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. 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. Sub-estructura optima

Para calcular las distancias mínimas de un nodo hacia otro se utiliza la distancia mínima calculado anteriormente para su predecesor mas el peso de llegar a el. Entonces la subestructura seria los caminos mínimos de todos los nodos predecesores al nodo final.

2.1.3. 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).

2.1.4. 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

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

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

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

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
```

2.2.2. Estructuras de datos

Para la representación del grafo decidimos manteneral simple:

- Un lista de aristas para almacenar las aristas tal cual como estan en el archivo.
- Un set de vértices para mantener un registro de los vértices ingresados en cada arista.

2.2.3. Implementación de Bellman-Ford en Python

Al igual que el pseudo-codigo, podemos describir la implementación de la siguiente manera:

1. Iniciamos:
   - un diccionario de distancias con clave `vertice` y valor infinito.
   - un diccionario de predecesores donde la clave `origen` se inicializa en `None`
   - la distancia de clave `origen` se cambia a `0`.
2. Iterar por la cantidad de vértices del grafo:
   - por cada arista, si la distancia guardada para llegar al origen de la arista mas el peso de moverse al nodo destino de la arista es menor a la distancia guardada para llegar al nodo destino de la arista, reemplazar la distancia guardada del nodo destino.
   - ademas, verificamos si no hubo un cambio en la iteración de aristas, si este es el caso, podemos confirmar que no existe ningún ciclo negativo por lo que devolvemos.
3. Verificar que no haya ciclos negativos
   - por cada arista, si se sigue cumpliendo la condición del punto anterior, entonces hay un ciclo negativo
   - si hay un ciclo negativo:
     - reconstruir los nodos predecesores hasta llegar al nodo que se detecto y sumar los pesos de sus aristas.
     - devolver el ciclo negativo y su peso

2.2.4. Calcular ciclo negativo

A partir del algoritmo de Bellman-Ford agregamos código cuando se detecta el ciclo negativo que agrega el nodo que se detecto termina el ciclo negativo y se reconstruye los nodos predecesores iterando hasta volver al nodo original mientras que se suman todos sus pesos en la variable `peso_ciclo`.

Luego devolvemos al `ciclo` reconstruido invertido y el `peso_ciclo` calculado.

2.2.5. Complejidad de la implementación

Con la simple estructura que decidimos usar, el codigo y el pseudo-codigo tiene pocas diferencias, y la complejidad termina siendo la misma $O(V * E)$.

En el código de Python tenemos las siguiente operaciones:

- Inicializar las distancias en infinito que lo hacemos con un simple `for` sobre `grafo.vertices` por lo que la complejidad computacional es `O(V)`.
- Luego hacemos un `for` anidado entre `grafo.vertices` (un *set* de python) y `grafo.aristas` (un *lista* de python), y como decidimos tener estructura simple (cuando creamos el grafo almacenamos los vértices y las aristas como vienen), la complejidad termina siendo la multiplicación de los dos ciclos es decir `O(V * E)` ya que en Python tanto iterar sobre listas o sobre sets es `O(n)`.

3. Parte 3: Un poco de teoría

3.1. División y Conquista

La división y conquista es una técnica algorítmica que consiste en 3 elementos claves. El primero es, dividir el problema en N subproblemas menores pero que se puedan resolver igual. Luego, resolver estos problemas de manera recursiva planteando un caso base, es decir un caso donde el problema se resuelva de forma trivial y no haga falta dividirlo. Finalmente, se combina la solución hallada de todos los subproblemas para formar la solución general.

Entonces, es evidente que para que un problema se pueda resolver por división y conquista debe tener subestructura optima. La complejidad temporal utilizando esta estrategia se calcula una vez obtenida la relación de recurrencia del problema. Para ello, se puede, o bien desarrollar la relación de recurrencia matemáticamente utilizando el costo del caso base o, utilizar el Teorema Maestro. Este ultimo teorema, no necesariamente resuelve todos los problemas de división y conquista pero por lo general es una herramienta muy útil.

3.2. Programacion Dinamica

3.3. Comparación de estrategias

No existe una mejor estrategia, algunos problemas directamente no se pueden plantear usando una de las tres estrategias vistas, por lo que plantear una mejor no tiene sentido.

En el caso de se pueda resolver con las tres, vas a terminar eligiendo una estrategia sobre otra por la que resuelva el problema o con mejor complejidad temporal o menor complejidad espacial como se explica para el siguiente problema:

3.4. El problema

Se trabaja sobre una matriz. El ejercicio no da mucha información acerca de la complejidad espacial, por lo que tenemos dar supuestos:

El Algoritmo Greedy realiza `N^3` operaciones sobre la matriz. En primero lugar, podríamos suponer que la complejidad espacial es `O(algo)`, en el caso del de Programación Dinámica que divide el problema en `n^2` sub-problemas, podríamos decir que la complejidad espacial de este ultimo es `O(algo * n^2)` por lo que siempre se cumpliría que la complejidad espacial de Programación Dinámica es mayor a la del Algoritmo Greedy.

El de Programación Dinámica realiza N^2 operaciones sobre la matriz, por lo que la *complejidad temporal* del Algoritmo Greedy es menor Programación Dinámica.

Frente a dos algoritmos que resuelven el mismo problema, siempre se deberia elegir aquel que tenga menor complejidad temporal y espacial. En caso de tener igual complejidad espacial, se elige el de menor complejidad temporal y viceversa.

El problema surgue cuando tenes que uno tiene una complejidad mayor que la otra, en ese caso vas a tener que tener en cuenta el sistema donde se va a ejecutar y las condiciones.

3.4.1. ¿Porque eligiria el Algoritmo Greedy?

Si yo tengo poca memoria (como el caso de sistema embedidos), yo tal vez ni siquiera tendria la posibilidad de poder ejecutar el algoritmo con Programación Dinamica porque no tengo suficiente memoria para procesarlo.

3.4.2. ¿Porque eligiria el de Programación Dinámica?

En computadoras modernas, la mayoría de los sistemas vienen con bastante memoria (>8GB) entonces podría despreciar el hecho de que necesito mas memoria para procesar el algoritmo en orden de ganar en tiempo de ejecución.

3.4.3. Conclusión

No hay un algoritmo perfecto para todos los casos, solo hay algoritmos optimos para un caso de uso. Elegir entre un algoritmo que tiene complejidad espacial $O(n^2)$ y complejidad espacial $O(n^2)$ y otro algoritmo de complejidad espacial $O(1)$ y complejidad temporal $O(n^3)$, dependiendo de las condiciones donde se ejecuta, el algoritmo eligiria uno sobre otro, pero la realidad es que a menos que este trabajando en sistemas embebidos (o algun sistema especifico con muy poca memoria), es mas probable que priorice la complejidad temporal que la espacial.