TEORÍA DE ALGORITMOS

Tema 8. Diseño de Algoritmos: Backtracking

Universidad de Granada

Curso 2010-2011

Motivación de la técnica

Hay problemas para los que

- NO se **conoce un algoritmo** para su resolución
- o al menos, NO cuentan con un **algoritmo eficiente** para calcular su solución

en estos casos, la único posibilidad es una **exploración directa** de todas las posibilidades

Por ejemplo, **el ajedrez**

- se conjetura que con blancas siempre se puede ganar
- no se conoce un modo eficiente de encontrar las jugadas

Motivación de la técnica

Por ejemplo, el ***n*-puzle** $\quad n = t^2 - 1$

2	3	8
0	1	7
6	5	4

$$\downarrow$$

0	1	2
3	4	5
6	7	8

- existen metodos de búsqueda efectivos para n pequeño
- para n bastante grande no hay un método eficiente

Descripción de la técnica

*La técnica Backtracking es un método de **búsqueda** de soluciones **exhaustiva** sobre grafos dirigidos acíclicos, el cual se acelera mediante **poda** de ramas poco prometedoras.*

Esto es,

- se representan todas las posibilidades en un árbol
- se resuelve buscando la solución por el árbol (de una determinada manera)
- hay zonas que se evitan por no contener soluciones (poda)

Descripción de la técnica

- la solución del problema se representa en una n-tupla $(X_1, X_2, \ldots, X_n)$ (no llenando necesariamente todas las componentes)
- cada X_i se escoje de un conjunto de **candidatos**
- a cada n-tupla se le llama **estado**
- se trata de buscar estados solución del problema

condiciones de parada

- cuando se consiga un estado solución
- cuando se consigan todos los estados solución

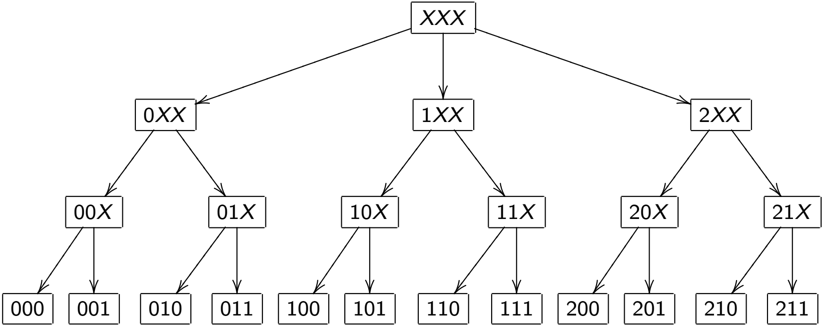
Elementos de la técnica

Al diseñar un algoritmo backtraking debemos considerar los siguientes elementos:

- Representación de la **solución** en una tupla $(X_1, \ldots, X_n)$
- Una **función objetivo** para determinar si la tupla a analizar es una solución
- Unas **restricciones** a los candidatos para rellenar la tupla:
  - **Implícitas** del problema. Valores que puede tomar cada valor X_i
  - **Explícitas** o externas al problema. Por ejemplo, problema mochila, el peso no debe superar la capacidad de la mochila
- Una **función de poda** para eliminar partes del árbol de búsqueda
- Organización del problema en un **árbol de búsqueda**

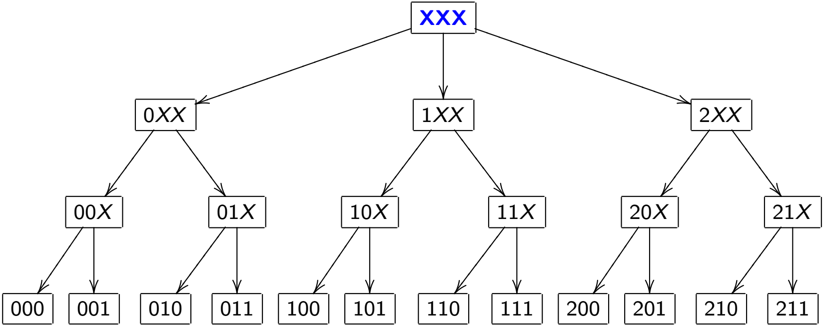
- Construir la solución al problema en distintas etapas.
- En cada paso se elige un candidato y se añade a la solución, y se avanza en la solución parcial.
- Si no es posible continuar en la construcción hacia una solución completa, se abandona ésta y la última componente se cambia por otro valor.
- Si no quedan más valores por probar, se retrocede al candidato anterior, se desecha, y se selecciona otro candidato.

Ejemplo para una terna, $X_1 \in \{0,1,2\}$, $X_2 \in \{0,1\}$, $X_3 \in \{0,1\}$

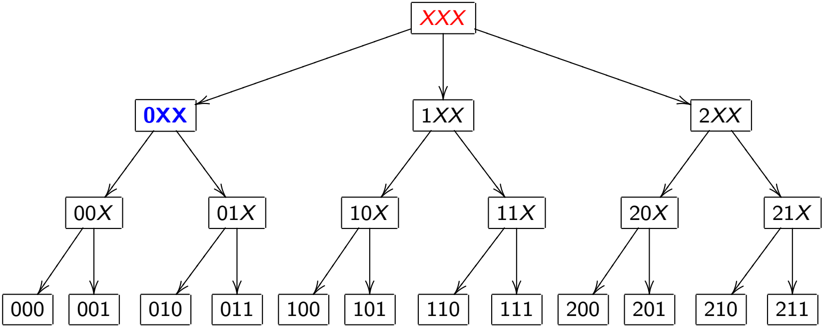


Búsqueda en el árbol de estados

Ejemplo para una terna, $X_1 \in \{0, 1, 2\}$, $X_2 \in \{0, 1\}$, $X_3 \in \{0, 1\}$

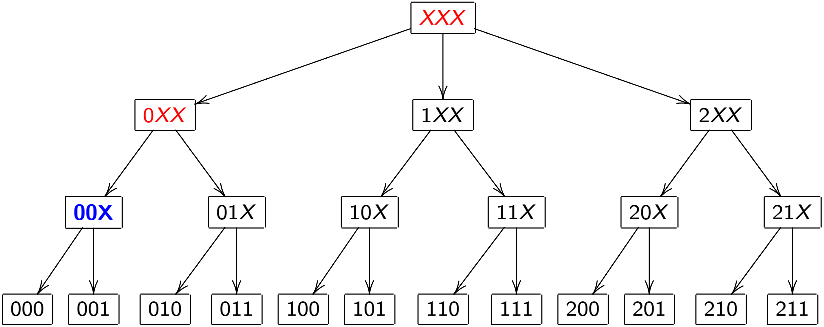


Ejemplo para una terna, $X_1 \in \{0, 1, 2\}$, $X_2 \in \{0, 1\}$, $X_3 \in \{0, 1\}$

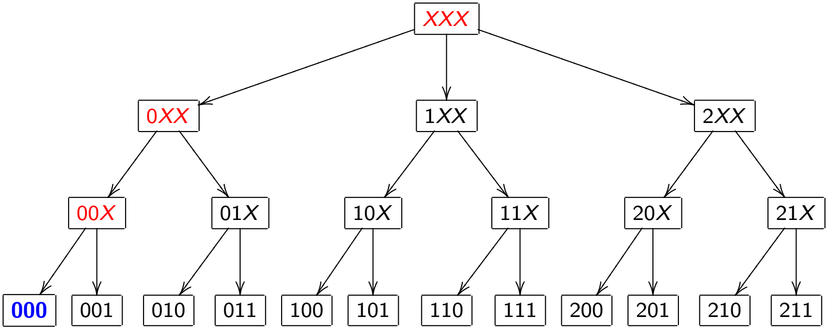


Búsqueda en el árbol de estados

Ejemplo para una terna, $X_1 \in \{0,1,2\}$, $X_2 \in \{0,1\}$, $X_3 \in \{0,1\}$

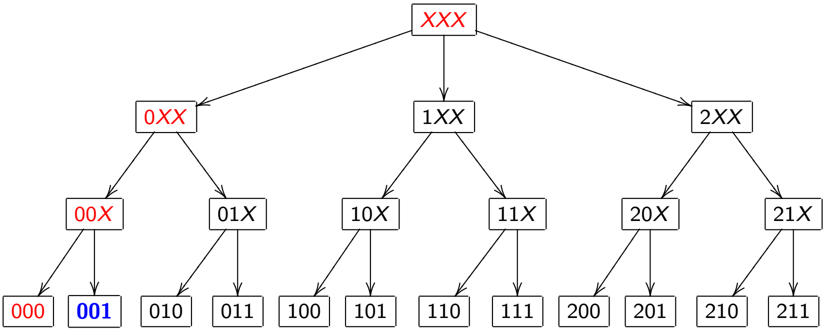


Ejemplo para una terna, $X_1 \in \{0,1,2\}$, $X_2 \in \{0,1\}$, $X_3 \in \{0,1\}$

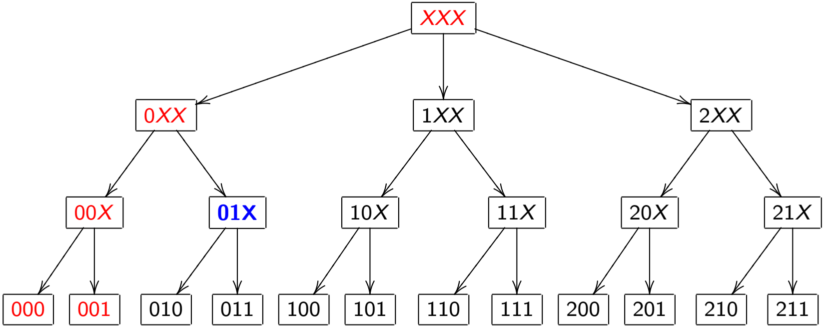


Búsqueda en el árbol de estados

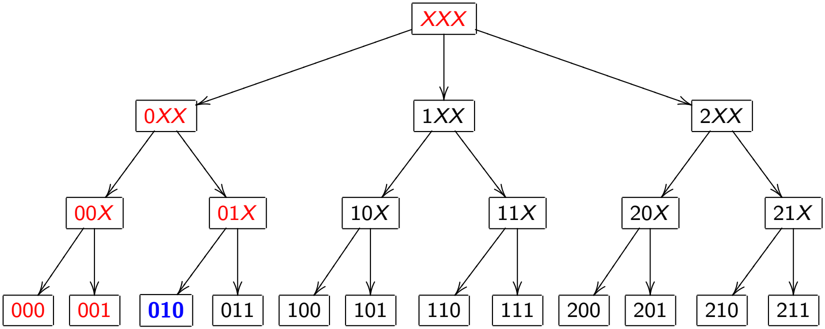
Ejemplo para una terna, $X_1 \in \{0, 1, 2\}$, $X_2 \in \{0, 1\}$, $X_3 \in \{0, 1\}$



Ejemplo para una terna, $X_1 \in \{0,1,2\}$, $X_2 \in \{0,1\}$, $X_3 \in \{0,1\}$

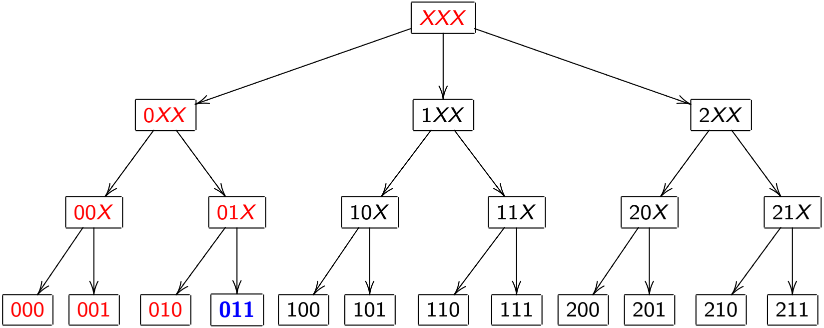


Ejemplo para una terna, $X_1 \in \{0, 1, 2\}$, $X_2 \in \{0, 1\}$, $X_3 \in \{0, 1\}$



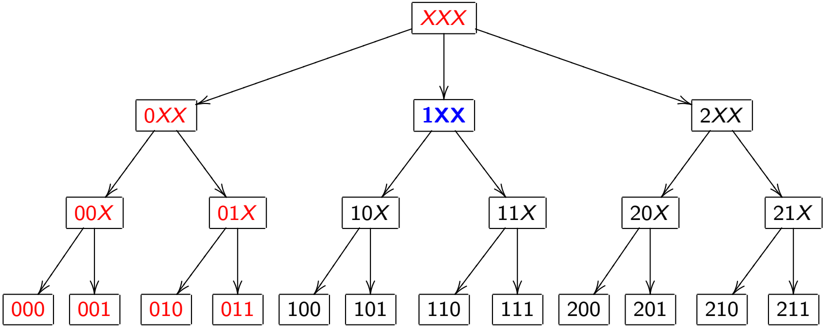
Búsqueda en el árbol de estados

Ejemplo para una terna, $X_1 \in \{0,1,2\}$, $X_2 \in \{0,1\}$, $X_3 \in \{0,1\}$



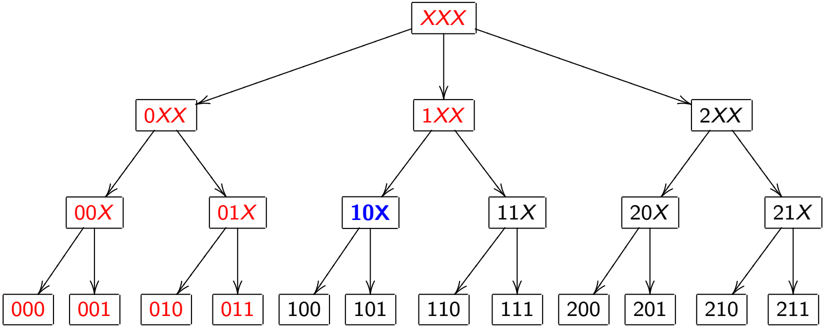
Búsqueda en el árbol de estados

Ejemplo para una terna, $X_1 \in \{0,1,2\}$, $X_2 \in \{0,1\}$, $X_3 \in \{0,1\}$

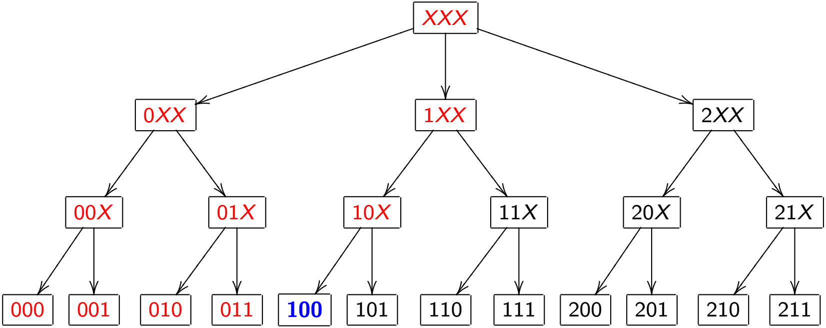


Búsqueda en el árbol de estados

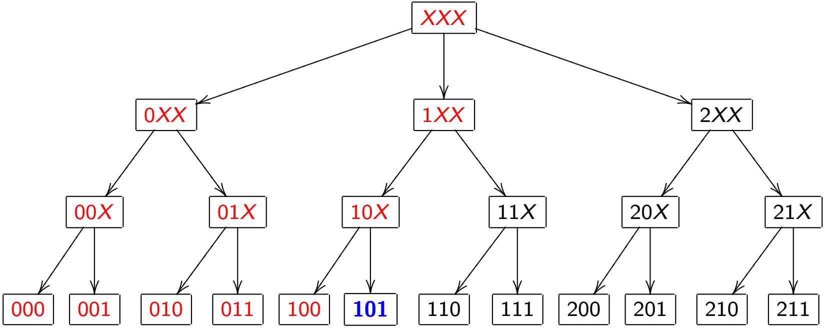
Ejemplo para una terna, $X_1 \in \{0,1,2\}$, $X_2 \in \{0,1\}$, $X_3 \in \{0,1\}$



Ejemplo para una terna, $X_1 \in \{0, 1, 2\}$, $X_2 \in \{0, 1\}$, $X_3 \in \{0, 1\}$

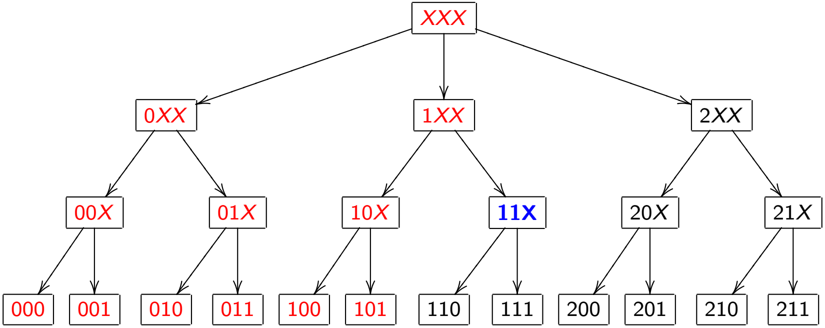


Ejemplo para una terna, $X_1 \in \{0,1,2\}$, $X_2 \in \{0,1\}$, $X_3 \in \{0,1\}$

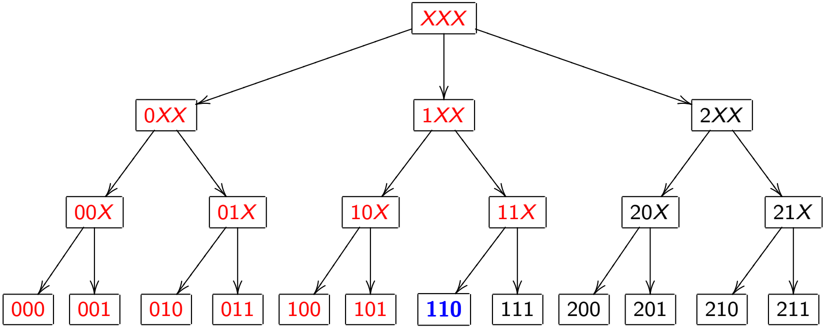


Búsqueda en el árbol de estados

Ejemplo para una terna, $X_1 \in \{0, 1, 2\}$, $X_2 \in \{0, 1\}$, $X_3 \in \{0, 1\}$

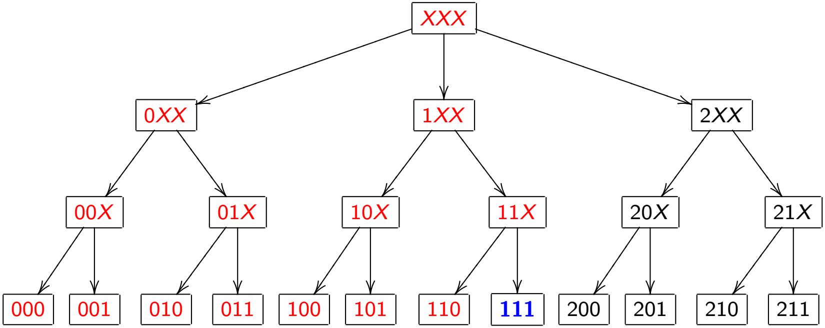


Ejemplo para una terna, $X_1 \in \{0,1,2\}$, $X_2 \in \{0,1\}$, $X_3 \in \{0,1\}$

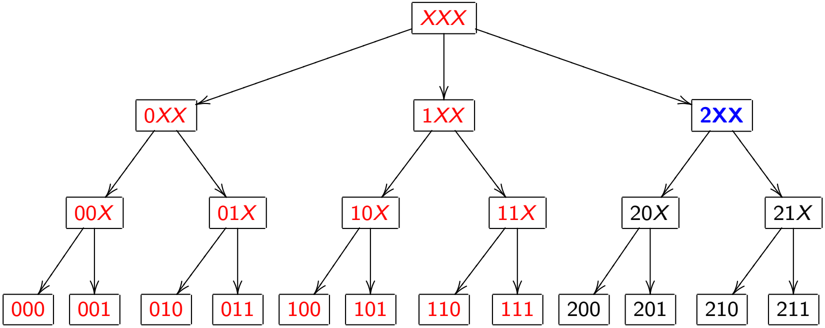


Búsqueda en el árbol de estados

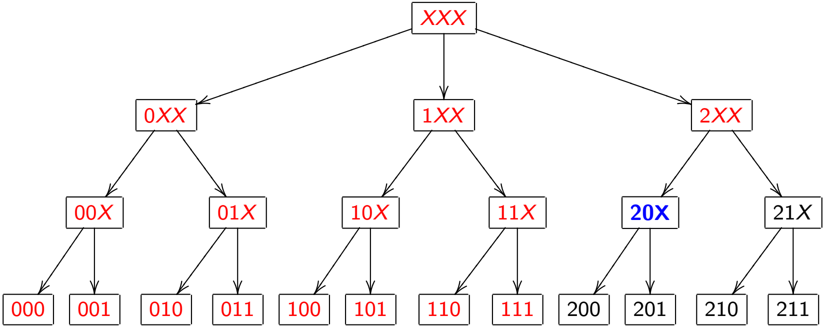
Ejemplo para una terna, $X_1 \in \{0, 1, 2\}$, $X_2 \in \{0, 1\}$, $X_3 \in \{0, 1\}$



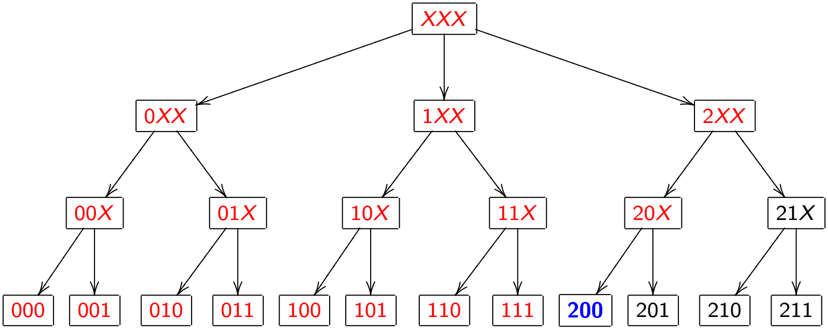
Ejemplo para una terna, $X_1 \in \{0,1,2\}$, $X_2 \in \{0,1\}$, $X_3 \in \{0,1\}$



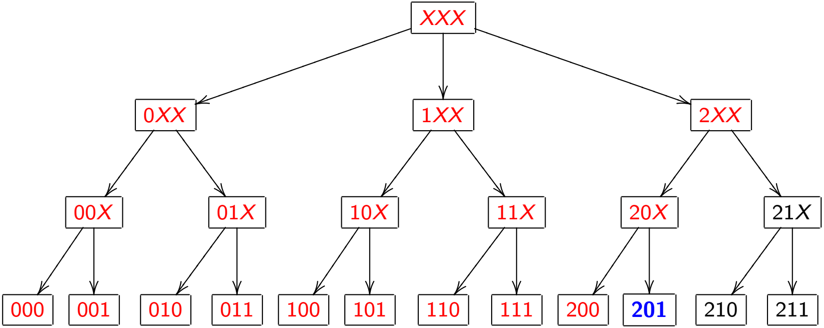
Ejemplo para una terna, $X_1 \in \{0,1,2\}$, $X_2 \in \{0,1\}$, $X_3 \in \{0,1\}$



Ejemplo para una terna, $X_1 \in \{0,1,2\}$, $X_2 \in \{0,1\}$, $X_3 \in \{0,1\}$

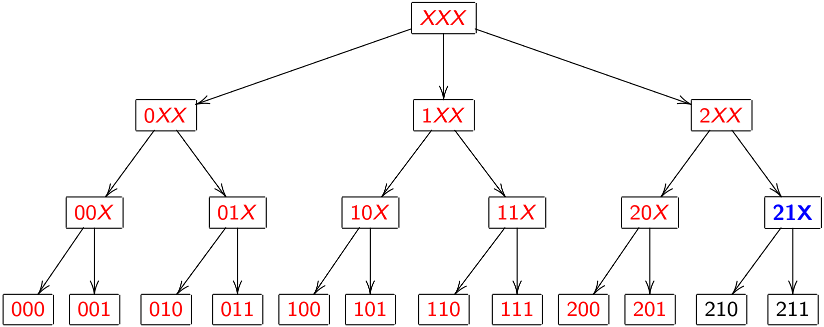


Ejemplo para una terna, $X_1 \in \{0, 1, 2\}$, $X_2 \in \{0, 1\}$, $X_3 \in \{0, 1\}$



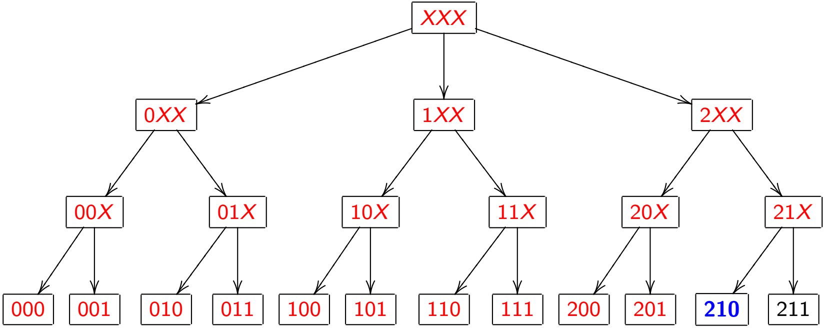
Búsqueda en el árbol de estados

Ejemplo para una terna, $X_1 \in \{0, 1, 2\}$, $X_2 \in \{0, 1\}$, $X_3 \in \{0, 1\}$

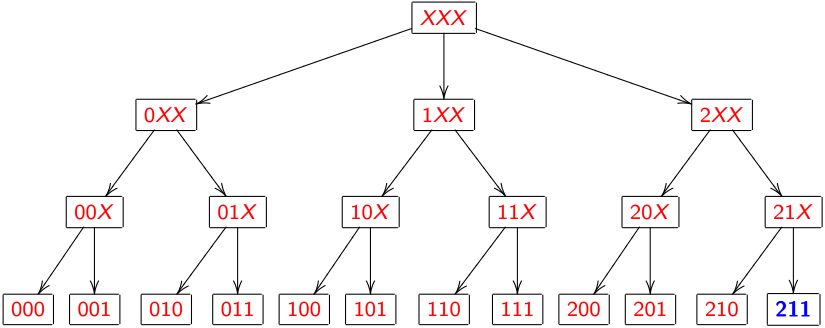


Búsqueda en el árbol de estados

Ejemplo para una terna, $X_1 \in \{0, 1, 2\}$, $X_2 \in \{0, 1\}$, $X_3 \in \{0, 1\}$



Ejemplo para una terna, $X_1 \in \{0, 1, 2\}$, $X_2 \in \{0, 1\}$, $X_3 \in \{0, 1\}$



Diseño de la técnica

Para diseñar un algoritmo con la técnica backtracking, debemos seguir los siguientes pasos:

- **Buscar una representación** del tipo $(X_1, X_2, \ldots, X_n)$ para las soluciones del problema
- **Identificar las restricciones** implícitas y explícitas del problema
- **Establecer la organización del árbol** que define los diferentes estados en los que se encuentra una (sub)solución
- Definir una **función solución** para determinar si una tupla es solución
- **Definir una función de poda** $B_k(X_1, X_2, \ldots, X_k)$ para eliminar ramas del árbol que puedan derivar en soluciones poco deseables o inadeciadas
- Aplicar la estructura genérica de un algoritmo backtracking

solucion[i]$\in$S$_i$ para i=1,2,...,n

(Algoritmo genérico backtracking)

funcion BACKTRACKING_REC (k , solucion[n])
    para j$\in$S$_i$
        si (PODA (k , j , solucion) == true) ***hacer***
            sol[k]= j
            si (TEST_SOL (solucion) == true) ***hacer***
                devolver solucion
            si (k < n)
                BACKTRACKING_REC(k+1,solucion[n])

Eficiencia

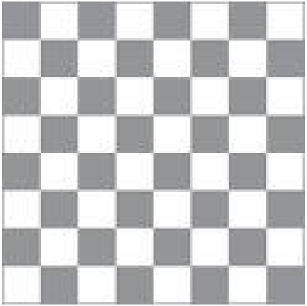
La eficiencia en un algoritmo backtracking suele ser de **tipo exponencial** a^n

- depende de la ramificación del árbol
- del tiempo de ejecución de la función solución
- del tiempo de ejecución de la función poda
- del ahorro de utilizar la poda

Nota: las buenas funciones de poda no son muy eficientes

Ejemplo: el problema de las n reinas

Supongamos que tenemos un tablero de ajedrez:

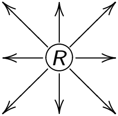


¿Podemos colocar 8 reinas sin que se amenacen?

Ejemplo: el problema de las *n* reinas

Recordemos que las reinas se mueven por el tablero

- cualquier número de casillas en horizontal
- cualquier número de casillas en vertical
- cualquier número de casillas en diagonal

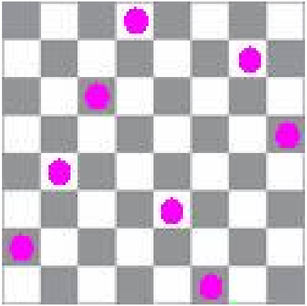


entonces **no** puede haber dos reinas

- en la **misma fila**
- en la **misma columna**
- en la **misma diagonal**

Ejemplo: el problema de las n reinas

Una solución al problema



(Problema)

Dado un tablero (cuadrado) con n casillas de lado, ¿podemos colocar n reinas en el tablero sin que se amenacen?

Ejemplo: el problema de las n reinas

Elementos de la técnica backtracking:

- **Representación del problema**. En n-tuplas $(x_1, x_2, \ldots, x_n)$, donde x_i es la fila donde está la reina de la columna i
- **Restricciones implicitas**. Las componentes $x_i \in \{1, 2, \ldots, n\}$
- **Restricciones explicitas**. No puede haber dos reinas en la misma fila, columna y diagonal
- **Árbol de estados**. En el nivel i se obtiene la posición de la reina i
- **Función objetivo**. La n-tupla está completa y cumple las restricciones.
- **Función poda**. Dada por las restricciones explicitas

`k`= columna (la reina de la columna k)

(Función de poda)

```
funcion PODA ( k , sol[n] )
    para (j=1) hasta (j=k-1)
        si (sol[j]==sol[k]) //misma fila
            devolver false
        si (sol[j]-sol[k]==j-k) //misma diagonal
            devolver false
        si (sol[j]-sol[k]==k-j) //misma diagonal
            devolver false

    devolver true
```

(Problema de las *n* reinas)

```
funcion REINAS ( k, n , solucion[n] )
    para (i=1) hasta (i=n)
        solucion[k]=i
        si ( PODA (k, solucion)== true )
            si ( k==n )
                devolver solucion
            sino
                REINAS ( k+1, n , solucion[n] )
```

La solución vendrá de `REINAS (1, n, solucion[n])`

(Ejercicio)

Implementar en C++ el problema de las n reinas y utilizarlo para calcular la solución de las 4, 8, 16, 32 y 64 reinas.

Ejemplo: el problema del *n*-puzle

Supongamos una cuadricula 3x3 con números del 0 al 8 (sin repetir)

2	3	8
0	1	7
6	5	4

¿Podemos llevar esta configuración a una ordenada?

0	1	2
3	4	5
6	7	8

Existen cuatro movimientos permitidos

1. Intercambiar el cero con la casilla superior

$$\begin{array}{|c|c|c|}\hline \mathbf{2} & 3 & 8 \\ \hline \mathbf{0} & 1 & 7 \\ \hline 6 & 5 & 4 \\ \hline \end{array} \longrightarrow \begin{array}{|c|c|c|}\hline \mathbf{0} & 3 & 8 \\ \hline \mathbf{2} & 1 & 7 \\ \hline 6 & 5 & 4 \\ \hline \end{array}$$

2. Intercambiar el cero con la casilla inferior

$$\begin{array}{|c|c|c|}\hline 2 & 3 & 8 \\ \hline \mathbf{0} & 1 & 7 \\ \hline \mathbf{6} & 5 & 4 \\ \hline \end{array} \longrightarrow \begin{array}{|c|c|c|}\hline 2 & 3 & 8 \\ \hline \mathbf{6} & 1 & 7 \\ \hline \mathbf{0} & 5 & 4 \\ \hline \end{array}$$

Existen cuatro movimientos permitidos

3. Intercambiar el cero con la casilla situada a la derecha

$$\begin{array}{|c|c|c|}\hline 2 & 3 & 8 \\ \hline \mathbf{0} & \mathbf{1} & 7 \\ \hline 6 & 5 & 4 \\ \hline \end{array} \longrightarrow \begin{array}{|c|c|c|}\hline 2 & 3 & 8 \\ \hline \mathbf{1} & \mathbf{0} & 7 \\ \hline 6 & 5 & 4 \\ \hline \end{array}$$

4. Intercambiar el cero con la casilla situada a la izquierda

$$\begin{array}{|c|c|c|}\hline 2 & 3 & 8 \\ \hline 7 & \mathbf{1} & \mathbf{0} \\ \hline 6 & 5 & 4 \\ \hline \end{array} \longrightarrow \begin{array}{|c|c|c|}\hline 0 & 3 & 8 \\ \hline 7 & \mathbf{0} & \mathbf{1} \\ \hline 6 & 5 & 4 \\ \hline \end{array}$$

Ejemplo: el problema del *n*-puzle

Utilizando estos movimientos no siempre es posible llegar a

0	1	2
3	4	5
6	7	8

(Teorema)

Dada una configuración del 8-puzle, siempre podemos llegar a

0	1	2
3	4	5
6	7	8

o bien, a

0	3	6
1	4	7
2	5	8

En general, podemos plantearnos el problema del n-puzle, donde $n = t^2 - 1$ *para* t *un entero que dos*

Ejemplo: el problema del n-puzle

Elementos de la técnica backtraking:
1=izquierda, 2=derecha, 3=arriba, 4=abajo

- **Representación del problema**. En n-tuplas $(x_1, x_2, \ldots, x_n)$, donde x_i es el movimiento i-ésimo.
- **Restricciones implicitas**. Las componentes $x_i \in \{1, 2, 3, 4\}$
- **Restricciones explicitas**. No hay
- **Árbol de estados**. En el nivel i se obtiene el movimiento i-ésimo
- **Función objetivo**. Determina si con los movimientos realizados la matriz es igual a una de las matrices objetivo.
- **Función poda**. Poda por profundidad, sólo se permiten cierto número de movimientos

`k`= movimiento actual (`k`-ésimo)
`n`= número máximo de movimientos

(Problema del *n*-puzle)

```
funcion PUZLE ( k , solucion[n] )
    si ( TEST_SOLUCION ( solucion )== true )
        devolver solucion

    si (k < n) // funcion de poda
        para (i=1) hasta (i=4)
            solucion[k] = i
            PUZLE (k+1, solucion )

    devolver "No encuentro solucion"
```

La solucion viene dada por `PUZLE (1, {0,0,...,0} )`

Ejemplo: el problema de la mochila

Supongamos la siguiente situación:

1. tenemos una mochila cuyo peso máximo de carga es M
2. una serie de objetos a transportar $1, 2, \cdots, n$ donde:
   - el objeto i tiene un peso p_i
   - el objeto i tiene un valor v_i

(Pregunta)

¿Cómo podemos llenar la mochila (respetando el límite de peso) para maximizar el valor de la carga ?

Dos variantes:

1. **Variante A**. Los objetos son **indivisibles**, esto es, no se pueden romper para meter un trozo en la mochila.
2. **Variante B**. Los objetos **se pueden dividir** en partes más pequeñas, con peso y valor proporcional al original.

para la técnica backtraking, consideramos la **Variante A**

Ejemplo: problema de la mochila

Elementos del algoritmo:

- Cada **tupla solución** `solucion[n]`, con valores $\{0, 1\}$ en cada componente. Indican si se lleva el objeto `i` en cada caso.
- **Organización del árbol de estados**. En cada nivel i del árbol seleccionamos (o no) llevar el objeto i.
- **Función objetivo**. Encontrar alguna solución óptima al problema de la mochila.
- **Restricciones implícitas**. Cada elemento de la tupla solución sólo podrá tener los valores 0 ó 1.
- **Restricciones explícitas**. La suma del peso de todos los objetos no debe ser superior a M.
- **Función de Poda**. Un nodo no será explorado si el peso del objeto asociado a tal nodo, añadido a la mochila, supera la capacidad máxima de ésta.

Ejemplo: problema de la mochila

```
funcion BENEFICIO ( k , solucion[n], valor[n] )
    suma=0;
    para (i=1) hasta (i=k)
        suma = suma+valor[i]*solucion[i]
    devolver suma
```

(Función de Poda)

```
funcion PODA ( k , solucion[n], peso[n] )
    si (BENEFICIO (k, solucion, peso) > M )
        devolver false
    devolver true
```

Ejemplo: problema de la mochila

(Problema de la mochila)

```
funcion MOCHILA (k,sol[n],peso[n],valor[n],valor_max)
    solucion[k]= 0
    si (k < n)
        MOCHILA (k+1,sol,peso,valor,valor_max)

    solucion[k]= 1
    si (PODA (k,sol,peso) == true)
        si ( BENEFICIO(k,sol,valor) > valor_max)
            valor_max=BENEFICIO(k,sol,valor)
        si (k < n)
            MOCHILA (k+1,sol,peso,valor,valor_max)
    devolver valor_max
```

Ejemplo: suma de subconjuntos

Supongamos que tenemos:

- un conjunto de enteros no negativos $X = \{x_1, \ldots, x_n\}$
- un entero M

(Problema)

Calcular qué subconjuntos de X suman exactamente M

Ejemplo: suma de subconjuntos

(Ejemplo)

Supongamos que $X = \{2, 3, 5, 10, 20\}$ *y* $M = 15$*, entonces existen dos soluciones posibles:*

- $x_1 = 2$, $x_2 = 3$ *y* $x_4 = 10$
- $x_3 = 5$ *y* $x_4 = 10$

Ejemplo: suma de subconjuntos

Elementos del algoritmo:

- Cada **tupla solución** `solucion[n]`, con valores $\{0, 1\}$ en cada componente. Indican si se considera el entero `i` en cada caso.
- **Organización del árbol de estados**. En cada nivel i del árbol seleccionamos (o no) el entero i.
- **Función objetivo**. Determina si la suma de los enteros seleccionados es M.
- **Restricciones implícitas**. Cada elemento de la tupla solución sólo podrá tener los valores 0 ó 1.
- **Restricciones explícitas**. La suma de los enteros seleccionados es M.

Ejemplo: suma de subconjuntos

Supuesto que ordenamos los enteros de menor a mayor:

- **Función de Poda**. La función de poda tendrá en cuenta dos condiciones. Si estamos en la etapa k y
  - $\sum_{i=1}^{k} v_i x_i + \sum_{i=k+1}^{n} v_i < M$ entonces podamos la rama (aún considerando todos los enteros restantes no llegamos a M)
  - $\sum_{i=1}^{k} v_i x_i + v_{k+1} > M$ entonces podamos la rama (nos pasamos con el entero restante más pequeño)

Ejemplo: suma de subconjuntos

(Función de poda)

```
funcion PODA ( k , solucion[n], enteros[n] )
    suma=0; suma2=0
    para (i=0) hasta (i=k-1)
        suma=suma + enteros[i]*solucion[i]

    para (i=k) hasta (i=n)
        suma2=suma2 + enteros[i]

    si ( suma + enteros[k] > M )
        devolver false
    si ( suma + suma2  < M )
        devolver false

    devolver true
```

Ejemplo: suma de subconjuntos

```
solucion[][n] // donde acumular las soluciones
j=0
```

(suma de subconjuntos)

```
funcion SUMA ( k , eleccion[n] , enteros[n] )
    eleccion[k]=0
    si ( k < n  && PODA (k, eleccion, enteros)==true )
        SUMA ( k+1 , eleccion , enteros )
    eleccion[k]=1
    si ( TEST_SOLUCION ( eleccion )== true )
        solucion[j] = eleccion
        j=j+1
    si ( k < n && PODA (k, eleccion, enteros)==true )
        SUMA ( k+1 , eleccion , enteros )
    devolver solucion
```

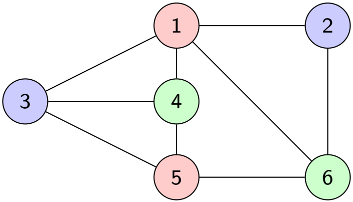
Ejemplo: colorear un grafo

(Problema)

Dado un grafo ***no dirigido****, ¿cuál es el número mínimo de colores que tenemos que utilizar para colorearlo de manera que dos vértices adyacentes no compartan el mismo color?*

(Problema)

Dado un grafo no dirigido, ¿cuál es su número cromático?



Ejemplo: colorear un grafo

Elementos del algoritmo:
Para un grafo con n vértices y k colores

- Cada **tupla solución**, con valores $\{1, 2, \ldots, k\}$ en cada componente. Indican con qué color está coloreado cada vértices.
- **Organización del árbol de estados**. En cada nivel i del árbol coloreamos el vértice i.
- **Función objetivo**. Determina el número mínimo de colores.
- **Restricciones implícitas**. Cada elemento de la tupla toma valores en $\{1, 2, \ldots, k\}$.
- **Restricciones explícitas**. No puede haber dos vértices adyacentes con el mismo valor.
- **Función de poda**. Determina si hay dos vértices adyacentes con el mismo color

Ejemplo: colorear un grafo

Implementar en C++ el algoritmo backtraking para resolver el problema de colorear un grafo