Billetes de avión

Curso 2019/2020

Universidad Miguel Hernández

Albert Lorenzo Segarra

Adrián Garrido Pantaleón

Pedro Gázquez Ruiz

23/05/20

Índice

1.- Algoritmo para calcular precios

1.1.- Descripción

El algoritmo escogido, conocido como algoritmo de Dijkstra, es un algoritmo que se encarga de encontrar la ruta más corta entre vértices o nodos de un grafo pero para poder entender cómo funciona este algoritmo primero necesitamos entender qué es un grafo y qué es un algoritmo.

Podríamos definir un algoritmo como un conjunto de operaciones que calculan, siguiendo una serie de reglas, una solución para un determinado problema.

Simplificando el concepto de grafo, decimos que un grafo es un mapa donde los vértices o nodos son ciudades y las aristas son carreteras o caminos. Finalmente, la función del algoritmo será calcular la ruta más rápida desde un vértice origen hacia el resto de nodos, es decir, de una ciudad como podría ser Madrid a todas las demás ciudades que tengamos en el mapa.

Digamos que, nuestra empresa con sede en Madrid, nos comunica que tres agentes comerciales quieren visitar Roma, Londres y Barcelona para hacer negocios con otras empresas pero no saben qué vuelos escoger para que el costo del viaje sea el mínimo. Aquí plantearíamos tres viajes, Madrid→Barcelona, Madrid→Londres y Madrid→Roma. Siguiendo esta tabla, con la información precios y rutas.

	Madrid	Barcelona	Roma	Londres
Madrid		30	120	
Barcelona	50		70	50
Roma	140	60		30
Londres	25	100	80	

Con el primer vistazo, podemos asegurar que el vuelo más económico hasta Barcelona, es el vuelo directo ya que costaría Madrid→Barcelona (30€). El siguiente vuelo, destino Roma, costaría 100€ ya que de la forma más económica el trayecto consistiría en Madrid→Barcelona (30€)→Roma (70€). Finalmente, para visitar Londres, la mejor ruta sería posible es Madrid→Barcelona (30€)→Londres (50€) con un coste total de 80€.

1.2.- Pseudocódigo

```
//Variables globales

num_ciudades:entero

matriz:entero[num_ciudades,num_ciudades]

ciudades:string[num_ciudades]

función Dijkstra(src: entero, opcion: entero)
        coste:entero[num_ciudade], visitado:boleano[num_ciudades], u, v, i:entero


        para i ← 0 hasta num_ciudades hacer
                coste_i ← inf
                visitado_i ← falso
        fpara


        coste_src ← 0


        para i ← 0 hasta num_ciudades hacer
                u ← coste_minimo(coste, visitado)
                visitado_u ← verdadero
                para v ← 0 hasta num_ciudades hacer
```

si !visitado y $matriz_{u, v}$ y $coste_u \neq$ inf y $coste_u + matriz_{u, v} < coste_v$

$coste_v \leftarrow coste_u + matriz_{u, v}$

fsi

fpara

fpara

si opcion = 1

mostrar_rutas(coste, arbol_ruta, src)

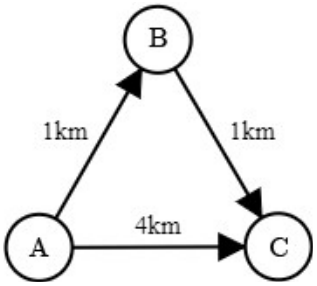
sino

mostrar_vuelos_baratos(src, coste)

ffunción

1.3.- Ejemplo

Para entender de forma más visual y detallada el algoritmo. Crearemos un ejemplo ficticio teniendo un conjunto formado por tres ciudades, *ciudades = {A, B, C}* y el siguiente grafo.



Observando las primeras posibles rutas, calculamos un coste de 4km para *A→C* pero, si consideramos tomar el atajo pasando por la ciudad *B* el coste final hasta *C* sería mucho menor por lo tanto, sólo necesitaríamos recorrer 2km en lugar de 4km, así que escogemos esta ruta en lugar de la directa.

Desde un punto de vista más técnico pero bastante sencillo, tomando como ejemplo el grafo anterior, para que el algoritmo funcione después de haber creado previamente este 'mapa', necesitaremos crear un conjunto o colección de ciudades que aún no hemos visitado.

Ej. nodos_no_visitados = {A, B, C} donde A-C es cualquier ciudad

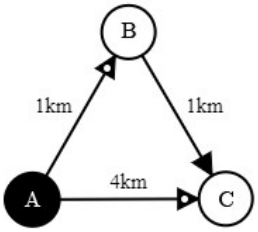
Seguidamente de haber creado el conjunto, elegimos un punto origen o ciudad de partida. Haciendo un ejercicio de imaginación, planteemos un viaje desde *{A}* hasta las demás ciudades *{B, C}*. Tras declarar este vértice o ciudad origen como ciudad actual en la que nos encontramos, tachamos de la lista de nodos no visitados la ciudad *A*.

Ej. nodos_no_visitados = {~~A~~, B, C}

Una vez llegados a este punto, crearemos otra colección con los costes de las rutas. Este conjunto se irá actualizando junto con nuestro avance por el mapa a la vez que estableceremos la ciudad actual que hemos identificado anteriormente como ciudad origen, con una distancia 0 puesto que nos encontramos en ella y para viajar desde A→A no necesitamos recorrer ningún camino.

Ej. coste_rutas = {A:0 , B: inf, C: inf} donde inf significa que no hemos establecido una ruta hacia ese nodo y 0 sería la distancia que hay desde el origen hacia ese propio nodo, indicando así que A es el punto origen.

El primer paso sería, desde la ciudad origen *A*, ver a qué ciudades podemos ir y plantear el coste de cada ruta. En este caso, tenemos disponibles los viaje *A→C* y *A→B* por lo tanto actualizaremos nuestro conjunto de costes.

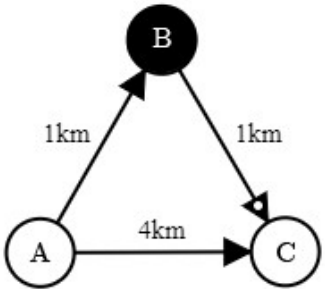


coste_rutas = {A:0 , B: 1, C: 4}

Después de plantear las rutas a las ciudades accesibles, seleccionamos la ruta más corta de entre todas las posibles rutas desde la ciudad origen, escogiendo la ciudad correspondiente a ese camino como ciudad actual sustituyendo a la anterior, es decir, si para *A→B* hay 1km y para *A→C* hay 4km, elegimos *A→B*. Finalmente, la tachamos del conjunto de ciudades no visitadas.

nodos_no_visitados = {A, ~~B~~, C}

Ahora que estamos situados en la ciudad *B*, recalculamos los costes teniendo en cuenta el punto de partida original hacia las demás ciudades que aún no hemos visto. Recordemos que. una vez tachada *B* del conjunto de ciudades no visitadas, la ruta más corta ya estaría calculada para *A→B* y no volveríamos a calcular para para *A→B*.

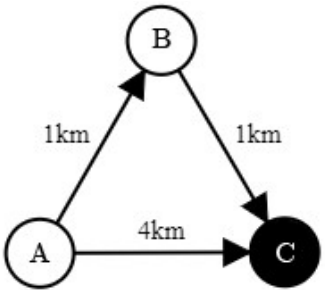


Una vez hecho el cálculo teniendo en cuenta la ruta anterior *A→B*, trazamos una nueva posible ruta para *A→C*, es decir, calculamos los costes de las posibles rutas para ciudades no visitadas conectadas con *B*. Para esto, miramos los caminos disponibles que hay desde *A→B→C* y comparamos si es mejor que el anterior *A→C*.

Observamos que para desplazarnos desde a *A→C* es mejor pasar por *B*, por lo tanto actualizamos el conjunto reemplazando la ruta establecida con anterioridad, sustituyendo 4km por 2km.

coste_rutas = {A:0 , B: 1, C: 2}

Finalmente, al movernos a la última ciudad no visitada, el algoritmo no encontrará ninguna ruta posible y, por lo tanto, se finalizará la ejecución del mismo.



1.4.- Estrategia de programación

La estrategia que el algoritmo sigue es de tipo voraz. Los algoritmos voraces son aquellos que resuelven un problema intentando tomar en cada paso la decisión correcta. Una vez tomada dicha decisión, no cambiaremos el resultado.

Algunas ventajas de esta estrategia es que suele ser fácil de implementar y las soluciones son eficientes a nivel de complejidad, además de que atacamos el problema para resolverlo de forma rápida.

Para profundizar un poco con esta estrategia, creamos una colección de candidatos, consideramos si es factible y si lo es nos preguntamos si ya tenemos la solución. El algoritmo voraz dejará de operar una vez no queden candidatos o tengamos una solución.

1.5.- Complejidad asintótica

Haciendo uso del anterior pseudocódigo:

```
para i ← 0 hasta num_ciudades hacer
        u ← coste_minimo(coste, visitado)
        visitado_u ← verdadero
        para v ← 0 hasta num_ciudades hacer
                si !visitado y matriz_{u, v} y coste_u ≠ inf y coste_u + matriz_{u, v} < coste_v
                        coste_v ← coste_u + matriz_{u, v}
                fsi
        fpara
fpara
```

Análisis por líneas:

1: O(n)

2: O(n)

3: O(1)

4: O(n)

5: O(1)

6: O(1)

Siguiendo las propiedades del análisis asintótico, eliminando constantes y guardando el elemento con más peso, la complejidad del algoritmo es $O(n^2)=O(V^2)$ ya que los bucles de las líneas 1-2 no están anidados pero 1-4 sí lo están, V representa el número de vértices.

Es muy importante entender que, según la estructura de datos utilizada para el grafo la complejidad podría reducirse. Para que esto sea posible podríamos utilizar la estructura de tipo *montículo binario* donde reduciríamos el crecimiento cuadrático a logarítmico siendo O(A log V) u O((A+V) log V) donde A es el número de aristas.

2.- Algoritmo para calcular precios y rutas

2.1.- Descripción

En esta parte del problema, nos centraremos en reconstruir la ruta que el algoritmo ha trazado para ir desde un vértice cualquiera hasta los demás vértices del 'mapa'.

Como hemos explicado anteriormente, cuando planteamos un viaje de *A*→*Z*, por ejemplo, analizamos por qué carreteras es mejor ir, si tenemos que pasar por otras ciudades, etc. El algoritmo de Dijkstra no está directamente pensado para almacenar dichas rutas sino para calcular el coste más económico de esas rutas o viajes.

Una manera de conocer estas rutas, se trata de emplear una reconstrucción desde la ciudad destino hasta la ciudad origen.

La reconstrucción de una ruta puede sonar abrumador pero realmente es algo bastante sencillo de entender. Si queremos ir desde Madrid hasta Londres y sabemos que nuestro algoritmo puede pasar por diferentes ciudades de ese mapa hasta llegar al destino, qué solución podemos aplicar para saber por dónde hemos pasado?

La solución consistirá en crear una lista y apuntar si hemos pasado o no por los diferentes puntos del mapa y desde el vértice o nodo final, echaremos un vistazo a las ciudades que hemos visto haciendo así una reconstrucción desde la ciudad final hasta llegar a la ciudad inicial.

2.2.- Estrategia de programación

Siguiendo con la estrategia principal, el algoritmo sigue una estrategia voraz para el algoritmo en sí pero para reconstruir la ruta haremos uso de la recursividad ya que necesitaremos reconstruir una ruta desde el nodo final hasta el nodo inicio mediante un caso base.

La solución consiste en saltar por los nodos por donde el algoritmo 'ha pasado' hasta llegar al caso base dando a entender que hemos reconstruido la ruta.

2.3.- Pseudocódigo

```
función Dijkstra(src: entero, opcion: entero)

    coste:entero[num_ciudade], arbol_rutas:entero[num_ciudades], visitado:boleano[num_ciudades], u,
    v, i:entero


    para i ← 0 hasta num_ciudades hacer
        coste_i ← inf
        visitado_i ← falso
    fpara


    coste_src ← 0
    arbol_rutas_src ← -1


    para i ← 0 hasta num_ciudades hacer
        u ← coste_minimo(coste, visitado)
        visitado_u ← verdadero
        para v ← 0 hasta num_ciudades hacer
            si !visitado y matriz_u,v y coste_u ≠ inf y coste_u + matriz_u,v < coste_v
```

```
                        arbol_rutas_v ← u
                        coste_v ← coste_u + matriz_u, v
                    fsi
                fpara
            fpara

            si opcion = 1
            mostrar_rutas(coste, arbol_ruta, src)
            sino
            mostrar_vuelos_baratos(src, coste)
ffunción
```

2.4.- Ejemplo

El ejemplo para el cálculo de las rutas es bastante sencillo. Primero, tomamos los conjuntos de *ciudades* y *arbol_rutas* una vez el algoritmo ha finalizado.

Ciudades = {A:0, B:1, C:2} donde {0,1,2} son los índices de cada ciudad.

Arbol_rutas = {-1, 0, 1}

Si hemos calculado el algoritmo tomando como origen el punto *A* y sabiendo que tenemos 3 ciudades, tendremos como máximo 3 diferentes rutas.

La primera ruta o primer caso, buscará directamente el nodo origen ya que estamos en el índice 0 así que no imprimirá ninguna ruta. También tendremos un caso base para indicar que se ha llegado al origen cuando el nodo sea igual a -1.

Llamamos a la función con los parámetros:

imprimir_ruta(Arbol_rutas = {-1, 0, 1}, 0)

arbol_ruta[0] == -1

return

Como estamos en la primera ciudad, no habrá que reconstruir nada ya que el índice es el origen.

Para el segundo caso, llamamos a la función con los siguientes parámetros:

imprimir_ruta(Arbol_rutas = {-1, 0, 1}, 1)

arbol_ruta[1] ≠ -1

return

imprimir_ruta(Arbol_rutas = {-1, 0, 1}, 0)

imprimir(ciudades[j])

Llamada recursiva:

imprimir_ruta(Arbol_rutas = {-1, 0, 1}, 0)

arbol_ruta[0] == -1

return

Con este proceso habríamos reconstruido la ruta A→B

Para el tercer caso:

imprimir_ruta(Arbol_rutas = {-1, 0, 1}, 2)

arbol_ruta[2] ≠ -1

return

imprimir_ruta(Arbol_rutas = {-1, 0, 1}, 1)

imprimir(ciudades[j])

imprimir_ruta(Arbol_rutas = {-1, 0, 1}, 1)

arbol_ruta[2] ≠ -1

return

imprimir_ruta(Arbol_rutas = {-1, 0, 1}, 0)

imprimir(ciudades[j])

imprimir_ruta(Arbol_rutas = {-1, 0, 1}, 0)

arbol_ruta[2] ≠ -1

return

Y con esto tendríamos la traza para la ruta A→B→C.

2.5.- Complejidad asintótica

Pseudocódigo a analizar.

```
para i ← 0 hasta num_ciudades hacer
    u ← coste_minimo(coste, visitado)
    visitado_u ← verdadero
    para v ← 0 hasta num_ciudades hacer
        si !visitado y matriz_{u,v} y coste_u ≠ inf y coste_u + matriz_{u,v} < coste_v
            arbol_rutas_v ← u
            coste_v ← coste_u + matriz_{u,v}
        fsi
    fpara
fpara
mostrar_rutas(coste, arbol_ruta, src)
```

Análisis por líneas:

1: O(n)

2: O(n)

3: O(1)

4: O(n)

5: O(1)

6: O(1)

7: O(1)

11: O(n)

Siguiendo las propiedades del análisis asintótico, eliminando constantes y guardando el elemento con más peso, la complejidad del algoritmo es $O(n^2)=O(V^2)$ como el anterior algoritmo.

3.- Bibliografía

- MIT OpenCourse sobre Dijkstra y problemas SSSP.

  URL: Enlace

  URL: Enlace
- How to Implement Dijkstra’s algorithm in C++

  URL: Enlace
- Algoritmo Dijsktra

  URL: Enlace
- 8 Printing paths in Dijkstra’s shortest path algorithm

  URL: Enlace
- Printing Paths in Dijkstra’s algorithm

  URL: Enlace
- Dijkstra's algorithm in 3 minutes

  URL: Enlace