AED Tema 9. Estategias algorítmicas

Martín González Dios

28 de noviembre de 2024

1. Divide y vencerás

Consiste en **descomponer un problema en un conjunto de problemas más pequeños** que se puedan resolver de forma independiente y combinar sus soluciones para obtener la solución al problema original. Los subproblemas se pueden ir dividiendo a su vez en problemas más pequeños hasta llegar a una solución trivial.

Un claro **problema** a resolver es la **forma de dividir el problema en subproblemas** de la forma más eficiente. Si no es posible hacer esta división de ninguna forma viable, significa que los subproblemas no se pueden resolver de forma independiente y por lo tanto no se puede usar la estrategia de divide y vencerás.

Ejemplos: multiplicación de enteros largos, ordenación rápida, etc.

2. Algoritmos voraces

Se parte de una solución vacía y se va **seleccionando la mejor solución** para cada paso dentro de un conjunto de posibles candidatos. Se suelen usar en **problemas de optimización**.

Componentes de algoritmo voraz:

- **C**: conjunto de candidatos que todavía se pueden seleccionar.
- **S**: candidatos ya seleccionados para la solución.
- **R**: candidatos seleccionados pero rechazados después.
- **solución(R)**: función que comprueba si un conjunto de candidatos **S** es una solución al problema.
- **seleccionar(C)**: función que selecciona el mejor elemento de un conjunto **C** de candidatos.
- **factible(S, x)**: función que indica si a partir de un conjunto de candidatos **S** añadiendo otro **x** se puede llegar a una solución.
- **insertar(S, x)**: función que añade un elemento **x** al conjunto **S** de candidatos seleccionados para la solución.
- **objetivo(S)**: función que dada una solución **S** devuelve el coste asociado a la misma.

Este tipo de algoritmos suelen tener un **orden de complejidad polinomial**, lo que los hace bastante rápidos, sin embargo no aseguran encontrar la solución óptima, por lo que si se necesita ésta, es mejor usar otras estrategias.

Ejemplos: algoritmo de Dijkastra, algoritmo de Prim, etc.

3. Vuelta atrás o backtracking

Se puede entender como el **opuesto a la estrategia voraz**, ya que añade elementos a la solución parcial y los **elimina si no obtiene la solución óptima**. Por ello, se dice que realiza una **búsqueda exhaustiva y sistemática** del espacio de soluciones. Al probar todas las combinaciones posibles, **se vuelve bastante ineficiente**, pero garantiza encontrar la solución óptima. Debido a esto, es muy usado en problema de optimización.

Existen diferentes **tipos de árboles de backtracking**:

- **Árboles binarios**: elegir ciertos elementos de entre un conjunto sin importar el orden de los elementos.
- **Árboles k-narios**: varias opciones para cada x_i.
- **Árboles permutacionales**: los x_i no se pueden repetir.
- **Árboles combinatorios**: los mismos que con árboles binarios.

Componentes del backtracking:

- **s**: solución parcial hasta cierto punto.
- $s_{inicial}$: valor al que se inicializa la solución parcial al empezar el algoritmo. Suele corresponderse con un valor inválido como solución, para evitar confundirla con una solución parcial generada por la ejecución del algoritmo.
- **nivel**: indica el nivel actual del árbol en el que se encuentra estudiando los candidatos el algoritmo.
- **fin**: indica si se ha encontrado una solución.
- **Generar(nivel, s)**: función que genera la solución parcial, resultado de añadir el siguiente candidato del nivel actual.
- **Solucion(nivel, s)**: función que compueba si la solución parcial actual es una solución válida para el problema.
- **Criterio(nivel, s)**: función que comprueba si a partir de la solución parcial y desde el nivel actual es posible alcanzar una solución válida. En caso contrario, se rechazan los descendientes en el árbol, teniendo lugar una **poda de esa rama**.
- **MasHermanos(nivel, s)**: función que indica si hay más candidatos por estudiar en el nivel actual (hermanos en el árbol de soluciones).
- **Retroceder(nivel, s)**: retrocede un nivel del árbol de soluciones. Disminuye en 1 el valor de nivel, y posiblemente tendrá que actualizar la solución actual, quitando los elementos retrocedidos.
- Almacenar(s): en el caso de que se quieran **almacenar todas las soluciones**.

Este tipo de algoritmos suelen tener un **orden de ejecución factorial o exponencial**, el cual, como se dijo anteriormente, puede ser bastante ineficiente, por lo que es mejor intentar buscar otro tipo de alternativas algo más rápidas. Una mejora puede ser la de los **mecanismos de poda**, ya que en backtracking su **comportamiento** suele ser **impredecible**.

4. Ramificación y poda

Se puede interpretar como una mejora de la estrategia de backtracking, ya que sobre un esquema muy similar, se centra en mejorar las estrategias de ramificación y de poda. Concretamente se tienen los siguientes cambios:

- **Estrategia de ramificación**: el recorrido no tiene porque ser en profundidad como en el caso de backtracking.
- **Estrategia de poda**: la poda se realiza estimando para cada nodo del árbol unas cotas de beneficio que se pueden obtener a partir del mismo, es decir, si seguimos o no descendiendo por él hacia su descendientes hasta encontrar una solución.

La mejora en la poda supone tener que realizar las **estimaciones de las cotas antes de explorar** cada nodo, lo que añade algo de complejidad al problema. Sin embargo, estas cotas aseguran hacer mejores podas que en el caso de backtracking, reduciendo el espacio de soluciones y por tanto la cantidad de nodos a estudiar.

A cada **nodo hay que añadirle 3 campos**:

- **CS**: cota superior, máximo valor que podemos alcanzar a partir del nodo actual.
- **CI**: cota inferior, mínimo valor que podemos alcanzar a partir del nodo actual.
- **BE**: beneficio estimado, media entre la CS y la CI. Se usa para estimar que nodo estudiar a continuación en la estrategia de ramificación (seleccionando el de mayor o menor BE entre los posibles candidatos).

Como estructura auxiliar se usa una lista denominada **lista de nodos vivos (LNV)** en la que se van almacenando todos los nodos generados todavía no estudiados que pueden presentar una solución factible.

Para sacar un nodo de la LNV se sigue una **estrategia de ramificación** seleccionando el de mayor o menor beneficio estimado. Hay **cuatro posibles casos**:

- **LC-FIFO**: se selecciona el nodo de **menor BE** y, en caso de empate, el **primero que se introdujo**.
- **MB-FIFO**: se selecciona el nodo de **mayor BE** y, en caso de empate, el **primero que se introdujo**.
- **LC-LIFO**: se selecciona el nodo de **menor BE** y, en caso de empate, el **último que se introdujo**.
- **MB-LIFO**: se selecciona el nodo de **mayor BE** y, en caso de empate, el **último que se introdujo**.

Las **funciones** que intervienen en la estrategia de ramificación y poda son:

- **Seleccionar(LNV)**: selecciona un nodo de la LNV siguiendo la estrategia de ramificación.
- **Solucion(y)**: comprueba para un nodo **y** si es una posible solución final del problema.
- **Valor(y)**: devuelve el valor de su solución actual para un nodo **y**.

Generalmente, este tipo de algoritmos **suelen suponer mejoras sobre la técnica de backtracking**, sin embargo, en el peor caso (cuando apenas se realizan podas), se pueden generar casi tantos nodos como en el backtracking convencional, añadiéndole a mayores el coste que conlleva calcular las cotas de cada nodo y seleccionar un nodo según la estrategia de ramificación.

Por tanto, debemos escoger entre emplear más tiempo para obtener mejores cotas y así intentar podar más nodos (**estimaciones precisas de las cotas**) o ahorrar tiempo al obtener cotas menos precisas, pero podando por lo general bastantes menos nodos (**estimaciones triviales o poco precisas**).