UNIDAD 7

La programación dinámica

La programación dinámica es una técnica para resolver problemas a partir de la solución e subproblemas y la combinación de esas soluciones. Un algoritmo que sigue esta técnica resuelve cada subproblema una sola vez y guarda su respuesta en una tabla.

La programación dinámica se aplica para resolver problema de optimización:

- Problemas en los que se pueden encontrar muchas soluciones
- Cada solución tiene un valor asociado
- Se busca una solución que tenga un valor asociado que sea óptimo (máximo o mínimo) entre las muchas soluciones que pueden existir

Desarrollar un algoritmo usando programación dinámica consta de los siguientes pasos:

- Caracterizar la estructura de la solución óptima
- Definir recursivamente el valor de una solución óptima
- Calcular el valor de una solución óptima de manera bottom-up
- Construir una solución óptima a partir de la información calculada

Introducción

Una subestructura óptima significa que se pueden usar soluciones óptimas de subproblemas para encontrar la solución óptima del problema en su conjunto. Por ejemplo, el camino más corto entre dos vértices de un grafo se puede encontrar calculando primero el camino más corto al objetivo desde todos los vértices adyacentes al de partida, y después usando estas soluciones para elegir el mejor camino de todos ellos. En general, se pueden resolver problemas con subestructuras óptimas siguiendo estos tres pasos:

1. Dividir el problema en subproblemas más pequeños.
2. Resolver estos problemas de manera óptima usando este proceso de tres pasos recursivamente.
3. Usar estas soluciones óptimas para construir una solución óptima al problema original.

Los subproblemas se resuelven a su vez dividiéndolos en subproblemas más pequeños hasta que se alcance el caso fácil, donde la solución al problema es trivial.

Decir que un problema tiene subproblemas superpuestos es decir que se usa un mismo subproblema para resolver diferentes problemas mayores. Por ejemplo, en la sucesión de Fibonacci ($F_3 = F_1 + F_2$ y $F_4 = F_2 + F_3$) calcular cada término supone calcular F_2. Como para calcular F_5 hacen falta tanto F_3 como F_4, una mala implementación para calcular F_5acabará calculando F_2 dos o más veces. Esto sucede siempre que haya subproblemas superpuestos: una buena implementación puede acabar desperdiciando tiempo recalculando las soluciones óptimas a problemas que ya han sido resueltos anteriormente.

Esto se puede evitar guardando las soluciones que ya hemos calculado. Entonces, si necesitamos resolver el mismo problema más tarde, podemos obtener la solución de la lista de soluciones calculadas y reutilizarla. Este acercamiento al problema se

llama memorización (en inglés "memorization"). Si estamos seguros de que no volveremos a necesitar una solución en concreto, la podemos descartar para ahorrar espacio. En algunos casos, podemos calcular las soluciones a problemas que de antemano sabemos que vamos a necesitar.

En resumen, la programación hace uso de:

- Subproblemas superpuestos
- Subestructuras óptimas
- Memorización

La programación toma normalmente uno de los dos siguientes enfoques:

- Top-down: El problema se divide en subproblemas, y estos se resuelven recordando las soluciones por si fueran necesarias nuevamente. Es una combinación de memorización yrecursión.

- Bottom-up: Todos los problemas que puedan ser necesarios se resuelven de antemano y después se usan para resolver las soluciones a problemas mayores. Este enfoque es ligeramente mejor en consumo de espacio y llamadas a funciones, pero a veces resulta poco intuitivo encontrar todos los subproblemas necesarios para resolver un problema dado.

Originalmente, el término de programación dinámica se refería a la resolución de ciertos problemas y operaciones fuera del ámbito de la Ingeniería Informática, al igual que hacía laprogramación lineal. Aquel contexto no tiene relación con la programación en absoluto; el nombre es una coincidencia. El término también lo usó en los años 40 Richard Bellman, un matemático norteamericano, para describir el proceso de resolución de problemas donde hace falta calcular la mejor solución consecutivamente.

Algunos lenguajes de programación funcionales, sobre todo Haskell, pueden usar la memorización automáticamente sobre funciones con un conjunto concreto de

argumentos, para acelerar su proceso de evaluación. Esto sólo es posible en funciones que no tengan efectos secundarios, algo que ocurre en Haskell pero no tanto en otros lenguajes.

Principio de optimalidad

Cuando hablamos de optimizar nos referimos a buscar alguna de las menores soluciones de entre muchas alternativas posibles. Dicho proceso de optimización puede ser visto como una secuencia de decisiones que nos proporcionan la solución incorrecta. Si, dada una subsecuencia de decisiones, siempre se conoce cuál es la decisión que debe tomarse a continuación para obtener la secuencia óptima, el problema es elemental y se resuelve trivialmente tomando una decisión detrás de otra, lo que se conoce como estrategia voraz.

A menudo, aunque no sea posible aplicar la estrategia voraz, se cumple el principio de optimalidad de Bellman que dicta que «dada una secuencia óptima de decisiones, toda subsecuencia de ella es, a su vez, óptima». En este caso sigue siendo posible el ir tomando decisiones elementales, en la confianza de que la combinación de ellas seguirá siendo óptima, pero será entonces necesario explorar muchas secuencias de decisiones para dar con la correcta, siendo aquí donde interviene la programación dinámica.

Contemplar un problema como una secuencia de decisiones equivale a dividirlo en problemas más pequeños y por lo tanto más fáciles de resolver como hacemos en Divide y Vencerás, técnica similar a la de programación dinámica. La programación dinámica se aplica cuando la subdivisión de un problema conduce a:

- Una enorme cantidad de problemas.
- Problemas cuyas soluciones parciales se solapan.

- Grupos de problemas de muy distinta complejidad

Sucesión de Fibonacci

Esta sucesión puede expresarse mediante la siguiente recurrencia:

$$Fib(n) = \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ Fib(n-1) + Fib(n-2) & \text{si } n > 1 \end{cases}$$

Una implementación de una función que encuentre el **n**-ésimo término de la sucesión de Fibonacci basada directamente en la definición matemática de la sucesión realizando llamadas recursivas hace mucho trabajo redundante, obteniéndose una complejidad exponencial:

```
FUNC fib(↓n: NATURAL): NATURAL
INICIO
   SI n = 0 ENTONCES
      DEVOLVER 0
   SINOSI n = 1 ENTONCES
      DEVOLVER 1
   SINO
      devolver fib(n-1) + fib(n-2)
   FINSI
FIN
```

Si llamamos, por ejemplo, a fib(5), produciremos un árbol de llamadas que contendrá funciones con los mismos parámetros varias veces:

1. fib(5)
2. fib(4) + fib(3)
3. (fib(3) + fib(2)) + (fib(2) + fib(1))
4. ((fib(2) + fib(1)) + (fib(1) + fib(0))) + ((fib(1) + fib(0)) + fib(1))
5. (((fib(1) + fib(0)) + fib(1)) + (fib(1) + fib(0))) + ((fib(1) + fib(0)) + fib(1))

En particular, fib(2) se ha calculado dos veces desde cero. En ejemplos mayores, se re calculan muchos otros valores de fin, o *su problemas*.

Para evitar este inconveniente, podemos resolver el problema mediante programación dinámica, y en particular, utilizando el enfoque de memorización (guardar los valores que ya han sido calculados para utilizarlos posteriormente). Así, rellenaríamos una tabla con los resultados de los distintos su problemas, para reutilizar los cuando haga falta en lugar de volver a calcularlos. La tabla resultante sería una tabla unidimensional con los resultados desde 0 hasta n.

Un programa que calculase esto, usando Botón-up, tendría la siguiente estructura:

```
FUNC Fibonacci (↓n: NATURAL): NATURAL
VARIABLES
   tabla: ARRAY [0..n] DE NATURAL
   i: NATURAL
INICIO
   SI n <= 1 ENTONCES
      devolver 1
   SINO
```

```
    tabla[0] ← 1
    tabla[1] ← 1
    PARA i ← 2 HASTA n HACER
      tabla[i] ← tabla[i-1] + tabla[i-2]
    FINPARA
    devolver tabla[n]
  FINSI
FIN
```

La función resultante tiene complejidad O(*n*), en lugar de exponencial.

Otro nivel de refinamiento que optimizaría la solución sería quedarnos tan sólo con los dos últimos valores calculados en lugar de toda la tabla, que son realmente los que nos resultan útiles para calcular la solución a los su problemas.

El mismo problema usando Top-don tendría la siguiente estructura:

```
FUNC Fibonacci (↓n: NATURAL, ↕tabla: ARRAY [0..n] DE NATURAL): NATURAL
  VARIABLES
    i: NATURAL
  INICIO
    SI n <= 1 ENTONCES
      Devolver 1
    FINSI
    SI tabla [n-1] = -1 ENTONCES
      Tabla[n-1] ← Fibonacci(n-1, tabla)
    FINSI
    SI tabla [n-2] = -1 ENTONCES
      Tabla [n-2] ← Fibonacci(n-2, tabla)
    FINSI
```

Tabla[n] ← tabla[n-1] + tabla[n-2]

Devolver tabla[n]

FIN

Suponemos que la tabla se introduce por primera vez correctamente inicializada, con todas las posiciones con un valor inválido, como por ejemplo -1, que se distingue por no ser uno de los valores que computa la función.

Coeficientes binomiales

El algoritmo recursivo que calcula los coeficientes binomiales resulta ser de complejidad exponencial por la repetición de los cálculos que realiza. No obstante, es posible diseñar un algoritmo con un tiempo de ejecución de orden O(nk) basado en la idea del Triángulo de Pascal, idea claramente aplicable mediante programación dinámica. Para ello es necesaria la creación de una tabla bidimensional en la que ir almacenando los valores intermedios que se utilizan posteriormente.

La idea recursiva de los coeficientes binomiales es la siguiente:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \quad \text{si } 0 < k < n$$

$$\binom{n}{0} = \binom{n}{n} = 1$$

La idea para construir la tabla de manera eficiente y sin valores inútiles es la siguiente:

El siguiente algoritmo memorizado de estrategia Botón-up tiene complejidad poli nómica y va rellenando la tabla de izquierda a derecha y de arriba abajo

Características de un Problema de Programación Dinámica

Para que un problema pueda ser resuelto con la técnica de programación dinámica, debe cumplir con ciertas características:

Naturaleza secuencial de las decisiones: El problema puede ser dividido en etapas.

Cada etapa tiene un número de estados asociados a ella.

La decisión ´optima de cada etapa depende solo del estado actual y no de las decisiones anteriores.

La decisión tomada en una etapa determina cual será el estado de la etapa siguiente.

En síntesis, la política optima es de un estado s de la etapa k a la etapa final esta constituida por una decisión que transforma s en un estado s De la etapa k + 1 y por la política optima desde el estado s hasta la etapa final.

Resolución de un Problema de Programacion Dinámica

Para resolver un problema de programación dinámica debemos al menos:

Identificación de etapas, estados y variable de decisión

Cada etapa debe tener asociado una o mas decisiones (problema de optimización), cuya dependencia de las decisiones anteriores esta dada exclusivamente por las variables de estado.

• Cada estado debe contener toda la información relevante para la toma de decisión Asociada al período.

• Las variables de decisión son aquellas sobre las cuales debemos definir su valor

De modo de optimizar el beneficio acumulado y modificar el estado de la próxima

Etapa.

Descripción de ecuaciones de recurrencia: Nos deben indicar como se acumula

La función de beneficios a optimizar (función objetivo) y como varían las funciones de estado de una etapa a otra.

Resolución Debemos optimizar cada sub problema por etapas en función de los resultados de la resolución del sub problema siguiente. Notar que las para que las recurrencias estén bien definidas requerimos de condiciones de borde.

Realizado por los creadores del sitio:

http://www.unefa-io.webnode.es

Correspondiente a la unidad 7.

Investigación de Operaciones