Tarea 6

Saul Ivan Rivas Vega

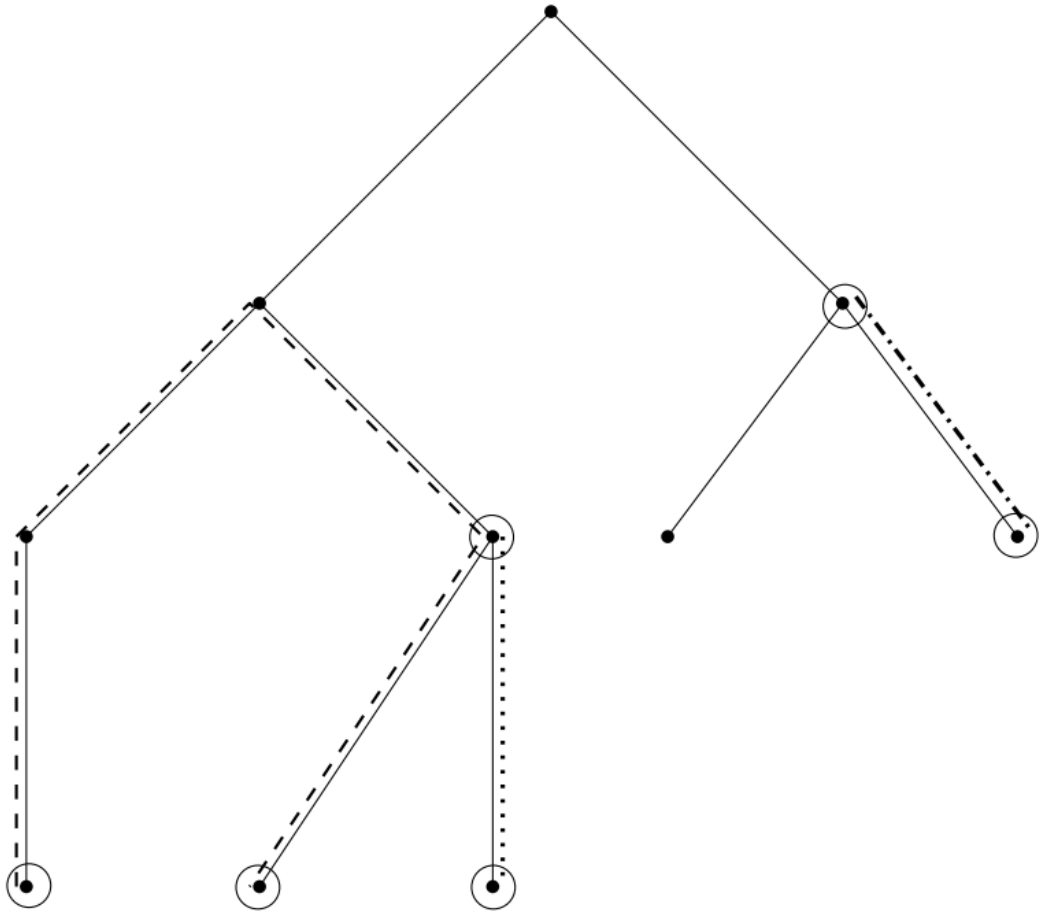
Diseño y análisis de algoritmos

Equipo Completo:
Yadira Fleitas Toranzo
Diego de Jesús Isla Lopez
Saul Ivan Rivas Vega

26 de octubre de 2019

1.

Sea T un árbol binario con raíz y S un conjunto con un número par de vértices de T. Diseña un algoritmo de tipo $divide - y - vencerás$ de tiempo $O(n)$ que empareje los nodos en S de forma tal que en T existan caminos disjuntos en aristas entre cada par.
La figura siguiente muestra un ejemplo del problema en el S tiene los seis vértices encerrados en círculos y los emparejamientos se denotan con tres tipos distintos de lineas punteadas.



1.1. Algoritmo

Data: Árbol T, donde los nodos tienen adicionalmente un valor True o False si es que pertenecen al conjunto S.
Result: Conjunto R con las parejas.

```
/* Inicializar el conjunto R.                                    */
1 R := {};
/* Llamar la función obtenerParejas' con la raíz del árbol y el conjunto
   respuesta.                                                    */
2 obtenerParejas'(raíz, R);
3 return R;
```

Algorithm 1: Algoritmo obtenerParejas.

Data: Nodo u y el conjunto actual de parejas R.
Result: Un nodo que no fue emparejado en el subárbol de u, o un valor vacío si no existe dicho nodo.

```
/* Inicializar los valores de los nodos sin emparejar encontrados en el subárbol
   del hijo izquierdo y del hijo derecho, originalmente vacíos                  */
1  nodo_libre_izq := ∅;
2  nodo_libre_der := ∅;
/* Realizamos llamadas recursivas en el subárbol del hijo izquierdo y derecho si
   es que existen                                                               */
3  if u tiene hijo izquierdo then
4  |  nodo_libre_izq:=obtenerParejas'(hijo izquierdo de u, R)
5  end
6  if u tiene hijo derecho then
7  |  nodo_libre_der:=obtenerParejas'(hijo derecho de u, R)
8  end
/* Si no hay nodos por emparejar en los subárboles checamos si u tiene que ser
   emparejado, si es el caso lo devolvemos como nodo por emparejar en su
   subárbol, en caso contrario devolvemos un valor vacío                        */
9  if nodo_libre_izq y nodo_libre_der son ∅ then
10 |  if u esta en S then
11 |  |  return u;
12 |  else
13 |  |  return ∅;
14 |  end
15 end
/* En este punto hay al menos un nodo por emparejar en alguno de los subárboles
   izquierdo o derecho.                                                         */
/* Checamos si u tiene que emparejarse.                                         */
16 if u esta en S then
   |  /* Checamos si hubo un nodo libre en el subárbol izquierdo y si es el caso,
   |     emparejamos u con ese nodo y devolvemos el valor de nodo_libre_der que puede
   |     ser vacío o un nodo libre.                                            */
17 |  if nodo_libre_izq no es igual a ∅ then
18 |  |  agregamos la pareja (u, nodo_libre_izq) a R;
19 |  |  return nodo_libre_der;
20 |  else
   |  |  /* En caso contrario, emparejamos u con el nodo libre en el subárbol
   |  |     derecho y devolvemos un valor vacío.                               */
21 |  |  agregamos la pareja (u, nodo_libre_der) a R;
22 |  |  return ∅;
23 |  end
24 end
/* Checamos si existen ambos nodos de los subárboles para emparejarlos o en su
   defecto retornar el nodo que no este vacío.                                  */
25 if no son vacíos nodo_libre_izq y nodo_libre_der then
26 |  agregamos la pareja (nodo_libre_izq, nodo_libre_der) a R;
27 |  return ∅
28 end
29 return el nodo que no este vacío entre nodo_libre_izq y nodo_libre_der;
```

Algorithm 2: Algoritmo obtenerParejas'.

1.2. Demostración complejidad $O(n)$

El algoritmo 1 realmente no nos dice mucho, solo inicializa el conjunto respuesta y lo retorna al terminar $obtenerParejas'$ el cual es el segundo algoritmo. Por esta razón realizaremos el análisis en el segundo algoritmo. Cuando el algoritmo es invocado en un nodo hoja, no se realizan mas llamadas recursivas.
Esto es, el algoritmo seguirá bajando en en árbol hasta llegar a una hoja, pues antes de realizar una llamada recursiva revisa si tiene un hijo izquierdo o derecho, como se observa en las líneas 3 y 6, como una hoja no tiene hijos pues se detendrá ahí.

Observación 1 El algoritmo es invocado en un nodo una única vez. Es decir, el algoritmo se llama sobre los hijos del nodo pero en ningún momento se realiza una llamada al padre o a un nodo que no este directamente conectado al nodo actual u. Así pues cuando el algoritmo pasa por el nodo u, no volveremos a procesar a u.

Observación 2 Las operaciones realizadas al ser invocado en una hoja, o al regresar de las llamadas recursivas, son de tiempo constante. Es decir que a partir de la línea 9 hasta el final de las operaciones, solo se realizan comparaciones y asignaciones que podemos considerar de tiempo constante.
Por la observación 1 podemos decir que como existen n nodos, se realizarán n veces las operaciones descritas en la Observación 2 y al ser de tiempo constante decimos que realizamos n veces operaciones de tiempo constante.
Finalmente decimos que el algoritmo es de complejidad $O(n)$.

1.3. Observación

Se pide solo el diseño del algoritmo con la complejidad $O(n)$ en el ejercicio pero se incluye una explicación de porque es correcto.

Correctitud. El algoritmo termina pues por la Observación 1, no se realizarán mas llamadas recursivas al llegar a las hojas, y en estas solo se toma en cuenta el if en la línea 9 si es que devolvemos o no al nodo para emparejar. Siendo el número de llamadas n y en cada llamada se realiza un número constante de operaciones.

Observación 3 En cada invocación se deja a lo más un nodo sin emparejar en el subárbol con raíz u. Esto lo podemos demostrar por inducción.
Invariante: Se deja a lo más un nodo sin emparejar en el subárbol con raíz u al terminar una invocación.
Caso base: Se considera un subárbol de un solo nodo, es decir una hoja. En este caso se devuelve el valor del único nodo si es que tiene que ser emparejado. Se cumple la invariante

pues solo podemos devolver el valor del nodo o un valor vacío.
Hipótesis inductiva: En cada invocación intermedia podemos emparejar los nodos libres que existan en el subárbol, de tal forma que dejemos a lo más uno libre.
Suponer verdadero hasta algún nodo intermedio en el árbol T.
Dicha invocación en el nodo u tiene los siguientes casos:

1. **u no tiene que emparejarse**

   a) La invocación del hijo izquierdo si devolvió un valor no vacío y también la invocación en el hijo derecho. En este caso se emparejan dichos nodos y no quedan nodos libres.

   b) Alguna de las dos invocaciones en los hijos devolvió un valor no vacío y otro sí. En este caso queda libre el nodo que no fue vacío de las invocaciones.

   c) Ninguna de las dos invocaciones en los hijos devolvió un valor no vacío. En este caso no quedan nodos libres.

2. **u si tiene que emparejarse**

   a) La invocación del hijo izquierdo si devolvió un valor no vacío y también la invocación en el hijo derecho. En este caso se emparejan el nodo u con el nodo libre de la invocación en el hijo izquierdo, dejando libre el nodo de la invocación en el hijo derecho.

   b) Alguna de las dos invocaciones en los hijos devolvió un valor no vacío y otro sí. En este caso se empareja la el nodo u con el nodo libre de las invocaciones y no quedan nodos libres.

   c) Ninguna de las dos invocaciones en los hijos devolvió un valor no vacío. En este caso queda libre el nodo u.

Finalmente como en ninguno de los casos se termina con dos o mas nodos libres podemos decir que en cada invocación se deja a lo más un nodo sin emparejar en el subárbol con raíz u. Probemos ahora que las parejas resultantes no comparten aristas en el camino que las conecta.
Por contradicción.
Entonces significa que hay por lo menos una arista por el cual pasan 2 caminos entre parejas.
Tomemos el subárbol T_j el cuál tiene como raíz al nodo inferior en dicha arista. Tal como se muestra en la figura siguiente.

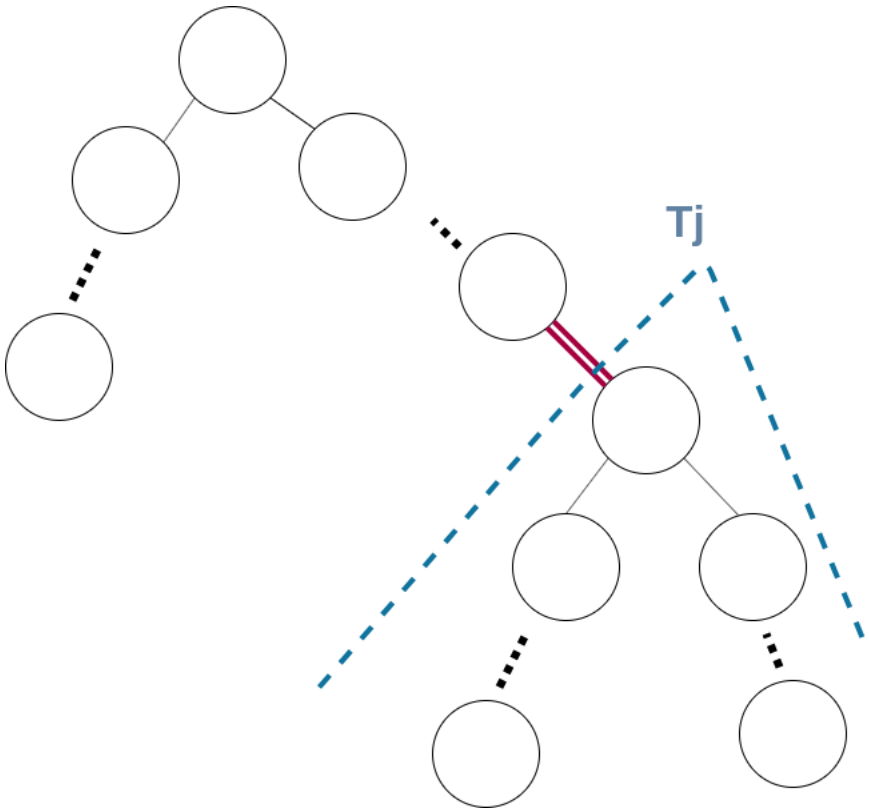


Figura 1: Subárbol T_j (línea punteada), arista con doble camino (en rojo).

En el subárbol T_j

1.4. Calendarizar clases usando el mínimo número de salones de clases posibles.

Data: Conjunto C con clases, definidos por su tiempo de inicio y tiempo de termino.
Result: Conjunto P de conjuntos T_i que contiene las clases para el salon i donde $0 < i \leq total\_de\_salones$.

```
1  Insertar todos las clases c ∈ C a una cola de prioridad Q ordenando por c.tiempo_inicio;
2  Inicializar una cola de prioridad adicional vacía S para las clases por salón;
   /* La cola de priordad S tendra parejas (clase, salon), ordenados por
      clase.tiempo_termino.                                              */
   /* Inicializar el conjunto P para la respuesta.                       */
3  P := {};
4  while Q no este vacía do
5  |  Sacar la clase c que este al frente de Q;
   |  /* Inicializar el salon para la clase c.                           */
6  |  salon_clase_c:=0;
7  |  if S tiene elementos then
   |  |  /* No sacar el elemento al frente de S solo obtener los valores. */
8  |  |  (clase_mas_cercana, salon) := S.frente();
9  |  |  if c.tiempo_inicio ≥ clase_mas_cercana.tiempo_termino then
10 |  |  |  Sacar el elemento al frente de S;
11 |  |  |  salon_clase_c:=salon;
12 |  |  end
13 |  end
   |  /* Si no se le asignó un salon entonces crear uno nuevo.           */
14 |  if salon_clase_c==0 then
   |  |  /* Sera el número de salones actuales mas uno.                  */
15 |  |  salon_clase_c:= S.tamano() + 1;
16 |  end
17 |  if  no existe P[salon_clase_c] then
18 |  |  Inicializar P[salon_clase_c] := [];
19 |  end
20 |  Insertar c en P[salon_clase_c];
21 |  Insertar (c, salon_clase_c) en S;
22 end
23 return P;
```

Algorithm 3: Algoritmo Calendarización de clases.

Demostración complejidad $O(nlogn)$ En una cola de prioridad las operaciones para insertar tiene complejidad $O(logn)$.
Al inicio se insertan los n elementos en la cola de prioridad Q, realizando n operaciones de inserción y como cada operación de inserción tiene complejidad $O(logn)$ tenemos que al terminar de insertarlos habremos realizado $nlogn$ operaciones.
En el ciclo *while* realizaremos iteraciones mientras aún haya elementos en Q, y como en cada iteración sacamos al elemento t que esta al frente. Es decir, el número de iteraciones es igual al número de elementos en Q. Al inicio insertamos a todos los trabajos en Q, siendo entonces n iteraciones.
En cada iteración realizaremos las siguientes operaciones:

- Sacar la clase c al frente de Q, que es la operación de remover en una cola de prioridad, siendo de complejidad $O(logn)$.
- Sacar el elemento al frente de S, siendo $O(logn)$ en una cola de prioridad.
- Insertar $(c, salon\_clase\_c)$ en S, siendo $O(logn)$ en una cola de prioridad.
- Comparaciones y asignaciones que podemos considerar como de complejidad $O(1)$.

Como hacemos n iteraciones entonces tenemos que el ciclo *while* es de complejidad $O(nlogn)$. Como tanto la parte inicial como el ciclo *while* tienen la misma complejidad dominante, finalmente decimos que el algoritmo es de complejidad $O(nlogn)$.

1.5. Calendarización para minimizar la latencia máxima.

Data: Conjunto T con trabajos, definidos por su tiempo requerido y su deadline.
Result: Arreglo S con los trabajos en el orden requerido para minimizar la latencia máxima.

```
1 Insertar todos los trabajos t ∈ T a una cola de prioridad Q ordenando por t.deadline;
  /* Inicializar el conjunto S para la respuesta.                    */
2 S := {};
3 while Q no este vacía do
4 |   Sacar al trabajo t que este al frente de Q;
5 |   insertar t en S;
6 end
7 return S;
```

Algorithm 4: Algoritmo minimizar la latencia máxima.

Demostración complejidad $O(nlogn)$ En una cola de prioridad las operaciones para insertar tiene complejidad $O(logn)$.
Al inicio se insertan los n elementos en la cola de prioridad Q, realizando n operaciones de inserción y como cada operación de inserción tiene complejidad $O(logn)$ tenemos que al terminar de insertarlos habremos realizado $nlogn$ operaciones.
En el ciclo *while* realizaremos iteraciones mientras aún haya elementos en Q, y como en cada iteración sacamos al elemento t que esta al frente. Es decir, el número de iteraciones es igual al número de elementos en Q. Al inicio insertamos a todos los trabajos en Q, siendo entonces n iteraciones.
En cada iteración además de remover un elemento de Q. Para la operación de remover a un elemento esta es de complejidad $O(logn)$ y como hacemos n iteraciones entonces tenemos que el ciclo *while* es de complejidad $O(nlogn)$.
Como tanto la parte inicial como el ciclo *while* tienen la misma complejidad dominante, finalmente decimos que el algoritmo es de complejidad $O(nlogn)$.

2. Ejercicio 2. Da un algoritmo de tipo *greedy* para alguno de los siguientes problemas.

Se escogió el problema:

- Encontrar un conjunto independiente maximal de una gráfica no dirigida G. Un conjunto independiente es un conjunto de vértices que no tienen aristas entre ellos, y es maximal si no hay otro conjunto independiente que lo contenga propiamente.

Data: Una gráfica G que es una dupla (V, E).
Result: Un conjunto independiente maximal C.

```
/* Inicializar un arreglo de tamaño |V| para el degree de cada vértice, indexado
   desde 1. Con valor inicial de 0.                                          */
degree := [1 : |V|, 0];
/* Inicializar el arreglo de listas de adyacencias de cada vértice, indexado
   desde 1. Con valor inicial de []                                          */
ady := [1 : |V|, []];
for cada arista (u, v) ∈ E do
    degree[u] + +;
    degree[v] + +;
    Agregar v a ady[u];
    Agregar u a ady[v];
end
Inicializar una cola de prioridad Q;
/* La cola de priordad Q tendra parejas (u, degree[u]), ordenados por degree[u]. */
for cada vértice u ∈ V do
    Insertar la pareja (u, degree[u]) en Q;
end
/* Inicializar el conjunto C para la respuesta.                              */
C := {};
/* Inicializar un arreglo de tamaño |V| para marcar los vértices en la respuesta
   indexado desde 1. Con valor inicial de 0.                                 */
marcado := [1 : |V|, 0];
while Q no este vacía do
    Sacar la pareja (u, degree[u]) de Q;
    /* Inicializar una bandera para saber si el vértice u comparte arista con
       alguno de los vértices marcados.                                      */
    valido := 1;
    /* Recorremos la lista de adyacencia de u.                               */
    for cada vértice v ∈ ady[u] do
        /* Ya no es válido si tiene como vecino inmediato a alguno de los vértices
           marcados como parte de la respuesta.                              */
        if marcado[v] == 1 then
            valido := 0;
            break;
        end
    end
    if valido == 1 then
        marcado[u] = 1;
        Insertar u en C;
    end
end
return C;
```

Algorithm 5: Algoritmo Conjunto Independiente Maximal

2.1. Explica claramente cual es la entrada y la salida de tu algoritmo.

Entrada. Una gráfica G que es una dupla (V, E), siendo V un conjunto con los vértices, y E un conjunto de aristas de la forma (u, v) donde $u, v \in V$ y $u \neq v$. También no hay aristas repetidas en E, es decir si $(u, v) \in E$ entonces $(v, u) \notin E$.

Salida. Un conjunto C con vértices de V, tal que no hay aristas entre cualquier par de vértices en C. De igual forma no existe otro conjunto C' tal que C es subconjunto propio de C'.

2.2. Explica claramente por qué tu solución puede ser catalogada como del tipo *greedy*.

Puede ser catalogada como del tipo *greedy* porque el algoritmo a diferencia de recorrer arbitrariamente los vértices y agregarlos al conjunto de respuesta mientras no compartan aristas, el algoritmo recorre los vértices en orden de su degree. Toma en cuenta su degree, porque la intuición es que. en las decisiones locales, si vamos agregando al conjunto de respuesta a los vértices que menos vecinos tengan dejará mas opciones para agregar en una iteración posterior. Esto funciona a corto plazo pues parece que si eliminamos menos opciones por iteración entonces tendremos mas iteraciones y por lo tanto mas vértices en el conjunto respuesta. Sin embargo como se demostrará posteriormente este método aunque correcto para un conjunto independiente maximal, no es óptimo.

2.3. Demuestra que tu algoritmo es correcto, también demuestra su complejidad de tiempo y espacio.

El algoritmo termina El ciclo de la línea 3 termina pues como recorremos una sola vez cada arista en E, a lo mas tendremos $|E|$ iteraciones.
El ciclo de la línea 10 termina pues como recorremos una sola vez cada vértice en V, a lo mas tendremos $|V|$ iteraciones.
El ciclo de la línea 15 termina pues en cada iteración removemos un elemento de Q y por construcción Q tiene la misma cardinalidad que V, por lo tanto a lo mas tendremos $|V|$ iteraciones.
El ciclo de la línea 18 termina pues se recorre cada elemento en su lista de adyacencia y por construcción es finita y solo contiene a los vértices con los que comparte arista.

El algoritmo es de complejidad en tiempo de $O(|V|log(|V|) + |E|)$ Las primeras asignaciones en las líneas 1 y 2, podemos considerarlas operaciones de tiempo $O(1)$.

Llevamos $O(1)$.

El ciclo en la línea 3, recorre todas las aristas $(u, v) \in E$, como recorre cada una una sola vez, tiene $|E|$ iteraciones. En cada iteración se realizan operaciones de asignación, acceso a memoria e incrementos, los cuales podemos considerar de tiempo $O(1)$. Así el ciclo de la línea 3 tiene complejidad $O(|E|)$

Llevamos $O(1 + |E|)$.

El ciclo en la línea 10, recorre todos los vértices $v \in V$, como recorre cada uno una sola vez, tiene $|V|$ iteraciones. En cada iteración se realiza una operación de inserción en la cola de prioridad Q lo cual tiene complejidad $O(logn)$ donde $n = |V|$ pues son los elementos a insertar. Así el ciclo en la línea 10 tiene complejidad $O(|V|log(|V|))$.

Llevamos $O(1 + |E| + |V|log(|V|))$

El ciclo de la línea 15 tiene una función de complejidad mas interesante para representar, en conjunto con la del ciclo en la línea 18.

Podemos decir que la función de complejidad para la primera iteración del ciclo en la linea 15 esta dada por:

$$f_{ciclo\_linea\_15}(1) = f_{remover\_de\_Q}(1) + f_{ciclo\_linea\_18}(|lista\_adyacencia\_u_1|) \tag{1}$$

Es decir que 1 iteración ($f_{ciclo\_linea\_15}(1)$) es la suma de la función de complejidad de remover un elemento de la cola de prioridad Q más la función de complejidad del ciclo en la línea 18 evaluada con la cardinalidad de la lista de adyacencia del primer vértice a evaluar u_1.
La función de complejidad de remover un elemento de la cola de prioridad Q es $log(|V|)$ pues tenemos que Q tiene a todos los vértices $v \in V$.
Al sustituirlo en la ecuación 1, la ecuación queda como:

$$f_{ciclo\_linea\_15}(1) = log(|V|) + f_{ciclo\_linea\_18}(|lista\_adyacencia\_u_1|) \tag{2}$$

Para obtener la función de complejidad para el ciclo de la linea 18, vemos que realiza la misma cantidad de iteraciones que la cardinalidad de su lista de adyacencia. Es decir que recorre una vez a cada vértice con el que comparte una arista. Para el vértice u_1 la función de complejidad para el ciclo de la linea 18 esta dada por:

$$f_{ciclo\_linea\_18}(|lista\_adyacencia\_u_1|) = |lista\_adyacencia\_u_1| \tag{3}$$

Al sustituirlo en la ecuación 2, la ecuación queda como:

$$f_{ciclo\_linea\_15}(1) = log(|V|) + |lista\_adyacencia\_u_1| \tag{4}$$

Ahora veamos como a la función de complejidad del ciclo en la línea 15 evaluándola en las $|V|$ iteraciones que tiene como vimos en la sección en donde definimos que el algoritmo termina.

$$f_{ciclo\_linea\_15}(|V|) = \sum_{i=1}^{|V|} (log(|V|) + |lista\_adyacencia\_u_i|) \tag{5}$$

Ahora veamos otra perspectiva para la ecuación 5.

$$\begin{aligned} f_{ciclo\_linea\_15}(|V|) &= \sum_{i=1}^{|V|} (log(|V|) + |lista\_adyacencia\_u_i|) \\ &= \sum_{i=1}^{|V|} log(|V|) + \sum_{i=1}^{|V|} |lista\_adyacencia\_u_i| \\ &= (|V|log(|V|)) + \sum_{i=1}^{|V|} |lista\_adyacencia\_u_i| \end{aligned} \tag{6}$$

Ahora si sumamos la cardinalidad de todas las listas de adyacencia tendremos $2|E|$ porque si un vértice v esta en la lista de adyacencia de u, significa que hay una arista (u, v), y de igual forma el vértice u estará en la lista de adyacencia de v, es decir, estamos contando una misma arista (u, v), en la lista de u y en la lista de v, para toda arista $(u, v) \in E$.

Realizando dicha sustitución en la ecuación 6 tenemos:

$$f_{ciclo\_linea\_15}(|V|) = |V|log(|V|) + 2|E| \tag{7}$$

Así tenemos que el ciclo en la línea 15 es de complejidad $O(|V|log(|V|) + |E|)$.

Llevamos $O(1 + |E| + |V|log(|V|) + |V|log(|V|) + |E|)$
Finalmente simplificamos con los factores dominantes y tenemos que la complejidad del algoritmo es:
$O(|V|log(|V|) + |E|)$

Correctitud: El algoritmo regresa un conjunto independiente maximal. Por inducción.
Invariante: Si un vértice esta en el conjunto resultante C, dicho vértice no comparte arista con ningún otro vértice en C.
Caso base: Se considera al primer vértice que esta al frente de Q, es decir tiene el menor *degree*. No hay ningún vértice marcado, ni en C. Así como no comparte ninguna arista con otro vértice en C, se inserta. C solo tiene a este vértice y cumple que no hay otro vértice en C con el que comparta arista.
Hipótesis inductiva: El siguiente vértice en insertar es parte del conjunto independiente máximal que incluye a los vértices actualmente en C. Suponer verdadero hasta el i-ésimo vértice en agregar a C.
Dicho vértice v_i no comparte arista con ningún otro vértice en C, pues esto se verifica antes de insertar al vértice al verificar su lista de adyacencia en el ciclo de la línea 18 del algoritmo 4.
Como esto se repetirá para los siguientes vértices posteriores a v_i, podemos decir que al recorrer todos los vértices no habrá otro conjunto C' tal que C sea conjunto propio de C'.
Podemos demostrarlo por contradicción, ya que de existir dicho conjunto C' significa que hay un vértice $v' \in C'$ y que $v \notin C$, tal que no comparte arista con ningún otro vértice en C' y por lo tanto tampoco comparte arista con ningún vértice en C ya que los elementos de C están en C'. Dicho vértice v' no está en C ya sea por que lo descartamos o no lo revisamos, pero solo descartamos un vértice si comparte arista con algún vértice en C, y revisamos todos los vértices $v \in V$, esto es una contradicción pues si existe dicho vértice v' entonces $v' \in C$ pero la suposición inicial es que $v' \notin C$. Ahora como no existe ni un solo vértice tal que $v' \in C'$ y que $v' \notin C$ decimos que $C = C'$ puesto que tienen los mismos elementos. Finalmente como C no es subconjunto propio de ningún otro conjunto independiente significa que el conjunto C que devuelve el algoritmo es un conjunto independiente maximal.

El algoritmo es de complejidad en espacio de $O(|V| + |E|)$ Realizaremos un conteo de las variables adicionales a la entrada que se utilizaron.
Se utiliza un arreglo *degree* de tamaño $|V|$.

Llevamos $O(|V|)$.

Se utiliza un arreglo de listas, como vimos en la demostración de complejidad la suma de las cardinalidades de las listas de adyacencia de todos los nodos será igual a $2|E|$.

Llevamos $O(|V| + 2|E|)$.

Se utiliza una cola de prioridad que en un inicio tiene cardinalidad $|V|$ y no realizamos otra operación mas que remover, por lo tanto su máxima cardinalidad es $|V|$.

Llevamos $O(|V| + 2|E| + |V|)$.

Se utiliza un arreglo *marcado* de tamaño $|V|$.

Llevamos $O(|V| + 2|E| + |V| + |V|)$.

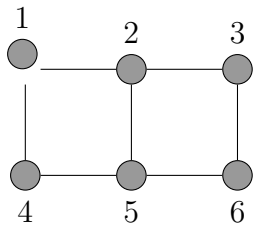
Se utiliza una variable *valido*.

Llevamos $O(|V| + 2|E| + |V| + |V| + 1)$.

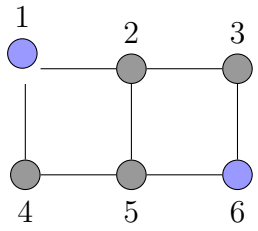
Simplificando tenemos a los factores dominantes:
$O(|V| + |E|)$.

2.4. Además, prueba que tu algoritmo, a pesar de ser correcto, no es óptimo. Es decir, da una gráfica G para la que existe una mejor solución que la que tu algoritmo produce.

En este caso podemos demostrarlo proponiendo una gráfica en la que exista un conjunto independiente de mayor cardinalidad a la que devuelve el algoritmo.
Sea dicho ejemplo la siguiente gráfica:
$G = (V, E)$
$V = \{1, 2, 3, 4, 5, 6\}$
$E = \{(1, 2), (1, 4), (2, 3), (2, 5), (3, 6), (4, 5), (5, 6)\}$

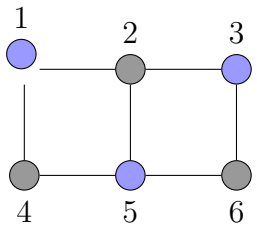


Siguiendo los pasos del algoritmo se tomará primero a uno de los nodos con menor *degree* que en este caso podría ser cualquiera de $\{1, 3, 4, 6\}$, ya que su degree es el menor el cual es $degree = 2$
Supongamos que el algoritmo toma a 1, ahora seguimos con otro nodo que sea de $degree = 2$.
No puede ser 4, puesto que comparte arista con 1.
Puede tomar ahora ya sea a 3 o a 6. Supongamos que toma a 6. Ahora no puede tomar a ningún otro nodo pues compartirá una arista ya sea con 1 o con 6.
El algoritmo retornará el conjunto independiente maximal $\{1, 6\}$.



El cuál no es óptimo.
La cardinalidad óptima, es decir la mayor cardinalidad en este ejemplo es 3, ya sea el conjunto independiente maximal $\{1, 3, 5\}$ o $\{4, 2, 6\}$.

Finalmente podemos decir que a pesar de ser correcto, el algoritmo no es óptimo al existir esta gráfica para la cual existe un conjunto independiente de mayor cardinalidad que el conjunto que devuelve el algoritmo.



3. Ejercicio 3.

Diseña un algoritmo que dada una cadena S encuentre y regrese el primer símbolo que no se repite (leyendo S de izquierda a derecha); si dicho símbolo no existe, entonces la salida debe ser #. Tu algoritmo debe recorrer una sola vez la cadena S y utilizar $O(1)$ en memoria (adicional a la entrada). La cadena S contiene solo letras minúsculas del alfabeto Español.

Data: Una cadena S.
Result: El primer símbolo que no se repite, o # si no existe.

```
   /* Sea Σ el conjunto de símbolos en el alfabeto español.                    */
   /* Inicializar un arreglo de tamaño |Σ| para marcar a cada símbolo, indexado desde
      1. Con valor inicial de 0.                                               */
 1 marcado := [1 : |Σ|, 0];
   /* Recorremos cada posición de la cadena S desde la primera hasta la última, de
      izquierda a derecha.                                                     */
 2 for i desde 1 hasta |S| do
   |  /* La función MAPEO, nos devuelve un identificador entre 1 y |Σ|, 1 para a, 2
   |     para b, etc.                                                         */
   |  /* Si no hemos visto este símbolo, guardamos su posición.               */
 3 |  if marcado[MAPEO(S[i])] == 0 then
 4 |  |  marcado[MAPEO(S[i])] := i;
 5 |  else
   |  |  /* Si ya lo vimos lo sobreescribimos con un -1.                      */
 6 |  |  marcado[MAPEO(S[i])] := −1;
 7 |  end
 8 end
   /* Inicializar la variable de respuesta con el valor #.                     */
 9 primer_no_repetido := #;
   /* Inicializar la variable que guardará la posición mas cercana al inicio donde
      vimos un símbolo que no se repitió.                                      */
10 posicion_mas_cercana := ∞;
   /* Recorremos cada símbolo del alfabeto español.                            */
11 for cada símbolo x ∈ Σ do
   |  /* Si no esta marcado como repetido y lo vimos en la cadena, (es decir no
   |     tiene valor -1 o valor 0).                                           */
12 |  if marcado[MAPEO(x)] > 0 then
   |  |  /* Si tiene una posición mas cercana al inicio actualizamos nuestra
   |  |     respuesta.                                                        */
13 |  |  if marcado[MAPEO(x)] < posicion_mas_cercana then
14 |  |  |  posicion_mas_cercana := marcado[MAPEO(x)];
15 |  |  |  primer_no_repetido := x;
16 |  |  end
17 |  end
18 end
19 return x;
```

Algorithm 6: Algoritmo Primer Símbolo que no se repite

Data: Un símbolo $x \in \Sigma$.
Result: Un identificador entre 1 y $|\Sigma|$ que es distinto para cada $x \in \Sigma$.

```
1  if x ==' a' then
2  |  return 1;
3  end
4  if x ==' b' then
5  |  return 2;
6  end
7  if x ==' c' then
8  |  return 3;
9  end
   /* Continua así hasta el último símbolo de Σ.                    */
10 if x ==' y' then
11 |  return 26;
12 end
13 if x ==' z' then
14 |  return 27;
15 end
```

Algorithm 7: función MAPEO

Observación Se pide solo el diseño del algoritmo en el ejercicio pero se incluye una pequeña explicación de porque es correcto, su complejidad de tiempo y que cumple con la restricción de memoria.

Correctitud. El algoritmo termina pues en la línea 2 el ciclo recorre una sola vez cada símbolo de la cadena $|S|$ haciendo solo $|S|$ iteraciones. En la línea 11 recorremos una sola vez cada símbolo del alfabeto español haciendo solo $|\Sigma|$ iteraciones, que como es constante el alfabeto son en total 27 iteraciones.
El algoritmo devolverá el primer símbolo que no se repite, o # si no existe. Por contradicción. Supongamos que hay un símbolo x' que es el primero que no se repite y es distinto al que devolvió el algoritmo. Entonces no lo reportó como respuesta por que o no se evaluó como mas cercano al inició que la respuesta que dimos o por que no lo evaluamos. Si se evaluó como mas cercano al inicio lo debimos escoger, y no pudimos no recorrerlo, pues en la línea 11 recorremos a todos los símbolos del alfabeto. De igual forma en la línea 2 recorremos todos los símbolos de la cadena entonces si lo marcamos. Por lo tanto de existir x' entonces es igual a la respuesta reportada por el algoritmo, pero es una contradicción pues supusimos que era distinta en un inicio. Finalmente decimos que el algoritmo devolverá el primer símbolo que no se repite, o # si no existe.

Complejidad en tiempo. Se realiza un solo recorrido de la cadena en la línea 2, por lo que llevamos $O(|S|)$.
Además de que en la línea 11 recorremos todos los símbolos en el alfabeto español, ahora llevamos $O(|S| + |\Sigma|)$.
Sin embargo la cardinalidad del alfabeto es constante y es 27. Así llevamos entonces $O(|S|+$

27).
Simplificando con el valor dominante $|S|$ tenemos finalmente $O(|S|)$.

Complejidad en espacio. Además de la entrada se tiene un arreglo *marcado* con un tamaño de $|\Sigma|$, por lo que llevamos $O(|\Sigma|)$.
También usamos un par de variables *primer_no_repetido* y *posicion_mas_cercana*, ahora llevamos $O(|\Sigma| + 2$.
Sin embargo la cardinalidad del alfabeto es constante y es 27. Así llevamos entonces $O(27 + 2)$, que es $O(29)$.
Simplificando con el valor dominante que es constante y tenemos finalmente $O(1)$.

4. Ejercicio 4.

Diseña un algoritmo tipo *greedy* para el siguiente problema de optimización. La entrada está compuesta de un conjunto con n pares de enteros positivos, $\{(w_1, v_1), ..., (w_n, v_n)\}$, y un entero positivo W; cada (w_i, v_i) representa una clase de objetos con peso w_i y valor v_i , y W representa una caja que puede cargar esa cantidad de peso. El objetivo es seleccionar algunos objetos de los n posibles a meter en la caja de forma tal que se maximice la suma del valor de los objetos seleccionados, permitiendo meter fracciones de objetos; por ejemplo, tu algoritmo podría decidir meter la unidad completa de un objeto y solo una fracción de otro. Formalmente, para el i-ésimo objeto, (w_i, v_i), tu algoritmo debe decidir meter una fracción $x_i \in [0, 1]$ de él, tal que $weight = \sum_{i=1}^{n} x_i w_i \leq W$ y el valor de la solución, $val = \sum_{i=1}^{n} x_i v_i$, sea el máximo posible.

Data: Un conjunto O de n parejas de la forma (w_i, v_i) que representa el conjunto de objetos.
Data: Un entero positivo W que representa la capacidad de peso de la caja.
Result: Una dupla $(valor\_maximo, objetos\_tomados)$ que representa el valor maximo alcanzado y el arreglo de pares de la forma $(id\_objeto, fraccion\_tomada)$ el identificador del objeto tomado, y la cantidad (que puede ser fraccional) que se tomó de dicho objeto.

```
1  Inicializar la cola de prioridad relaciones_vp para guardar las relaciones valor/peso;
   /* La cola de prioridad relaciones_vp manejará parejas de la forma (objeto_i, -v_i/w_i) y
      ordenará por -v_i/w_i, en este caso esta negativo para ordenar de mayor a menor
      relación valor/peso.                                                          */
   /* Inicializar la variable para guardar el valor máximo obtenido                 */
2  valor_maximo := 0;
   /* Inicializar el conjunto de parejas en la respuesta.                           */
3  objetos_tomados := {};
   /* Recorremos cada posición de las parejas de la entrada.                        */
4  for i desde 1 hasta n do
   |  /* Insertamos la pareja de los valores del objeto en la posición i en la cola
   |     de prioridad.                                                             */
5  |  Insertamos (O[i], -O[i].v_i/O[i].w_i) en relaciones_vp;
6  end
   /* Recorremos los elementos en la cola de prioridad.                             */
7  while relaciones_vp no este vacía do
8  |  Sacamos al objeto o al frente de relaciones_vp;
   |  /* Tomamos la cantidad que podamos de este objeto, o 1 unidad pues no podemos
   |     tomar una fracción mayor a 1 unidad.                                      */
9  |  cantidad_objeto := min(1, W/o.w_i);
   |  /* Nuestra respuesta se incrementa en el valor del objeto multiplicado por la
   |     cantidad que tomamos.                                                     */
10 |  valor_maximo := valor_maximo + (o.v_i × cantidad_objeto);
11 |  Insertamos la pareja (o.id, cantidad_objeto) a objetos_tomados;
   |  /* Disminuimos el peso disponible en la caja.                                */
12 |  W := W − (o.w_i × cantidad_objeto);
   |  /* Si ya no hay peso disponible en la caja rompemos el ciclo.                */
13 |  if W = 0 then
14 |  |  break
15 |  end
16 end
17 return (valor_maximo, objetos_tomados);
```

Algorithm 8: Algoritmo Mayor valor de objetos fraccionales.

4.1. Explica por qué tu solución puede ser catalogada como del tipo *greedy*

Es de tipo *greedy* por que en su decisión local escoge llenar su espacio disponible con las fracciones del objeto que da mas valor por cada unidad de peso. Entonces toma el de mayor valor, pero como no podemos repetir objetos, aunque podamos llenar toda la capacidad W con fracciones de ese objeto, no podemos tomar mas que 1 unidad de ese objeto. Lo que nos quede de espacio disponible lo intentaremos llenar con el siguiente mas grande así sucesivamente hasta terminar de procesar todos los objetos o que no tengamos peso disponible.

4.2. Demuestra que tu algoritmo es correcto.

Observación 1 Demostremos que en el caso en que todos los objetos tienen el mismo peso la respuesta es completar el peso W tomando la fracción mas grande, que no sea mayor a 1 unidad, de los objetos en orden desde el objeto con mayor valor hasta el de menor valor. Si tomamos fracciones del objeto con mayor valor significa que tomamos c veces el objeto de mayor valor, donde $c = min(1, W/peso\_objeto\_mayor\_valor)$. Esto se ve como:

$$Valor\_total = c \times valor\_objeto\_mayor\_valor \tag{8}$$

Es claro que todos los objetos tienen el mismo valor para c, pues todos tienen el mismo peso. Entonces lo único que varia es el valor del objeto por el que multiplicamos a c. Cualquier valor de otro objeto será menor o igual. Por lo tanto tenemos:

$$c \times valor\_objeto\_mayor\_valor \geq c \times valor\_i \tag{9}$$

Para todo objeto i en el conjunto de objetos.
Finalmente decimos que si todos los objetos tienen el mismo peso, el valor que obtenemos completar el peso W tomando la fracción mas grande, que no sea mayor a 1 unidad, de los objetos en orden desde el objeto con mayor valor hasta el de menor valor es el valor máximo que podemos obtener.

Observación 2 Cualquier entrada para el problema puede reducirse al caso tratado en la Observación 1.
Como podemos tomar fracciones de los objetos en la entrada entonces podemos tomar fracciones de peso 1 de cada objeto. Entonces tendremos una cantidad w_i de fracciones de peso 1 y valor v_i/w_i para todo objeto i en el conjunto de objetos. Pero esto lo podemos interpretar como si tuviéramos w_i objetos de peso 1 y valor v_i/w_i para todo objeto i en el conjunto de objetos. Por lo tanto podemos representar a la entrada como objetos del mismo peso pero con posible distinto valor, que es justo el caso que se trata en la Observación 1.

Finalmente por las observaciones 1 y 2, podemos obtener el valor máximo al completar el peso W tomando la fracción mas grande, que no sea mayor a 1 unidad, de los objetos en orden desde el objeto con mayor relación valor/peso hasta el de menor relación valor/peso. Lo cual hacemos después de insertar los objetos para procesarlos en orden de mayor relación valor/peso en el ciclo de la línea 7, tomando en consideración la restricción de no tomar mas de 1 unidad del objeto en la línea 9, así como terminando el ciclo si ya no nos queda espacio en la caja con la evaluación en la línea 13.

4.3. Demuestra su complejidad de tiempo.

Hasta antes de la línea 4 solo realizamos inicializaciones las cuales podemos considerar de complejidad $O(1)$.

En el ciclo de la línea 4 realizamos n iteraciones pues el conjunto de objetos es de cardinalidad n y recorremos cada posición entre 1 y n. En cada iteración de este ciclo realizamos una inserción en la cola de prioridad $relaciones\_vp$, la cual tiene complejidad $O(logn)$. Por lo tanto este ciclo es de complejidad $O(nlogn)$.
En el ciclo de la línea 7 realizamos n iteraciones pues sacamos un elemento de la cola de prioridad en cada iteración y solo tiene n elementos.En cada iteración de este ciclo removemos un elemento de la cola de prioridad $relaciones\_vp$, la cual tiene complejidad $O(logn)$. También realizamos unas evaluaciones, asignaciones y operaciones aritméticas básicas las cuales podemos considerar de complejidad $O(1)$. Por lo tanto este ciclo es de complejidad $O(nlogn)$, n iteraciones donde el factor dominante en cada una es $logn$.

Finalmente como el factor dominante en los ciclos $O(nlogn)$ el algoritmo es de complejidad en tiempo de $O(nlogn)$.

4.4. Demuestra su complejidad de espacio (cantidad de memoria adicional a la entrada).

Además de la entrada se declaran 2 variables en las lineas 2 y 3 del algoritmo respectivamente.
Así como una cola de prioridad que crecerá con respecto a la entrada teniendo n elementos. Como el número de variables adicionales se incrementa de manera lineal con respecto a la entrada podemos decir que el algoritmo tienen una complejidad en memoria de $O(n)$.