METODOLOGÍA DE LA PROGRAMACIÓN Y ALGORITMIA

Billetes de avión

Curso 2023/2024

Componentes del grupo:

- Lorena Almoguera Romero
- María Isabel Salinas Pérez

Índice

1.- Algoritmo para calcular precios..... 3
1.1- Descripción del algoritmo.....3
1.2 - Tipificación del problema..... 3
1.3. - Estrategia de programación y pseudocódigo.....4
1.4.- Ejemplo del funcionamiento del algoritmo..... 6
1.5. - Complejidad asintótica.....8
2 - Algoritmo para calcular precios y rutas..... 13
2.1. - Descripción del algoritmo.....13
2.2. - Tipificación del problema..... 13
2.3. - Estrategia de programación y pseudocódigo.....14
2.4. - Ejemplo del funcionamiento del algoritmo..... 15
2.5.- Complejidad asintótica.....18
3. - Bibliografía..... 21

1.- Algoritmo para calcular precios

1.1- Descripción del algoritmo

El algoritmo que se ha empleado para la realización de calcular precios de los vuelos es el algoritmo de Dijkstra. Este es un método eficiente para encontrar el camino más corto (económico) desde un origen hasta todos los demás destinos. Es decir: *Calculamos el coste del camino más económico (corto) desde un vértice inicial / origen a todos los demás vértices*.

Por lo tanto, en el algoritmo empleado para la realización de esta práctica se calcula el coste de los caminos sumando los costes o pesos asociados a las aristas que unen los vértices del camino. Esto se ilustrará más adelante en el ejemplo de funcionamiento del algoritmo.

1.2 - Tipificación del problema

1. Datos necesarios para el funcionamiento del algoritmo

   Para el funcionamiento del algoritmo se necesitará lo siguiente:

   a. Total de ciudades. Este total será extraído de un fichero. *En el código será representado por:* int **n**

   b. Un conjunto de ciudades que se guardará dentro de un arreglo (vector). Este conjunto de ciudades será extraído de un fichero. *En el código será representado por:* int * **vCiudades** (en la función de dijkstra para los índices representativos de cada una de ellas) y **string * vCiudades** (en el main para guardar los nombres de estas. En la función de dijkstra se utilizará para representar si una ciudad ha sido visitada o no.

   c. Una matriz que representará los precios de los vuelos entre cada ciudad. Si no existe un vuelo entre una ciudad y otra se establecerá como 0. Esta matriz será extraída de un fichero. *En el código será representada por:* **int ** matriz**

   d. Un punto de partida (ciudad origen). Este punto de partida será establecido por el usuario. *En el código será representado por:* **int ciudad_origen** (en el main) y **int ciudad_inicial** (en el dijkstra).

   e. Vector int * S, en el cual se irán almacenando los costos de las ciudades. Explicaremos su inicialización a continuación.

2. Estructura de datos y configuración de estos

El algoritmo para calcular precios recibirá los datos mencionados anteriormente como entrada. Además recibirá un arreglo de ‘S’ en el cual se almacenará el costo mínimo para llegar a cada ciudad desde la ciudad destino.

A continuación vamos a explicar paso por paso nuestro algoritmo.

1. Inicialización del vector S y vCiudades

En la parte inicial de nuestro algoritmo inicializamos mediante el uso de dos bucles (en ambos se recorre todo el vector desde la posición inicial 1 hasta la posición n) los vectores S y vCiudades.

Para el vector vCiudades inicializamos todas las posiciones (de i hasta n) a 1, para controlar cuando se visitará una ciudad. Si una ciudad ha sido visitada posteriormente esta será actualizada (en su índice) a 0.

A continuación, se explica la inicialización del bucle S:

A. Si en la posición i encontramos la ciudad inicial: Se inicializará la posición i dentro del vector S_i a 0.
B. Si en la matriz dónde se cruzan ciudad_inicial e i se encuentra una conexión directa (es decir el valor es diferente a 0): Se inicializa S_i al costo de la conexión directa.
C. Finalmente si no existe ninguna conexión de vuelo directa y en la posición i no se encuentra la ciudad inicial entonces inicializamos esa posición a INFINITO, en el caso de nuestro programa será INT_MAX, que lo utilizamos para representar un número muy grande, es decir infinito.

2. Encontrar el camino más corto a cada uno de los destinos

A continuación iteramos n - 1 veces para actualizar y determinar la distancia mínima desde el origen a cada uno de los otros nodos. Dentro de este bucle realizaremos lo siguiente:

A. Seleccionar el vértice no visitado que posee la distancia más corta desde el origen.
B. Si no existen más destinos por visitar retornará -1.
C. Marcaremos el índice como visitado dentro de nuestro vector vCiudades, igualando x a 0.
D. Actualizamos las distancias mínimas, realizando comprobaciones para posibles distancias mínimas nuevas. Si existiesen distancias nuevas, se realizará una comprobación para obtener el mínimo entre la distancia previa y la nueva y actualizar este valor en el vector S.

1.3. - Estrategia de programación y pseudocódigo

El algoritmo de dijkstra pertenece a la técnica de diseño de algoritmos voraces (greedy algorithms).
Esta estrategia se enfoca en la toma de decisiones que se basa en información disponible de modo inmediato y la selección de un elemento más prometedor en cada momento según un criterio de selección. El algoritmo utilizado para la resolución del problema del camino más barato a todos los destinos es voraz ya que:

- Se selecciona la mejor opción local: (nodo con el menor costo conocido desde el nodo origen) en cada iteración. Dicha elección se realiza de manera voraz, ya que en cada paso se elige la opción que parece ser la mejor en ese momento sin considerar el efecto de esta elección en el futuro.

- Actualización de distancias: Se actualizan las distancias de los demás destinos tras la selección del nodo actual, basándonos en la información inmediata disponible, algo característico de los algoritmos voraces.

- No revisión de decisiones anteriores: Dijkstra no revisa ni modifica las decisiones tomadas en pasos anteriores, ya que al marcar un nodo como visitado esta decisión es final, es decir, no se podrá cambiar.

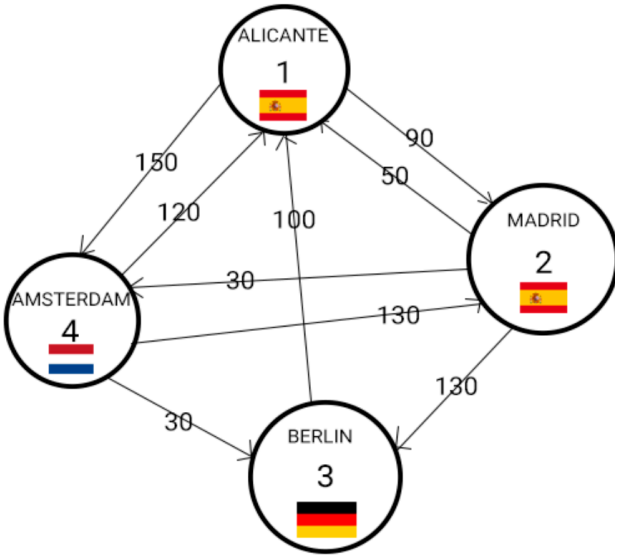
Se concluye que el algoritmo efectivamente pertenece a la técnica de algoritmos voraces (greedy algorithms).

A continuación, presentamos el algoritmo diseñado para la realización del ejercicio:

```
funcion dijkstra(n: real, vCiudades: &entero + U {0} [1 . . . n],
matriz:real+[1 … n, 1 … n],
S:&real[1 … n], ciudad_inicial:real)

      i,j,x:entero+

      para i ← 1 hasta n hacer
            vCiudades_i ← 1
      fpara

      para i ← 1 hasta n hacer
            si i = ciudad_inicial
                  S_i ← 0
            si no matriz_ciudad_inicial , i != 0
                  S_i ← matriz_ciudad_inicial , i
            si no
                  S_i ← inf
            fsi
      fpara

      para i ← 1 hasta n-1 hacer
            x ← elegir_vertice_coste_min(vCiudades, S, n)
            si x = 1
                  romper
            fsi
            para j ← 1 hasta n hacer
                  si vCiudades_j != 0 y matriz_x , j != 0 y S_x != inf
                        S_j ← minimo(S_j , S_x + matriz x , j)
                  fsi
            fpara
      fpara
ffuncion
```

1.4.- Ejemplo del funcionamiento del algoritmo

A continuación, presentamos un ejemplo para visualizar el funcionamiento del algoritmo.



Grafo dirigido que representa las conexiones y distancias entre las ciudades

Visualizamos la tabla de adyacencia construida a partir de este grafo dirigido.

	ALICANTE	MADRID	BERLIN	AMSTERDAM
ALICANTE	0	90	+∞	150
MADRID	50	0	130	30
BERLIN	100	∞	0	∞
AMSTERDAM	120	130	50	0

Las ciudades que formarán parte de nuestro arreglo serán Alicante, Madrid, Berlín, Amsterdam. Observamos que Alicante no tiene vuelos directos a Berlín.

Si seleccionamos Alicante como ciudad origen (ciudad origen = 1) observaremos la siguiente traza:

Paso	**x**	**vCiudades**	**S**			
			1	**2**	**3**	**4**
1	1	{0,2,3,4}	0	90 ↑	0	150
2	2	{0,0,3,4}	0	0	220	120 ↑
3	4	{0,0,3,0}	0	0	170 ↑	0
4	3	{0,0,0,0}	0	0	0	0

Explicación de la traza por pasos:

- **Paso 1:**X toma el valor del nodo de origen. Seleccionamos el nodo con menor coste (Madrid = 90) y se establece este valor a 0.

- **Paso 2:** X toma el valor del nodo recientemente visitado (x= 2 (Madrid)) y se iguala el costo en este índice dentro del vector S a 0. Se actualiza el costo de Berlín (90+130 = 220). Además se actualiza el costo de Amsterdam ya que se puede llegar a dicho nodo con un menor coste (90+30 = 120). Finalmente, seleccionamos el nodo de menor coste que en este caso es Amsterdam.

- **Paso 3:** X toma el valor del nodo recientemente visitado (x=4 (Amsterdam)) y se iguala el costo en este índice dentro del vector S a 0. Se actualiza el Costo de Berlín (120+50 = 170) ya que se puede llegar a Berlín con un menor coste. Finalmente seleccionamos el nodo de menor coste que en este caso es Berlín.

- **Paso 4:** Todos los posibles destinos han sido visitados y por lo tanto hemos terminado.

Por lo tanto los caminos de menor coste desde el origen Alicante a todos los disponibles destinos son los siguientes:

ALICANTE → ALICANTE:	0
ALICANTE → MADRID:	90
ALICANTE → BERLÍN:	170
ALICANTE → AMSTERDAM:	120

1.5. - Complejidad asintótica

A continuación, vamos a resolver la complejidad asintótica del pseudocódigo paso por paso:

En primer lugar identificamos el tamaño del problema. El tamaño del problema en nuestro caso es n, ya que siempre trataremos con una matriz cuadrada es decir de tamaño n x n.

En segundo lugar identificamos los órdenes de cada una de las líneas del pseudocódigo.
Todo es de O(1) menos lo siguiente

- Los bucles i ← 1 o j ← 1 hasta n o n-1 son de O(n)
- La llamada a la función de elegir_vertice_coste_min es de Orden O(n) ya que está función emplea un bucle para hasta n.

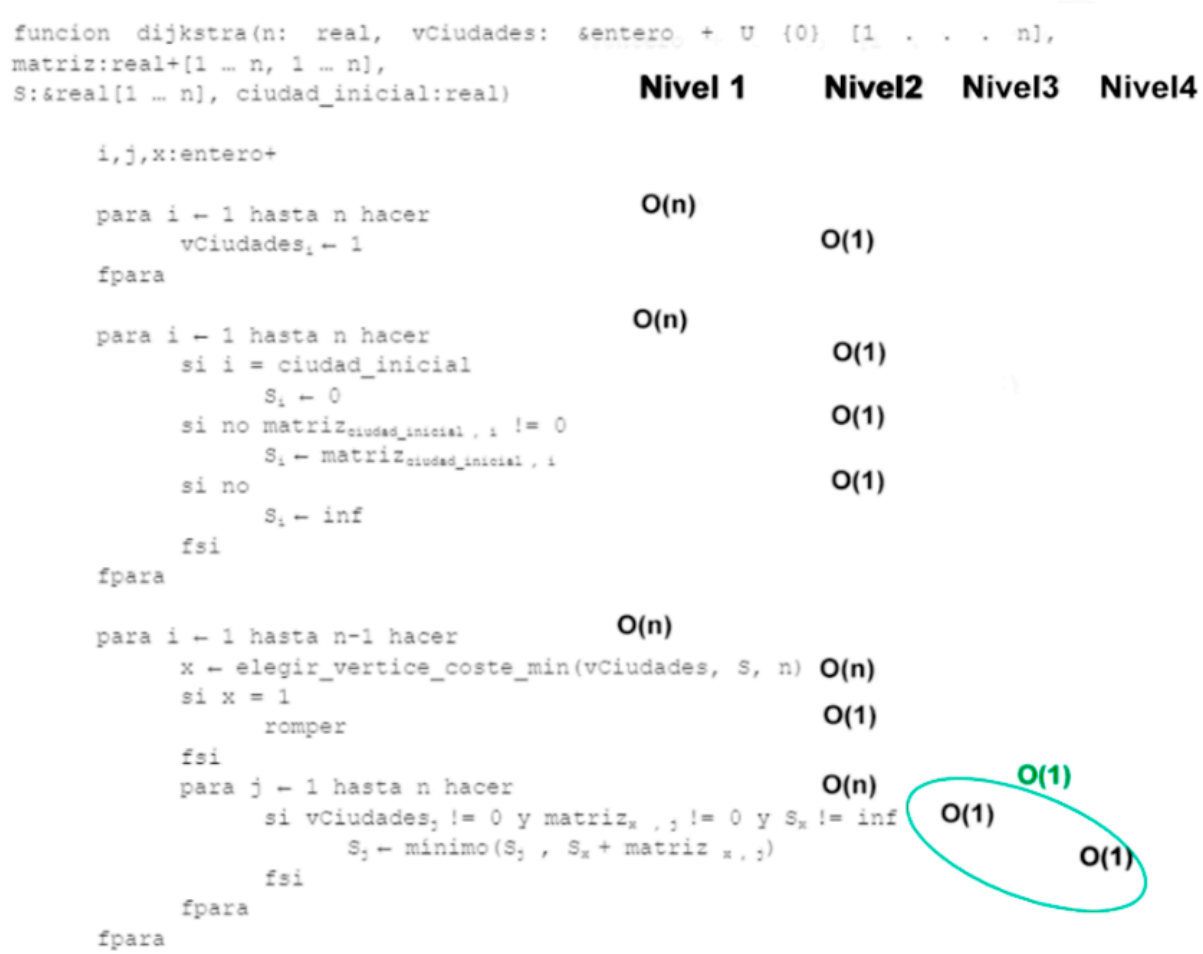


Imagen que muestra la evolución del nivel 4 al nivel 3

Como podemos observar el máximo del nivel 4 es de O(1). A continuación, lo multiplicamos por el nivel superior. O(1) * O(1) nos dará un nuevo orden de O(1). Es decir, la complejidad asintótica es la misma.

Proseguimos con el siguiente nivel en la siguiente imagen.

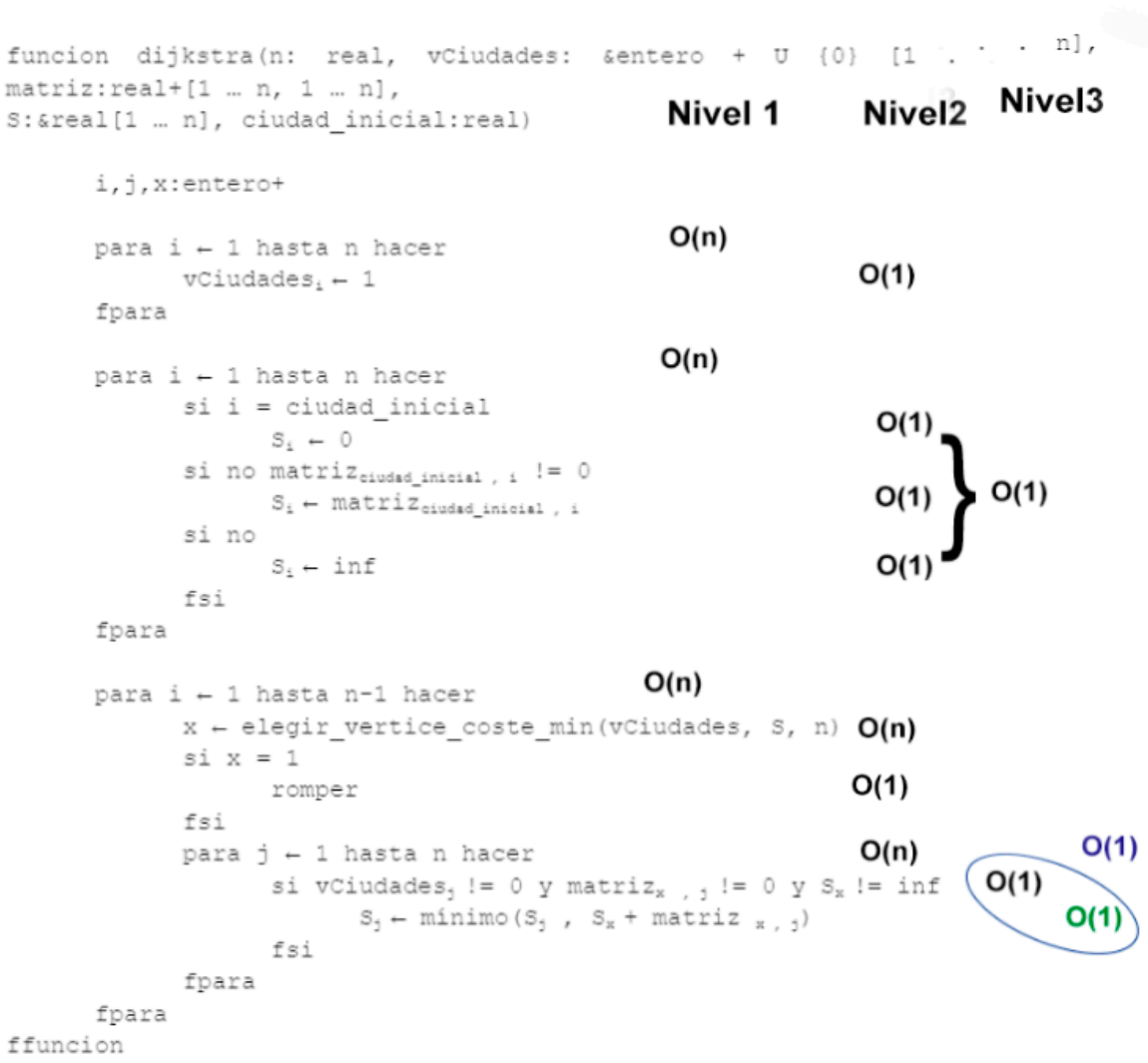


Imagen que muestra la evolución del nivel 2 y 3

Como podemos observar en el bucle para buscamos el nivel máximo de las sentencias Si, ya que todas ellas se encuentran al mismo nivel. Por lo tanto esto se nos queda como O(1) en el nivel 2.

Como hemos comentado en la imagen dónde se muestra la evolución del nivel 4 al 3, la complejidad asintótica en la línea en la cual le asignamos a S_j el coste mínimo dentro de la sentencia Si es de O(1). Esta complejidad la necesitamos para continuar con la evolución del nivel 3 y así poder pasar al nivel 2.

Para ello multiplicamos la complejidad del bucle para j ← 1 hasta n O(n) por la complejidad que acabamos de unir dentro de este bucle que es O(1). Esto nos dará como complejidad resultante O(n). Por lo tanto en el bucle inferior del algoritmo nos encontramos con que tenemos 3 órdenes. O(n), O(1) y O(n).

Observamos la imagen:

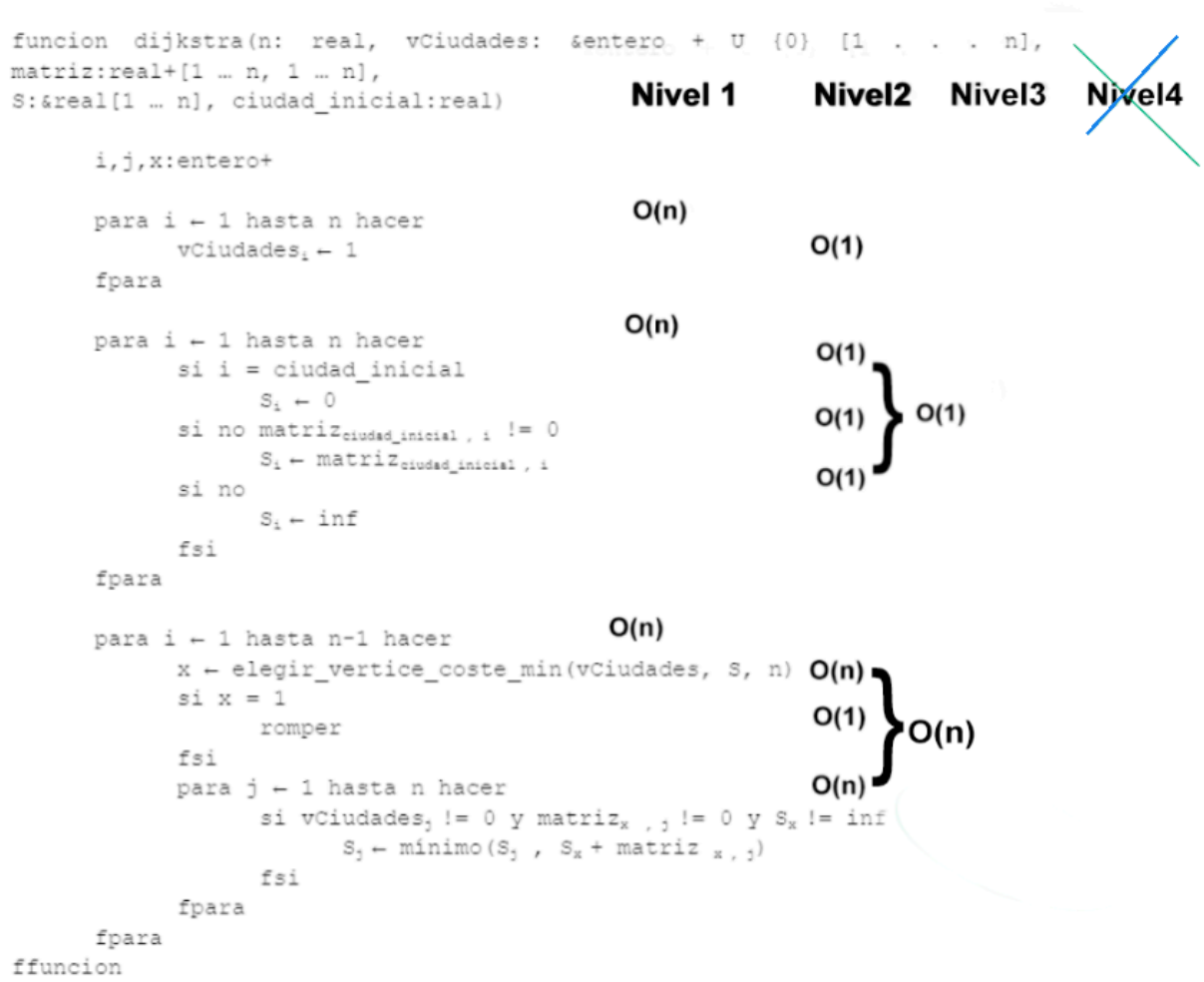


Imagen que muestra la evolución del nivel 2 y 3

Tras haber realizado el producto del bucle para j ← 1 hasta n hacer (orden O(n)) y la sentencia sí que se encuentra dentro de este (orden O(1)) nos encontramos con que dentro del bucle para i ← 1 hasta n-1 hacer tenemos 3 complejidades. O(n), O(1) y O(n). Al encontrarnos en el mismo nivel buscamos la complejidad máxima de estas tres. Por lo tanto, la complejidad del contenido de este bucle es de O(n).

Continuamos operando en el nivel 2.

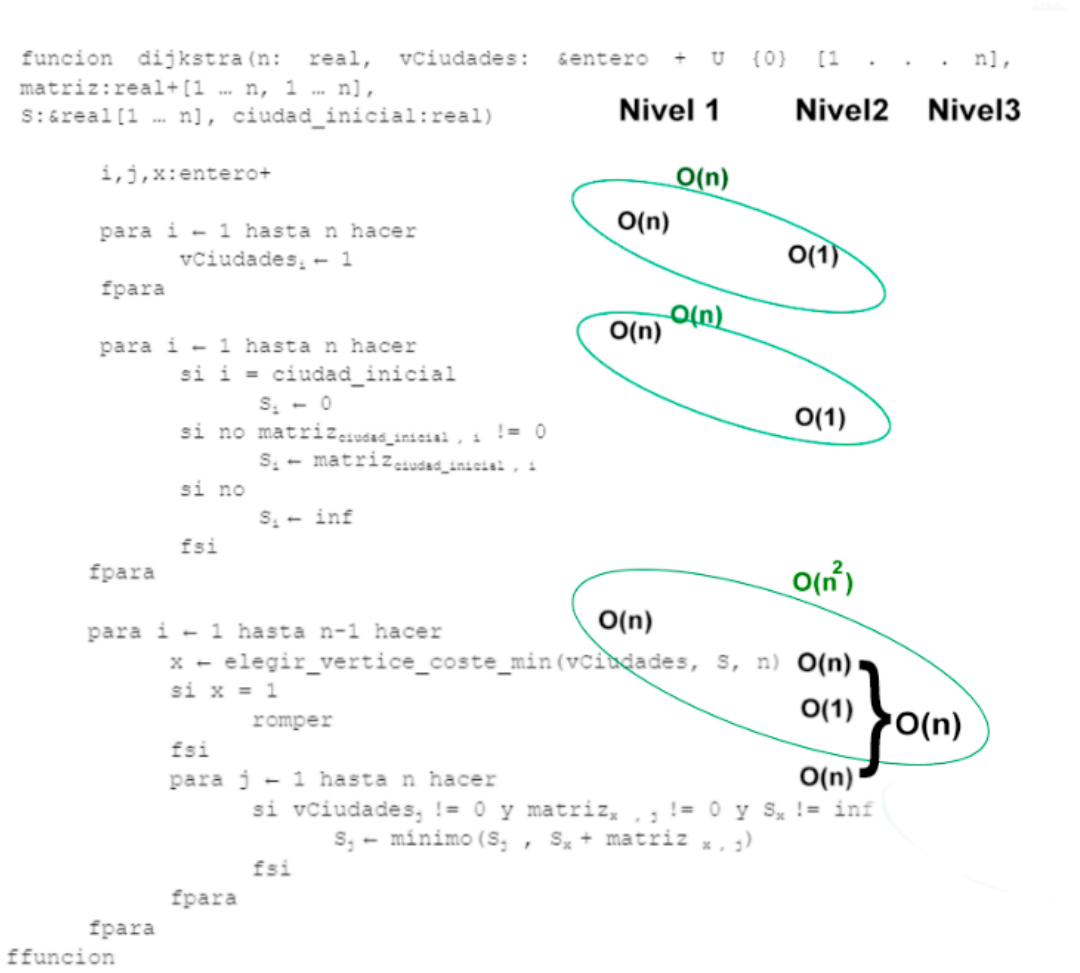


Imagen que muestra la evolución del nivel 2 al nivel 1

Como hemos comentado antes hemos buscado la complejidad máxima entre la llamada a la función de elegir_vertice_coste_min, dónde le asignamos el valor a la variable x, y lo que hemos ido calculando anteriormente en el nivel 2.

Al haber realizado las uniones ya podemos pasar al nivel 1 únicamente nos queda realizar la unión de los niveles 2 y 1 en los bucles. Esto se hace realizando tres multiplicaciones. Comenzamos con el bucle que se encuentra en la parte inferior del pseudocódigo es decir el bucle i ← 1 hasta n-1 hacer.

El bucle es de orden O(n) y lo que tenemos dentro del nivel 2 de este bucle es de orden O(n). Por lo tanto realizamos una multiplicación entre O(n) y O(n). Esto nos da como resultado $O(n^2)$.

Los dos bucles superiores realizan lo mismo. Al tratar con la misma complejidad en los mismos niveles el resultado será el mismo para ambos. En el bucle para i ← hasta n hacer tenemos una complejidad de O(n) mientras que lo que nos encontramos dentro del bucle tiene una complejidad de O(1). Al pasar de un nivel a otro realizamos la multiplicación O(n) * O(1). Esto significa que como resultado tenemos O(n) para ambos bucles superiores.

De esta manera hemos llegado al nivel 1.

```
funcion dijkstra(n: real, vCiudades: &entero + U {0} [1 . . . n],
matriz:real+[1 … n, 1 … n],
S:&real[1 … n], ciudad_inicial:real)

      i,j,x:entero+

      para i ← 1 hasta n hacer
            vCiudades_i ← 1
      fpara

      para i ← 1 hasta n hacer
            si i = ciudad_inicial
                  S_i ← 0
            si no matriz_ciudad_inicial , i != 0
                  S_i ← matriz_ciudad_inicial , i
            si no
                  S_i ← inf
            fsi
      fpara

      para i ← 1 hasta n-1 hacer
            x ← elegir_vertice_coste_min(vCiudades, S, n)
            si x = 1
                  romper
            fsi
            para j ← 1 hasta n hacer
                  si vCiudades_j != 0 y matriz_x , j != 0 y S_x != inf
                        S_j ← mínimo(S_j , S_x + matriz _x , j)
                  fsi
            fpara
      fpara
ffuncion
```

Nivel 1 **Nivel2** **Nivel3**

O(n)

O(n)

O(n²)

complejidad del algoritmo de dijkstra actividad 1 es de **O(n²)**

Imagen que muestra la evolución del nivel 1

Tras haber llegado al nivel 1 únicamente tenemos que mirar cual de los órdenes del nivel 1 es el máximo: O(n) o O(n^2). Evidentemente, el órden máximo es de O(n^2) y por lo tanto la complejidad asintótica del algoritmo de djikstra que se ha diseñado para la realización del ejercicio es de órden O(n^2).

2 - Algoritmo para calcular precios y rutas

2.1. - Descripción del algoritmo

Este algoritmo es una implementación del algoritmo de Dijkstra, que es un método utilizado para encontrar el camino más corto desde un nodo de origen hasta todos los demás nodos en un grafo ponderado dirigido. En el contexto de este algoritmo, el grafo representa una red de rutas aéreas, donde los nodos son ciudades y las aristas son los vuelos entre ellas.

El propósito del algoritmo es calcular las rutas más económicas desde una ciudad de origen hasta todas las demás ciudades en la red de rutas. Para hacer esto, modifica el algoritmo de Dijkstra, que funciona de la siguiente manera:

Inicializa las estructuras de datos necesarias, una para marcar ciudades como visitadas, otra para mantener las distancias mínimas desde la ciudad de origen hasta cada ciudad, y otra para almacenar las rutas óptimas. Establece las distancias iniciales desde la ciudad de origen hasta todas las demás ciudades. Si hay una conexión directa entre dos ciudades, se utiliza el costo de esa conexión como la distancia. De lo contrario, se establece una distancia infinita.

Utiliza el algoritmo de Dijkstra para explorar todas las posibles rutas desde la ciudad de origen hasta todas las demás ciudades. En cada iteración del algoritmo, selecciona la ciudad más cercana que aún no haya sido visitada y actualiza las distancias mínimas a sus ciudades vecinas si se encuentra una ruta más económica. Una vez completado el proceso, la función imprime las rutas más económicas, mostrando el destino, el precio del vuelo y la ruta exacta desde la ciudad de origen hasta el destino.

2.2. - Tipificación del problema

Variables del algoritmo:

1. **n (int)**: Número de ciudades en la red de rutas aéreas.
2. ***vCiudades (int)****: Vector que marca si una ciudad ha sido visitada (0) o no (1).
3. **matriz (int**)**: Matriz de adyacencia que representa el grafo de la red de rutas aéreas, donde matriz[i][j] es el costo del vuelo directo desde la ciudad i hasta la ciudad j.
4. ***S (int)***: Vector que almacena las distancias mínimas desde la ciudad de origen hasta cada ciudad.
5. **origen (int)**: Índice de la ciudad de origen desde la cual se calcularán las rutas más económicas.
6. ***sCiudades (string)***: Vector de nombres de las ciudades.
7. ***ruta (int)***: Vector que almacena la ciudad previa en la ruta óptima para cada ciudad

Función objetivo:

Minimizar el costo total de viajar desde la ciudad de origen hasta todas las demás ciudades. Para cada ciudad j (donde j varía de 1 a n), queremos minimizar S[j], donde S[j] es la distancia mínima acumulada desde la ciudad de origen hasta la ciudad j.

Restricciones:

- **Condición de no visitados**: Cada ciudad debe ser considerada no visitada inicialmente.

$$vCiudades[i] = 1 \ \forall \ i \in \{1,2,...,n\}$$

- **Inicialización del costo**: La distancia a la ciudad de origen es 0, y las distancias a todas las demás ciudades son inicializadas con infinito (o un valor muy grande).

$$S[origen] = 0$$

2.3. - Estrategia de programación y pseudocódigo

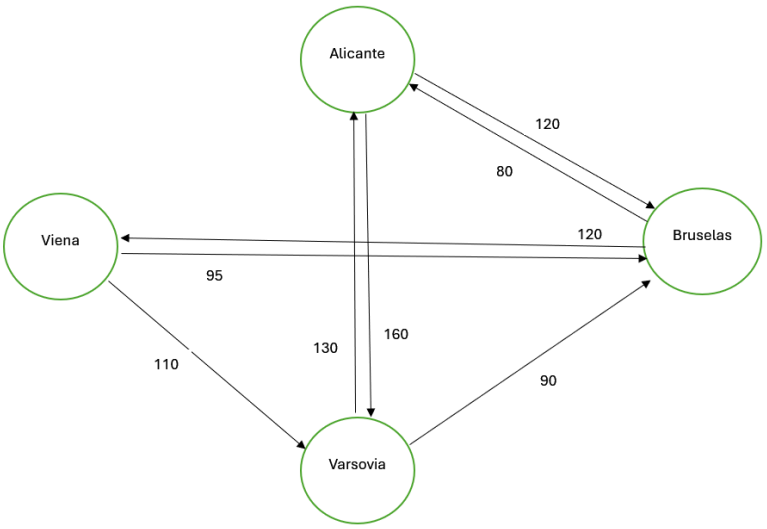
Para el desarrollo de este algoritmo se ha utilizado la técnica de diseño de algoritmos voraces, esta técnica encuentra una solución evaluando los elementos que forman parte del problema, descartandolos o seleccionandolos para formar parte de la solución siguiendo un criterio determinado de selección.

```
funcion dijkstra_v2(n, vCiudades, matriz, S, origen, sCiudades)
    para i ← 1 hasta n hacer
        ruta[i] ← 0
    fpara

    para i ← 1 hasta n hacer
        vCiudades[i] ← 1
    fpara

    // Inicialización de distancias desde el origen
    para i ← 1 hasta n hacer
        si i = origen hacer
            S[i] ← 0
        si matriz[origen][i] ≠ 0 hacer
            S[i] ← matriz[origen][i]
        si no
            S[i] ← INT_MAX
        fsi
    fpara

    // Aplicación del algoritmo de Dijkstra
    para i ← n - 1 hacer
        x ← elegir_vertice_coste_min(vCiudades, S, n)
        si x = -1 hacer
            salir del bucle
        fsi
        vCiudades[x] ← 0
        para j ← 1 hasta n hacer
```

```
            si vCiudades[j] ≠ 0 && matriz[x][j] ≠ 0 && S[x] ≠ INT_MAX hacer
                S[j] ← min(S[j], S[x] + matriz[x][j])
                si S[j] == S[x] + matriz[x][j] hacer
                    ruta[j] ← x
                fsi
            fsi
        fpara
    fpara

    // Impresión de resultados
    para i ← 1 hasta n hacer
        si i ≠ origen hacer
            imprimirRuta(ruta, i)
        fsi
    fpara

ffuncion
```

2.4. - Ejemplo del funcionamiento del algoritmo

Para realizar la traza de este algoritmo, utilizaremos este grafo:



Una vez leídos los datos, obtenemos la matriz de adyacencia:

	Alicante	**Bruselas**	**Varsovia**	**Viena**
Alicante	0	120	160	∞
Bruselas	80	0	0	120
Varsovia	130	90	0	∞
Viena	∞	95	110	0

A continuación, seleccionamos Alicante como ciudad origen (ciudad 1) y mostramos la traza del algoritmo paso a paso.

Paso 1:

- **X** toma el valor del nodo de origen (Alicante).
- Seleccionamos Bruselas como el nodo con menor costo (120) y se establece este valor a 0.

Desde Alicante (1):

- La distancia a sí mismo es 0.
- La distancia a Bruselas (2) es 120.
- La distancia a Varsovia (3) es 160.
- No hay vuelo directo a Viena (4), por lo que la distancia es ∞.

Entonces, las variables tendrán los siguientes valores iniciales:

X: 1 (Alicante) **vCiudades**: {2, 3, 4} **S** (distancias): [0, 120, 160, ∞]

Paso 2:

- **X** toma el valor de Bruselas (el nodo con menor costo).
- Actualizamos el costo de Viena: 120 (costo a Bruselas) + 120 (costo de Bruselas a Viena) = 240.
- Seleccionamos Viena como el siguiente nodo con menor costo.

Desde Bruselas (2):

- La distancia a Alicante (1) es 80 (ya visitado, no se actualiza).
- No hay vuelo directo a Varsovia (3), por lo que el costo permanece en 160.
- La distancia a Viena (4) se actualiza: 120 + 120 = 240.

Las variables ahora son:

X: 2 (Bruselas) **vCiudades**: {3, 4} **S** (distancias): [0, 120, 160, 240]

Paso 3:

- **X** toma el valor de Viena (el siguiente nodo con menor costo).
- No hay actualización de costos ya que los destinos restantes tienen menores costos.

Desde Varsovia (3):

- La distancia a Alicante (1) es 130 (ya visitado, no se actualiza).
- La distancia a Bruselas (2) es 90 (ya visitado, no se actualiza).
- No hay vuelo directo a Viena (4), el costo permanece en 240.

Las variables ahora son:

X: 3 (Varsovia) **vCiudades**: {4} **S** (distancias): [0, 120, 160, 240]

Paso 4:

- **X** toma el valor de Varsovia.
- Todos los destinos han sido visitados.

Desde Viena (4):

- La distancia a Alicante (1) es ∞ (ya visitado).
- La distancia a Bruselas (2) es 95 (ya visitado).
- La distancia a Varsovia (3) es 110 (ya visitado).

Las variables finales son:

X: 4 (Viena) **vCiudades**: {} **S** (distancias): [0, 120, 160, 240]

Resultados Finales

Los caminos de menor costo desde Alicante (ciudad origen) a todos los destinos son

Destino	Precio	Ruta
Bruselas	120	Alicante -> Bruselas
Varsovia	160	Alicante -> Varsovia
Viena	240	Alicante -> Bruselas -> Viena

2.5.- Complejidad asintótica

Tal y como en la actividad anterior, el tamaño del problema es n, pues utilizamos una matriz cuadrada nxn.

A continuación identificamos los órdenes de cada una de las líneas del pseudocódigo.
Todo es de O(1) excepto los bucles i ← 1 o j ← 1 hasta n o n-1 son de O(n), la llamada a la función de elegir_vertice_coste_min es de Orden O(n) ya que está función emplea un bucle para hasta n y la función imprimirRuta que es de orden O(n) que utiliza un bucle por cada ciudad que forma parte del problema.

```
funcion dijkstra_v2(n, vCiudades, matriz, S, origen, sCiudades)                    Nivel 1   Nivel 2   Nivel 3   Nivel 4
    para i ← 1 hasta n hacer
        ruta[i] ← 0
    fpara

    para i ← 1 hasta n hacer                                                       O(n)
        vCiudades[i] ← 1                                                                 O(1)
    fpara

    // Inicialización de distancias desde el origen
    para i ← 1 hasta n hacer                                                       O(n)
        si i = origen hacer                                                              O(1)
            S[i] ← 0                                                                              O(1)
        si matriz[origen][i] ≠ 0 hacer                                                   O(1)
            S[i] ← matriz[origen][i]                                                              O(1)
        si no
            S[i] ← INT_MAX                                                                        O(1)
        fsi
    fpara

    // Aplicación del algoritmo de Dijkstra
    para i ← n - 1 hacer                                                           O(n)
        x ← elegir_vertice_coste_min(vCiudades, S, n)                                    O(n)
        si x = -1 hacer                                                                  O(1)
            salir del bucle                                                                       O(1)
        fsi
        vCiudades[x] ← 0                                                                 O(1)
        para j ← 1 hasta n hacer                                                         O(n)
            si vCiudades[j] ≠ 0 && matriz[x][j] ≠ 0 && S[x] ≠ INT_MAX hacer                       O(1)
                S[j] ← min(S[j], S[x] + matriz[x][j])                                                 O(1)
                si S[j] == S[x] + matriz[x][j] hacer                                                  O(1)
                    ruta[j] ← x
                fsi
            fsi
        fpara
    fpara

    // Impresión de resultados
    para i ← 1 hasta n hacer                                                       O(n)
        si i ≠ origen hacer                                                              O(1)
            imprimirRuta(ruta, i)                                                                 O(n)
        fsi
    fpara
ffuncion
```

Una vez asignadas las complejidades de todas las líneas del algoritmo, agrupamos las líneas del nivel más alto, como ambas son O(1), se multiplica con el orden de la sentencia if a la que pertenecen, pero como esta también es O(1), se queda tal como está.

A continuación, para el nivel 3 vuelve a ocurrir lo mismo, todas las sentencias tienen una complejidad de O(1) por lo que no surgen muchos cambios. En cambio, para la función imprimirRuta, como tiene una complejidad de n, al multiplicarlo con la sentencia del if con O(1) al que pertenece, se queda como O(n).

```
funcion dijkstra_v2(n, vCiudades, matriz, S, origen, sCiudades)                    Nivel 1   Nivel 2   Nivel 3
    para i ← 1 hasta n hacer
        ruta[i] ← 0
    fpara

    para i ← 1 hasta n hacer                                                       O(n)
        vCiudades[i] ← 1                                                                 O(1)
    fpara

    // Inicialización de distancias desde el origen
    para i ← 1 hasta n hacer                                                       O(n)
        si i = origen hacer                                                              O(1)
            S[i] ← 0                                                                     O(1)      O(1)
        si matriz[origen][i] ≠ 0 hacer                                                   O(1)
            S[i] ← matriz[origen][i]                                                     O(1)
        si no                                                                                      O(1)
            S[i] ← INT_MAX                                                               O(1)
        fsi
    fpara

    // Aplicación del algoritmo de Dijkstra
    para i ← n - 1 hacer                                                           O(n)
        x ← elegir_vertice_coste_min(vCiudades, S, n)                                    O(n)
        si x = -1 hacer                                                                  O(1)
            salir del bucle                                                              O(1)      O(1)
        fsi
        vCiudades[x] ← 0                                                                 O(1)
        para j ← 1 hasta n hacer                                                         O(n)
            si vCiudades[j] ≠ 0 && matriz[x][j] ≠ 0 && S[x] ≠ INT_MAX hacer              O(1)      O(n)
                S[j] ← min(S[j], S[x] + matriz[x][j])
                si S[j] == S[x] + matriz[x][j] hacer
                    ruta[j] ← x
                fsi
            fsi
        fpara
    fpara

    // Impresión de resultados
    para i ← 1 hasta n hacer                                                       O(n)
        si i ≠ origen hacer                                                              O(1)      O(n)
            imprimirRuta(ruta, i)                                                        O(n)
        fsi
    fpara
ffuncion
```

En el nivel 2, hay dos casos en los que hay sentencias con O(n) y O(1) dentro del mismo bucle, por lo que entre estas, se elige la de mayor complejidad para multiplicarla por la complejidad del bucle.

```
funcion dijkstra_v2(n, vCiudades, matriz, S, origen, sCiudades)                    Nivel 1   Nivel 2
    para i ← 1 hasta n hacer
        ruta[i] ← 0
    fpara

    para i ← 1 hasta n hacer                                                       O(n)      O(n)
        vCiudades[i] ← 1                                                                 O(1)
    fpara

    // Inicialización de distancias desde el origen
    para i ← 1 hasta n hacer                                                       O(n)
        si i = origen hacer                                                              O(1)
            S[i] ← 0                                                                     O(1)      O(1)
        si matriz[origen][i] ≠ 0 hacer                                                                 O(n)
            S[i] ← matriz[origen][i]
        si no
            S[i] ← INT_MAX
        fsi
    fpara

    // Aplicación del algoritmo de Dijkstra                                                           O(n²)
    para i ← n - 1 hacer                                                           O(n)
        x ← elegir_vertice_coste_min(vCiudades, S, n)                                    O(n)
        si x = -1 hacer                                                                  O(1)      O(n)
            salir del bucle
        fsi
        vCiudades[x] ← 0                                                                 O(1)
        para j ← 1 hasta n hacer                                                         O(n)      O(n)
            si vCiudades[j] ≠ 0 && matriz[x][j] ≠ 0 && S[x] ≠ INT_MAX hacer
                S[j] ← min(S[j], S[x] + matriz[x][j])
                si S[j] == S[x] + matriz[x][j] hacer
                    ruta[j] ← x
                fsi
            fsi
        fpara
    fpara

    // Impresión de resultados
    para i ← 1 hasta n hacer                                                       O(n)
        si i ≠ origen hacer                                                              O(n)
            imprimirRuta(ruta, i)
        fsi                                                                                    O(n²)
    fpara
ffuncion
```

Al multiplicar las complejidades, quedaría la siguiente captura:

```
funcion dijkstra_v2(n, vCiudades, matriz, S, origen, sCiudades)                    Nivel 1
    para i ← 1 hasta n hacer
        ruta[i] ← 0
    fpara

    para i ← 1 hasta n hacer                                                       O(n)
        vCiudades[i] ← 1
    fpara

    // Inicialización de distancias desde el origen
    para i ← 1 hasta n hacer                                                       O(n)
        si i = origen hacer
            S[i] ← 0
        si matriz[origen][i] ≠ 0 hacer
            S[i] ← matriz[origen][i]
        si no
            S[i] ← INT_MAX
        fsi
    fpara

    // Aplicación del algoritmo de Dijkstra
    para i ← n - 1 hacer                                                           O(n^2)
        x ← elegir_vertice_coste_min(vCiudades, S, n)
        si x = -1 hacer
            salir del bucle
        fsi
        vCiudades[x] ← 0
        para j ← 1 hasta n hacer
            si vCiudades[j] ≠ 0 && matriz[x][j] ≠ 0 && S[x] ≠ INT_MAX hacer
                S[j] ← min(S[j], S[x] + matriz[x][j])
                si S[j] == S[x] + matriz[x][j] hacer
                    ruta[j] ← x
                fsi
            fsi
        fpara
    fpara

    // Impresión de resultados
    para i ← 1 hasta n hacer                                                       O(n^2)
        si i ≠ origen hacer
            imprimirRuta(ruta, i)
        fsi
    fpara
ffuncion
```

Para sacar la complejidad asintótica del problema, elegimos el orden más alto del nivel 1 tras realizar el ejercicio, por lo tanto, la complejidad del problema es n^2.

3. - Bibliografía

1. **Título:** “Tema 4, Algoritmos Voraces”
   Autor/-es: Yolanda Marhuenda
   Año: 2023/2024

2. **Título:** Algoritmia / Algoritmos Voraces
   URL: https://es.wikibooks.org/wiki/Algoritmia/Algoritmos_voraces

3. **Título:** Algoritmos Voraces
   URL: https://aprende.olimpiada-informatica.org/algoritmia-voraz

4. **Título**: C/C++ Program for Dijkstra’s shortest path algorithm | Greedy Algo-7
   URL:
   https://www.geeksforgeeks.org/c-program-for-dijkstras-shortest-path-algorithm-greedy-algo-7/