Universidad de Huelva

Universidad de Huelva

Grado en Ingeniería Informática

Práctica 1: Estrategias Algorítmicas

Memoria de Prácticas

Autor: Alberto Fernández Merchán
Asignatura: Algorítmica y Modelos de Computación

Curso 2021/2022

Índice general

Parte I

Análisis de algoritmos exhaustivos y Divide y Vencerás: El problema del punto más cercano a otros dos.

Capítulo 1

Introducción

Dados un conjunto de puntos situados en un plano $P = \{(x_1, y_1), (x_2, y_2), ..., (x_k, y_k)\}$, se pide encontrar el punto p_i más cercano a otros dos p_j, p_k tal que la suma de la disancia de p_i a p_j más la suma de la distancia de p_i a p_k sea mínima.

En esta práctica, abordaremos el problema de dos posible formas: mediante el método **exhaustivo** y mediante la técnica de **divide y vencerás**.

Para cada metodología, estudiaremos los casos que pueden darse tanto de forma **teórica** como de forma **experimental**.

Además, se incluirán los pseudocódigos de cada algoritmo y se expondrán unas gráficas donde se podrá comparar el coste de cada uno de ellos.

Para finalizar, se incluirá el código fuente del programa donde se podrán comprobar experimentalmente los resultados obtenidos en este documento. Este programa podrá recibir datos mediante ficheros .tsp o bien, generarlos aleatoriamente.

Capítulo 2

Cálculo del tiempo teórico

2.1. Método Exhaustivo

Definición. El método exhaustivo analiza todas las combinaciones de tres puntos posibles. Termina seleccionando el conjunto de puntos con la distancia mínima.

2.1.1. Pseudocódigo y Análisis del Coste

Pseudocódigo

Algorithm 1 Método *Exhaustivo*

Entrada: T: lista de puntos
Salida: d_{min}: Distancia mínima entre tres puntos

```
1:  d_min = +∞
2:  for i; i++; i < talla(T) do
3:      for j = i; j++; j < talla(T) do
4:          for k = j; k++; k < talla(T) do
5:              if (distancia(T[i],T[j]) + distancia(T[i],T[k])) < d_min then
6:                  d_min = distancia(T[i],T[j]) + distancia(T[i],T[k])
7:              if (distancia(T[j],T[i]) + distancia(T[j],T[k])) < d_min then
8:                  d_min = distancia(T[j],T[i]) + distancia(T[j],T[k])
9:              if (distancia(T[k],T[i]) + distancia(T[k],T[j])) < d_min then
10:                 d_min = distancia(T[k],T[i]) + distancia(T[k],T[j])
11: return d_min
```

Caso Mejor

Definición. El caso mejor se da cuando el algoritmo realiza el mínimo número de operaciones elementales posible.

En el caso del método exhaustivo, el mejor caso se dará si los tres primeros puntos son los que forman la distancia mínima entre ellos. De esta forma, no se entrará en la condicional más de una vez, evitando así, 8 operaciones elementales (OE).

Igualmente, la complejidad de este algoritmo viene dada por las veces que realiza las comparaciones de las condicionales. Esta comparación no se puede evitar de ninguna forma, por lo que la complejidad será de $O(n^3)$:

$$
\begin{aligned}
T(n) &= \sum_{i=0}^{n-1}\sum_{j=i+1}^{n-1}\sum_{k=j+1}^{n-1}(1) \\
&= \sum_{i=0}^{n-1}\sum_{j=i+1}^{n-1}(n-j-1) \\
&= \sum_{i=0}^{n-1}\cdot(\sum_{j=i+1}^{n-1}(n) - \sum_{j=i+1}^{n-1}(j) - \sum_{j=i+1}^{n-1}(1)) \\
&= \sum_{i=0}^{n-1}((n^2 - n\cdot i - n) - (\frac{(n^2 - n - ni + ni - i - i^2)}{2}) - (n-i-1)) \\
&= \sum_{i=0}^{n-1}(\frac{2n^2 - 2n - 2ni - n^2 + n + i + i^2 - 2n + 2 + 2i}{2}) \\
&= \frac{1}{2}\cdot\sum_{i=0}^{n-1}(n^2 - 3n - 2ni + 3i + i^2 + 2) \\
&= \frac{1}{2}\cdot(n^3 - 3n^2 - 2n\cdot\sum_{i=0}^{n-1}(i) + 3\cdot\sum_{i=0}^{n-1}(i) + \sum_{i=0}^{n-1}(i^2) + 2n) \\
&= \frac{1}{2}\cdot(n^3 - 3n^2 - 2n\cdot(\frac{(n-1)n}{2}) + 3\cdot(\frac{(n-1)n}{2}) + (\frac{n^2(2n-2)}{6}) + 2n) \\
&= \frac{1}{2}\cdot(n^3 - 3n^2 - n^2\cdot(n-1) + \frac{3}{2}\cdot(n^2-n) + (\frac{n^2(n-1)}{3}) + 2n) \\
&= \frac{1}{2}\cdot(-3n^2 + n^2 + \frac{3}{2}(n^2-n) + 2n + \frac{n^3-n^2}{3}) \\
&= \frac{1}{2}\cdot(n^3 - 3n^2 - n^2\cdot(n-1) + \frac{3}{2}\cdot(n^2-n) + (\frac{n^2(n-1)}{3}) + 2n) \\
&= -n^2 + \frac{3}{4}(n^2) - \frac{3}{4}(n) + n + \frac{n^3-n^2}{6} \\
T(n) &= \frac{1}{6}n^3 - \frac{1}{12}n^2 + \frac{1}{4}n \rightarrow T(n) \in O(n^3)
\end{aligned}
$$

Caso Peor

Definición. El caso peor se da cuando el algoritmo realiza el mayor número de operaciones elementales posible.

En este algoritmo, el peor caso se realizará si los puntos solución son los 3 últimos del vector T. De esta forma, entrará siempre en la condicional y realizará más OE.

Como he comentado anteriormente, la operación básica de este algoritmo es la comparación en la condicional. En el peor caso, se realizará n^3 veces por lo que el orden de complejidad será de $O(n^3)$.

La demostración es análoga a la del caso mejor.

2.1.2. Gráficas de Coste

A continuación se exponen las gráficas de coste para el método exhaustivo. Como el caso peor y mejor son idénticos, solo se mostrará una gráfica.

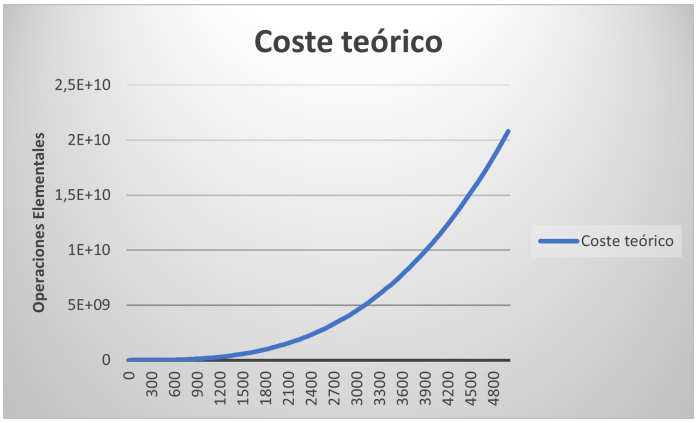

Figura 2.1: Coste teórico del Método Exhaustivo.

En la gráfica podemos observar como para tamaños más pequeños el número de operaciones elementales es prácticamente 0. Sin embargo, en cuanto comenzamos a aumentar un poco el tamaño del problema, es decir, empezamos a incluir una mayor cantidad de puntos en el vector, el número de operaciones elementales se dispara.

2.1.3. Conclusiones

El método exhaustivo, aunque es sencillo de implementar, requiere un orden de ejecución demasiado elevado. Para problemas de un tamaño considerable podría dar fallos de rendimiento.

En la gráfica podemos ver como este algoritmo, al tener orden cúbico, nos sirve para tamaños muy cercanos a cero. Pero, en cuanto empezamos a tener un gran número de puntos, el algoritmo empieza a ralentizarse y realiza una gran cantidad de operaciones elementales.

Por este motivo, es aconsejable buscar un método que reduzca el orden de complejidad de este problema.

2.2. Método Divide y Vencerás

Definición. La técnica de divide y vencerás consiste en dividir el conjunto de puntos P en otros dos conjuntos de tamaño mitad P_i y P_d. Si los tres puntos se encontraran en alguno de esos dos conjuntos, la solución se resolvería de forma **recursiva**. Sin embargo, puede que alguno de esos tres puntos se encuentren en zonas distintas. Para resolver este caso, buscamos los puntos que hay entre el punto medio (x_m) ± distancia mínima. Esos puntos serán los que tengan la menor distancia entre sí.

2.2.1. Pseudocódigo y Análisis del Coste

Pseudocódigo

(Siguiente página)

Caso Mejor

Definición. El mejor caso se dará cuando el algoritmo haga el mínimo número de $OE's$. En el caso de este algoritmo sucede cuando en los bucles *while* se encuentre una distancia mayor que la d_{min} en la primera iteración. De esta forma, el coste del bucle sería **constante**.

Caso Peor

Definición. El peor caso se dará cuando el algoritmo haga el máximo número de $OE's$. En este algoritmo ocurre cuando los bucles *while* realicen el máximo número de iteraciones, es decir, que su coste será de $\frac{n}{2}$.

Al tener el vector ordenador tanto por las coordendas X como por las coordenadas Y, no se nos presenta ningún caso en el que podamos tener el vector no ordenado. Por ejemplo, si no hubiese decidido ordenar también por la coordenada Y podría darse el caso en el que todos los puntos compartieran la misma coordenada X y, por tanto, el vector no se ordenaría correctamente. En ese caso el algoritmo acabaría por realizar tantas operaciones como el algoritmo exhaustivo.

No obstante, en el algoritmo de ordenación que he usado he añadido una condición para que, en caso de igualdad en las coordenadas X, los ordene en función de la coordenada Y. De esta forma evito que el vector pueda quedar desordenado en algún momento y que el algoritmo divide y vencerás no funcione correctamente.

Por lo tanto el sistema recurrente que habrá que solucionar será:

$$T(n) = \begin{cases} c_1 & \text{si } n \leq 4, \\ 2 \cdot T(\frac{n}{2}) + n \cdot \log_2 n \cdot c_2 + c_3 \cdot n & \text{si } n > 4 \end{cases}$$

El sistema recurrente está compuesto por el caso base (c_1), que es constante porque el algoritmo exhaustivo se ejecuta un número determinado de veces, y la ecuación recurrente, en la que podemos observar que el problema se divide en dos subproblemas de la mitad del tamaño del problema original.

Además, añadimos el coste de ordenar el vector de puntos ($O(n \log_2 n)$) y el coste de combinar los resultados $O(n)$.

Para solucionar el sistema recurrente anterior existen 3 formas de hacerlo:

Algorithm 2 Método *Divide y Vencerás*

Entrada: T: lista de puntos ; i: índice izquierdo; d:Índice derecho
Salida: d_{min}: Distancia mínima entre tres puntos

```
1: Ordenar(T)// Usando el algoritmo de HeapSort
2: if (d − i + 1) >= 4 then
3:     xm = (d + i)/2
4:     di = dyv(T, i, xm)
5:     dd = dyv(T, xm + 1, d)
6:
7:     d_min = minimo(di, dd)
8:
9:     a = x_m
10:    fin = false
11:    while a >= i AND NOT(fin) do
12:        if T[x_m + 1].getX() − T[a].getX() > d_min then
13:            fin = true
14:        else a = a − 1
15:
16:    b = x_m + 1
17:    fin = false
18:    while b <= d AND NOT(fin) do
19:        if T[b].getX() − T[x_m].getX() > d_min then
20:            fin = true
21:        else b = b + 1
22:
23:    for k = a + 1; k <= x_m; k + + do
24:        for m = x_m + 1; m <= b − 1; m + + do
25:            for n = m + 1; n <= d; n + + do
26:                if distancia(T[k], T[m]) + distancia(T[k], T[n]) < d_min then
27:                    d_min = distancia(T[k], T[m]) + distancia(T[k], T[n])
28:                if distancia(T[m], T[k]) + distancia(T[m], T[n]) < d_min then
29:                    d_min = distancia(T[m], T[k]) + distancia(T[m], T[n])
30:                if distancia(T[n], T[m]) + distancia(T[n], T[k]) < d_min then
31:                    d_min = distancia(T[n], T[m]) + distancia(T[n], T[k])
32:
33:    for k = a + 1; k <= x_m; k + + do
34:        for m = k + 1; m <= x_m; m + + do
35:            for n = x_m + 1; n <= b; n + + do
36:                if distancia(T[k], T[m]) + distancia(T[k], T[n]) < d_min then
37:                    d_min = distancia(T[k], T[m]) + distancia(T[k], T[n])
38:                if distancia(T[m], T[k]) + distancia(T[m], T[n]) < d_min then
39:                    d_min = distancia(T[m], T[k]) + distancia(T[m], T[n])
40:                if distancia(T[n], T[m]) + distancia(T[n], T[k]) < d_min then
41:                    d_min = distancia(T[n], T[m]) + distancia(T[n], T[k])
       return d_min
42: else return exhaustivo(T, i, d)
```

Expansión de Recurrencias

$$
\begin{aligned}
T(n) &= 2 \cdot T(\frac{n}{2}) + n \cdot \log_2 n + n \\
&= 2 \cdot (2 \cdot T(\frac{n}{4}) + \frac{n}{2} \cdot \log_2 \frac{n}{2} + \frac{n}{2}) + n \cdot \log_2 n + n \\
&= 2^2 \cdot T(\frac{n}{4}) + n \cdot \log_2 \frac{n}{2} + n + n \cdot \log_2 n + n \\
&= ... \\
&= 2^i \cdot T(\frac{n}{2^i}) + \sum_{j=0}^{i} n + \sum_{j=0}^{i} n \cdot \log_2 (\frac{n}{2^j}) \\
&= 2^i \cdot T(\frac{n}{2^i}) + n \cdot (i+1) + n \cdot \sum_{j=0}^{i-1} \log_2 n - n \cdot \sum_{j=0}^{i} \log_2 2^j \\
&= 2^i \cdot T(\frac{n}{2^i}) + n \cdot (i+1) + n \cdot \log_2 n \cdot (i) - n \cdot \sum_{j=0}^{i} j \\
&= 2^i \cdot T(\frac{n}{2^i}) + n \cdot (i+1) + n \cdot \log_2 n \cdot (i) - n \cdot \frac{i \cdot (i+1)}{2}
\end{aligned}
$$

Como el caso base es $T(1) = O(1) \to T(1) = 1 \to 1 = \frac{n}{2^i} \to i = \log_2 (n)$

$$
\begin{aligned}
T(n) &= 2^{\log_2 n} + n \cdot (\log_2 n + 1) + n \cdot \log_2 n \cdot \log_2 n - n \cdot \frac{\log_2^2 n + \log_2 n}{2} \\
&= 2^{\log_2 n} + n \cdot (\log_2 n + 1) + n \cdot \log_2^2 n - \frac{n}{2} \cdot (\log_2^2 n + \log_2 n) \\
&= 2n + n \cdot \log_2 n + n \cdot \log_2^2 n - \frac{n}{2} \cdot (\log_2^2 n + \log_2 n) \\
T(n) &\in O(n \cdot \log_2^2 n)
\end{aligned}
$$

Ecuación Característica

$T(n) = 2 \cdot T(\frac{n}{2}) + n \cdot \log_2 n + n$

Hacemos un cambio de variable $(2^k = n) \leftrightarrow (k = \log_2 n)$

$T(n) - 2 \cdot T(\frac{n}{2}) = n \cdot \log_2 n + n$

$T(2^k) - T(2^{k-1}) = 2^k \cdot k + 2^k$

Para convertirla en una ecuación homogénea hacemos otro cambio de variable $(T(2^k) = t_k)$

$t_k - t_{k-1} = 2^k \cdot (k+1)$

$(x-2) \cdot (x-2)^2$

$r_1 = 2$ (Raíz Triple)

De esta forma, el polinomio característico de esta recurrencia será:

$T(k) = c_1 \cdot 2^k + c_2 \cdot k \cdot 2^k + c_3 \cdot k^2 \cdot 2^k$

Deshaciendo los cambios de variables nos queda...

$T(n) = c_1 \cdot n + c_2 \cdot n \cdot \log_2 n + c_3 \cdot n \cdot (\log_2 n)^2$

$T(n) \in O(n \cdot \log_2^2 n)$

Teorema Maestro Según el teorema maestro:

$$T(n) \in \begin{cases} O(n^{\log_b a}) & \text{si } a > b^k \\ O(n^k \cdot \log_2^{p+1} n) & \text{si } a = b^k \\ O(n^k \cdot \log_2^p n) & \text{si } a < b^k \end{cases}$$

Como $a = 2$, $b = 2$ y $k = 1$ el órden de complejidad es $O(n \cdot \log_2^2 n)$

2.2.2. Gráficas de Coste

A continuación se exponen las gráficas de coste para el método divide y vencerás.

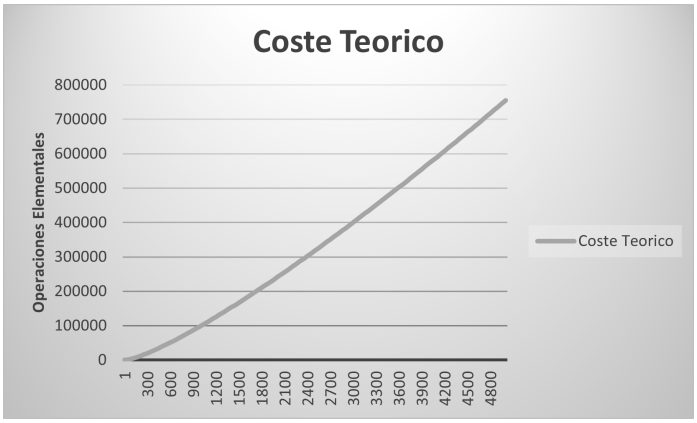

Figura 2.2: Coste teórico del Método Divide y Vencerás.

2.2.3. Conclusiones

La técnica de Divide y Vencerás es una técnica más compleja que el método exhaustivo. Este método utiliza la recursividad para dividir el problema en un tamaño menor. De esta forma consigue una eficacia mucho mejor que la del método exhaustivo tal y como podemos ver en la gráfica.

El método de Divide y Vencerás divide el problema de tamaño n en 2 subproblemas de tamaño $\frac{n}{2}$. De esta forma, se soluciona el problema de una forma más rápida al tener menos elementos cada subproblema.

Tras los análisis de coste confirmamos lo que ya habíamos predicho. El coste de ejecutar el algoritmo Divide y Vencerás es de $O(n \log_2^2 n)$, mientras que el del método exhaustivo tiene un coste cúbico ($O(n^3)$). A continuación compararemos ambas gráficas y veremos como evoluciona cada orden de complejidad.

2.3. Comparación Gráficas Teóricas

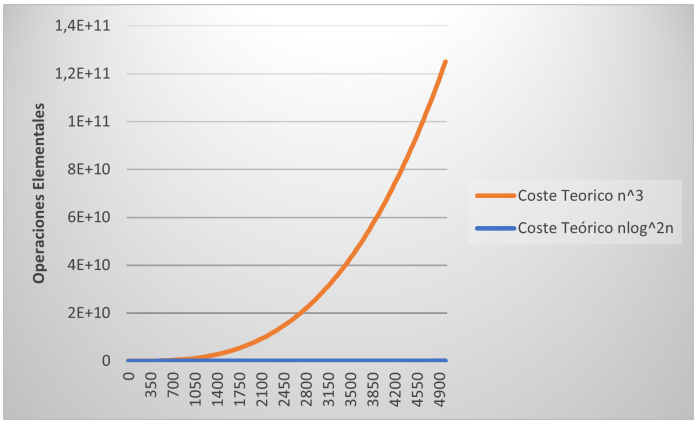

Figura 2.3: Coste teórico del Método Divide y Vencerás.

Podemos observar como la gráfica del método exhaustivo crece mucho más rápido que la del método divide y vencerás. Mientras que la función cúbica llega tener una gran número de operaciones elementales, la función logarítmica se mantiene casi constante en comparación.

Capítulo 3

Cálculo del tiempo experimental

3.1. Método Exhaustivo

3.1.1. Ejecuciones

Caso Aleatorio

Talla 200 Tras diez ejecuciones, el tiempo que tarda el algoritmo exhaustivo en encontrar la solución en un vector de 200 puntos es:

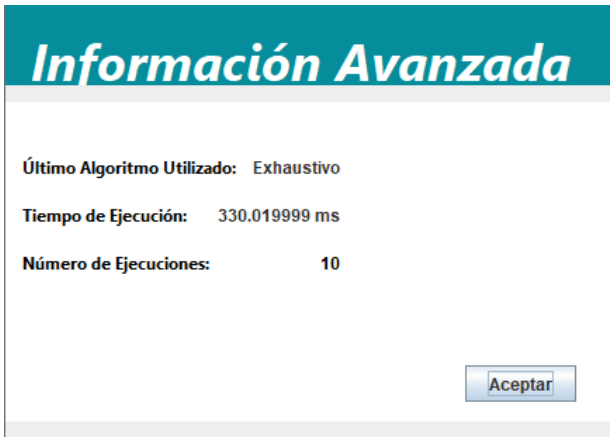

Figura 3.1: Tiempo Empírico tras 10 ejecuciones. Algoritmo Exhaustivo (1500 puntos)

Talla 500 Tras diez ejecuciones, el tiempo que tarda el algoritmo exhaustivo en encontrar la solución en un vector de 500 puntos es:

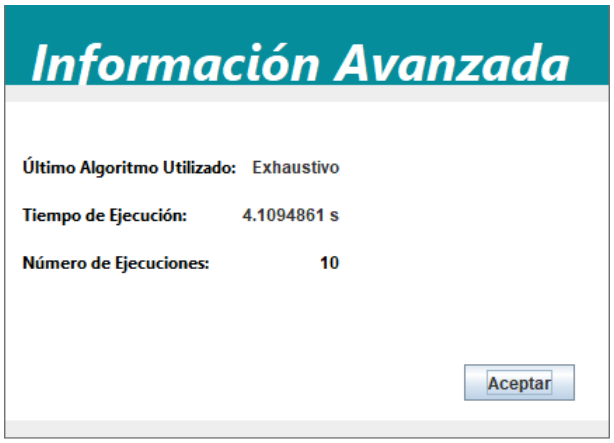

Figura 3.2: Tiempo Empírico tras 10 ejecuciones. Algoritmo Exhaustivo (500 puntos)

Talla 1500 Tras diez ejecuciones, el tiempo que tarda el algoritmo exhaustivo en encontrar la solución en un vector de 1500 puntos es:

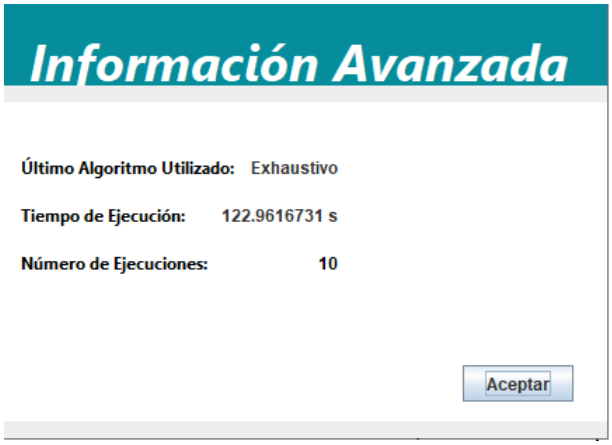

Figura 3.3: Tiempo Empírico tras 10 ejecuciones. Algoritmo Exhaustivo (1500 puntos)

Talla 5000* El algoritmo exhaustivo es tan ineficiente que mi ordenador no puede soportar el coste computacional que este supone para un tamaño de problema de 5000 puntos. He intentado reducir el tamaño, pero lo máximo que llega a procesar son 2000 puntos y es una talla muy similar a la anterior.

3.1.2. Conclusiones

Tras la ejecución del algoritmo exhaustivo podemos comprobar como usarlo para problemas de gran tamaño es muy ineficiente llegando a superar los 2 minutos en ejecutarse.

No he diferenciado el caso mejor y el peor ya que el algoritmo consiste en 3 bucles anidados que se ejecutan completamente. No hay ninguna forma de reducir el número de operaciones elementales que realiza este algoritmo y, por tanto, el coste computacional que requiere es el mismo en todos los casos.

No obstante, para encontrar la solución a problemas pequeños es casi inmediato. Es por esto que lo uso en el caso base del algoritmo divide y vencerás.

3.2. Método Divide y Vencerás

3.2.1. Ejecuciones

Talla 200 Calculamos el tiempo que tarda el algoritmo divide y vencerás en encontrar la distancia mínima en 200 puntos:

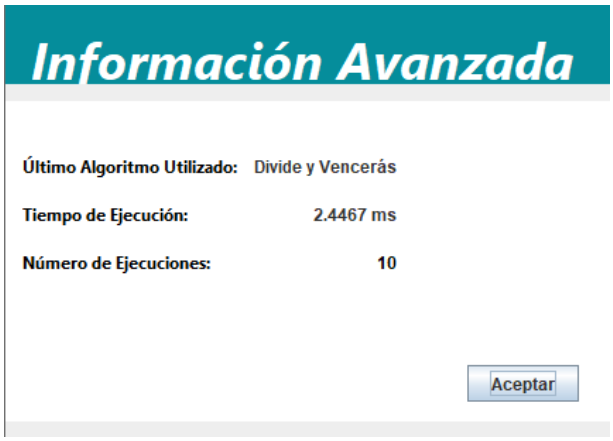

Figura 3.4: Tiempo Empírico tras 10 ejecuciones. Algoritmo Divide y Vencerás (200 puntos)

Talla 500 Calculamos el tiempo que tarda el algoritmo divide y vencerás en encontrar la distancia mínima en 500 puntos:

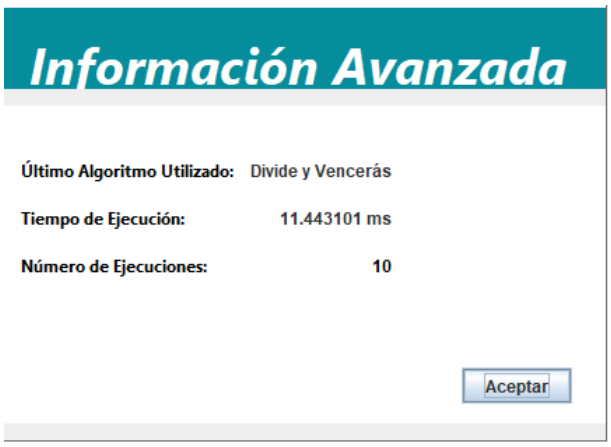

Figura 3.5: Tiempo Empírico tras 10 ejecuciones. Algoritmo Divide y Vencerás (500 puntos)

Talla 1500 Calculamos el tiempo que tarda el algoritmo divide y vencerás en encontrar la distancia mínima en 1500 puntos:

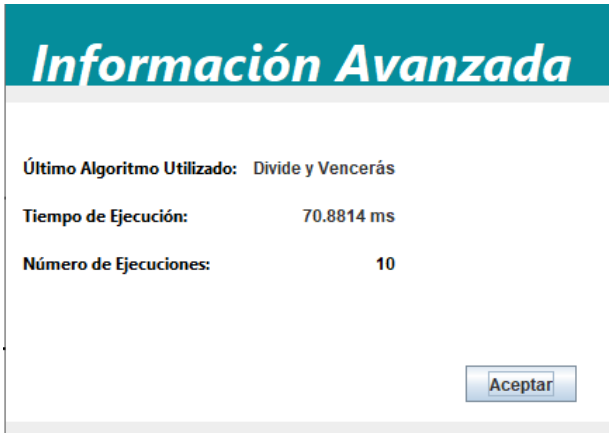

Figura 3.6: Tiempo Empírico tras 10 ejecuciones. Algoritmo Divide y Vencerás (1500 puntos)

Talla 5000 El algoritmo de divide y vencerás es mucho más eficiente que el exhaustivo y, por tanto, con un tamaño de 5000 puntos mi ordenador puede procesarlo correctamente.
Calculamos el tiempo que tarda el algoritmo divide y vencerás en encontrar la distancia mínima en 5000 puntos:

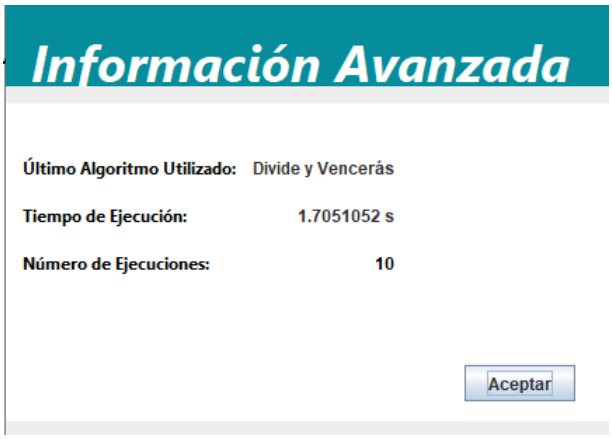

Figura 3.7: Tiempo Empírico tras 10 ejecuciones. Algoritmo Divide y Vencerás (5000 puntos)

Caso Peor

La diferencia con el caso mejor es poco apreciable ya que ambos tiene coste $O(n \log_2^2 n)$. Como he utilizado el algoritmo de **HeapSort**, el peor caso es del mismo orden que el mejor y por tanto no cambia su complejidad.

En lo único que podría cambiar sería en el valor de ciertas constantes relacionadas con el número de comparaciones que se realizan, pero en notación asintótica no se tienen en cuenta esas constantes por lo que sigue siendo $O(n \log_2^2 n)$.

Casos Límite

A continuación estudiaré los 3 tipos de casos límite que podemos encontrar en este tipo de problema.

1. **Solución a la izquierda del vector** En este caso, la solución al problema se encuentra entre los primeros puntos de la lista (después de ser ordenados). En esta situación, el algoritmo divide y vencerás divide el vector hasta que encuentra la solución y la propaga mientras recombina los resultados.

   Los puntos serían los siguientes:

```
1 0.0000 0.00000
2 1.0000 0.00001
3 2.0000 0.00002
4 80.75100 9.83700
5 50.87800 1.04090
6 90.75100 1.13610
7 100.87800 1.27580
8 800.87800 1.49810
9 900.64000 1.50440
10 40.11640 1.21230
11 30.01480 1.14880
12 20.05290 1.13610
```

Figura 3.8: Ejemplo del caso de la solución a la izquierda del vector (texto)

El gráfico se vería de la siguiente forma:

Figura 3.9: Ejemplo del caso de la solución a la izquierda del vector (puntos)

2. **Solución en el centro del vector** Este es el caso más complejo. La solución se encuentra en la mitad del vector, entre dos subrproblemas. El algoritmo, tras encontrar la solución de un subproblema, busca en los puntos que hay entre el punto extremo de esa solución y el resto de puntos del interior del vector. Los puntos serían los siguientes: // El gráfico se vería de la siguiente forma:

```
1 0.0000 0.00000
2 1000.0000 0.00001
3 2000.0000 0.00002
4 3000.75100 9.83700
5 5000.87800 1.04090
6 5500.75100 1.13610
7 5750.87800 1.27580
8 8000.87800 1.49810
9 9000.64000 1.50440
10 10000.11640 1.21230
11 11000.01480 1.14880
12 12000.05290 1.13610
```

Figura 3.10: Ejemplo del caso de la solución en el centro del vector (texto)

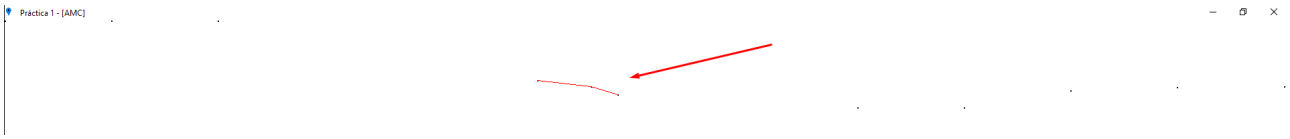

Figura 3.11: Ejemplo del caso de la solución en el centro del vector (puntos)

3. **Solución a la derecha del vector** En este caso, la solución se encuentra entre los puntos del final de la lista (tras ser ordenada). Divide y vencerás encuentra la solución cuando realiza la última subdivisión del problema y propaga la solución mientras combina los resultados. Los puntos serían los siguientes: // El gráfico se vería de la siguiente forma:

```
1 0.0000 0.00000
2 1000.0000 0.00001
3 2000.0000 0.00002
4 3000.75100 9.83700
5 4000.87800 1.04090
6 5000.75100 1.13610
7 6000.87800 1.27580
8 7000.87800 1.49810
9 8000.64000 1.50440
10 9000.11640 1.21230
11 9500.01480 1.14880
12 9750.05290 1.13610
```

Figura 3.12: Ejemplo del caso de la solución a la derecha del vector (texto)

Figura 3.13: Ejemplo del caso de la solución a la derecha del vector (puntos)

3.2.2. Conclusiones

Tras ejecutar el algoritmo de divide y vencerás para diferentes tamaños del problema, nos fijamos en que el tiempo de ejeución aumenta muy poco a poco en comparación con el algoritmo exhaustivo. Además, al usar el algoritmo de ordenación de HeapSort en lugar del de ordenación rápida, nos aseguramos que el coste sea el mismo tanto en el mejor como en el peor caso.

Capítulo 4

Comparación de Resultados

A continuación se muestra la gráfica comparativa de los costes empíricos de ambos algoritmos:

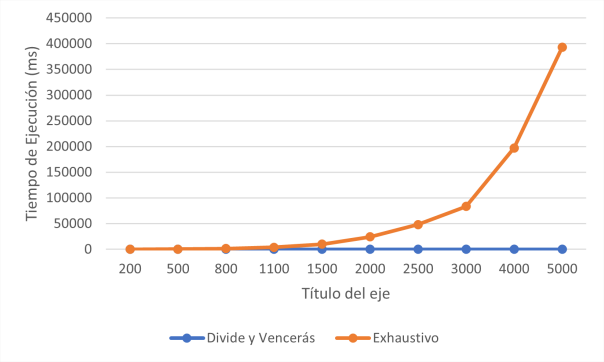

Figura 4.1: Gráfica comparativa del método exhaustivo y el método empírico

He añadido varias simulaciones más para poder observar como la curva de la gráfica coincide con la curva de la gráfica teórica. Los resultados pueden no coincidir exactamente debido a diversos factores como, por ejemplo, el rendimiento del procesador en el momento de la simulación o las constantes del estudio teórico.

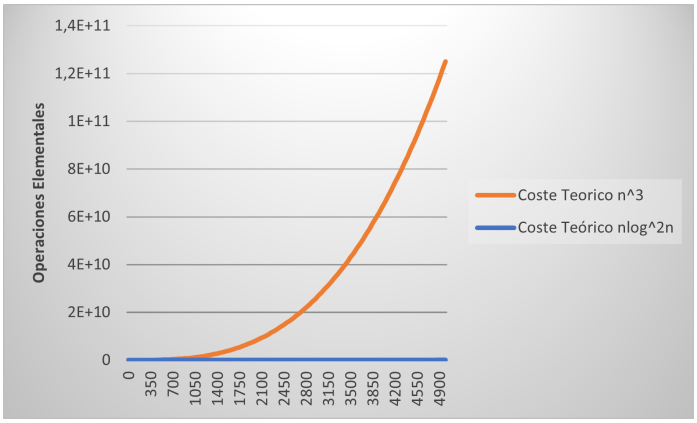

Figura 4.2: Coste teórico del Método Divide y Vencerás.

Parte II

Análisis de Algoritmos Voraces: El problema del camino mínimo.

Capítulo 5

Introducción

Este problema es uno de los más conocidos en el campo de la optimización por su gran cantidad de usos. Consiste en, dadas una series de ciudades (puntos), encontrar los caminos mínimos desde una ciudad concreta (origen) hasta el resto de ciudades.

Para solucionar este problema encontramos el algoritmo de **Dijkstra**, es un algoritmo voraz que consiste en elegir, en cada iteración, un nuevo camino siguiendo como criterio la ciudad más cercana a la ciudad de origen.

Para la implementación de este algoritmo usaré dos tablas auxiliares donde guardaré las distancias que hay entre cada punto con el resto de puntos (Tabla de Costes) y un vector con las distancias mínimas en cada momento.

A continuación, se muestran los cálculos de la eficiencia del algoritmo de Dijkstra, el pseudocódigo correspondiente y las comparaciones entre el estudio teórico y el experimental.

Capítulo 6

Cálculo del tiempo teórico

6.1. Pseudocódigo y Análisis del Coste

Pseudocódigo: A continuación se muestra el pseudocódigo del algoritmo de Dijkstra.

Algorithm 3 Algoritmo de **Dijkstra**

Entrada: P: Punto de Origen, L: Tabla de pesos, T: Vector de puntos
Salida: $Distancias$: Vector de distancias mínimas

```
1: Candidatos = {2, 3, ..., n}
2: Solucion = {1}
3: for i = 2; i < talla(T); i++ do
4:     Distancias[i] = L[1][0]
5:     for i = 1; k < talla(T) − 2; k++ do
6:         v = elige el punto que más cerca esté del origen
7:         Candidatos = C − v
8:         Solucion = S + v
9:         for j = 1; j < talla(Candidatos); j++ do
10:            Distancias[Candidatos[j]] = minimo(Distancias[Candidatos[j]], Distancias[T[v]] + L[v][Candidatos[j]])
11: return Distancias
```

Análisis de Casos

Definición: El mejor caso se dará cuando el algoritmo haga la mínima cantidad de operaciones elementales. En el caso de este algoritmo, al no haber ninguna estructura condicional que permita bifurcar la ejecución, es el mismo que en el caso peor.

El algoritmo explora **todas** las posibles soluciones mediante un bucle *for*. Por ello no es posible disminuir el número de operaciones elementales que realiza.

Para calcular el coste de este algoritmo tendremos que dividirlo en dos partes:

1. **Inicialización:** Como está formada por 2 bucles **no anidados**, su orden de complejidad es de $O(n)$
2. **Elección de v:** Recorre el vector D(n elementos) una vez por cada candidato que hay (n-1 candidato), por tanto, el coste será $O(n^2)$
3. **Bucle Voraz:** Como está formado por 2 bucles **anidados**, su orden de complejidad es de $O(n^2)$

Por lo tanto si sumamos todas las complejidades: $T(n) = n + n^2 + n^2 = 2 \cdot n^2 + n \rightarrow T(n) \in O(n^2)$

6.2. Gráficas de Coste

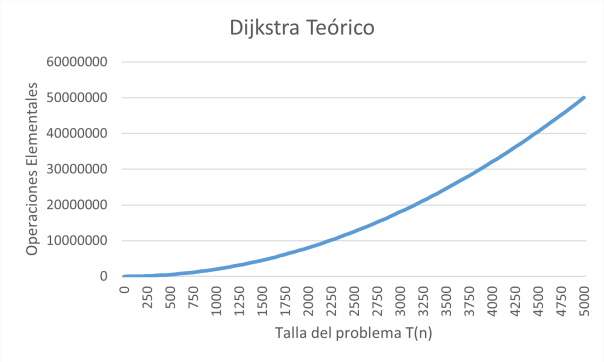

Figura 6.1: Gráfica teórica del algoritmo de Dijkstra.

6.3. Conclusiones

Tras haber analizado el algoritmo podemos comprobar que se trata de un algoritmo polinomial, en concreto, un algoritmo cuadrático. Esto coincide con la definición de algoritmos voraces, ya que solucionan problemas con un coste polinomial y aportando la solución óptima.

El algoritmo de Dijkstra podría mejorar su rendimiento ordenando el conjunto de candidatos en función de la distancia con el punto de origen como un montículo de mínimos. De esta forma el coste de extraer un punto candidato sería logarítmico ($O(\log_2 n)$). De esta forma, cuando queramos modificar la distancia del nodo seleccionado tan solo tendremos que utilizar la operación de flotar ($O(\log_2 n)$) y, por tanto, en el peor de los casos nos daría un coste de ($O(n \cdot \log_2 n)$).

Sin embargo, esta solución daría resultados evidentes tan solo en grafos muy densos. En nuestro caso la implementación polinómica nos sirve para la mayoría de los casos propuestos.

Capítulo 7

Cálculo del tiempo experimental

Talla 200 El algoritmo de Dijkstra, tras 10 ejecuciones y con una talla de 200 puntos tarda:

Figura 7.1: Tiempo del algoritmo de Dijkstra tras ejecutarlo 10 veces sobre un problema de 200 puntos

Talla 500 El algoritmo de Dijkstra, tras 10 ejecuciones y con una talla de 500 puntos tarda:

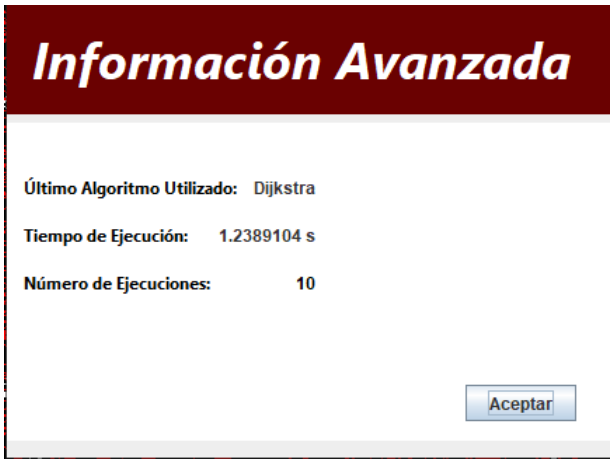

Figura 7.2: Tiempo del algoritmo de Dijkstra tras ejecutarlo 10 veces sobre un problema de 500 puntos

Talla 1500 El algoritmo de Dijkstra, tras 10 ejecuciones y con una talla de 15000 puntos tarda:

Figura 7.3: Tiempo del algoritmo de Dijkstra tras ejecutarlo 10 veces sobre un problema de 1500 puntos

Talla 4000* El algoritmo de Dijkstra, tras 10 ejecuciones y con una talla de 4000 puntos tarda: Debido a que con 5000 puntos la máquina virtual de Java sufre un error de agotamiento de memoria, he decidido utilizar una talla de 4000 puntos.

Figura 7.4: Tiempo del algoritmo de Dijkstra tras ejecutarlo 10 veces sobre un problema de 4000 puntos

7.1. Gráficas de Coste

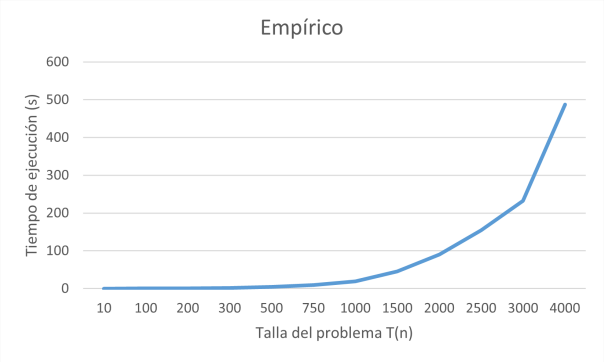

Figura 7.5: Gráfica empírica del algoritmo de Dijkstra.

7.2. Conclusiones

Tras comprobar el tiempo que tarda el algoritmo de Dijkstra en ejecutarse para varios tamaños, podemos concluir que, para tallas de problema grandes, comienza a ser ineficiente.

En mi caso, he intentado realizar un estudio empírico de diez ejecuciones seguidas sobre un vector de 5000 puntos, sin embargo, la máquina virtual de Java ha producido un error de agotamiento de memoria.

Capítulo 8

Resultado sobre conjunto de datos

A continuación mostraré como funciona el algoritmo de Dijkstra con tres conjuntos de datos. Mostraré tres apartados de cada conjunto: El mapa de todas las conexiones, el mapa con las conexiones mínimas y, por último, mostraré la solución óptima siendo esta la suma de todas las distancias mínimas desde el punto inicial hasta el resto de puntos.

8.1. Berlin52

Visualización de grafo Los caminos se ven de la siguiente forma:

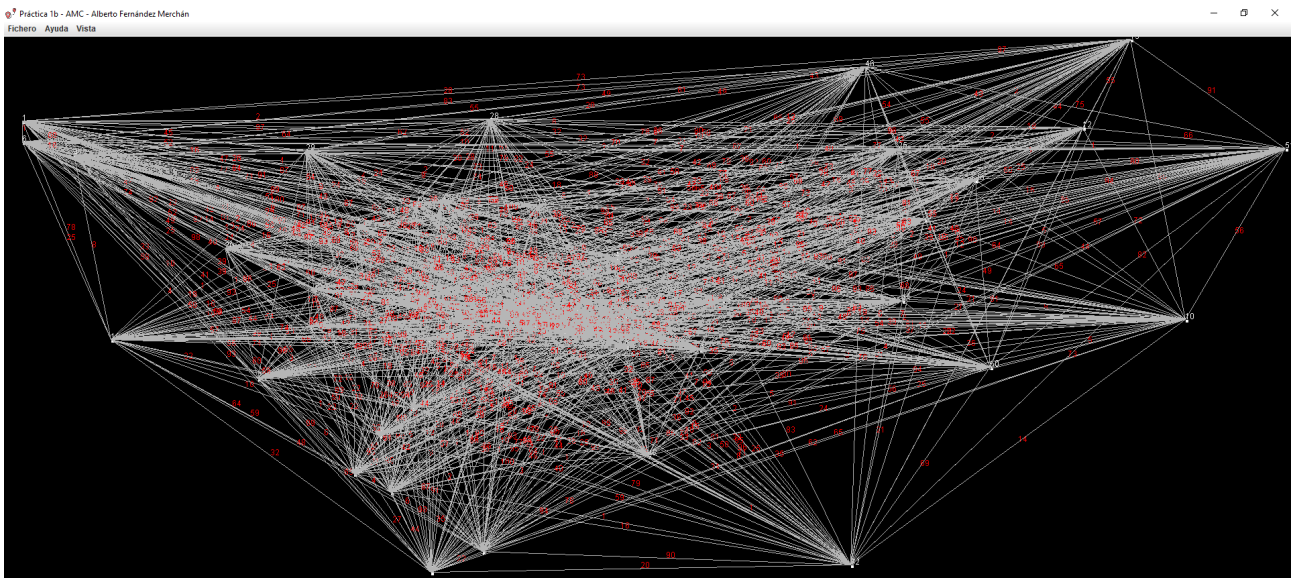

Figura 8.1: Visualización de los caminos de berlin52.

Visualización de caminos mínimos Y los caminos mínimos se muestran en la siguiente imagen:

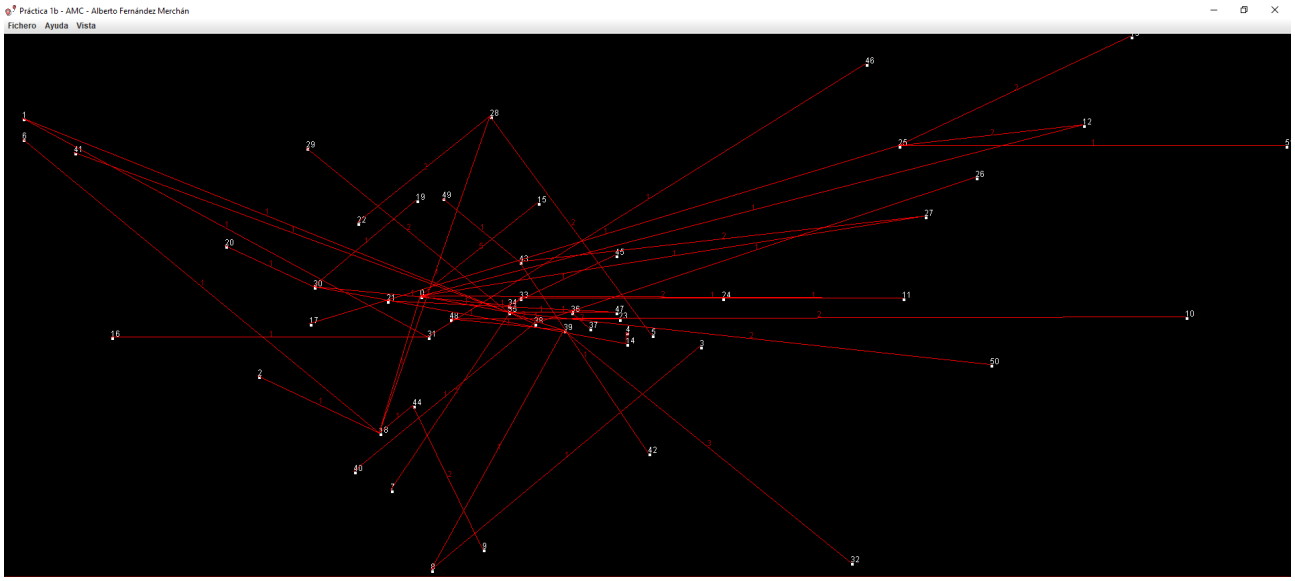

Figura 8.2: Visualización de los caminos mínimos de berlin52.

Solución Óptima La solución óptima para este conjunto de datos es de 162.

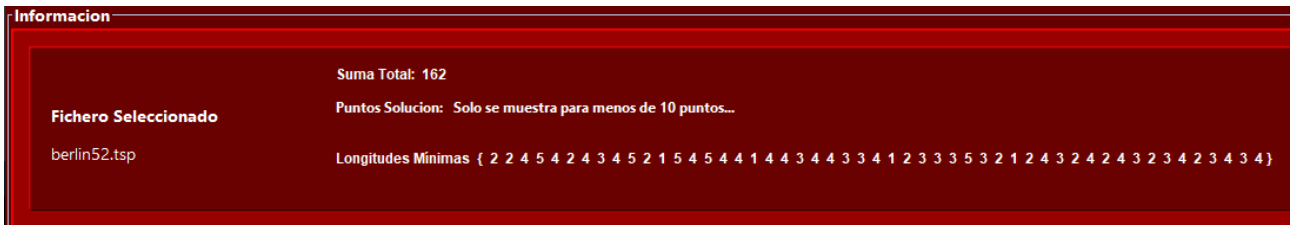

Figura 8.3: Solución óptima del conjunto de datos de berlin52.

8.2. Ch130

Visualización de grafo Los caminos se ven de la siguiente forma:

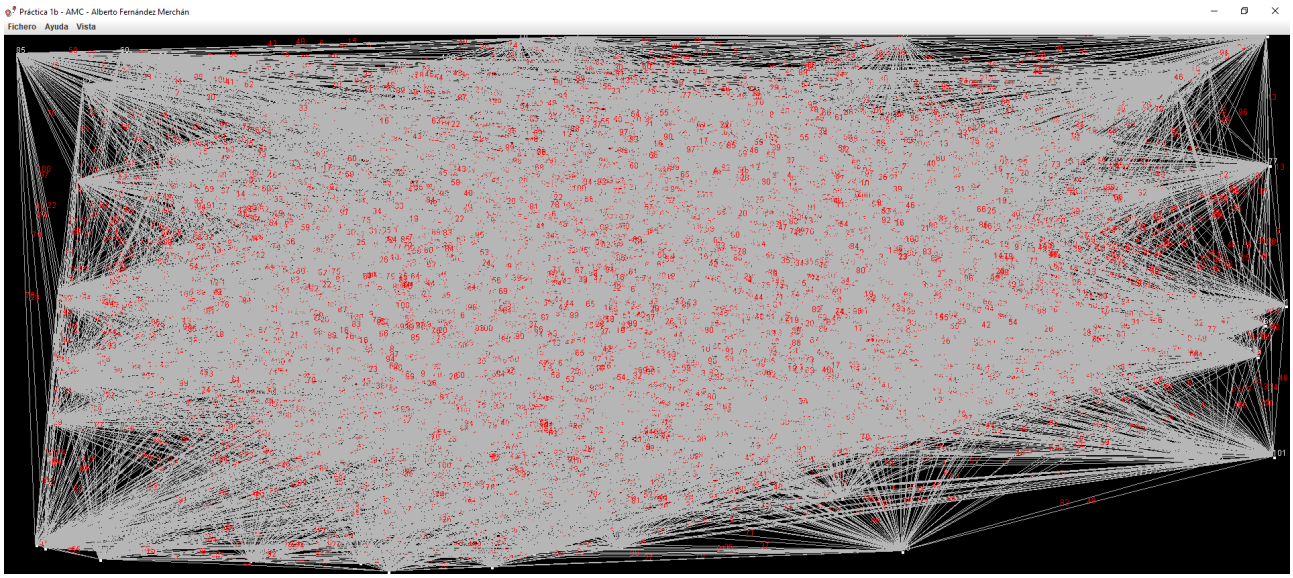

Figura 8.4: Visualización de los caminos de ch130.

Visualización de caminos mínimos Y los caminos mínimos se muestran en la siguiente imagen:

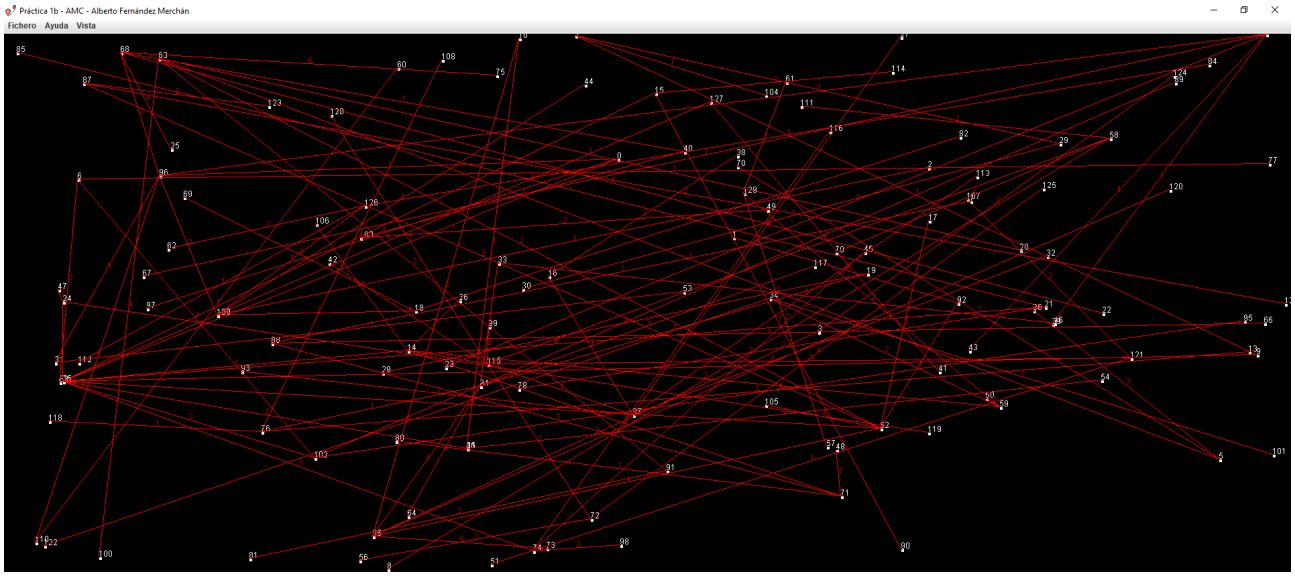

Figura 8.5: Visualización de los caminos mínimos de ch130.

Solución Óptima La solución óptima para este conjunto de datos es de 788.

Figura 8.6: Solución óptima del conjunto de datos ch130

8.3. Ch150

Visualización de grafo Los caminos se ven de la siguiente forma:

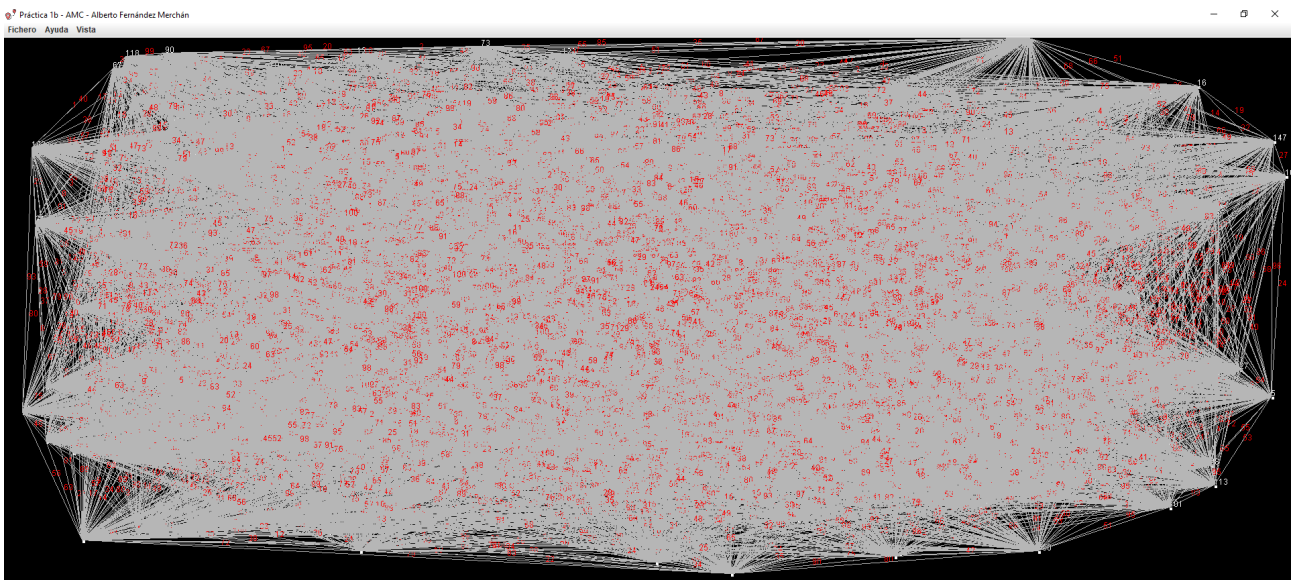

Figura 8.7: Visualización de los caminos de ch150.

Visualización de caminos mínimos Y los caminos mínimos se muestran en la siguiente imagen:

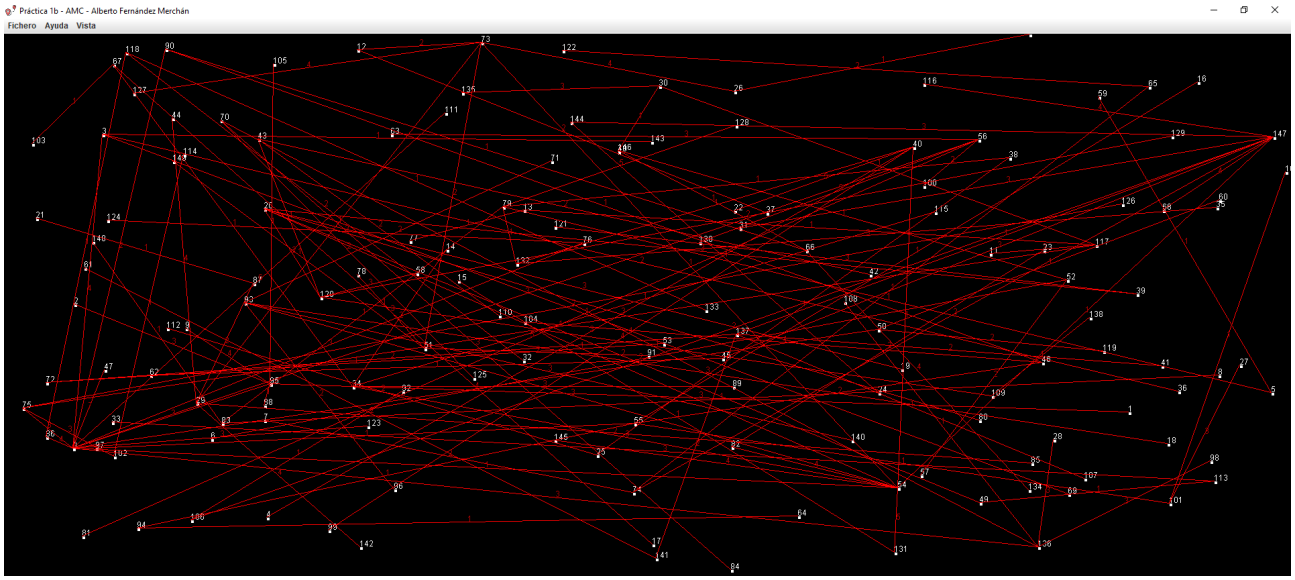

Figura 8.8: Visualización de los caminos mínimos de ch150.

Solución Óptima La solución óptima para este conjunto de datos es de 807.

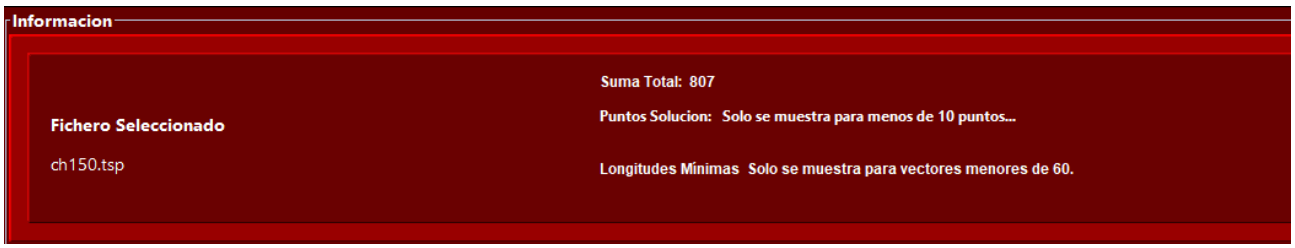

Figura 8.9: Solución Óptima del conjunto de datos ch150.

Capítulo 9

Comparación de Resultados

A continuación se muestran ambas gráficas sobre el algoritmo de Dijkstra. En primer lugar mostraré la gráfica del estudio teórico:

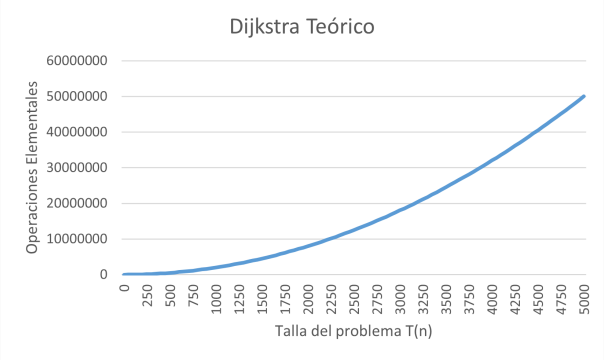

Figura 9.1: Gráfica teórica del algoritmo de Dijkstra.

Y a continuación, la gráfica del estudio empírico:

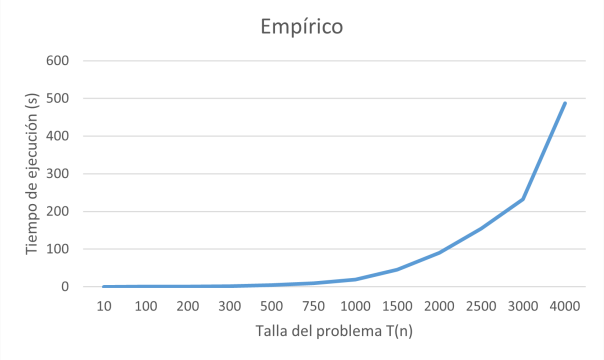

Figura 9.2: Gráfica empírica del algoritmo de Dijkstra.

Podemos observar como la gráfica teórica está más suavizada que la empírica. Esto es debido al número de simulaciones que hemos realizado en el estudio empírico (12), en comparación con las del estudio teórico (21). Debido a esta diferencia, el incremento de tiempo en la gráfica empírica resulta más abrupto que en la teórica.

Sin embargo, podemos determinar que la gráfica experimental sigue la misma función que hemos calculado en el estudio teórico: una función de orden cuadrático.