Práctica 3. Primera parte. Ingeniería de requisitos: Análisis y especificación de requisitos

Noelia Escalera Mejías   Alejandro Menor Molinero
Javier Núñez Suárez   Adra Sánchez Ruiz
Jesús Torres Sánchez

6 de mayo de 2019

1. Vecino más cercano

1.1. Descripción

La primera heurística que usaremos es bastante sencilla: escogeremos una ciudad inicial y a partir de ahí seleccionaremos la ciudad más cercana a la última escogida (que no haya sido seleccionada previamente) hasta que no queden ciudades por añadir al circuito. Haremos varias ejecuciones, empezando cada vez de una ciudad distinta, y escogeremos la opción con distancia mínima.

```

VecinosCercanos(distancias, n, resultado){
  completados;
  todas_las_ciudades;

  // Metemos los indices de las ciudades
  for (i=1; i<=n; i++)
    todas_las_ciudades.insert(i);

  // Iniciamos cada vez en una ciudad diferente
  for(i=1; i<=n; i++){
    candidatos = todas_las_ciudades;
    candidatos.erase(i);
    seleccionados.push_back(i);
    distancia = 0;

    //Creamos el circuito de la ciudad por la que empezamos
    while (!candidatos.empty()){
      actual = seleccionados.back();
      mas_cercano = *candidatos.begin();
      min = distancias[actual][mas_cercano];

      // Averiguamos la ciudad mas cercana
      for(c : candidatos){
        d = distancias[actual][c];
        if (d < min){
          mas_cercano = c;
```

```
            min = d;
          }
        }

        seleccionados.push_back(mas_cercano);
        distancia += min;
        candidatos.erase(mas_cercano);
      }
      distancia += (distancias[seleccionados.front()][seleccionados
    .back()]);

      completados[distancia] = seleccionados;
    }
    resultado = completados.begin()->second;
  }


```

Listing 1: Pseudocódigo de la primera heurística

1.2. Resultados

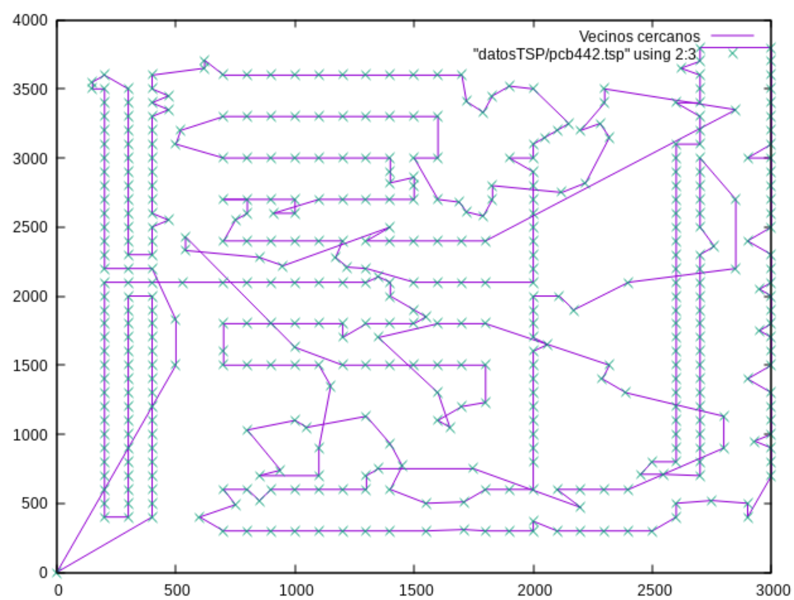

Figura 1: Vecinos cercanos para tamaño 442

1.3. Eficiencia

Este algoritmo es de orden $O(n^2)$, es bastante eficiente si tenemos en cuenta que en cada iteración hay menos ciudades entre las que hallar la ciudad más cercana. De todos los que analizaremos este es el más eficiente.

2. Inserción más económica

2.1. Descripción

En este segundo algoritmo, empezamos con un circuito que contiene tres ciudades y posteriormente añadimos las ciudades restantes al recorrido. En cada iteración del algoritmo se busca la ciudad que tenga la inserción con menos coste en el circuito ya existente. Entendiendo como coste, la distancia que hay entre la ciudad a (ya existente) c (a insertar) más la distancia entre c y b(ya existente) menos la que hay entre a y b.

```

Insercion(distancias, n, resultado, ciudades){
  n = buscarCiudadNorte();
  s = buscarCiudadSur();
  e = buscarCiudadEste();

  resultado.aniade(n,s,e);
  distanciaFinal = distancias[n][e]+distancias[e][s]+distancias[s][n];

  for(int i=1;i<=n;i++){
    if(i!=n && i!=s && i!=e){
      candidates.insert(i);
    }
  }

  while(!candidatos.empty()){
    for(c:candidatos){
      for(it=resultado.begin(); it!=resultado.end();it++){
        siguiente = it;
        siguiente++;

        if(siguiente == resultado.end())
          siguiente = resultado.begin();

        diferencia = -distancias[*it][*siguiente];
        diferencia += distancias[*it][c];
        diferencia += distancias[c][*siguiente];

         if (it == resultado.begin() || diferencia < distanciaMinima){
        distanciaMinima = diferencia;
        insercionMinima = it;
        }
      }
      if (distanciaMinima < calculoMinimo){
      calculoMinimo = distanciaMinima;
      posicionMinima = insercionMinima;
      candidataMinima = c;
      }
    }
    // Borramos de candidatos
    candidatos.erase(candidataMinima);

    // Insertamos
```

```
            posicionMinima++;
            resultado.insert(posicionMinima, candidataMinima);

            // Actualizamos la distancia con la insercion
            distanciaFinal += calculoMinimo;
        }
        return distanciaFinal;
    }

```

Listing 2: Pseudocódigo de la segunda heurística

2.2. Resultados

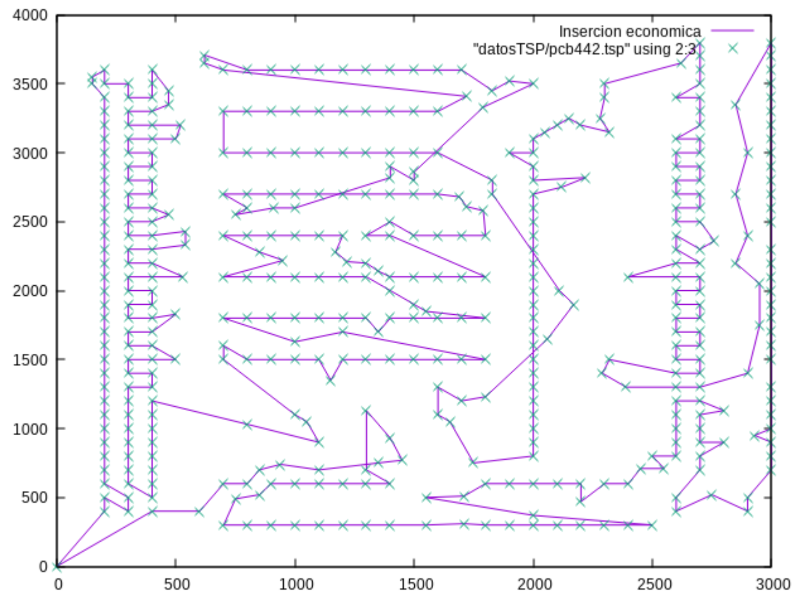

Figura 2: Inserción económica para tamaño 442

2.3. Eficiencia

Si bien la inserción económica representa en muchos casos una mejoría respecto a los vecinos cercanos en resultados, es peor en eficiencia. En este caso estamos hablando de un orden de $O(n^3)$

3. Estrategia adicional

Hemos diseñado para este apartado dos algoritmos de inserción, la diferencia con respecto el segundo apartado es el circuito parcial del que partimos. Además de la eficiencia, (ambos son de orden $O(n^2)$)

En vez de elegir tres puntos (norte, sur y este), utilizamos el algoritmo de Jarvis para obtener la envolvente convexa, *convex hull*, del conjunto de ciudades dadas. Este algoritmo pertenece al orden $O(n^2)$ de eficiencia.

Vamos a empezar definiendo qué es un **conjunto convexo**:

Un conjunto es convexo si dados dos puntos cualesquiera, el segmento que los une está contenido en el conjunto.

*Una **envolvente convexa** es el menor conjunto convexo de un conjunto de puntos que los contiene.*

De esta envolvente convexa nos interesan las ciudades que hacen de límite para empezar con el circuito que forman, el algoritmo de inserción.

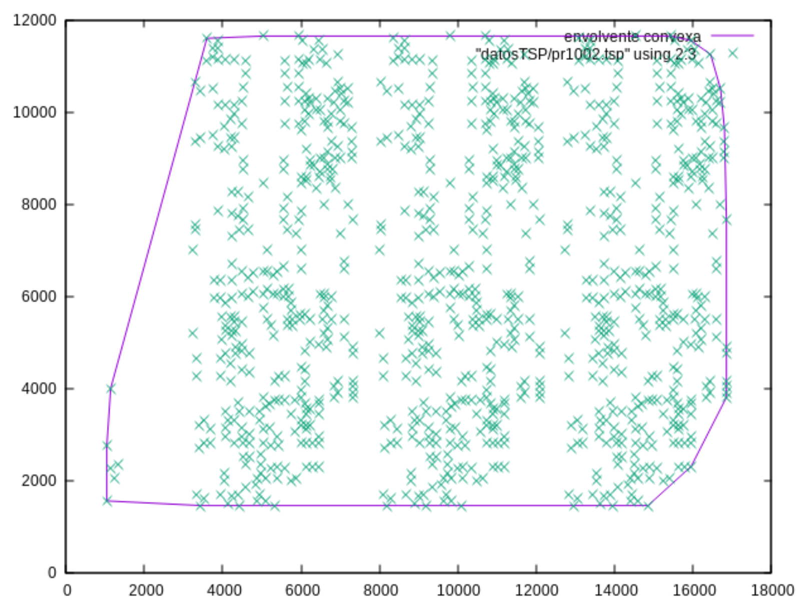

Figura 3: Envolvente convexa del conjunto de puntos pr1002.tsp

3.1. Inserción de la ciudad más lejana

Partiendo del circuito que acabamos de explicar, vamos a insertar ciudades del conjunto de candidatos (al principio las que no están en el circuito inicial) con la estrategia de insertar la ciudad más lejana respecto al conjunto de ciudades seleccionadas.

Es decir, primero iteramos por las seleccionadas para hallar la distancia a la que está la ciudad candidata de la ciudad mas lejana en el conjunto de seleccionadas. Después, de todas estas distancias, escogemos la mayor y la insertamos en el lugar más óptimo posible igual que en el segundo apartado.

```

circuitoInicial = crearEnvolturaConvexa(ciudades)

// Contiene todas las ciudades menos las que forman parte del circuito inicial
candidatos = crear.conjunto()

mientras(!candidatos.empty()){
  int ciudadMasLejana;
  int maximaDistancia;

  // Hallamos la ciudad mas lejana respecto
  // al conjunto de seleccionadas
  for (int c :candidatos){
    int dMasLejana;
    for (int s : seleccionadas){
      if (distancia(c,s) > dMasLejana)
        dMasLejana = distancia(c,s);
    }
    if (dMasLejana > maximaDistancia)
      ciudadMasLejana = c;
  }
  list<int>::iterator insercionMasBarata;
  int cuantoCuesta;

  for (auto it = resultado.begin() ; it != resultado.end() ; it++){
  auto siguiente = it;
  siguiente++;

  if (siguiente == resultado.end())
  siguiente = resultado.begin();

  int coste = -distancias[*it][*siguiente];

  coste += distancias[*it][ciudadMasLejana];
  coste += distancias[ciudadMasLejana][*siguiente];

  if (it == resultado.begin() || coste < cuantoCuesta){
    insercionMasBarata = it;
    cuantoCuesta = coste;
  }


  candidatos.erase(ciudadMasLejana);
  distancia += cuantoCuesta;
  insercionMasBarata++;
  resultado.insert(insercionMasBarata, ciudadMasLejana);
}
return distancia;

```

Listing 3: Pseudocódigo de inserción con envoltura convexa y ciudad más lejana

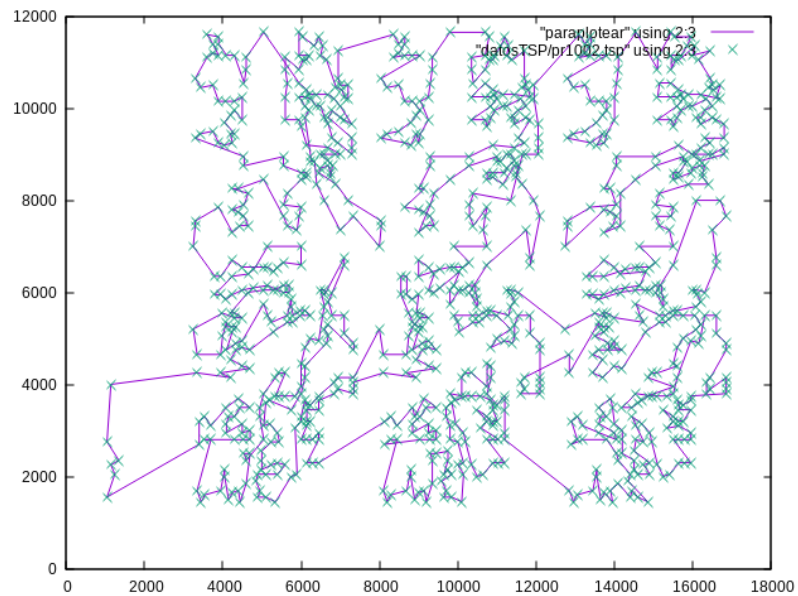

Figura 4: Resultado con estrategia de inserción "mas lejana" y partiendo de la envolvente convexa de pr1002.tsp

3.2. Inserción de la ciudad más cercana

De nuevo partimos del circuito que hemos obtenido con la envolvente convexa, pero ahora vamos a escoger para la inserción la ciudad que más cerca esté del conjunto de seleccionadas.

Igual que antes, primero iteramos por las seleccionadas para hallar la distancia a la que está la ciudad candidata de la ciudad mas cercana en el conjunto de seleccionadas. Después, de todas estas distancias, escogemos la menor y la insertamos en el lugar más óptimo posible igual que en el segundo apartado.

```

circuitoInicial = crearEnvolturaConvexa(ciudades)

// Contiene todas las ciudades menos las que forman parte del
    circuito inicial
candidatos = crear.conjunto()

mientras(!candidatos.empty()){
int ciudadMasCercana;
int minimaDistancia = INT_MAX;
```

```

// Hallamos la ciudad mas cercana respecto
// al conjunto de seleccionadas
for (int c :candidatos){
int dMasCercana;
for (int s : seleccionadas){
if (distancia(c,s) < dMasCercana)
dMasCercana = distancia(c,s);
}
if (dMasCercana < minimaDistancia)
ciudadMasCercana = c;
}
list<int>::iterator insercionMasBarata;
int cuantoCuesta;

for (auto it = resultado.begin() ; it != resultado.end() ; it++){
auto siguiente = it;
siguiente++;

if (siguiente == resultado.end())
siguiente = resultado.begin();

int coste = -distancias[*it][*siguiente];

coste += distancias[*it][ciudadMasLejana];
coste += distancias[ciudadMasLejana][*siguiente];

if (it == resultado.begin() || coste < cuantoCuesta){
insercionMasBarata = it;
cuantoCuesta = coste;
}


candidatos.erase(ciudadMasLejana);
distancia += cuantoCuesta;
insercionMasBarata++;
resultado.insert(insercionMasBarata, ciudadMasLejana);
}
return distancia;
```
Listing 4: Pseudocódigo de inserción con envoltura convexa y ciudad más lejana

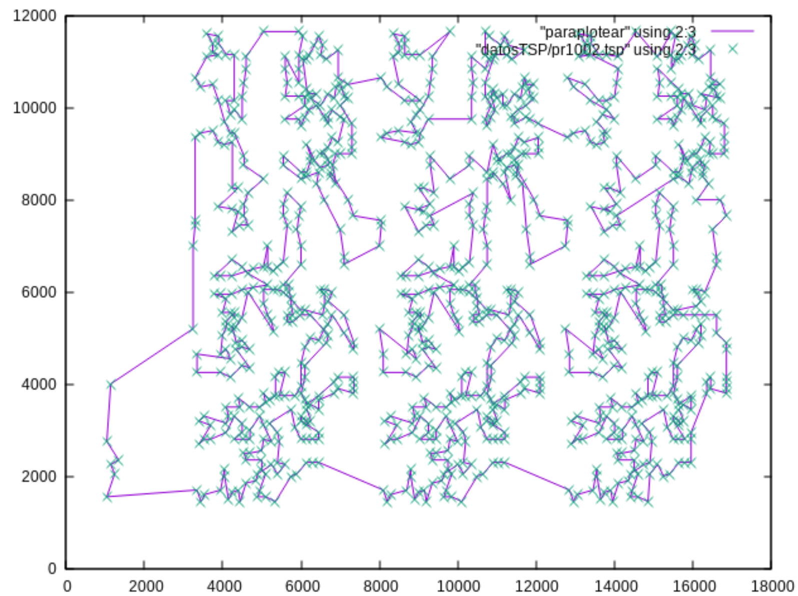

Figura 5: Resultado con estrategia de inserción "más cercana" y partiendo de la envolvente convexa de pr1002.tsp

4. Comparativa final y conclusiones

A continuación, hemos realizado una comparativa con las soluciones para cada una de las 4 versiones. Para ello, hemos ejecutado los algoritmos a partir de una serie de mapas y, para cada distancia calculada de cada algoritmo, hemos obtenido el **cociente comparativo** entre la distancia obtenida y la distancia óptima (datos ya proporcionados).
Tras realizar este estudio, podemos observar los siguientes resultados:

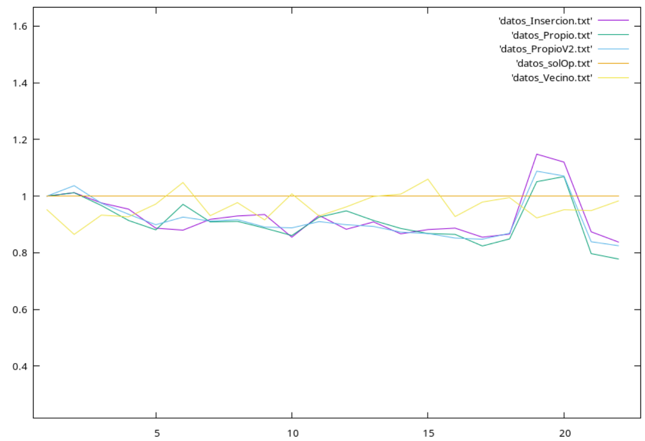

Figura 6: Gráfica comparativa de resultados

He aquí la tabla con los resultados de las ejecuciones representadas en la gráfica anterior:

NºCiudades	Solución óptima	Vecino más cercano	Inserción	Jarvis v1	Jarvis v2
16	80	84	80	80	80
22	84	96	83	83	81
29	9091	9981	9317	9402	9325
48	33551	37928	35163	36714	35863
51	461	535	520	523	513
52	7570	8207	8598	7799	8174
70	715	837	779	787	784
76	585	644	629	642	638
96	548	640	586	618	615
100	21345	24758	24960	24803	24049
101	701	809	752	757	770
105	14418	16976	16324	15204	16045
120	1732	1907	1905	1896	1939
130	6179	7076	7128	6972	7081
150	6612	7073	7496	7616	7614
225	3854	4682	4346	4458	4523
280	2613	3120	3055	3173	3082
442	50850	59016	58747	59908	5850
561	19610	18498	17076	18661	18023
666	4238	3973	3784	3965	3957
1002	259439	312609	296711	325604	309215
2392	378761	459486	451725	487131	458975

Y las medias de los costes:

Óptima	Vecino más cercano	Inserción	Jarvis v1	Jarvis v2
37410,7727272727	44497,0454545455	43171,0909090909	46218	41690,7272727273

Como conclusión, podemos observar como, aunque en general el algoritmo basado en la selección del vecino más cercano parece el más óptimo, no podemos afirmar que lo sea. Dependiendo del mapa obtenemos resultados muy variados; en algunos destaca el algoritmo del vecino, pero en muchos otros es menos óptimo que el resto. Esto ocurre debido a que la optimalidad de nuestros algoritmos no solo depende del tamaño del problema, sino que también depende de otros factores como la disposición de las ciudades en el mapa.

Finalmente, para tener una visión más amplia de los resultados generados por nuestros algoritmos, podemos observar los resultados para un mapa sencillo (22 ciudades):

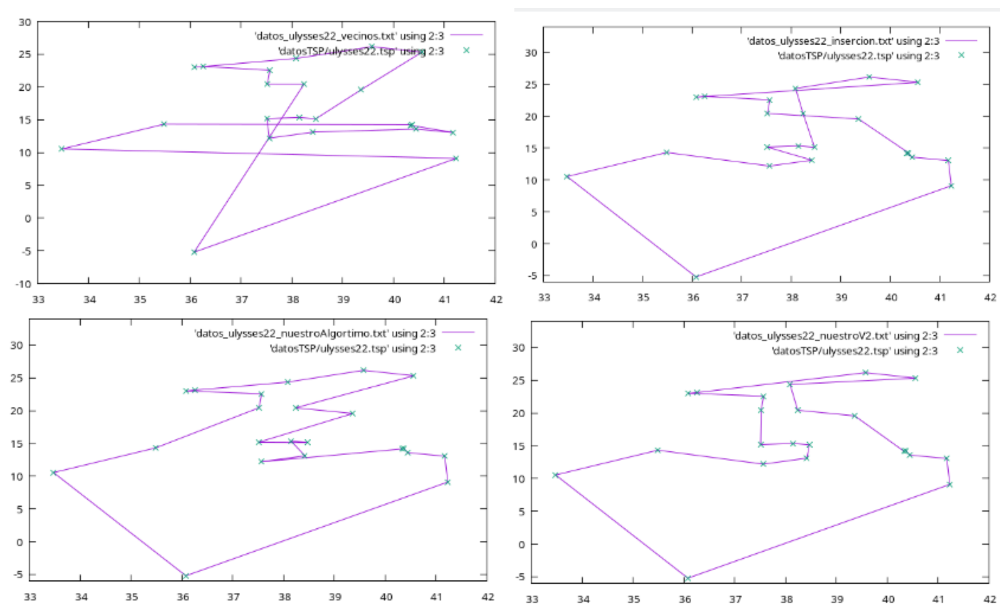

Figura 7: Comparativa con el ejemplo de 22 ciudades

5. Fuentes

Para acabar mencionar brévemente de dónde nos hemos informado para implementar los algoritmos del apartado 3:

- Página de wikipedia sobre la envoltura convexa
- MIT Open Course Ware sobre heurísticas del TSP

- Página de la wikipedia sobre el algoritmo de Jarvis