Diseño y Análisis de Algoritmos

Proyecto # 2: Tito el corrupto

Laura Victoria Riera Pérez
Marié del Valle Reyes

Cuarto año. Ciencias de la Computación.
Facultad de Matemática y Computación, Universidad de La Habana, Cuba

20 de mayo de 2023

I. Repositorio del proyecto

https://github.com/computer-science-crows/algorithms-design-and-analysis

II. Definición inicial del problema

Tito se dió cuenta de que la carrera de computación estaba acabando con él y un día decidió darle un cambio radical a su vida. Comenzó a estudiar Ingeniería Industrial. Luego de unos años de fiesta, logró finalmente conseguir su título de ingeniero. Luego de otros tantos años ejerciendo sus estudios, consiguió ponerse a la cabeza de un gran proyecto de construcción de carreteras.

La zona en la que debe trabajar tiene n ciudades con m posibles carreteras a construir entre ellas. Cada ciudad que sea incluida en el proyecto aportará a_i dólares al proyecto, mientras que cada carretera tiene un costo de w_i dólares. Si una carretera se incluye en el proyecto, las ciudades unidas por esta también deben incluirse.

El problema está en que Tito quiere utilizar una de las habilidades que aprendió en sus años de estudio, la de la malversación de fondos. Todo el dinero necesario para el proyecto que no sea un aporte de alguna ciudad, lo proveerá el país y pasará por manos de Tito. El dinero aportado por las ciudades no pasará por sus manos. Tito quiere maximizar la cantidad de dinero que pasa por él, para poder hacer su magia. Ayude a Tito a seleccionar el conjunto de carreteras a incluir en el proyecto para lograr su objetivo.

III. Definición en términos matemático - computacionales

La entrada de nuestro problema es la cantidad de ciudades n, la cantidad de carreteras m, una lista a con el dinero aportado por cada ciudad y una lista w con cada carretera y su costo. La salida del problema son dos lista, una que contiene las ciudades seleccionadas para construir las carreteras y otra que contiene las carreteras seleccionadas. El objetivo es encontrar la combinación de ciudades y carreteras que maximice el dinero que llega a Tito, teniendo en cuenta que el dinero de las ciudades no pasa por él.

En la resolución del problema se construye un grafo dirigido $G' = (V', E')$, donde $|V'| = m + n + 2$ y $|E'| = 3 \cdot \mathrm{m} + \mathrm{n}$. El conjunto V' de vértices está conformado por vértices que representen cada ciudad y cada carretera, además de un vértice s que represente la fuente y un vértice t que representa el receptor. El conjunto de los arcos E', contiene los siguientes arcos:

- Arcos desde s hasta cada vértice que representa una carretera. La capacidad de estos arcos es el valor w_i dado de entrada, que representa el costo de cada carretera.
- Arcos desde cada vértice que representa una ciudad hacia el vértice t. La capacidad de estos arcos es el valor a_i que representa el aporte de cada ciudad.
- Arcos desde cada vértice que representa una carretera hacia cada vértice que representa una ciudad. La capacidad de estos arcos es $infinito$.

La idea es aplicar un algoritmo de flujo a G', particularmente el algoritmo de Ford-Fulkerson.

IV. Solución

I. Backtrack

Como primera solución al problema fue implementado un *backtrack*. Esta es una solución correcta, ya que prueba todas las combinaciones y se queda con la que más valor aporte, pero muy ineficiente, $O(2^{|E|})$, ya que por cada arista del grafo original, se decide si escogerla o no en la solución. En una computadora de 32GB de RAM, intel core i7-11na generación, se puede resolver para una cantidad máxima 5 ciudades con 5 aristas. Dicha solución puede ser encontrada en src/solutions/backtrack_solutions.py.

II. Corruption-Strategy

ii.1. Preliminares

Definición 1. *Una **red de flujo** $G = (V, E)$ es un grafo dirigido en el que a cada par ordenado (u,v), u, v∈V, se le asocia una función de capacidad no negativa c(u,v)≥ 0 y en el que se distinguen dos vértices: la fuente s y el receptor t.*

Definición 2. *Sea G=⟨V,E⟩ una red de flujo con función de **capacidad** c y vértices fuente y receptor s y t respectivamente. Un **flujo** en G es una función real $f : VxV \to \mathbb{R}^+$ que satisface las siguientes propiedades:*

1. *Restricción de capacidad: Para todo u,v∈V se cumple que*

$$0 \leq f(u, v) \leq c(u, v) \tag{1}$$

2. *Conservación de flujo: Para todo u∈V- {s,t}, se cumple que*

$$\sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v) \tag{2}$$

Definición 3. *A la cantidad no negativa f(u,v) se le denomina **flujo neto** de u a v.*

Definición 4. *El **valor de un flujo** se define como:*

$$|f| = \sum_{v \in V} f(s, v) \tag{3}$$

Definición 5. *Sea G=⟨V,E⟩ una red de flujo con origen s y receptor t. Sea f un flujo en G, y sean u,v ∈ V. Se define la **capacidad residual** mediante la siguiente función:*

$$c_f(u, v) = \begin{cases} c(u, v) - f(u, v) & \text{, si } (u,v) \in E \\ f(v, u) & \text{, si } (v,u) \in E \\ 0 & \text{, en otro caso} \end{cases} \tag{4}$$

*La **red residual** de G inducida por f es una red* $G_f = (V, E_f)$*, donde* $E_f = \{(u,v) \in VxV \mid c_f(u,v) > 0\}$.

Definición 6. *Si f es un flujo en la red original G y f' es un flujo en la red residual correspondiente G_f, entonces, se define el aumento del flujo f (en la red original) por f', y denotado por $f \uparrow f'$, a la función $f \uparrow f'$: VxV→ ℝ dada por la expresión:*

$$f \uparrow f'(u,v) = \begin{cases} f(u,v) + f'(u,v) - f'(v,u) & \text{si (u,v)} \in E \\ 0 & \text{en otro caso} \end{cases} \tag{5}$$

Lema 1. $|f \uparrow f'| = |f| + |f'|$

Definición 7. *Dada una red de flujo G=⟨V,E⟩ y un flujo f, un **camino aumentativo** p es un camino simple de s a t en la red residual G_f.*

Definición 8. *La capacidad residual de un camino aumentativo p, denotada por $c_f(p)$, es el valor máximo en el cual es posible aumentar el flujo en cada arista del camino sin violar la restricción de capacidad*

$$c_f(p) = min\{c_f(u,v) | (u,v) \in p\} \tag{6}$$

Lema 2. *Sea G=⟨V,E⟩ una red de flujo, sea f un flujo en G y sea p un camino aumentativo en G_f. Si se define el flujo f_p , sobre la red residual, como*

$$f_p(u,v) = \begin{cases} c_f(p) & \text{si (u,v)} \in p \\ 0 & \text{en otro caso} \end{cases} \tag{7}$$

entonces el valor de f_p es $|f_p| = c_f(p) > 0$

Corolario 1. *Sea G=⟨V,E⟩ una red de flujo, sea f un flujo en G y sea p un camino aumentativo en G_f. Sea f_p definida mediante la ecuación * y supóngase que se aumenta f por f_p. Entonces la función $f \uparrow f_p$ es un flujo en G con valor $|f \uparrow f_p| = |f| + |f_p| > |f|$.*

Definición 9. *Un **corte** (S,T) de una red de flujo G = (V,E) es una partición de V en dos conjuntos S y T = V-S de modo ue $s \in S$ y $t \in T$.*

Definición 10. *Si f es un flujo, entonces el **flujo neto** f(S,T) a través del corte (S,T) se define como*

$$f(S,T) = \sum_{u \in S} \sum_{v \in T} f(u,v) - \sum_{u \in S} \sum_{v \in T} f(u,v) \tag{8}$$

*Además, la **capacidad** del corte (S,T) se define como*

$$c(S,T) = \sum_{u \in S} \sum_{v \in T} c(u,v). \tag{9}$$

Lema 3. *Sea f un flujo en una red de flujo G con origen s y receptor t. Sea (S,T) un corte cualquiera de G. Entonces el flujo neto a través del corte f(S,T) es igual al valor del flujo, osea, $|f|$*

$$f(S,T) = |f| \tag{10}$$

Corolario 2. *El valor de cualquier flujo en una red de flujo G está acotado superiormente por la capacidad de cualquier corte de G.*

$$f(S,T) = |f| \leq c(S,T) \tag{11}$$

Teorema 1. *Si f es un flujo en una red de flujo G=⟨V,E⟩ con un origen s y un receptor t, entonces las siguientes condiciones son equivalentes:*

1. *f es un flujo máximo en G.*
2. *En la red residual G_f no se pueden encontrar más caminos aumentativos.*
3. *$|f|$=c(S,T) para algún corte (S,T) de G.*

ii.2. Algortimo Ford-Fulkerson

El algoritmo Ford-Fulkerson es utilizado para encontrar el flujo máximo en una red de flujo [2]. Esta red se representa como un grafo dirigido donde cada arista tiene una capacidad que indica la cantidad máxima de flujo que puede pasar por ella.

El algoritmo encuentra el flujo máximo en la red de flujo utilizando la técnica de aumentar caminos. En cada iteración del algoritmo, se busca un camino aumentativo [7], es decir, un camino desde el nodo fuente hasta el nodo receptor donde todas las aristas tienen capacidad positiva y suficiente para aumentar el flujo actual. Una vez encontrado el camino aumentativo, se aumenta el flujo en esa ruta tanto como sea posible. Este proceso se repite hasta que ya no haya caminos aumentativos en el grafo residual.

El grafo residual se obtiene a partir del grafo original restando el flujo actual del flujo máximo para obtener la capacidad residual de cada arista. De esta manera, se pueden buscar caminos amentativos en el grafo residual sin utilizar aristas que ya están completamente saturadas.

El algoritmo Ford-Fulkerson garantiza que, una vez que se alcanza el flujo máximo, no hay caminos aumentativos en el grafo residual y, por lo tanto, el flujo es óptimo.

ii.3. Propuesta de solución óptima

Una vez creado el grafo G' explicado en la sección III, se le aplica el algortimo Ford-Fulkerson con el objetivo de

III. Complejidad Temporal

La función principal es CORRUPTION_STRATEGY(n, m, a, w). La función tiene como parámetros: n, el número de ciudades, m, el número de carreteras, a, lista con el aporte de cada ciudad, w, lista con el costo de cada carretera. El pseudocódigo siguiente permite un acercamiento a la implementación realizada.

1: **function** CORRUPTION_STRATEGY((n, m, a, w))
2: G = BUILD_GRAPH(n, m, a, w)
3: G_r = GET_MIN_CUT_RESIDUAL_GRAPH(G)
4: *cities*, *roads*, *profit* = GET_PROJECT_DISTRO(G_r, G)
return *cities, roads, profit*
5: **end function**

Para el trabajo con grafos por razones de comodidad se utilizó la librería networkx de Python. En el cálculo de la complejidad temporal no se tiene en cuenta el costo de las operaciones sobre grafos creados con la librería anterior.

En la línea 2 del método anterior, se llama a la función BUILD_GRAPH(n, m, a, w), la cual se encarga de construir el grafo G' explicado en la sección III. La complejidad temporal de dicha función es $O(|V'| + |E'|)$, ya que por cada carretera se contruye una arista entre el nodo fuente y el nodo que representa a dicha carretera, por cada par de ciudades que conforman una carretera, se crean dos arcos, cada una desde la carretera hasta las respectivas ciudades y también, por cada nodo que representa a una ciudad, se contruye una arista con el nodo receptor.

Una vez creado el grafo, se llama a la función GET_MIN_CUT_RESIDUAL_GRAPH(G), que tiene como parámetro el grafo G'. El pseudocódigo del método se muestra a continuación.

La función consiste en aplicar el algortimo Ford-Fulkerson al grafo G', por tanto tiene complejidad temporal $O(|E'||f*|)$, donde $f*$ es el flujo máximo calculado por el algoritmo. Primero, se construye el grafo residual de G', G'_r (línea 2), el cual se explica en la subsección II.2, y se

```
1:  function GET_MIN_CUT_RESIDUAL_GRAPH(G)
2:      G_r = GET_RESIDUAL_GRAPH(G)
3:      s = 'source'
4:      t = 'sink'
5:      p = FIND_AUGMENTING_PATH(G_r, s, t)
6:      while p != None do
7:          capacity_p = RESIDUAL_CAPACITY_PATH(G_r, p)
8:          for (u, v) in p do
9:              try
10:                 G[u][v]['flow'] = G[u][v]['flow'] + capacity_p
11:             catch
12:                 G[v][u]['flow'] = G[v][u]['flow'] - capacity_p
13:             end try
14:             G_r = GET_RESIDUAL_GRAPH(G)
15:             p = FIND_AUGMENTING_PATH(G_r, s, t)
16:         end for
17:     end while
          return G_r
18: end function
```

encuentra un camino aumentativo p en dicho grafo (línea 5). El método GET_RESIDUAL_GRAPH(G), tiene una complejidad temporal $O(|\mathrm{E}'|)$, ya que recorre todas los arcos de G' y construye G'_r con los mismos nodos de V', pero con los arcos de E' cuya capacidad es mayor que su flujo. La función FIND_AUGMENTING_PATH(G_r, s, t) realiza un DFS por tanto tiene complejida temporal $O(|V'| + |E_r|)$, donde E_r son el conjunto de arcos del grafo residual G'_r. Luego, se continua actualizando el grafo residual cada vez que se encuentra un camino aumentativo. El ciclo termina cuando no exista camino aumentativo en la red residual. Este método retorna el grafo residual resultante al hallar el flujo máximo.

Después de aplicar el algoritmo de flujo a G', se obtiene el último G'_r y a este grafo se le aplica el método GET_PROJECT_DISTRO(G_r, G) que halla el corte mínimo (S, T) y determina las ciudades, carreteras y el valor de la ganancia que es la solución del problema, a partir de los nodos que conforman la partición S. Este método tiene complejidad temporal $O(|S|)$.

En conclusión, la función CORRUPTION_STRATEGY(n, m, a, w) tiene complejidad temporal,

$$\begin{aligned} O(|V'| + |E'| + |E'||f*| + |S|) &= \text{máx}\{|V'|, |E'|, |E'||f*|, |S|\} \\ &= O(|E'||f*|) \end{aligned} \tag{12}$$

ya que $|V'| < |E'|$, por la forma en que está construído G' y $|S| < |V'|$, debido a que al menos el nodo receptor t no pertenece a S.

IV. Complejidad espacial

La complejidad espacial de nuestra implementación es $O(2n + 4m)$. El algoritmo utiliza como entrada una lista de tamaño n y otra de tamaño m como se explicó en la definición del problema. Además, se crea un grafo dirigido con la librería networkx de Python que tiene $2 + m + n$ vértices y $n + 3m$ aristas, cuyo costo asumimos no es mayor que $O(2n + 4m)$.

V. Generador de casos de prueba

En *src/app/generator.py* fue implementado un generador, el cual recibe una cantidad *s* de muestras a producir, genera valores random con el formato de entrada de los algoritmos implementados, halla la solución óptima con *backtrack* y las guarda en *json/test_cases.json*. Se generaron 3000 casos de prueba.

VI. Tester

En *src/app/tester.py* fue implementado un tester, que recibe una función y prueba el desempeño de la misma en cuanto a si obtuvo la solución óptima o no, y el tiempo que demoró en hacerlo, comparando con los casos de prueba obtenidos con el generador. Dichos resultados se muestran en consola de la siguiente forma:

VII. Comparación de soluciones implementadas

Referencias

[1] Cormen, Thomas H. y otros. *Introduction to Algorithms*. The MIT Press. 4ta Edición. Cambridge, Massachusetts. 2022.