DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico N°1

Backtracking, Programación Dinámica, Algoritmos Golosos

27 de junio de 2022 — Algoritmos y Estructura de Datos 3

Integrante	LU	Correo electrónico
Donzis, Tomás	443/20	donzis.tomy@gmail.com
Collasius, Federico	164/20	fede.collasius@gmail.com
Totaro, Facundo Ariel	43/20	facutotaro@gmail.com
Venturini, Julia	159/20	juliaventurini00@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2610 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (++54 +11) 4576-3300
https://exactas.uba.ar

Índice

1. Backtracking **1**

1.1. Descripción del problema 1

1.2. Algoritmo 1

1.2.1. Clique mas influyente 1

1.2.1.1. Estructura 1

1.2.1.2. Funcion Principal 2

1.2.1.3. Superposicion de Subproblemas 2

1.2.2. Funciones auxiliares: 2

1.3. Resultados 3

2. Podando el árbol de backtracking **3**

2.1. Algoritmo Goloso 5

3. Programación Dinámica **5**

3.1. Descripción del problema 5

3.2. Relación con el problema de la clique más influyente 5

3.3. Descripción recursiva del problema 6

3.4. Superposición de subproblemas 7

3.5. Algoritmos 8

3.5.1. Top-Down 8

3.5.2. Bottom-Up 9

3.6. Reconstrucción de la solución 9

3.7. Resultados 10

4. Algoritmos Golosos **10**

4.1. Descripción del problema 10

4.2. Estrategia Golosa 10

4.3. Algoritmo 13

4.4. Resultados 13

1. Backtracking

1.1. Descripción del problema

Una red social G se representa como una tripla (V,E,p) donde: V es el conjunto de actores de la red social, E es el conjunto de amistades de la red social, considerando que la relacion de amistad es simetrica, no reflexiva y no transitiva y p es la funcion p:V $\rightarrow N_{>0}$ que denota el poder de influencia de un actor.
Dada una red social G queremos determinar cual es la clique de mayor influencia en G. Una clique es un conjunto Q $\subseteq$ V donde para cada par de actores v,w $\in$ E, v y w son amigos. El poder de influencia de la clique Q es p(Q) = $\sum_{v \in Q} p(v)$. Lo que buscamos con este ejercicio es devolver una clique Q que maximice la suma de las influencias de los actores que la conforman.

1.2. Algoritmo

1.2.1. Clique mas influyente

Definimos a n como $|V|$, es decir la cantidad de actores en el grafo.

1.2.1.1. Estructura

Vamos a empezar por definir las estructuras que vamos a usar. Tenemos una clase **RedSocial**, cuya parte privada esta conformada por un vector de actores y una matriz de amistades. Los actores están representados con el struct **Actor**, cuya parte privada esta conformada por el *id* y la *influencia* de dicho actor (ambos enteros).

Vamos a representar la Red Social G como una **Matriz de Adyacencia**. Su construccion sera de orden O(n^2) y sera pagado en la construccion del grafo. Saber si dos actores i , j estan relacionados tendra costo O(1) ya que requiere pedir la posicion $[i][j]$ de la matriz. Al ser relaciones simetricas, la posicion $[j][i]$ debera ser equivalente a la anterior.

Al construir la Red Social, armamos el vector de **Actores** así como el de **Amistades** e inicializamos la **Matriz de Adyacencia** marcando con true aquellas posiciones i, j y j, i donde los actores de id i y j sean amigos.

Luego ordenamos el vector de actores de menor a mayor según la influencia de cada actor. Al experimentar con el orden del vector de actores llegamos a la conclusión de que ordenarlo de menor a mayor en nuestro caso es mas eficiente. Esto se debe a que, como tomamos el ultimo actor de K en cada iteración para agregarlo a Q, al ser el de mayor influencia en K, nuestra clique Q arranca teniendo una influencia relativamente alta, lo que permite realizar podas mas groseras mucho mas rápido (en niveles mas altos del árbol de llamadas). El cuadro demuestra como varian los tiempos en cada caso.

Nombre de Instancia	Tiempo menor a mayor (seg.)	Tiempo mayor a menor(seg.)
brock200_1.clq	25.80	718
johnson16-2-4.clq	3.11	9.67
phat700-1.clq	1.26	2.69

Una vez que nuestra red social esta construida, se llama a la función cliqueMasInfluyente, que toma como parámetros un vector de actores Q (por referencia) y un vector de actores K (por referencia). En este caso, vamos a llamar a la función con $Q =<>$ y $K = V$.

1.2.1.2. Funcion Principal

La función **cliqueMasInfluyente** utiliza la técnica algorítmica de backtracking. En cada llamada a la funcion primero nos fijamos si estamos en el caso base: K vacío. Si es cierto, vemos si la influencia de la solución parcial Q es mayor a la influencia máxima vista hasta el momento. En caso afirmativo actualizamos la influencia maxima actual por la influencia de Q y además guardamos al clique Q como el clique mas influyente hasta el momento.

Si no estamos en el caso base, es decir que quedan actores en K, nos fijamos si la influencia de la solución parcial Q más la suma de las influencias de todos los actores de K es menor a la influencia máxima vista hasta el momento. Esto sirve a modo de **poda por optimalidad**, ya que si esto se cumple, descartamos la solución parcial.

En caso que lo anterior sea falso, guardamos el vector Q y el vector K en nuevos vectores que llamamos Qd y Kd. Luego, agregamos el ultimo elemento de K (llamémoslo v) a Q (que sera el de mayor influencia dentro de K) y lo sacamos de K. Ahora, actualizamos el vector K llamando a la función **soloAmigosDeQEnK**, que saca aquellos actores que no sean amigos de v a Q. Como sabemos que los actores que ya estaban en K eran amigos de todos los de Q hasta el momento, entonces solo basta con sacar aquellos que no sean amigos de v para que se mantenga la primera parte del invariante (todos los actores en K son amigos de todos los de Q). Para mantener la segunda parte del invariante (todo actor en K tiene por lo menos un no amigo en K) lo que hacemos es llamar a la función **filtrarPopulares** que lo que hace es buscar aquellos actores que sean amigos de todos los actores de K (los llamamos **populares**) y los agrega a Q. Una vez hecho esto, llamamos a la función **cliqueMasInfluyente** con el Q y K actual. Así estamos computando las soluciones que contienen a v en la clique.

Para computar las soluciones donde v no es parte de la clique lo que hacemos es: Restaurar el Q y el K igualándolos a Qd y Kd respectivamente. Luego sacamos el ultimo elemento de K (v) y agregamos el ahora ultimo (llamémoslo v'). Ahora resta hacer el mismo procedimiento que en el caso previo. Es decir, actualizamos K para sacar todos aquellos actores que no sean amigos de v', agregamos los actores populares de K a Q y llamamos a **cliqueMasInfluyente** con el Q y K actual.

1.2.1.3. Superposicion de Subproblemas

En cada llamada a la función estamos considerando el caso donde agregamos al actor v a la clique y el caso donde no. Cada una de estas dos llamadas computa uno de los dos subárboles de la extensión del nodo (Q, K). No va a haber superposición de problemas ya que en cada iteración del árbol vamos a tomar una decisión de sacar un actor de K o agregarlo a Q. Por lo tanto, vamos a tener una rama en la que el actor v esta en Q y otra en la que ya no está en K por lo tanto no puede formar parte de futuros Q. Esto por sí solo ya nos garantiza que no vamos a tener superposición de problemas ya que los pares (K, Q) de cada nodo siempre serán distintos.

1.2.2. Funciones auxiliares:

En esta sección vamos a detallar la complejidad y explicar el uso de la funciones auxiliares que usamos en la funcion principal **cliqueMasInfluyente**.

void soloAmigosDeQEnK: Esta función toma como parámetros dos vectores de actores (Q y K, por referencia) y lo que hace es iterar sobre cada actor de K, fijándose en cada iteración i si el ultimo actor de Q es amigo del actor i-esimo de K. Si son amigos, agrega al actor $K[i]$ al vector soloAmigosDeQEnK que comienza la función siendo vacío. Al finalizar el ciclo, K es igualado a soloAmigosDeQEnK, de modo que todos los actores de K son amigos del ultimo actor de Q. La complejidad temporal de esta función es $O(n)$ ya que recorre el vector K (que en peor caso es igual a V) y luego se iguala K a soloAmigosDeQEnK. En peor caso, $soloAmigosDeQEnK = V$, entonces la complejidad termina siendo $O(n + n) = O(2n) = O(n)$.

int influenciaDeGrupo: Esta función recibe como parámetro un vector de actores sobre el cual va a iterar, sumando la influencia del actor i a la influencia total, que comienza siendo cero. Al finalizar el ciclo, la función devuelve la influencia máxima. La complejidad temporal de esta función es O(n) ya que recorre un vector de actores que en peor caso es igual a V.

void filtrarPopulares: Esta función recibe como parámetros 3 vectores de actores Q, K y $sinPopulares$ (por referencia). Lo que hace es, para cada actor de K, se fija qué actores de K son sus amigos (fijarse si dos actores son amigos cuesta $O(1)$), guardándose la cantidad de amigos en una variable de tipo entero llamada amigos, que comienza siendo 0 para cada actor. Al finalizar el ciclo interno, se fija si la cantidad de amigos de este actor es igual a la cantidad de actores-1, lo que significaría que es "*popular*", es decir, que es amigo de todos menos de si mismo. Si esto es cierto, entonces agrega este actor a Q, sino lo agrega a $sinPopulares$. La complejidad temporal de esta función es O(n^2) ya que para cada actor en K recorre K, que en peor caso es igual a V.

bool sonAmigos: Esta función toma como parámetros dos actores de id i y j y devolviendo el booleano que se encuentra en la matriz de adyacencia en la posicion (i, j). La matriz de amistades se crea en el constructor con complejidad temporal O(n^2) ya que tiene n filas y n columnas. De este modo, averiguar si dos actores son amigos nos cuesta $O(1)$.

Luego la complejidad de procesamiento de cada nodo será: $O(n) + O(n) + O(n^2) = O(n^2)$. Luego como en cada llamado recursivo estamos reduciendo a K en 1 hasta 0, tenemos que la complejidad total del algoritmo será $O(2^n.n^2)$.

1.3. Resultados

2. Podando el árbol de backtracking

La catedra introduce la siguiente poda para reducir los candidatos a revisar: sea G = (V,E,p) una red social. Un conjunto de actores I $\subseteq$ V de G es independiente cuando $vw \notin E$ para todo $v, w \in V$. El poder de influencia de un conjunto independiente I esta definido como $\overline{\text{p}}(I) = max\{\text{p(v)}|\text{v} \in I\ \}$. Luego para cada nodo (Q, K), si $p(Q) + \sum_{i=1}^{k} p(I_i) \leq p(S)$ entonces podemos descartar ese nodo.
Esta poda surge de la siguiente desigualdad: $p(Q) + \sum_{i=1}^{k} p(I_i) \geq p(Q \cup Q')\ \forall Q' \subset K$ con Q' un clique. Esta es verdadera ya que dado que $I_1...I_k$ forma una partición de subconjuntos independientes y Q' es una clique, entonces a lo sumo Q' tiene un elemento por partición (o ninguno) ya que por definición todos los elementos de Q' están relacionados entre sí. Luego como la influencia de Q' es la suma de las influencias de los actores que pertenece a él, para conseguir

Nombre de Instancia	Resultado Obtenido	Tiempo Ej 1 (seg.)	Tiempo Ej 2 (seg.)
brock200_1.clq	2821	25.80	1.31
brock200_2.clq	1428	0.16	0.05
brock200_3.clq	2062	0.63	0.13
brock200_4.clq	2107	1.86	0.28
brock400_2.clq	3350	>800	553.11
brock400_3.clq	3471	>800	467.74
brock400_4.clq	3626	>800	377.46
C125.9.clq	2529	138.38	1.30
C250.9.clq	5092	>800	472.86
c-fat200-1.clq	1284	0.00	0.02
c-fat200-2.clq	2411	0.01	0.00
c-fat200-5.clq	5887	0.01	0.01
c-fat500-10.clq	11586	0.09	0.01
c-fat500-1.clq	1354	0.04	0.02
c-fat500-2.clq	2628	0.04	0.03
c-fat500-5.clq	5841	0.05	0.02
DSJC500_5.clq	1725	17.85	3.63
gen200_p0.9_44.clq	5043	>800	92.28
gen200_p0.9_55.clq	5416	>800	30.65
hamming6-2.clq	1072	0.01	0.00
hamming6-4.clq	134	0.00	0.00
hamming8-2.clq	10976	>800	0.07
hamming8-4.clq	1472	4.58	0.41
johnson16-2-4.clq	548	3.23	0.71
johnson8-2-4.clq	66	0.00	0.00
johnson8-4-4.clq	511	0.03	0.00
MANN_a9.clq	372	0.53	0.01
p_hat1000-1.clq	1514	7.01	2.67
p_hat1500-1.clq	1619	57.70	21.09
p_hat300-1.clq	1057	0.06	0.03
p_hat300-2.clq	2487	6.13	0.39
p_hat300-3.clq	3774	>800	14.07
p_hat500-1.clq	1231	0.33	0.18
p_hat500-2.clq	3920	>800	10.50
p_hat700-1.clq	1441	1.33	0.59
p_hat700-2.clq	5290	>800	262.86
san1000.clq	1716	>800	25.33
san200_0.9_2.clq	6082	>800	19.40
san200_0.9_3.clq	4748	>800	177.80
san400_0.7_1.clq	3941	>800	29.31
san400_0.7_2.clq	3110	>800	37.73
san400_0.7_3.clq	2771	>800	39.93
sanr200_0.7.clq	2325	5.50	0.51
sanr200_0.9.clq	5126	>800	79.34
sanr400_0.5.clq	1835	5.10	1.17

Cuadro 1: Las instancias se corrieron en una computadora con procesador AMD Ryzen 5 3500U.

el mayor Q' posible debería tomar un actor de cada subconjunto, que estén relacionados y ser el que tiene máxima influencia allí. Luego la desigualdad es verdadera.

2.1. Algoritmo Goloso

A continuación, presentamos un algoritmo goloso que busca obtener una particion de K que minimice la suma del poder de influencia de los conjuntos independientes de K. El algoritmo goloso, que llamamos influenciaGruposIndependientes toma un conjunto de conjunto de actores I que modela las particiones de K y un conjunto de actores K ordenado de menor a mayor influencia. La idea es empezar desde el último elemento e ir colocando los actores de K en el primer conjunto de I en el que no sean amigos de nadie. Si el actor tiene por lo menos un amigo en todos los conjuntos de la particion I hasta ahora, se lo agrega en un nuevo conjunto que contendra a ese actor unicamente. En cada paso el algoritmo busca minimizar el poder de influencia de toda la partición ya que se intenta colocar a los nuevos actores en las particiones ocupadas por elementos con mayor influencia que van a absorber la suya. Esto es porque la influencia de un conjunto independiente es la influencia máxima entre todos los actores del conjunto. En mejor caso serán todos no amigos entre sí y la suma de las particiones será igual al actor más influyente de K.

La complejidad del algoritmo será O(n^2) en peor caso. Luego la cota de procesamiento se sigue cumpliendo.

Ver el cuadro 1 para notar la diferencia en tiempos de ejecucion entre el algoritmo de backtracking con y sin la poda.

3. Programación Dinámica

3.1. Descripción del problema

Tenemos el siguiente problema: "Dado un conjunto de actividades $\mathcal{A} = \{A_0,...,A_{n-1}\}$ y una función de beneficio b: $\mathcal{A} \longrightarrow \mathbb{N}$ el problema de selección de actividades consiste en encontrar un subconjunto de actividades $\mathcal{S}$ cuyo beneficio $(\mathcal{S}) = \sum_{\mathcal{A}\in\mathcal{S}}$ b($\mathcal{S}$) sea máximo entre todos aquellos subconjuntos de actividades que no se solapan en el tiempo. Cada actividad A_i se realiza en algún intervalo de tiempo $[s_i, t_i]$ siendo $s_i \in \mathbb{N}$ su momento inicial y $t_i \in \mathbb{N}$ su momento final. Suponemos que $0 \leq$s$_i$ $< t_i$ 2n para todo $0 \leq i < n$. A los subconjuntos que no se solapan en el tiempo los llamamos compatibles."

3.2. Relación con el problema de la clique más influyente

Para la resolución de este problema, podríamos usar la solución al problema de la clique más influyente. Para hacerlo tenemos que pensar a las actividades de la siguiente manera: cada actividad es una persona, y su beneficio corresponde a su influencia. Dos actividades son amigas entre si cuando no se superponen en el tiempo. Luego, encontrar un grupo de actividades que no se solapen entre si equivale a encontrar una clique, ya que tenemos un grupo de actividades en el que todas son amigas de todas, que es lo mismo que decir que ninguna se superpone con otra de ese conjunto. Luego, tener a la clique más influyente equivale a pedir a cual es la clique que tiene la mayor suma de influencia entre todos sus miembros. En este caso, sería de todos los subconjuntos de actividades que colisionan entre si, cual es el que mayor suma de todas las influencias tiene, y

en este caso seria la suma de sus beneficios.

3.3. Descripción recursiva del problema

Asumimos que el conjunto de actividades $\mathcal{A} = \{A_0,...,A_{n-1}\}$ está ordenado por orden de inicio de las actividades.

Sea K_i el conjunto de actividades $\{A_i, A_{i+1}, \ldots, A_{n-1}\}$. Como las actividades de A estan ordenadas, las actividades de K_i respetan el orden. Definiremos la función recursiva B : N $\longrightarrow$ N tal que B(i) consiste en encontrar el máximo beneficio que nos puede dar el subconjunto de actividades de K_i. Por lo que la solución al problema consiste en encontrar B(0). Notemos que si i >= n, K_i es vacío por lo que no tiene beneficio. Puede pasar que haya casos en los que A_i se superponga con alguna de las actividades que le siguen, y traiga más beneficio no incluir a la función A_i, por lo que B(i) = B(i+1) ya que tomariamos como el beneficio al beneficio que nos puede dar el subconjunto K_{i+1} (ya que no contamos a A_i). Tambien otra opción para maximizar el beneficio es tomar la suma entre el beneficio que nos puede dar la actividad i y el beneficio máximo de un subconjunto de actividades que arrancan después de que termina la actividad i (si es que existe alguna actividad que comience después de que termine la actividad i). Cómo estamos buscando el máximo beneficio, tomamos el máximo entre estos dos.

La función recursiva que obtenemos sería la siguiente:

B : $\{0, ..., n\} \to \mathbb{N}$

$$\mathrm{B}(i) = \begin{cases} 0 & \text{si } i = n \\ \max\left(B(i+1),\ b(i) + B(\Pi(i))\right) & \text{en caso contrario} \end{cases}$$

Siendo Π: $\mathbb{N}_0 \longrightarrow \mathbb{N}$, una función que para el conjunto $\mathcal{A}$ dada un número de actividad i, nos devuelve el número de la primera actividad de $\mathcal{A}$ que comienza despues de que termina A_i, es decir que no se solapa con la actividad i, o devuelve n si no hay ninguna actividad que no se solapa con A_i.

Sin embargo, nos faltaría probar que la función planteada es correcta. En principio lo que debemos hacer es reformular el enunciado para interpretarlo como un problema de optimización combinatoria. Se nos ocurrió que podríamos redefinirlo como
Dado un conjunto de actividades A = $A_1, \ldots, A_n$ con sus respectivos rangos de tiempo asociados $[s_i, t_i]$ y sus beneficios b = $b_1, \ldots, b_n$ queremos encontrar Bf* que maximiza la sumatoria de los beneficios del subconjunto AC de aquellas actividades que no se solapan mutuamente (AC $\subseteq$ $\mathcal{A}$).
Esta reformulación la podemos reescribir matemáticamente como
Bf* = $máx\{\sum_{A_i \in AC} b_i \mid AC \subseteq A, (\forall A_i, A_j \in AC / i < j), t_i < s_j)\}$

Probemos por inducción que la solución que queremos encontrar es B(0), es decir B(0) = Bf*. Para esta inducción podemos mantener las variables como estaban antes, es decir hacer inducción en i, o podemos plantear $i = n - k$ y hacer inducción en k. Esto lo podemos hacer ya que la función B crece cuando i decrece.

Caso base:
Con $k = 0$, $B(n - k) = B(n) = 0$. Luego Bf*(n) es 0 porque el conjunto AC no contiene ningún elemento, por lo que la sumatoria es nula.

Ahora tomemos P(k): B(n-k) = Bf*(n-k) k: 0, . . . , n.
Paso inductivo: ¿$\forall k : 0 \leq m \leq k$, P(m) $\implies P(k+1)$?

En este caso queremos ver que Bf*(n-(k+1)) es equivalente a tomar el máximo entre B(n-(k+1)+1) y (b(n-(k+1)) + B(Π$(n - (k + 1))$)). B(n-(k+1)+1) es igual a B(n-k) y por hipotesis inductiva sabemos que B(n-k) = Bf*(n-k) (esto es P(k)). Se podría decir que este es el máximo beneficio que se puede obtener del conjunto $\{A_{n-(k+1)}, A_{n-k+1}...A_{n-1}\}$.También, Π$(n - (k + 1))$ nos dará un n-k' tal que n-k' > n-(k+1) ya que es una actividad que comienza despúes de n-(k+1). De esta forma, k' < k+1, por lo que vale, por hipotesis inductiva que B(Π$(n - (k + 1)))$ = Bf*(Π$(n - (k + 1)))$.

Entonces, Bf*(n-(k+1)) se reduce a tomar el máximo entre Bf*(n-k) y (b(n-(k+1)) + Bf*(Π$(n-(k + 1))))$. Sabemos entonces que Bf*(n-k) es el máximo beneficio del subconjunto de todas las actividades que aparecen despues de la actividad n-(k+1) en $\mathcal{A}$ y Bf*(Π$(n-(k+1)))$ es el máximo beneficio del subconjunto más grande de actividades compatibles que comienzan despues de que finaliza la actividad n-(k+1). Luego al sumarle b(n-(k+1)) tomaremos el beneficio máximo que se puede obtener agregando la actividad n-(k+1), que sería el máximo beneficio que se puede obtener del subconjunto $\{A_{n-(k+1)}, A_{n-k+1}...A_{n-1}\}$ agregando la actividad n-(k+1).

Por eso, B(n-(k+1)) se traduce en tomar el máximo entre el máximo beneficio que puedo obtener insertando o no la actividad n-(k+1) a un subconjunto de actividades que sumadas maximizan el beneficio de un subconjunto $\{A_{n-(k+1)}, A_{n-k+1}...A_{n-1}\}$, y este será el máximo que se puede obtener de $\{A_{n-(k+1)}, A_{n-k+1}...A_{n-1}\}$. Obteniendo así que B(n-(k+1)) = Bf*(n-(k+1)) por lo que vale P(k+1), demostrando así el paso inductivo.

3.4. Superposición de subproblemas

En la función recursiva, en el peor de los casos, es cuando ninguna actividad se solapa entre si. Ya que en cada llamado recursivo (salvo cuando i = n), tenemos 2 llamados recursivos, ambos a B(i-1) ya que es la siguiente que no colisiona, y por lo tanto vamos avanzando "de a uno". Por lo que tenemos sumatoria desde i=0 hasta n-1 de 2^{i+1} llamados recursivos. Que esto está acotado por O(2^n). La cantidad de subproblemas es n+1 ya que i va desde 0 a n, y son todos los posibles casos de B(i). Por lo que la cantidad de subproblemas es n. Luego, como n $\ll 2^n$ / n entonces se cumple con la propiedad de superposición de subproblemas y nos sirve utilizar la técnica de programación dinámica para resolver el problema.

3.5. Algoritmos

3.5.1. Top-Down

Algoritmo 1 Enfoque top-down

Input : Número de la actividad A_i para el cual se quiere encontrar el máximo beneficio entre $\{A_i..A_{n-1}\}$. Tambien contamos con siguienteNoColision que es un arreglo preprocesado que contiene la prómixa actividad que no colisiona con la actividad i (que si no hay ninguna entonces se encuentra n en esa posición), el arreglo beneficio que tiene los beneficios respectivos para cada actividad, la estructura de memoización M(que es un arreglo de n+1 elementos) y la constante totalDeActividades que equivale a n.

Output: Máximo beneficio entre $\{A_i..A_{n-1}\}$

```
1:  function topDown(i : ℕ) → ℕ
2:      if i == totalDeActividades then
3:          return 0                          ▷ Si no hay mas actividades que recorrer devuelvo 0
4:      else
5:          if M[i] == 0 then        ▷ Si no calcule el beneficio para el subconjunto de actividades entre i y n-1
6:              k ← 0
7:              if M[i + 1] == 0 then              ▷ Evaluamos si lo calculamos entre i+1 y n-1
8:                  k ← topDown(i + 1)                                  ▷ Si no lo hicimos lo calculamos
9:              else
10:                 k ← M[i + 1]                    ▷ Si lo hicimos no hace falta volver a calcular
11:             end if
12:             M[i] ← máx (k, beneficio[i] + topDown(siguienteNoColision[i]))
13:         end if
14:     end if
15:     return M[i]
```

El algoritmo consiste en una función recursiva que recibe un parametro i la cual calcula el maximo benefico entre las actividades $\{A_i...A_n - 1\}$. Cómo mencionamos antes, solo hay n+1 subproblemas posibles, entonces una vez resuelto uno de ellos, no es necesario volverlo a calcular. Entonces vamos a tener una estructura de memoización para guardar las instancias de estos n+1 subroblemas que va a ser un arreglo inicializado con todos 0's (ya que ninguna actividad tiene un beneficio menor a 1). Entonces la idea del algoritmo consiste en ir calculando el beneficio para todos los posible subconjuntos de actividades $\{A_i...A_{n-1}\}$ y resolver el problema sería hacer el llamado a la función con i = 0. Como se mencionó previamente, si se tiene el máximo beneficio de $\{A_{i+1}...A_{n-1}\}$, luego el máximo beneficio para $\{A_i...A_{n-1}\}$ es el máximo entre resolver el problema para $\{A_{i+1}...A_{n-1}\}$ (no agregar la actividad) y sumar el beneficio de la actividad i con el mayor subconjunto $\{A_k...A_{n-1}\}$ que no colisionan con i. Dicho esto, para no tener que resolver instancias innecesarias ya que muchas se repiten, cuando se llama a la función para un cierto i, esta se fija en la estructura de memoización si ya lo tiene calculado (es decir si en esa posición hay un elemento distinto de 0). Si es así lo retorna en tiempo constante (O(1)). Lo que sucede es que para tomar el máximo simpre se llaman a instancias más chicas, que pueden estar o no caluladas. Al hacer esto se va llenando la tabla y el cálculo se hace una vez para cada actividad y al usarse posiciones ya calculadas o retornadas de cuando se llama al cálculo, vamos llenando el arreglo de a uno desde las subinstancias más chicas hacia las mas grandes, sin hacer calculos innecesarios y calculando una sola vez para cada posición del arreglo, ya que cuando las instancias más chicas se hayan calculado, la comparación tiene tiempo constante, entonces esto nos lleva O(n) operaciones

elementales.

3.5.2. Bottom-Up

Algoritmo 2 Enfoque bottom-up

Input : Contamos con siguienteNoColision que es un arreglo preprocesado que contiene la próxima actividad que no colisiona con la actividad i (que si no hay ninguna entonces se encuentra n en esa posición), el arreglo beneficio que tiene los beneficios respectivos para cada actividad, la estructura de memoización M(que es un arreglo de n+1 elementos) y la constante totalDeActividades que equivale a n.

Output: Máximo beneficio entre $\{A_i..A_{n-1}\}$

1: **function** BOTTOMUP $\rightarrow \mathbb{N}$
2: $M[totalDeActividades] \leftarrow 0$
3: **for** $i \leftarrow totalDeActividades - 1 : i \geq 0 : i \leftarrow i - 1$ **do**
4: $M[i] \leftarrow \max(M[i+1],\ beneficio[i] + M[siguienteNoColision[i]])$
5: **end for**
6: **return** $M[0]$

Lo que se hace aca es seguir el método que proponiamos antes de ir viendo que pasa a medida que vamos agregando una actividad. Entonces vamos rellenando la estructura de memoización casilla por casilla y toma el maximo entre una instancia de un subconjunto más chico(sin agregar la actividad) y una suma que utiliza también una instancia de un subconjunto más chico(si se agrega la actividad se suma con el siguiente que no colisiona), como vamos haciendo el cálculo a medida que vamos agregando a estos subconjuntos, entonces ya estan calculados sus resultados en la estructura de memoización y acceder a estos es O(1) y hacer la comparación tambien. Esto lo hacemos para los n elementos y al tener n operaciones con complejidad temporal O(1) tenemos una complejidad total de O(n). El resultado de resolver el problema es el valor que se encuentra en la primera casilla de la estructura de memoización.

3.6. Reconstrucción de la solución

La idea es la siguiente. Dada una estructura de memoización ya completa, podemos notar que todos los beneficios son mayores o iguales a los de sus casillas siguientes, ya que para cada i tenemos el máximo beneficio del conjunto $\{A_i...A_{n-1}\}$ y al agregarle una actividad, como estan en orden de inicio, se puede considerar que su beneficio aumenta al beneficio actual, o conviene no considerarla ya que no estaría en el cronograma. Tras esta idea, en la estructura de memoización, si tenemos una posición M[i] = M[i+1] significa que el beneficio de la actividad i no se tiene en cuenta para el máximo beneficio, mientras que si M[i] > M[i+1] significa que agregamos esta actividad a las actividades de la solución optima. Entonces lo que se hace es ir recorriendo la estructura de memoización secuencialmente desde la posición 0. Una vez que encontramos que M[i] > M[i+1] agregamos el número de la actividad al vector solución y seguimos recorriendo a partir de la siguiente actividad que no colisiona con i ya que sino estariamos construyendo una solución con actividades que son incompatibles ya que colisionan entre si. El peor caso seria cuando ninguna actividad colisiona con otra, por lo que estariamos recorriendo la estructura de

memoización entera y esto tiene un costo de O(n) ya que realizar las comparaciones de accesos a posiciones de un arreglo es O(1).

3.7. Resultados

Nombre de Instancia	Resultado Obtenido	Tiempo (seg.)
instancia_1	15	0.00
instancia_2	27	0.00
instancia_3	30	0.00
instancia_4	15	0.00
instancia_5	1456	0.00
instancia_6	1416	0.01
instancia_7	1578	0.01
instancia_8	5100	0.04
instancia_9	5103	0.04
instancia_10	4834	0.04
instancia_11	15566	0.34
instancia_12	15382	0.32
instancia_13	15665	0.35

Cuadro 2: Las instancias se corrieron en una computadora con procesador AMD Ryzen 5 3500U.

Notemos que los tiempos se encuentran en el orden esperado

4. Algoritmos Golosos

4.1. Descripción del problema

El problema que se nos presenta ahora es similar al que enfrentamos con programación dinámica, donde tenemos que seleccionar el subconjunto de actividades, con sus beneficios asociados que no se solapan en el tiempo y que maximizan el beneficio total obtenido. A diferencia del problema anterior, en este caso todas las actividades tienen el mismo nivel de beneficio, 1.

4.2. Estrategia Golosa

Para enfrentar el problema se nos propuso una estrategia golosa, que al igual que los algoritmos de esta naturaleza utiliza el criterio de selección más sencillo pero correcto para que la solución alcance su resultado óptimo.
En este caso la estrategia consiste en elegir siempre la actividad que más temprano termina que no

se solape con actividades elegidas previamente. Básicamente querríamos conseguir siempre la actividad que más temprano termina de entre las actividades que empiezan en el momento siguiente al final de la última actividad elegida (o desde el comienzo, en el caso de la primera actividad).
Analizando el enunciado y la estrategia propuesta llegamos a la conclusión de que si todas las actividades aportan el mismo beneficio, el problema terminaría traduciendose a encontrar el subconjunto del conjunto de actividades de mayor cardinal tal que las actividades pertenecientes no se solapan en el tiempo. Es decir, independizandonos del beneficio de cada actividad. Sin embargo, antes de implementar la idea tuvimos que respondernos, **¿realmente la estrategia es correcta?**.

Sea B(i) el beneficio de tomar i actividades. Es trivial ver que, como el beneficio de todas las actividades es 1, B(i) = i y B(i) < B(i+1). Por lo que deducimos que el beneficio es creciente en relación a la cantidad de actividades que tomemos. Entonces, si tomamos la máxima cantidad de actividades que no se solapan, maximizaríamos el beneficio.

Ahora tenemos que ver por qué la forma de selección de actividades maximiza la cantidad de actividades que podemos seleccionar. La proposición P(i) es dado el subconjunto $\{A_0, A_1...A_i\}$ de A, pero ahora con la particularidad de que las actividades estan ordenadas crecientemente según su tiempo de finalización, utilizando la estrategia en este subconjunto, se maximiza la cantidad de actividades elegidas.

Caso base: i = 0 Para este caso, nuestro subconjunto de actividades es $\{A_0\}$ y con esta estrategia, tomamos la actividad A_0 y esto máximiza la cantidad de actividades del subconjunto, ya que tomamos todas las actividades del mismo.

Paso inductivo: $\forall i | 0 \leq i < n$, ¿P(i) $\Rightarrow$ P(i+1)?

Por hipotesis inductiva, sabemos que tenemos un subconjunto AC $\in \{A_0, A_1...A_i\}$ que máximiza la selección de actividades. Sea A_k la última actividad agregada en AC, es decir la que más tarde termina de estas. Si $s_{i+1} > t_k$, podemos agregar A_{i+1} a AC, llamemoslo AC'. Como AC era un conjunto que maximizaba la cantidad de actividades, AC' va a ser maximal porque A_{i+1} es compatible con todos los elementos de un subconjunto que era máximal.

Por otra parte, si $s_{i+1} \leq t_k$, no podemos agregar A_{i+1} a AC. Ahora hay probar por qué AC sigue siendo el subconjunto con más actividades compatibles entre $\{A_0, A_1...A_{i+1}\}$. Por hipotesis inductiva sabemos que el conjunto es maximal, además sabemos que, por como están dispuestas las actividades, A_k termina lo antes posible, luego, si existiese algún subconjunto AC* que incluya A_{i+1} que tenga más actividades, se deberían eliminar de AC las actividades A_k y todas las que no son compatibles con A_{i+1} y agregar otras de las actividades pertenecientes a $\{A_0, A_1...A_i\}$ que si lo sean, pero como A_{i+1} es la última que finaliza, ninguna de estás actividades es compatible con A_{i+1} por lo que es un absurdo. Por lo que si A_{i+1} no es compatible con A_k, tomando el mismo AC, el subconjunto de actividades que seleccionamos sigue siendo maximal. De esta forma demostramos que con esta estrategia, P(i) $\Rightarrow$ P(i+1).
Quedó demostrado entonces que la estrategia funciona para conjuntos $\{A_0, A_1...A_i\}$, y en particular será correcta para $\{A_0, A_1...A_n\}$.

Por último notemos que esta estrategia solamente podría funcionar en el caso de que todas las actividades tengan el mismo beneficio, porque en caso contrario supongamos que en el caso del

paso inductivo, la siguiente actividad que termina más temprano se solapa con las siguientes. Si entre esas siguientes existe una actividad de beneficio mayor, como estamos tomando las que finalizan antes, nos quedaríamos con la primera actividad que aparece y el beneficio no sería máximo. La mejor forma de demostrar esto es con un sencillo contraejemplo. Supongamos el siguiente conjunto de actividades con sus beneficios asociados.

Inicio	Fin	Beneficio
1	5	3
1	2	1
3	4	1

Notemos que siguiendo nuestra estrategia, elegiríamos la actividad 2, luego la 3 y finalmente no habría más actividades que seleccionar, ya que la 1 se solapa con todas, pese a ser la última en terminar. Sin embargo la sumatoria de los beneficios de las actividades 2 y 3 da como resultado 2, que es menor al beneficio que se obtendría sencillamente eligiendo la actividad 1, que sería 5. Por lo tanto podemos afirmar que esta estrategia no funcionaría en el caso en que las actividades tengan beneficios distintos asociados.
Sin embargo, como para este problema estamos suponiendo que todas las actividades tienen el mismo beneficio, entonces lo que nos conviene es que el subconjunto de actividades que hagamos sea maximal, ya que de esa forma el beneficio sería de $máx_{1 \leq k \leq n}\{\sum \mathrm{A}_i \in AC \text{ b(A)} — \forall A_i y A_j \in AC, i < j, t_i < s_j\}$ donde b(A) es una constante.

4.3. Algoritmo

Algoritmo 3 Algoritmo Goloso

Input : Contamos con un vector Actividades con las actividades posibles representadas por su tiempo de comienzo y fin y un vector Sched de $2n + 1$ posiciones, inicializadas en -1, que indica el horario de finalización de cada actividad. Además creamos un vector Solución para ir agregando las actividades elegidas.

Output: Máximo beneficio entre $\{A_i..A_{n-1}\}$, o sea la mayor cantidad de actividades que se pueden hacer

```
1:  function ScheduleGoloso → ℕ
2:      for actividad ∈ Actividades do
3:          if Sched[actividad] == −1 ∨ Sched[actividad.fin].inicio < actividad.inicio then
4:              Sched[actividad.fin] ← actividad;
5:          end if
6:      end for
7:      Se crea vector ordenFin;
8:      for i ← 0, ..., |Sched| − 1 do
9:          if Sched[i] ≠ −1 then
10:             AgregarAtrás(Sched[i], ordenFin);
11:                                  ▷ Las actividades quedan ordenadas por Sched[i].fin
12:         end if
13:     end for
14:     inicializo res = 1, Agrego la primera actividad de ordenFin a Solu;
15:     for actividad ∈ ordenFin do
16:         if actividad.comienzo > ultimo(ordenFin).fin then
17:                                  ▷ Condición de colisión de actividades
18:             AgregarAtrás(actividad, Solu);
19:             res ← res + 1;
20:         end if
21:     end for
22:     return res
```

Basandonos en la experiencia de la resolución de los problemas, podemos decir que el algoritmo goloso es un acercamiento a la solución más directa e intuitiva que aquella que usamos para el problema 3. Sin embargo, podemos tomar pros y contras sobre como fue el proceso de pensamiento para cada uno de estos, ya que más allá de que la estrategia golosa se nos fue provista desde un principio, la función recursiva que utilizamos para la programación dinámica fue un recurso pensado por nosotros mismos. Esto hizo que implementar el algoritmo en C++ no sea una tarea tan ardua, más allá del trabajo de preprocesamiento, que es la que más tiempo nos llevó. Sin embargo, la implementación final del algoritmo goloso fue más directa y sencilla.

4.4. Resultados

Nombre de Instancia	Resultado Obtenido	Tiempo (seg.)
interval_instance_1	3	0.00
interval_instance_2	2	0.00
interval_instance_3	2	0.00
interval_instance_4	1	0.00
interval_instance_5	2747	0.00
interval_instance_6	1539	0.00
interval_instance_7	592	0.00
interval_instance_8	27776	0.02
interval_instance_9	15646	0.02
interval_instance_10	5708	0.02
interval_instance_11	276866	0.17
interval_instance_12	155455	0.17
interval_instance_13	57523	0.17

Cuadro 3: Las instancias se corrieron en una computadora con procesador AMD Ryzen 5 3500U.