Resolución primer parcial 1c2021

Santiago Cifuentes

Compilado: 20 de septiembre de 2021

1. Enunciado

1. Carle quiere convertirse en le mejor speedruner de GastaVidas. GastaVidas es un juego con n niveles que deben ser pasados uno después del otro. En GastaVidas, le jugadore empieza con una sola vida y en cada nivel i puede conseguir hasta v_i vidas, para algún $0 \leq v_i \leq n$. La ventaja de conseguir muchas vidas es que las mismas pueden gastarse en los distintos niveles para acelerar la resolución del juego. Por ejemplo, si un nivel tiene un piso de espinas, le jugadore puede pasar por encima de ellas en unos pocos segundos perdiendo una vida, o puede buscar un camino alternativo y más largo sin gastar vidas. Obviamente, las vidas solo se pueden gastar luego de ser conseguidas y el juego termina una vez que le jugadore se queda sin vidas. Para ser considerade le mejor, Carle debe participar en la categoría World Class que tiene tres reglas particulares: 1. cada jugadore debe recolectar todas las vidas de todos los niveles, 2. las vidas recolectadas en el nivel i no se pueden gastar en el nivel i y, 3. en cada nivel se pueden gastar a lo sumo 5 vidas. Luego de un arduo entrenamiento, Carle ha logrado resolver cada nivel i en tiempo $t_{ip} \in \mathbb{N}$ cuando gasta p vidas, para $0 \leq p \leq 5$. Ciertamente, $t_{ip} \geq t_{ip+p'}$ para todo $0 \leq p \leq p + p' \leq 5$.

   a) Definir en forma recursiva la función $G : \{1, ..., n\} \times \mathbb{N} \to \mathbb{N}$ tal que $G(i, v)$ denota la mínima cantidad de tiempo que le toma a Carle resolver los niveles $i, ..., n$ si empieza con v vidas.

   b) Demostrar que G tiene la propiedad de superposición de subproblemas.

   c) Definir un algoritmo top-down para calcular $G(i, v)$ cuando $V = \{v_i | 1 \leq i \leq n\}$ y $T = \{t_{ip} | 1 \leq i \leq n, 0 \leq p \leq 5\}$ son dados como input, indicando claramente las estructuras de datos utilizadas y la complejidad resultante. Nota: para que el ejercicio se considere bien es necesario que la complejidad temporal del algoritmo sea $O(n^2)$.

   d) Escribir el (pseudo-)código del algoritmo top-down resultante.

2. Dado un digrafo G y dos vértices s y t, queremos encontrar un recorrido de s a t que tenga longitud par y use la menor cantidad de aristas. Modelar este problema como una instancia particular de camino mínimo que pueda ser resuelto con **BFS** (sin necesidad de modificar el algoritmo visto en clase; observar que el recorrido podría no existir). **Justificar** que el modelo propuesto resuelve el problema.

3. Decimos que un grafo es *fácil* cuando cada una de sus aristas pertenece a lo sumo a un ciclo del grafo. Diseñar un algoritmo para encontrar un árbol generador mínimo de un grafo fácil G en $O(|V(G)|)$ tiempo. El algoritmo debe ser robusto en el sentido de que debe reportar un error en caso que G no sea fácil. **Justificar** que el algoritmo propuesto es correcto. **Ayuda**: para el algoritmo, separe su algoritmo en dos partes; primero encuentre un árbol generador (apropiado) de G en tiempo lineal; luego 'corrija' el árbol para obtener uno mínimo. Para la complejidad, determine cuántas aristas puede tener un grafo fácil.

4. Diseñar un algoritmo eficiente que dado un digrafo G con pesos no negativos, dos vértices s y t y una cota c, determine una arista de peso máximo de entre aquellas que se encuentran en algún

recorrido de s a t cuyo peso (del recorrido, no de la arista) sea a lo sumo c. **Justificar** que el algoritmo propuesto es correcto.

2. Resolución

1. a) En el caso base, cuando $i = n$, lo mejor que puede hacer Carle es usar la máxima cantidad de vidas que pueda para pasar el nivel en el menor tiempo. Esto es, $G(n, v) = t_{np}$, donde p es el máximo valor tal que $0 \leq p \leq 5$ y $v - p \geq 0$ (pues Carle no puede usar más vidas de las que tiene). Notemos que este valor es exactamente $min(5, v)$. Por otro lado, si $i < n$, Carle tiene potencialmente varias opciones sobre cuántas vidas gastar para el nivel i: podrá elegir cualquier cantidad $p \in \{0, .., 5\}$ tal que $v - p \geq 0$. Tras elegir la cantidad de vidas a gastar p, pasará al siguiente nivel con $v - p + v_i$ vidas, y la estrategia óptima que debe seguir en ese estado le consumirá tiempo $G(i + 1, v - p + v_i)$. Naturalmente Carle deberá elegir entonces el p que optimice $t_{ip} + G(i + 1, v - p + v_i)$ con la condición de que $v - p \geq 0$ y $p \in \{0, ..., 5\}$. Más formalmente:

$$G(i, v) = \begin{cases} t_{n,min(5,v)} & i = n \\ \min\limits_{p \in \{0,...,5\}, v \geq p} (t_{ip} + G(i + 1, \text{mín}\{5n, v - p + v_i\})) & cc. \end{cases}$$

La cota $\text{mín}\{5n, v - p + v_i\}$ que aparece en el caso recursivo sirve para reducir la cantidad de subinstancias recursivas. La explicación es la siguiente: si en algún momento Carle tiene $5n$ vidas entonces ya tiene suficientes para gastar 5 en cada uno de los siguientes niveles, sin importar cuántas pueda recoger. Por ende da lo mismo si Carle tiene $5n$ o más vidas: en ambos casos ya puede usar la máxima cantidad de vidas posibles en todo el resto de los niveles.

b) Notemos que el árbol de recursión que se genera al hacer el llamado $G(i, v)$ puede tener un tamaño exponencial en función de n: si $v \geq 5$ en cada subllamado entonces cada nodo genera 6 llamados nuevos, por lo que el árbol tendrá una cantidad de nodos del orden de $O(6^{n-i})$. En particular, cuando $i = 1$, el tamaño del árbol será exponencial en función de n (observemos que la llamada que nos interesa es $G(1, 1)$, la cual resuelve el problema).

Por otro lado, si contamos la cantidad de llamadas distintas que se pueden hacer a G vemos que son $O((n - i)n)$: el primer parámetro está acotado en el rango $[i, n]$, mientras que el segundo lo está por $0 \leq v \leq 5n$. En particular, cuando $i = 1$, la cantidad de llamadas es $O(n^2)$.

c) Para este ejercicio supongo que el conjunto V viene representado como un vector de tamaño n y que de la misma forma T viene representado por una matriz de tamaño $n \times 6$. Así puedo obtener los valores v_i y t_{ip} en $O(1)$ y el input es de tamaño $O(n)$.

El algoritmo computa la función recursiva G definida anteriormente memoizando los resultados en una matriz M de tamaño $n \times (5n + 1)$ inicializada con $\perp$ en todas las posiciones y cambiando el llamado recursivo para hacer $G(i + 1, \text{mín}(5n, v - p + v_i))$ (y de esta forma mantener acotado el valor del segundo parámetro). Los casos base se resuelven en $O(1)$, pues solo hay que computar el valor de $\text{mín}(5, v)$. Para el caso recursivo solo hay que hacer a lo sumo 5 subllamados, por lo que cada llamado realiza $O(1)$ operaciones.

Como hay $O(n^2)$ llamados posibles que se resuelven en $O(1)$ y aparte los memoizamos la complejidad temporal del algoritmos es $O(n^2)$.

d) El algoritmo de programación dinámica que computa $G(i, v)$ es el siguiente:

Algorithm 1 Algoritmo para computar $G(i, v)$ suponiendo que V, T y M son variables globales

```
1: procedure G(i, v)
2:     n ← |V|
3:     M ← inicializarMatriz(n, 5n)
4:     return PD(i, v)
5: end procedure
6: procedure PD(i, v)
7:     if i = n then
8:         return t_{i,mín(5,v)}
9:     end if
10:    if M[i][v] ≠ ⊥ then
11:        return M[i][v]
12:    end if
13:    res ← ∞
14:    for p = 0, ..., 5 do
15:        if v ≥ p then
16:            res ← mín(res, t_ip + PD(i + 1, mín(5n, v − p + v_i)))
17:        end if
18:    end for
19:    M[i][v] ← res
20:    return res
21: end procedure
```

2. Dado $G = (V, E)$ vamos a construir una nueva instancia $G' = (V', E')$ de la siguiente forma:

   - Cada nodo de G va a estar duplicado en G'. Es decir, $V' = \{v^1 : v \in V\} \cup \{v^2 : v \in V\}$.
   - Por cada eje vw en G vamos a poner dos ejes en G': uno que vaya de la primera copia de v a la segunda de w y otro que vaya de la segunda copia de v a la primera de w. Es decir, $E = \{v^1w^2 : vw \in E\} \cup \{v^2w^1 : vw \in E\}$.

   El grafo subyacente de este digrafo es bipartito: el conjunto de nodos con superíndice 1 no tiene ejes entre sí, y lo mismo para el conjunto de nodos con superíndice 2. Por ende, todo camino entre nodos con superíndice 1 va a tener longitud par (y lo mismo para los de superíndice 2).

   Vale que todo recorrido de longitud par de G que va de s a t se corresponde con un recorrido de la misma longitud en G' de s^1 a t^1, y viceversa. Para ver esto, tomemos un camino de longitud par $v_1v_2....v_{2k}v_{2k+1}$ en G con $v_1 = s$ y $v_{2k+1} = t$. El camino $v_1^1v_2^2v_3^1...v_{2k-1}^1v_{2k}^2v_{2k+1}^1$ es un camino en G' por construcción de G' (es fácil ver que cada eje de ese camino existe en G' partiendo de la definición de G' y que $v_1v_2...v_{2k+1}$ es un camino en G). Con el mismo razonamiento podemos tomar cualquier recorrido de s^1 a t^1 en G' y construir uno 'análogo' en G teniendo en cuenta que ese recorrido deberá tener longitud par (ya que G' es bipartito y s^1 y t^1 están en la misma partición) y que si un eje v^1w^2 está en G' entonces el eje vw está en G (y lo mismo para los ejes v^2w^1).

Por ende, un camino par de longitud mínima en G de s a t se corresponde con un camino de longitud par mínimo en G' de s^1 a t^1, y viceversa. Aún mejor, como todos los caminos entre s^1 y t^1 en G' son pares entonces el camino mínimo entre s^1 y t^1 será de longitud par, y a partir de él podremos construir el camino par mínimo de G.

Si G viene dado como una lista de adyacencia podemos construir G' en tiempo lineal en G recorriendo su estructura. Luego ejecutamos BFS sobre G' y s^1, lo cual toma tiempo $O(V'+E') = O(2V + 2E) = O(V + E)$. Finalmente reconstruímos el camino entre s^1 y t^1 usando el vector genereado por BFS y lo 'traducimos' al camino en G de s a t en tiempo $O(|V|)$.

El algoritmo encuentra el camino mínimo de longitud par entre s y t en tiempo $O(|V| + |E|)$ (lineal en G).

3. Para empezar, acotemos la cantidad de ejes que tiene un grafo fácil.

   Sea $G = (V, E)$ un grafo cualquiera conexo con $|V| = n$ y T un árbol generador de G obtenido tras ejecutar DFS sobre G. Pensemos en T como un árbol enraizado en el nodo r desde donde se ejecutó DFS.

   Por lo visto en la guía, para todo eje vw de E que no esté en T vale que w es un ancestro de v en T. Por ende, cada arista de las que no están en T define un ciclo que contiene a alguna de las aristas de T.

   Como T es un árbol solo tiene $n-1$ aristas. Luego, si G es fácil, no puede tener más de n aristas aparte de las de T, pues en ese caso alguna arista estaría en más de un ciclo, contradiciendo el hecho de que G es fácil. Concluimos entonces que G, si es fácil, tiene menos de $2n$ aristas.

   Por otro lado, para encontrar un árbol generador mínimo sobre un grafo fácil G alcanza con quitar, de cada ciclo de G, una de las aristas de mayor peso. Esto no rompe la conexión de G (si es conexo) y genera un árbol de costo mínimo (cualquier otro árbol tuvo que haber surgido de quitar ejes a estos ciclos, y por lo tanto la mejor opción es sacar los ejes más pesados).

   El algoritmo para encontrar un árbol generador mínimo sobre grafos fáciles es el siguiente:

   - Hacemos DFS sobre G, parando si detectamos que ya quedaron más de n aristas fuera del árbol que estamos generando.
   - Luego, por cada eje que haya quedado afuera recorremos el ciclo que define sobre T y marcamos esas aristas como pertenecientes a un ciclo. Aparte buscamos la más pesada, la cual quitaremos de G para encontrar un árbol generador mínimo.

   Si G es un grafo fácil nunca vamos a marcar una arista como perteneciente a ciclos distintos. Si G no es fácil entonces detectaremos que marcamos una arista dos veces y reportaremos un error.

   Asumamos que nos dan G como una lista de adyacencias. El DFS de la primera parte del algoritmo hace $O(n)$ operaciones debido a la condición de parada. En la segunda parte se recorre cada eje del grafo a lo sumo una vez (en caso de que repitamos un eje el algoritmo termina), y si marcamos los ejes sobre una estructura similar a la lista de adyacencias de G haremos $O(n)$ operaciones totales. Finalmente, el algoritmo es $O(n) = O(|V(G)|)$.

4. Llamemos a las aristas que pertenecen a recorridos de s a t de costo a lo sumo c aristas candidatas.

   Dada una arista vw de G podemos decidir si es candidata calculando el peso del recorrido mínimo entre s y t que pasa por vw, si es que existe tal recorrido. Una vez que tenemos ese peso alcanza con compararlo con c para decidir si vw es una de las aristas candidatas.

Supongamos que existe el recorrido del párrafo anterior, y llamémoslo p. Como el eje vw está en p podemos separar a p en 3 partes $p = p_1\ vw\ p_2$, donde p_1 va de s a v y p_2 va de w a t. Como p es un recorrido de costo mínimo que pasa por vw debe ocurrir que el peso de p_1 sea igual a $d(s, v)$ y que el peso de p_2 sea $d(w, t)$. Caso contrario, podríamos construir un recorrido de s a t que pase por vw y tenga costo menor.

Por ende, podemos decidir si vw es una arista candidata verificando si $d(s, v) + peso(vw) + d(w, t) \leq c$. El algoritmo que resuelve el problema entonces puede recorrer los ejes de G y quedarse con la arista candidata de mayor peso.

Supongamos que el grafo de entrada G está representado como una lista de adyacencias. Para calcular las distancias $d(s, \cdot)$ y $d(\cdot, t)$ podemos usar el algoritmo de Dijkstra desde s y hacia t, respectivamente. La complejidad de Dijkstra es $O(\text{mín}\{n^2, m \log(n)\})$. Finalmente se recorren las aristas en $O(m)$ y se selecciona a la candidata de mayor peso.

La complejidad final es $O(\text{mín}\{n^2, m \log(n)\})$.