Tabla de Contenidos

Introducción

- Bautizado por D. J. Lehmer (1950)
- Formalizado por Walker, Golom y Baumert (1960)
- Método general aplicable a numerosos problemas
  - Localización
  - Optimización
- Busca soluciones de tipo n-tupla (x1, …, Xn)
- Realiza un estudio exhaustivo del conjunto de "posibles soluciones" que deben ser conocidas a priori

Estudio de las soluciones

- Exploración metódica y ordenada
- División/descomposición en etapas
- Representación en forma de árbol
  - Cada nodo es un fragmento de la solución formado por las k etapas previas
  - Los sucesores de un nodo son las prolongaciones de la solución
  - Los recorridos desde la raíz del árbol a las hojas constituyen las posibles soluciones
- Puede existir el árbol de forma explícita o construirse durante la resolución del problema

Representación en forma de árbol

- Organización del espacio de soluciones en estructura de árbol
  - Fijar descomposición en etapas que se va a sergir
  - Analizar las opciones disponibles en cada etapa

- Recorrido del árbol
  - **Nodos Solución**
    - → Nodos en los que se ha alcanzado la solución
  - **Nodos Problema**
    - → Fragmentos de solución
    - → Requiere explorar sus descendientes
  - **Nodos Fracaso**
    - → Sabemos que ninguno de sus descendientes puede ser nodo solución
    - → Requieren retroceso a su antecesor

Consideraciones

- Acotación de los nodos solución
- Aplicación de test de solución antes de llegar a explorar un nodo fracaso
- Los tests también consumen un tiempo importante
- Regla general
  - Test sencillos cuando los árboles no sean excesivamente grandes
  - Test sofisticados en árboles muy grandes para reducir el espacio de búsqueda

Tipos de problemas a resolver

- **Combinaciones**
  - La soluciones (si existen) son subconjuntos o permutaciones de un conjunto dado
- **Permutaciones**
  - En etapa *i* se decide el *i*-ésimo elemento de la solución
  - Los nodos solución son las hojas situadas a profundidad *n*
- **Subconjuntos con varios enfoques**
  - Estudio de la solución
    - → Similar a los de tipo permutación
    - → Cualquier nodo puede ser solución (aun sin ser hoja)
  - Estudio de elementos
    - → En cada etapa se fija un elemento y se estudia si interesa o no su inclusión en la solución
    - → Se generan árboles binarios
    - → Los nodos solución son las hojas situadas a profundidad *n*

Recursividad e Iteración

- Recorrido de un árbol
  - Proceso recursivo por naturaleza (sencillez)
- Parámetros representativos
  - Contador de **etapas** (profundidad del árbol)
  - Descripción del nodo en el que estamos (**estado**) y la trayectoria que hemos seguido
- Procedimiento de diseño organizado en tres fases

Procedimiento de diseño

- Fase 1
  - **Test de solución**
    - → Si el nodo en el que estamos es posible solución
    - → Hay que efectuar la comprobación
- Fase 2
  - **Test de fracaso**
    - → Si disponemos de un test para comprobar si el nodo en el que estamos es un nodo fracaso, lo aplicamos
- Fase 1
  - Generación de **descendientes** si no es nodo fracaso
    - → Dependerá del estado y de la etapa
    - → Llamaremos recursivamente al procedimiento para tratar la etapa siguiente (*etapa+1*)

Algoritmo recursivo


```
solucion_t Backtracking (int etapa, estado_t T)  {
    // T[1,…,n]
    solucion_t solucion, x;
    if ( esSolucion(T,etapa) ) solucion = T;
    else {
        if ( esFracaso(T,etapa) || (etapa == n) ) solucion = fracaso;
        else {
                s = genSucesores(etapa, T);
                for (k=1; k<=longitud(s); k++) {
                        T[etapa+1] = k;
                        x = Backtracking (etapa+1, T);
                        if ( esSolucion(x,etapa) solucion = x;
                }
                solucion = fracaso;
        }
    }
    return solucion;
}
```

Algoritmo iterativo


```
solucion_t Backtracking (int etapa)  {
    // T[1,…,n]
    k = 1; T = ∅; solucion_t solucion = fracaso;
    while ( k>1 ) {
        if ( esSolucion(T,k) ) solucion = T;
        else {
            if ( esFracaso(T,etapa) || (k>n) ) {
                k = k – 1; //vuelta atrás
                T[k] = siguienteSucesor(T,k);
            } else {
                k = k + 1;
                T[k] = siguienteSucesor(T,k);
            }
        }
    }
    return solucion;
}
```

Complejidad en Backtracking

Factores a considerar

- Número de elementos por etapa $\Rightarrow$ (k)
- Tiempo para generar sucesores $\Rightarrow$ O(1)
- Tiempo para aplicar los tests $\Rightarrow$ O(1)

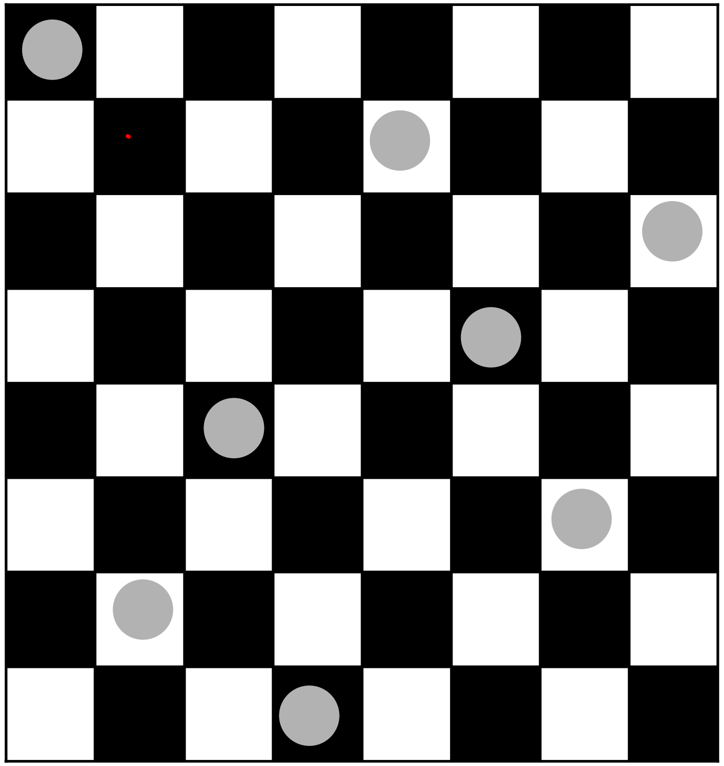
Complejidad

$$T(n) = \begin{cases} O(1) & n = 1 \\ O(1) + k \cdot T(n-1) & n \geq 1 \end{cases} \Rightarrow O(k^n)$$

Problema

- Colocación de 8 reinas en un tablero de ajedrez sin que se amenacen

Ejemplo

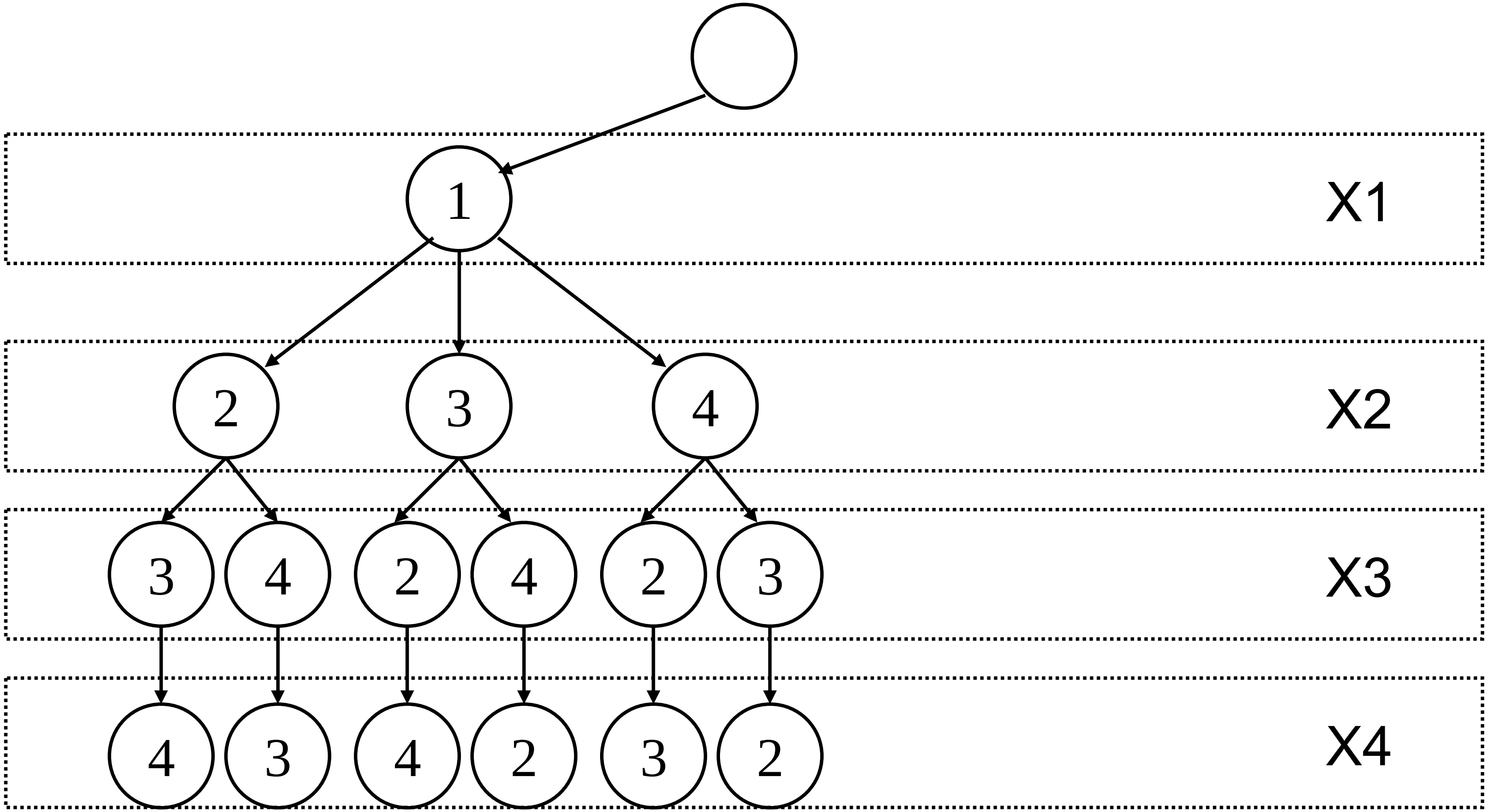


Estrategia

- Solución en forma de tupla $(X_1, X_2, X_3, \ldots X_8)$
- La reina **i** estará siempre en la fila **i**
- $\mathbf{X_i} \equiv$ columna en la que está colocada la reina **i**
- Propiedades
  - $\mathbf{X_i} \in \{1, \ldots, 8\}$
  - No puede haber 2 reinas en la misma columna
  - $\mathbf{X_i} \neq \mathbf{X_j} \quad \forall i,j \in \{1,\ldots 8\}$
  - No pueden coincidir en las diagonales
- Combinaciones o espacio de búsqueda
  - Permutaciones de $\{1,\ldots 8\}$ $\Rightarrow$ $\boxed{8^8 \longrightarrow 8!}$

Acotación de nodos solución

Árbol de soluciones para 4 reinas



Algoritmo

- Comprobación de diagonales
  - Dos casillas (i,j) e (k,l) están en la misma diagonal si cumplen: i+j=k+l ó k-l=i-j ó |i-k|=|j-l|
- Intento de colocación de una reina y comprobación de nodos fracaso

```
int coloca(k) {
    for (i=1; i<=k; i++)
        if ( x(i)==x(k) || abs(x(i)-x(k))==abs(i-k)
            return 0;
    return 1;
}
```

- Implementación

Suma de subconjuntos

Problema

- Dado un conjunto ***C*** de valores N_i y un parámetro ***M*** tenemos que determinar los subconjuntos de elementos que suman ***M***
- Representación de la solución mediante tuplas
  - Utilizamos los índices de los elementos o tomando 1's y 0's para indicar la pertenencia de cada elemento al subconjunto
- Restricciones explícitas
  - $x_i \in \{j : 1 \leq j \leq n\}$
- Restricciones implícitas

$$\sum x_i = M \qquad x_i \neq x_j, \forall i, j \qquad x_i < x_{i+1}$$

Para no proporcionar soluciones repetidas con sus elementos en distinto orden

Ejemplo

- Si el conjunto de valores es (11,13,24,7) y M = 31
- Los subconjuntos serían
  - (11,13,7) *(1,2,4) ó (1,1,0,1)*
  - (24,7) *(3,4) ó (0,0,1,1)*

Representaciones de la solución

Consideraciones para el algoritmo

- Elementos del conjunto C
  - (W1, W2, ….,Wn)
- Elementos de la solución X
  - Xi$\in${0,1}
- Restricciones en *k*
  - $$\sum_{i=1}^{k-1} w(i)\cdot x(i) + w(k) \leq M$$

    No se sobrepasa M
- Comprobaciones en *k*
  - $$\sum_{i=1}^{k-1} w(i)\cdot x(i) + \sum_{i=k}^{n} w(i) \geq M$$

    La solución tiene futuro
  - $$\sum_{i=1}^{k} w(i)\cdot x(i) = M$$

    Se ha alcanzado una solución

Algoritmo genérico

```
SumaSub(conjunto_t C, sol_t x, int k) {
        for (x[k]=0; x[k]<=1; x[k]++) {
                if ( restricciones(C,x,k) ) {
                        if ( esSolucion(C,x,k) )
                                mostrar(x);
                } else
                        SumaSub(C,x,k+1);
        }
}
```

- *No se ha considerado si la solución puede tener futuro. Este aspecto mejoraría la eficiencia que así es de* 2^n

Problema

- ***N*** vértices
- ***M*** *colores*
- Colorear el grafo con ***M*** ó menos colores
- Dos nodos adyacentes nunca deben tener el mismo color
- Se puede considerar como un problema de optimización si se pretende buscar el mínimo valor de ***M***

Aspectos a considerar

- Forma de la tupla solución
  - (X1, …, Xn) y Xi $\in$ {1, …, m}
- Restricciones
  - Sea $\boldsymbol{L_{ij}}$ la matriz de adyacencia
  - Dado X_k, $X_k \neq X_j$, $\forall j$ / L(j,k)
  - Función de restricción en la etapa *k*

```
for (j=1; j<n; j++)
        if( (L(j,k)==1) && (x[j]!=x[k]) )
                return 0;
return 1;
```

Algoritmo

```
coloreado(grafo_t L, sol_t x, int k) {
        for( x[k]=1; x[k]<=m; x[k]++)
                if ( restricciones(L,x,k) ) {
                        if( k==n ) mostrar(x);
                        else coloreado(L,x,k+1);
                }
}
```

Optimización

- Generar todas las soluciones
- Buscar la que emplea menos colores

Mochila (versión 0/1)

Planteamiento del problema

- ***n*** objetos y una mochila de capacidad ***M***
- El objeto ***i*** para $\boldsymbol{w_i}$ y proporciona un peso $\boldsymbol{p_i}$
  - Objetivo $Max \sum_{1 \le i \le n} b_i x_i$ $\left\{ \begin{array}{c} x_i \equiv \text{fracción del objeto } i \\ 0 \le x_i \le 1 \end{array} \right\}$
  - Restricción $\sum_{1 \le i \le n} p_i x_i \le M$
- La complejidad de la versión 0/1 es muy elevada con una estrategia voraz

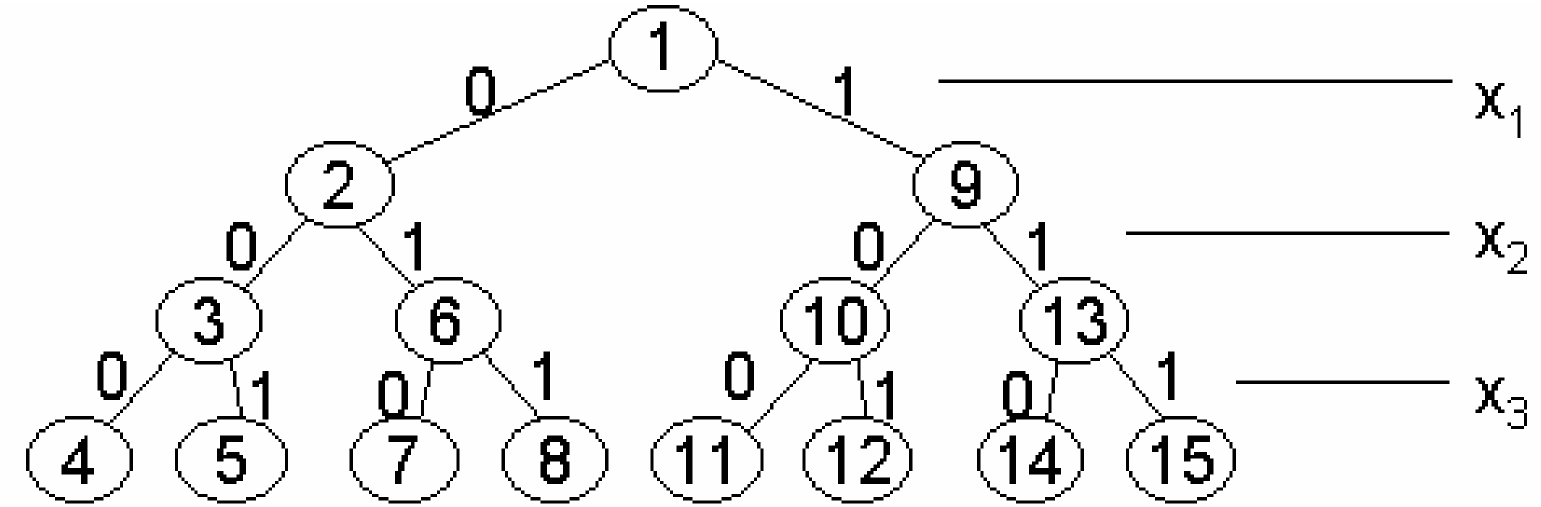
Mochila (versión 0/1)

Aproximación backtracking

- Explosión combinatoria: se generan todas las combinaciones posibles
  - Al menos habrá una solución (no incluir ningún objeto)
- Problema del tipo “*subconjunto*”
- Enfoque de “*estudio de la solución*”
- En cada paso se decidirá qué elemento de la solución se va a incluir
- En la etapa *k* habrá (*n*-*k*) elementos para incluir

Diseño de la solución con backtracking

- $(x_1, \ldots, x_n)$ con $x_i \in \{0,1\}$
- En cada nivel *k* (etapa *k*) probaremos incluir o no el objeto *k*
- En la etapa *k* habrá (*n*-*k*) elementos para incluir



Mochila (versión 0/1)

Técnicas para mejorar la eficiencia

- Definición de *test* para nodos *fracaso* y nodos *problema*
- Considerar como beneficio máximo el de las soluciones ya calculadas (el de la mejor)
- ***Estimar*** el beneficio máximo posible en un nodo de la solución que estamos calculando
- Si el valor estimado es menor que el calculado consideraremos que estamos en un nodo *fracaso*

Mochila (versión 0/1)

Algoritmo genérico

```
void mochila (int M, conjunto_t objs, solucion_t x) {
        b = -1;   p =0; v=0;          /* beneficio, peso y valor inicial */
        solucion_t y;
        k = 1;                        /* Primera etapa  */
        do {
                while ( (k<=tamanyo(n)) && (p<=M) ) do {
                        /* introducimos el objeto k */
                        v += objs[k].valor;
                        p += objs[k].peso;
                        y[k] = 1;   k ++;
                        if (k>n) { /* nodo terminado de estudiar*/
                                b = v; x = y;
                        } else { /* sacar el último objeto */
                        }
                }
        } while (1);
```