## Grado en Ingeniería Información

# Estructura de Datos y Algoritmos

## Sesión 14

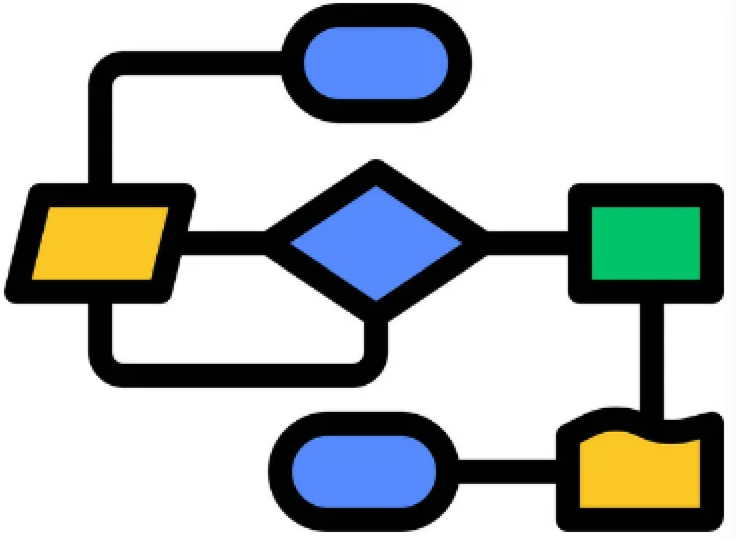
**Curso 2023-2024**

Marta N. Gómez

UNIVERSIDAD NEBRIJA

## T4. Técnicas Algorítmicas

- Divide y Vencerás
- Algoritmos Voraces
- Programación Dinámica
- Backtracking

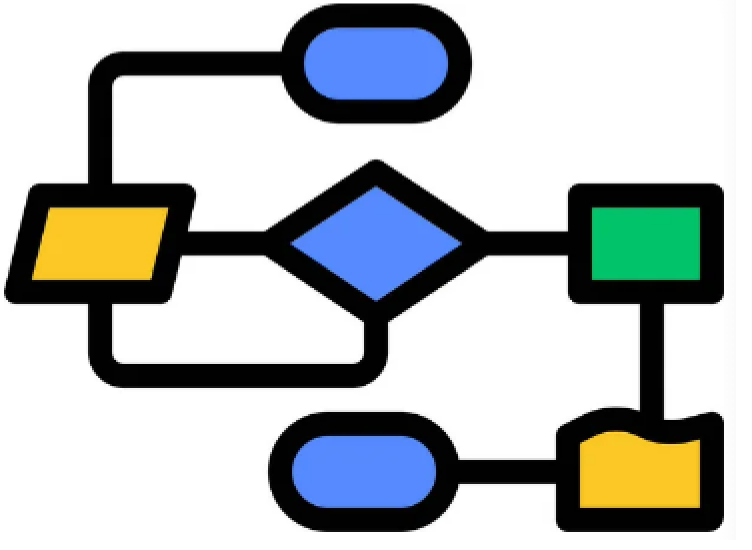


NEBRIJA UNIVERSIDAD

**ÍNDICE**

**T4. Técnicas Algorítmicas**

- Divide y Vencerás
- Algoritmos Voraces
- Programación Dinámica
- **Backtracking**



UNIVERSIDAD NEBRIJA

# ÍNDICE

El diseño o técnica **Backtracking** o **Vuelta Atrás** proporciona una manera **sistemática** de generar **todas las posibles soluciones** siempre que dichas soluciones sean susceptibles de **resolverse en etapas**.

Esta técnica realiza una búsqueda **exhaustiva** y **sistemática** en el **espacio de soluciones**, donde añade y quita los elementos para **probar todas las combinaciones posibles**.

**En general**, se consideran **problemas** donde la **solución** se construye por **etapas** y puede **representarse como una n-tupla** $(x1, x2, ..., xn)$, donde cada $xi$ es seleccionado de un conjunto $S_i$ que representa la decisión de la **etapa** ***i-ésima***, del **conjunto de alternativas existente**.

Además, una solución tendrá que **minimizar**, **maximizar** o simplemente **satisfacer** una cierta **función objetivo**.

Por tanto, hay **dos tipos de restricciones**:

- **Explícitas**: indican los conjuntos $S_i$.
- **Implícitas**: relaciones entre los componentes de la **tupla solución** para satisfacer **la función objetivo**.

El **espacio de soluciones deben satisfacer las <u>restricciones explícitas</u>** y se puede estructurar en un **<u>árbol de exploración</u>**.

En **cada nivel se toma la decisión de la etapa** correspondiente:

- **Nodo estado**: nodo correspondiente a una **tupla parcial** o **completa** que satisface las **restricciones explícitas**.
- **Nodo solución**: nodo de las **tuplas completas** que satisfacen las **restricciones implícitas**.

Otro elemento importante en la aplicación de esta técnica son las **funciones de poda o test de factibilidad** (se obtienen a partir de la **función objetivo**).

Estas funciones permiten **determinar cuándo una tupla parcial nunca llegará a ser solución**. Por tanto, no se debe de mantener.

De las posibles **formas de recorrer el árbol**, destacan dos que dan lugar a **dos técnicas**:

- **Vuelta atrás**: **recorrido en profundidad**, de forma que los **nodos vivos se gestionan mediante una pila**. Método sencillo y eficiente en espacio.
- **Ramificación y poda**: corresponde a una ´búsqueda más inteligente porque siempre **se expande el nodo vivo más prometedor**, de forma que los nodos vivos se gestionan a través de una cola con prioridad.

Según el tipo de problema que se tenga que resolver, existen diferentes formas de aplicar la técnica Backtraking o Vuelta atrás donde partiendo del **árbol de exploración:**

- **Vuelta atrás para una solución:** el algoritmo realiza un recorrido del árbol en profundidad hasta encontrar la **primera solución**. Es el caso más sencillo.
- **Vuelta atrás para todas las soluciones:** el algoritmo recorre el árbol completo (salvo las zonas podadas) para obtener y guardar **todas las soluciones que encuentra**.
- **Vuelta atrás para la mejor solución:** el algoritmo recorre el árbol comparando cada solución que encuentra con la mejor solución obtenida hasta el momento y, se queda con la mejor, la **solución óptima**. Cuando ya no hay más soluciones, devuelve la solución mejor. Suele aplicarse a problemas de óptimización.

Durante el proceso, **para cada nodo se van generando sus nodos sucesores**:

- **Nodos vivos**: los que todavía tienen hijos pendientes de generarse.
- **Nodos en expansión**: los que tienen hijos que están siendo generados.
- **Nodos muertos**: los que no pueden ser expandidos, porque no han superado el test de factibilidad o porque todos sus hijos ya han sido generados.

## Esquema general:

Cuando se hace el recorrido en profundidad y se alcanza un nodo muerto, hay que deshacer la última decisión tomada, para tomar la siguiente alternativa (igual que en un laberinto al alcanzar un callejón).

```
void vueltaAtras (vector<int>  &S, int k)  {
    preparar_recorrido_nivel (k);
    while (!ultimo_hijo_nivel(k)) {
        S.at(k) = siguiente_hijo_nivel(k);
        if (esSolucion(S, k)) {
            tratarSolucion(S);
        } else if (esCompletable(S, k)) {
                vueltaAtras(S, k+1);
            }
    }
}
```

El **coste temporal** de un algoritmo de **Backtraking** o **Vuelta atrás** suele depender de:

- ***v(n)***: Número nodos del espacio de búsqueda que se visitan.
- ***f(n)***: El coste de las funciones **esSolucion** y **esCompletable** en cada nodo.

Luego, el total será: $O(v(n)f(n))$

Si la función **esCompletable descarta** muchos nodos, ***v(n)* se reduce**:

- Si queda **un solo nodo**, el coste será: $O(nf(n))$
- Si **no se descarta ninguno** (peor caso), el coste será: $O(n^k f(n))$

## El problema de las N reinas

Consiste en **ubicar *N* reinas en un tablero** (dimensión N x N) **sin que se "amenacen"** la una a la otra, es decir, que nunca se encuentren dos reinas **ni en la misma fil, ni en la misma columna, ni en la misma diagonal.**

Si numeramos las filas y las columnas de ***1*** a ***n*** y lo mismo con las reinas. Se puede asumir que la reina ***i*** estará en alguna posición de la fila ***i***. Luego las soluciones se pueden representar por tuplas: (***x1, …, xn***) donde ***xi*** será la columna que ocupa la reina ***i*** en la fila ***i***.

## El problema de las N reinas

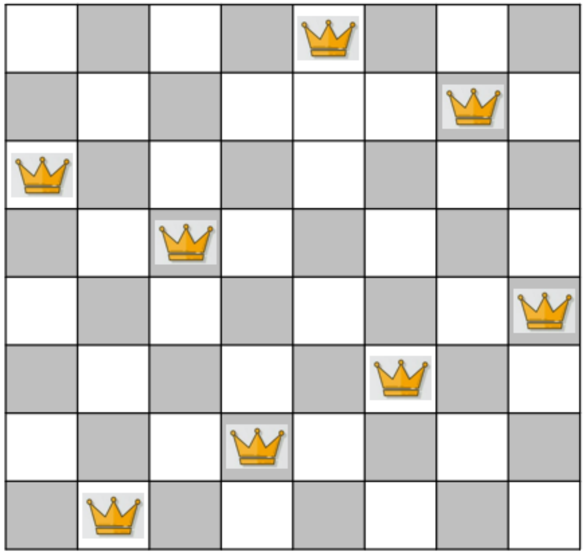
Vamos a realizar el análisis para **8 reinas** en un **tablero de 8x8**.

La solución estará representada por:

$$(\boldsymbol{x1, x2, x3, x4, x5, x6, x7, x8})$$

Donde ***xi***: **columna** donde está la **reina de la fila *i***.

Una posible solución sería: (***5, 7, 1, 3, 8, 6, 4, 2***)



Las restricciones a considerar son:

- **Restricciones explícitas**: la tupla solución debe de contener los valores (***1, 2, 3, 4, 5, 6, 7, 8***).
- **Restricciones implícitas**:

  **(1)** **dos reinas no pueden ocupar la misma columna**, luego no pueden existir dos ***xi*** iguales en la tupla. Requiere que cada tupla sea permutación de (***1,…, n***).

  **(2)** **dos reinas no pueden estar en la misma diagonal**. Así, si las coordenadas de dos reinas en el tablero son ***(x, y)*** y ***(x', y')***, **están en la misma diagonal si y solo si**:

$$|x - x'| = |y - y'|$$

Por tanto, en cada etapa ***k*** se irán generando las k-tuplas que sean factibles (con posibilidad de ser solución).

Respecto a la **segunda restricción** hay que tener en cuenta que para:

- Las posiciones sobre una **misma diagonal descendiente** se cumple que tienen el mismo valor de la **resta entre la fila y la columna**.

$$x - y = x' - y' \Rightarrow x - x' = y - y'$$

- Mientras que posiciones sobre una **misma diagonal ascendiente** se cumple que tienen el mismo valor de la **suma entre la fila y la columna**.

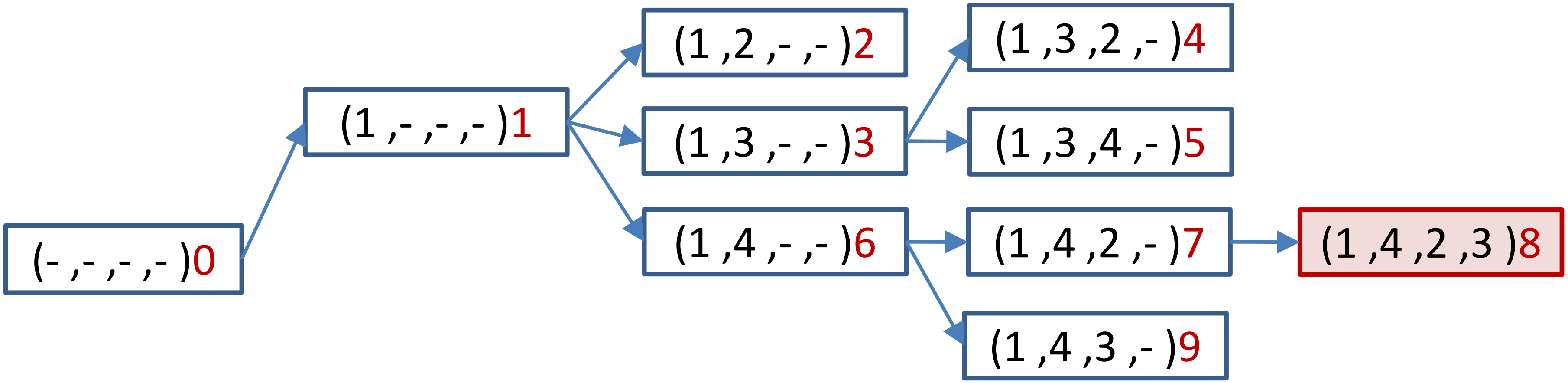
$$x + y = x' + y' \Rightarrow x - x' = y' - y$$

De ahí se obtiene que **están en la misma diagonal** si y solo si:

$$|x - x'| = |y - y'|$$

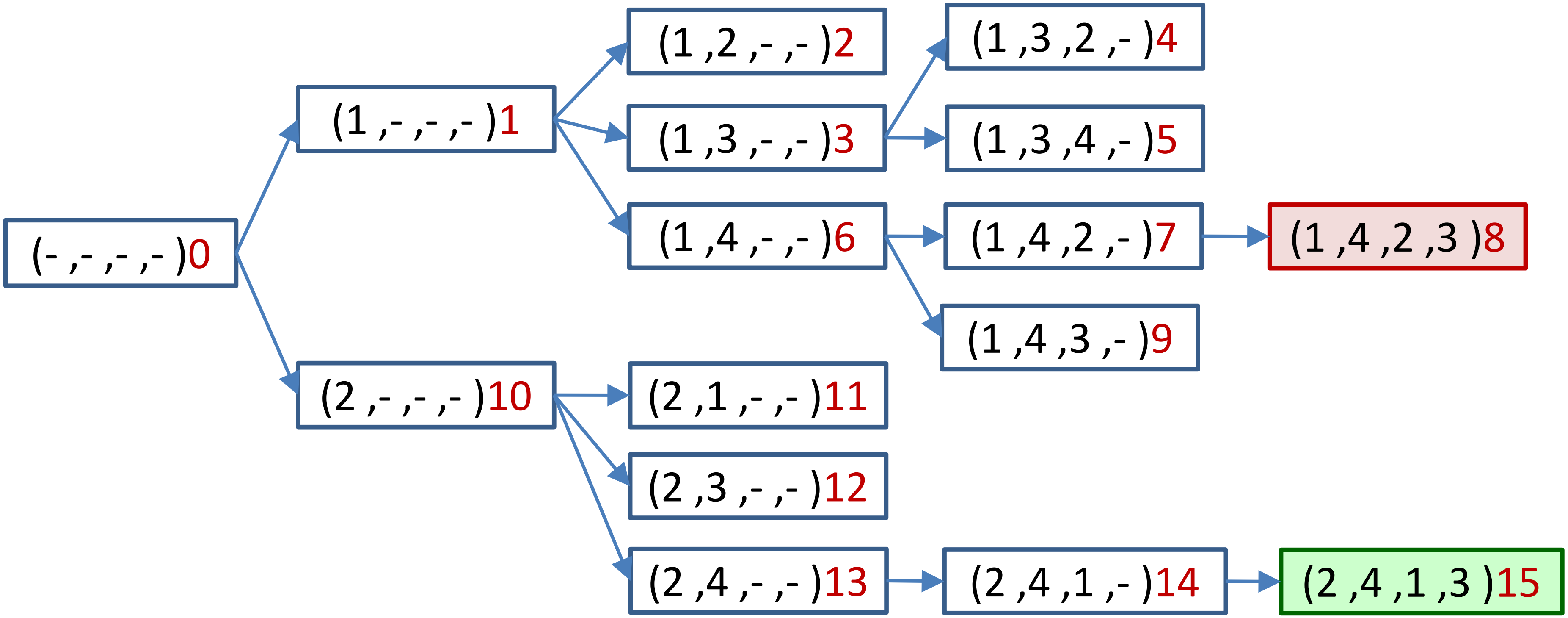
# EJEMPLO 1: Vuelta Atrás o Backtracking

## Problema de las N reinas

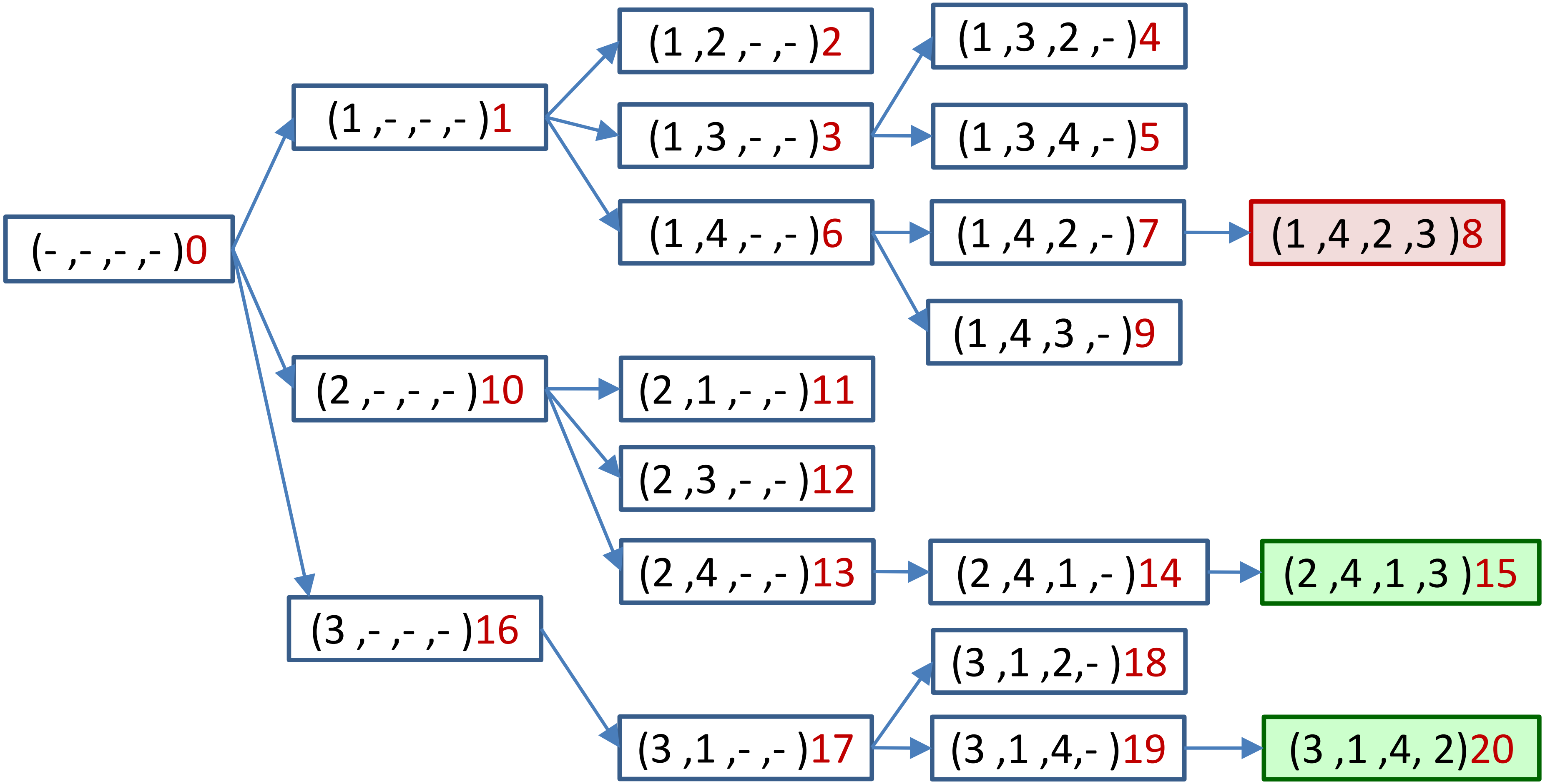


# **EJEMPLO 1:** Vuelta Atrás o Backtracking

## Problema de las N reinas

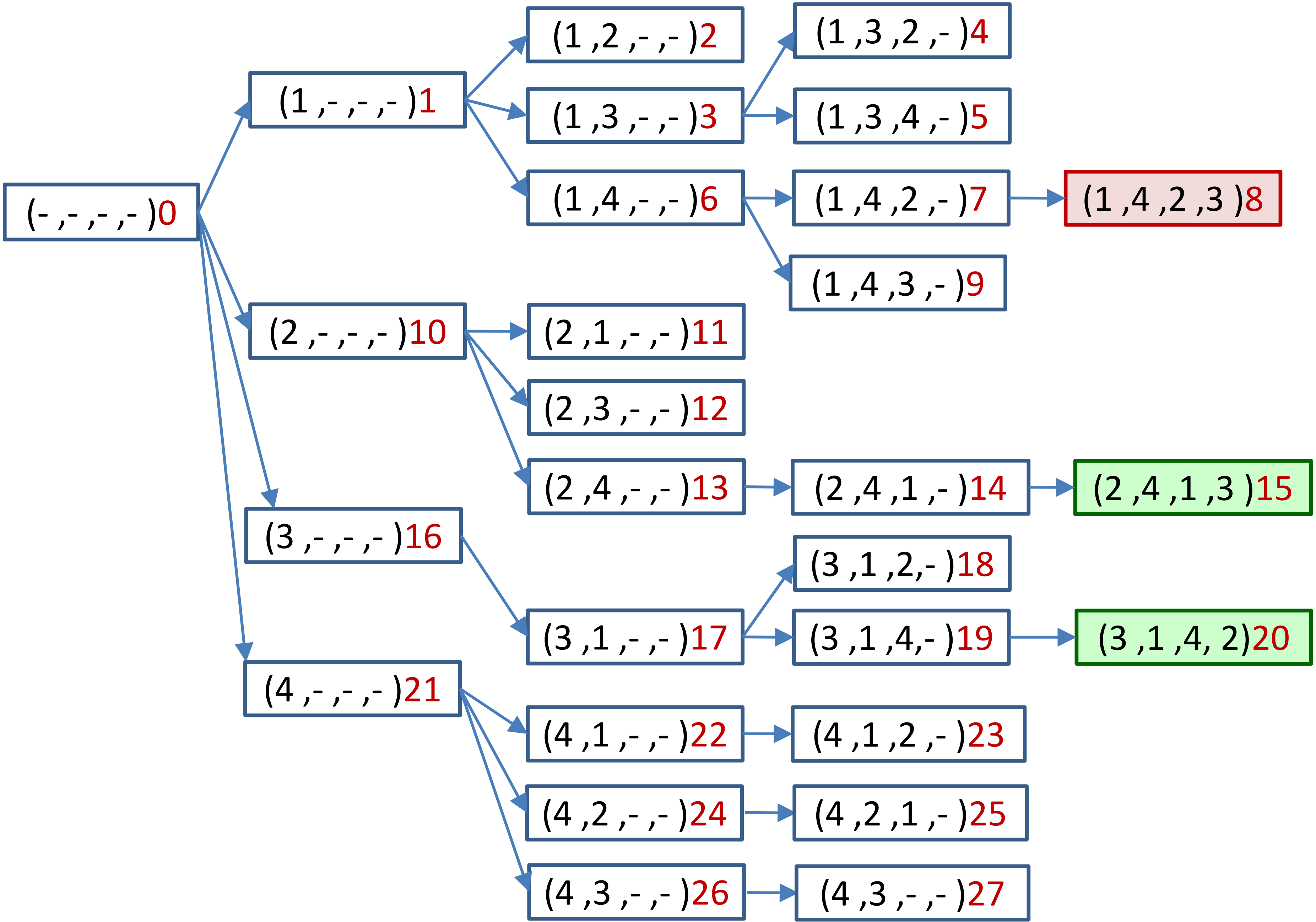


# EJEMPLO 1: Vuelta Atrás o Backtracking
## Problema de las N reinas



# EJEMPLO 1: Vuelta Atrás o Backtracking

## Problema de las N reinas



```
// Calcula las soluciones
void colocarReinasTodasSolucionesREC(int numReinas,
                                     array<int,REINAS> &tablero){
    if (numReinas==REINAS){
        mostrarSolucion(tablero);
    }
    else{
        for (tablero.at(numReinas)=0; tablero.at(numReinas)<REINAS;
                                      tablero.at(numReinas)++){
            if (esValido(numReinas, tablero)){
                colocarReinasTodasSolucionesREC(numReinas+1, tablero);
            }
        }
    }
}
```

# EJEMPLO 2: Vuelta Atrás o Backtracking

## Problema de las N reinas

```
// Función que genera las posibles soluciones del problema.
// Proceso recursivo
bool solucionNReinas(vector<vector<int>> &tablero, int col) {
    // Caso base, cuando la columna sea igual que la dimensión
    if (col == N) {
        mostrarSolucion(tablero);
        return true;
    }
    bool res = false;
    for (int i{0}; i < N; ++i) {
        if (esSolucionFactible(tablero, i, col)) {
            tablero.at(i).at(col) = 1;
            res = solucionNReinas(tablero, col + 1) || res;
            tablero.at(i).at(col) = 0;
        }
    }
    return res;
}
```

UNIVERSIDAD NEBRIJA