Selfies

Autor: Diego Niquefa
Solución: Diego Niquefa

Primero note que en el caso en donde sólo hay una atracción basta hacer un BFS desde la posición de origen de Mario y ver cuál es la ruta mínima para tomarle foto a una atracción, esta solución tiene complejidad temporal $\mathcal{O}(hw)$ y debería pasar la primer subtarea.

Para la segunda subtarea basta con notar que si no hay lagunas y para cada atracción $min_i = max_i = 0$ entonces la cantidad de pasos necesarios para moverse de una coordenada (x, y) a una atracción ubicada en la posición (x_i, y_i) siempre va a ser $|x - x_i| + |y - y_i|$. Por ende basta probar todos los órdenes para visitar las atracciones y luego ver cuánto cuesta visitarlas usando la fórmula propuesta. Esto se puede hacer en complejidad temporal $\mathcal{O}(n! \cdot n)$. Note que en este caso específico el problema es análogo al *Problema del Agente Viajero*[1], por ende usando programación dinámica se puede abordar en complejidad temporal $\mathcal{O}(2^n n^2)$, sin embargo esto no era necesario para resolver la segunda subtarea.

La solución que daba puntuación completa parte de notar que en cualquier parte del recorrido de Mario sólo me importa saber dos cosas: la posición actual de él y cuáles atracciones ha visitado. En este sentido puedo representar todos los estados del problema con una tripla $[x, y, visit]$, que representa que Mario está en la posición (x, y) habiéndole tomado fotos a todas las atracciones en $visit$. Note que nunca vale la pena visitar dos veces una coordenada si $visit$ no ha cambiado. $visit$ se puede representar usando máscaras de bits, donde por ejemplo $(1010)_2$ representa que Mario le ha tomado una foto a las atracciones 1 y 3, pero no a las atracciones 2 y 4.

Entonces, para explorar la solución basta recorrer los estados posibles del problema usando una BFS. Note que una cota a la cantidad de estados que puedo visitar es $2^n hw$, es decir la cantidad de triplas distintas[2]. Entonces recorriendo todos los estados con una BFS obtengo una complejidad temporal $\mathcal{O}(n2^n hw)$ que puede ser reducida a $\mathcal{O}(2^n hw)$ si se precalcula para cada casilla del mapa a cuáles atracciones les puedo tomar una foto desde dicha coordenada y se almacena esto en una máscara de bits. Note que la complejidad espacial también es $\mathcal{O}(2^n hw)$. Además como no se dan cotas individuales para h o w puede ser problemático almacenarlo en memoria sin usar demasiado espacio. Una sugerencia es convertir la matriz de lagunas en un solo arreglo y recorrerlo con cuidado, los detalles de la implementación se le dejan como ejercicio al lector.

[1] Incluso si hubiera lagunas seguiría siendo análogo, pero es importante que $min_i = max_i = 0$

[2] Esto es una cota muy alta, pero razonable. Durante la producción de los casos de entrada logramos producir una entrada que visita la mitad de los casos posibles.

Contraseñas

Autor: Rafael Mantilla
Solución: Rafael Mantilla

Este problema originalmente había sido planteado con una sola subcadena prohibida, pero logramos generalizarlo a más cadenas. Lo primero que se podía notar era que la cadena prohibida específica sí importaba a la hora de contar cuántas cadenas podía formar. Por ejemplo si quisiera contar usando fórmulas de combinatoria cuántas cadenas prohibidas hay de longitud 7, que no contengan la cadena *ciic* tendría el problema de contar dos veces la palabra *ciiciic*. Esto no se presenta si la cadena prohibida es abcd.

Ahora note que si tengo una cadena no prohibida de longitud n, puedo producir una de longitud $n+1$ si tengo cuidado que el sufijo no esté prohibido. Por ende basta contar cuántas palabras no prohibidas hay de longitud n cuyo sufijo sea prefijo de alguna cadena prohibida, para todo n (decimos que la cadena vacía λ también es prefijo de cualquier cadena). Por ejemplo para la cadena prohibida *abac* basta contar cuántas palabras de longitud n terminan con la cadena a, ab, aba o no terminan con algún prefijo de la cadena prohibida. Entonces para este ejemplo:

$$Cant_{n,pal} := \text{La cantidad de palabras de longitud} n \text{que terminan con la cadena} pal$$

Ahora es necesario considerar preguntas del siguiente tipo: ¿Si mi palabra termina en *aba* qué pasa si le añado otra letra? Por ejemplo si le añado una c se volvería prohibida, si le añado una b ahora terminaría en ab, si le añado una a ahora terminaría en a. Para el caso particular de *abac* tendría el siguiente conjunto de ecuaciones de recurrencia:

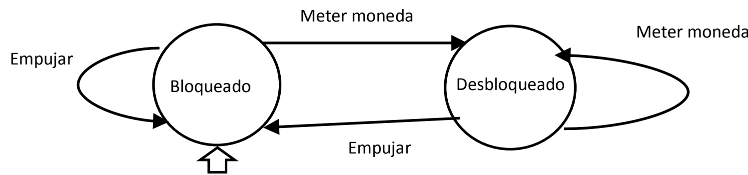
$$\begin{aligned}
Cant_(n+1,\lambda) &= 25Cant_(n,\lambda) + 24Cant_(n,a) + 25Cant_(n,ab) + 23Cant_(n,aba) \\
Cant_(n+1,a) &= Cant_(n,\lambda) + Cant_(n,a) + Cant_(n,aba) \\
Cant_(n+1,ab) &= Cant_(n,a) + Cant_(n,aba) \\
Cant_(n+1,aba) &= Cant_(n,ab)
\end{aligned}$$

Pero producir estas ecuaciones no es nada trivial. En el caso particular de tener una sola cadena prohibida sólo hay k prefijos distintos relevantes donde k es la longitud de esa cadena. Puedo entonces probar en tiempo $26k$, todos los casos y terminar con una solución $\mathcal{O}(lk)$ que sirve para la primera subtarea. En el caso particular de la segunda subtarea puedo considerar sólo cuál es la última letra que contiene la palabra, y entonces tendría una solución $\mathcal{O}(nl)$. Para resolver

esto introducimos el concepto de autómata finito con pila. La definición rigurosa de un autómata finito es una quíntupla $[Q, \sigma, q_0, \delta, F]$ donde:

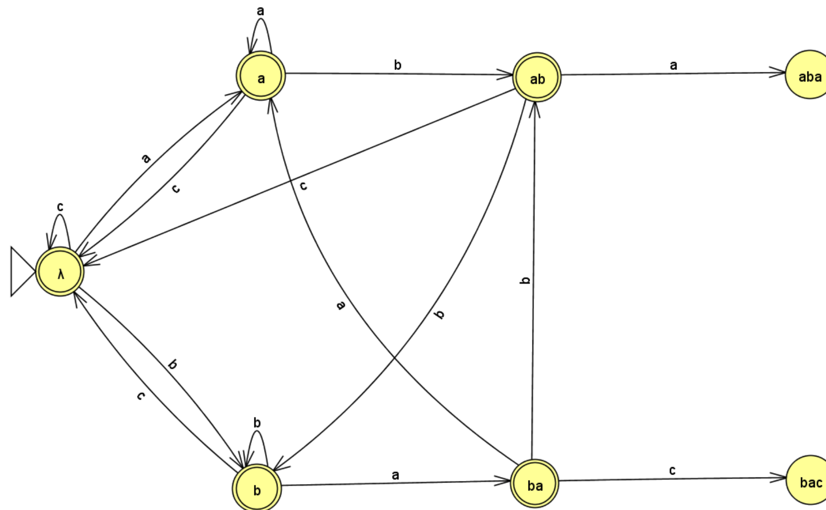
- Q es un conjunto finito de estados.
- Σ es un alfabeto finito.
- $q_0 \in Q$ es el estado inicial.
- $\delta : Q \times \Sigma \to Q$ se llama la función de transición.
- $F \subseteq Q$ se llama el conjunto de estados de aceptación.

Imagine una autómata como una máquina que de acuerdo a lo que lee cambia de estado, por ejemplo considere el siguiente autómata para un torniquete:

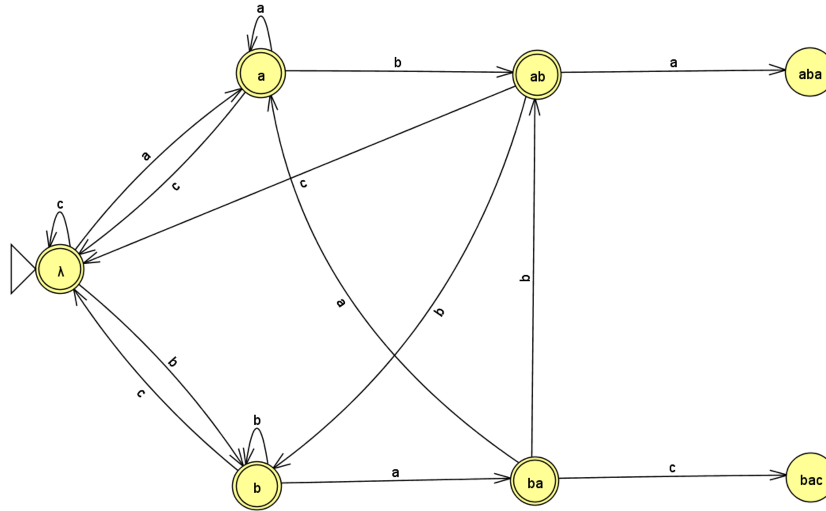


Tengo dos acciones posibles, meter una moneda o empujar y eso me cambia el estado de mi máquina.

Volviendo a nuestro problema, suponga que mi alfabeto es $\{a, b, c\}$, el siguiente autómata me reconoce todas las cadenas que no contengan la palabra aba o bac:



Con el ánimo de proveer más ejemplos, sobre el alfabeto $\{a, b\}$, el siguiente autómata me reconoce todas las cadenas que no contengan la palabra *aba*, *bab* o *bba*:



Para construir el autómata que me permita reconocer las palabras que no tienen un conjunto de cadenas dadas y luego contar cuántas palabras válidas son posibles procedo de la siguiente forma:

- Construyo un trie que me represente el conjunto de palabras. Esto lo puedo hacer en complejidad temporal $\mathcal{O}(NK)$ donde K es la longitud máxima de todas las cadenas a evitar.

- Para cada nodo del trie pruebo poner cada una de las 26 letras posibles, y luego verifico cuál es el sufijo más largo que está completamente incluido en el trie. Si hago esto de la forma más lenta posible me cuesta $\mathcal{O}(ANK^3)$, donde A es el tamaño del alfabeto, es decir 26. Esta solución no entra en tiempo, por lo que debo optimizarlo. Para hacer esto puedo precalcular para cada nodo del trie cuál es el sufijo estricto más largo de mi nodo que está en el trie (y de paso guardo cuál nodo es). Para hacer esto de manera óptima note que si quiero calcular el sufijo estricto más largo de un nodo, puedo revisar el de su padre y ver si este sumado con la letra del nodo que estoy revisando está en el trie. En caso de que no esté, reviso el sufijo más largo del sufijo más largo que tiene mi padre, si este no me sirve reviso el sufijo más largo que está en el trie de él y así sucesivamente. Una implementación de esto se puede ver en la solución oficial. Con esta optimización puedo construir todo el autómata en complejidad $\mathcal{O}(ANK^2)$. Cabe anotar que el autómata tiene a lo sumo $\mathcal{O}(NK)$ nodos.

- Por último procedo a contar cuántas palabras válidas hay. Para cada nodo

del autómata guardo cuántas palabras de longitud m hay, para calcular cuántas hay de longitud $m+1$ simplemente avanzo por cada una de las rutas posibles del autómata sumando uno a todos los nodos que puedo alcanzar desde cada uno de mis nodos, si llego a un estado prohibido del autómata no lo cuento. Cuando la longitud m llegue a l, la cantidad de palabras válidas va a ser la suma de todos los nodos del autómata que no sean nodos finales de las palabras prohibidas. Cabe anotar que es necesario hacer todos estos cálculos módulo $10?^9 + 7$. Esta forma de proceder es una forma de generalizar el sistema de ecuaciones que se dió en el ejemplo del principio. La complejidad temporal de esto es $\mathcal{O}(ANKl)$. Teniendo en cuenta que en cada paso sólo me importan los estados de longitud m y $m+1$ la complejidad espacial de esto es $\mathcal{O}(NK)$.

La complejidad temporal de todo esto suma $\mathcal{O}(ANK^2+ANKl)$ que es suficiente para pasar todos los casos de prueba. Note como detalle final que todo el problema se puede representar con un sistema de ecuaciones lineales similar al que se dió de ejemplo al principio. Usando un truco conocido para calcular la evolución de sistemas de ecuaciones lineales la complejidad se puede cambiar a $\mathcal{O}(ANK^2 + (NK)^3 \log_2 l)$. Claramente esa solución no sirve para nuestro problema, pero es interesante notar que en ese caso poco importa el tamaño de l.

Ciclas

Autor: Diego Niquefa.
Solución: Diego Niquefa, Diego Caballero, Rafael Mantilla.

El primer acercamiento al problema sugiere probar quitar cada carretera del grafo y ver si se modifica alguna distancia del grafo. Puedo hacer esto con un Floyd-Warshall que nos resulta en una complejidad $\mathcal{O}(mn^3)$ suficiente para obtener 40% de la puntuación máxima.

Ahora procedemos a demostrar la siguiente propiedad: Una carretera de u a v se puede convertir en una ciclorruta si y solo si existe otra ruta de u a v que tenga menor o igual coste.

Primero vea que si no existe una ruta de u a v que tenga menor o igual coste, esto implica que quitar la carretera de u a v empeora el mejor camino desde u hasta v, entonces claramente no lo puedo quitar.

Ahora supongamos que existe una ruta de u a v que tenga menor o igual coste. Veamos que ninguna ruta entre algún par de nodos a, b se vuelve más larga al quitar la carretera entre u y v. Entonces suponga que la ruta mínima de a a b pasa por la carretera entre u y v. Si remuevo la carretera entre u y v entonces puedo pasar ahora por la ruta de menor o igual coste, en caso de que esto me fuerce a repetir nodos, puedo acortar la ruta contradiciendo la minimalidad de mi ruta actual. Si no repito nodos, esta ruta es de menor o igual coste y no usa la carretera que removí. Por ende para probar si una carretera de u a v se puede volver una ciclorruta basta probar si existe un camino de menor o igual coste entre esos dos nodos sin usar la carretera. Se le deja como ejercicio al lector que todas las rutas que se pueden volver ciclorrutas se pueden volver simultáneamente ciclorrutas, es decir, que puedo quitarlas todas del grafo sin afectar alguna distancia mínima. Para que esto sea cierto se requiere que no haya más de una carretera entre cada par de nodos[3].

Esta observación permite hacer el siguiente algoritmo: Para cada carretera pruebo removerla y hacer un Dijkstra en sus extremos para ver si existe otra ruta con un coste menor o igual. Esta solución tiene complejidad temporal $\mathcal{O}(mn^2)$ o $\mathcal{O}(m(n\log_2 n + m))$ dependiendo de la implementación. Ambas implementaciones bastan para obtener 60% del puntaje máximo[4].

[3] Una sugerencia para esta demostración es analizar los caminos mínimos que pasan por la mayor cantidad de nodos. Esta demostración no posee ningún tipo de utilidad algorítmica y es algo larga, por lo que la demostración se omite

[4] Esta era la mejor solución que habíamos planteado, pero al realizar pruebas beta de la competencia uno de nuestros probadores encontró una mejor solución que es la que presentamos

Ahora observe el pseudocódigo de una parte del Floyd-Warshall[5]:

```
for k from 1 to |V|
   for i from 1 to |V|
      for j from 1 to |V|
         if dist[i][j] > dist[i][k] + dist[k][j]
             dist[i][j] <- dist[i][k] + dist[k][j]
         end if
```

Note que la línea `dist[i][j]` > `dist[i][k] + dist[k][j]` verifica si hay alguna ruta de i a j usando como nodo intermediario a k. Puedo entonces revisar la siguiente condición:
`dist[i][j]` >= `dist[i][k] + dist[k][j] && i!=k && k!=j`.
Si la condición se cumple esto implica que hay una ruta de i a j de coste menor o igual que no pasa por la carretera directa. Usando esa modificación puedo hacer un algoritmo que me encuentre cuántas carreteras se pueden volver ciclorruta con complejidad temporal $\mathcal{O}(n^3)$, suficiente para pasar todos los casos de prueba. Como observación final note que se le puede hacer una modificación similar al Dijkstra, con complejidad $\mathcal{O}(n(n\log_2 n + m))$. Esto es mejor para grafos dispersos, pero no es necesario para resolver el problema.

[5]Sacado de http://en.wikipedia.org/wiki/Floyd_Warshall

Sumas

Autor: Rafael Mantilla
Solución: Rafael Mantilla

Denotamos la forma de sumar de Diego con la notación $a \oplus b$. Note que probar todos los rangos y ver si suman lo deseado se puede hacer con complejidad $\mathcal{O}(m^2 n)$ suficiente para obtener 15% del puntaje total. Si tengo un número $a = a_1 a_2 \dots a_n$ y considero el siguiente número $b = b_1 b_2 \dots b_n$ donde $b_i = (10 - a_i)\%10$, ocurre que $a \oplus b = 0$. Debido a las propiedades de este número voy a decir que $b = -a$ si $a \oplus b = 0$. Se le deja como ejercicio al lector probar la siguiente propiedad interesante:

$$-(a \oplus b) = -a \oplus -b$$

Ahora para una serie de números $a_1, \dots, a_m$ y una suma deseada t deseo ver cuántas secuencias contiguas suman t bajo la forma de sumar de Diego. Defino lo siguiente:

$$\begin{aligned} s_0 &= 0 \\ s_{i+1} &= s_i \oplus a_{i+1} \end{aligned}$$

Note lo siguiente para $i \leq j$:

$a_i \oplus a_{i+1} \oplus \dots a_j = (a_1 \oplus -a_1) \oplus (a_2 \oplus -a_2) \oplus \dots a_i \oplus a_{i+1} \dots \oplus a_j$
$= (a_1 \oplus a_{i-1} \oplus a_i \oplus \dots \oplus a_j) \oplus (-a_1 \oplus \dots \oplus -a_{i-1})$
$= (a_1 \oplus \dots \oplus a_{i-1} \oplus a_i \oplus \dots \oplus a_j) \oplus -(a_1 \oplus \dots \oplus a_{i-1})$
$= s_j \oplus -s_{i-1}$

Entonces verificar cuántas secuencias contiguas distintas suman t bajo la forma de sumar de Diego basta verificar para cuantas parejas $i \leq j$ cumplen que

$$t = s_j \oplus -s_{i-1}$$

Note que esto implica que

$$s_{i-1} = s_j \oplus -t$$

Por lo que puedo hacer un algoritmo así:

- Itero i desde 1 hasta n.
- Calculo s_i.

- Miro para todos los que he calculado anteriormente cuántos de esos son iguales a $s_i \oplus -t$. Para hacer este paso óptimamente los puedo almacenar en un BBST o un map. El coste de esta consulta entonces se convierte en $\mathcal{O}(n \log m)$.

El coste total del algoritmo es $\mathcal{O}(mn \log m)$, suficiente para obtener 100

Contactos

Rafael Mantilla rjmantilla@gmail.com
Diego Niquefa niquefa.diego@gmail.com
Diego Caballero dicr1094@gmail.com