PRÁCTICA 1
AMC

Grupo: AMC-L2

Ismael Da Palma Fernández

ÍNDICE

Estudio teórico:

Estudio empírico:

Estudio teórico

Para el análisis de la complejidad temporal en los algoritmos de búsqueda de la ciudad más cercana a otras dos entre un conjunto de ciudades se tomará la comparación de los tríos de ciudades como instrucción crítica en lugar de contar operaciones elementales.

Para el algoritmo de Dijkstra se realizará una estimación del conteo de operaciones elementales.

Algoritmo de Búsqueda exhaustiva:

```
Procedimiento exhaustivo (P[i ... f] : Punto, i : entero, f: entero) : Trio
        Trio sol = Trio(P[i], P[i+1], P[i+2])
        Para a1 = i Hasta f Hacer
                Para a2 = a1+1 Hasta f-2 Hacer
                        Para a3 = a2+1 Hasta f-1 Hacer
                                aux = Trio(P[a1], P[a2], P[a3])
                                Si aux < sol Hacer
                                        sol = aux
                                fSi
                        fPara
                fPara
        fPara
        Devolver sol
fProcedimiento
```

$\boldsymbol{n} = f - i + 1$

$$T(n) = \sum_{a1=1}^{n-2} \sum_{a2=a1+1}^{n-1} \sum_{a3=a2+1}^{n} 1 = \sum_{a1=1}^{n-2} \sum_{a2=a1+1}^{n-1} (n - a2) = \sum_{a1=1}^{n-2} \frac{(1-n)(a1-n+1)}{2}$$
$$= \frac{n(n^2 - 3n + 2)}{6} \in \boldsymbol{O(n^3)}$$

En el algoritmo exhaustivo no importa la distribución y la cantidad de puntos ya que el número de comprobaciones que realiza es siempre el mismo, por lo tanto este algoritmo carece de casos.

Algoritmo Divide y Vencerás (Quicksort):

```
Procedimiento DyV (P[inicio … fin] : Punto, inicio : entero, fin : entero) : Trio
        Si (fin – inicio + 1 < 5) Hacer
                Devolver exhaustivo(P, inicio, fin)  → caso base
        fSi

        q = (inicio + fin) / 2 : entero
        izq = DyV(P, inicio, q) : Trio
        der = DyV(P, q+1, fin) : Trio

        Si (izq.getDMin() < der.getDMin()) Hacer
                minimo = izq
                dmin = izq.getDMin()
        Sino
                minimo = der
                dmin = der.getDMin()
        fSi
        Para a1 = q Hasta inicio Hacer
                Si (P[q+1].getX() – P[a1].getX() > dmin) Hacer
                        Salir del bucle
                fSi
        fPara
        Para a2 = q+1 Hasta fin Hacer
                Si (P[a2].getX() – P[q].getX() > dmin) Hacer
                        Salir del bucle
                fSi
        fPara
        Para a3 = a1+1 Hasta q Hacer
                Para a4 = q+1 Hasta a2 Hacer
                        Para a5 = a4+1 Hasta a2 Hacer
                                aux = Trio(P[a3], P[a4], P[a5])
                                Si (aux.getDMin() < minimo.getDMin()) Hacer
                                        minimo = aux
                                fSi
                        fPara
                fPara
        fPara
        Para a3 = a1+1 Hasta q Hacer
                Para a4 = a3+1 Hasta q Hacer
                        Para a5 = q+1 Hasta a2 Hacer
                                aux = Trio(P[a3], P[a4], P[a5])
                                Si (aux.getDMin() < minimo.getDMin()) Hacer
                                        minimo = aux
                                fSi
                        fPara
                fPara
        fPara
        Devolver minimo
fProcedimiento
```

Teniendo en cuenta solo la instrucción crítica y suponiendo que la búsqueda en la zona de la mitad tiene un coste lineal. Esta búsqueda depende más de la organización de los puntos que de la cantidad.

Sin tener en cuenta el tiempo de ordenación, sólo la búsqueda:

$\boldsymbol{n} = f - i + 1$

$$\boldsymbol{T(n)} = \begin{cases} 1 & \boldsymbol{si}\ n < 5 \\ 2T\left(\frac{n}{2}\right) + nc_1 + c_2 & \boldsymbol{en\ otro\ caso} \end{cases}$$

<u>Resolviendo por ecuación característica:</u>

$T(n) - 2T\left(\frac{n}{2}\right) = nc_1 + 1 \quad --\rightarrow \boldsymbol{Cambio\ de\ variable}: \boldsymbol{n = 2^k} --\rightarrow\ T(2^k) - 2T(2^{k-1}) = 2^k c_1 +\ 1$

$T_k - 2T_{k-1} =\ 2^k c_1 +\ 1$

$\boldsymbol{2^k c_1\ es\ de\ la\ forma\ 2^k + p(k)\ donde\ p(k)\ es\ un\ polinomio\ de\ grado\ 0, por\ lo\ que}:$

$(x-2)(x-2) = 0\ --\rightarrow\ \boldsymbol{Obtenemos\ como\ solucion}: \boldsymbol{2}\ (\boldsymbol{doble}) --\rightarrow\ T_k =\ 2^k c_1 +\ 2^k k c_2$

$\boldsymbol{Deshacemos\ cambio}: \boldsymbol{k} = \mathbf{log}\,\boldsymbol{n}\ --\rightarrow\ T(n) =\ 2^{logn} c_1 +\ 2^{logn} lognc_2 --\rightarrow\ T(n) = nc_1\ + nlognc_2 \in \boldsymbol{O(nlogn)\ si\ c_2 \neq 0}$

El coste de ordenar también es de nlogn ya que utilizamos el algoritmo Quicksort, el coste total del algoritmo DyV sigue siendo nlogn.

El **mejor caso** se dará cuando los puntos se distribuyan de tal forma que al dividir las mitades nunca haya más de dos puntos en el centro, obteniendo cero comprobaciones en la zona de la mitad. **O(nlogn).**

El **peor caso** será cuando todos los puntos estén distribuidos por la zona de la mitad. En ese caso la búsqueda en esa zona se acercaría al coste del algoritmo exhaustivo. **Aprox. O(n^3)**.

Algoritmo voraz de Dijkstra:

Sabiendo que **n** es el número total de puntos

```
Procedimiento algoritmoDijkstra (matrizAd[1 ... n][1 ... n] : vector enteros) : vector[1 ... n] enteros
        solucion = entero [1 ... n]
        seleccionados = booleano [1 ... n]
        posPmin = 0

        Para i=1 Hasta n Hacer  → Inicialización
                seleccionados[i] = false
                solucion[i] = matrizAd[0][i]
        fPara
        solución[0] = 0

        Para i = 0 Hasta n Hacer  → Bucle voraz
                min = +∞
                Para j = 0 Hasta n Hacer
                        Si (!seleccionados[j] Y solucion[j] < min) Hacer
                                min = solucion[j]
                                posPmin = j
                        fSi
                fPara

                seleccionados[posPmin] = true

                Para j = 0 Hasta n Hacer

                        Si (!seleccionados[j] Y solucion[j] > solución[posPmin]+matrizAd[posPmin][j])
                        Hacer
                                solucion[j] = solucion[posPmin] + matrizAd[posPmin][j]
                        fSi
                fPara
        fPara

        Devolver solucion

fProcedimiento
```

En el algoritmo Dijkstra resulta complicado obtener un caso mejor y peor debido a la aleatoriedad de los puntos. Pero podemos saber que su orden es cuadrático debido al tiempo de la inicialización (n) y el tiempo del bucle voraz.

$O(n*n) = \mathbf{O(n^2)}$

<u>Resultados obtenidos de los ficheros</u>:

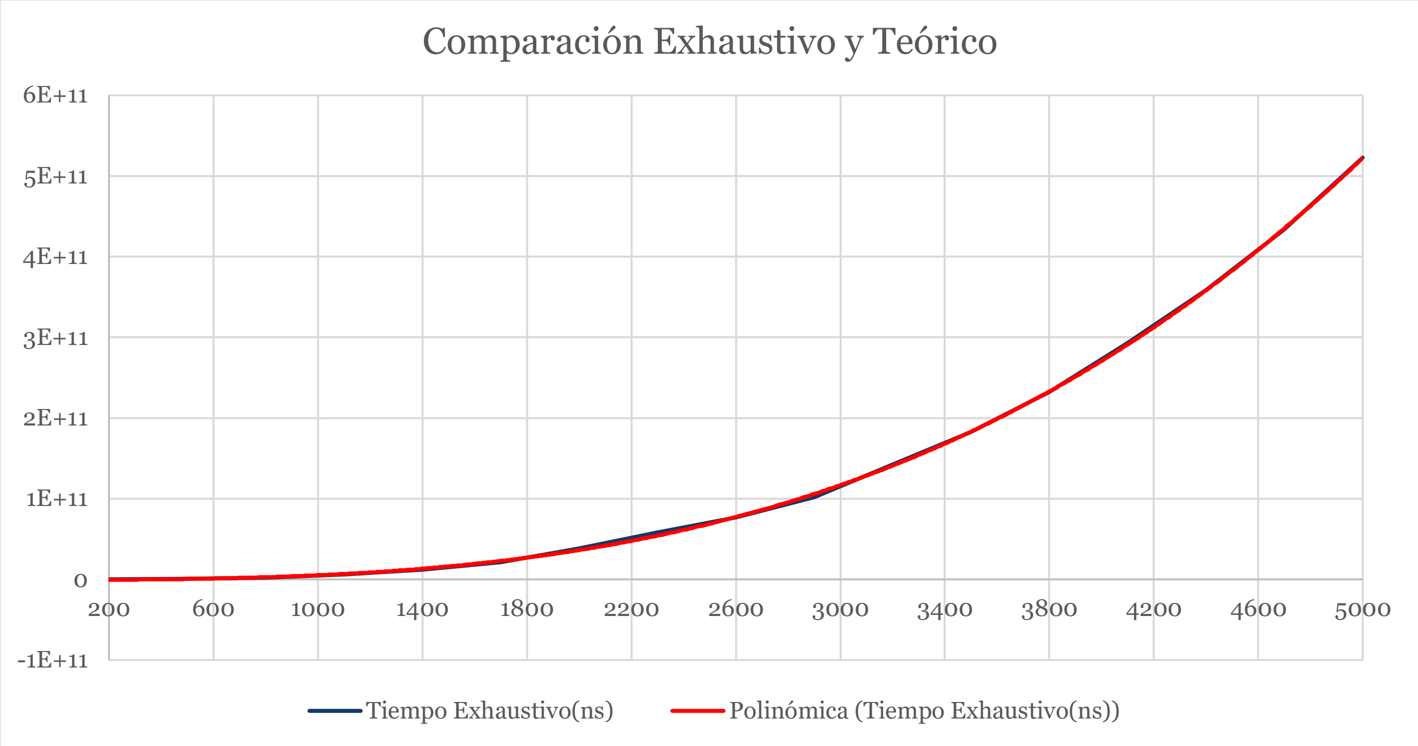
- **berlin52:** Tamaño 52 ciudades. Coste de la solución óptima: **162**
- **ch130:** Tamaño 130 ciudades. Coste de la solución óptima: **788**
- **ch150:** Tamaño 150 ciudades. Coste de la solución óptima: **807**

Estudio empírico

Algoritmo de Búsqueda exhaustiva:

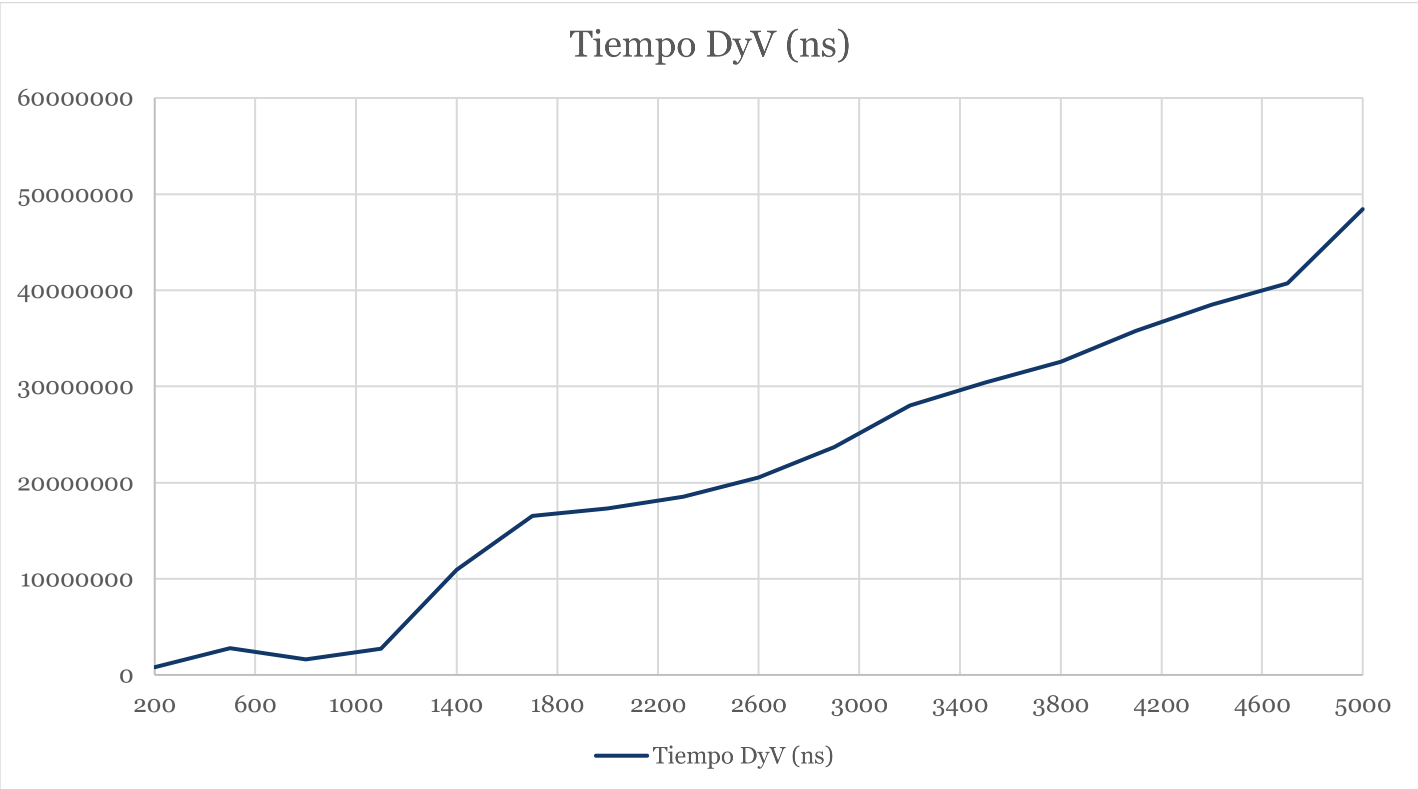
Algoritmo exhaustivo ejecutado en tallas de 200 a 5000 con incremento de 300.

En tallas bajas, el algoritmo tarda poco tiempo. Pero cuando la talla aumenta, el tiempo que tarda es muchísimo mayor, como es de esperar de una complejidad O(n^3). Con 5000 elementos ha llegado a tardar casi 7 minutos. Aunque esto también depende del lo sobrecargado que esté el ordenador de trabajo.



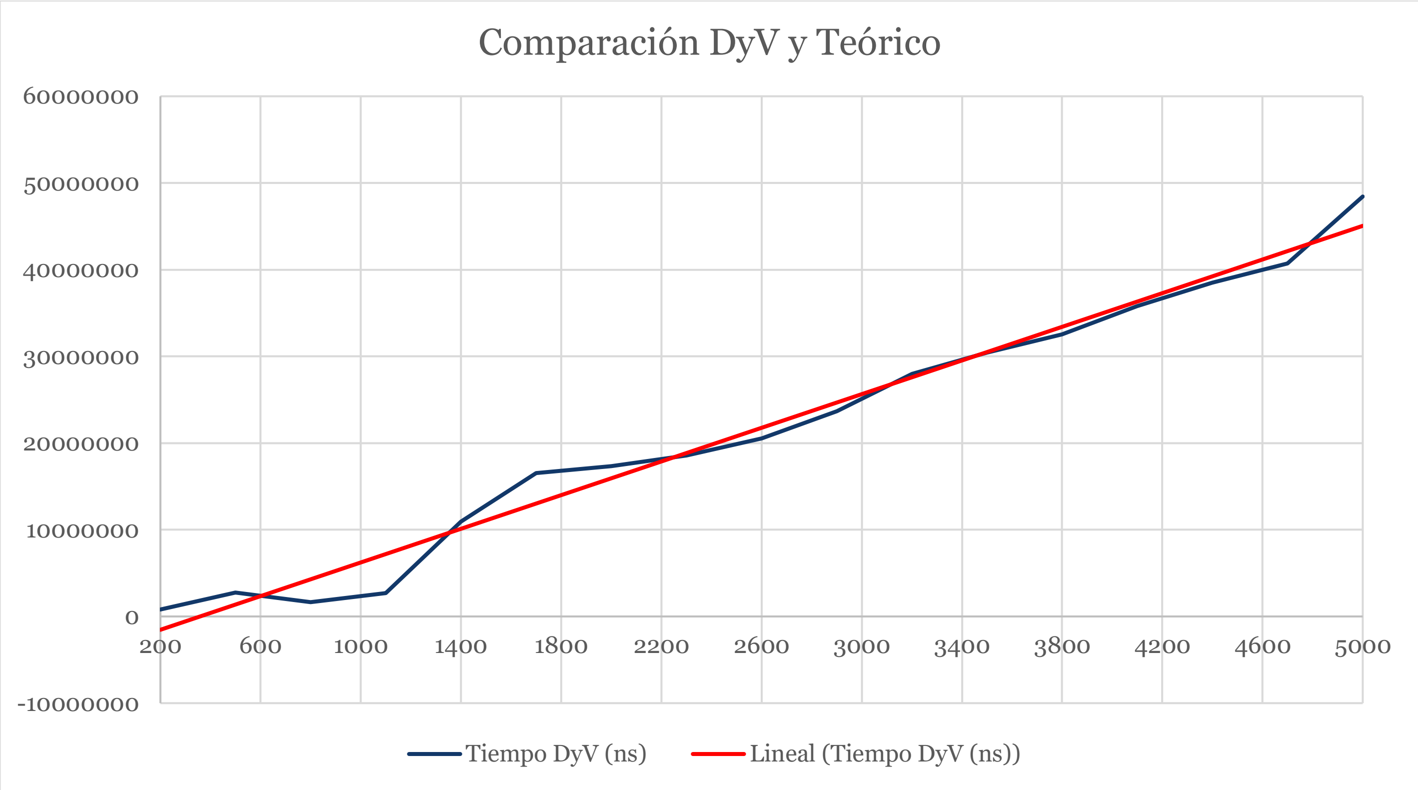
Añadiéndole una línea de tendencia polinómica de n^3 podemos ver que el ajuste es perfecto con respecto a lo obtenido empíricamente.

Algoritmo Divide y Vencerás (Quicksort):



Algoritmo Divide y Vencerás con Quicksort ejecutado en tallas de 200 a 5000 con incremento de 300.

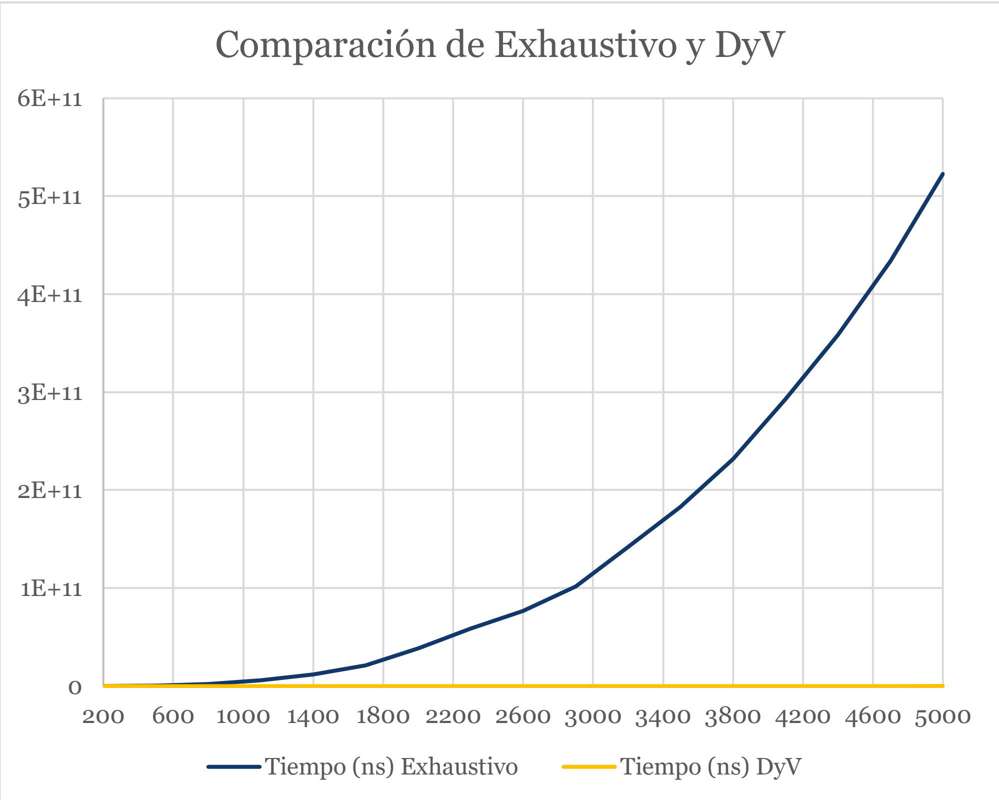
Podemos observar que este algoritmo no supera las 5 milésimas de segundo en su talla más alta.



Aunque parezca lineal, la complejidad del DyV es de O(nLogn) por el estudio teórico y se demuestra con el ajuste que se realiza con una función lineal (ya que se asemeja a la nlogn).

No se ajusta completamente a lo esperado teóricamente debido a la aleatoriedad de los puntos de cada talla, ya que dependerá de cómo se agrupen estos puntos.

Comparación de Búsqueda exhaustiva y DyV:

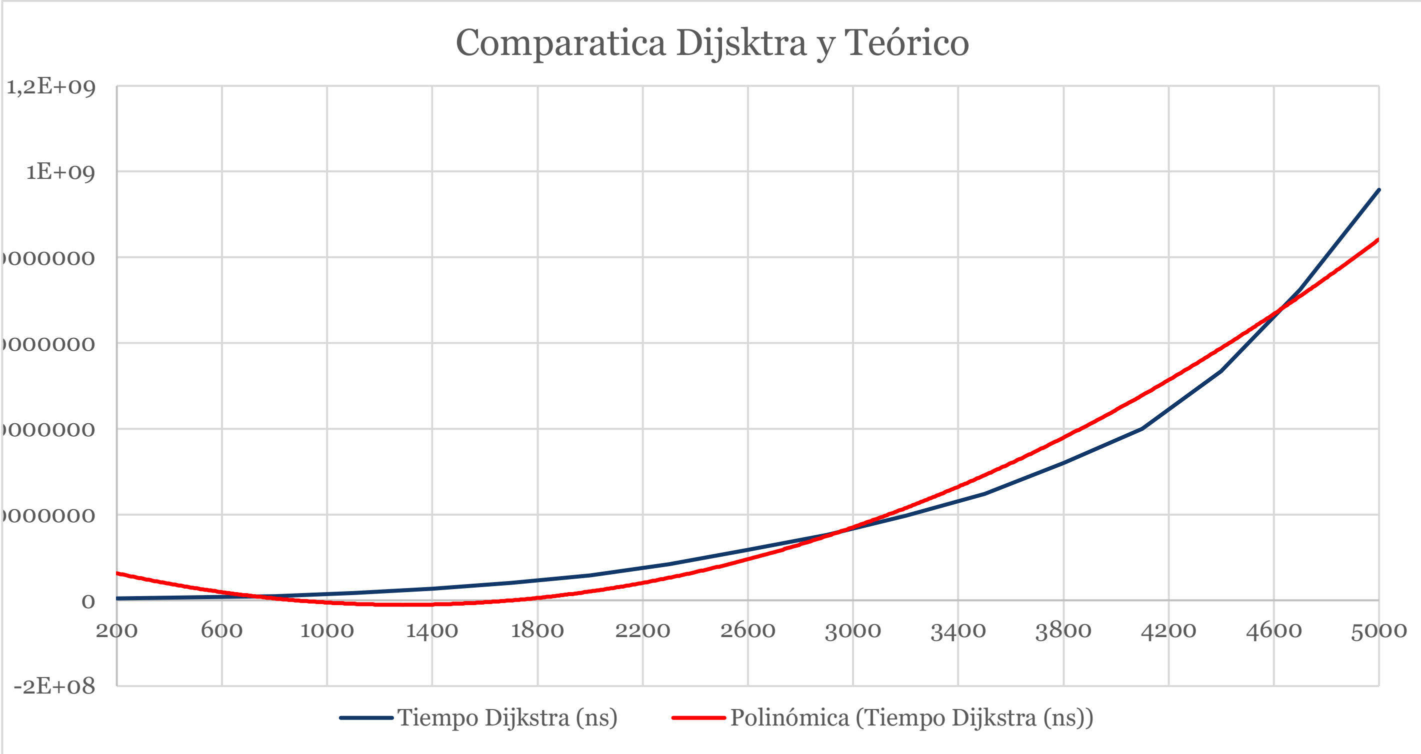


DyV parece prácticamente constante en comparación con el Exhaustivo. La diferencia entre ambos es muy grande.

Debido a que el Exhaustivo crece muy rápido resulta difícil representar estos dos algoritmos.

Una diferencia entre ambos es que el DyV tarda más o menos tiempo en función de como se agrupen los puntos, mientras que el Exhaustivo realiza siempre el mismo número de operaciones para cada una de las tallas.

Comparación Dijkstra del estudio teórico y empírico:



Algoritmo Dijkstra ejecutado con tallas de 200 hasta 5000 con incremento de 300. Cada talla se ha ejecutado 10 veces y se ha realizado su media para poder representarlo.

Podemos ver que el algoritmo Dijkstra se ajusta más o menos a lo esperado con el orden temporal teórico O(n2). Tiene algunas desviaciones debido a que tardará más o menos tiempo en obtener el vector solución según el reparto de los puntos.