Universidad Nacional Autónoma de México

Facultad de Ciencias

Análisis y Diseño de Algoritmos

Tarea 6

Martínez Avila Santiago

Reynoso Sánchez Arturo Yitzack

26 de noviembre de 2021

1 Tarea

Un árbol es un tipo de gráfica muy simple: es conexo y no tiene ciclos, lo cual permite que en ellos se puedan resolver problemas de forma más simple que en las gráficas en general. Una trayectoria es un tipo de árbol con estructura aún más simple: una gráfica es una *trayectoria*, si todos sus vértices, excepto a lo más dos, tienen grado exactamente dos (intuitivamente, se pueden dibujar con todos los vértices sobre una línea, y las aristas conectan vértices consecutivos sobre la línea). Esta estructura aún más simple permite resolver problemas de forma aún más fácil que en árboles.

Definición: En una gráfica G, un conjunto S de vértices es *independiente*, si para todo par de vértices $v_i, v_j \in V(G), (v_i, v_j) \notin E(G)$.

Considera el siguiente problema (sobre gráficas en general se sospecha de complejidad exponencial, pero sobre árboles se puede resolver de forma relativamente simple, y sobre trayectorias aún más).

1. 10 puntos. Problema: *conjunto independiente de peso máximo (en trayectorias)*. Entrada: Una secuencia de n pesos $W = [w_1, w_2, ..., w_n]$. Salida: un conjunto independiente (representado como la lista de índices de sus vertices) de peso máximo en la trayectoria de vértices $v_1, v_2, .., v_n$, con v_i adyacente a v_{i+1}, y siendo w_i el peso del vértice v_i.

   Ejemplos de instancias:

   (a) Entrada: [2,4,1]. Salida: [2] (un conjunto independiente de peso máximo es $\{v_2\}$).

   (b) Entrada: [8, 2, 4, 6]. Salida: [1, 4] (un conjunto independiente de peso máximo es $\{v_1, v_4\}$).

   (c) Entrada: [6,7,5]. Salida: [1,3]

   Tips: Observa que en una trayectoria, hay dos tipos de conjuntos independientes: i) los que no contienen al último vértice y ii) los que sí contienen al último vértice. Observa que los de tipo i) son exactamente los conjuntos independientes de la trayectoria que resulta de eliminar el último vértice; y observa que los de tipo ii) son exactamente los conjuntos independientes de la trayectoria que resulta de eliminar el último y el penúltimo vértice. Esto sugiere considerar los subproblemas de la forma: ¿cuál es el peso máximo de algún conjunto independiente en la subtrayectoria de vértices $v_1, v_2, ..., v_i$? Si definimos entonces $opt(i)$ como la solución del subproblema que corresponde a la subtrayectoria $v_!, ..., v_i$, la observación ya mencionada nos permite establecer la ecuación de Bellman (la que expresa el valor de la solución óptima de una subinstancia, en función de los valores óptimos de sus subinstancias recursivas), para este problema:

$$opt(i) = \begin{cases} 0 & (\text{i==0}) \\ W[i] & (\text{i==1}) \\ max(opt(i-1), opt(i-2) + W[i]) & (i > 1) \end{cases}$$

a) **3 puntos.** Propón un algoritmo recursivo (de complejidad exponencial) que resuelva el problema, en su versión del cálculo del valor óptimo, pero no de la estructura óptima. Analiza su corrección y complejidad.

Algorithm 1 Algoritmo Recursivo que devuelve el valor al problema conjunto independiente de peso máximo

```
procedure OPT(W, i)
    if i == 0 then                                   ▷ O(1)
        return 0                                     ▷ O(1)
    else if i == 1 then                              ▷ O(1)
        return W[1]                                  ▷ O(1)
    else:                                            ▷ O(1)
        opciona = OPT(W, i − 1)
        opcionb = OPT(W, i − 2) + W[i − 1]
        return max(opciona, opcionb)
    end if
end procedure
```

Ahora vamos a demostrar la corrección del algoritmo OPT por medio de inducción.

DEMOSTRACIÓN DE CORRECCIÓN

Dado un arreglo W de pesos y un índice i, el algoritmo OPT(W, i) regresa el máximo peso de un conjunto independiente en el subarreglo $W[1:i]$.

Caso base
Si $i = 0$, entonces regresamos 0, pues no hay elementos.
Si $i = 1$, entonces regresamos el único elemento $W[1]$.

Hipótesis de inducción
Supongamos que para $i < n$, el algoritmo con entradas OPT(W,i) devuelve el máximo peso de un conjunto independiente en el subarreglo $W[1:i]$.

Paso inductivo
Sea $n > 1$. Entonces hay dos tipos de conjuntos independientes: i) los que no contienen al último vértice y ii) los que sí tienen al último vértice. Los del tipo ii) son los conjuntos independientes que resultan de considerar trayectorias a las

que hemos eliminado el penúltimo vértice.

Por lo tanto, necesitamos comparar los valores máximos que nos arroja: 1) el algoritmo con entrada (W, n-1), y 2) el algoritmo con entrada (W, n-2) más el valor de W[n]. Como por hipótesis de inducción, para $i < n$, el algoritmo nos regresa el valor máximo de pesos de conjuntos independientes en $W[1 : i]$, entonces OPT(W,n-2) y OPT(W,n-1) nos regresa el valor máximo. Como tanto la opciona y opcionb nos regresan los valores correctos, regresamos el valor máximo de entre los dos. Por lo tanto el algoritmo es correcto.

COMPLEJIDAD EN TIEMPO DEL ALGORITMO RECURSIVO

Ahora, vamos a mostrar que la complejidad en tiempo del algoritmo OPT es exponencial.

Notamos que el algoritmo tiene instrucciones que toman un tiempo constante y dos llamadas recursivas. Tenemos entonces la siguiente ecuación de recurrencia:

$$T(n) = T(n-1) + T(n-2) + d$$

con $d > 0$. Esta ecuación la podemos resolver al considerar la solución a la ecuación homogénea:

$$T(n) = T(n-1) + T(n-2)$$

y una solución particular. Para la ecuación homogénea, tenemos la ecuación característica:

$$r^2 = r + 1$$

con raíces $r_1 = \frac{1+\sqrt{5}}{2}$ y $r_2 = \frac{1-\sqrt{5}}{2}$. Entonces la solución a la relación de recurrencia lineal homogénea es:

$$\begin{aligned} T_n &= \alpha_1 r_1^n + \alpha_2 r_2^n \\ &= \alpha_1 \left(\frac{1+\sqrt{5}}{2}\right)^n + \alpha_2 \left(\frac{1-\sqrt{5}}{2}\right)^n \end{aligned}$$

Para determinar α_1 y α_2, notamos que las condiciones iniciales son $T(0) = 0$, $T(1) = W[1]$.

$$\begin{aligned} 0 &= \alpha_1 + \alpha_2 \\ W[1] &= 1.618\alpha_1 - 0.618\alpha_2 \\ \implies \alpha_2 &= -\frac{W[1]}{\sqrt{5}}, \quad \alpha_1 = \frac{W[1]}{\sqrt{5}} \end{aligned}$$

Por lo tanto, la solución a la relación de recurrencia homogénea es $T(n) = \frac{W[1]}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right]$.

La solución de la ecuación de recurrencia del algoritmo es de la forma $\{T(n)^{(p)} + T(n)^{(h)}\}$ donde $T(n)^{(h)}$ es la solución a la ecuación de recurrencia homogénea y $T(n)^{(p)}$ es una solución particular a la ecuación de recurrencia original. Para encontrar una ecuación particular, supongamos que $T(n) = C$, con c una constante.

Entonces

$$\begin{aligned} & c = c + c + d \\ \implies & c = 2c + d \\ \implies & c = -d \end{aligned}$$

Por lo tanto, la solución a la ecuación de recurrencia del algoritmo OPT es

$$T(n) = \frac{W[1]}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right] - d$$

Usando notación *Big oh* tenemos que

$$\begin{aligned} T(n) &= O\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right) \\ &= O\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right) \\ &\approx O(1.6180^n) \end{aligned}$$

Por lo tanto, el algoritmo OPT es exponencial.

b) **3 puntos.**Propón la versión recursiva con memorización del ejercicio anterior. Asegúrate que devuelve el mismo valor que el algoritmo anterior, y por eso se transfiere su correción, y argumenta su complejidad contando el número máximo de subinstancias distintas en el árbol de recursión.

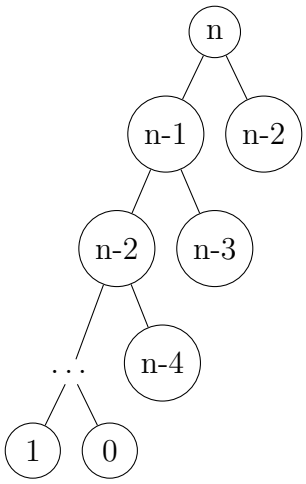
Algorithm 2 Algoritmo Recursivo con Memorización que devuelve el peso máximo de un conjunto independiente

```
M = [−1] ∗ (len(W))
procedure OPT(W, i, M)
    if M[i]! = −1 then                                          ▷ O(1)
        return M[i]                                             ▷ O(1)
    else:                                                       ▷ O(1)
        if i == 0 then                                          ▷ O(1)
            return 0                                            ▷ O(1)
        else if i == 1 then                                     ▷ O(1)
            M[1] = W[1]                                         ▷ O(1)
            return M[1]                                         ▷ O(1)
        else:                                                   ▷ O(1)
            opciona = OPT(W, i − 1, M)
            opcionb = OPT(W, i − 2, M) + W[i]
            M[i] = max(opciona, opcionb)                        ▷ O(1)
            return M[i]                                         ▷ O(1)
        end if
    end if
end procedure
```

En este algoritmo creamos un arreglo M tal que, para cada índice i, guarda el peso máximo de un conjunto independiente contenido en el subarreglo $M[i]$. Para cada índice i, el algoritmo del inciso a) y el algoritmo b) tienen la misma salida, i.e., el algoritmo del algoritmo con memorización hereda la corrección del algoritmo recursivo :

- Si $i = 0$, ambos devuelven 0.
- Si $i = 1$, ambos devuelven 1.
- Si $i > 1$, ambos devuelven el valor máximo entre la opción a y la opción b.

Complejidad en tiempo: algoritmo recursivo con memoización



Tenemos el árbol de recursión del algoritmo. En el árbol, estamos suponiendo que, primero se llama la opcion_a y después la opcion_b. Esto hace que cada vértice izquierdo tenga dos hijos, puesto que, una vez que se realizan las llamadas recursivas en el algoritmo, cuando evaluamos la opción_b, los valores ya están guardados en el arreglo M, por lo tanto, cada nodo derecho ya no tiene hijos. El número de subinstancias distintas es $n+1$, y en cada una de ellas toma un número constante de operaciones. Por lo tanto, la complejidad del algoritmo es $O(n)$.

c) 3 puntos. Propón la versión interativa de programación dinámica, del algoritmo anterior (declara las matrices M_{opt} y $M_{choices}$, establece la semántica de cada una de sus celdas, y propón un orden de llenado correcto, describe la función de llenado). Asegúrate que devuelve el mismo valor que el algoritmo anterior, y por eso se transfiere su corrección, y demuestra que su complejidad es $O(n)$.

Algorithm 3 Algoritmo Iterativo que devuelve el peso máximo de un conjunto independiente

```
procedure OptD(W)
    n = len(W)                                                          ▷ O(1)
    Mopt = [0] * (n + 1)                                                ▷ O(1)
    Mchoices = [0] * (n + 1)                                            ▷ O(1)
    Mopt[1] = W[0]                                                      ▷ O(1)
    Mchoice[0] = −1                                                     ▷ O(1)
    Mchoice[1] = 1                                                      ▷ O(1)
    if n == 0 then                                                      ▷ O(1)
        Mopt[0] = 0                                                     ▷ O(1)
        return Mopt[0]                                                  ▷ O(1)
    end if
    if n == 1 then                                                      ▷ O(1)
        Mopt[0] = 0                                                     ▷ O(1)
        Mopt[1] = W[1]                                                  ▷ O(1)
        return Mopt[1]                                                  ▷ O(1)
    else
        Mopt[0] = 0                                                     ▷ O(1)
        Mopt[1] = W[1]                                                  ▷ O(1)
        for i in range(2, n + 1) do                                     ▷ O(n)
            Mopt[i] = max(Mopt[i − 1], Mopt[i − 2] + W[i])              ▷ O(1)
            if Mopt[i] == Mopt[i − 1] then                              ▷ O(1)
                Mchoice[i − 1]+ = 1                                     ▷ O(1)
            else                                                        ▷ O(1)
                Mchoice[i − 2]+ = 1                                     ▷ O(1)
                Mchoice[i − 1]− = 1                                     ▷ O(1)
                Mchoice[i]+ = 1                                         ▷ O(1)
            end if
        end for
        return Mopt[n]                                                  ▷ O(1)
    end if
end procedure
```

La comparación de este algoritmo con el algoritmo anterior que utiliza memoización se hará sobre n, el tamaño del arreglo W, puesto que el algoritmo de programación dinámica no tiene entrada i.

Para este algoritmo, construimos dos arreglos M_{opt}, donde $M_{opt}[i]$ guardará los pesos máximos de un conjunto independiente hasta el índice i y $M_{choices}$, donde

$M_{choices}[i]$ registra la opción máxima entre la opcion_a y la opción_b para el índice i. Vamos a suponer que el índice de M_{opt} y de $M_{choices}$ inician en 0., para el caso en que W sea un arreglo vacío, de modo que el algoritmo devuelve el valor 0 para este caso.

- Si el arreglo de pesos W no tiene elementos, $n = 0$ y actualizamos $M_{opt} = 0$, $M_{choices}[0] = -1$ se queda con su valor sentinela y regresamos $M_{opt}[1]$. Misma entrada, misma salida respecto al algoritmo anterior.
- Si W tiene un elemento, $n = 1$, actualizamos M_{opt} en sus primeras dos entradas a 0 y $W[1]$ respectivamente, y las entradas de $M_{choices}$ se quedan con su valor sentinela, puesto que las elecciones tienen sentido a partir del índice 2. Regresamos $M_{opt}[1]$. Misma entrada, misma salida respecto al algoritmo anterior.
- Si $n >= 2$, actualizamos las dos primeras entradas de M_{opt} a los casos base 0 y $W[1]$ respectivamente. Ahora bien, para las entradas $i >= 2$, actualizamos $M_{opt}[i]$ al máximo entre $M_{opt}[i-1]$ y $M_{opt}[i-2] + W[i]$ que, usando la terminología del algoritmo con memoización, son las opciones opcion_a y opcion_b respectivamente. Además, si $M_{opt}[i] = M_{opt}[i-1]$, entonces la primera opción (opción a)) fue mayor y actualizamos $M_{choice}[i] = 0$. Si no, entonces actualizamos $M_{choice}[i] = 1$. Regresamos $M_{opt}[n]$. Misma entrada, misma salida respecto al algoritmo anterior.

  El orden de llenado del arreglo M_{opt} es llenarlo desde la primera entrada hasta la última. La entrada $M_{opt}[i]$ necesita el valor de las dos entradas anteriores, por lo que para la iésima entrada ya están disponibles las anteriores cuando entramos al loop del for.

  De la misma manera, el orden de llenado del arreglo $M_{choices}$ es de izquierda a derecha. Sólo que, la entrada $M_{choices}[i]$ necesita el valor de las entradas $M_{opt}[i]$ y $M_{opt}[i-1]$ que ya están disponibles en la iésima iteración del for loop.

  Por lo tanto, la correción del algoritmo recursivo con memoización se transfiere al algoritmo con programación dinámica.

Complejidad en tiempo: algoritmo iterativo

El algoritmo consiste de instrucciones de complejidad *O(1)* y un *for*. Debido a que *for* toma valores en *range(2,n+1)*, itera *n-1* veces; y las instrucciones dentro de este son de complejidad *O(1)*, por lo que tiene complejidad *O(n)*. Por lo tanto, la complejidad total del algoritmo es *O(n)*.

d) 1 punto. Propón el algoritmo que, a partir de la tabla M_{choice} generada por el algoritmo anterior, calcula el conjunto independiente de peso máximo.

Algorithm 4 Algoritmo Iterativo que devuelve el peso máximo de un conjunto independiente

```
procedure RECUPERA_SOLUCION(Mchoice, n)
    A = []                                                              ▷ O(1)
    if Mchoice[n] >= Mchoice[n − 1] then                                ▷ O(1)
        A.insert(0, n)                                                  ▷ O(1)
        n = n − 1                                                       ▷ O(1)
        for i in range(n − 1, 1, −1) do                                 ▷ O(1)
            if Mchoice[i] == max(Mchoice[i + 1], Mchoice[i], Mchoice[i − 1]) then ▷ O(1)
                A.insert(0, i)                                          ▷ O(1)
                continue                                                ▷ O(1)
            end if
        end for
        if Mchoice[1] > Mchoice[2] then                                 ▷ O(1)
            A.insert(0, 1)                                              ▷ O(1)
            return A                                                    ▷ O(1)
        end if
```