UNIVERSIDAD SIMÓN BOLÍVAR
CI-5651 - Diseño de Algoritmos I
Prof. Ricardo Monascal

Resuelto por:
Christopher Gómez

Tarea 2: Algoritmos voraces

1. En un nuevo túnel, diversos productos quieren colocar vallas para promover sus productos. Usted debe dar los permisos apropiados para colocar estas vallas y no desea que ninguna de estas se intersecte con otra (sin embargo, dos vallas pueden estar una al lado de la otra, esto es, compartir un extremo).

   Las propuestas de publicidad incluyen el punto en donde la valla empezará (en metros, contando desde el inicio del túnel) y el tamaño de la misma (también en metros). Ninguna de estas condiciones es negociable. Los departamentos de publicidad de cada uno de los productos interesados han llegado a la conclusión de que quieren la valla en ese exacto punto, con esas exactas dimensiones o prefieren no colocar publicidad alguna. Todas las vallas se colocarán planas en una sola pared del túnel.

   Dadas n peticiones de publicidad, donde la i-ésima petición tiene p_i (el punto de inicio de la valla) y t_i (el tamaño de la valla), diseñe un algoritmo *eficiente* que permite escoger un conjunto maximal de peticiones tal que ninguna valla intersecte con otra.

   Su algoritmo debe usar tiempo $O(n \log n)$ y memoria adicional $O(n)$.

   Justifique *informalmente* por qué su algoritmos es correcto y cumple con el orden asintótico propuesto.

   Este problema puede ser resuelto utilizando un algoritmo voraz, el cual consiste en ordenar las peticiones de publicidad en orden creciente de sus puntos finales $(p_i + t_i)$ y luego recorrer la lista, seleccionando la que se está considerando si no se superpone con las ya seleccionadas. Veámoslo en pseudocódigo:

   ```
   def publicidad(peticiones: Conjunto[Peticion]) -> Conjunto[Peticion]:
       peticiones.odernar(lambda p: p.comienzo + p.tamaño)
       seleccionadas = {}
       for peticion in peticiones:
           if not se_superpone(peticion, seleccionadas):
               seleccionadas.añadir(peticion)
       return seleccionadas
   ```

   Una noción intuitiva de por qué este algoritmo es correcto es que, al seleccionar la petición con el punto final más pequeño que no se superpone con las ya seleccionadas, se deja el mayor espacio posible para las peticiones restantes. Esto es, se selecciona la petición que deja el mayor espacio a la derecha para las peticiones restantes.

Agregando un poco más de formalidad, definamos que un conjunto de peticiones es prometedor si constituye un subconjunto de alguno de los conjuntos maximales de peticiones que no se superponen. Luego, el algoritmo voraz es correcto si en cada iteración del ciclo, el conjunto de peticiones seleccionadas es prometedor, hasta que no hay más peticiones por considerar.

Caso base: El conjunto vacío es prometedor, ya que es subconjunto de cualquier conjunto. En particular, de los conjuntos solución.

Caso inductivo: Supongamos que P es prometedor. En una iteración dada se selecciona una petición p tal que:

a) No se superpone con ninguna de las peticiones seleccionadas.

b) Tiene el punto final más pequeño de las no seleccionadas.

Se tiene entonces que $P \cup \{p\}$ es prometedor, ya que de no serlo (un subconjunto maximal de peticiones que no se superponen), entonces existen al menos dos peticiones q_1 y q_2 no seleccionadas que no se superponen entre sí, ni con ninguna de las peticiones seleccionadas, y que tienen puntos finales más pequeños que p capaces de sustituir a p para formar un conjunto más grande. Pero esto es una contradicción, ya que p fue seleccionada por tener el punto final más pequeño de las no seleccionadas.

Por lo tanto, el invariante se mantiene en todo momento y que el algoritmo es correcto.

Luego, el algoritmo tiene un tiempo de ejecución de $O(n \log n)$ debido a la ordenación inicial, que se puede implementar en $O(n \log n)$, y el recorrido por la lista de peticiones ordenadas una sola vez, que tarda $O(n)$, ya que se puede implementar `se_superpone` en $O(1)$, manteniendo una variable que almacene el último punto final seleccionado y comparando con este en cada iteración.

Además, utiliza memoria adicional $O(n)$ para almacenar las peticiones, y una implementación *in-place* de la ordenación no requiere más que memoria adicional $O(1)$, por lo que cumple con el orden asintótico propuesto.

Una implementación de este algoritmo en Rust se puede encontrar aquí.

2. Considere las siguientes definiciones:

   Conjuntos definitivos de firmas funcionales

   Considere un conjunto de tipos T y sea F todas las firmas de funciones posibles entre los tipos de T .

   Por ejemplo, si $T = \{A, B, C\}$ entonces F contiene

   - $A \to A$
   - $A \to B$
   - $A \to C$
   - $B \to A$
   - $B \to B$
   - $B \to C$
   - $C \to A$
   - $C \to B$
   - $C \to C$

   Diremos que un subconjunto $F' \subseteq F$ es *definitivo* si cada tipo ocurrente en F' aparece a lo sumo una vez como imagen en F'.

   Por ejemplo, para el mismo T anterior:

   - $\{A \to B, B \to C\}$ es definitivo

- $\{A \to B, C \to B\}$ no es definitivo.

Matroide asociada

Dado un conjunto de tipos T y su conjunto de todas las firmas de funciones F, definimos $M_T = (F, I)$, donde $F' \in I$ si y sólo si F' es definitivo.

Potencial de una firma funcional

Definimos el *potencial* de una firma de función $X \to Y$ como la cantidad de funciones posibles que existen con X como dominio y Y como imagen. Recordemos que esto es igual a $|Y|^{|X|}$. Puede suponer que todos los tipos tienen al menos un elemento (T no contiene al tipo vacío).

El potencial de un conjunto de firmas de funciones es la suma del potencial para cada una de la firmas que contiene.

Se desea que:

a) Demuestre que M_T es una matroide. b) Plantee una función de costo w de tal forma que para la matroide pesada M_T , con w como función de peso, los sub-conjuntos óptimos son los subconjuntos *definitivos* con *potencial máximo*.