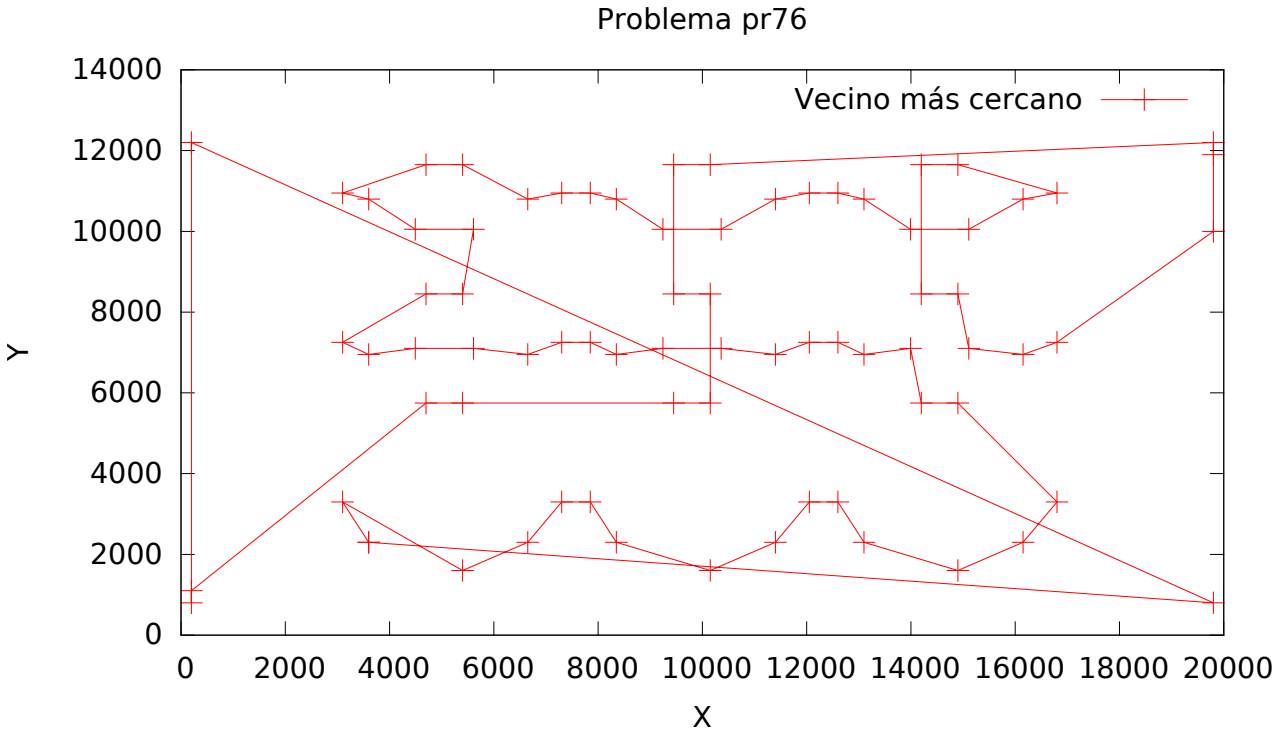

Figura 15: Camino proporcionado por el algoritmo del vecino más cercano

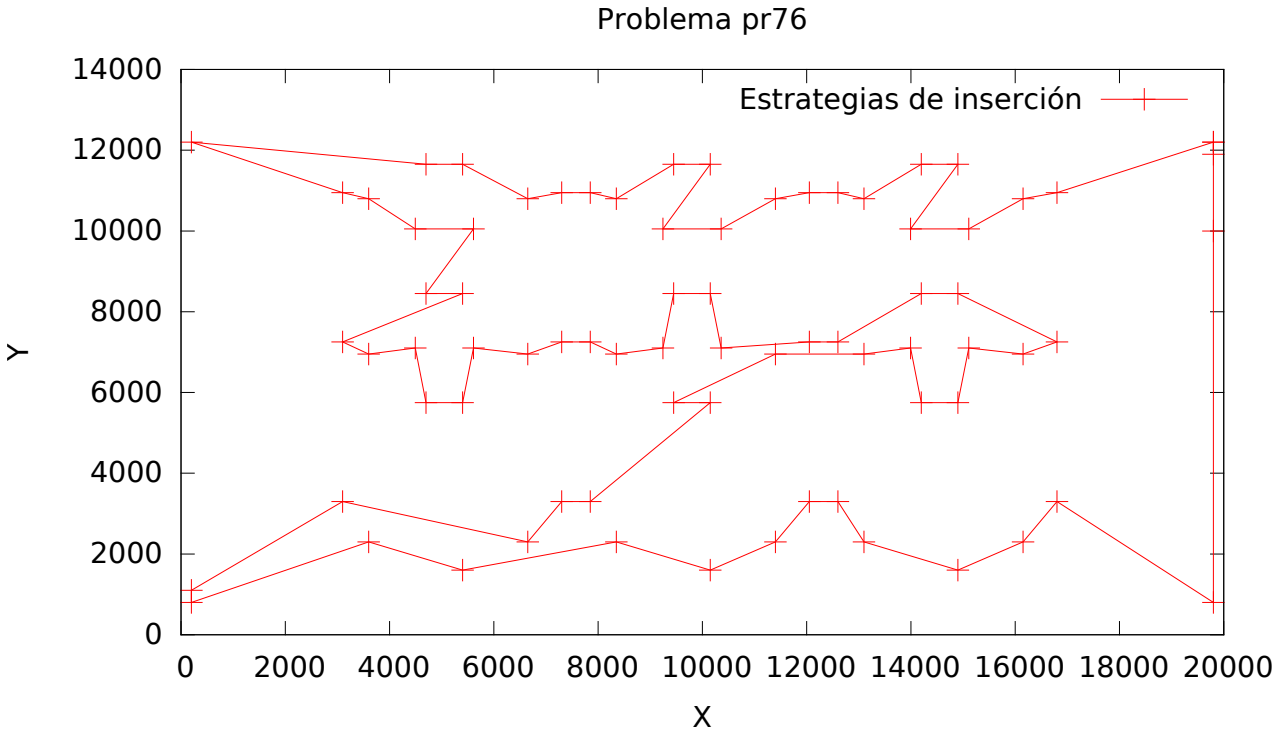

Figura 16: Camino proporcionado por el algoritmo de inserción

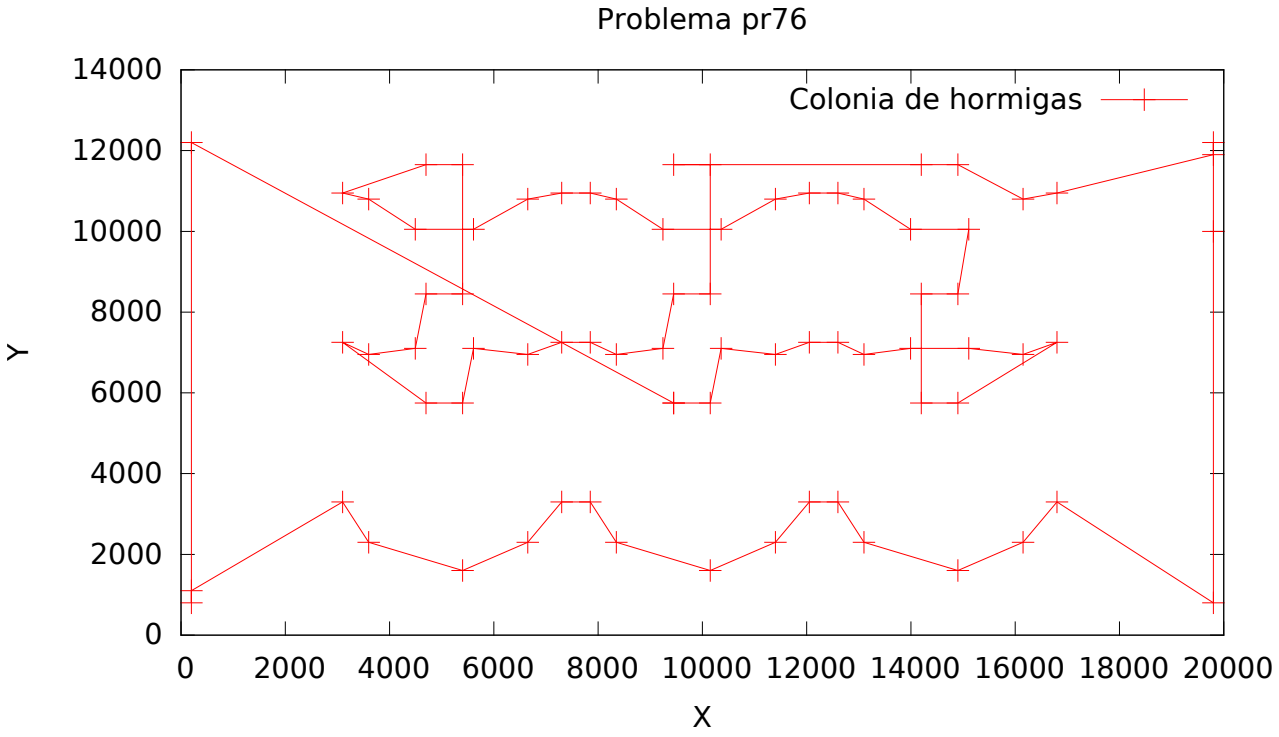

Figura 17: Camino proporcionado por el algoritmo de la colonia de hormigas