FACULTAD DE INGENIERIA
Universidad de Buenos Aires

Trabajo Práctico 2 — Algoritmos D&C y Programacion Dinamica

[75.29/95.06] Teoria de Algoritmos I
Primer cuatrimestre de 2022

Alumno	Padron	Email
BENITO, Agustin	108100	abenito@fi.uba.ar
BLÁZQUEZ, Sebastián	99673	sblazquez@fi.uba.ar
DEALBERA, Pablo Andres	106585	pdealbera@fi.uba.ar
DUARTE, Luciano	105604	lduarte@fi.uba.ar
PICCO, Martín	99289	mpicco@fi.uba.ar

Entrega:	Primera
Fecha:	Miercoles 27 de Abril del 2022

Índice

1. Parte 1: Un evento exclusivo **2**

2. Parte 2: Ciclos negativos **2**
2.1. Solución con Bellman-Ford 2
2.1.1. Algoritmo 2
2.1.2. Complejidad 3
2.1.3. Relación de Recurrencia 3
2.2. Detalles de implementación 4
2.2.1. Ejecución del programa 4

3. Parte 3: Un poco de teoría **4**

1. Parte 1: Un evento exclusivo

2. Parte 2: Ciclos negativos

En esta segunda parte del trabajo practico, se nos presenta el problema de analizar un grafo dirigido ponderado con valores enteros y verificar si tiene, por lo menos, un ciclo negativo. En el caso de tenerlo, debemos mostrar en pantalla los nodos que contienen a dicho ciclo. Además, se nos pide que la solución presentada utilice programación dinámica.

2.1. Solución con Bellman-Ford

Para resolver el problema enunciado podemos utilizar el algoritmo Bellman-Ford para calcular el camino mínimo en un grafo ponderado con aristas negativas a partir de un nodo de origen. Lógicamente, no podría encontrarse un camino mínimo distinto a (menos infinito) si es que hay ciclos negativos. Esto es porque caeríamos en un punto donde resulta conveniente recorrer dicho ciclo infinitamente ya que cada vez se reduce más la longitud del camino mínimo. Vamos a ver que esto puede ser utilizado para encontrar ciclos negativos.

Primero, veamos como calcular el camino mínimo:

2.1.1. Algoritmo

Primero, se inicializa la distancia del nodo de origen n_s hasta todos los vértices como infinito y la distancia al nodo n_s como cero. Luego por cada arista se verifica si la distancia guardada para llegar al vértice de origen de la arista sumado al peso de la arista es menor a la guardada para llegar al vértice destino de la arista. Esto se repite $N \check{}1$ veces con N el número de nodos del grafo. De esta manera, en cada iteración i, el algoritmo encuentra el camino mínimo de longitud máxima i. Es por esto que el ciclo se repite $N \check{}1$ veces, porque el camino mínimo sin ciclos podría ser de esa longitud. Es en este punto donde el algoritmo nos es de utilidad. Podemos aplicar el mismo procedimiento una vez, es decir viendo si se puede encontrar un camino mínimo de longitud N que sea menor al encontrado de longitud $N \check{}1$. Si esto sucede, implica que estamos agregando una arista negativa formando un ciclo negativo. Es decir, identificamos el ciclo negativo que se nos pide en el enunciado.

El algoritmo de Bellman-Ford termina ahí, en el caso de encontrar un ciclo negativo devuelve error y en caso contrario devuelve el camino mínimo o una estructura para reconstruirlo. En nuestro caso, necesitamos adaptar el algoritmo para que devuelva un ciclo negativo si es que hay o nada en caso de no haber. Entonces, lo que podemos hacer es una vez que sabemos que hay un ciclo negativo, iterar una vez mas y reconstruir el ciclo negativo.

1. Pseudo-codigo

```
funcion Bellman_Ford(Grafo grafo, Nodo origen)
    siendo distancias un diccionario
    siendo predecesores un diccionario

    por cada vertice v del grafo:
        distancias[v] = infinito

    distancias[origen] = 0
    padres[origen] = None

    por cada vertice:
        por cada arista de origen v y destino w y peso:
            si distancias[v] + peso < distancias[w]:
                predecesores[w] = v
                distancias[w] = distintas[v] + peso

    por cada arista de origen v y destino w y peso:
        si distancias[v] + peso < distancias[w]:
            tirar error "Hay un ciclo negativo"
```

2.1.2. Complejidad

Podemos analizar la complejidad a partir del pseudo-codigo. Primero, tenemos un ciclo donde inicializamos las distancias de cada vértice al origen como infinito. Es decir O(V), donde V es la cantidad de vértices. Luego, por cada vértice sin contar el origen recorremos todas las aristas adyacentes que en el peor de los casos resulta O(E). Es decir que el ciclo entero tiene una complejidad de O(V * E). Finalmente, encontrar el ciclo negativo y devolverlo tiene una complejidad de O(E) porque se recorren todas las aristas una vez más. Entonces, la complejidad final del algoritmo es de O(V * E).

2.1.3. Relación de Recurrencia

Sea n_s en nodo de inicio, T el nodo final, N_i un nodo y $predecesores[N_i]$ es el conjunto de los nodos adyacentes a N_i, sabemos que para llegar desde el nodo n_s al nodo N_i en una cantidad de pasos j debemos haber llegado a alguno de sus predecesores en $j-1$ pasos. Entonces, siendo la longitud la cantidad de nodos que se recorren hasta llegar al nodo N_i, se deduce de lo planteado que el camino mínimo hasta el nodo N_i dada una longitud máxima L es el mínimo de los caminos hacia sus predecesores mas la longitud de llegar del predecesor a N_i. Por lo tanto, nuestra ecuación de recurrencia resulta:

Definiendo:

- n_s: nodo origen o *source node*
- n_i: otro nodo distinto al origen
- j: longitud máxima para llegar de n_s a n_i
- $minPath(n_i, j)$: función recursiva para llegar al camino mínimo de s a n_i
- n_x: predecesores a n_i
- k: cantidad de predecesores a n_i
- $w(n_x, n_i)$: peso de la arista n_x y n_i

$$minPath(n_s, j) = 0$$

$$minPath(n_i, 0) = +\infty \text{ con } n_i \neq S$$

$$minPath(n_i, j) = min \left\{ \begin{array}{l} minPath(n_i, j-1) \\ min \left\{ \begin{array}{l} minPath(n_{x1}, j-1) + w(n_{x1}, n_i) \\ minPath(n_{x2}, j-1) + w(n_{x2}, n_i) \\ ... \\ minPath(n_{xk}, j-1) + w(n_{xk}, n_i) \end{array} \right. \end{array} \right\}$$

2.2. Detalles de implementación

El algoritmo fue implementado en Python y no tiene dependencias aparte de tener instalado cualquier versión de `python3`.

2.2.1. Ejecución del programa

El programa contiene un `shebang` para ser ejecutado en una terminal de la siguiente forma:

```
./src/parte_2.py <filename>
```

El comprimido entregado incluye un archivo ejemplo en `assets/grafo.txt` con grafos ejemplos, por ejemplo:

```
B
D,A,-2
B,A,3
D,C,2
C,D,-1
B,E,2
E,D,-2
A,E,3
```

```
./src/parte_2.py ./assets/grafo.txt
```

```
Existen al menos un ciclo negativo en el grafo. A,E,D → costo: -1
```

3. Parte 3: Un poco de teoría