DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Informe: Trabajo Práctico 1

Algoritmos y Estructuras de Datos III

Integrante	LU	Correo electrónico
Casado Farall, Joaquin	072/20	joakinfarall@gmail.com
Chumacero, Carlos Nehemias	492/20	chumacero2013@gmail.com
Reyna Maciel, Guillermo José	393/20	guille.j.reyna@gmail.com
Sánchez Sorondo, Marco	708/19	msorondo@live.com.ar
Vitali, Lucas	278/20	lucasvitali001@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
http://www.exactas.uba.ar

Índice

Introducción

En este informe vamos a detallar cómo resolvimos los problemas planteados para el primer Trabajo Práctico de la materia Algoritmos y Estructuras de Datos III. Los problemas son sobre técnicas de diseño de algoritmos. Se incluye en cada punto el enunciado del problema. Al final del documento se puede encontrar un apéndice con los resultados de nuestra experimentación del tiempo de ejecución de nuestras implementaciones.

El código de nuestra implementación, de la experimentación, y de éste informe se pueden encontrar en el repositorio git en el siguiente link: `https://github.com/guillereyna/TP1-algo3`. Las instrucciones de compilación se encuentran el archivo `README.MD` en el directorio raíz del repositorio.

En el directorio `src/` del repositorio se puede encontrar el código fuente y el ejecutable de nuestra implementación de cada ejercicio. Cada ejercicio tiene un archivo `.cpp` y un binario ejecutable propio. Los ejecutables toman el input de la instancia problema por entrada estándar en el formato provisto por la cátedra e imprimen por la terminal el resultado y la solución en el formato en el que están los outputs de referencia del enunciado.

Problema 1: Clique más influyente

Consideremos una red social similar a Facebook, en la que cada actor (persona, organización, etc.) tiene asociado un valor que determina su poder de influencia sobre la red. Una clique es un subgrupo de actores que son todos amigos entre sí. En este ejercicio resolvemos el problema de la *clique más influyente*: dada la descripción de una red social, queremos encontrar una clique cuya influencia sea máxima (la suma de la influencia de sus actores).

Formalmente, una red social es una tripla $G = (V, E, p)$ tal que:

- $V = 1, ..., n$ es el conjunto de actores, cada uno de los cuales se identifica con un número,
- E es el conjunto amistades y está formado por subconjuntos de V que tienen exactamente dos actores, y
- $p : V \to N > 0$ es la función que denota el poder de influencia de cada actor.

Por simplicidad, vamos a escribir vw para denotar al conjunto v, w para todo $v, w \in V$. Decimos que v y w son amigos cuando $vw \in E$ y que son no-amigos en caso contrario. Un conjunto $Q \subseteq V$ es una clique de G cuando v y w son amigos para todo $v, w \in Q$. El poder de influencia de una clique Q es $p(Q) = \sum_{v \in Q} p(v)$. Las cliques de influencia máxima de G son aquellas cliques de G cuyo poder de influencia es $\max\{p(Q) \mid Q \text{ es clique de } G\}$. Dada una red social $G = (V, E, p)$, el problema de la clique más influyente consiste en determinar una clique de influencia máxima.

1. Backtracking

1.1. Diseño de algoritmo

Diseñar un algoritmo de backtracking que, dada una red social G, resuelva el problema de la clique más influyente de G. El algoritmo debe estar basado en las siguientes ideas y cada nodo debe procesarse en $O(n^2)$ tiempo:

- *Todas las soluciones parciales del árbol de backtracking son cliques de G.*
- *Cada nodo del árbol de backtracking representa un conjunto de soluciones (cliques) $\mathcal{Q}$ que se obtienen al visitar las hojas de su subárbol; todas las cliques de $\mathcal{Q}$ contienen a la solución parcial (clique) Q. Intuitivamente, en el proceso de extensión ya se tomó la decisión de que cada actor de Q esté en las soluciones que se generan a partir de este nodo.*
- *Además de Q, cada nodo del árbol de backtracking tiene asociado un conjunto de actores K que es disjunto a Q. Intuitivamente, K contiene los actores sobre los que aún no se tomó ninguna decisión en el proceso de extensión. A partir de ahora, hablamos del nodo (Q, K) para referirnos a aquel nodo correspondiente a una solución parcial Q cuyo conjunto asociado es K.*
- *En cada nodo (Q, K), el conjunto K satisface dos propiedades invariantes. Por un lado, cada $v \in K$ es amigo de todos los actores de Q, lo que implica que $Q \cup v$ es una clique de G. Por otro lado, cada $v \in K$ tiene un no-amigo en K , lo que implica que al menos una clique de G contiene a Q pero no a v.*
- *La extensión del nodo (Q, K) genera dos subárboles a partir de un actor $v \in K$. En uno de estos subárboles, cada hoja se corresponde a una clique que contienen a $Q \cup v$. En el otro subárbol, cada hoja representa una clique que contiene a Q y no a v.*

- *Para reducir el espacio de búsqueda, se poda cada nodo (Q, K) tal que $p(Q) + \sum_{v \in K} p(K) \leq p(S)$ para la clique S de mayor influencia encontrada hasta el momento.*
- *El árbol se recorre recursivamente, avanzando primero en profundidad; pensar qué hijo conviene visitar primero en cada nodo.*

Nuestro algoritmo de backtracking para resolver este problema recibe como parámetros dos conjuntos, Q y K, representados como vectores. Tomamos como caso base a las instancia de (Q, K) tal que K este vacío. En este caso no hay elemento en K sobre los cuales decidir si incluir en la clique Q. Luego se procesa Q, es decir, se calcula su influencia con la mayor influencia anteriormente vista (variable global) y de ser mayor se almacena la influencia de Q y a Q en una variable global.

En el caso recursivo, sea v el ultimo elemento de K, se llama recursivamente a nuestro algoritmo para las instancias en donde se agrega v a Q y no se agrega v a Q, siempre sacando a v de K. En ambos casos, luego de sacar a v de K y agregar o no v a Q, se chequean ambos invariantes de K, es decir:

1. en K no haya elementos que no sean amigos de todos en Q
2. en K no haya elementos que sean amigos de todos en K.

En el primer caso si se agrego a v, se puede asumir que K cumplía el invariante, por lo que cumplía (1) antes de agregar a v, por lo que solamente hace falta eliminar a los elementos en K que no son amigos de v, lo cual se hace en $O(|K|) < O(n)$. Si no se agrega a v a Q este invariante no hace falta chequear por que ya se cumplía.

En el segundo caso, por cada elemento en K tenemos que ver su relación con los demás elementos en K, y eliminarlo de K y agregarlo a Q en caso de que sea amigos de todos en K. Como eliminar un elemento de K se puede hacer en $O(|K|)$ (copiar K sin el elemento) y agregar a Q se puede hacer en $O(1)$ (agregar atrás del vector), la complejidad dominante seria la de realizar la comparación entre todos los elementos de K, que es $O(\mathrm{K}^2) < O(n^2)$.

Antes de hacer las llamadas del caso recursivo se aplica la poda dada, para la cual se calcula la suma de los elementos de Q y K, los cual es O(—Q—+—K—) ¡O(n).

Cabe aclarar que luego de cada llamado recursivo se restaura los valores de Q y K que se pasan por parámetros, a través de una copia previamente hecha, lo cual es $O(|Q| + |K|) = O(n)$.

Finalmente para ejecutar el algoritmo desde el principio, se pasa en el lugar de Q un conjunto vació y en el lugar de K el conjunto V de actores de la red. Luego de la ejecución de la función se imprime a la salida la máxima influencia encontrada y su Q correspondiente que fueron almacenados en variables globales.

1.2. Ordenamientos

Considerar al menos dos opciones para ordenar a los actores antes de ejecutar el algoritmo: de mayor a menor influencia y de menor a mayor influencia.

Testeamos ambos casos.

1.3. Experimento de ordenamiento

Realizar un experimento con los casos de test de ambas opciones y reportar los resultados, justificando los mismos a través de analizar los efectos de la poda.

La poda del punto uno corta las ramas de backtracking en donde sin importar que actores de K se sumen a Q, Q no va a tener una influencia mayor a una ya anteriormente vista.

Luego si en primer lugar se considera agregar o no al actor de menor influencia posible (ordenar de menor a mayor), la clique Q va a estar formada en primer lugar por actores con la menor influencia posible. En dicho caso se encontraría rápido la clique más influyente si existiesen cliques grandes de actores con poca influencia que superen a cliques pequeñas con actores de mucha influencia.

En cambio, si se considera agregar o no al actor de mayor influencia posible (ordenar de mayor a menor), la clique Q va a estar formada primero por actores con la mayor influencia posible. Por lo tanto, al contrario del caso anterior, se encontraría la clique mas influyente mas rápido si la misma tiene actores con mucha influencias.

Experimento:

instancia	mayor	menor
bock200_2:	0.19	0.59
bock200_3:	0.59	6.21
bock200_4:	1.78	41.69
c-fat-200_1:	0.00	0.03
c-fat-500_2:	0.01	0.01
c-fat-500_5:	0.01	0.07
c-fat-500_10:	0.05	0.18
hamming8_4:	3.76	133.32
MANNa9:	0.80	0.93

Finalmente, como muestra el experimento, ordenar a los actores de mayor a menor según influencia parece ser mas conveniente que de la otra manera. Pensamos que esto es así por que es poco probable tener una clique grande que no contenga ningún elemento con influencia grande y a la vez ser lo suficientemente grade como para ser la clique de mayor influencia. Por lo tanto si se ordenan los actores de mayor a menor influencia se encuentra las cliques mas grandes primero lo que hace mas eficiente la poda.

1.4. Operación de extensión

La operación de extensión define una función recursiva cuya descripción formal puede ser compleja. Sin definir formalmente esta función, explicar por qué la misma no tiene la propiedad de superposición de subproblemas.

Dados dos conjuntos Q y K de tamaño acotado tal que $Q \leq |r|$ y $K \leq |s|$, es decir $r + s = n$ vértices en el grafo:

- Se puede elegir o no poner uno de los r elementos en Q y uno de los s elementos en K. Luego la cantidad de subproblemas es $O(2^r \cdot 2^s)$
- En cada llamada recursiva se evalúan los casos en que al elemento que saco de K, lo agrego o no a Q. Luego la cantidad de llamadas recursivas es $O(2^s)$

Como la cantidad de subproblema $O(2^r \cdot 2^s) \gg O(2^s)$ la cantidad de llamados recursivos, este problema no cumple con la propiedad de superposición de subproblemas.

2. Podas

Sea $= (V, E, p)$ 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 está definido como $\overline{p} = \text{máx}\{p(v) \mid v \in I\}$. En este ejercicio vamos a implementar una poda para el problema de la clique más influyente a partir de las influencias de los conjuntos independientes.

2.1. Conjuntos independientes

Sea (Q, K) un nodo del árbol de backtracking y sea $I_1, ..., I_k$ una partición de K en conjuntos independientes. Observar que $p(Q) + \sum_{i=1}^{k} \overline{p}(I_i) \geq p(Q \cup Q')$ para toda clique $Q' \subseteq K$. Explicar por qué.

Explicación: Dada Q' una clique en K, supongamos que queremos particionar K en conjuntos independientes... Ocurrirá entonces que para particionar los elementos de Q' debere tener AL MENOS $\#Q'$ particiones pues Q' es una clique y por lo tanto para 'independizar' sus vértices deberé tomar a cada uno por separado, para que vayan todos a conjuntos independientes disjuntos.

Esto quiere decir que del lado izquierdo de la inecuación tendré (en el peor de los casos) $\#Q'$ conjuntos independientes cuyo poder de influencia estará acotado inferiormente por el poder de influencia de cada elemento de Q' incluido en el respectivo conjunto (puede ocurrir que hayan elementos que no estén en Q' y que no tengan relaciones con alguno de los elementos en Q' y que tengan poder de influencia mayor y que por lo tanto el poder de influencia del conjunto exceda al valor del elemento en Q' del conjunto independiente).

2.2. Algoritmo

La idea detrás del algoritmo que resuelve la obtención de una partición es:

1. Ordenar K decrecientemente en funcion de las influencias de los actores.

2. Para cada elemento en K, buscar un conjunto independiente no vacío donde no tenga amigos e insertarlo.

3. Si no existe tal conjunto, crear uno nuevo que contenga al elemento actual.

Al ordenar la clique por influencia, ésto me garantiza que siempre que encuentre un conjunto no vacío en el paso 2, se anule el poder de influencia del elemento a insertar en el lado izquierdo de la inecuación (pues todos los elementos anteriormente insertados tienen una influencia mayor). Esto reduce significativamente el $\sum_{i=1}^{k} \overline{p}(I_i)$ y por lo tanto 'fuerza' a que se encuentren casos de poda.

Complejidad: El paso 1 del algoritmo se implementa con la función de ordenamiento estándar de C++, que en peor caso es $O(n \cdot logn)$. El paso 2 del algoritmo tiene 3 bucles:

1. El principal, que itera por K ya ordenado.

2. El segundo, embebido en el 1, que itera por los conjuntos independientes.

3. El tercero, embebido en el 2, que itera por los elementos de cada conjunto independiente.

Si bien puede darse la ilusión de que éste paso tenga complejidad cúbica (por haber 3 ciclos), en realidad el segundo y tercer ciclo iteran linealmente a lo largo de todos los elementos de los conjuntos independientes, y la suma de todos los elementos de los conjuntos independientes está acotada por la cantidad de elementos en K (pues son partición de la misma). Luego el paso 2 tiene complejidad $O(n^2)$. El paso 3 del algoritmo tiene complejidad constante (amortizado) pues se trata de una inserción de un elemento en un vector vacío y una posterior inserción de éste vector en la partición.

Nota: Asumimos que agregar un nuevo elemento al final de un vector es O(1) -amortizando-.

2.3. Poda

Para implementar ésto, basta con computar la suma de las influencias de Q, sumarla con la suma de los poderes de influencia de la partición obtenida (el primer elemento de cada conjunto, pues están en orden) y chequear que sea menor al poder de influencia máximo hasta el momento (ya calculado). Caso contrario se corta la recursión.

2.4. Experimentación

Los resultados del experimento se encuentran en el apéndice, en la sección 5.2. Los resultados muestran lo contrario a lo esperado: el tiempo de ejecución de nuestra implementación con podas es mayor a nuestra implementación anterior. Esto puede significar que nuestro algoritmo de partición es muy lento, y deberíamos encontrar uno mejor.

Problema 2: Selección de actividades

Dado un conjunto de actividades $A = \{A_0, ..., A_{n-1}\}$ y una función de beneficio $b : A \to \mathbb{N}$, el problema de selección de actividades consiste en encontrar un subconjunto de actividades S cuyo beneficio $b(S) = \sum_{A \in S} b(A)$ sea máximo de 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 su momento final. Suponemos que $0 \leq s_i < t_i \leq 2n$ para todo $0 \leq i < n$.

A los subconjuntos S de actividades que no se solapan en el tiempo los llamamos compatibles.

3. Programación dinámica

3.1. Solución con problema de cliques

Describir cómo se puede resolver el problema de selección de actividades utilizando cualquier solución al problema de la clique más influyente.

El problema de la clique más influyente se puede modelar con un grafo cuyos nodos representan los *actores*, cuyo peso es su *influencia*, y cuyas aristas representan las *relaciones de amistad* con otros actores.

El problema de la clique más influyente se abstrae entonces a encontrar el subgrafo tal que todos los nodos del subgrafo están relacionados por aristas a todos los otros nodos del subgrafo (i.e. todos los actores son amigos entre sí: una clique) y cuya sumatoria de pesos sea máxima entre todos los grafos que cumplen la anterior condición (i.e. la clique con mayor influencia).

Para describir cómo el problema de selección de actividades es equivalente al problema de cliques, basta con modelarlo como un problema en un grafo con la misma solución.

Proponemos representar a cada actividad como un nodo en un grafo, cuyo peso representa su beneficio y que está conectado por aristas a todas las otras actividades con las que es compatible. Por lo tanto, cualquier subconjunto de actividades compatibles va a ser representado en el grafo como un subgrafo de nodos que están conectados por aristas con todos los otros nodos en el subgrafo. Encontrar el subconjunto de actividades compatibles con mayor beneficio es entonces equivalente a encontrar al subgrafo completo de peso máximo.

Si tenemos una solución para el problema de la clique más influyente, tenemos una solución para el problema de encontrar el subgrafo con mayor peso de un grafo, y por lo tanto la solución al problema de selección de actividades.

3.2. Función recursiva

Supongamos que A está ordenado por orden de comienzo de la actividad, i.e., $s_i \leq s_{i+1}$ *para todo* $1 \leq i < n$. *Escribir una función recursiva* $b : 0, ..., n \to \mathbb{N}$ *tal que:*

- $b(i)$ *denota el máximo beneficio entre todos los subconjuntos de actividades compatibles incluidos en* $A_i, ..., A_{n-1}$ *(Notar que* $b(n) = 0$ *por definición.)*
- b *satisface la propiedad de superposición de subproblemas.*

Proponemos una función $b(i)$ que en el caso recursivo devuelve el mayor entre $b(i+1)$, es decir el máximo beneficio en el caso en que la actividad A_i no se elige, y la suma del beneficio de A_i con el beneficio máximo alcanzable considerando las actividades posteriores compatibles con A_i, en el caso en que se elige A_i.

$$b(i) = \begin{cases} 0 & i = n \\ max\{b(i+1), b(p_i) + b_i\} & \text{en otro caso} \end{cases}$$

Donde p_i es el índice de la primera actividad compatible con la actividad de índice i, con $p_i > i$ (utilizando fuertemente que A está ordenado de menor a mayor tiempo de inicio).

En cada llamado recursivo, se consideran ambos el caso en que no se selecciona una actividad y el caso en el que se selecciona la actividad, para una cantidad de llamados recursivos en el orden de 2^n. Considerando que $0 \leq i \leq n$, sabemos que hay $n + 1$ valores posibles de b, satisfaciendo la propiedad de superposición de problemas.

3.3. Demostración

Sea $A = \{A_0, ..., A_{n-1}\}$ nuestro conjunto de actividades ordenadas por comienzo tal que $s_i \leq s_{i+1}, \forall 1 \leq i < n$

Siendo $G_i = \{G \subseteq \{A_i, ..., A_{n-1}\} / \forall (A_j, A_k) \Rightarrow t_j < s_k, \;\; i \leq j < k < n, \forall A_j, A_k \in G\}$ el espacio de soluciones, es decir los subconjuntos de actividades a partir de i que no se solapan.
Será $g_i = max\{\sum_{i \in G} b_i / G \in G_i\}$ el maximo beneficio que se pueda obtener de un subconjunto de actividades $\in$ G

Paso Inductivo:
$HI : b(k) = g_k, \forall 0 < h \leq k \leq n$
$Qvq : b(h-1) = G_{h-1}$

$b(h-1) = max\{b(h), b(P_{h-1} + b_{k-1}\} \Leftrightarrow$
$b(h-1) = max\{g_h, g_{P_{h-1}} + b_{h-1}\}$
Luego:

- Si A_{h-1} es compatible con $G_h \Rightarrow g_{h-1} = g_h + b_{h-1} = max\{g_h, g_h + b_{h-1}\} = max\{g_h, g_{P_{h-1}} + b_{h-1}\}$
- Si A_{h-1} no es compatible con $G_h \Rightarrow g_{h-1} = max\{g_h, g_{P_{h-1}} + b_{h-1}\}$

3.4. Implementación top-down

Implementar un algoritmo de programación dinámica top-down para el problema de selección de actividades. El algoritmo debe computar el beneficio de la solución óptima en $O(n)$.

Para no repetir cálculos de subproblemas, creamos un vector `memo` de memoización de tamaño $n+1$, con `memo[i]` = b(i), donde guardamos cada valor la primera vez que es computado. Inicializamos el vector con -1 en cada posición excepto el caso base `memo[n]`, que se inicializa en $b(n) = 0$.

El problema principal para implementar la función anteriormente planteada en tiempo $O(n)$ es que no se conoce a priori cuál es la próxima actividad compatible p_i para cualquier actividad A_i.

Para lograrlo creamos un vector `p` de tamaño $2n+1$ donde cada posición j contiene el índice de la primer actividad que comienza en el tiempo j o mayor. Entonces, para cualquier tiempo j, `p[j]` es el índice de la primer actividad que inicia a partir de ese tiempo. Para llenar los valores en el vector primero recorremos el conjunto de actividades y para cada actividad A_i con tiempo de inicio s_i asignamos `p`$[s_i]$ `= i`, en tiempo $O(n)$.

Para llenar el resto de las posiciones, asignamos `p[2n] = n` (representa que no hay más actividades, dado que $b(n) = 0$) y recorremos en $O(2n)$ el vector comenzando desde la posición `j = 2n-1` en orden decreciente con la siguiente regla: si `p[j]` no está definido (en nuestra implementación, si es igual a -1), asignamos `p[j] = p[j+1]`.

Entonces, para cualquier actividad A_i con tiempo de fin t_i, `p`$[t_i$`+1]` es el índice de la primer actividad compatible con A_i.

La implementación de la función `b(i)` queda de la siguiente forma:

- Si `memo[i]` no está definido, asigna a `memo[i]` el máximo entre `b(i+1)` y b_i `+ b(p[`$t_i + 1$`])`.
- Devuelve `memo[i]`.

El caso base queda cubierto por la inicialización de `memo[n] = 0`. Como cada subproblema se calcula una única vez en $O(1)$, la función tiene tiempo en orden $O(n)$.

Para obtener la solución óptima para un conjunto basta con llamar a b con parámetro 0.

La complejidad del algoritmo es de $O(n) + O(2n) + O(n) = O(n)$.

3.5. Implementación bottom-up

Implementar una versión bottom-up del algoritmo anterior, indicando claramente el orden de cómputo de cada subinstancia.

Las estructuras `memo` (con `memo[n] = 0`) y `p` son idénticas a la implementación *top-down*, y `p` se computa de la misma manera. Para llenar los valores de `memo`, se recorre `memo` en orden decreciente desde la posición `i = n-1`. En cada iteración, se asigna en $O(1)$ `memo[i] = max(memo[i+1],` b_i `+ memo[p[`t_i`+1]])`. Como $p_i > i$, esta expresión nunca se indefine. Finalmente, se devuelve `memo[0]`.

Se itera n veces una operación en $O(1)$, por lo que el algoritmo bottom-up es $O(n)$

3.6. Reconstrucción de solución

Implementar un algoritmo que reconstruya un conjunto de actividades compatibles de beneficio máximo.

La solución de reconstrucción del subconjunto de actividades se basa en reutilizar nuestra estructura de memoización, en este caso el vector `memo`.

Con una variable `i` inicializada en 0, recorreremos el vector `memo` mediante un `while` donde la condición será que `i` sea menor a n (tamaño del conjunto de tareas).

En cada iteración preguntaremos si el beneficio guardado en `memo[i]` es mayor a `memo[i+1]`, ya que si esto se cumple significa que la tarea `i` se tuvo en cuenta y aumento el beneficio, por ende hacemos print de `i` (el índice de la tarea que estamos observando) y actualizamos `i` con el valor de la tarea más cercana que no se solape, que lo obtenemos del vector precalculado `p`.

En caso contrario, si `memo[i]` no es mayor que `memo[i+1]` solo incrementamos `i` en 1, ya que significa que la tarea actual no se tuvo en cuenta, por ende pasamos a la próxima.

Una vez tomada la decisión, se seguirá con el recorrido de memo hasta cumplir la condición del `while`.

4. Algoritmo goloso

Considerar la siguiente estrategia golosa para resolver el problema de selección de actividades compatibles para el caso en que $b(A) = 1$ para toda actividad $A \in A$: elegir la actividad cuyo momento final sea lo más temprano posible, de entre todas las actividades que no se solapen con las actividades ya elegidas.

4.1. Demostración

Sea $S \subseteq \{A_0, ..., A_{n-1}\}$ donde $b_i = 1\ \forall i$ y $b(S) = \sum_{a \in S} b(a)$.

Defino $Y = E_0, ..., E_{n-1}$ una secuencia de elecciones (i.e. si el elemento i está en una solución al problema de selección de actividades S_Y, o no) con $E_i \in \{0, 1\}\ \forall i$. Tengo que $A_i \in S_Y \iff E_i = 1,\ \forall i$.

Defino la secuencia de elecciones parcial $Y_h = E_0, ..., E_h$ con $0 \leq h \leq n-1$. Luego, $Y = Y_{n-1}$.

Lo que queremos demostrar es que el algoritmo goloso elige las actividades óptimas para cualquier subsecuencia de elecciones (i.e. el beneficio goloso es por lo menos igual al óptimo):

Qvq $\overline{b}(S_{Y'}) = max\{b(S_Y)\}$ siendo Y' las elecciones del algoritmo goloso
Qvq $\overline{b}(S_{Y'_h}) = max\{b(S_{Y_h})\}\ \forall h$

Inducción en h:

Caso base: $h = 0$

- Y'_0 tiene a $E'_0 = 1$, ya que es compatible con el conjunto vacío de actividades y es de tiempo de finalización más temprano, entonces $\overline{b}(S) = 1$.
- Dada S_{Y_0}, la suma máxima de beneficios de actividades compatibles es 1, ya que hay una única actividad seleccionable, entonces $b(S) = 1$.

Paso inductivo:

HI: $\overline{b}(S_{Y'_h}) = max\{b(S_{Y_h}\}$
Qvq: $\overline{b}(S_{Y'_{h+1}} = max\{b(S_{Y_h+1})\}$

$\overline{b}(S_{Y'_{h+1}} = \overline{b}(S_{Y'_h}) + b_{A_{h+1}} \cdot E'_{h+1} \overset{HI}{=} max\{b(S_{Y_h})\} + b_{A_{h+1}} \cdot E'_{h+1}$

- Si A_{h+1} no es compatible, entonces $E'_{h+1} = 0 \implies \overline{b}(S_{Y'_{h+1}}) = max\{b(S_{Y_h})\}$:
  - Como A_{h+1} no es compatible y por HI $S_{Y'_h}$ es óptimo, en la política óptima $E_{h+1} = 0$ luego $max\{b(S_{Y_{h+1}})\} = max\{b(S_{Y_h})\}$
- Si A_{h+1} es compatible:
  - $E'_{h+1} = 1$ ya que es la actividad compatible con menor tiempo de finalización, luego $\overline{b}(S_{Y'_{h+1}}) = max\{b(S_{Y_h})\} + 1$.
  - $$\left.\begin{array}{l} E_{h+1} = 0 \implies \text{en el óptimo hasta h } max\{b(S_{Y_h})\} \\ E_{h+1} = 0 \implies \text{en el óptimo hasta h } max\{b(S_{Y_h})\} + 1 \end{array}\right\} = max\{b(S_{Y_h})\} + 1$$

Entonces, $max\{b(S_{Y_{h+1}})\} = max\{b(S_{Y_h})\}$. □

4.2. Contraejemplo para caso general

Mostrar que la solución no es correcta cuando $b(A)$ no es necesariamente 1 para todo A.

Como contraejemplo, considerar el siguiente conjunto A de actividades:

i	$[s_i$ - $t_i]$	b_i
0:	1 - 2	1
1:	1 - 8	500
2:	3 - 4	1
3:	5 - 6	1

El algoritmo goloso devuelve 3 como óptimo, eligiendo las actividades A_0, A_2, y A_3. La selección óptima es A_1, con beneficio 500.

4.3. Implementación

Implementar un algoritmo basado en la idea anterior cuya complejidad temporal sea O(n).

La idea fundamental de nuestro algoritmo es la siguiente:

1. Ordenar el conjunto de actividades de menor a mayor tiempo de fin en $O(n)$.
2. Recorrer en $O(n)$ el conjunto de actividades en orden, en cada iteración eligiendo en $O(1)$ la actividad si es compatible con las actividades ya elegidas.

Cada vez que se elige una actividad, va a ser agregada al final de un vector de solución que al final de la ejecución va a ser impreso por pantalla.

Los puntos principales que debemos asegurar con nuestro diseño son el ordenamiento del conjunto en $O(n)$ y la decisión de la compatibilidad de una actividad en $O(1)$.

Para resolver el primer problema, creamos un vector `Schedule` de $2n+1$ elementos, donde cada posición j representa las actividades con tiempo de fin j. Cada elemento es inicialmente un vector vacío. Para obtener la información que necesitamos, recorremos en $O(n)$ el conjunto de actividades, y agregamos cada actividad A_i al final del vector `Schedule[`t_i`]` (asumiendo la última operación es $O(1)$).

Para resolver el problema de decisión de compatibilidad, introducimos una variable `h` que guarda el tiempo de fin de la última actividad que fue elegida. Para saber si una actividad A_i es compatible con el subconjunto de actividades ya elegidas, basta con saber si el tiempo de inicio s_i es mayor al tiempo de fin de la última actividad elegida, es decir si `h` $< s_i$.

Finalmente, para recorrer el conjunto de actividades en orden de tiempo de fin y devolver el beneficio óptimo, empleamos una estrategia recursiva: la función `actividadesGoloso(Schedule, h)`, dado un `Schedule` y un último tiempo ocupado `h`, intenta elegir la próxima actividad A_i compatible en `Schedule` y devuelve 0 si no hay más actividades compatibles o 1 + `actividadesGoloso(Schedule,` t_i`)` en otro caso.

Para elegir la próxima actividad compatible, definimos la función `elegirGoloso(Schedule, h)` que recorre `Schedule` en orden creciente iniciando en `Schedule[h+1]`, recorriendo en cada iteración i el vector de actividades que terminan en el tiempo `Schedule[i]` hasta encontrar la primer actividad que tenga tiempo de inicio s_i mayor a `h`, y devuelve t_i y agrega i al vector de solución. Si no hay más actividades compatibles, devuelve -1.

La complejidad es $O(n)$ porque cada elemento de `Schedule` es visitado una única vez en total por la función `elegirGoloso`, ya que la misma recorre `Schedule` desde algún índice `h` hasta encontrar la primer actividad compatible y termina inmediatamente, y la próxima vez que es llamada, va a ser con un parámetro estríctamente mayor a `h`. Nunca se va a considerar más de una vez ninguna actividad. Adicionalmente, todas las demás operaciones son comparaciones, asignaciones, y agregar a vectores, en $O(1)$. Como `Schedule` contiene en total n actividades (es decir, hay n actividades a ser consideredas en $2n + 1$ vectores de actividades, de la cual por lo menos la mitad van a estar vacíos) esto toma tiempo $O(n)$.

Para obtener el óptimo beneficio de un conjunto de actividades A de beneficio 1, llamamos a la función `actividadesGoloso` con parámetros el vector `Schedule` generado a partir de A y `h = 0`.

La complejidad del algoritmo es $O(2n + 1) + O(n) + O(n) = O(n)$.

4.4. Comparación con programación dinámica

Comparar este algoritmo con el de programación dinámica, marcando cuál es más simple de implementar.

La idea fundamental del algoritmo goloso de ordenar el conjunto y recorrerlo surgió rápidamente, y las estructuras necesarias para respetar la complejidad requerida fueron decididas en tan sólo un par de iteraciones de implementación y salieron naturalmente a partir de la idea inicial.

En comparación, el algoritmo de programación dinámica necesitó varias iteraciones de diseño e implementación. No resultó obvio cómo plantear la función, y luego no resultó obvio cómo plantear la memoización, y luego no resultó obvio cómo precalcular la información adicional necesaria y mantenerlo dentro de la cota de complejidad requerida.

Si bien nos pareció más elegante nuestra implementación del algoritmo de programación dinámica, nos resultó mucho más simple la implementación del algoritmo goloso.

5. Apéndice: Resultados de Experimentos

Los programas fueron compilados con la opción `-O3` y los experimentos fueron ejecutados en un procesador AMD Ryzen 5 2600X 6-core (single-thread) con un timeout de 600 segundos a las 2 de la mañana un martes. Los resultados fueron parseados a mano.

5.1. Backtracking

Instancia	Tiempo (seg.)
brock200_1.clq	20.49
brock200_2.clq	0.09
brock200_3.clq	0.52
brock200_4.clq	1.57
brock400_2.clq	>600
brock400_3.clq	>600
brock400_4.clq	>600
C125.9.clq	131.82
C250.9.clq	>600
c-fat200-1.clq	0.02
c-fat200-2.clq	0.01
c-fat200-5.clq	0.00
c-fat500-10.clq	0.06
c-fat500-1.clq	0.00
c-fat500-2.clq	0.01
c-fat500-5.clq	0.01
DSJC500_5.clq	11.37
gen200_p0.9_44.clq	>600
gen200_p0.9_55.clq	>600
hamming6-2.clq	0.01
hamming6-4.clq	0.00
hamming8-2.clq	>600
hamming8-4.clq	3.17
johnson16-2-4.clq	3.35
johnson8-2-4.clq	0.00
johnson8-4-4.clq	0.02
MANN_a9.clq	0.72
p_hat1000-1.clq	2.64
p_hat1500-1.clq	24.53
p_hat300-1.clq	0.01
p_hat300-2.clq	4.46
p_hat300-3.clq	>600
p_hat500-1.clq	0.13
p_hat500-2.clq	>600
p_hat700-1.clq	0.58
p_hat700-2.clq	>600
san1000.clq	>600
san200_0.9_2.clq	>600
san200_0.9_3.clq	>600
san400_0.7_1.clq	>600
san400_0.7_2.clq	>600
san400_0.7_3.clq	>600
sanr200_0.7.clq	4.55
sanr200_0.9.clq	>600
sanr400_0.5.clq	3.48

Conclusiones: los tiempos de ejecución coinciden a grandes rasgos con los tiempos de referencia provistos por la cátedra y entran dentro del rango esperable de tiempo.

5.2. Podas

Instancia	Tiempo (seg.)
brock200_1.clq	23.40
brock200_2.clq	0.18
brock200_3.clq	0.72
brock200_4.clq	4.13
brock400_2.clq	>600
brock400_3.clq	>600
brock400_4.clq	>600
C125.9.clq	322.27
C250.9.clq	>600
c-fat200-1.clq	0.00
c-fat200-2.clq	0.00
c-fat200-5.clq	0.01
c-fat500-10.clq	0.05
c-fat500-1.clq	0.01
c-fat500-2.clq	0.03
c-fat500-5.clq	0.02
DSJC500_5.clq	25.62
gen200_p0.9_44.clq	>600
gen200_p0.9_55.clq	>600
hamming6-2.clq	0.00
hamming6-4.clq	0.00
hamming8-2.clq	47.96
hamming8-4.clq	5.99
johnson16-2-4.clq	1.56
johnson8-2-4.clq	0.00
johnson8-4-4.clq	0.06
MANN_a9.clq	0.00
p_hat1000-1.clq	6.00
p_hat1500-1.clq	54.82
p_hat300-1.clq	0.05
p_hat300-2.clq	26.55
p_hat300-3.clq	>600
p_hat500-1.clq	0.31
p_hat500-2.clq	>600
p_hat700-1.clq	1.26
p_hat700-2.clq	>600
san1000.clq	203.78
san200_0.9_2.clq	>600
san200_0.9_3.clq	>600
san400_0.7_1.clq	>600
san400_0.7_2.clq	>600
san400_0.7_3.clq	>600
sanr200_0.7.clq	10.16
sanr200_0.9.clq	>600
sanr400_0.5.clq	6.79

Conclusiones: los tiempos de ejecución no coinciden con los provistos por la cátedra, y en general son peores que los del ejercicio anterior, salvo por algunos casos particulares. El óptimo es correcto en cada caso que no tuvo timeout.

5.3. Programación dinámica

Instancia	Tiempo (seg.)
instancia_1.txt	0.03
instancia_2.txt	0.00
instancia_3.txt	0.00
instancia_4.txt	0.00
instancia_5.txt	0.00
instancia_6.txt	0.00
instancia_7.txt	0.00
instancia_8.txt	0.03
instancia_9.txt	0.02
instancia_10.txt	0.03
instancia_11.txt	0.26
instancia_12.txt	0.23
instancia_13.txt	0.25

Cuadro 5.3.1: Tiempos de ejecución con implementación top-down.

Instancia	Tiempo (seg.)
instancia_1.txt	0.02
instancia_2.txt	0.00
instancia_3.txt	0.00
instancia_4.txt	0.00
instancia_5.txt	0.00
instancia_6.txt	0.00
instancia_7.txt	0.00
instancia_8.txt	0.01
instancia_9.txt	0.01
instancia_10.txt	0.02
instancia_11.txt	0.21
instancia_12.txt	0.21
instancia_13.txt	0.19

Cuadro 5.3.2: Tiempos de ejecución con implementación bottom-up.

Conclusiones: los tiempos de ejecución coinciden a grandes rasgos con los tiempos de referencia provistos por la cátedra y entran dentro del rango esperable de tiempo.

5.4. Algoritmo goloso

Instancia	Tiempo (seg.)
instancia_1	0.00
instancia_2	0.00
instancia_3	0.00
instancia_4	0.00
instancia_5	0.00
instancia_6	0.00
instancia_7	0.00
instancia_8	0.03
instancia_9	0.02
instancia_10	0.02
instancia_11	0.43
instancia_12	0.38
instancia_13	0.38

Conclusiones: los tiempos de ejecución coinciden a grandes rasgos con los tiempos de referencia provistos por la cátedra y entran dentro del rango esperable de tiempo.