Programación Dinámica

Agustín Santiago Gutiérrez

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Septiembre 2013

Soluciones recursivas a problemas

- Muchos algoritmos de utilidad son recursivos: para resolver un problema, se utilizan las soluciones a subproblemas fuertemente relacionados.
- En estos algoritmos, se divide el problema en varios subproblemas que luego se resuelven y se combinan las soluciones obtenidas para resolver el original.
- Un ejemplo de técnica recursiva de diseño de algoritmos es la técnica de Divide and Conquer, vista en algoritmos 2.

¿En qué consiste la programación dinámica?

- La programación dinámica es una técnica de solución de problemas recursiva.
- Al igual que Divide and Conquer, la técnica propone descomponer el problema a resolver en **subproblemas más pequeños de la misma especie**, para resolverlos recursivamente y combinar esas soluciones en una solución al problema original.
- La diferencia esencial que lo contrasta con Divide and Conquer, es que mientras que en esta técnica los subproblemas que se resuelven son independientes entre sí y se resuelven individualmente, la programación dinámica es aplicable cuando los subproblemas **no son independientes**.
- En estos casos, un algoritmo de Divide and Conquer realizaría el mismo trabajo múltiples veces, ya que la solución a un mismo subproblema puede ser **recalculada** muchas veces si se la reutiliza como parte de varios subproblemas más grandes.

¿En qué consiste la programación dinámica? (2)

- La solución que propone la técnica de programación dinámica es **almacenar** las soluciones a subproblemas ya calculadas, de manera de calcularlas una sola vez, y luego leer el valor ya calculado cada vez que se lo vuelve a necesitar.
- Uno de los usos más importantes de esta técnica es en problemas de **optimización**: En estos problemas interesa encontrar la solución que maximiza un cierto puntaje u objetivo, en un espacio de soluciones posibles.
- Un indicador central de la aplicabilidad de la técnica lo constituye el **principio del óptimo**. Este principio afirma que **las partes de una solución óptima** a un problema, deben ser **soluciones óptimas de los correspondientes subproblemas**, y es lo que permite obtener una solución óptima al problema original a partir de soluciones óptimas de los subproblemas.

El esquema general

Los algoritmos de programación dinámica se pueden organizar típicamente en 4 pasos que responden al siguiente esquema general:

1. Caracterizar la estructura de una solución óptima.
2. Definir recursivamente el valor de una solución óptima.
3. Computar el **valor** de una solución óptima. Se calcula de manera *bottom-up*.
4. Construir una solución óptima a partir de la información obtenida en el paso 3

El paso 4 es optativo, ya que si solo nos interesa el valor o puntaje de una solución óptima pero no la solución en sí, este paso de reconstrucción no es necesario.

Problema del recorrido óptimo en una matriz

Ejercicio 3.8, práctica 3 de Algoritmos y Estructuras de Datos 3:

Sea $M \in \mathbb{N}^{m \times n}$ una matriz de números naturales. Se desea obtener un camino que empiece en la casilla superior izquierda $(1, 1)$, termine en la casilla inferior derecha (m, n) y tal que minimice la suma de los valores de las casillas por las que pasa. En cada casilla (i, j) hay dos movimientos posibles: ir hacia abajo (a la casilla $(i + 1, j)$), o ir hacia la derecha (a la casilla $(i, j + 1)$).

a. Diseñar un algoritmo eficiente basado en programación dinámica que resuelva este problema.
b. Determinar la complejidad del algoritmo propuesto (temporal y espacial).
c. Exhibir el comportamiento del algoritmo sobre la matriz que aparece a continuación.

$$\begin{bmatrix} 2 & 8 & 3 & 4 \\ 5 & 3 & 4 & 5 \\ 1 & 2 & 2 & 1 \\ 3 & 4 & 6 & 5 \end{bmatrix}$$

Fórmula recursiva

La matriz de resultados parciales almacena en $best(i,j)$ la mínima longitud de un camino que empiece en (i,j) y llegue a (m,n), haciendo solo movimientos hacia abajo y hacia la derecha.

- $best(i,j) = M_{i,j} + min(best(i+1,j), best(i,j+1))$ para $1 \leq i < m$ y $1 \leq j < n$
- $best(i,n) = M_{i,n} + best(i+1,n)$ para $1 \leq i < m$
- $best(m,j) = M_{m,j} + best(m,j+1)$ para $1 \leq j < n$
- $best(m,n) = M_{m,n}$ para $i = m$ y $1 \leq j < n$

La longitud del mínimo camino entre esquinas, que constituye la solución al problema, viene dada por $best(1,1)$. Con esto ya podríamos implementar una solución top-down recursiva:

- Si para calcular un $best(i,j)$ necesitamos un resultado ya calculado, lo usamos directamente.
- Sino, lo calculamos recursivamente, almacenamos su valor en la tabla de resultados y luego lo utilizamos.

Algoritmo bottom-up

```
longitudCaminoMinimo(Matriz, m, n):
    best <- matrizDeEnteros(m,n)
    best[m,n] = Matriz[m,n]
    FOR i <- m-1 DOWNTO 1 DO
        best[i,n] = Matriz[i,n] + best[i+1,n]
    FOR j <- n-1 DOWNTO 1 DO
        best[m,j] = Matriz[m,j] + best[m,j+1]
    FOR i <- m-1 DOWNTO 1 DO
        FOR j <- n-1 DOWNTO 1 DO
            best[i,j] = Matriz[i,j] + min(best[i+1,j] + best[i,j+1])
    RETURN best[1,1]
```

La complejidad del algoritmo resultante es $O(nm)$, tanto espacial como temporal. Se puede bajar la complejidad espacial a $O(min(n, m))$ si no interesa reconstruir el camino sino solo su longitud.

Cálculo en el ejemplo

Matriz de entrada:

$$\begin{bmatrix} \mathbf{2} & 8 & 3 & 4 \\ \mathbf{5} & 3 & 4 & 5 \\ \mathbf{1} & \mathbf{2} & \mathbf{2} & \mathbf{1} \\ 3 & 4 & 6 & \mathbf{5} \end{bmatrix}$$

Matriz de best:

$$\begin{bmatrix} \mathbf{18} & 21 & 15 & 15 \\ \mathbf{16} & 13 & 12 & 11 \\ \mathbf{11} & \mathbf{10} & \mathbf{8} & \mathbf{6} \\ 18 & 15 & 11 & \mathbf{5} \end{bmatrix}$$

En ambas matrices, se indica un camino óptimo en negrita.

Referencias

- Introduction to Algorithms, 2nd Edition. MIT Press. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein Página 323: 15 Dynamic Programming