Universidad Politécnica de Madrid
ETSI SISTEMAS INFORMÁTICOS

Algorítmica y Complejidad

Tema 6 – Vuelta Atrás (*Backtracking*).

1. Introducción.

2. El viaje del caballo de ajedrez.

3. Las 8 reinas.

4. Selección óptima.

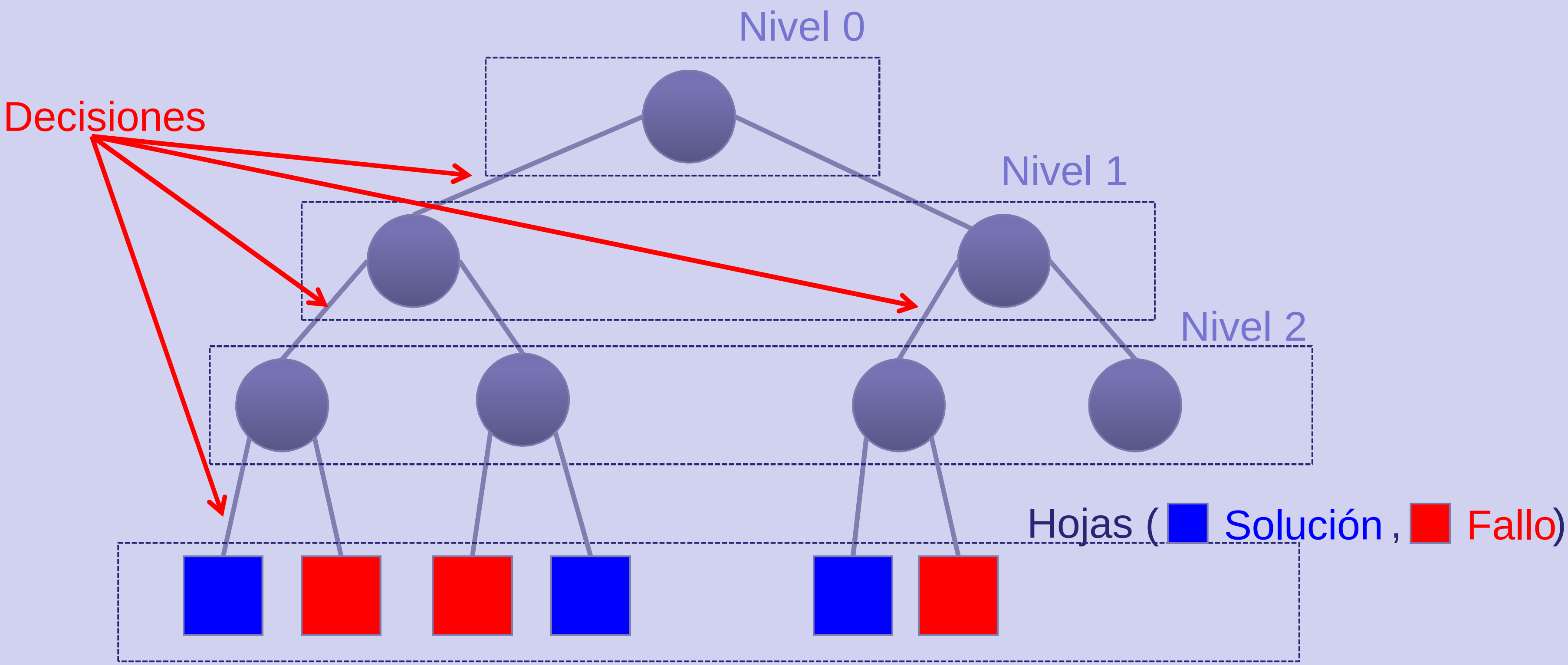
1. Introducción.
2. El viaje del caballo de ajedrez
3. Las 8 reinas
4. Selección óptima

Introducción

- Supongamos un problema en el que tenemos que tomar una serie de decisiones entre varias opciones
  - No disponemos de la información suficiente para elegir una de las opciones
  - Cada decisión que tomamos genera nuevas opciones
  - En algún momento una o varias secuencias de decisiones forman una solución al problema
- La Vuelta Atrás consiste en explorar un conjunto de decisiones hasta que una de ellas es “solución”
  - Solución analítica difícil de encontrar
  - Popularidad gracias al ordenador
- Ejemplos típicos:
  - Juegos
  - Inteligencia artificial
  - Reconocimiento de lenguajes regulares
  - Lenguajes de programación (Prolog)

Introducción

- Los algoritmos de Vuelta Atrás generan un árbol con un recorrido en profundidad (Espacio de búsqueda)

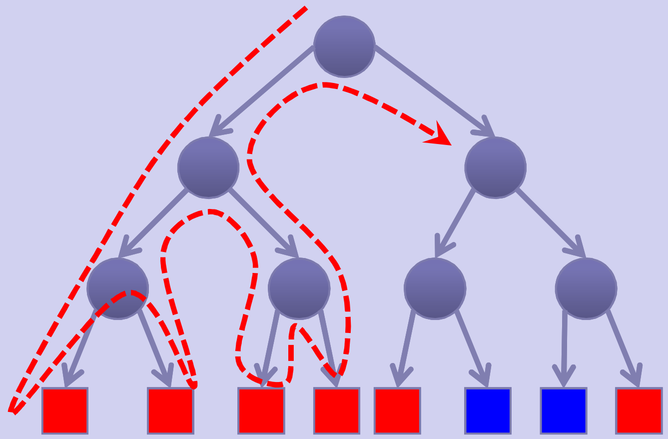


  - Cada nivel representa una parte de la solución o soluciones
  - Cada nodo intermedio es una solución parcial
  - La solución (decisiones) es el recorrido desde la raíz hasta la solución

Algoritmo básico de Vuelta Atrás

Encuentra todas las soluciones

```
procedure VueltaAtras (N: Estado; exito : out boolean) is
begin
   if esHoja (N) then
      exito := esSolución(N);
   else
      while Mas_Decisiones (N) loop
         VueltaAtras (NuevaDecisión (N), exito);
      end loop;
   end if;
end VueltaAtras;
```



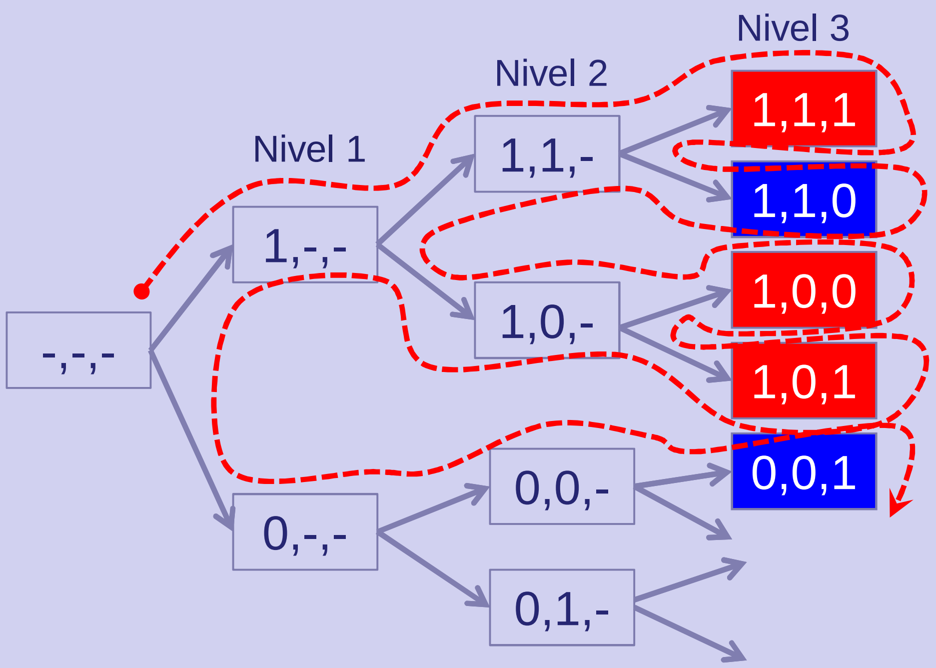
Recorrido en profundidad

- **esSolución** comprueba que un nodo hoja es una solución al problema
- El algunos problemas es necesario guardar estado a medida que se recorre el árbol y también deshacerlo en caso de no encontrar una solución

Algoritmo SumaSubconjunto

Dado un conjunto de números, seleccionar todos los subconjuntos que sumen un valor determinado.

Ejemplo: conjunto = {2, 5, 7} valor = 7



Existen dos soluciones:

solución = {1, 1, 0}

solución = {0, 0, 1}

(1 y 0 indican la selección o no del correspondiente elemento del conjunto)

Algoritmo SumaSubconjunto

Estructuras de datos:

```
type vector is array (1..3) of natural;

conjunto : vector  := (2, 5, 7);
solucion : vector  := (others => 0);

valor    : natural := 7;

indice   : natural := conjunto'first;
```

Algoritmo SumaSubconjunto

```
procedure SumaSubconjunto (conjunto      : vector;
                           solucion      : in out vector;
                           valor         : natural;
                           indice        : natural) is
begin
   if indice > conjunto'last then
      if EsSolucion (conjunto, solucion,valor) then
         MostrarSolucion (solucion);
      end if;
   else
      solucion (indice) := 0;
      SumaSubconjunto (conjunto, solucion, valor, indice + 1);
      solucion (indice) := 1;
      SumaSubconjunto (conjunto, solucion, valor, indice + 1);
   end if;
end SumaSubconjunto;
```

Algoritmo SumaSubconjunto

```
function EsSolucion (conjunto: vector;
                     solucion : vector;
                     valor    : natural) return boolean is
   suma : natural := 0;
begin
   for k in conjunto'range loop
      suma := suma + solucion (k) * conjunto (k);
   end loop;
   return suma = valor;
end EsSolucion;
```

Algoritmo SumaSubconjunto (con poda)

¿Se puede mejorar el algoritmo? A veces no es necesario explorar todas las posibilidades. Por ejemplo:

conjunto = {8, 5, 7}   valor = 7

No es necesario explorar las soluciones del tipo:

solución = { 1 , – , – }

```
function SolucionPosible return Boolean is
    suma _parcial : Natural := 0;
begin
   for k in conjunto'First .. indice loop
        suma_parcial := suma_parcial + solucion (k) * conjunto (k);
   end loop;
   return suma_parcial <= objetivo;
end SoluciónPosible;
```

Algoritmo SumaSubconjunto (con poda)

```
procedure SumaSubconjunto  (conjunto : vector;
                            solucion : in out vector;
                            valor    : natural;
                            indice   : natural) is
begin
   . . . . . . . . . . . . . . . . . . . .
   . . . . . . . . . . . . . . . . . . . .
      If  SolucionPosible  then
         SumaSubconjunto (conjunto, solucion, valor, indice + 1);
      end if;
      . . . . . . . . . . . . . . . . . . . .
      If  Solucion posible  then
         SumaSubconjunto (conjunto, solucion, valor, indice + 1);
      end if;
 . . . . . . . . . . . . . . . . . . . .
end SumaSubconjunto;
```

Algoritmo SumaSubconjunto (con poda adelantada)

- Para que una rama contenga una solución, debe cumplir:

$$\underbrace{\sum_{k=1}^{\text{indice}} \text{conjunto}(k) \times \text{solucion}(k)}_{\text{Ya exploradas}} + \underbrace{\sum_{k=\text{indice}+1}^{n} \text{solucion}(k)}_{\text{Quedan por explorar}} \geq \text{valor}$$

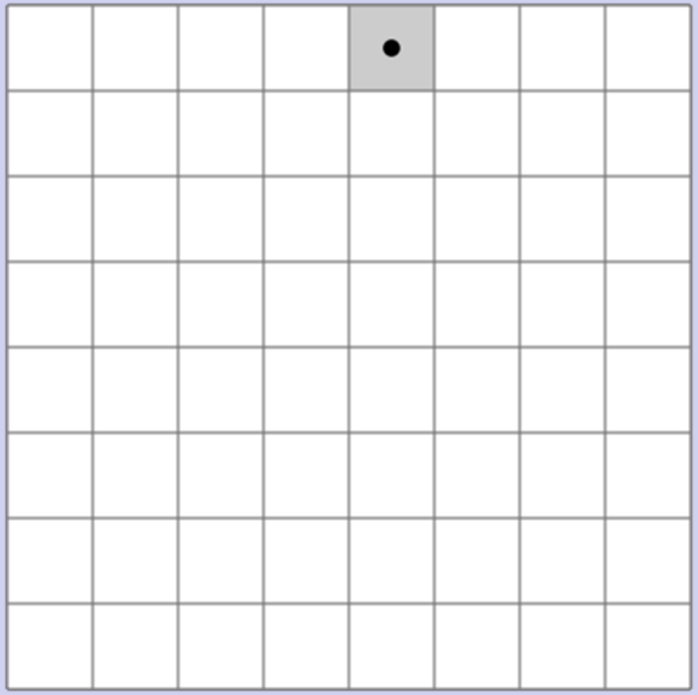
- Si el conjunto está ordenado de menor a mayor

$$\sum_{k=1}^{\text{indice}} \text{conjunto}(k) \times \text{solucion}(k) + \text{conjunto}(\text{indice}+1) \leq \text{valor}$$

1. Introducción
2. El viaje del caballo de ajedrez.
3. Las 8 reinas
4. Selección óptima

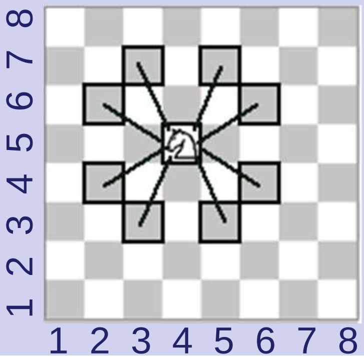
Descripción

- Dado un tablero de ajedrez de tamaño $n \times n$ cuadros
- Dada una posición inicial del caballo en la cuadricula (x_0, y_0)
- Encontrar, si existe, un recorrido de n^2-1 movimientos en el que cada cuadro se visite una única vez



Decisiones de implementación

- El tablero se representará mediante una matriz (T)
- Cada cuadro almacenará en que posición del viaje ha sido visitado
- Movimientos del caballo (M)



type Tablero is array (1..N, 1..N) of Natural;

Movimientos:

- Dada una posición inicial (u, v)
- Las posibles nuevas posiciones se obtienen:

  $x = u + \text{Movimientos}\ [i].X$

  $y = v + \text{Movimientos}\ [i].Y$

  - Si x ó y no están en el intervalo $[1, n]$, el movimiento no es válido
  - Además es necesario que el cuadro (x, y) este a 0 $(Ta[x, y] = 0)$
  - No están disponibles los 8 movimientos posibles en todos los cuadros del tablero

Posibles Movimientos	
X	Y
2	1
1	2
-1	2
-2	1
-2	-1
-1	-2
1	-2
2	-1

```
type Movimientos is array (1..8) of Record
                                x: Integer; y: Integer;
                              End Record;
```

Algoritmo

- **T** : Tablero
- **M**: Movimientos

```
procedure Ensayar ( i       : Natural;
                    x, y    : Natural;
                    exito : Boolean) is
      k: Natural;
     Q: Boolean;
      u, v: Natural;   -- Nuevo movimiento
begin
   k := 0;
  loop
    k  := k + 1;
    Q := false;
    NuevoMov (u, v, k);
```

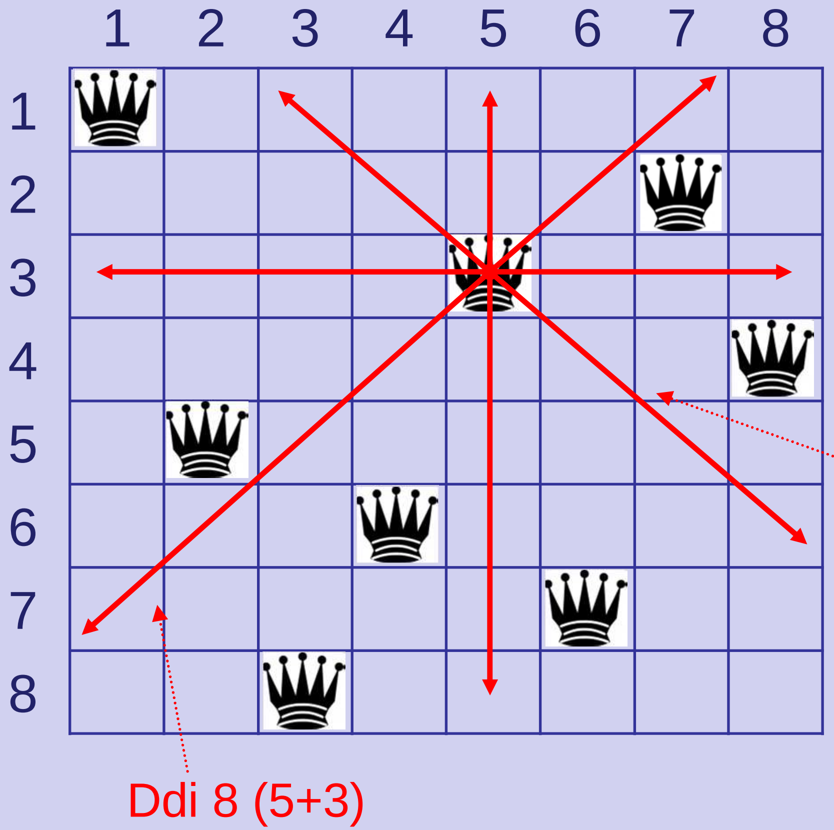
```
    if esValido (u, v) and T(u, v) = 0 then
        T(u, v) := i;
        if i < n**n then
              Ensayar (i+1, u, v, Q);
              if not Q then
                  T (u, v) := 0;   --- Vuelta Atrás
              end if;
        else
              Q := true;   --- Completado
        end if;
    end if;
    exit when Q or (k = 8);
  end loop;
  exito := Q;
end Ensayar;
```

Algoritmo:

```
procedure ViajeDelCaballo is
     T       : Tablero;
      x, y : Natural ;
      exito: Boolean;
begin
     T := (others => 0);
      x := 4;  y := 4;   --- Posición inicial
      T(x, y) := 1;
      Ensayar (2, x, y, exito);
      if exito then
         ImprimirTablero (T);
      else
         Put_Line (“No se ha encontrado la solución”);
      end if;
end ViajeDelCaballo;
```

1. Introducción
2. El viaje del caballo de ajedrez
3. Las 8 reinas.
4. Selección óptima

- Colocar 8 reinas en un tablero de ajedrez, de forma que ninguna de ellas pueda acabar con alguna de las otras
- Algoritmo típico de prueba y error
- Analizado por Gauss en 1850 (no lo resolvió completamente)
- Admite varias soluciones distintas
  - aunque sólo 12 son distintas
- El esquema del algoritmo es similar a los anteriores
  - Se colocan las reinas una tras otra hasta colocarlas todas
  - En cada llamada recursiva se intenta colocar una reina de forma segura
  - El algoritmo puede encontrar una solución o todas



Propiedades

- Dada una coordenada (x, y), sobre la diagonal izquierda-derecha se cumple que x – y es constante
- Cualquier coordenada (x, y), sobre la diagonal derecha-izquierda cumple que x + y es idéntica para cualquier cuadro

Ejemplo coordenada (5,3)

- Reina [5] = 3
- Fila [3] ocupada
- Did [2] ocupada
- Ddi [8] ocupada

Representación (en un tablero de 8x8):

- Reina [i] posición de la reina en la columna i (1..8)
- Fila [j] ¿ocupada la fila j (1..8)?
- Did [k] ¿ocupada diagonal k (-7..7) izquierda-derecha?
- Ddi [k] ¿ocupada diagonal k (2..16) derecha-izquierda?

Operaciones:

– ***ponerreina (j, i)*** (Fila j, Columna i)

Reina [i] ← j

Fila [j] ← false

Did [i-j] ← false

Ddi [i+j] ← false

– ***quitarreina (j, i)***

Fila [j] ← true

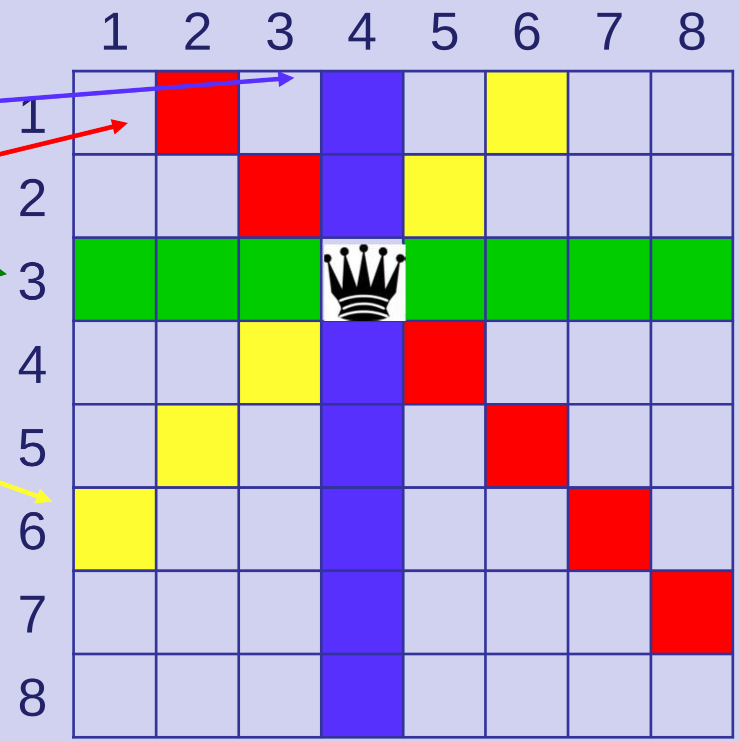
Did [i-j] ← true

Ddi [i+j] ← true

– ***posiciónsegura (j, i)***

Si se cumple que:

Fila [j] && Did [i-j] && Ddi [i+j]



Algoritmo para búsqueda de una única solución

```
typedef T_Reina is array (1..8) of Natural;
typedef T_Fila is array (1..8) of Boolean;
typedef T_Did is array (-7..7) of Boolean;
typedef T_Ddi is array (2..16) of Boolean;

procedure OchoReinas is
    Reina: T_Reina := (others => 0);
    Fila    : T_Fila := (others => false);
    Did     : T_Did := (others => false);
    Ddi     : T_Ddi := (others => false);
    exito  : Boolean := false;
begin
    Ensayar (1, exito);
    if exito then
       ImprimirSolucion;
    else
        Put_Line (“No hay solución”);
    end if;
end OchoReinas;
```

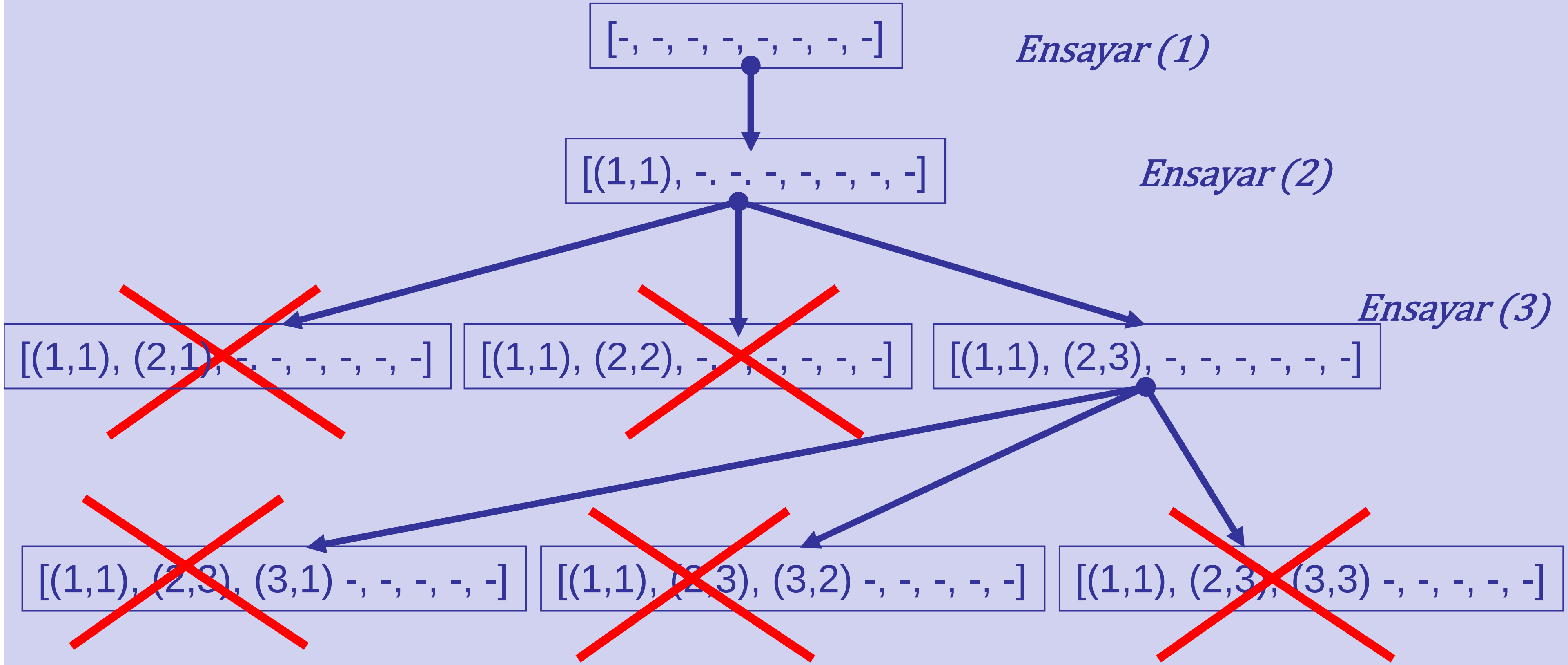
Algoritmo para búsqueda de una única solución

```
procedure ponerReina (j: Natural; i: Natural) is
begin
   Reina (i) := j;
   Fila (j)    := false;
   Did (i-j)  := false;
   Ddi (i+j) := false;
end ponerReina;

procedure quitarReina (j: Natural; i: Natural) is
begin
   Fila (j)    := true;
   Did (i-j)  := true;
   Ddi (i+j) := true;
end quitarReina;

function posicionSegura (j: Natural; i: Natural) is
begin
   return Fila (j) and Did (i-j) and Ddi (i+j);
end posicionSegura;
```

Algoritmo para búsqueda de una única solución

```
procedure Ensayar (i: Natural;  exito: out Boolean) is
      j : Natural := 0;
begin
    loop
        j := j + 1;   exito := false;
        if posicionSegura (j, i) then
            ponerReina (j, i);
            if  i < 8 then
                 Ensayar (i+1, exito);
                 if not exito then
                      quitarReina (j, i);
                 end if;
            else
                  exito := true;
            end if;
        end if;
        exit when exito or j = 8;
     end loop;
end Ensayar;
```

Árbol de búsqueda de la solución



El siguiente movimiento es: [(1,1), (2,3), (3,5), -, -, -, -, -]

Algoritmo de búsqueda de todas las soluciones

```
procedure Ensayar (i: Natural) is
begin
   for j in 1..8 loop
      if posicionSegura (j, i) then
         ponerReina (j, i);
         if i < 8 then
            Ensayar (i+1);
         else
            ImprimirSolucion (Reina);
         end if;
         quitarReina (j, i);
      end if;
   end loop;
end Ensayar;
```

Diferencias con la versión anterior del algoritmo:

- Bucle for para contemplar todas las posibles soluciones
- Eliminado el parámetro **éxito** para la búsqueda de una única solución
- Una vez colocadas las 8 reinas se imprime la solución
- Se van quitando las reinas en orden inverso a su colocación, para continuar con la búsqueda de soluciones

- 92 soluciones
- Solo 12 significativamente distintas

1. Introducción
2. El viaje del caballo de ajedrez
3. Las 8 reinas
4. Selección óptima.

- Buscar entre todas las posibles soluciones de un problema, la óptima
- Las posibles soluciones (candidatas) se van generando con un esquema de vuelta-atrás
- En los algoritmos anteriores habíamos visto la búsqueda de una solución y la obtención de todas las soluciones, ahora buscamos la solución óptima
- Para que sean solución las diferentes candidatas tienen que cumplir una o varias restricciones del problema, de entre ellas obtendremos la óptima
- Supongamos un conjunto de objetos O con n elementos, siendo la solución óptima $S \subseteq \mathrm{O}$
- El número de soluciones candidatas es 2^{n}
- Las restricciones del problema reducirán el número de soluciones candidatas
- De entre las soluciones candidatas tendremos que establecer un criterio para obtener la solución óptima

Ejemplo (Mochila)

- Dados un conjuntos de objetos $\mathbf{o_1}, \mathbf{o_2}, \ldots, \mathbf{o_n}$, caracterizados por un peso $\mathbf{p}$ y un valor $\mathbf{v}$.
- El conjunto óptimo es aquél en el que la suma de los valores de sus componentes es máxima
- Existiendo una restricción en la suma de sus pesos

O	1	2	3	4	5	6	7	8	9	10
P	10	11	12	13	14	15	16	17	18	19
V	18	20	17	19	25	21	27	23	25	24
10										
30										
40										
50										

Limite de peso

Solución

Esquema del algoritmo

- **i ∈ O**

```
Ensayar (i):
 if inclusión de i aceptable then
    incluir i
    if i < n then  // Probar con todos
        Ensayar (i+1)
    else
        comprobar si es óptimo
    endif
    eliminar objeto i
 endif

 if exclusión de i es aceptable then
    if i < n then // Probar con todos
        Ensayar (i+1)
    else
        comprobar si es óptimo
    endif
 endif
```

- Si **i** no supera el peso permitido (inclusión aceptable) se siguen añadiendo objetos
- En caso contrario se abandona la inclusión de más objetos

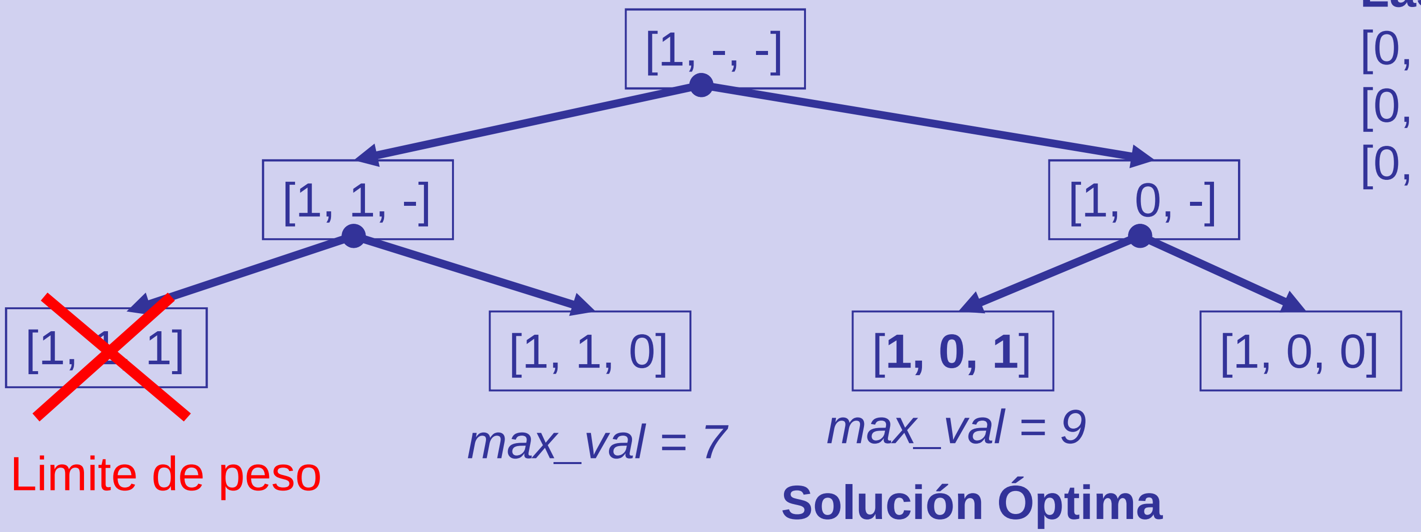
- Se excluye **i** si con su valor no se alcanza el óptimo
- En caso contrario se mantiene

Ejemplo (Mochila)

O	1	2	3
P	2	3	4
V	4	3	5

- **Limite de peso 6**

- Combinaciones posibles de objetos $2^3=8$
- Soluciones candidatas (restricción de peso)
  - S_1 = {1, 1, 0} (*pesos 2 y 3*) y S_2 = {1, 0, 1} (*pesos 2 y 4*)
- Solución óptima (teniendo en cuenta el valor)
  - S = {1, 0, 1} (*pesos 2 y 4*)



Las soluciones:
[0, 0, 0] Mochila vacía
[0, 1, 0] No óptima
[0, 0, 1] No óptima

- No se exploran todas las soluciones
  - Limite de **peso** (inclusión aceptable)
  - Mejora del **valor** transportado (exclusión aceptable)
  - Algoritmo *Branch and Bound*
- El algoritmo utiliza dos variables globales:
  - **max_val** para guardar el máximo valor obtenido (solución óptima) hasta el momento
  - ***S*** solución del problema (objetos incluidos de **O**), cuando max_val aumenta de valor sobre una combinación de objetos previa
- La exclusión se lleva a cabo cuando el valor alcanzable por la selección en curso es menor que el máximo valor alcanzado (max_val)

Estructuras de Datos y variables:

```
typedef Objeto is record
                P: Natural;   --- Peso;
                V: Natural;   --- Valor;
         end Record;

typedef Objetos is array (1..Max) of Objeto;

limitePeso: constant Natural := .....;

--- Variables
O    : Objetos;
max_val: Natural;    --- máximo valor alcanzado por una solución
tot_val   : Natural;    --- valor de todos los elementos del conjunto O
S          : Set;        --- solución (conjunto de elementos del conjunto O)
tmpS      : Set;        --- solución temporal
```

```
procedure Ensayar (i          : Natural;
                   peso_tot: Natural;
                   va         : Natural) is
  va_tmp: Natural; --- valor alcanzado
begin
   if (peso_tot + O(i).P <= limitePeso) then
     tmpS := tmpS + {i};
     if i < O'Last then
        Ensayar (i+1, peso_tot + O (i).p, va);
     elsif va > max_val then
        max_val := va;
        S := tmpS;
      end if;
      tmpS := tmpS – {i};
   end if;
```

```
   va_tmp := va – O(i).V;
   if va_tmp > max_val then
     if i < O'Last then
        Ensayar (i+1, peso_tot, va_tmp);
     else
        max_val := va_tmp;
        S := tmpS;
     end if;
   end if;
end Ensayar;

begin --- programa principal
  max_val := 0;
  tmpS := {};
  S := {};
  tot_val := Σ O[i].V;
  Ensayar (1, 0, tot_val);
end Programa;
```