Nombre: Daniel Andrés Vasquez Murillo
Código de estudiante: 8963154

Tarea 2

Problemas conceptuales

1. Problema 15.1: Coin changing (Cormen et al., página 446).

15-1 Coin changing

Consider the problem of making change for n cents using the smallest number of coins. Assume that each coin's value is an integer.

a. Describe a greedy algorithm to make change consisting of quarters, dimes, nickels, and pennies. Prove that your algorithm yields an optimal solution.

b. Suppose that the available coins are in denominations that are powers of c: the denominations are $c^0, c^1, \ldots, c^k$ for some integers $c > 1$ and $k \geq 1$. Show that the greedy algorithm always yields an optimal solution.

c. Give a set of coin denominations for which the greedy algorithm does not yield an optimal solution. Your set should include a penny so that there is a solution for every value of n.

d. Give an $O(nk)$-time algorithm that makes change for any set of k different coin denominations using the smallest number of coins, assuming that one of the coins is a penny.

Solución

a. Para realizar una solución óptima con un algoritmo voraz primero se debe tener:

Entrada: Un numero entero N que se quiere cambiar tal que $N \geq 1$ y un arreglo A[0...H) que contenga las denominaciones de las monedas tal que $H \geq 1$

Salida: Un numero entero K tal que $K \geq 0$, que es cantidad más baja de monedas que se puede dar como cambio de N con las denominaciones de A[0…H).

Estrategia voraz: Ordenar el arreglo A de denominación de monedas de mayo a menor, y mientras N sea diferente de 0 verificar cuantas veces cabe cada una de las monedas empezando desde la posición 0 a H - 1 e ir restando a N el número de las monedas que caben multiplicado por su denominación y sumando a K el número de monedas final.

Teorema de optimización local: Sea n un valor entero positivo y A[1,5,10,25] un arreglo con las denominaciones de las monedas, sea A_n la denominación más grande, tal que $n \geq A_n$, entonces la división entera n // A_n hace parte a un cambio de monedas óptimo para n.

Demostración:

Siguiendo una demostración por contradicción supongamos que hay una solución óptima S' que es diferente de la estrategia voraz y que da una cantidad de monedas K' menor que la

obtenida por la estrategia voraz. Esto significa que K' < K, donde K es la cantidad de monedas obtenida por la estrategia voraz.

Entonces si S' utiliza un conjunto de monedas diferentes de la estrategia voraz en algún punto del proceso. Esto significa que, en algún momento, S' elige una denominación diferente de la que el enfoque voraz habría elegido.

Sea d_h la denominación elegida por S' en ese momento y d_z la denominación elegida por el enfoque voraz en ese mismo momento. Debido a que estamos asumiendo que S' es una solución óptima, debe ser el caso que d_h sea mayor que d_z. De lo contrario, si d_h fuera menor o igual a d_z, podríamos reemplazar d_h con d_z en S' y obtener una solución igual o mejor, lo cual contradice nuestra suposición de que S' es óptimo.

Sin embargo, aquí es donde llegamos a una contradicción. Si d_h es mayor que d_z, entonces el enfoque voraz debería haber elegido d_h en lugar de d_z en ese momento, ya que el enfoque voraz elige la denominación más grande que cabe en el número restante (N) en cada paso. Esto significa que S' y el enfoque voraz deberían haber elegido la misma denominación en ese momento.

Como hemos llegado a una contradicción al suponer que S' es una solución óptima que contradice la estrategia voraz, podemos concluir que la estrategia voraz proporciona la solución óptima al problema de dar cambio. Por lo tanto, no existe una solución diferente que sea mejor que la estrategia voraz, lo que demuestra por contradicción que la estrategia voraz es la solución óptima.

Seudocódigo:

```
función cambio( N , A):
    denominaciones -> ordenar_mayor(A)
    monedas_aceptadas -> 0
    numero_monedas -> 0
    iterador -> 0
    mientras N sea diferente de 0:
        monedas_aceptadas -> N // denominaciones[ iterador ]
        numero_monedas  += monedas_aceptadas
        N -= monedas_aceptadas * denominaciones[ iterador ]
        iterador += 1
```

b. Para demostrar que el algoritmo sigue produciendo una solución óptima teniendo en cuenta el algoritmo realizado en el punto a para números enteros c y k donde $c > 1$ y $k \geq 1$ se debe entonces, reformular el Teorema de optimización local:

Teorema de optimización local: Sea n un valor entero positivo y el arreglo A ahora un arreglo A[0..H) donde H es ≥ 1 un arreglo con las denominaciones de las monedas, sea A_n la denominación más grande, tal que $n \geq A_n$, entonces la división entera n // A_n hace parte a un cambio de monedas óptimo para n.

En este caso la demostración por contradicción seguiría siendo la misma pues solo hay una única solución valida, para cualquier conjunto de monedas con denominaciones construidas como se describió anteriormente, por lo que la solución que propone el algoritmo sigue siendo optima.

c. Dado el conjunto de denominaciones de monedas ordenado igual a D[6,4,1] para un numero por ejemplo de N = 8 como el algoritmo prioriza siempre el número más grande entonces daría como respuesta que K = 4 utilizando 1 monedas de 6 y 2 de 1, pero la respuesta correcta debe ser K = 2, usando 2 monedas de 4.

d.

Seudocódigo:

```
función cambio(N , A):
    denominaciones -> ordenar_mayor(A)
    monedas_aceptadas -> 0
    numero_monedas -> 0
    iterador -> 0
    mientras N sea diferente de 0:
        monedas_aceptadas -> N // denominaciones[ iterador ]
        numero_monedas  += monedas_aceptadas
        N -= monedas_aceptadas * denominaciones[ iterador ]
        iterador += 1
```

2. Ejercicio 23: Climbing (Erickson, página 184).

23. One day Alex got tired of climbing in a gym and decided to take a large group of climber friends outside to climb. They went to a climbing area with a huge wide boulder, not very tall, with several marked hand and foot holds. Alex quickly determined an "allowed" set of moves that her group of friends can perform to get from one hold to another.

The overall system of holds can be described by a rooted tree T with n vertices, where each vertex corresponds to a hold and each edge corresponds to an allowed move between holds. The climbing paths converge as they go up the boulder, leading to a unique hold at the summit, represented by the root of T.

Alex and her friends (who are all excellent climbers) decided to play a game, where as many climbers as possible are simultaneously on the boulder and each climber needs to perform a sequence of *exactly* k moves. Each climber can choose an arbitrary hold to start from, and all moves must move away from the ground. Thus, each climber traces out a path of k edges in the tree T, all directed toward the root. However, no two climbers are allowed to touch the same hold; the paths followed by different climbers cannot intersect at all.

(a) Describe and analyze a greedy algorithm to compute the maximum number of climbers that can play this game. Your algorithm is given

a rooted tree T and an integer k as input, and it should compute the largest possible number of disjoint paths in T, where each path has length k. Do ***not*** assume that T is a binary tree. For example, given the tree below as input, your algorithm should return the integer 8.

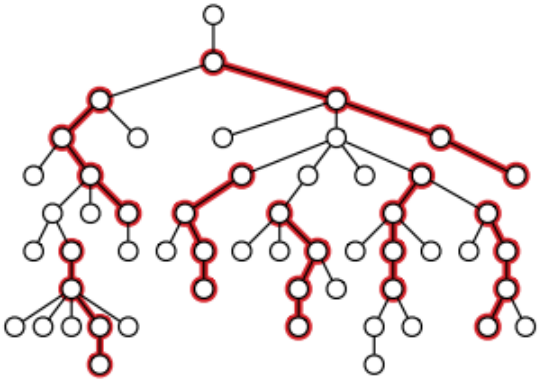


Figure 4.6. Seven disjoint paths of length $k = 3$. This is *not* the largest such set of paths in this tree.

(b) Now suppose each vertex in T has an associated *reward*, and your goal is to maximize the total reward of the vertices in your paths, instead of the total number of paths. Show that your greedy algorithm does *not* always return the optimal reward.

(c) Describe an efficient algorithm to compute the maximum possible reward, as described in part (b).

a.

Especificación del problema:

Entrada: Un numero entero K, tal que $K \geq 0$, y árbol T = (V, E) conexo

Salida: Un numero entero X, tal que $X \geq 0$ sea el máximo numero de caminos disjuntos que tiene el árbol

Estrategia voraz: Primero utilizando T se recorre por anchura hasta cubrir cada nodo del árbol de tal forma que se tenga una cola de prioridad de mayor a menor, con una tupla que contenga (profundidad, nodo) de esta forma conociendo que tan profundo este cada nodo presente en el árbol, donde el nodo raíz siempre tiene profundidad de 1; Segundo utilizando esta cola se empieza a recorrer el árbol desde el nodo más bajo hacia arriba, orientado a su nodo raíz, y mientras realiza esa acción descontando para ese camino un K_T donde K_T empieza siendo igual que K, donde lo que se busca es con cada recorrido desde el nodo hijo al padre más cercano ir construyendo un camino para 1 escalador de tal forma que mientras se pueda construir 1 camino (lo que significa que K_T termina siendo 0; no poder construir camino significa que K_T es diferente de 0 y el nodo hijo en el que estamos no puede subir más) el algoritmo borrara las conexiones de los nodos padre del último nodo al que se subió

para construir un camino y el algoritmo volverá a empezar ahora con el siguiente nodo mas profundo, tomando el hecho de que el camino que ya se tomo se separo del árbol original,

Principio de optimización local:

Dado un número entero K, tal que $K \geq 1$, un árbol $T = (V, E)$ conexo y PQ una cola de prioridad ordenada de mayor a menor con la altura de cada nodo del árbol sea R el nodo más profundo del árbol y R_p el nodo padre de R, entonces $\{R\} \cup \{R_p\}$ hacen parte un camino disjunto optimo, que se puede construir en el árbol

Demostración:

Caso base (K=0):

Cuando K es igual a cero, significa que no hay restricción en la cantidad de caminos disjuntos que se pueden tomar, por lo que el algoritmo simplemente recorrerá el árbol utilizando el recorrido por anchura sin ninguna restricción adicional. En este caso, el número de caminos disjuntos será igual al número de nodos en el árbol, que es el máximo posible. Por lo tanto, el caso base se cumple.

Caso inductivo:

Supongamos que la estrategia voraz funciona correctamente para algún valor de K. Queremos demostrar que también funciona para K+1.

Cuando K es igual a K+1, el algoritmo seguirá el siguiente proceso:

Inicializa la cola de prioridad y el árbol.

Utiliza el recorrido por anchura para marcar la profundidad de cada nodo en el árbol.

Comienza a recorrer el árbol desde el nodo más profundo hacia arriba, tratando de construir un camino.

Mientras pueda construir un camino, reducirá K_T en 1 en cada paso.

Si K_T llega a 0, elimina todas las conexiones al nodo actual y agrega 1 al contador de caminos.

Reinicia K_T a K y continúa el proceso con el siguiente nodo más profundo.

Ahora supongamos que existe una estrategia alternativa "W" que puede generar más caminos disjuntos que la estrategia voraz para K+1. Vamos a analizar lo que ocurre cuando la estrategia voraz se encuentra en un punto en el que no puede construir más caminos adicionales debido a la restricción de K_T. En este punto, el contador de caminos de la estrategia voraz se incrementa en 1, y K_T se reinicia a K.

Si "W" fuera una estrategia superior, implicaría que "W" podría construir más caminos disjuntos utilizando el mismo árbol y la misma cantidad de K_T que la estrategia voraz. Sin embargo, dado que la estrategia voraz ha tomado decisiones óptimas localmente en cada

paso hasta este punto, no es factible que "W" pueda superar la cantidad de caminos disjuntos construidos por la estrategia voraz en esta etapa del algoritmo.

Por lo tanto, la estrategia voraz maximiza el número de caminos disjuntos incluso cuando K+1, y esta conclusión se basa en que ninguna estrategia alternativa podría mejorar el resultado de la estrategia voraz en este contexto. La demostración por inducción confirma que la estrategia voraz es óptima para maximizar el número de caminos disjuntos en un árbol conexo, y esto se logra mediante un enfoque de optimización local.

realizando estas acciones hasta que no se pueda construir un camino.

Pseudocodigo:

Inicializar K, KT, cola de prioridad (PQ), árbol (T)

Definir nodo raíz (R)

Cuenta de caminos global -> 0

función set_level (R, T):

Mientras T no esté vacío:

Para cada nodo N en T adyacente a R:

Si N no ha sido visitado:

Marcar N como visitado

Añadir (profundidad(N), N) a PQ

set_level(N, T - {N})

función Camino(PQ, T, K_T):

Mientras PQ no esté vacío y $K_T > 0$:

Extraer (profundidad, nodo) de PQ

Si nodo en T tiene un padre y $K_T > 0$:

$K_T = K_T - 1$

nodo -> nodo.padre

Si $K_T == 0$:

Eliminar todas las conexiones al nodo en T

Añadir camino + 1

$K_T = K$

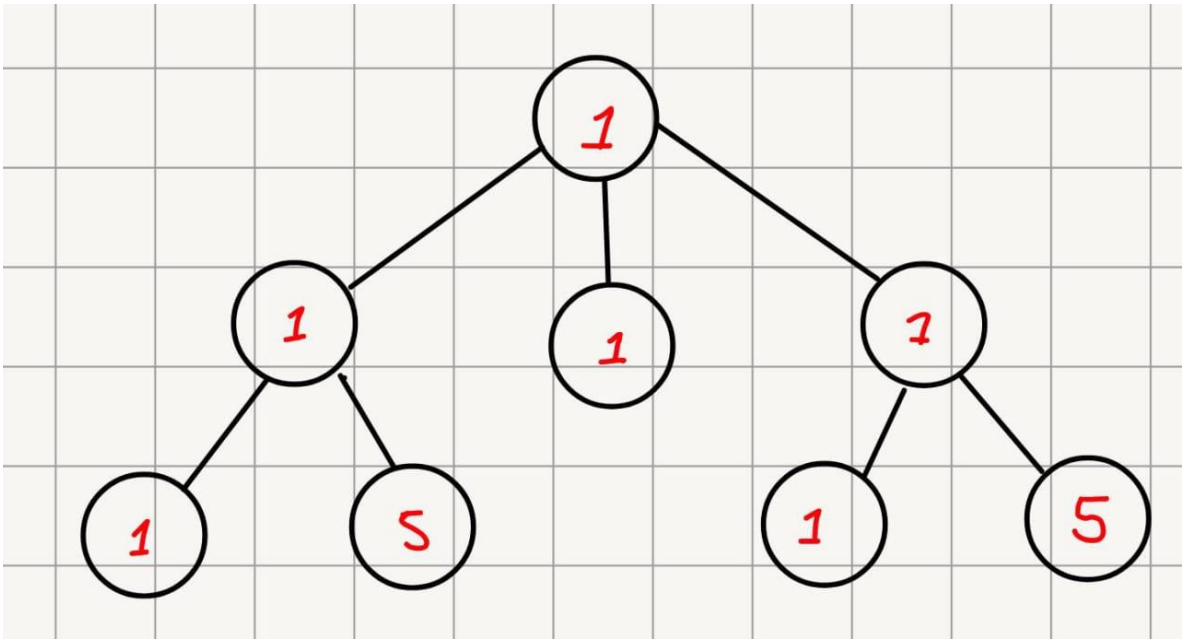
Camino(PQ, T, K_T)

función principal ():

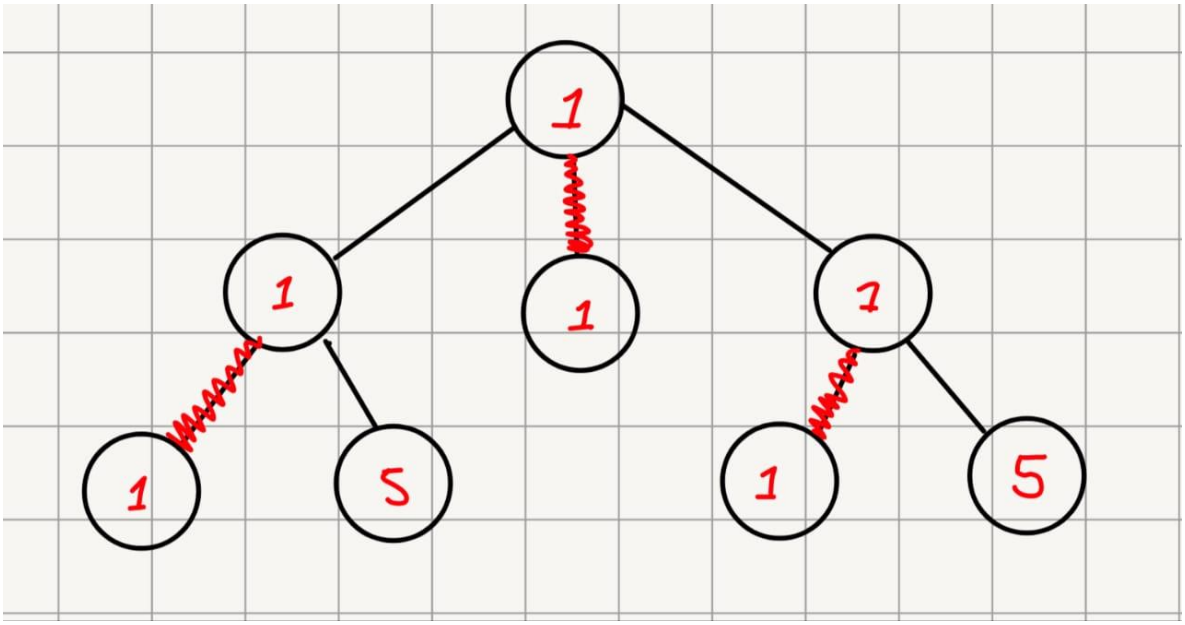
set_level(R, T)

Camino(PQ, T, K)

b. Para encontrar un ejemplo que no funcione con el algoritmo anteriormente descrito entonces dado el siguiente árbol T con un K = 1



Como el algoritmo voraz funciona seleccionara, después de calcular a cada uno de los vértices su profundidad, cualquiera de los vértices con la mayor profundidad y comenzara a realizar las verificaciones e iteraciones necesarias para empezar a construir caminos de abajo para arriba, donde la respuesta que de volvería seria 3 el cual es el número máximo de caminos que se pueden construir, pero no son los caminos con mayor valor dado a que podría tomar los siguientes caminos no óptimos:



c. Para realizar un algoritmo que ahora priorice no el numero de caminos, si no que maximice el valor de cada vértice presente en los caminos entonces se debe ya no ordenar por profundidad, ahora se debe cambiar el algoritmo para que funcione de manera dinámica, donde partiendo del nodo raíz, tenga la opción de tomar un nodo o no, siempre y cuando no se este generando un camino por esa rama, si se esta generando un camino puede coger cualquiera de los nodos hijos para seguir con el camino y los demás podrán o no comenzar otro camino, para al final devolver los caminos que tengan los valores más grandes.

Entrada: Un numero entero K, tal que $K \geq 0$, y árbol T = (V, E) conexo

Salida: Un numero entero G, tal que $G \geq 0$ siendo este el valor máximo de valores presentes en los vértices de caminos disjuntos construibles en el árbol.

función objetivo: phi(H, T, KT, R, valor) donde K es la altura que debe tener cada camino, KT la altura evaluada en la construcción de cada camino, H es la memoria, R el nodo que se esta revisando en un momento determinado, y Valor es la suma de los valores en cada camino.

Reformulación:

Entrada: Un numero entero K, tal que $K \geq 0$, y árbol T = (V, E) conexo

Salida: phi(H, T, KT, R, valor)

Seudocódigo:

Inicializar K, KT, diccionario_memoria (H), árbol (T)

Definir nodo raíz (R)

Cuenta de caminos global -> 0

```
Cuenta de valores global -> 0
función Camino(H, T, KT, R, valor):
  Si R tiene nodos hijos:
    Si R hace parte de un camino y KT > 0:
      KT = KT – 1
      Para todos los hijos de R:
        Llamar a Camino(H, T, KT, hijo, valor + R.valor)
    Si R no hace parte de un camino:
      Puede tomar a R como inicio del camino:
        KT = KT – 1
        Para todos los hijos de R:
          Llamar a Camino(H, T, KT, hijo, valor + R.valor)
      O también para todos los hijos de R:
        Llamar a Camino(H, T, KT, hijo, valor)
  Si R no tiene hijos:
    Si KT != 0:
      Guardar en el diccionario H el nodo padre del camino como llave con 0 como suma
    Si KT == 0:
      Guardar en el diccionario H el nodo padre del camino como llave con la suma de los
valores
función principal ():
  Llamar a Camino(H, T, K, R, 0)
  Devolver el valor máximo en el diccionario H
```