Algorítmica y Modelos de Computación

Práctica 1

Estrategias Algorítmicas

Universidad de Huelva

Realizado por: Álvaro Esteban Muñoz y Álvaro Díaz Rivero

Índice

Parte 1: Análisis de algoritmos exhaustivos y Divide y Vencerás

Análisis del algoritmo de resolución exhaustivo

```
procedimiento Triangulo Exhaustivo(Punto T[], int izq, int der) → 1 paso de vector
    var: n = der – izq + 1
    Triangulo minT <= nuevo Triangulo(T[izq], T[izq + 1], T[izq + 2]) → 3 acceso al vector
    para i = 0 hasta n hacer → 1 asig + 1 comparación + 1 inc + 1 salto + T(bucle1)
        para j = i+1 hasta n-1 hacer → 1 asig + 2 ops + 1 comp + 1 inc + 1 salto + T(bucle2)
            para k = j+1 hasta n-2 hacer → 1asig + 2ops + 1comp + 1 inc + 1 salto + T(bucle3)
                Triangulo auxT <= nuevo Triangulo(T[i], T[j], T[k]) → T(crearTriangulo)
                si auxT.GetPerim() < minT.GetPerim() entonces → 1 comp + 1 salto
                    minT <= auxT → 1 asig
                sino si auxT.GetPerim() = minT.GetPerim() entonces → 1 comp + 1 salto
                    si auxT.GetArea() > minT.GetArea() entonces → 1 comp + 1 salto
                        minT <= auxT → 1 asig
                    fsi
                fsi
            fpara
        fpara
    fpara
    devuelve minT → 1 return
fprocedimiento
```

$$T(n) = 9 + T(bucle1)$$

$$T(bucle1) = \sum_{i=0}^{n} \big(6 + T(bucle2)\big) = (n+1) * \big(6 + T(bucle2)\big)$$

$$T(bucle2) = \sum_{j=i+1}^{n-1} \big(6 + T(bucle3)\big) = (n-1) * \big(6 + T(bucle3)\big)$$

Aquí nuestro problema se ramifica en un mejor caso si encontráramos el triángulo a la primera.

$$T(bucle3) = \sum_{k=j+1}^{n-2} (T(crearTriangulo) + 4) = (n-3) * (T(crearTriangulo) + 4)$$

o un peor caso si todos nuestros triángulos tuvieran igual perímetro y diferente área.

$$T(bucle3) = \sum_{k=j+1}^{n-2} (T(crearTriangulo) + 7) = (n-3) * (T(crearTriangulo) + 7)$$

crearTriangulo es el constructor de la clase Triangulo, se encarga de inicializar sus atributos y calcular su área y perímetro. El coste en operaciones elementales de dicho constructor se corresponde con: 1 por crear el objeto + 3 asignaciones + 29 operaciones para calcular el perímetro + 11 para calcular el área (usamos el perímetro por eso cuesta tan pocas operaciones este último cálculo). En resumen:

$$T(crearTriangulo) = 44$$

Calculando hacia atrás obtenemos el coste de T(n), que para el caso peor sería:

$T(bucle3) = (n-3) * 51 = 51n - 153$

$T(bucle2) = (n-1) * \big(6 + (51n - 153)\big) = (n-1) * (51n - 147) = 51n^2 - 198n + 147$

$T(bucle1) = (n+1) * \big(6 + (51n^2 - 198n + 147)\big) = (n+1) * (51n^2 - 198n + 153)$
$= 51n^3 - 147n^2 - 45n + 153$

$$T(n) = 51n^3 - 147n^2 - 45n + 162 \in \mathbf{O}(\boldsymbol{n^3})$$

y para el caso mejor sería:

$T(bucle3) = (n-3) * 48 = 48n - 144$

$T(bucle2) = (n-1) * \big(6 + (48n - 144)\big) = (n-1) * (44n - 138) = 44n^2 - 182n + 138$

$T(bucle1) = (n+1) * \big(6 + (44n^2 - 182n + 138)\big) = (n+1) * (44n^2 - 182n + 144)$
$= 44n^3 - 138n^2 - 38n + 144$

$$T(n) = 44n^3 - 138n^2 - 38n + 153 \in \mathbf{O}(\boldsymbol{n^3})$$

como el resultado es prácticamente igual no vamos a estudiar el caso medio.

Análisis del algoritmo implementado mediante la estrategia Divide y Vencerás.

```
procedimiento Triangulo DyV(Punto T[], int izq, int der)
    si (der – izq + 1 <= 6) entonces
        devolver Exhaustivo(T, izq, der);
    fsi

    //División del problema
    var: mitad <= (der + izq)/2
    Triangulo ti = DyV(T, izq, mitad)
    Triangulo td = DyV(T, mitad+1, der)

    // Combinación y resolución
    si (ti.GetPerimetro() > td.GetPerimetro() OR ti.GetPerimetro() == td.GetPerimetro() AND
ti.GetArea > td.GetArea()) entonces
        Triangulo tmin <= ti
    sino
        Triangulo tmin <= td
    fsi
    para a <= mitad + 1 hasta der hacer
        si (T[a].GetX – T[mitad + 1] > tmin.GetPerimetro()) entonces
            break
        fsi
    fpara
    para b <= izq hasta mitad hacer
        si (T[mitad].GetX – T[b].GetX > tmin.GetPerimetro()) entonces
            break
        fsi
    fpara
    para c <= b+1 hasta mitad hacer
        para d <= mitad+1 hasta a-1 hacer
            para e <= d+1 hasta a-1 hacer
                Triangulo taux = new Triangulo(T[c], T[d], T[e])
                si (taux.GetPerimetro() < tmin.GetPerimetro()) entonces
                    tmin <= taux
                sino si (taux.GetPerimetro() == tmin.GetPerimetro() AND taux.GetArea() >
                tmin.GetArea()) entonces
                    tmin <= taux
                fsi
            fpara
        fpara
    fpara
```

```
    para c <= b+1 hasta mitad hacer
        para d <= mitad+1 hasta a-1 hacer
            para e <= d+1 hasta a-1 hacer
                Triangulo taux = new Triangulo(T[c], T[d], T[e])
                si (taux.GetPerimetro() < tmin.GetPerimetro()) entonces
                    tmin <= taux
                sino si (taux.GetPerimetro() == tmin.GetPerimetro() AND taux.GetArea() >
                tmin.GetArea()) entonces
                    tmin <= taux
                fsi
            fpara
        fpara
    fpara
    devolver tmin
fprocedimiento
```

Para realizar el análisis de nuestro algoritmo deberemos tener una serie de condiciones en cuenta. Para empezar, nuestro algoritmo DyV() está implementado para recibir un array ordenado (de menor a mayor según la coordenada x), para ordenar dicho array hemos usado el algoritmo Quicksort que ordenara nuestro array en un tiempo de n*log(n), dicho tiempo deberá ser tenido en cuenta a la hora de calcular el tiempo que tardara DyV() en ejecutarse.

También debemos observar que hacemos uso del algoritmo Exhaustivo, sin embargo, este no se realiza en un tiempo de n^3 como hemos analizado anteriormente, sino que tenemos el número de comparaciones que realizará limitado, al igual que los dos conjuntos de bucles triple que se ven al final del algoritmo que cumplen la misma función que el algoritmo exhaustivo y que también tienen limitado el número de iteraciones que realizan.

Para saber el coste computacional de nuestro algoritmo tendremos que determinar las diferentes partes de las que se compone el mismo. Al final del algoritmo podemos observar que se realizan una serie de bucles en busca de un triángulo en la zona central, si observamos bien nos daremos cuenta que nunca se supera la complejidad de O(n), básicamente se recorre primero una mitad del array y luego la otra para encontrar a que distancia ejecutar un exhaustivo muy simplificado, dicho exhaustivo tomará un coste de O(1) ya que prácticamente no varía con la talla, no ocurre lo mismo para buscar esa zona de en medio que como hemos dicho antes explora ambas mitades del array O(2*n/2) = O(n).

Caso base

Puesto que en el caso base solo se entra si hay 6 puntos o menos, nuestro algoritmo exhaustivo realizará en el peor caso → $\binom{6}{3}$ = 220 combinaciones + 3 operaciones.

Puesto que para cada combinación de puntos tenemos que realizar:

$$c = 220 * \textbf{CreaTriangulo}() + 7 \approx 11200$$

El coste de resolver será el número de llamadas recursivas multiplicado por el tiempo que se tardará en resolver el mismo problema dividido entre dos.

$$T(n) = 2T\left(\frac{n}{2}\right) + nk + c$$

Por lo tanto, obtenemos el siguiente sistema recurrente:

$$T(n)\begin{cases} 11200 & si\ n \leq 6 \\ 2T\left(\frac{n}{2}\right) + nk + c \end{cases}$$

Resolveremos la recurrencia por ecuación característica:

Realizamos el cambio de variable → n = 2^m

$$T(2^m) = 2T(2^{m-1}) + 2^m * k + c * m^0 * 1^m$$

$$T(2^m) - 2T(2^{m-1}) = k * m^0 * 2^m + c * m^0 * 1^m \rightarrow No\ homogenea$$

$$(x-2)^2(x-1) = 0$$

Nuestra ecuación característica quedaría de la siguiente forma:

$$T(2^m) = 2^m * C1 + 2^m * m * C2 + 1^m * C3$$

Deshaciendo el cambio y teniendo en cuenta que $1^m = 1$ obtenemos lo siguiente:

$$T(n) = 2^{\log_2 n} * C1 + 2^{\log_2 n} * \log_2 n * C2 + C3$$

Por propiedades de los logaritmos:

$$T(n) = n^{\log_2 2} * C1 + n^{\log_2 2} * \log_2 n * C2 + C3$$

$$T(n) = n^1 * C1 + n^1 * \log_2 n * C2 + C3$$

Calculamos las constantes:

$$T(6) = 6 * C1 + 6 * \log_2 6 * C2 + C3$$

$$11200 = 6 * C1 + 6 * \log_2 6 * C2 + C3 \rightarrow C1 = \frac{11200 - 15{,}5 * C2 - C3}{6}$$

$$T(12) = 2 * C1 + 12 * \log_2 12 * C2 + C3$$

$$12 * C1 + 12 * \log_2 12 * C2 + C3 = 2 * 11200$$

$$12 * \frac{11200 - 15{,}5 * C2 - C3}{6} + 12 * \log_2 12 * C2 + C3 = 2 * 11200$$

$$6 * (11200 - 15{,}5 * C2 - C3) + 12 * \log_2 12 * C2 + C3 = 22400$$

$$67200 - 93C2 - 6C3 + 42C2 + C3 = 22400$$

$$67200 - 51C2 - 5C3 = 22400$$

$$C2 \approx \frac{44800 - 5C3}{51}$$

$$T(24) = 24 * C1 + 24 * \log_2 24 * C2 + C3$$

$$24 * C1 + 108 * C2 + C3 = 22400 * 2$$

$$24 * \frac{11200 - 15{,}5 * \frac{44800 - 5C3}{51} - C3}{6} + 108 * \frac{44800 - 5C3}{51} + C3 = 44800$$

$$4 * \left(11200 - 15{,}5 * \frac{44800 - 5C3}{51} - C3\right) + 108 * \frac{44800 - 5C3}{51} + C3 = 22400 * 2$$

$$4 * (11200 - 0{,}3 * (44800 - 5C3) - C3) + 2{,}12 * (44800 - 5C3) + C3 = 44800$$

$$4 * (-2240 - 4{,}5C3) + 94976 - 9{,}6C3 = 44800$$

$$\frac{86016 - 44800}{27{,}6} = C3 \approx 1500$$

$$C2 \approx 730$$

$$C1 \approx -270 \rightarrow Constante\ negativa!$$

El resultado será de forma aproximada algo como lo siguiente:

$$T(n) = -270n + 730n \log_2 n + 1500 \in \mathbf{O}(\boldsymbol{n \log_2 n})$$

Teniendo en cuenta que este algoritmo emplea el método Quicksort O(nlog n), este coste computacional deberá ser sumado al coste de nuestro algoritmo, sin embargo, puesto que su complejidad es la misma que la de nuestro algoritmo, realmente no es algo que nos influya.

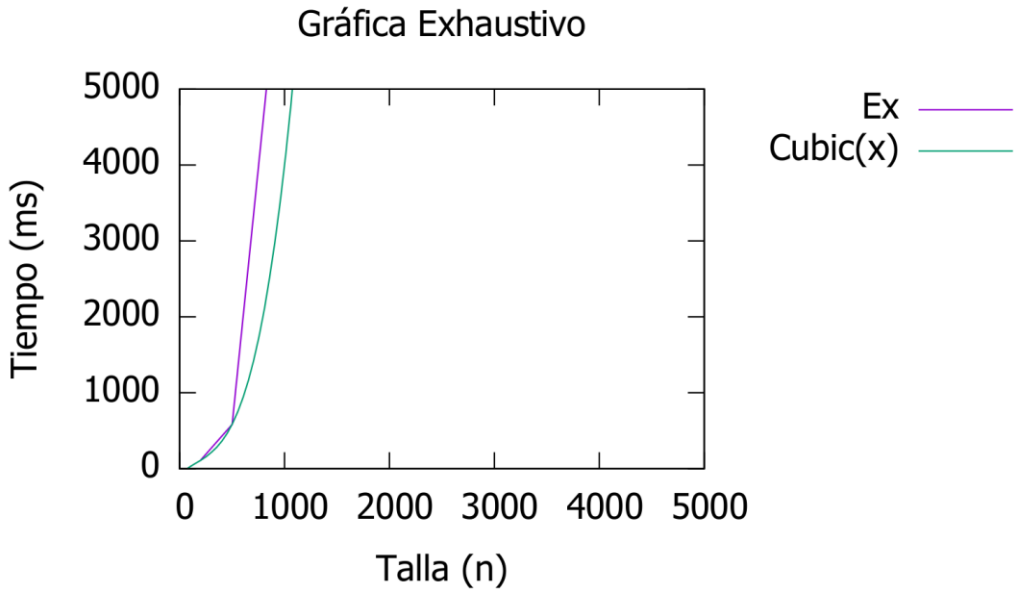
Si nos pusiéramos en el peor caso de nuestro algoritmo, la mayoría de los puntos se encontrarían en la franja de en medio y por lo tanto la fuerza bruta dejaría de ser lineal para pasar a ser 3 bucles anidados de n/2 cada uno en el peor caso.

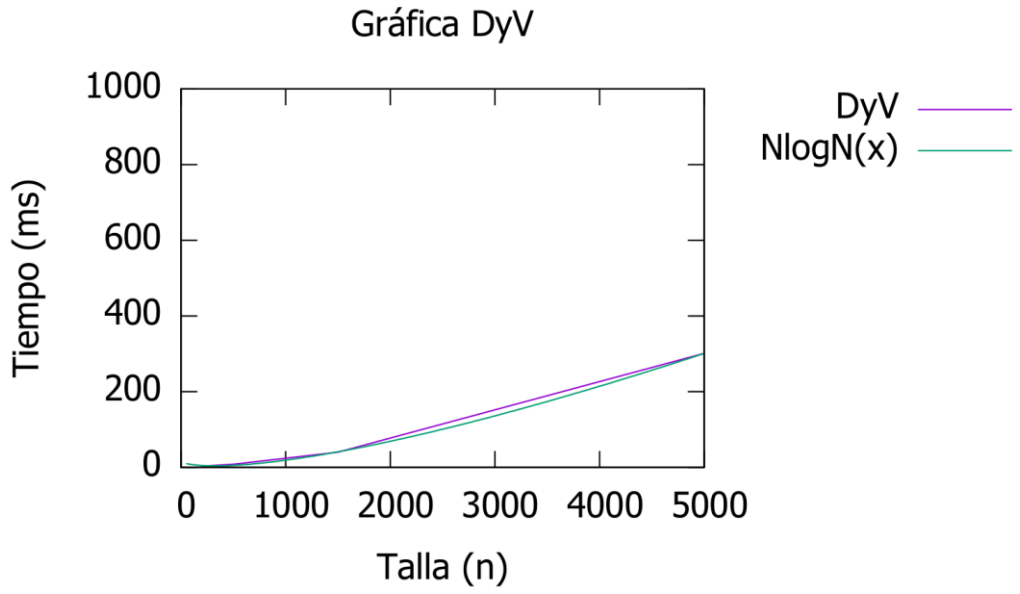
Esto supondría un coste de $\mathbf{O}\left(\left(\frac{n}{2}\right)^3\right)$ o lo que es lo mismo $\mathbf{O}\left(\frac{n^3}{8}\right)$

Comparación de resultados teóricos y experimentales

Para poder realizar una comparación de los resultados teóricos y los resultados experimentales, vamos a fijarnos en el crecimiento de los datos. El motivo de esto es debido a que no podemos comparar un conteo de operaciones elementales con el tiempo que tarda el algoritmo en ser ejecutado, ambos valores están separados por una gran cantidad de variables que produciría resultados muy dispares, ya sea por la velocidad del procesador, la implementación del algoritmo, los recursos de la máquina en la que lo corramos o incluso las condiciones ambientales. Por eso es mucho más visible comparar el crecimiento de la función formada por los datos teóricos y experimentales.

Como podemos ver por los resultados teóricos, el crecimiento de nuestras funciones será $O(n^3)$ para el algoritmo exhaustivo y $O(\text{nlog}\,n)$ para el algoritmo de divide y vencerás. Representaremos los datos experimentales en una gráfica junto a la curva de crecimiento que deberían tener a nivel teórico observar que nuestros datos evolucionan de la forma esperada.





Como podemos observar en las gráficas de arriba, nuestros datos teóricos se corresponden de forma bastante similar con los datos prácticos. Podemos observar que el crecimiento del algoritmo exhaustivo (Ex) se corresponde de forma bastante parecida al crecimiento de la función cúbica (Cubic(x)), mientras que el crecimiento del algoritmo DyV es muy parecido, por no decir exacto, al de la función NlogN(x). Para la generación de esta gráfica se han usado una serie de datasets aleatorios de 200, 500, 1500 y 5000 puntos.

Para realizar esta gráfica hemos hecho uso del programa gnuplot, que como su nombre indica tiene licencia GNU. El código del script usado para generarlas se adjunta con la práctica.

Parte 2: Análisis de algoritmos Voraces

Análisis del algoritmo de Kruskal

Vamos a realizar el análisis sobre el pesudo código que tenemos en las diapositivas aportadas en las clases teóricas de esta asignatura.

```
función Kruskal ( G =(N,A) ) : árbol
    Ordenar A según longitudes crecientes;
    n := |N|;
    T := conjunto vacío;
    inicializar n conjuntos, cada uno con un nodo de N;
    /* bucle voraz */
    repetir
        a := (u,v) : arista más corta de A aún sin considerar;
        Conjunto U := Buscar (u);
        Conjunto V := Buscar (v);
        si Conjunto U <> Conjunto V entonces
            Fusionar (Conjunto U, Conjunto V);
            T := T U {a}
        fsi
    hasta |T| = n-1;
    devolver T
ffunción
```

Suponiendo que tenemos n vértices y m aristas en nuestro grafo, tendríamos lo siguiente:

El coste de ordenar nuestras aristas de menor a mayor, teniendo en cuenta que hacemos otra vez uso de un algoritmo de ordenación rápida (Quicksort en nuestro caso), el coste de ordenar m aristas será: $O(m \log m)$.

Inicializar n conjuntos nos cuesta: $O(n)$.

Puesto que el bucle se repite para $n - 1$ aristas, la complejidad de nuestro bucle será $(n - 1) * X$, donde X será lo complejidad del cuerpo de nuestro bucle.

La función buscar en un conjunto disjunto de vértices tiene un coste de $\log n$ en el peor caso, siempre que los conjuntos se hayan implementado de la forma correcta (usando padres de nodos como representantes de un conjunto)

Para finalizar el análisis de las diferentes funciones que componen este algoritmo nos queda hablar de la función fusionar, cuyo coste computacional sería también $\log n$, una vez más, siempre y cuando los conjuntos disjuntos de vértices estén implementados de la forma correcta.

Todo esto establecería el coste de nuestro algoritmo en $O(m \log m + (n - 1) \log n)$, dejando como mayor factor de peso $m \log m$. Puesto que para representar la complejidad

de nuestro algoritmo nos vendría bien tener ambas variables en la complejidad, haremos un pequeño cambio:

Puesto que sabemos que m va a alcanzar n^2 en el peor de los casos podemos decir que la complejidad de $m \log m$ va a ser la misma que la de $m \log n$.

$$m \leq n^2 \rightarrow \log m \leq \log n^2$$

Por lo tanto:

$$O(m \log m) = O(2m \log n)$$

$$\boldsymbol{T(n) \epsilon O(m \log n)}$$

Análisis del algoritmo de Prim

A continuación, realizaremos el análisis del pesudo código (Algoritmo de Prim) que tenemos en las diapositivas aportadas en las clases teóricas de esta asignatura como el anterior mencionado (Algoritmo de Kruskal).

```
función Prim v.2 ( L[1..n,1..n] ) : árbol
  DistanciaMínima[1] := -1;
  T := conjunto vacio;
  para i := 2 hasta n hacer
      MásPróximo [i] := 1;
      DistanciaMínima [i] := L[i,1]
  fpara
  /* bucle voraz */
  repetir n-1 veces:
      min := infinito;
      para j := 2 hasta n hacer
          si 0 <= DistanciaMínima [j] < min entonces
             min := DistanciaMínima[j];
             k := j
          fsi
      fpara
      T := T U {(MásPróximo[k], k)};/*añadir arista a T
      DistanciaMínima [k] := -1;  /* añadir k a B */
      para j := 2 hasta n hacer
          si L[j,k] < DistanciaMínima[j] entonces
             DistanciaMínima [j] := L[j,k];
             MásPróximo[j] := k
          fsi
      fpara
  finrepetir
  devolver T
finfunción
```

Suponiendo que tendríamos lo mismo que en Kruskal, n vértices y m aristas en nuestro grafo, tendríamos lo siguiente:

En la parte de inicialización:

Creamos un array de tantas posiciones como vértices hay en el grafo y una vez creado tendremos que iniciar el primer valor a -1 debido a que la distancia que hay del primer nodo a si mismo seria 0 y eso crearía ciclos, y los demás valores del array lo inicializaríamos a la distancia mínima.

Eso crearía un coste de: $O(n)$

En la parte del bucle voraz:

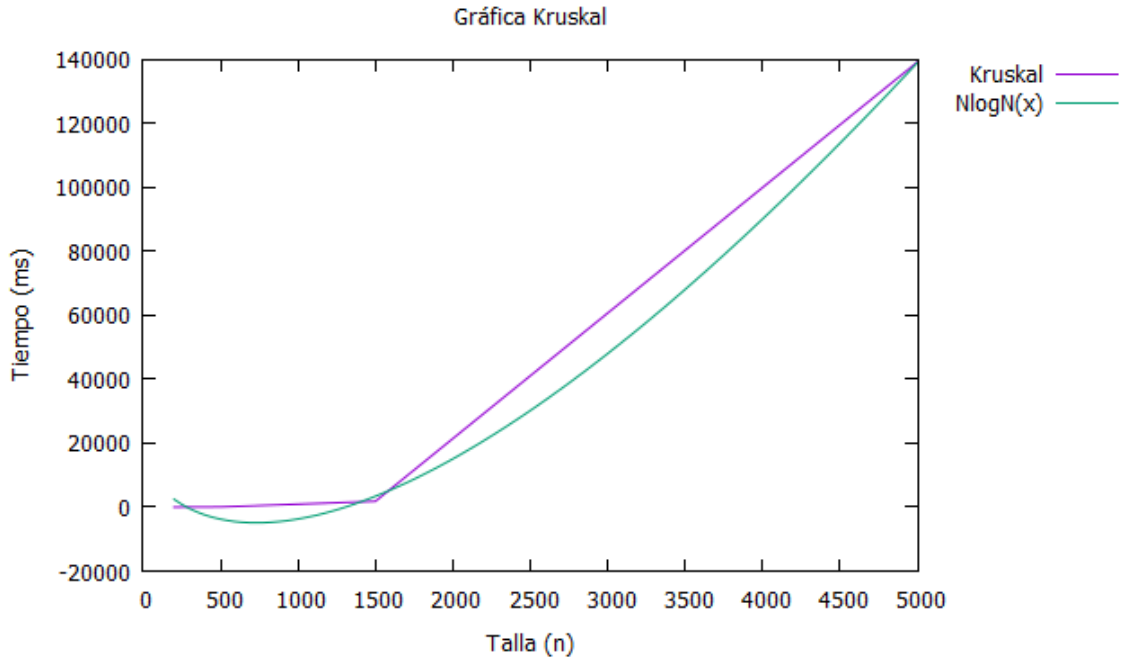
Recorremos la matriz creada para sacar la distancia mínima comparándola con una variable "min" que inicializaremos a infinito debido que no podemos saber con exactitud cuál será la mayor distancia mínima. Una vez recorrido el array almacenaremos en "min" la distancia mínima encontrada y en "k" guardaremos el índice de donde hemos obtenido esa distancia mínima, es decir, la arista seleccionada.

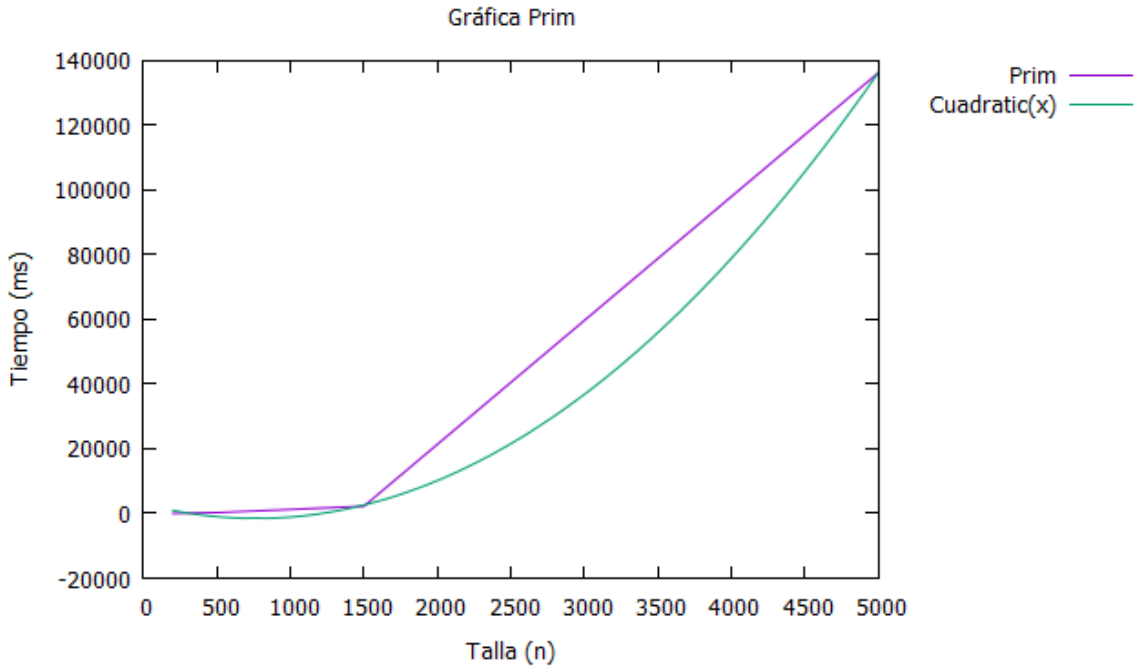
Lo que haremos después de obtener "k" y "min" será añadir la arista encontrada "k" al conjunto T y pondremos el valor de la distancia de "k" a -1 para no generar ciclos más adelante, después realizaremos un bucle que se repetirá n veces para hallar el vértice más próximo. Para finalizar se devuelve el conjunto T.

Todo esto lo repetiremos en un bucle $(n-1)$ veces, lo que generara un coste de: $O(n^2)$

Comparación de los resultados teóricos y experimentales

Hemos seguido el mismo ejemplo del caso anterior y hemos realizado las gráficas mediante la aplicación GNUplot para poder superponer la función matemática obtenida en el análisis teórico con los resultados experimentales.





A pesar de que los resultados de ambas gráficas puedan asimilarse, notaremos que el resultado de ajustar la curva logarítmica a los datos de Kruskal distorsiona los resultados, puesto que la implementación de nuestro Kruskal es bastante ineficiente debido a que no hacemos uso de los conjuntos disjuntos para detectar ciclos en el grafo, esto complica la eficiencia y es algo que, aunque no se vea reflejado en los resultados gráficos se debe tener en cuenta. Por parte de los resultados de Prim, todo parece coincidir de forma bastante similar y podemos concluir que no hay ninguna distorsión en los datos, puesto que se obtiene lo esperado al recorrer una matriz de dos dimensiones, un ajuste muy aproximado a una función cuadrática.

Solución a los problemas propuestos

En la práctica se nos plantea dar a conocer la solución ofrecida por ambos algoritmos voraces a los ficheros **"berlin52.tsp"**, **"ch130"** y **"ch150"**. Esta solución se obtiene de forma gráfica, si ejecutamos ambos algoritmos para los ficheros solicitados en nuestra aplicación, dicha solución se verá reflejada en forma de texto en un fichero a parte titulado **"<algoritmo>_<fichero>.dat"** (Ejemplo: Kruskal_berlin52.dat).

La solución a los ficheros solicitados por la práctica es la siguiente:

- **berlin52** → 6081.63
- **ch130** → 5165.39
- **ch150** → 5880.47
- **d493** → 29277.52
- **d657** → 42484.47

Bibliografía

Time complexity of the Kruskal's Algorithm

Kruskal's Minimum Spanning Tree Algorithm | Greedy Algo-2

Disjoint Set (Or Union-Find) | Set 1 (Detect Cycle in an Undirected Graph)

Prim's Algorithm Time Complexity