Análisis y Diseño de Algoritmos

Tema 6: Vuelta Atrás

Titulación: Grado en Ingeniería del Software

UNIVERSIDAD DE MÁLAGA

E.T.S. INGENIERÍA INFORMÁTICA

Contenido

- Introducción
- Técnicas de Vuelta Atrás
  - Conceptos generales
  - El problema de las n-reinas
  - El problema del laberinto
- Extensiones
  - Backtracking para enumeración
  - Backtracking para optimización
- Referencias

Introducción

- Características de los algoritmos voraces
  - Son métodos constructivos para el cálculo de soluciones a un problema por pasos.
  - En cada paso del algoritmo se toma una decisión que produce una solución parcial más cercana a la solución completa del problema.

ESQUEMA GREEDY

```
Solucion sol = solInic;
while !esSolucion(sol){
  cont = seleccionarMejor(sol);
  sol = sol + cont;
}
```

Introducción

- El método *seleccionarMejor* escoge la mejor de entre todas las continuaciones posibles, teniendo en cuenta algún criterio que sólo se aplica a las soluciones parciales, pero que no tiene que ser el mejor para encontrar la solución óptima al problema.

ESQUEMA GREEDY

```
Solucion sol = solInic;
while !esSolucion(sol){
  cont = seleccionarMejor(sol);
  sol = sol + cont;
}
//sol no tiene por qué ser la solución
// óptima
```

Introducción

- Una vez escogida una continuación no hay forma de cambiar de opinión, la decisión es irreversible.
- La ventaja es la eficiencia del algoritmo.
- La desventaja es que en muchos casos se obtienen soluciones lejanas a la óptima.

ESQUEMA GREEDY

```
Solucion sol = solInic;
while !esSolucion(sol){
  cont = seleccionarMejor(sol);
  sol = sol + cont;
}
//sol no tiene por qué ser la solución
// óptima
```

Vuelta atrás (backtracking)

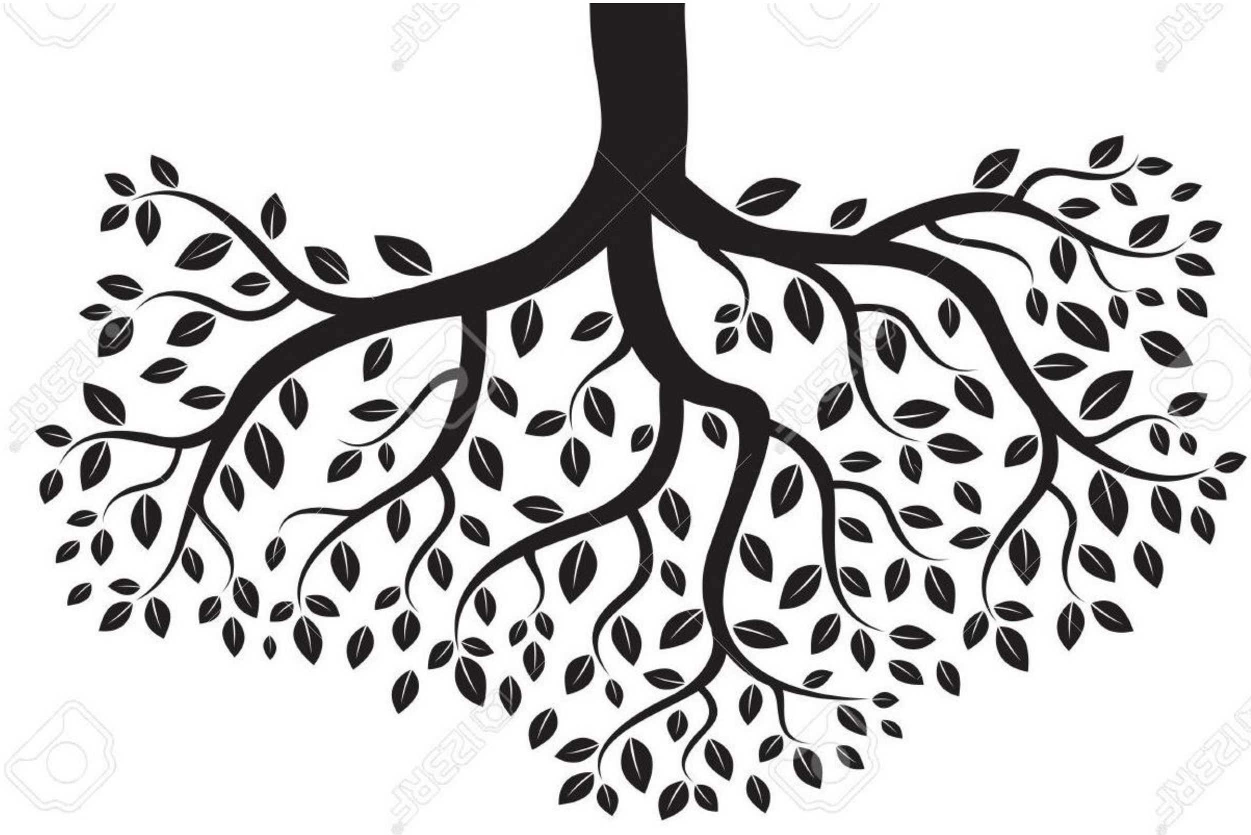
- En una hoja del árbol de la figura está la solución a un problema. ¿Cómo llegar a ella?

Vuelta atrás (backtracking)

- Nosotros vemos los árboles boca abajo

Vuelta atrás (backtracking)

- Nosotros vemos los árboles boca abajo y con la raíz reducida a un punto



Vuelta atrás (backtracking)

- Un video de ejemplo:
- https://www.youtube.com/watch?v=g1hiCeANfDo
- Amarillo avance según un criterio sistemático
- Gris oscuro: aún no visitado
- Gris claro zona visitada y retrocedido.
- Segundo 0:32, dónde se va a producir un retroceso importante.
- Segundo 0:44, ya cerca de la salida, pero ha elegido otro camino y después rectifica.
- Segundo 1:32, tras haber visitado casi todo el cuadrante superior izquierdo, ha retrocedido para inicial nueva ruta desde el origen.

Vuelta atrás (backtracking)

- El backtracking es también un método algoritmo constructivo, en el que las soluciones se obtienen mediante un proceso iterativo en el que en cada paso se obtiene una solución más cercana a la solución final.
- A diferencia de los métodos voraces, las decisiones tomadas durante la ejecución del algoritmo pueden deshacerse (puede volverse atrás y tomar un camino distinto).

ESQUEMA VueltaAtrás

```
Solucion backtracking(Solucion sol){
    if (esSolucion(sol)) return sol;
    else{
        Cont setCont = posContinuacion(sol);
        Solucion solAux = null;
        while (!esVacia(setCont) && solAux==null){
            cont = selecciona(setCont);
            setCont = SetCont – {cont};
            solAux=backtracking(sol+cont);
        }
        return solAux;
    }
}
```

Vuelta atrás (backtracking)

Si sol es una solución terminamos

Construimos el conjunto de posibles continuaciones

Si es vacío, nuestra solución parcial no lleva a ninguna solución y devolvemos null

Si hay alguna posible continuación, la escogemos y continuamos de forma recursiva con ella.

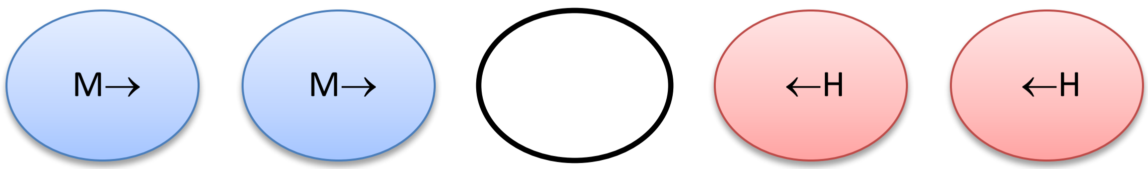
Si la llamada recursiva nos devuelve una solución terminamos, sino continuamos con la siguiente posible continuación.

ESQUEMA VueltaAtrás

```
Solucion backtracking(Solucion sol){
    if (esSolucion(sol)) return sol;
    else{
        Cont setCont = posContinuacion(sol);
        Solucion solAux = null;
        while (!esVacia(setCont) && solAux==null){
            cont = selecciona(setCont);
            setCont = SetCont – {cont};
            solAux=backtracking(sol+cont);
        }
        return solAux;
    }
}
```

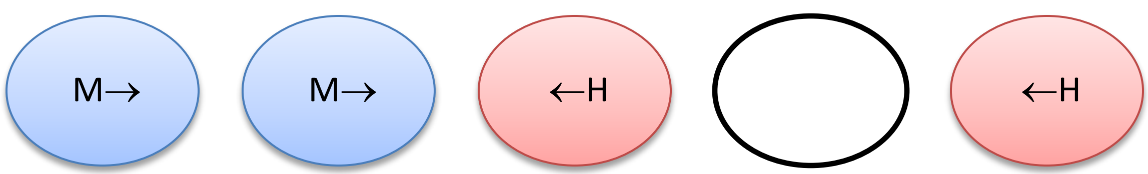
Backtracking: árbol de búsqueda
El problema de las ranas

Supón que hay n (=2) ranas macho (M) y n ranas hembra (H), dispuestas sobre la fila de 2n+1 piedras tal y como aparece en la figura:



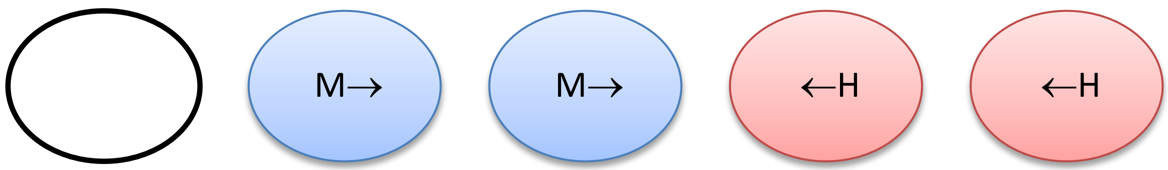
Cada rana está mirando en la dirección de la flecha, y sólo pueden moverse en esa dirección

Cada rana puede saltar a la piedra adyacente si está vacía:

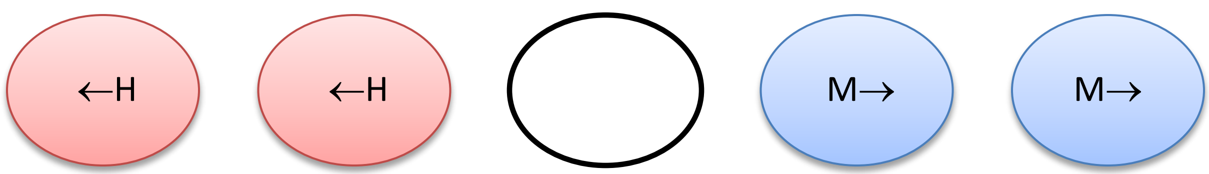


Backtracking: árbol de búsqueda
El problema de las ranas

- O puede salta a la segunda piedra adyacente, si está vacía, saltando sobre otra rana.



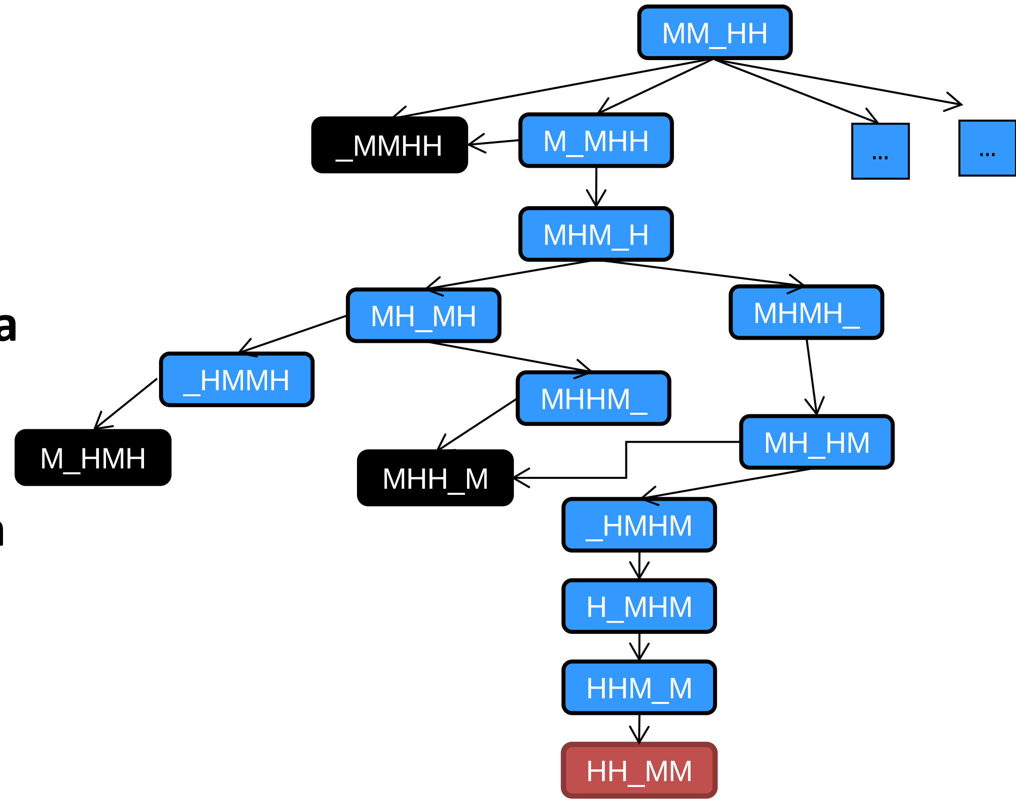
- ¿existe una secuencia de movimientos que intercambien a las ranas tal como aparecen en el dibujo?



Backtracking: árbol de búsqueda
El problema de las ranas

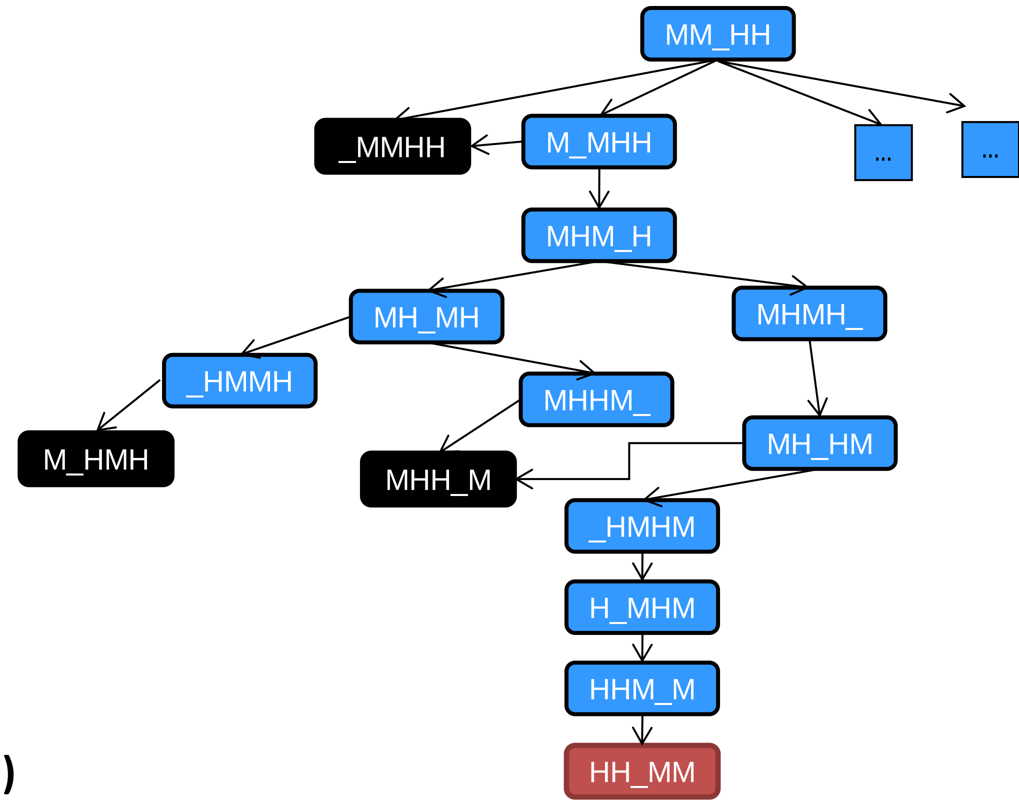
El árbol del espacio de estados es el árbol que construye el algoritmo durante la búsqueda de la solución.

- **La raíz del árbol es el estado inicial.**
- **Los nodos del primer nivel corresponden a la elección hecha para la primera componente de la solución.**
- **Los nodos del segundo nivel son las elecciones realizadas para la segunda componente de una solución.**



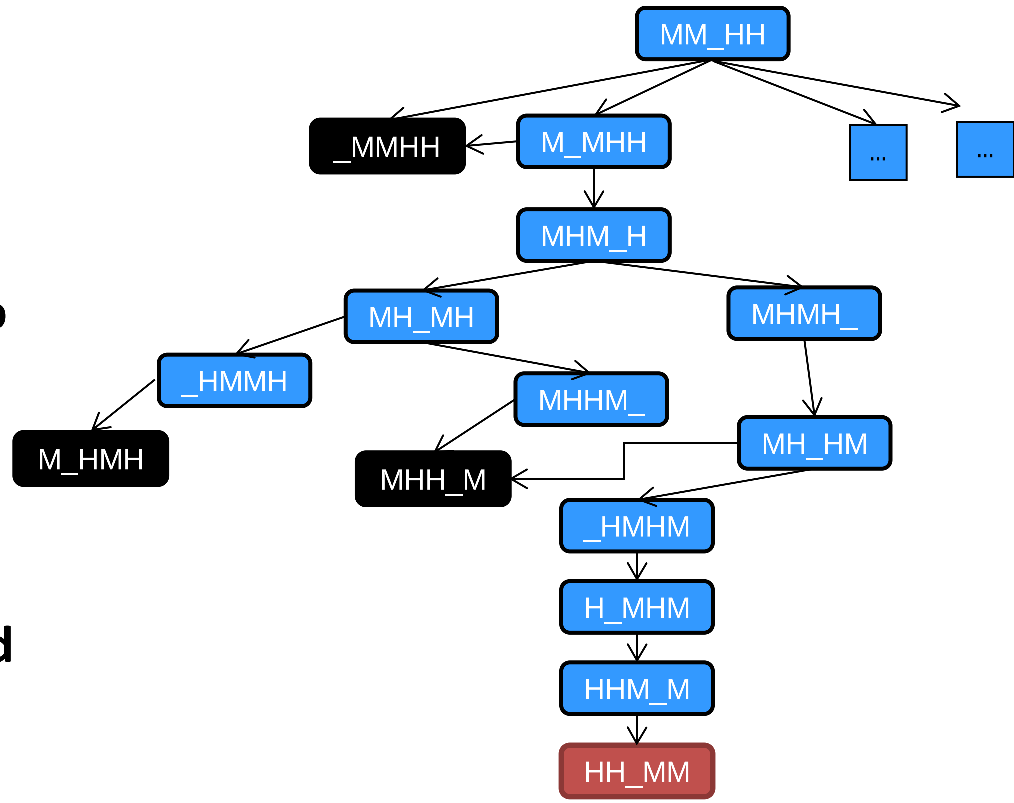
Backtracking: árbol de búsqueda El problema de las ranas

- **Hay nodos del árbol que corresponden a soluciones parciales que pueden llevar a una solución global. Estos nodos tienen sucesores en el árbol.**
- **Hay otros nodos, que deben descartarse, porque corresponden a soluciones parciales a partir de las cuales no es posible encontrar una solución global. En estos nodos se realiza el backtracking.**
- **El algoritmo puede parar cuando encuentra una solución o seguir buscando soluciones distintas por otros caminos.**
- **En la búsqueda de las soluciones, los nodos suelen visitarse utilizando un recorrido dfs (primero en profundidad)**



Backtracking: árbol de búsqueda
El problema de las ranas

- **Cada nodo es un estado**
- **Entre dos nodos hay una transición entre estados**
- **El árbol de estados puede ser muy grande, pero normalmente no existe de forma explícita, sino que los estados se van construyendo a medida que el algoritmo avanza (on-the-fly)**
- **Sólo se tienen almacenados los estados de la pila de recursividad que en cada momento se está resolviendo.**

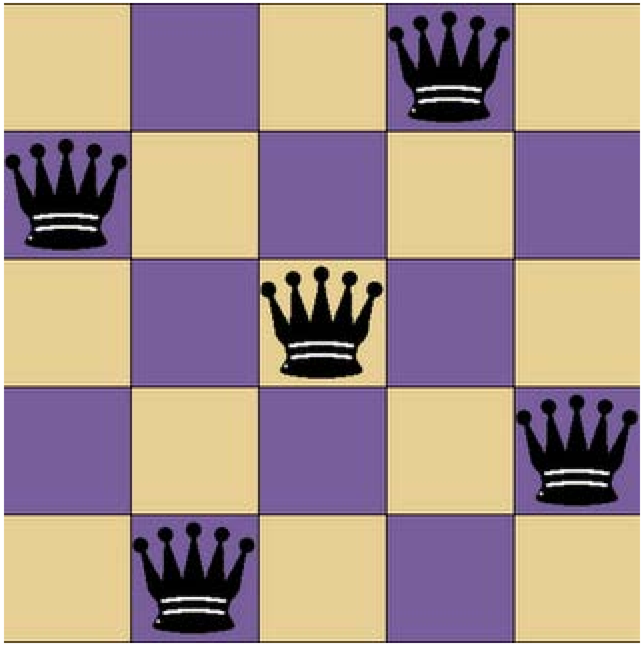


Backtracking: caracterícticas generales

- Cada vez que se hace backtracking se está podando el árbol: nos ahorramos considerar todas las decisiones pendientes desde ese punto.
- El coste es que, en el caso peor, el algoritmo tendrá que recorrer todo el espacio de búsqueda , lo que puede ser equivalente a un algoritmo de "fuerza bruta".
- En cualquier caso, la técnica permite encontrar soluciones "simples" a problemas aparentemente difíciles de resolver.
- Los algoritmos que utilizan backtracking suelen combinar recursividad e iteración, por lo que hay que tener mucho cuidado a la hora de manejar los casos bases de la recursión, y las condiciones de terminación de los bucles.

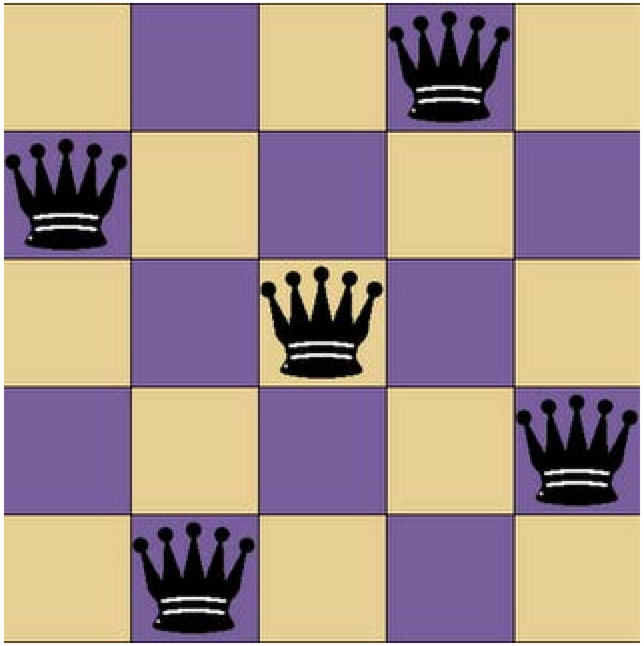
El problema de las n reinas

- Supongamos que tenemos un tablero de ajedrez de tamaño nxn. Queremos colocar n reinas de manera que no se ataquen, es decir, que no haya
  - dos reinas en la misma fila
  - dos reinas en la misma columna
  - dos reinas en la misma diagonal



Solución al problema

- Como dos reinas no pueden estar en la misma filas, las soluciones pueden ser listas como l=[3,0,2,4,1] en las que cada elemento l[i] representa la columna de la fila i en la que está la reina.



Solución al problema

- La lista l se construye de forma incremental.
- Partimos de una lista vacía *[]*
- Suponiendo que la lista es $[a_0,\ldots,a_{i-1}]$, para añadir una nueva reina en la fila i,
  - escogemos la primera posición a_i de una reina que no ataca a ninguna de las que ya están en el tablero.
  - si no es posible, retrocedemos y cambiamos a_{i-1} para buscar otra solución
  - y así sucesivamente…

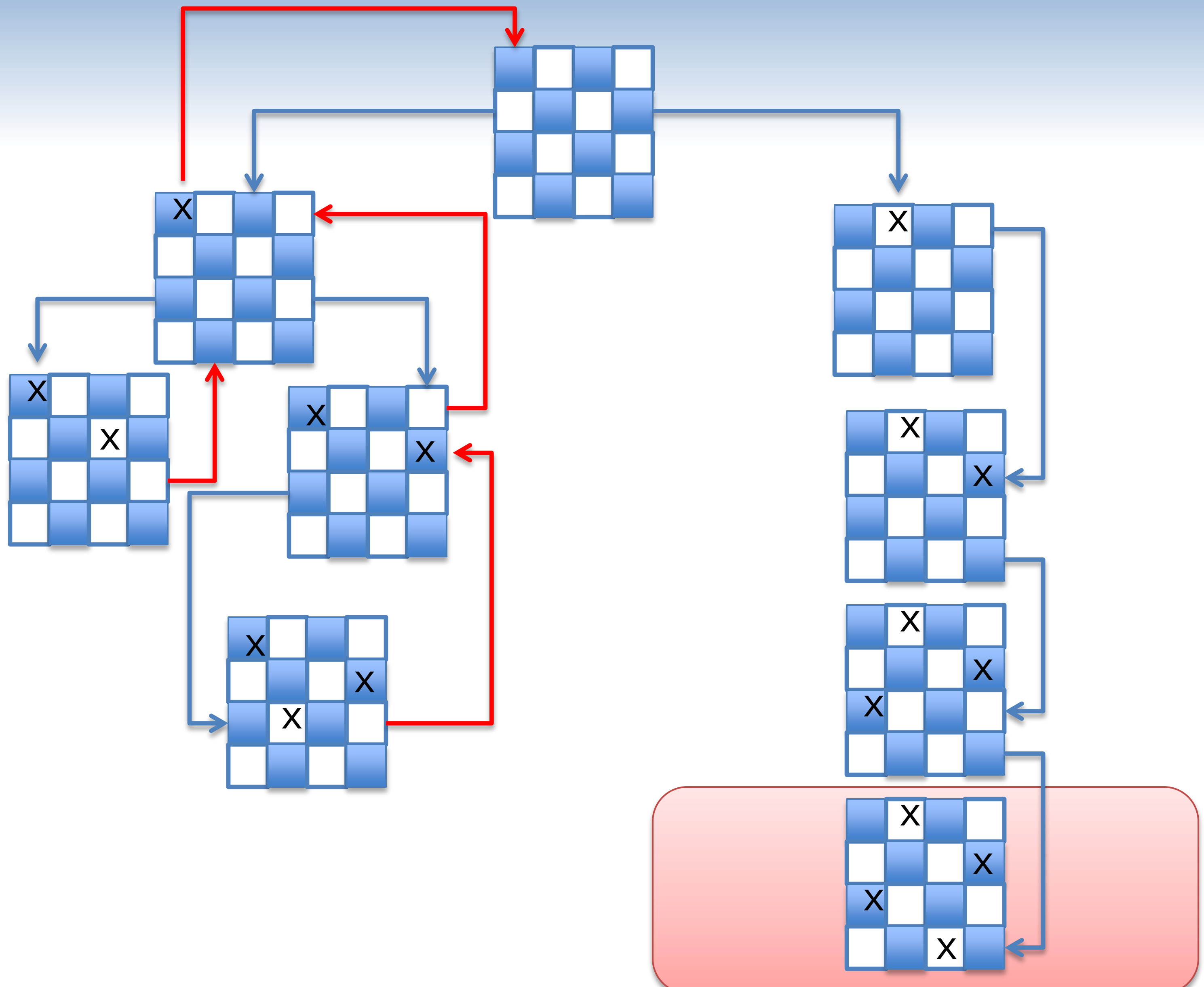
El problema de las n reinas código java

```
public static List<Integer> reinas(int n){
   for (int i = 0; i<n; i++){
       List<Integer> lista = new ArrayList<Integer>();
       lista.add(i);
       boolean sol = reinas(n,lista);
       if (sol) return lista;
   }
   return null;
}

public static boolean reinas(int n, List<Integer> lista){
    if (lista.size()==n) return true;
    for (int i = 0; i<n; i++){
       if(!lista.contains(i)&& noCome(i,lista)){
          lista.add(i);
          if (reinas(n,lista)) return true;
          else lista.remove(new Integer(i));
       }
    }
    return false;
}
```

El problema de las n reinas código java

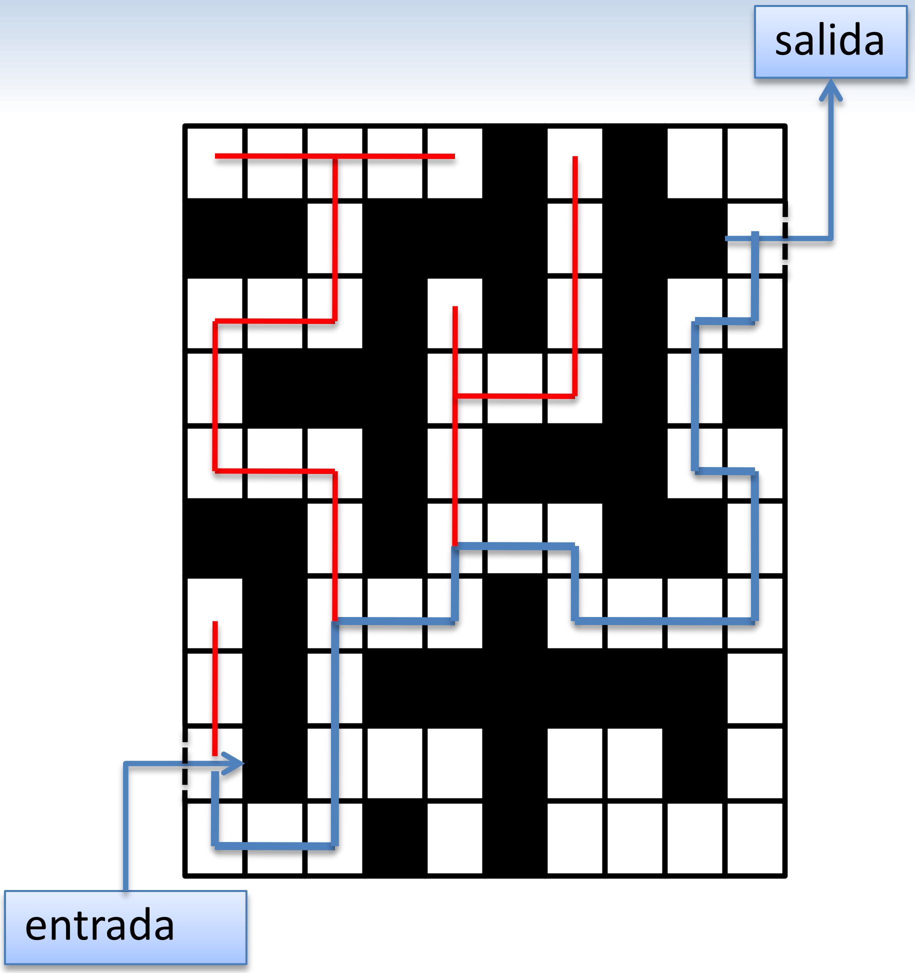
```java
public static boolean noCome(int i, List<Integer> lista){

   int j = lista.size();

   for (int k = 0; k <lista.size(); k++){

      if (j-i == k - lista.get(k)) return false;

      if (i+j == k + lista.get(k)) return false;
   }
   return true;
}
```

Ejemplo (caso n = 4)



El problema del laberinto

- Tenemos un laberinto que queremos atravesar desde un punto de entrada hasta un punto de salida.
- El laberinto está representado por un array bidimensional (cuadrado), en el que cada componente puede ser o no un obstáculo.
- La solución al problema es un camino que transcurra desde la entrada hasta la salida a través de componentes no bloqueadas.



El problema del laberinto: Una solución

- Una solución puede representarse como una lista de posiciones válidas del laberinto, que empieza en la entrada, y termina en la salida.

```
public class Posicion {
   private int x;
   private int y;
   public Posicion(int i,int j){
     this.x = i; this.y = j;
   }
   public int x(){
     return x;
   }
   public int y(){
     return y;
   }
   public boolean equals(Object o){
     return (o instanceof Posicion)&&
            ((Posicion)o).x==x &&
            ((Posicion)o).y==y;
   }
   public int hashCode(){
      return x*7+y*17;
   }

   public String toString(){
      return "("+x+","+y+")";
   }
...}
```

El problema del laberinto: Una solución

- Dada una posición, la siguiente en el camino de salida puede ir hacia el norte, sur, este u oeste.

```
public class Posicion {
………
   public Posicion siguiente(int dir){
     if (dir==0) return new Posicion(i-1,j);
     else if (dir==1) return new Posicion(i+1,j);
     else if (dir==2) return new Posicion(i,j+1);
     else return new Posicion(i,j-1);
   }
}
```

El problema del laberinto: Una solución

- El laberinto es un objeto que tiene un array bidimensional, la entrada, la salida, y un método para saber si una posición está dentro del laberinto

```
public class Laberinto {
   private int[][] laberinto;
   private Posicion entrada,salida;

   public Laberinto(int[][] lab,Posicion ent,Posicion sal){
     this.laberinto = lab;
     this.entrada = ent;
     this.salida = sal;
   }

   public  boolean estaEnLaberinto(Posicion pos){
     if (pos.x()<0 || pos.x()>=laberinto.length) return false;
     if (pos.y()<0 || pos.y()>=laberinto.length) return false;
     return true;
   }
……
```

El problema del laberinto: Una solución

- La clase laberinto tiene un método *solucion()* que produce una lista de posiciones con un camino en el laberinto desde la entrada hasta la salida, o null, si no existe ninguna solución
- El método llama a un método local que recorre el laberinto de forma recursiva.

```
public class Laberinto {
     …..
     public List<Posicion> solucion(){
        List<Posicion> lista = new ArrayList<Posicion>();
        lista.add(entrada);
        if (solucion(lista)) return lista;
        else return null;
     }
……
```

El problema del laberinto: Una solución

```
public class Laberinto {

   private boolean solucion(List<Posicion> lista){
        if (lista.isEmpty()) return false;
        Posicion posAct = lista.get(lista.size()-1);
        if (posAct.equals(salida)) return true;
        for (int i=0; i<4; i++){
           Posicion posSig = posAct.siguiente(i);
           if (!lista.contains(posSig)&&  // no hay ciclos en el camino
               estaEnLaberinto(posSig)&&  // la nueva posición es correcta
               laberinto[posSig.x()][posSig.y()]!=1){
                   lista.add(posSig);
                   if (! solucion(lista)) lista.remove(lista.size()-1);
                   else return true;
           }
         }
         return false;
    }
```

backtracking

El backtracking puede hacer más cosas

- El enfoque y aplicaciones mostradas anteriormente corresponden a la resolución de problemas de decisión o satisfacción, es decir,
  - determinar si un problema tiene o no solución de una forma dada
  - construir una solución de un tipo específico o determinar que no existe
- El backtracking puede utilizarse también en otro tipo de situaciones:
  - Problemas de conteo o enumeración: determinar cuantas soluciones de las características deseadas existen, o encontrarlas todas
  - Problemas de optimización: encontrar una solución que haga máxima alguna función de calidad.

Backtracking para encontrar todas las soluciones

- En los problemas anteriores, el objetivo era encontrar una solución válida, por lo que en el momento en el que se encuentra, la búsqueda termina.
- Para adaptar este enfoque a problemas de enumeración hay que continuar la búsqueda hasta que se han encontrado todas las soluciones.
- El árbol de búsqueda ha de recorrerse completamente, visitando todos los nodos prometedores.

Backtracking por enumeración

ESQUEMA VueltaAtrás (encontrar el número de soluciones)

```
int backtracking(Solucion sol){
    if (esSolucion(sol)) return  1;
    else{
        Cont setCont = posContinuacion(sol);
        int num = 0;
        while (!esVacia(setCont)){
            cont = selecciona(setCont);
            setCont = SetCont – {cont};
            int num1=backtracking(sol+cont);
            num += num1
        }
        return num;
      }
}
```

Backtracking por enumeración

ESQUEMA VueltaAtrás (encontrar todas las soluciones)

```
void backtracking(List<Solucion> listaSol, Solucion sol){
     if (esSolucion(sol))  listaSol.add(sol);
    else{
         Cont setCont = posContinuacion(sol);
         while (!esVacia(setCont)){
              cont = selecciona(setCont);
              setCont = SetCont – {cont};
              backtracking(listaSol, sol+cont);
          }
    }
}
```

Tamaño del árbol de búsqueda

- El proceso de enumeración puede ser muy costoso debido al tamaño del árbol de búsqueda.
- Este tamaño puede estimarse como sigue:
  - Sea c_1 el número de decisiones posibles en el nodo inicial
  - Se elige una decisión al azar, y se repite el procedimiento obteniéndose c_i, $1 \leq i \leq L$
  - Si c_i se toma como promedio de las decisiones factibles en el nivel i-ésimo, entonces el tamaño T del árbol es

$$T = c_1 + c_1 c_2 + c_1 c_2 c_3 + \cdots + = \sum_{i=1}^{L} \prod_{j=1}^{i} c_i$$

El problema del la n reinas

```
public static void reinas(int n,List<List<Integer>> todas){
  List<Integer> actual = new ArrayList<Integer>();
  reinas(n,actual,todas);
}

public static void reinas(int n, List<Integer> actual, List<List<Integer>> todas){
    if (actual.size()==n)  todas.add(actual);
    else{
      for (int i = 0; i<n; i++){
         if(!actual.contains(i)&& noCome(i,actual)){
            List<Integer> actualaux = new ArrayList<Integer>(actual);
            actualaux.add(i);
            reinas(n,actualaux,todas);
         }
       }
    }
}
```

Ejemplo para n = 5

```
[[0, 2, 4, 1, 3],
 [0, 3, 1, 4, 2],
 [1, 3, 0, 2, 4],
 [1, 4, 2, 0, 3],
 [2, 0, 3, 1, 4],
 [2, 4, 1, 3, 0],
 [3, 0, 2, 4, 1],
 [3, 1, 4, 2, 0],
 [4, 1, 3, 0, 2],
 [4, 2, 0, 3, 1]]
```

Sólo queremos lo mejor

- En el caso de problemas de optimización, nos interesa la mejor, de entre todas las soluciones factibles.
- El esquema de la solución es muy similar, ya que hay que explorar todas las soluciones válidas posibles.
- Tenemos que arrastrar la mejor solución conocida en cada momento para compararla con cada solución que se encuentre.

Backtracking para optimización

ESQUEMA VueltaAtrás (la mejor solución)

```
Solucion backtracking(Solucion sol,Solucion msol, int mcal){
    if (esSolucion(sol)) {
        int cal = calidad(sol);
        if (mcal  > cal)  return msol;
        else  return sol;
    };
    else{
        Cont setCont = posContinuacion(sol);

        while (!esVacia(setCont)){
            cont = selecciona(setCont);
            setCont = SetCont – {cont};
            Solucion otra=backtracking(sol+cont,msol,mcal);
            if (calidad(otra)>mcal) msol = otra; mcal = calidad(otra)
        }
        return msol;
    }
}
```

El problema de la suma de subconjuntos

Dado un conjunto $S = \{s_1, \cdots, s_n\}$ de números naturales, y un número d encontrar un subconjunto $S' \subseteq S$ tal que

- $\sum_{s \in S'} s \leq d$
- Si $R \subseteq S$, entonces
  - o bien $\sum_{r \in R} r > d$,
  - o $\sum_{r \in R} r \leq \sum_{s \in S'} s$

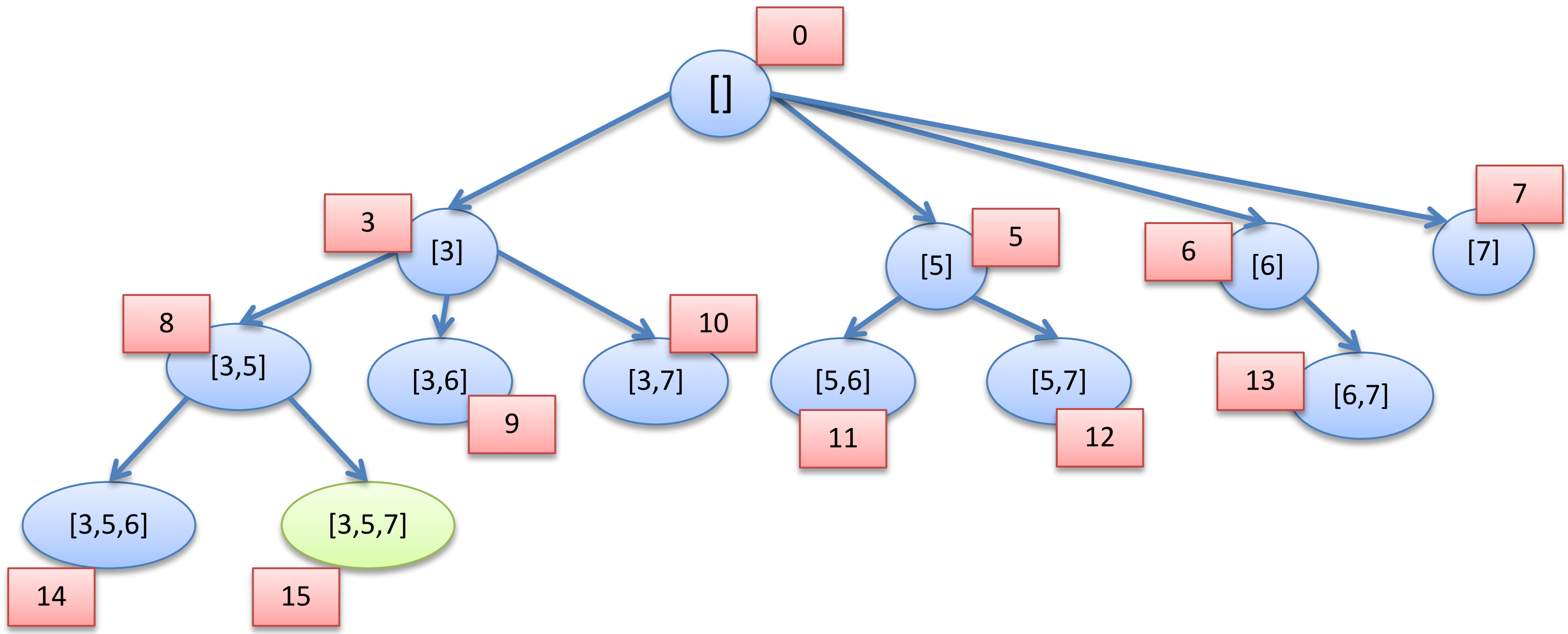
El problema de la suma de subconjuntos

```
public static SortedSet<Integer> mejorSub(SortedSet<Integer> conj, int d){
   SortedSet<Integer> sub = new TreeSet<Integer>();
   return mejorSub(conj,sub,d,0);
}
```

```
public static SortedSet<Integer> mejorSub(SortedSet<Integer> conj,
                                           SortedSet<Integer> mejor,int d,int cal){
   SortedSet<Integer> conjMejor = mejor;
   int sumaMejor = cal;
   for (int elem:conj){
      if (!mejor.contains(elem) && noMenor(elem,mejor)&& cal+elem <= d ){
         SortedSet<Integer> aux = new TreeSet<Integer>(mejor);
         aux.add(new Integer(elem));
         SortedSet<Integer> otro = mejorSub(conj,aux,d,cal+elem);
         int suma = 0;
         for (int e:otro) suma+=e;
         if (suma > sumaMejor) {
            mejor = otro;sumaMejor = suma;
         }
      }
   }
   return conjMejor;
}
```

```
public static boolean noMenor(int elem, SortedSet<Integer>
set){
   for (int e:set){
       if (elem <= e) return false;
   }
   return true;
 }
```

Árbol de Búsqueda



Referencias

- *Introduction to The Design & Analysis of Algorithms*. A. Levitin. Ed. Adison-Wesley
- *Introduction to Algorithms*. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. Ed. The MIT Press