Práctica 4 - Programación dinámica

Memoria de práctica

Germán Castilla López
Jorge Gangoso Klöck
Pedro Morales Leyva
Clara Mª Romero Lara

Índice de Contenidos

1. Descripción general de la práctica

En esta práctica exploraremos técnicas de **programación dinámica** aplicadas sobre el problema del Viajante de Comercio (o TSP, por sus siglas en inglés). Obtendremos una solución óptima y realizaremos comparaciones respecto a su versión *Greedy*, ya desarrollada en la práctica número 3.

El enunciado del TSP es el siguiente:

> Dado un conjunto de ciudades y una matriz con las distancias entre todas ellas, el viajante ha de recorrer todas las ciudades exactamente una vez y regresar al punto de partida, de forma tal que la distancia recorrida sea la mínima.

2. Problema del viajante de comercio

2.1. Descripción

Esta solución basada en programación dinámica utiliza el enfoque de la **memoización**: almacenamos resultados de los subproblemas para poder utilizarlos cuando los volvamos a necesitar. Se ha implementado con un `<map>` de la STL llamado `memoizacion`. Este está definido de la siguiente manera:

- Una clave en `string`, definida por la ciudad actual y las ciudades que quedan por visitar
- Un entero, la distancia asociada

El programa consta de las siguientes funciones (a parte de las empleadas previamente en la práctica 3 para entrada de datos y similares):

- `subTSP(int actual, vector<int> sin_visitar, vector<vector<int>> D)`: esta función recibe: la ciudad actual, el vector de ciudades sin visitar y la matriz de distancias. Se llamará de forma recursiva. Devuelve un número entero, que corresponde con la **menor distancia** entre la ciudad `actual` y la última de las ciudades `sin_visitar`.

  Su funcionamiento es el siguiente: dada la ciudad `actual`, se comprueba lo primero que queden ciudades por visitar. Si este es el caso, continuamos y comprobamos en el `<map>` si ya hemos calculado la mejor salida para este caso. Esto se hace comprobando la **clave**, que hemos definido previamente.

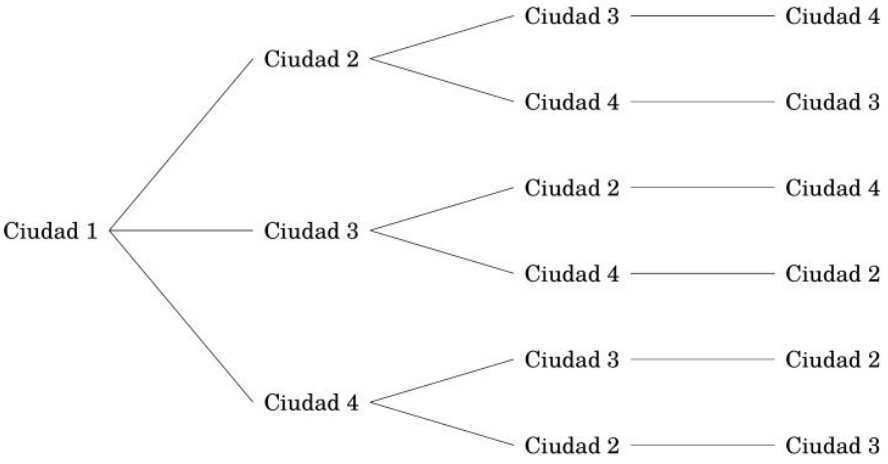
  Si la búsqueda no ofrece resultados, procederemos a crear una nueva clave con la ciudad actual y las ciudades por visitar. Es en este punto que empleamos la recursividad: obtendremos las distancias óptimas de los subproblemas (de los cuales podremos ahorrarnos cálculos si los resultados ya se encuentran en el map `memoizacion`) y haremos el sumatorio de los subproblemas óptimos. Si al inicio se comprueba que no quedan ciudades libres, finalmente se suma la distancia hasta la ciudad 0.

  Así cumplimos el **principio de optimalidad de Bellman**: hemos obtenido una respuesta óptima a partir de las subsoluciones óptimas del problema.

- `TSP(int actual, vector<int> sin_visitar, vector<vector<int>> D)`: esta función recibe los mismos parámetros que la anterior, y opera de forma extremadamente similar. La diferencia únicamente recae en la salida: si bien `subTSP` devuelve la menor distancia total, `TSP` devuelve **los índices a recorrer para obtener esa distancia**. Contiene una llamada a `subTSP`, para seleccionar el índice correcto cada ciclo.

En el main llamamos una primera vez a `subTSP` para generar el map y obtener la distancia total. Luego llamamos a `TSP` para obtener el recorrido; y, como ya se llamó previamente a `subTSP`, esta llamada interna no tomará apenas tiempo porque ya se había generado el map.

Tomemos un ejemplo simplificado de 4 ciudades:



Partimos de la ciudad inicial, y cada nivel del árbol nos muestra las ciudades aún no visitadas de las que disponemos. En la función `subTSP` inicial, este árbol no se ha generado, así que tendremos que calcular todas las rutas.

En la llamada a `TSP`, todas las claves ya han sido generadas así que cada vez que queramos buscar la solución a una ruta, podremos encontrar en el map la distancia `mejor`, cumpliendo así el principio de la programación dinámica de encontrar subsoluciones superpuestas (*overlapping solutions*).

2.2. Pseudocódigo

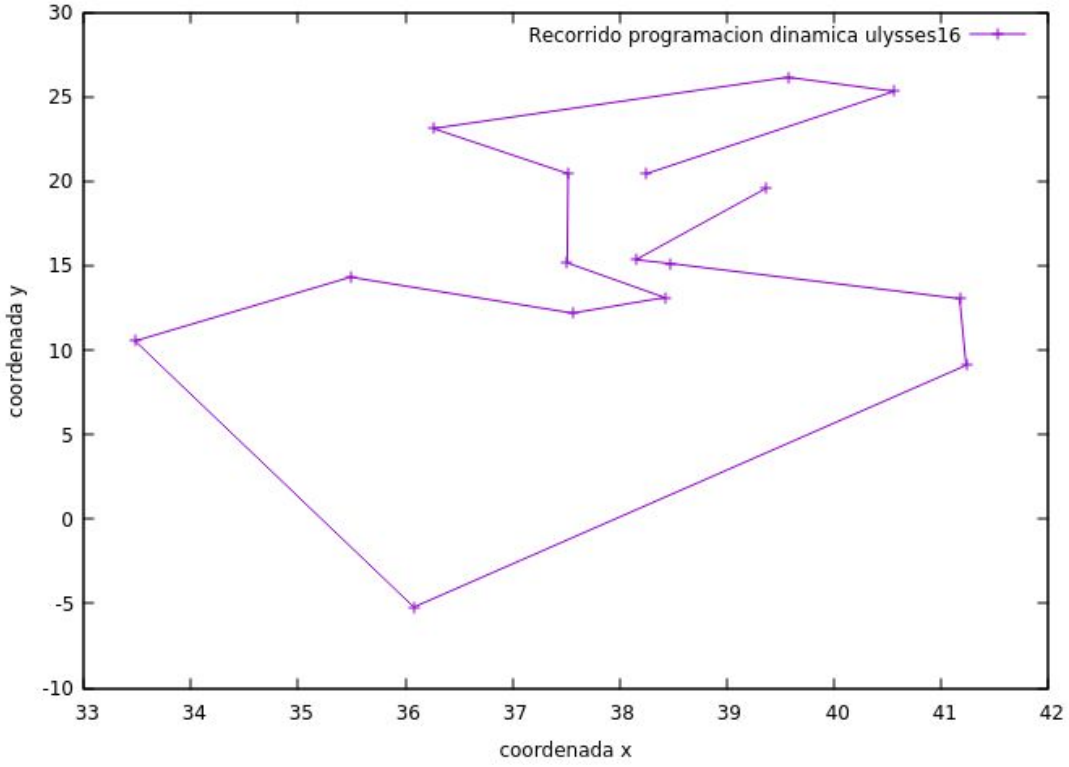
```
subTSP(actual, ciudades_por_visitar, D)
   if ciudades_por_visitar > 0
      if clave !encontrada   //buscamos con un iterador el map
         definir clave
            for(i=0; i<ciudades_por_visitar; i++)
              erase(i) from ciudades_por_visitar
              subTSP(i, ciudades_por_visitar, D)
      else
         get D clave     //tomamos la distancia asociada a la clave
   else
      return D[actual][0]
```

2.3. Supuesto de ejecución

Compilamos con el makefile proporcionado, y ejecutamos con `./TSP <nombre_fichero>`

Tomaremos como supuesto el archivo *Ulysses16.tsp,* ya que es lo suficientemente grande como para permitirnos ver un recorrido interesante sin llegar a tomar un tiempo de procesamiento excesivo.

En la gráfica no se muestra, pero para los cálculos se ha tenido en cuenta la distancia entre la última y la primera ciudad visitada



- **Distancia total:** 71
- **Camino final:** 1 3 2 4 8 14 7 6 15 5 11 9 10 12 13 16

3. Comparativa con enfoque greedy

3.1. Supuestos de ejecución

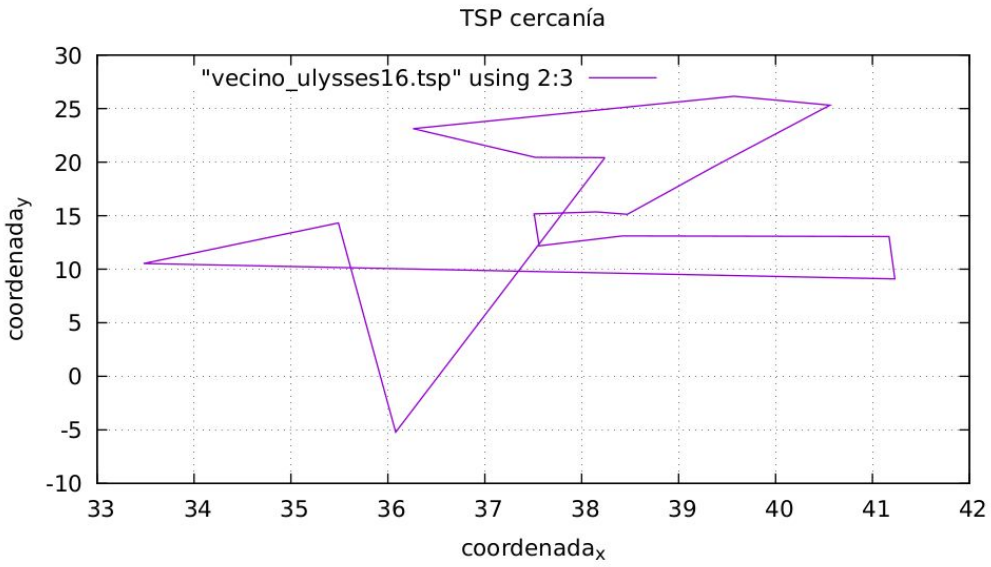
Como era de esperar, la ejecución de un algoritmo con programación dinámica tiene una duración mucho mayor que la ejecución de un algoritmo greedy, sin embargo nos da la solución más óptima. Como podemos comprobar, el recorrido en programación dinámica no se cruza sobre sí mismo en ningún momento.

Como hemos comentado, solo hemos ejecutado con programación dinámica el archivo dado de *ulysses16.tsp*. Así que la comparación la basaremos en este recorrido, comparando con el método greedy del vecino más cercano, que fue el más eficiente dentro de los desarrollados en la práctica anterior.

También hemos querido comparar la diferencia de tiempo que hay entre los métodos greedy y la programación dinámica, así que hemos tomado el archivo de *ulysses16.tsp* y hemos medido los tiempos desde tener 10 ciudades hasta las 16.

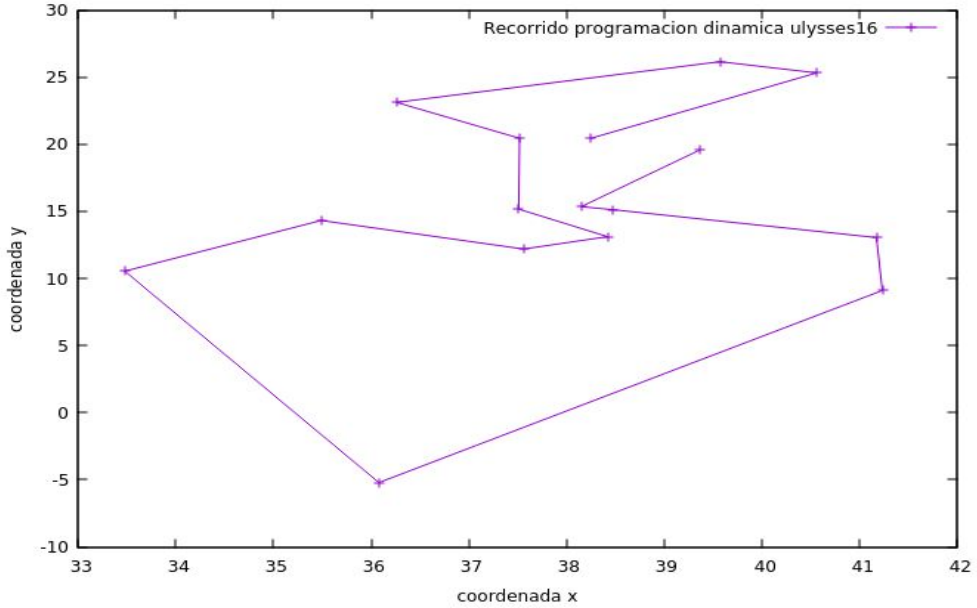
> Estas mediciones se han tomado usando ctime, en una máquina con las siguientes características:
>
> - Intel® Core™ i7-7700HQ CPU @ 2.80GHz × 8
> - 16GB de RAM
> - SO: Ubuntu 18.0404 LTS.

Con greedy:



- Distancia total recorrida: 79
- Camino final: 1 8 4 2 3 16 12 13 14 6 7 10 9 5 15 11

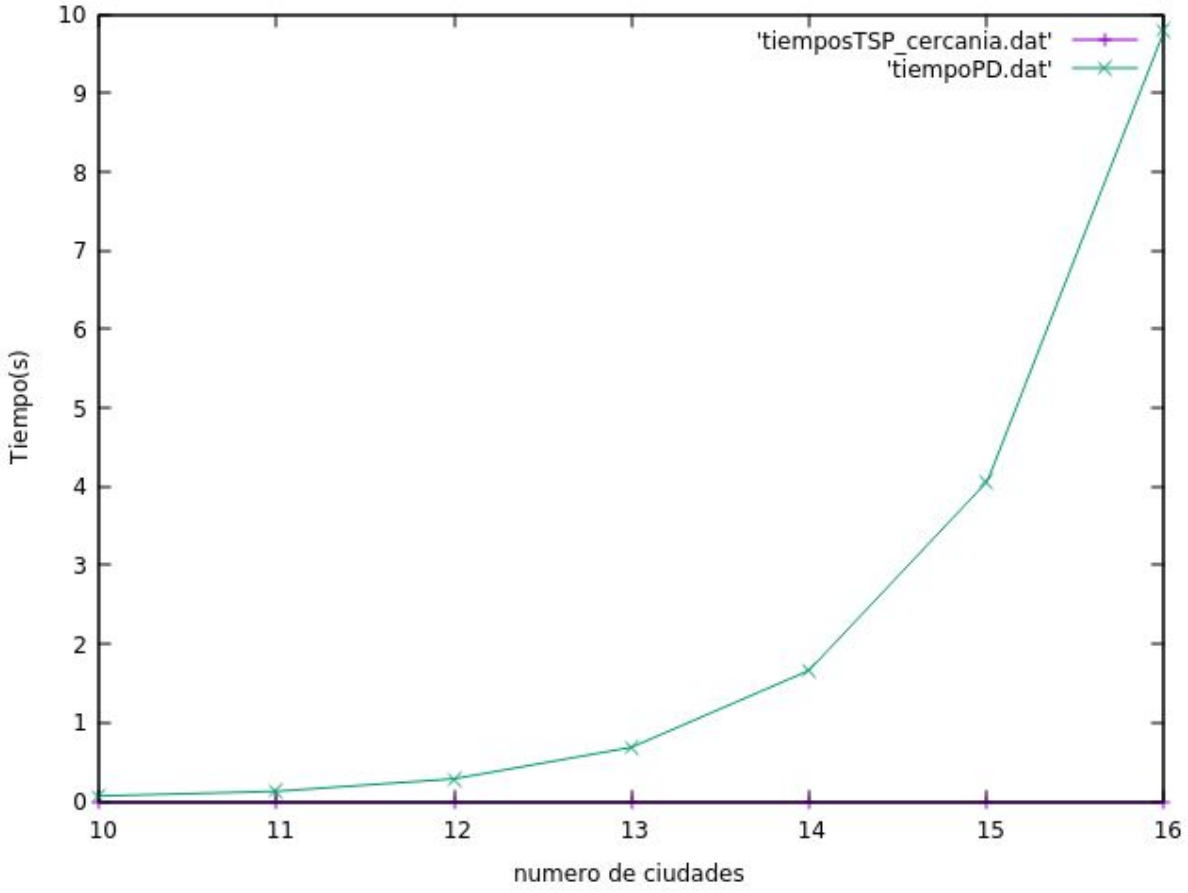
Con programación dinámica:



- Distancia total recorrida: 71
- Camino final: 1 3 2 4 8 14 7 6 15 5 11 9 10 12 13 16

Aunque el recorrido total no parece que mejore mucho, esto es debido a la pequeña cantidad de ciudades que se están ejecutando, pero con la gráfica se ve mucho más claro que el recorrido por programación dinámica es mucho más eficiente.

Ahora vamos a mostrar cómo se han comportado los algoritmos en cuestión del tiempo utilizado:



Como podemos comprobar en la gráfica, el tiempo que tarda el algoritmo de programación dinámica es exageradamente mayor que el de greedy, el cual parece inexistente al lado del otro, ya que está pegado al eje x.

Vemos que la curva ascendente de la programación dinámica corresponde con una función cuadrática, y que, con sólo 16 ciudades ya tarda 10 segundos, una barbaridad teniendo en cuenta que el greedy no se separa de las milésimas de segundos.

4. Conclusiones

Las conclusiones que hemos inferido de esta práctica son las siguientes:

- Nuestro algoritmo consigue una solución óptima fraccionando el problema en subproblemas y resolviéndolos mediante recursividad para posteriormente combinar esas soluciones y encontrar una solución final. Esto nos permite comprobar que el principio de optimalidad de Bellman es cierto

- La programación dinámica hace uso de una gran cantidad de recursos. Utiliza mucha memoria, ya que almacena los resultados de los subproblemas para utilizarlos posteriormente y consume mucho tiempo de procesamiento debido al crecimiento exponencial del tiempo frente al crecimiento de los datos. Por ello se hace prácticamente imposible encontrar la solución a problemas de gran tamaño

- Las comparativas de tiempo con otros algoritmos menos exactos (como los algoritmos greedy) son abismales debido a la enorme cantidad de tiempo que tarda el algoritmo de programación dinámica respecto a otros menos óptimos