Temas Avanzados de Programación Dinámica

Agustín Santiago Gutiérrez

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Training Camp Argentina 2017

Contenidos

Contenidos

Visión tradicional

- Programación Dinámica y Divide and Conquer son dos técnicas basadas en recursión
- En ambas queremos calcular algún(os) valor(es) de f, una función recursiva
- En divide and conquer, los subproblemas son completamente independientes entre sí, de modo que podemos implementar la recursión directa y resolver
- Se suele explicar programación dinámica, como “Divide and Conquer, con memoria para reutilizar subproblemas repetidos”

Visión constructiva

- Mi forma favorita de pensar programación dinámica
- Dado el problema que queremos encarar, imaginamos un "proceso de construir una solución"
- El proceso parte de un **estado** inicial (eventualmente varios)
- Existen **transiciones** posibles, a través de las cuáles podemos pasar de un estado a otro
- El grafo de estados y transiciones define un DAG (si hay ciclos, no se puede aplicar DP)
- El estado **captura toda la información** que necesitamos **para seguir construyendo la solución** desde ese punto

Visión constructiva (cont)

- El proceso de construcción de la solución consiste en recorrer un camino por los estados, hasta llegar a algún “estado solución” (casos base)
- Nuestro plan es calcular alguna “cosa” sobre las soluciones que se pueden construir desde cada estado
- Cosas típicas:
  - Cantidad de soluciones (cantidad de caminos)
  - Solución óptima (“mejor” camino)
  - Solución lexicográficamente más chica
  - Conjunto de “valores” que puede tomar una solución (por ejemplo, conjunto de “estados solución” alcanzables desde cada estado)

Ejemplos

- Dominós en un tablero de $2 \times n$
- Cruzar la matriz
- Problema de la mochila
- “Camino en DAG” (mínimo, cantidad, etc). Con la visión constructiva, todo problema de DP es caso particular de este (como todos los anteriores)

Cómo se implementa con DP

- Vamos a calcular con DP:
  $f(e) =$ valor de la cosa que queremos calcular en el estado e
- Principio de optimalidad: Se podrá aplicar DP al problema, cuando para los estados elegidos se pueda calcular $f(e)$, a partir de los $f(e_s)$ para todos los sucesores e_s de e.
- A lo anterior nos referíamos cuando decíamos que el estado captura la información **necesaria**:
  - Siempre podemos elegir un estado que haga válido el principio del óptimo. El trivial es tomar **todo el camino** recorrido como estado
  - La gran ganancia de programación dinámica surge de olvidar casi todos los detalles sobre el camino exacto utilizado, quedándose solamente con lo importante para poder completar la solución
  - Por ejemplo para mochila, no podemos sacar del estado la capacidad restante, o no sabremos si un objeto entra o no

Visión constructiva: Ventajas

- Al reconstruir el camino, la reconstrucción lo recorre **en el orden del proceso de construcción de la solución** ("al derecho").
- Por la propiedad anterior, es relativamente simple modificarlo para encontrar:
  - La solución lexicográficamente más chica
  - La solución lexicográficamente más grande
  - Más en general, la *i*-ésima solución en orden lexicográfico (¡Notar que depende del orden! La visión constructiva ayuda a razonar en el orden en que queremos, **de entrada**)
- En mi experiencia personal subjetiva, esta forma de razonar me ayuda mucho a encontrar algoritmos de DP para problemas más difíciles.
  - Evidencia anecdótica: Cuando expliqué esta forma en una charla de DP en el TC UBA 2014, un equipo Cordobés me dijo que le encantó y que "al fin entendimos lo que nuestro coach nos decía todo el tiempo: '¡Busquen el estado! ¡Busquen el estado!' "

Contenidos

Planteo

- En todos los ejemplos que siguen, primero queremos plantear el proceso de crear una solución, como vimos
- Queremos plantearlo de forma que “cada camino hasta un estado final” lleve a una solución al problema

Conteo de soluciones

- Contar cuántas soluciones hay, es contar caminos
- Usaremos la recursión $f(e) = \sum_{e_s} f(e_s)$, con $f(e) = 1$ para los estados finales

Solución lexicográficamente más chica

- Suponemos que algunas soluciones "funcionan", y otras no. Queremos la lexicográficamente más chica que funciona
- Calculamos
  $f(e) =$ desde e se puede llegar a una solución que funciona
- Al reconstruir el camino, elegimos siempre el sucesor más chico que funciona
- Es muy común combinar esto con camino mínimo (solución óptima lexicográficamente más chica)

Solución *i*-ésima en orden lexicográfico

- Suponemos que i indexa desde 0 (la solución 0 es la que vimos antes, la lexicográficamente menor)
- En lugar de calcular solo si hay solución que funciona como en la anterior, las contamos sumando como ya vimos
- $f(e) =$ Cantidad de soluciones que funcionan desde e
- Al reconstruir el camino, si buscamos la i-ésima y la primera opción tiene $T > i$ soluciones, nos movemos a esa opción
- Sino, le restamos T a i, y verificamos lo mismo para la segunda opción.
- Seguimos así hasta mandarnos por una opción. Si ninguna funcionó, i es demasiado grande y por lo tanto no hay i-ésima
- Cuando llegamos a un estado final con $i = 0$, el camino que recorrimos describe la i-ésima solución

Contenidos

Idea

- El estado contiene un **subconjunto** $S \subseteq T$ de algún conjunto T relevante al problema
- Implementación: número de 0 a $2^n - 1$ (máscara de bits: `0` a `(1«n)-1`)
  - $\cup$ es el `|`
  - $\cap$ es el `&`
  - $\{i\}$ es `1«i`
  - T es `(1«n)-1`
  - $\emptyset$ es `0`
  - S^c es `T&(~S)` o `T-S`
  - $A - B = A \cap B^c$ es `A&(~B)` o `A&(T-S)`
- Notar que el estado podría tener más cosas, o más de un subconjunto

Contenidos

Ejemplos

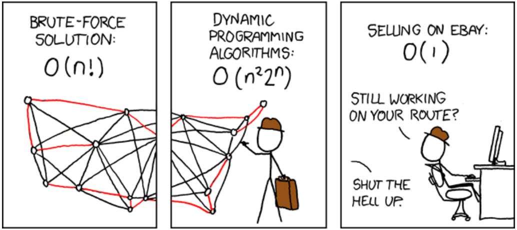
- Matching perfecto de costo mínimo en grafo completo: $O(N2^N)$

Ejemplos

- Matching perfecto de costo mínimo en grafo completo: $O(N2^N)$
- Minimum set cover: $O(M2^N)$

Ejemplos

- Matching perfecto de costo mínimo en grafo completo: $O(N2^N)$
- Minimum set cover: $O(M2^N)$
- TSP: $O(N^2 2^N)$



- Todas son **muchísimo** más eficientes que el backtracking directo

Contenidos

Idea

- Tenemos que considerar formas de “llenar” un tablero.
- Como siempre en programación dinámica, queremos “olvidar lo más posible”
- Al ir llenando el tablero, suele alcanzar con saber únicamente la situación en el **frente** por dónde vamos llenando.
- Por ejemplo si queremos llenar un tablero con 0 y 1 pero sin que haya dos 1 pegados:

$$\begin{matrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \end{matrix}$$

Idea

- Tenemos que considerar formas de "llenar" un tablero.
- Como siempre en programación dinámica, queremos "olvidar lo más posible"
- Al ir llenando el tablero, suele alcanzar con saber únicamente la situación en el **frente** por dónde vamos llenando.
- Por ejemplo si queremos llenar un tablero con 0 y 1 pero sin que haya dos 1 pegados:

0	?	?	?
?	?	?	?
?	?	?	?
?	?	?	?

Idea

- Tenemos que considerar formas de “llenar” un tablero.
- Como siempre en programación dinámica, queremos “olvidar lo más posible”
- Al ir llenando el tablero, suele alcanzar con saber únicamente la situación en el **frente** por dónde vamos llenando.
- Por ejemplo si queremos llenar un tablero con 0 y 1 pero sin que haya dos 1 pegados:

$$\begin{array}{cccc} \color{red}{0} & ? & ? & ? \\ \color{red}{1} & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \end{array}$$

Idea

- Tenemos que considerar formas de “llenar” un tablero.
- Como siempre en programación dinámica, queremos “olvidar lo más posible”
- Al ir llenando el tablero, suele alcanzar con saber únicamente la situación en el **frente** por dónde vamos llenando.
- Por ejemplo si queremos llenar un tablero con 0 y 1 pero sin que haya dos 1 pegados:

0	?	?	?
1	?	?	?
1	?	?	?
?	?	?	?

Idea

- Tenemos que considerar formas de “llenar” un tablero.
- Como siempre en programación dinámica, queremos “olvidar lo más posible”
- Al ir llenando el tablero, suele alcanzar con saber únicamente la situación en el **frente** por dónde vamos llenando.
- Por ejemplo si queremos llenar un tablero con 0 y 1 pero sin que haya dos 1 pegados:

$$\begin{array}{cccc} \color{red}{0} & ? & ? & ? \\ \color{red}{1} & ? & ? & ? \\ \color{red}{1} & ? & ? & ? \\ \color{red}{0} & ? & ? & ? \end{array}$$

Idea

- Tenemos que considerar formas de “llenar” un tablero.
- Como siempre en programación dinámica, queremos “olvidar lo más posible”
- Al ir llenando el tablero, suele alcanzar con saber únicamente la situación en el **frente** por dónde vamos llenando.
- Por ejemplo si queremos llenar un tablero con 0 y 1 pero sin que haya dos 1 pegados:

$$\begin{array}{cccc} 0 & \color{red}{1} & ? & ? \\ \color{red}{1} & ? & ? & ? \\ \color{red}{1} & ? & ? & ? \\ \color{red}{0} & ? & ? & ? \end{array}$$

Idea

- Tenemos que considerar formas de "llenar" un tablero.
- Como siempre en programación dinámica, queremos "olvidar lo más posible"
- Al ir llenando el tablero, suele alcanzar con saber únicamente la situación en el **frente** por dónde vamos llenando.
- Por ejemplo si queremos llenar un tablero con 0 y 1 pero sin que haya dos 1 pegados:

0	1	?	?
1	0	?	?
1	?	?	?
0	?	?	?

Idea

- Tenemos que considerar formas de “llenar” un tablero.
- Como siempre en programación dinámica, queremos “olvidar lo más posible”
- Al ir llenando el tablero, suele alcanzar con saber únicamente la situación en el **frente** por dónde vamos llenando.
- Por ejemplo si queremos llenar un tablero con 0 y 1 pero sin que haya dos 1 pegados:

0	1	?	?
1	0	?	?
1	0	?	?
0	?	?	?

Idea

- Tenemos que considerar formas de "llenar" un tablero.
- Como siempre en programación dinámica, queremos "olvidar lo más posible"
- Al ir llenando el tablero, suele alcanzar con saber únicamente la situación en el **frente** por dónde vamos llenando.
- Por ejemplo si queremos llenar un tablero con 0 y 1 pero sin que haya dos 1 pegados:

$$\begin{array}{cccc} 0 & \color{red}{1} & ? & ? \\ 1 & \color{red}{0} & ? & ? \\ 1 & \color{red}{0} & ? & ? \\ 0 & \color{red}{1} & ? & ? \end{array}$$

Idea

- Tenemos que considerar formas de “llenar” un tablero.
- Como siempre en programación dinámica, queremos “olvidar lo más posible”
- Al ir llenando el tablero, suele alcanzar con saber únicamente la situación en el **frente** por dónde vamos llenando.
- Por ejemplo si queremos llenar un tablero con 0 y 1 pero sin que haya dos 1 pegados:

0	1	0	?
1	0	?	?
1	0	?	?
0	1	?	?

Idea

- Tenemos que considerar formas de “llenar” un tablero.
- Como siempre en programación dinámica, queremos “olvidar lo más posible”
- Al ir llenando el tablero, suele alcanzar con saber únicamente la situación en el **frente** por dónde vamos llenando.
- Por ejemplo si queremos llenar un tablero con 0 y 1 pero sin que haya dos 1 pegados:

0	1	0	?
1	0	0	?
1	0	?	?
0	1	?	?

Estado

- El estado tendrá la posición (x, y) actual en el tablero.
- Además, para un tablero de $N \times M$, guardará N valores (o a veces $N + 1$) con el frente.
- Si los valores son binarios como en el ejemplo hay $O(NM2^N)$ estados, pero en cambio hay 2^{NM} tableros.

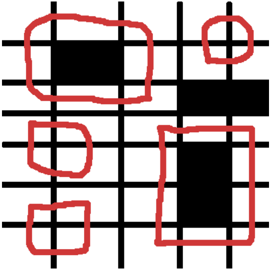
Contenidos

Ejemplos

- Cantidad de maneras de cubrir con dominós, un tablero con agujeros en posiciones dadas

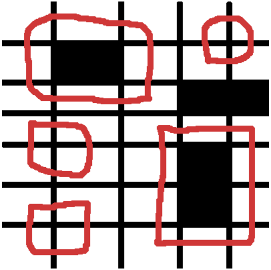
Ejemplos

- Cantidad de maneras de cubrir con dominós, un tablero con agujeros en posiciones dadas
- Cantidad de maneras de cubrir con “tuberías” cerradas (ciclos), un tablero con agujeros en posiciones dadas



Ejemplos

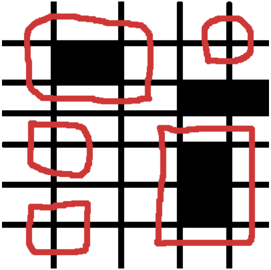
- Cantidad de maneras de cubrir con dominós, un tablero con agujeros en posiciones dadas
- Cantidad de maneras de cubrir con “tuberías” cerradas (ciclos), un tablero con agujeros en posiciones dadas



- El frente anterior tiene $O(2^N)$ valores posibles. ¿Y si quisiéramos saber la mínima cantidad de ciclos necesarios?

Ejemplos

- Cantidad de maneras de cubrir con dominós, un tablero con agujeros en posiciones dadas
- Cantidad de maneras de cubrir con “tuberías” cerradas (ciclos), un tablero con agujeros en posiciones dadas



- El frente anterior tiene $O(2^N)$ valores posibles. ¿Y si quisiéramos saber la mínima cantidad de ciclos necesarios?
- Se puede con un frente con $O(3^N)$ valores posibles.

Contenidos

Ejemplo motivador

- Dados n valores enteros distintos v_i, junto a sus frecuencias f_i, dar un árbol binario de búsqueda óptimo para los valores.

Ejemplo motivador

- Dados n valores enteros distintos v_i, junto a sus frecuencias f_i, dar un árbol binario de búsqueda óptimo para los valores.
- $dp(i,j) = \min_{k=i}^{j-1} dp(i,k) + dp(k+1,j) + sum_f(i,j)$
- Complejidad: $O(n^3)$
- ¿Se podrá mejorar?

Contexto

- ¡Sí! Con la optimización de Knuth
- Dado un algoritmo de dp en rangos cualquiera, es decir $dp(i,j)$ con $0 \leq i \leq j \leq N$
- Si su recursión tiene la forma $dp(i,j) = \min_{k=i}^{j-1} g(i,k,j)$ para cierta g que solo usa los rangos $dp(a,b)$ contenidos en (i,j)
- Podemos definir $K(i,j)$ como el menor k en donde se alcanza el mínimo de la expresión para $dp(i,j)$

Condición de Knuth

- $K(i, j-1) \leq K(i, j) \leq K(i+1, j)$
- En criollo:
  - Si agregamos un elemento por **izquierda**, el K se mueve **a la izquierda**
  - Si agregamos un elemento por **derecha**, el K se mueve **a la derecha**
- Llamamos a la anterior la *Condición de Knuth*
- Suele ser mucho más difícil demostrar que se cumple, que convencerse o intuir que así será

Optimización de Knuth

- Como vale la condición de Knuth, es muy simple cambiar en el código la recursión usando las cotas para iterar menos:
- $dp(i,j) = \min_{k=K(i,j-1)}^{K(i+1,j)} g(i,k,j)$
- Los K los podemos ir calculando en el mismo algoritmo junto a los valores *dp*.
- Las cotas sirven para iterar menos... ¿Pero estamos mejorando la complejidad asintótica?

Optimización de Knuth

- Como vale la condición de Knuth, es muy simple cambiar en el código la recursión usando las cotas para iterar menos:
- $dp(i,j) = \min_{k=K(i,j-1)}^{K(i+1,j)} g(i,k,j)$
- Los K los podemos ir calculando en el mismo algoritmo junto a los valores *dp*.
- Las cotas sirven para iterar menos... ¿Pero estamos mejorando la complejidad asintótica?
- Teorema: Con este sencillísimo cambio al código básico, el algoritmo es $O(N^2)$
- Demostración: La sumatoria de los costos es telescópica en 2D
- Si evaluar g no es $O(1)$, el costo son $O(N^2)$ evaluaciones de g

Ejercicio

- Ejercicio: Verificar que la condición de Knuth aplica en la recursión que vimos antes
- Intuitivamente tiene muchísimo sentido, ¡pero demostrarlo es mucho más difícil que programarlo!

Contenidos

Ejemplo motivador

- Dados n valores x_i enteros positivos, el costo de un intervalo $[i, j)$ es $\sum_{i \leq a < b < j} x_a x_b$. O sea, sumar los productos de a pares.
- Particionar el arreglo $[0, n)$ en k intervalos, minimizando la suma de los k costos.

Ejemplo motivador

- Dados n valores x_i enteros positivos, el costo de un intervalo $[i, j)$ es $\sum_{i \leq a < b < j} x_a x_b$. O sea, sumar los productos de a pares.
- Particionar el arreglo $[0, n)$ en k intervalos, minimizando la suma de los k costos.
- $dp(n, k) = \min_{i=0}^{n-1} dp(i, k-1) + val(i, n)$
- Complejidad: $O(n^2 k)$
- ¿Se podrá mejorar?

Contexto

- ¡Sí! Con la optimización de Divide and Conquer
- Dado un algoritmo de dp "de particionar" cualquiera, es decir $dp(n, k)$ con $0 \leq n \leq N$ y $0 \leq k \leq K$
- Si su recursión tiene la forma $dp(n, k) = \min_{i=0}^{n-1} g(n, k, i)$ para cierta g que solo usa los $dp(j, k-1)$
- Podemos definir $I(n, k)$ como el menor i en donde se alcanza el mínimo de la expresión para $dp(n, k)$

Condición de Divide and Conquer

- $I(n, k) \leq I(n+1, k)$
- Para cada k, el I es creciente en n.
- En criollo: si para k fijo agrando el rango, el último punto de corte también (mejor dicho: no retrocede)
- Llamamos a la anterior la *Condición de Divide and Conquer*
- Igual que antes, suele ser mucho más difícil demostrar que se cumple, que convencerse o intuir que así será

Optimización de Divide and Conquer

- Esta optimización no es tan simple de implementar como la de Knuth, pero la idea también es sencilla.
- Supongamos que para calcular todos los $dp(n, k)$ para k fijo, calculamos primero el $dp(n', k)$:
  - Para los $n > n'$, alcanza con probar el i **desde** $I(n', k)$
    Es decir, no más de $L_1 = n - I(n', k) + 1$ valores
  - Para los $n < n'$, alcanza con probar el i **hasta** $I(n', k)$
    Es decir, no más de $L_2 = I(n', k) + 1$ valores
- Esto nos parte el rango $[0, N)$ que debíamos calcular en dos restantes: $[0, n')$ y $[n' + 1, N)$.
- En la primera parte hay que probar hasta L_1 valores, y en la segunda hasta L_2 valores.

Si profundizamos...

- Podemos seguir partiendo estos rangos en dos recursivamente
- En el paso k tendremos 2^k rangos, cada uno con hasta L_i opciones factibles para el i.
- Observación: En cada paso, los L_i suman $O(N)$
- Por lo tanto, procesar cada paso es $O(N)$
- Partiendo siempre a la mitad, serán $O(\lg N)$ pasos y la complejidad es $O(N \lg N)$
- El algoritmo final resulta costar $O(KN \lg N)$ evaluaciones de g

Ejercicio

- Ejercicio: Verificar que la condición de Divide and Conquer aplica en la recursión que vimos antes
- Pasa lo mismo que antes: Es más fácil intuirlo que probarlo