Programación II

Tema 11. Elección del esquema algorítmico

11.1 Resumen de los esquemas algorítmicos

Cuando nos enfrentamos a un problema nuevo

- tendremos que idear un algoritmo que nos permita resolverle
- basado en alguno de los esquemas algorítmicos que conocemos
- es necesario saber identificar el o los esquemas que ofrecen mayores posibilidades de éxito para ese problema particular

La mayoría de los problemas pueden resolverse utilizando varios esquemas algorítmicos

- elegiremos la mejor opción en base a su eficiencia temporal y espacial y, en menor medida, a su facilidad de implementación
- en ocasiones no quedará más remedio que implementar las diferentes alternativas para poder compararlas

Algoritmos voraces

Aplicables a problemas que:

- **la solución se construye añadiendo en cada etapa un nuevo elemento a la solución parcial**
- **verifican el Principio de Optimalidad: "en una secuencia óptima de decisiones toda subsecuencia ha de ser también óptima"**

Es fundamental disponer de una función de selección

- **es necesario probar que con esa función de selección conducen a la solución óptima**

Un algoritmo voraz (en el caso de que exista) suele ser la alternativa más eficiente y sencilla de resolver un problema

Ejemplos:

- **dar el cambio, mochila fraccionada, MST, caminos mínimos, ...**

Algoritmos heurísticos y aproximados

Se utilizan en problemas muy complicados para los que

- **no sabemos como encontrar su solución óptima, o bien,**
- **conocemos el algoritmo que permite encontrar la solución óptima, pero requiere una cantidad excesiva de tiempo (complejidad exponencial)**

Muchos de los algoritmos heurísticos y aproximados son algoritmos voraces

Permiten obtener soluciones más o menos cercanas a la óptima

Divide y Vencerás

Aplicables a problemas que admitan una formulación recursiva

Para que resulte eficiente aplicar DyV debe verificarse que:

- **la formulación recursiva nunca resuelva el mismo subproblema más de una vez**
  - **cuando no es posible generar subproblemas independientes suele resultar más eficiente usar Programación Dinámica**
- **la descomposición en subproblemas y la combinación de las soluciones sean operaciones eficientes**
- **los subproblemas sean aproximadamente del mismo tamaño**

Ejemplos:

- **búsqueda binaria, selección, subvector de suma máxima, *mergesort*, *quicksort*, ...**

Programación dinámica

Utilizable en problemas que admitan una formulación recursiva

- **las ecuaciones recursivas se utilizan para generar una tabla**

Aplicables en el mismo tipo de problemas que los voraces:

- **la solución se construye añadiendo en cada etapa un nuevo elemento a la solución parcial**
- **verifican el Principio de Optimalidad "relajado": "la solución óptima de un problema es una combinación de soluciones óptimas de *algunos* de sus subcasos"**

Ejemplos:

- **dar el cambio, mochila entera, ...**

Vuelta Atrás y Ramificación y Poda

Realizan una exploración exhaustiva del espacio de soluciones

- **por lo que suelen ser menos eficientes que los Voraces o PD**

Aplicables a problemas en los que

- **la solución se construye añadiendo en cada etapa un nuevo elemento a la solución parcial**
- **en cada etapa se elige entre un conjunto finito de valores**

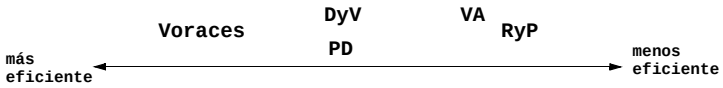
RyP es una variante de VA utilizada si existe una función "coste" que permita

- **ordenar los nodos por probabilidad de conducir a la solución**
- **detectar los nodos que no pueden conducir a la solución óptima**

Ejemplos: mochila, n reinas, laberinto, asignación, ...

Eficiencia de los distintos esquemas

Clasificación *orientativa* por eficiencia temporal:



Clasificación *orientativa* por eficiencia espacial:

- **voraces: lo normal es que no requieran memoria adicional**
- **DyV: puede ser necesario utilizar memoria adicional en la recombinación de soluciones**
- **VA: máxima "profundidad" de las llamadas recursivas, depende de la longitud de la solución**
- **PD: tabla con las soluciones parciales**
- **RyP: cola de nodos vivos, cada nodo con la información correspondiente a su estado**

11.2 Caso de estudio: fontanero con penalización

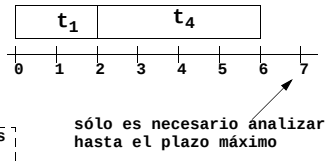
Un fontanero tiene varios trabajos pendientes, cada trabajo tiene:

- **duración: días que tarda en realizarse**
- **plazo de finalización: día en que debe estar acabado**
- **penalización: lo que el fontanero deja de ganar si no acaba el trabajo dentro del plazo**

Deberemos encontrar el orden de realización de los trabajos que minimice las penalizaciones, p.e.:

Trabajo	Dur.	Plazo	Penali.
t_1	2	3	3
t_2	3	4	2
t_3	2	5	1
t_4	4	7	5

Orden óptimo: t_1, t_4
Penalización: 3



sólo es necesario analizar hasta el plazo máximo

Se puede demostrar que la realización de los trabajos ordenados de menor a mayor plazo conduce a mejores soluciones que si no se sigue ese orden

Análisis del problema

Características:

- **se trata de un problema de optimización**
- **la solución se construye añadiendo en cada etapa un nuevo elemento a la solución**
  - **el siguiente trabajo a realizar (conjunto finito de posibilidades)**
- **verifica el Principio de Optimalidad "relajado"**
  - **la solución óptima del problema es una combinación de soluciones óptimas de algunos de sus subcasos**

Buscamos la solución óptima:

- **descartamos heurísticos y aproximados**

Con las características citadas, los esquemas más apropiados para resolver el problema podrían ser:

- **voraces, programación dinámica, vuelta atrás y ramificación y poda**

No parece apropiado utilizar DyV en este caso

- **no es posible partir el problema en partes y resolverlas de forma independiente para luego combinarlas**

Comenzaremos explorando la posibilidad de resolver el problema utilizando el esquema más sencillo y probablemente más eficiente: voraces

Solución "voraz"

Debemos encontrar una función de selección que nos permita elegir el siguiente trabajo a realizar en cada etapa

Podemos pensar varias alternativas de selección de los trabajos:

- **primero los de menor duración o primero los de menor plazo o primero los de mayor penalización**
- **primero los de mayor relación penalización/duración**

Para todos las selecciones citadas existe un contraejemplo:

$(d_1=1, pl_1=1, pe_1=3)$
$(d_2=2, pl_2=2, pe_2=4)$

menor duración o menor plazo: t_1
óptima: t_2

$(d_1=5, pl_1=5, pe_1=6, r_1=6/5)$
$(d_2=4, pl_2=8, pe_2=4, r_2=4/4)$
$(d_3=4, pl_3=8, pe_3=3, r_3=3/4)$

mayor relación o penalización: t_1
óptima: t_2, t_3

Estas funciones de selección nos servirán para implementar heurísticos, pero no para encontrar la solución óptima

Solución basada en Programación Dinámica

Supongamos que el fontanero tiene `T` trabajos pendientes ($t_i=(dur_i, pl_i, pe_i)$, $1 \le i \le T$) ordenados de menor a mayor plazo

- **el plazo máximo será pl_T y la penalización total $ptotal=\Sigma pe_i$**

Crearemos una tabla `p[0..T,0..`pl_T`]`

- **`p[t,d]` representa la penalización hasta el día `d` si sólo se pudieran realizar trabajos en el rango `1..t`**

Expresión recursiva para cada posición `p[t,d]` de la tabla:

- **si no se ha cumplido el plazo ($d \le pl_t$) elegiremos el mínimo de:**
  - **no realizar el trabajo `t`, entonces: `p[t,d]=p[t-1,d]`**
  - **realizar el trabajo `t`, entonces: `p[t,d]=p[t-1,d-`dur_t`]-`pe_t**
- **si se ha cumplido el plazo (`d>`pl_t):**
  - **se mantiene el valor del día anterior: `p[t,d]=p[t,d-1]`**

De lo anterior, junto con las condiciones de contorno, se obtiene la expresión recursiva que nos permite calcular la tabla:

$$p[t,d] = \begin{cases} ptotal & si \quad d = 0 \ \ o \ \ t = 0 \\ p[t, d-1] & si \quad d > pl_t \\ p[t-1, d] & si \quad dur_t > d \\ \min(p[t-1, d], p[t-1, d-dur_t] - pe_t) & en\ otro\ caso \end{cases}$$

Ejemplo:

Trabajo	Día					
	0	**1**	**2**	**3**	**4**	**5**
-	6	6	6	6	6	6
t_1=(2,3,2)	6	6	4	4	4	4
t_2=(3,4,3)	6	6	4	3	3	3
t_3=(2,5,1)	6	6	4	3	3	2

Para saber los trabajos a realizar (en orden opuesto a su realización) comenzamos en `t=T` y `d=`pl_T

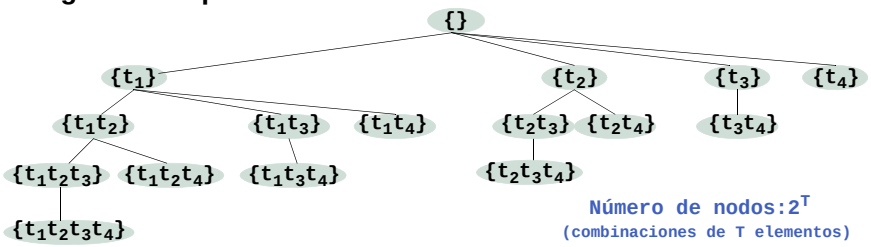
- **si `p[t,d]==p[t-1,d]` el trabajo `t` no se realiza y pasamos a la fila superior (`t=t-1`)**
- **sino, se realiza el trabajo `t` y pasamos a `t=t-1` y `d=min(d,`pl_t`)-`dur_t**
- **se continua de esta manera hasta llegar a la fila o columna `0`**

- **Eficiencia**
  - **Eficiencia del algoritmo: `max(O(T log T),O(T·`pl_T`))`**
    - **ordenación de trabajos de menor a mayor plazo (`O(T log T)`)**
    - **creación de la tabla `O(T·`pl_T`)`**
  - **Eficiencia del algoritmo para reconstruir la solución: `O(T)`**
  - **Requerimientos de memoria: `O(T·`pl_T`)` (el tamaño de la tabla)**

Solución basada en Vuelta Atrás

Suponemos que los `T` trabajos ($t_i=(dur_i,pl_i,pe_i)$**, `1`≤`i`≤`T`) están ordenados de menor a mayor plazo**

El algoritmo explorará todo el árbol de soluciones:



- **pero sin repetir ninguna combinación (p.e. explora `{`t_1t_2`}` pero no `{`t_2t_1`}`)**
  - **ambas tienen la misma duración, pero es más fácil que sea posible `{`t_1t_2`}` que lo sea `{`t_2t_1`}` ya que** $pl_1 \leq pl_2$

La solución se basa en el método recursivo `fontaneroVARec` que:

- **recibe como parámetro el día actual (d) y el último trabajo realizado (`ut`)**
- **retorna una lista con los trabajos a realizar**
- **la primera llamada se debe realizar con `d=0` y `ut=-1`**

En cada invocación se consideran los trabajos t_i **(`ut`<`i`≤`T`)**

- **si es posible realizar** t_i **antes de que se cumpla su plazo, es decir, si** $d+dur_i \leq pl_i$**:**
  - **se realiza una invocación recursiva con `ut=i` y `d=d+`**dur_i
  - **se añade** t_i **a la lista retornada por la invocación recursiva**
  - **si la lista mejora la mejor solución encontrada hasta el momento pasa a convertirse en la mejor solución**
- **después de haber considerado todos los trabajos** t_i **(`ut`<`i`≤`T`) el método retorna la mejor solución encontrada**

```
SolParcial fontaneroVARec(int ut,int d,Trabajo[] t){
  SolParcial sol = new SolParcial();
  SolParcial máximo = new SolParcial();
  for(int i=ut+1; i<t.length; i++){
    if (t[i].dur()+d <= t[i].pl()) {
      // el trabajo se puede hacer antes del plazo
      sol = fontaneroVARec(i, d+t[i].dur(), t);
      sol.lista.add(t[i]);
      sol.penaAhorrada+=t[i].pe();
      if (sol.penaAhorrada>máximo.penaAhorrada) {
        // mejora la mejor solución hasta ahora
        máximo.lista=new LinkedList<Trabajo>(sol.lista);
        máximo.penaAhorrada=sol.penaAhorrada;
      }
    }
  }
  return máximo;
}
```


Solución basada en Ramificación y Poda

Igual que en Vuelta Atrás, en RyP se realiza un recorrido exhaustivo del espacio de soluciones

- **pero RyP utiliza una cola de nodos vivos**
- **en la que los nodos se encuentran ordenados por probabilidad de conducir a la solución óptima**

Deberemos encontrar la función de coste para:

- **ordenar los nodos en la cola de nodos vivos**
- **podar aquellos nodos desde los que no se pueda mejorar la mejor solución encontrada hasta el momento**

Una buena función de coste deberá:

- **ser rápida de calcular**
- **proporcionar una buena estimación (optimista) de la solución que se puede obtener desde un nodo**

Una solución parcial (o nodo) contendrá:

- **`lista`: trabajos elegidos para ser realizados**
- **`ult`: índice del último trabajo incluido**
- **`d`: duración de los trabajos incluidos en la lista**

Estimaremos la mejor solución que se puede alcanzar desde un nodo (su coste) como la suma de:

- **la penalización ahorrada por los trabajos incluidos en `lista`**
- **la penalización de todos los trabajos aún no considerados para ser incluidos en `lista`**
  - **es decir, suponemos que todos los demás trabajos se van a poder realizar: suposición optimista**

Se puede pensar en funciones de coste más precisas

- **pero habría que valorar si su complejidad de cálculo compensa el posible incremento en número de nodos podados**

```
LinkedList<Trabajo> fontaneroRyP(Trabajo[] t){

  SolParcial eNodo, nHijo;
  PriorityQueue<SolParcial> colaVivos=
                    new PriorityQueue<SolParcial>();

  // crea la mejor solución inicial
  //(muy mala: penaAhorrada=0)
  SolParcial máximo=new SolParcial();

  // crea el primer e-nodo (lista de trabajos vacía)
  eNodo=new SolParcial();
```



```
  // mientras queden nodos vivos
  do {
    // ramifica el eNodo
    for(int i=eNodo.i+1; i<t.length; i++) {
      if (t[i].dur()+eNodo.d <= t[i].pl()) {
        // se puede crear un nodo hijo válido
        nHijo=new SolParcial(eNodo,t[i],i);
        if (nHijo.penaAhorrada>máximo.penaAhorrada){
          máximo=nHijo.copia(); // mejor solución
        }
        if (eNodo.coste>máximo.penaAhorrada){ // no poda
          colaVivos.add(nHijo);
        }
      } // fin if hijo válido
    } // fin for ramificación
    eNodo=colaVivos.remove(); // siguiente eNodo
  } while(eNodo!=null && eNodo.coste>máximo.penaAhorrada);
  return máximo.lista;
}
```



```
// clase Solución parcial
class SolParcial implements Comparable{
  // lista de trabajos realizados
  private LinkedList<Trabajo> lista =
                          new LinkedList<Trabajo>();

  // penalización ahorrada por los trabajos
  // incluidos en la lista
  private int penaAhorrada;
  // índice en el array de trabajos del último
  // trabajo incluido en la lista
  private int ult;
  // día de finalización del último trabajo
  // contenido en la lista
  private int d;
  // penalización que, como máximo, se podría
  // llegar a ahorrar desde este nodo
  private int coste;
```

```
/**
   * crea una solución parcial tomando otra como
   * base (orig) a la que añade un nuevo trabajo (t)
   * @param orig solución parcial base
   * @param t trabajo a incluir en la lista
   * @param i índice en el array de trabajos del
   *          trabajo a incluir
   */
   public SolParcial(SolParcial orig, Trabajo t, int i) {
      lista = new LinkedList<Trabajo>(orig.lista);
      lista.add(t);
      penaAhorrada=orig.penaAhorrada+t.pe();
      this.ult=i;
      d=orig.d+t.dur();
      coste=penaAhorrada+calculaCoste(i+1,d);
   }

   ... // resto de métodos de la clase SolParcial
}
```


Eficiencia de los algoritmos VA y RyP

Si el fontanero tiene T trabajos pendientes el número de nodos a evaluar en el peor caso es 2^T (tanto para VA como para RyP)

- **luego la eficiencia en el peor caso será $O(2^T)$**
- **en el caso promedio se evalúan muchos menos nodos (especialmente en el caso de RyP)**

Eficiencia espacial:

- **VA: máxima profundidad de las llamadas recursivas (máxima longitud de la solución)**
  - **eficiencia: $O(T)$**
- **RyP: máxima longitud de la cola de nodos vivos**
  - **habría que determinarla realizando medidas en distintos casos**

11.3 Bibliografía

[1] **Brassard G., Bratley P., *Fundamentos de algoritmia*. Prentice Hall, 1997.**

[2] **Aho A.V., Hopcroft J.E., Ullman J.D., *Estructuras de datos y algoritmos*. Addison-Wesley, 1988.**

[3] **Sartaj Sahni, *Data Structures, Algoriths, and Applications in Java*. Mc Graw Hill, 2000**

[4] **Entrada en la *Wikipedia* para “algoritmo”**