

# 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

23 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. El objetivo es maximizar la diferencia de la suma de los costos de construir las carreteras seleccionadas y la suma del aporte de cada una de las ciudades que intervienen en esas carreteras. La salida del problema es la distribución del proyecto como dos listas, una que contiene las ciudades y otra las carreteras a construir tal que se obtenga la ganancia máxima.

entre ciuda-  
des

#### I. Problema "Proyectos e Instrumentos"

Uno de los problemas de flujo más conocidos es el de "Proyectos e Instrumentos". En este, se tienen un conjunto de proyectos que podemos hacer, cada uno con su costo, y un conjunto de instrumentos, cada uno con algún costo también. Cada proyecto depende de algunos instrumentos, y cada instrumento se puede utilizar cualquier número de veces. Se debe elegir un subconjunto de proyectos y un subconjunto de instrumentos tal que, si se elige un proyecto, también se elijan todos los instrumentos de los que depende este proyecto, y maximizar la diferencia entre la suma de los costos de los proyectos elegidos y la suma de los costes de los instrumentos elegidos.

Como puede observarse, este es precisamente el problema que nos ocupa, siendo los proyectos, las carreteras y los instrumentos, las ciudades. Este se puede modelar mediante una red de flujo contruyendo un grafo dirigido  $G = (V, E)$  como el que se describe a continuación:

- El conjunto  $V$  está formado por  $n + m + 2$  vértices que representan cada ciudad y cada carretera, además de un vértice  $s$  que representa la fuente y un vértice  $t$  que representa el receptor.
- El conjunto  $E$  está formado por  $n + 3 \cdot m$  arcos, distribuidos de la siguiente forma:
  - Arcos desde  $s$  hasta cada vértice que representa una carretera con capacidad igual al valor  $w_i$  dado de entrada, que representa el costo de construir la carretera  $i$ .
  - Arcos desde cada vértice que representa una ciudad hacia el vértice  $t$  con capacidad igual al valor  $a_i$  que representa el aporte de la ciudad  $i$  al ser incluida en el proyecto.
  - Arcos desde cada vértice que representa una carretera hacia cada vértice ciudad que la conforma (extremos) con capacidad igual a  $\infty$ .

### IV. SOLUCIONES IMPLEMENTADAS

#### I. Backtrack

Como primera solución al problema fue implementado un *backtrack*. Esta es una solución correcta, ya que prueba todas las combinaciones de posibles carreteras y se queda con la que más ganancia aporte, pero muy ineficiente  $O(2^m)$ . 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_solution.py`.

#### II. Solución utilizando flujo

##### ii.1. Propuesta de solución

Se aplica un algoritmo de flujo sobre  $G$  y se toma el corte mínimo  $(S, T)$ , construyendo la respuesta basada en este corte de la siguiente manera:

- si un proyecto-vértice pertenece a  $S$ , entonces tomamos este proyecto;
- si un vértice-instrumento pertenece a  $S$ , entonces tomamos este instrumento; todos los demás proyectos e instrumentos se descartan.

el valor del recorte es igual al costo total de los instrumentos tomados y los proyectos descartados, y debemos minimizarlo. Entonces, el corte mínimo en esta red denota la mejor respuesta.

## ii.2. 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:    $G = \text{BUILD\_GRAPH}(n, m, a, w)$ 
3:    $G_r = \text{GET\_MIN\_CUT\_RESIDUAL\_GRAPH}(G)$ 
4:    $\text{cities, roads, profit} = \text{GET\_PROJECT\_DISTRO}(G_r, G)$ 
   return  $\text{cities, roads, profit}$ 
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 algoritmo 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 ??, y se encuentra un camino aumentativo  $p$  en dicho grafo (línea 5). El método `GET_RESIDUAL_GRAPH( $G$ )`, tiene una complejidad temporal  $O(|E|)$ , 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 complejidad 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,

$$\begin{aligned}
 O(|V| + |E| + |E||f * | + |S|) &= \max\{|V|, |E|, |E||f * |, |S|\} \\
 &= O(|E||f * |)
 \end{aligned}
 \tag{1}$$

---

```
1: function GET_MIN_CUT_RESIDUAL_GRAPH(G)
2:   Gr = GET_RESIDUAL_GRAPH(G)
3:   s = 'source'
4:   t = 'sink'
5:   p = FIND_AUGMENTING_PATH(Gr, s, t)
6:   while p != None do
7:     capacity_p = RESIDUAL_CAPACITY_PATH(Gr, 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:      Gr = GET_RESIDUAL_GRAPH(G)
15:      p = FIND_AUGMENTING_PATH(Gr, s, t)
16:    end for
17:  end while
18:  return Gr
19: end function
```

---

ya que  $|V| < |E|$ , por la forma en que está construido  $G$  y  $|S| < |V|$ , debido a que al menos el nodo receptor  $t$  no pertenece a  $S$ .

### ii.3. 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

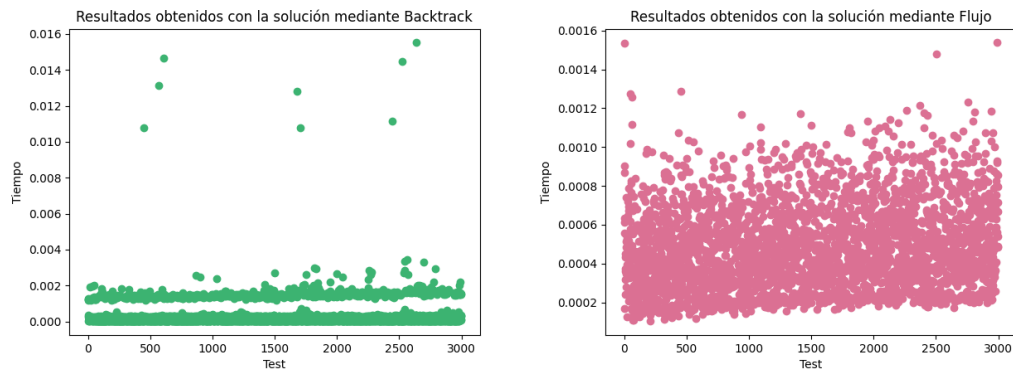
En `src/app/generator.py` fue implementado un generador, el cual recibe una cantidad  $s$  de muestras a producir, genera valores random con el formato de entrada de los algoritmos implementados, halla la solución óptima con `backtrack` y las guarda en `json/test_cases.json`. Se generaron 3000 casos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

## VI. TESTER

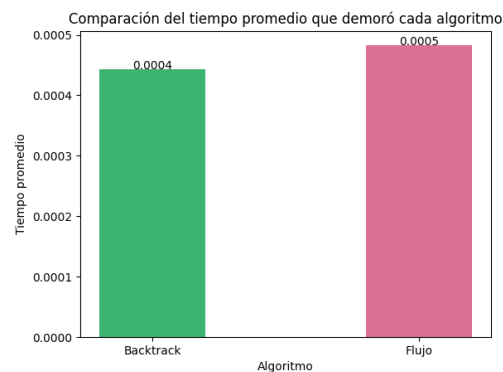
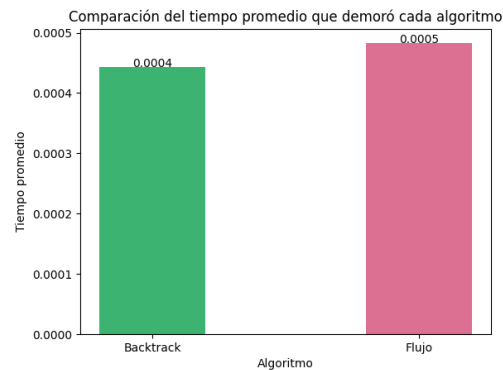
En `src/app/tester.py` fue implementado un tester, que recibe una función y prueba el desempeño de la misma en cuanto a si obtuvo la solución óptima o no, y el tiempo que demoró en hacerlo, comparando con los casos de prueba obtenidos con el generador. Además, estos resultados se guardan en un `.json` con el nombre de la función en la carpeta `tests`. La solución implementada fue testeada para todos los casos de prueba generados y puede encontrarse en `json/tests/corruption_strategy_solution.json`.

## VII. COMPARACIÓN DE SOLUCIONES IMPLEMENTADAS

A continuación se muestra una gráfica por cada algoritmo con el tiempo que demoró en cada uno de los 3000 tests. Como podemos observar, en los casos analizados, el *backtrack* puede demorar desde 0.000 hasta 0.016 segundos en ejecutarse, pero la mayoría de los casos oscilan entre 0.000 y 0.002. La *solución con flujo* para todos los casos se demora menos de 0.0016 segundos, lo cual lo convierte en una solución significativamente mejor que el *backtrack*.



Además, en la siguiente gráfica se ofrece una comparación del tiempo promedio que le toma a cada algoritmo ejecutar los casos de prueba.



chequear  
gráficas

## REFERENCIAS

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