DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Técnicas Algoritmicas

Algoritmos y Estructuras de Datos III
1er cuatrimestre 2022

Integrante	LU	Correo electrónico
Agustin Polleschi	[redacted]	[redacted]
Constanza García	[redacted]	[redacted]
Mariano Oca	[redacted]	[redacted]
Lucas Rango	[redacted]	[redacted]
Agustin Gianolini	[redacted]	[redacted]

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Índice

1. Introducción

En este trabajo, vamos a resolver 4 problemas con distintas técnicas algorítmicas: **Backtracking (BT)**, **Programación Dinámica (PD)** y algoritmos **Golosos**.

En cada sección presentamos el problema a resolver, proponemos un algoritmo que lo resuelve, analizamos su complejidad y, luego de implementerlo, lo comparamos con otros resultados.

2. Ejercicio 1: Backtracking

2.1. Descripción del problema

Definimos una **red social** como una tripla $G = (V, E, p)$, donde:

- $V = \{1, ..., n\}$ es el conjunto de actores que forman parte de G
- E es el conjunto de *amistades* de G, es decir, dados $v, w \in V$ (con $v \neq w$) decimos que son amigos si y sólo si $\{v, w\} \in E$
- $p : V \longrightarrow \mathbb{N}_+$ es una función que da un *nivel de influencia* para cada actor de V

Definimos una **clique** como un subconjunto de V tal que todos sus actores son amigos entre sí. Formalmente, $Q \subset V$ es una clique si para todo $v, w \in Q$, $\{v, w\} \in E$. El **poder de influencia** de una clique Q queda definido como $p(Q) = \sum_{v \in Q} p(v)$.

El objetivo del problema es hallar la clique de influencia máxima de G.

2.2. El algoritmo

La idea del algoritmo de backtracking para resolver el problema es ir considerando los actores uno a uno, en algún orden, y para cada uno evaluar dos opciones: agregarlo a la clique o no. Pero, para que nuestras soluciones sean en efecto cliques, necesitamos restricciones sobre los actores que consideramos al momento de agregarlos a nuestra solución parcial. Entonces, nuestras soluciones parciales se componen de la clique formada hasta el momento, y los actores que podríamos agregar a la misma.

Si llamamos (Q, K) a una solución parcial, entonces esta cumple:

- $Q \subset V$ es una clique
- K tiene los actores sobre los que aún no tomamos una decisión
- cada actor de K es amigo de todos los actores de Q

Manteniendo esta estructura, aseguramos que cuando K esté vacío podemos terminar la recursión, y tendremos en Q una clique.

Sin perder de foco el objetivo del problema, que es encontrar la clique de influencia máxima, podemos observar que sólo nos interesa considerar aquellas que sean *lo más grandes posibles*. Como la influencia de un actor es un entero positivo, es claro que cuánto más grande sea la clique, mayor será su influencia. De esta forma, podemos poner una restricción más sobre K:

- los actores de K no son todos amigos entre sí

Esto es porque, si lo fueran, entonces K sería una clique, y en consecuencia $Q \cup K$ también. Por lo explicado antes, llegado ese caso no tiene sentido continuar con la recursión, y podemos retornar la influencia de $Q \cup K$, ya que será la clique de mayor influencia que contenga a Q.

Manteniendo todo lo anterior, vamos avanzando sobre los actores de K, hasta que el mismo esté vacío o sus actores sean amigos entre sí. En ese momento, si Q es la primer clique que consideramos, devolvemos su influencia, si no, devolvemos la mayor entre la previamente calculada y la de Q.

Podas

El algoritmo descrito hasta ahora funciona, pero se puede mejorar. Implementamos una poda basada en la siguiente idea: si la influencia de Q hasta el momento más la influencia de todos los actores de K no supera la influencia máxima encontrada, no vale la pena continuar la recursión porque seguro que la influencia de esa solución no superará a la máxima ya encontrada.

Ordenamiento

Notemos que el orden en el que consideramos los actores de K puede influir en el tiempo de ejecución. No es lo mismo si la primer clique encontrada es la de influencia máxima, que si las vamos encontrando con influencia creciente. Esto se debe a la poda de nuestro algoritmo: cuanto antes encontremos la influencia máxima (o una influencia lo suficientemente alta) más veces se ejecutará la misma. Vamos a considerar dos órdenes posibles: de menor a mayor influencia y de mayor a menor. En la sección 2.3 analizamos los tiempos de ejecución para ambos.

Complejidad

Veamos el pseudocódigo del algoritmo para analizar su complejidad.

Algorithm 1

```
1:  infMax = 0
2:  Qmax = {}
3:  function influenciaMaxima(Q, K, infQ, infK)
4:      if infQ + infK ≤ infMax then return
5:      if sonTodosAmigos(K) then
6:          Q = Q ∪ K
7:          infQ = infQ + infK
8:          if infQ > infMax then
9:              infMax = infQ
10:             Qmax = Q
11:         return
12:     sea v ∈ K
13:     K' = K - {v}
14:     infK' = infK - p(v)
15:     Q = Q ∪ {v}
16:     sacarNoAmigos(v, K, infK)
17:     influenciaMaxima(Q, K, infQ + p(v), infK)
18:     Q = Q - {v}
19:     influenciaMaxima(Q, K', infQ, infK')
```

La función auxiliar `sonTodosAmigos(K)` chequea si los actores de K son todos amigos entre sí. En K hay a lo sumo n actores, y para cada uno tenemos que ver que sea amigo de todo el resto, por lo que tiene complejidad $O(n^2)$.

La función `sacarNoAmigos(v, K, infK)` saca de K a todos los actores que no sean amigos de v, y modifica $inf K$ acordemente. Para esto, alcanza con consultar si cada actor de K es amigo de v, que se hace en $O(1)$. A lo sumo hay n actores en K, asi que la complejidad es $O(n)$.

El resto de las operaciones del algoritmo, son sumas, comparaciones, y agregar y quitar elementos de conjuntos, que son todas $O(1)$.

Por lo tanto la complejidad de procesar cada nodo es $O(n^2)$.

Superposición de problemas

Para que haya superposición de problemas, debemos llamar más de una vez a nuestra función con los mismos parámetros. Es evidente si pensamos en el arbol de recursión de nuestro algoritmo que en este caso no sucede.

Todo nodo (Q, K) tiene dos hijos, uno en el que agregamos un actor a Q y otro en el que no agregamos nada. Con esto se puede concluir que no hay dos nodos en el arbol que tengan el mismo Q.

2.3. Resultados

Instancia	Mayor a menor	Menor o mayor
brock200_1.clq	10.52	510.38
brock200_2.clq	0.08	0.48
brock200_3.clq	0.37	4.06
brock200_4.clq	1.0	26.07
brock400_2.clq	> 600	> 600
brock400_3.clq	> 600	> 600
brock400_4.clq	> 600	> 600
C125.9.clq	37.41	> 600
C250.9.clq	> 600	> 600
c-fat200-1.clq	0.0	0.0
c-fat200-2.clq	0.0	0.01
c-fat200-5.clq	0.01	7.53
c-fat500-10.clq	1.1	> 600
c-fat500-1.clq	0.01	0.02
c-fat500-2.clq	0.01	0.05
c-fat500-5.clq	0.02	92.15
DSJC500_5.clq	10.75	110.95
gen200_p0.9_44.clq	> 600	> 600
gen200_p0.9_55.clq	> 600	> 600
hamming6-2.clq	0.0	6.88
hamming6-4.clq	0.0	0.0
hamming8-2.clq	> 600	> 600
hamming8-4.clq	2.53	107.33
johnson16-2-4.clq	3.16	16.07
johnson8-2-4.clq	0.0	0.0
johnson8-4-4.clq	0.02	0.34
MANN_a9.clq	0.69	17.92
p_hat1000-1.clq	3.78	13.35
p_hat1500-1.clq	35.71	150.81
p_hat300-1.clq	0.02	0.07
p_hat300-2.clq	2.58	474.1
p_hat300-3.clq	> 600	> 600
p_hat500-1.clq	0.19	0.63
p_hat500-2.clq	385.35	> 600
p_hat700-1.clq	0.78	2.64
p_hat700-2.clq	> 600	> 600
san1000.clq	> 600	> 600
san200_0.9_2.clq	> 600	> 600
san200_0.9_3.clq	> 600	> 600
san400_0.7_1.clq	> 600	> 600
san400_0.7_2.clq	> 600	> 600
san400_0.7_3.clq	> 600	> 600
sanr200_0.7.clq	2.66	90.9
sanr200_0.9.clq	> 600	> 600
sanr400_0.5.clq	3.27	28.0

Como podemos ver, cuando ordenamos K según las influencias de los actores de mayor a menor, el algoritmo es más rápido para todas las instancias.

Dijimos en la sección 2.2 que para que nuestra poda se ejecute más veces, quisiéramos encontrar alguna clique con influencia alta lo antes posible. Como hacemos primero la llamada recursiva que agrega un actor en Q, cuando tenemos los actores ordenados de mayor a menor influencia, las primeras ramas que exploramos tienen mejores posibilidades de tener influencia alta.

3. Ejercicio 2: Backtracking con poda

3.1. Descripción del problema

El problema sigue siendo el mismo que en el ejercicio 1: dada una red social $G = (Q, K, p)$ buscamos la clique de influencia máxima.

Definimos ahora un conjunto de actores **independiente** como aquel $I \subseteq V$ que cumple que $\{v, w\} \notin E$ para todo $v, w \in I$. El **poder de influencia** de un conjunto independiente está dado por $\bar{p}(I) = \max\{p(v) : v \in I\}$.

3.2. El algoritmo

El algoritmo es el mismo de la seccion 2.2, pero con una poda nueva.

Sea (Q, K) una solución parcial, y sea $I_1, I_2, ..., I_k$ una partición en conjuntos independientes de K.

Veamos que para toda clique $Q' \subseteq K$ vale lo siguiente:

$$p(Q) + \sum_{i=1}^{k} \bar{p}(I_k) \geq p(Q \cup Q') \tag{1}$$

En primer lugar, tenemos que $p(Q \cup Q') = p(Q) + p(Q')$ ya que ambas son cliques y están disjuntas. Entonces lo que queremos probar es que

$$\sum_{i=1}^{k} \bar{p}(I_k) \geq p(Q') \tag{2}$$

Tomemos una clique $\tilde{Q} \subseteq K$ tal que tenga exactamente un actor de cada I_k, y que tenga al actor de mayor influencia de cada I_k (es decir, el $v \in I_k / \bar{p}(I_k) = p(v)$).

En ese caso,

$$p(\tilde{Q}) = \sum_{v \in \tilde{Q}} p(v) = \sum_{v \in \tilde{Q}} \bar{p}(I_k) = \sum_{i=0}^{k} \bar{p}(I_k) \tag{3}$$

Solo queda ver que $p(\tilde{Q}) \geq p(Q')$ para toda clique $Q' \subseteq K$

Sabemos que I_k es un conjunto independiente para cada k, es decir, dados dos actores en él, seguro no son amigos. Entonces, Q' puede tener a lo sumo un actor de cada I_k, porque tiene que ser clique. Entonces, podemos ver que $\tilde{Q}$ tiene la mayor influencia posible, ya que no existe una clique con más elementos, ni una con elementos que tengan más influencia.

En palabras, acabamos de probar que para toda clique Q' de K (en particular la más grande) y para toda partición de K, la influencia de $Q \cup Q'$ es menor o igual que la de Q más la sumatoria de las influencias de cada conjunto de la partición.

Entonces proponemos hallar alguna partición en conjuntos independientes de K y calcular la sumatoria de sus influencias. Luego, si esa sumatoria más la influencia de Q no superan la influencia máxima encontrada hasta el momento, tampoco lo hará ninguna clique que se pueda formar con los elementos de K.

Podemos ver que si queremos que la poda

$$p(Q) + \sum_{i=0}^{k} \bar{p}(I_k) \leq p(S) \tag{4}$$

se ejecute la mayor cantidad de veces posible, es conveniente encontrar una partición de K que minimice esa sumatoria.

El algoritmo goloso diseñado para encontrar dicha partición se basa en la siguiente idea: uno de los mejores escenarios para minimizar la sumatoria de influencias es que todos los actores de influencias altas puedan estar en el mismo conjunto. De esta forma, solo se considera la de un sólo actor, y el resto *"desaparecen"*.

Para hacer eso, ordenamos los actores de K según su influencia, de mayor a menor, y los vamos incluyendo en el conjunto siempre y cuando no sean amigos de ninguno de los que ya pretenecen a él. Cuando ya intentamos meter a todos los actores, repetimos lo mismo, con un conjunto nuevo y sólo considerando aquellos actores que aún no fueron agregados a ningún conjunto.

Complejidad

Probamos que el tiempo de ejecución de cada nodo en el algoritmo del problema 1 es $O(n^2)$. Entonces, para que la poda propuesta en esta sección valga la pena, necesitamos que su complejidad no sea mayor.

Para evaluar la complejidad de hallar una partición en conjuntos independientes de K, analicemos primero cuánto cuesta hallar un único conjunto independiente.

K tiene a lo sumo n actores, y para cada uno tenemos que ver que no sea amigo de nadie perteneciente al conjunto armado hasta el momento. Este conjunto, de nuevo, tiene a lo sumo n elementos[1]. Entonces, si para cada actor tenemos que chequear que no sea amigo de n actores, tenemos que hallar un conjunto independiente en K toma a lo sumo $O(n^2)$.

Habiendo construido un conjunto independiente, para hallar el resto de la partición tenemos que repetir el proceso, pero con un K más chico. De forma que construir cada uno de los conjuntos faltantes toma $O(k^2)$, con $k < n$. La complejidad de encontrar la partición completa, es la suma de las complejidades de encontrar cada uno de los conjuntos independientes, que vimos que son a lo sumo $O(n^2)$. Por lo tanto el proceso completo lleva $O(n^2)$, que es lo que queríamos ver.

3.3. Resultados

Vamos a comparar los tiempos de ejecución de los ejercicios 1 y 2, con K ordenado de mayor a menor influencia.

[1]Esta cota es bastante generosa ya que si hay k actores en K no puede haber más de $n - k$ en el conjunto. Sin embargo, nos alcanza con probar que la complejidad no supera a $O(n^2)$, por lo que no importa que esta cota no sea la más ajustada.

Instancia	Tiempo Ej. 1	Tiempo Ej. 2
brock200_1.clq	10.52	2.43
brock200_2.clq	0.08	0.07
brock200_3.clq	0.37	0.17
brock200_4.clq	1.0	0.41
brock400_2.clq	> 600	> 600
brock400_3.clq	> 600	> 600
brock400_4.clq	> 600	> 600
C125.9.clq	37.41	178.01
C250.9.clq	> 600	> 600
c-fat200-1.clq	0.0	0.0
c-fat200-2.clq	0.0	0.0
c-fat200-5.clq	0.01	0.0
c-fat500-10.clq	1.1	0.0
c-fat500-1.clq	0.01	0.0
c-fat500-2.clq	0.01	0.0
c-fat500-5.clq	0.02	0.0
DSJC500_5.clq	10.75	6.15
gen200_p0.9_44.clq	> 600	186.47
gen200_p0.9_55.clq	> 600	63.9
hamming6-2.clq	0.0	0.0
hamming6-4.clq	0.0	0.0
hamming8-2.clq	> 600	0.39
hamming8-4.clq	2.53	0.66
johnson16-2-4.clq	3.16	12.43
johnson8-2-4.clq	0.0	0.0
johnson8-4-4.clq	0.02	0.01
MANN_a9.clq	0.69	4.57
p_hat1000-1.clq	3.78	4.24
p_hat1500-1.clq	35.71	33.59
p_hat300-1.clq	0.02	0.03
p_hat300-2.clq	2.58	0.73
p_hat300-3.clq	> 600	25.77
p_hat500-1.clq	0.19	0.23
p_hat500-2.clq	385.35	19.17
p_hat700-1.clq	0.78	0.83
p_hat700-2.clq	> 600	482.07
san1000.clq	> 600	21.89
san200_0.9_2.clq	> 600	41.09
san200_0.9_3.clq	> 600	366.15
san400_0.7_1.clq	> 600	41.75
san400_0.7_2.clq	> 600	51.87
san400_0.7_3.clq	> 600	54.59
sanr200_0.7.clq	2.66	0.88
sanr200_0.9.clq	> 600	165.3
sanr400_0.5.clq	3.27	1.99

Podemos observar que en la mayoría de los casos fue más rápido el segundo ejercicio, pero no en todos. En esos casos en los que el primer algoritmo funciona mejor es probable que el tiempo que lleva calcular la partición no valga la pena para la cantidad de podas que se realizan.

4. Ejercicio 3: Programación dinámica

4.1. Descripción del problema

Tenemos un conjunto de actividades $\mathcal{A} = \{A_0, ..., A_{n-1}\}$, donde cada A_i se realiza en un intervalo de tiempo $[s_i, t_i]$ con $s_i, t_i \in \mathbb{N}$ y $0 \leq s_i < t_i \leq 2n$. Además, tenemos una función de beneficio $b : \mathcal{A} \longrightarrow \mathbb{N}$; el objetivo del problema es encontrar un subconjunto de actividades $S \subseteq \mathcal{A}$ tal que se maximice el beneficio $b(S) = \sum_{A \in S} b(A)$, sin que haya solapamiento de actividades.

Notemos que es posible resolver este problema planteándolo como el de la clique más influyente de los ejercicios anteriores:

- Las actividades representan los actores
- La relación de *no solapamiento* representa la amistad
- El beneficio representa la influencia
- Un subconjunto de actividades sin solapamiento representa una clique

De esta forma, buscar el subconjunto de beneficio máximo es equivalente a buscar la clique de influencia máxima.

Sin embargo, vimos que los algoritmos planteados para resolver el problema de la clique más influyente toman $O(n^2)$ por cada nodo, pero es posible encontrar algoritmos más eficientes para el problema de selección de actividades.

4.2. El algoritmo

Supongamos que $\mathcal{A}$ esta ordenado según el tiempo de inicio de las actividades, es decir, $s_i \leq s_{i+1}$ para todo $0 \leq i < n - 1$.

Definimos la función recursiva $mb : \{0, ..., n\} \longrightarrow \mathbb{N}$ como:

$$mb(i) = \begin{cases} 0 & \text{si } i = n \\ max\{mb(i+1); mb(k) + b(A_i)\} & \text{cc.} \end{cases} \tag{5}$$

Donde A_k es la primer tarea que cumple que $t_i < s_k$.

El problema de selección de actividades es de **optimización combinatoria**, ya que dentro de un conjunto de soluciones posibles finito buscamos aquella que maximice alguna medida. Es decir que, si exploramos todas las soluciones posibles y nos quedamos con la mejor, podemos asegurar que el agoritmo es correcto.

Nuestra función recursiva mb considera para todas las actividades dos opciones: incluirla o no. Y en cada paso, nos quedamos con aquella opción que nos de el mejor resultado. Esto es suficiente para demostrar que la función es correcta.

La implementación de esta función recursiva es igual de simple que su definición. Pero, para reducir la complejidad de $O(2^n)$, podemos hacer programación dinámica, e ir guardando los resultados de cada llamada. Al hacer esto, no se realizarán más de n llamadas recursivas, que es claramante más eficiente.

Como en cualquier algoritmo de programación dinámica, hay dos formas de implementarlo: **top-down** o **bottom-up**. En este caso, los tiempos de ejecución en ambos casos no deberían diferir mucho, ya que para obetener el resultado necesitamos resolver los n subproblemas.

Reconstrucción de la solución

Implementamos un algoritmo para recuperar el conjunto de actividades que nos provee el beneficio máximo que encontramos, y este se basa en comparar el resultado de cada llamada con los resultados de las dos llamadas que necesitó para calcularse. Dependiendo de a cuál sea igual, sabemos si agregamos o no la actividad en cuestión.

Complejidad

Por lo explicado antes, hacemos a los sumo n llamadas recursivas, y en cada una de ellas buscamos a la siguiente actividad que no se solapa. Para hacer esto en orden constante, nos guardamos antes de llamar a la función un vector que tenga la primer actividad que empieza después de un tiempo t, para todo $0 \leq t \leq 2n$. Entonces, la complejidad de la función total es $O(n)$.

Quedaría ver que podemos obtener el vector que guarda la siguiente actividad en orden lineal. Veamos el pseudocódigo:

Algorithm 2

```
1: function COMPUTARSIGUIENTES
2:     for act ∈ actividades do
3:         for t from (act - 1).comienzo to act.comienzo do
4:             siguientes[t] = act
```

Recorremos una vez todas las actividades, que son n. Además, recorremos una vez el vector `siguientes`, de tamaño $2n$, ya que por cada actividad llenamos todos los t que la tienen como siguiente[2]. Entonces la complejidad queda $O(n + 2n) = O(n)$.

El algoritmo de reconstrucción de la solución también lleva $O(n)$, ya que recorremos los n lugares de la memoria y para algunos de ellos buscamos la próxima actividad compatible en $O(1)$.

4.3. Resultados

Instancia	top-down	bottom-up
instancia_1.txt	0.0	0.0
instancia_2.txt	0.0	0.0
instancia_3.txt	0.0	0.0
instancia_4.txt	0.0	0.0
instancia_5.txt	0.0	0.0
instancia_6.txt	0.0	0.0
instancia_7.txt	0.0	0.0
instancia_8.txt	0.0	0.0
instancia_9.txt	0.0	0.0
instancia_10.txt	0.0	0.0
instancia_11.txt	0.09	0.05
instancia_12.txt	0.09	0.05
instancia_13.txt	0.09	0.05

[2]Cada t tiene una única actividad siguiente.

Podemos notar que si bien ambas implementaciones tienen complejidad lineal, bottom-up parece ser más rápida que top-down. Esto puede deberse a que la implementación bottom-up es iterativa, mientras que la top-dow utiliza llamadas recursivas, las cuales son más demandantes para el sistema.

5. Ejercicio 4: Greedy

5.1. Descripción del problema

Queremos resolver el problema de selección de actividades del ejercicio anterior, pero ahora suponiendo que $b(A_i) = 1$ para todo $0 \leq i < n$. Notemos que, entonces, lo que buscamos es el subconjunto de actividades sin superposición de mayor cardinal.

5.2. El algoritmo

La estrategia del algoritmo es una golosa: en cada paso, elegimos la actividad que no se solape con las ya elegidas cuyo tiempo de fin sea lo más temprano posible.

Demostremos por inducción que usando este algoritmo siempre hallamos una solución óptima:

> Sea $S = \{s_0, ..., s_j\}$ una solución óptima al problema (ordenada por tiempo de fin de las actividades),
> sea $G = \{g_0, ..., g_i\}$ la solución golosa.
>
> Llamamos S_k y G_k a $\{s_0, ...s_k\}$ y $\{g_0, ..., g_k\}$ respectivamente.
>
> Queremos probar $\mathcal{P}(k) \equiv |G_k| \geq |S_k|$, para todo $0 \leq k < j$.
>
> **Caso base:** $k = 0$
> $S_0 = \{s_0\}$, $G_0 = \{g_0\}$, con $fin(g_0) \leq fin(s_0)$
> $|G_0| = |S_0| = 1 \Rightarrow \mathcal{P}(0)$
>
> **Paso inductivo:** $\mathcal{P}(k) \Rightarrow \mathcal{P}(k+1)$, con $0 < k \leq j$
> Si $k \leq i$, tenemos que $G_{k+1} = G_k \cup \{g_{k+1}\}$ y $S_{k+1} = S_k \cup \{s_{k+1}\}$
> $\Rightarrow |G_{k+1}| = |G_k| + 1 \geq |S_k| + 1 = |S_{k+1}| \Rightarrow \mathcal{P}(k+1)$
> Si $k > i$, g_i es la última actividad de G.
> Tenemos que $com(s_{k+1}) > fin(s_k) > fin(s_i) \geq^{*} fin(g_i)$, lo cual es absurdo, pues en ese caso s_{k+1} estaría en G por la definición de nuestra estrategia golosa.
> $\therefore k \leq i$
>
> * Probemos por inducción que $fin(s_k) \geq fin(g_k)$:
> **Caso base:** $k = 0$ es trivial por la definición de la estrategia.
> **Paso inductivo:** $fin(s_k) \geq fin(g_k) \Rightarrow fin(s_{k+1}) \geq fin(g_{k+1})$
> Tenemos que $com(s_{k+1}) > fin(s_k) \geq fin(g_k)$
> $\Rightarrow s_{k+1}$ es una actividad compatible con $G_k \Rightarrow fin(g_{k+1}) \leq fin(s_{k+1})$ pues G_{k+1} se arma con la estrategia golosa y g_{k+1} es compatible con G_k por definición de la estrategia golosa.

Observemos que, si bien probamos que el algoritmo es correcto cuando todos los beneficios son iguales, esto no es así en otros casos. Por ejemplo:

$$\mathcal{A} = \{[0,2];[1,3];[3,4]\}, \text{ con beneficios } 1,3,1 \text{ respectivamente.}$$

En este caso, el algoritmo goloso nos devuelve el subconjunto $S = \{[0,2];[3,4]\}$, que es el de mayor cardinal, pero de beneficio 2. Sin embargo la solución al problema es $S' = \{[1,3]\}$ de beneficio 3.

La implementación del algoritmo goloso es muy simple: suponiendo que tenemos $\mathcal{A}$ ordenado según el fin de las actividades de menor a mayor, las recorremos una a una, y si las podemos agregar sin solapamiento con las ya elegidas lo hacemos.

Complejidad

El algoritmo recorre las n posibles actividades, y para cada una de ellas hace operaciones constantes (comparación, agregar a un vector y asignar un entero). Por lo tanto, suponiendo que las actividades están ordenadas de menor a mayor por tiempo de fin, tiene complejidad $O(n)$.

Si por el contrario no están ordenadas, de cualquier forma podemos hacerlo en $O(n)$, ya que tenemos el tiempo de fin acotado por $2n$ lo que nos permite hacer Bucket Sort.

Comparación entre implementaciones

Notemos que este algoritmo es sustancialmente más sencillo de implementar que el anterior: simplemente se ordenan las actividades y luego se las recorre, y listo. En cambio, el algoritmo de programación dinámica requiere pensar y estructurar bien la recursión, almacenando además muchos más datos en memoria, lo que es claramente más elaborado. Así mismo, el algoritmo de programación dinámica da el resultado óptimpo para cualquier instancia, mientras que el goloso se limita a hacerlo sólo en aquellas donde $b(A_i) = 1$ para todo $0 \leq i < n$.

5.3. Resultados

Instancia	Tiempo
interval_instance_1	0.0
interval_instance_2	0.0
interval_instance_3	0.0
interval_instance_4	0.0
interval_instance_5	0.0
interval_instance_6	0.01
interval_instance_7	0.0
interval_instance_8	0.07
interval_instance_9	0.07
interval_instance_10	0.07
interval_instance_11	0.87
interval_instance_12	0.89
interval_instance_13	0.87

6. Conclusión

Vimos diferentes formas de implementar y resolver el problema de determinar la clique más influyente en una red social y cómo maximizar el beneficio organizando bien una agenda con

actividades.

A destacar del ejercicio de la clique más influyente, vimos que el orden de los datos de entrada (en este caso los actores) puede determinar la eficacia de una poda y que la implementación de podas adicionales (como la del ejercicio 2) que no aumentan la complejidad teórica, puede resultar en un algoritmo que tarda más en la práctica para algunas instancias.

Del problema de selección de actividades podemos destacar la posibilidad de reinterpretarlo para resolverlo con el algoritmo de la clique más influyente, y que cuando los beneficios de todas las actividades son iguales, se puede utilizar un algoritmo goloso más sencillo de implementar para resolver el problema obteniendo una solución óptima.