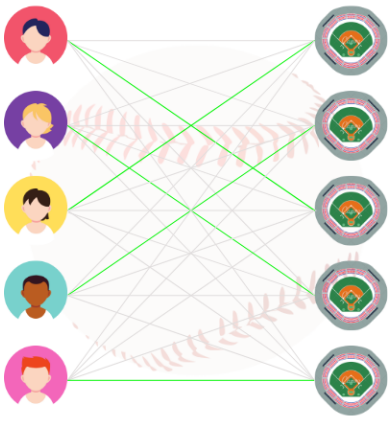
Diseño y Análisis de Algoritmos

Proyecto # 1: La Pelota



Laura Victoria Riera Pérez
Marié del Valle Reyes

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

9 de abril de 2023

I. Repositorio del proyecto

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

II. Definición inicial del problema

Para un campeonato de pelota, el manager debe elegir de un conjunto de n personas, a su equipo de p jugadores, y a k espectadores especiales para que suban la moral del equipo. De cada persona i, el manager conoce el valor que aporta a la moral del equipo a_i y el valor que aporta siendo situado en la posición j, $s_{i,j}$. Determine una alineación entre jugadores en el campo y espectadores de forma que el equipo tenga la mayor cantidad de valor acumulado posible.

III. Definición en términos matemático - computacionales

1. Preliminares

Definición 1. *Sea $G = (V, E)$ un grafo. Se dice que G es bipartito si $V(G)$ es la unión de dos conjuntos independientes disjuntos. Si entre todo par de nodos de diferentes particiones existe una arista, se dice que es un grafo bipartito completo.*

Definición 2. *Un emparejamiento M es un conjunto de aristas en un grafo que son independientes, o sea, que no comparten vértices.*

Definición 3. *Dado un emparejamiento M:*

- *Si la arista $e = (v, w) \in M$, se dice que v y w son saturados por M.*
- *Un conjunto de vértices es saturado por M cuando M satura a todos los vértices del conjunto.*
- *Se dice que M es perfecto si satura a $V(G)$.*
- *M es máximo cuando no existe M_1 tal que $|M_1| > |M|$.*

Definición 4. *Sea G un grafo y M un emparejamiento del mismo. Un camino simple en G es M-alternativo si sus aristas alternan entre pertenecer y no pertenecer a M.*

Definición 5. *Sea G un grafo y M un emparejamiento del mismo. Un camino simple en G es M-aumentativo si es M-alternativo y sus extremos no son saturados por M.*

Teorema 1. *Sea G un grafo y M un emparejamiento del mismo. M es maximal si y solo si no existen caminos M-aumentativos.*

II. Problema de asignación

La entrada de nuestro problema es el número de personas n, el número de posiciones a ser asignadas $m = p + k$, donde p es la cantidad de posiciones dentro del equipo y k el número de asientos en el estadio a ser ocupados. Además, se recibe una lista a de valor aportado de cada persona por ser parte de la asignación, y una matriz s del valor aportado por cada persona en cada una de las posiciones. La salida del algoritmo debe ser la asignación de personas a posiciones que maximice el valor del equipo.

Dado un grafo bipartito completo ponderado $G = (V, E)$, donde $V = L \cup R$. Se asume que los vértices de los conjuntos L y R contienen n vértices cada uno, por tanto el grafo contiene n^2 aristas. Para todo $l \in L$ y $r \in R$, se denota el peso de la arista (l, r) como $w(l, r)$, lo cual representa ganancia de emparejar el vértice l con el vértice r. A encontrar un emparejamiento perfecto M^* cuyas aristas tengan el peso máximo total de todos los emparejamientos perfectos posibles se le llama *problema de asignación*. Sea $w(M) = \sum_{(l,r)\in M} w(l, r)$ el peso total de las aristas en el emparejamiento M, se quiere encontrar el emparejamiento perfecto M^* tal que,

$$w(M*) = \max\{w(M) : M \text{ es un emparejamiento perfecto}\}.$$

Así, nuestro problema puede ser modelado como un grafo bipartito completo ponderado $G = (L \cup R, E)$ donde L representan las personas y R las posiciones en el estadio a ser asignadas; y donde el peso de cada arista $e = (u, v)$ es $w(e) = a[u] + s[u][v]$.

IV. Línea de pensamiento

Como primera solución al problema de asignación 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, $O(n!)$. En una computadora de 32GB de RAM, intel core i7-11na generación, se puede resolver para una cantidad máxima de personas y/o posiciones de 11. Dicha solución puede ser encontrada en *src/solutions/backtrack_solutions.py*.

Se intentó resolver mediante un algoritmo *greedy*, tomando por cada persona el máximo del valor que puede aportar al equipo, y luego ir asignando de forma descendente a las posiciones. El

problema de este enfoque está en que cuando dos o más personas aportan el mismo valor en una misma posición se debe escoger aquel cuyo siguiente máximo sea menor (pues de lo contrario en una próxima iteración se perdería en valor en otra posición). Nótese que para el peor de los casos, cuando todos las personas aporten el mismo valor en todas las posiciones, esta solución es tan mala como backtrack.

El próximo pensamiento fue resolverlo con *programación dinámica*. Para poder aplicar esta solución el problema debe cumplir con dos características fundamentales: suproblemas solapados, es decir, que subproblemas iguales se repitan una y otra vez, por lo que se necesitaría volver a calcular sus soluciones; y subestructura óptima, lo cual significa que una solución óptima para un subproblema forma parte de la solución óptima del problema. Luego de graficar el backtrack y aplicar memorización para varios casos se observó que ninguno presentaba subproblemas solapados, por lo que la intuición dice que probablemente no tenga. Además se cree que este problema no tiene subestructura óptima (al menos para las formas de pensar y picar el mismo analizadas) dado que asignar las personas que más valor den a un subconjunto de las posiciones no garantiza un óptimo, puede ocurrir que esas personas aportaran más en otras posiciones fuera del subconjunto.

Este problema puede ser modelado también como un problema de optimización lineal. Sea $x_{i,j}$ una variable binaria que indica que se asignó la persona i a la posición j. Se quiere maximizar la suma de $w(i,j) = a[i] + s[i,j] \forall i \in n, \forall j \in m$. Las restricciones añadidas garantizan que todos los vértices tengan exactamente un vecino, y por tanto la solución hallada sea un emparejamiento perfecto.

$$\begin{cases} \text{máx} & z = \sum_i \sum_j x_{i,j} \cdot (s_{i,j} + a_i) & \\ \text{s.a.} & \sum_{j=1}^{m} x_{i,j} = 1 & \forall i \in n \\ & \sum_{i=1}^{n} x_{i,j} = 1 & \forall j \in m \end{cases} \tag{1}$$

Esta solución fue probada utilizando el algoritmo *Simplex* implementado en la librería scipy y puede ser encontrada en *src/solutions/simplex_solution.py*. La complejidad temporal del simplex para el caso peor es O(2^n), pero en general es bastante rápido.

Investigando el estado del arte y siguiendo la idea de modelar el problems mediante grafos se decidió implementar para su solución el algoritmo Hungarian descrito en [1], el cual es una especie de greedy y se ejecuta en tiempo polinomial, y es explicado en la próxima sección. Dicha implementación está en *src/solutions/hungarian_solution.py*.

V. Algoritmo Hungarian

El método Húngaro es un algoritmo de optimización-combinatoria que resuelve el problema de asignación en tiempo polinomial. En vez de trabajar con un grafo bipartito completo G, el algoritmo Hungarian trabaja con un subgrafo de G llamado **subgrafo de igualdad**. El subgrafo de igualdad depende de asignar un atributo h a cada vértice, llamado **etiqueta** del vértice. Se dice que h es un **etiquetado de vértice factible** de G si $l.h + r.h \geq w(l,r)$ para todo $l \in L$ y $r \in R$. Un etiquetado de vértice factible siempre existe, como el **etiquetado de vértice por defecto** dado por

$$l.h = \max\{w(l,r) : r \in R\} \qquad \text{para todo } l \in R, \tag{2}$$

$$r.h = 0 \qquad \text{para todo } r \in R \tag{3}$$

Dado un etiquetado de vértice factible h, el **subgrafo de igualdad** $G_h = (V, E_h)$ de G consiste de los mismos vértice de G y el subconjunto de aristas $E_h = \{(l,r) \in E : l.h + r.h = w(l,r)\}$.

El subgrafo de igualdad puede cambiar en el tiempo y tiene la propiedad que cualquier emparejamiento perfecto en el subgrafo de igualdad es también una solución óptima del problema de asignación como se demuestra en el siguiente teorema.

Teorema 2. *[1] Sea $G = (V, E)$, donde $V = L \cup R$, un grafo bipartito completo donde cada arista $(l, r) \in E$ tiene peso $w(l, r)$. Sea h un etiquetado de vértice factible de G y G_h el subgrafo de igualdad de G. Si G_h contiene un emparejamiento perfecto M^*, entonces M^* es una solución óptima del problema de asignación G.*

Demostración. Si G_h tiene un emparejamiento perfecto M^*, entonces debido a que G_h y G tienen el mismo conjunto de vértices, M^* es también un emparejamiento perfecto en G. Debido a que cada arista de M^* pertenece a G_h y cada vértice tiene exactamente una arista incidente del emparejamiento perfecto, entonces se tiene

$$
\begin{aligned}
w(M*) &= \sum_{(l,r)\in M*} w(l,r) && && (4)\\
&= \sum_{(l,r)\in M^*} (l.h + r.h) && \text{(porque todas las aristas de } M^* \text{ pertenecen a } G_h) && (5)\\
&= \sum_{l\in L} l.h + \sum_{r\in R} r.h && \text{(porque } M^* \text{ es un emparejamiento perfecto)} && (6)\\
& && && (7)
\end{aligned}
$$

Sea M un emparejamiento perfecto cualquiera de G, se tiene

$$
\begin{aligned}
w(M) &= \sum_{(l,r)\in M} w(l,r) && && (8)\\
&\leq \sum_{(l,r)\in M} (l.h + r.h) && \text{(porque } h \text{ es un etiquetado de vértice factible)} && (9)\\
&= \sum_{l\in L} l.h + \sum_{r\in R} r.h && \text{(porque } M \text{ es un emparejamiento perfecto)} && (10)
\end{aligned}
$$

Entonces se tiene

$$w(M) \leq \sum_{l\in L} l.h + \sum_{r\in R} r.h = w(M^*), \tag{11}$$

por tanto M^* es un emparejamiento perfecto de máximo costo en G. ■

Entonces, el objetivo del algoritmo es encontrar un emparejamiento perfecto en un subgrafo de igualdad.

1. Explicación del algoritmo

El algoritmo Hungarian empieza con un etiquetado de vértices factible h que es el etiquetado por defecto [2] y cualquier emparejamiento M en el subgrafo de igualdad G_h. En la resolución de este problema se utilizó un algoritmo de emparejamiento maximal greedy, el cual consiste en escoger aristas que conformen un emparejamiento de tal forma que la suma total de sus pesos sea máxima. Luego, el algoritmo repetidamente encuentra un **camino M-aumentativo** P en G_h utilizando una variante de Búsqueda Primero a lo Ancho (*en inglés*, **BFS**).

El algoritmo BFS empieza la búsqueda desde todos los vértices no saturados de L, los cuales al inicio se insertan en la cola Q. La condición de parada es que se descubra algún vértice no

saturado de R, ya que un camino M-aumentativo es aquel que empieza en un vértice no saturado de L y termina en un vértice no saturado de R, tomando aristas no saturadas de L a R y aristas saturadas de R a L. El resultado del algoritmo es un bosque primero a lo ancho $F = (V_f, E_f)$, donde cada vértice no saturado de L es raíz de algún árbol de F.

Una vez encontrado un camino M-aumentativo, se actualiza el emparejamiento para que este sea la diferencia simétrica de M y P, incrementando así el tamaño del emparejamiento por lema [1]. La **diferencia simétrica** de dos conjuntos A y B, se define como $A\Delta B = (A \cup B) - (A \cap B)$. Mientras haya algún subgrafo de igualdad que contenga un camino M- aumentativo, el tamaño del emparejamiento puede incrementar, hasta que un emparejamiento perfecto se logre.

Lema 1. *[1] Sea M un emparejamiento en un grafo no dirigido $G = (V, E)$, y sea P un camino M-aumentativo. Entonces el conjunto de aristas $M' = M\Delta P$ es también un emparejamiento en G con $|M'| = |M| + 1$.*

Demostración. Sea m la cantidad de aristas que contiene P, tal que $\lceil m/2 \rceil$ aristas pertenezcan a $E - M$ y $\lfloor m/2 \rfloor$ aristas pertenezcan a M y sean las aristas $(v_1, v_2), (v_2, v_3), ..., (v_m, v_{m+1})$. Por deficición [5] los extremos v_1 y v_{m+1} de P no están saturados y todos los demás vértices si están saturados. Las aristas $(v_1, v_2), (v_3, v_4), ..., (v_m, v_{m+1})$ pertenecen a $E - M$, y las aristas $(v_2, v_3), (v_4, v_5), ..., (v_{m-1}, v_m)$ pertenecen a M. La diferencia simétrica $M' = M\Delta P$ intercambia los conjunto de aristas M y $E - M$, quedando en M' las aristas $(v_1, v_2), (v_3, v_4), ..., (v_m, v_{m+1})$. Cada vértice $v_1, v_2, ..., v_m, v_{m+1}$ están saturadas en M', teniendo M' una arista más en relación con M. ■

La búsqueda falla cuando la cola Q se vacía sin que se halla llegado a encontrar un vértice no saturado de R que conforme un camino M-aumentativo. Cuando esto ocurre, el algoritmo Hungarian actualiza el etiquetado de vértices factible h de acuerdo al siguiente lema, para adicionar a G_h al menos una arista nueva.

Lema 2. *[1] Sea h un etiquetado de vértice factible en el grafo bipartito completo G con el subgrafo de igualdad G_h, y sea M un emparejamiento para G_h y F el bosque construído a partir de un BFS sobre el subgrafo de igualdad G_h. Entonces, la etiqueta h',*

$$v.h' = \begin{cases} v.h - \delta & si\ v \in F_l, \\ v.h + \delta & si\ v \in F_r, \\ v.h & e.o.c \end{cases} \tag{12}$$

donde

$$\delta = min\{l.h + r.l - w(l, r) : l \in F_l, r \in R - F_r\} \tag{13}$$

con $F_l = L \cap V_f$ y $F_r = R \cap V_f$ son vértices del bosque F que pertenecen a L y a R, respectivamente, es una etiqueta de vértice factible para G con las siguientes propiedades:

1. *Si (u,v) es una arista de bosque F para G_h, entonces (u,v) $\in E_{h'}$.*
2. *Si (l,r) pertenece al emparejamiento M para G_h, entonces (l,r) $\in E_{h'}$.*
3. *Existen vértices $l \in F_l$ y $r \in R - F_r$ tales que (l,r) $\notin E_h$, pero (l,r) $\in E_{h'}$.*

Demostración. Primero se demuestra que h' es un etiquetado de vértices factible para G. Debido a que h es un etiquetado de vértice factible, se tiene $l.h + r.h \geq w(l, r)$ para todo $l \in L$ y $r \in R$. Para que h' no sea un etiquetado de vértice factible, se necesitaría que $l.h' + r.h' < l.h + r.h$ para algún $l \in L$ y $r \in R$. La única forma en que esto pudiera ocurrir sería para vértices $l \in F_l$ y $r \in R - F_r$. En esta instancia, la cantidad de decrecimiento es igual a δ, entonces $l.h' + r.h' = l.h - \delta + r.h$.

Por ecuación [13], se tiene que $l.h - \delta + r.h \geq w(l,r)$ para cualquier $l \in F_l$ y $r \in R - F_r$, por tanto $l.h' + r.h' \geq w(l,r)$. Para cualquier otra arista, se tiene $l.h' + r.h' \geq l.h + r.h \geq w(l,r)$. Por tanto, h' es un etiquetado de vértice factible.

A continuación se demostrará la veracidad de las propiedades:

1. Si $l \in F_l$ y $r \in F_r$, entonces se tiene $l.h' + r.h' = l.h + r.h$ debido a que δ se adiciona a la etiqueta de l y se substrae de la etiqueta de r. Entonces, si una arista pertenece a F para el grafo G_h, también pertenece a $G_{h'}$.

2. Se afirma que para el momento en que el algoritmo Hungarian computa el nuevo etiquetado de vértice factible h', para toda arista $(l,r) \in M$, se tiene que $l \in F_l$ si y solo si $r \in F_r$.

   Para demostrar por qué, se considera el vértice saturado r y la arista $(l,r) \in M$. Primero se supone que $r \in F_r$, entonces la búsqueda encuentra r y lo pone en la cola. Cuando r se quita de la cola, l es descubierto, entonces $l \in F_l$.

   Luego se supone que $r \notin F_r$, por tanto r no se ha descubierto. Se demostrará que $l \notin F_l$. La única arista en G_h que entra el vértice l es (r,l), y dado que r no se ha descubierto, la búsqueda no ha tomado esta arista; si $l \in F_l$, no es por la arista (r,l). La única otra forma que un vértice en L puede estar en F_l es si es raíz de la búsqueda, pero solo vértices no saturados de L son raíces y l está saturado. Por tanto, $l \notin F_l$ y la afirmación se cumple.

   Se conoce que para $l \in F_l$ y $r \in F_r$ se cumple $l.h' + r.h' = l.h + r.h$. En caso contrario, cuando $l \in L - F_l$ y $r \in R - F_r$, se tiene que $l.h' = l.h$ y $r.h' = r.h$, entonces $l.h' + r.h' = l.h + r.h$. Por tanto, si la arista (l,r) está en el emparejamiento M para el grafo G_h, entonces $(l,r) \in E_{h'}$.

3. Sea (l,r) una arista que no pertenece a E_h, tal que $l \in F_l$, $r \in R - F_r$ y $\delta = l.h + r.h - w(l,r)$. Entonces, por definición de δ, existe al menos una de esas arista. Luego, se tiene

$$\begin{aligned} l.h' + r.h' &= l.h - \delta + r.h \\ &= l.h - (l.h + r.h - w(l,r)) + r.h \\ &= w(l,r) \end{aligned}$$

   y por tanto $(l,r) \in E_{h'}$.

■

II. Complejidad Temporal

La complejidad temporal del algoritmo Hungarian implementado es $O(n^4)$, donde $|V| = 2n$ y $|E| = n^2$ en el grafo original G. El pseudocódigo siguiente permite un acercamiento a la implementación de dicho algoritmo.

```
1: function HUNGARIAN_SOLUTION(n, m, a, s)
2:     G = BUILD_GRAPH(n, m, a, s)
3:     G_h = BUILD_EQUALITY_SUBGRAPH(G)
4:     M = INITIAL_GREEDY_BIPARTITE_MATCHING(G_h)
5:     while en G_h no se encuentre emparejamiento perfecto do
6:         P = FINDING_AUGMENTING_PATH(G_h)
7:         SYMMETRIC_DIFFERENCE(P,G_h)
```

8: **end while**
9: **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.

Las líneas 2-4 se encargan de crear las condiciones necesarias para hallar un emparejamiento perfecto M^* en G. En la línea 2, el método BUILD_GRAPH(n,m,s,a) construye un grafo bipartito completo $G = (V, E)$, con $V = L \cup R$, a partir de los datos de entrada del problema. Si $m < n$, se adicionan vértices y aristas artificiales a G para que pueda quedar bipartito completo. En este método además de insertar n vértices en un grafo de la librería networkx, se insertan n^2 aristas con sus respectivos pesos, por tanto el método tiene un costo de $O(n^2)$.

El método BUILD_EQUALITY_SUBGRAPH(G), construye el subgrafo de igualdad G_h a partir de G y el método INITIAL_GREEDY_BIPARTITE_MATCHING(G_h) halla un emparejamiento M en G_h como se explicó en la sección anterior. Ambos métodos tienen una complejidad temporal de $O(n^2)$, ya que hay que recorrer todas las aristas del grafo G.

El ciclo de las líneas 5-8 termina cuando se encuentra un emparejamiento perfecto M de G_h. Por definición [3], M es perfecto si satura a V y $|M| = \frac{|V|}{2} = n$. En el método FINDING_AUGMENTING_PATH(G_h) se halla un camino M-aumentativo P en G_h utilizando BFS, con el cual incrementa en 1 la cardinalidad de M. Por tanto, el ciclo itera a lo sumo n veces. El método SYMMETRIC_DIFFERENCE(P,G_h) actualiza el emparejamiento de G_h a partir del nuevo camino M-aumentativo encontrado. Para ello se utiliza la diferencia simétrica de P y el emparejamiento existente en G_h. En dicho método se comparan y actualizan las aristas de ambos grafos según las siguiente condiciones:

- Si existe una arista en P que no está saturada en G_h, entonces esa arista se satura en G_h.
- Si existe una arista saturada en P y en G_h a la vez, entonces esa arista deja de estar saturada en G_h.

A continuación se observa el pseudocódigo del método FINDING_AUGMENTING_PATH(G_h).

1: **function** FINDING_AUGMENTING_PATH(G_h)
2: $Q = \emptyset$
3: $F_l = \emptyset$
4: $F_r = \emptyset$
5: $d = \{\}$
6: **for** l no saturado en L **do**
7: $d[l] =$ NIL
8: ENQUEUE(l,Q)
9: $F_l = F_l \cup \{l\}$
10: **end for**
11: **repeat**
12: **if** Q está vacía **then**
13: $\delta = min\{l.h + r.l - w(l,r) : l \in F_l, r \in R - F_r\}$
14: **for** cada $l \in F_l$ **do**
15: $l.h = l.h - \delta$
16: **end for**
17: **for** cada $l \in F_l$ **do**
18: $r.h = r.h - \delta$
19: **end for**

```
20:             MODIFY_EQUALITY_SUBGRAPH(G_l, G)
21:             for cada arista (l, r) nueva en G_l do
22:                 if r ∉ F_r then
23:                     d[r] = l
24:                     if r no saturado then
25:                         break
26:                     else
27:                         ENQUEUE(Q,r)
28:                         F_r = F_r ∪ {r}
29:                     end if
30:                 end if
31:             end for
32:         end if
33:         u = DEQUEUE(Q)
34:         if u ∈ R then
35:             for cada v ∈ G_l.neighbors(u) do
36:                 if (u, v) está saturada then
37:                     d[v] = u
38:                     F_l = F_l ∪ {v}
39:                     ENQUEUE(Q)
40:                 end if
41:             end for
42:         else
43:             for cada v ∈ G_l.neighbors(u) do
44:                 if (u, v) no está saturada and v ∉ F_r then
45:                     d[v] = u
46:                     if v no está saturado then
47:                         break
48:                     else
49:                         F_r = F_r ∪ {v}
50:                         ENQUEUE(Q)
51:                     end if
52:                 end if
53:             end for
54:         end if
55:     until se halle camino M-aumentativo
56:     P = CONSTRUCT_AUGMENTING_PATH(d)
57:     return P
58: end function
```

Las línea 2-5 inicializan las estructuras necesarias en la búsqueda en $O(1)$ cada línea. El diccionario d almacena el predecesor del vértice v en $d[v]$. El **for** de la línea 6 itera $|L| = n$ veces. En el cuerpo del mismo se insertan los vértices de L no saturados por M que se encuentran en la cola Q y en el conjunto F_l y se asigna NONE a $d[l]$, ya que dichos vértices serán raíces del bosque F que conforma la búsqueda.

Omitiendo las líneas 12-32, se puede apreciar que el método es un BFS. La complejidad temporal del BFS es $O(|V| + |E|) = O(n^2)$. Las líneas 12-32 se ejecutan cuando se vacía Q y no se encontró un vértice no saturado de R. En este caso, se actualiza el etiquetado de vértices factible

de los vértices de G_h y así, por el lema [13] se adiciona al menos una arista nueva a G_h. Como la arista nueva es de la forma, (l, r) con $l \in F_l$ y $r \in R - F_r$, se garantiza que se descubra un vértice de R. Por tanto, esa secuencia de líneas de código se ejecutan a lo sumo $|R| = n$.

Hallar el valor de δ en la línea 13 tiene costo $O(n^2)$ y el método MODIFY_EQUALITY_SUBGRAPH(G_h, G), que actualiza G_h dado el nuevo etiquetado h', también tiene costo $O(n^2)$.

Como las líneas 12-32 se pueden ejecutar hasta n veces y, en cada ejecución se modifica G_h y se realiza un BFS sobre el nuevo G_h con costo $O(n^2)$, se concluye que la ejecución de las líneas 11-55 tiene un costo de $O(n^3)$.

El método CONSTRUCT_AUGMENTING_PATH(d) de la línea 56 construye un grafo P a partir del camino M-aumentativo registrado en d. Este método tiene costo $O(n)$.

Por lo anteriormente explicado, se llega a que el método FINDING_AUGMENTING_PATH(G_h) tiene costo $O(n^3)$, y como este se encuentra dentro de un ciclo que itera a lo sumo n veces en el método HUNGARIAN_SOLUTION(n, m, a, s), entonces este último método tiene costo $O(n^4)$.

III. Complejidad espacial

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

VI. 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, con n máximo igual a 11, dado que, como se mencionó anteriormente, es lo que puede ejecutar el *backtrack*.

VII. 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:

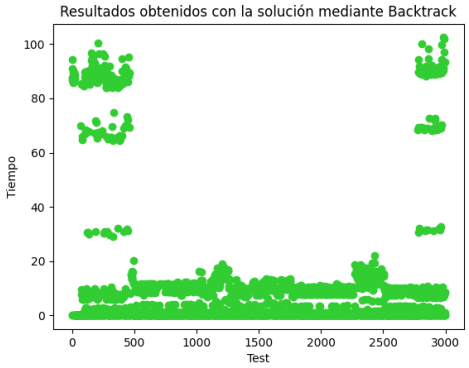
```
-----------------------------------------
Test case #2999 -> SUCCESS
  result: [2, 0, 4, 3, 1]
  value: 70
  elapsed time: 3.814697265625e-06
-----------------------------------------
Test case #3000 -> FAILED
  result: [2, 0, 4, 3, 1]
  value: 70
   optimal value: 116
  elapsed time: 5.7220458984375e-06
-----------------------------------------
```

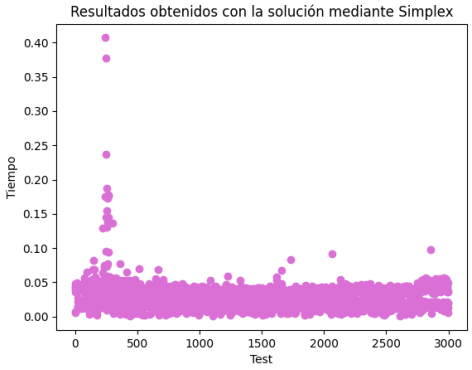
Figura 1. Ejemplo de cómo se muestran los tests de una función en consola.

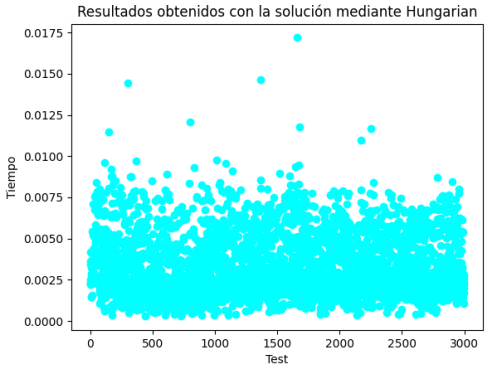
Además, estos resultados se guardan en un *.json* con el nombre de la función en la carpeta tests. Las soluciones implementadas fueron testeadas para todos los casos de prueba generados y pueden encontrarse en *json/tests/simplex_solution.json* y *json/tests/hungarian_solution.json*

VIII. Comparación de soluciones implementadas

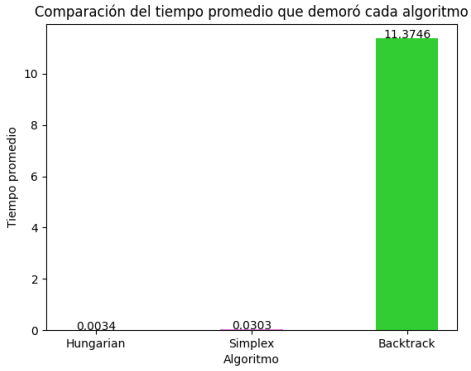
En los casos analizados, el *backtrack* puede demorar desde menos de un segundo hasta casi dos minutos en ejecutarse. El *simplex* demora para todos los casos menos de medio segundo, manteniéndose para la mayoría por debajo de los 0.2 segundos. Por último el *hungarian* implementado es el que mejores resultados muestra, tomando menos de 0.02 segundos para todos los casos, la décima parte de lo que demora el *simplex*. A continuación se muestra una gráfica por cada algoritmo con el tiempo que demoró en cada uno de los 3000 tests.







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



Referencias

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