Teoría de Algoritmos

Capitulo 5: Algoritmos para la Exploración de Grafos.

Tema 14: Backtracking y Branch and Bound

- Problema de las 8 reinas
- Problema de la suma de subconjuntos
- Coloreo de grafos
- Laberintos
- Ciclos hamiltonianos

Solución para las 8 reinas

- Generalizamos el problema para considerar un tablero nxn y encontrar todas las formas de colocar n reinas que no se ataquen.
- Podemos tomar $(x_1,\ldots,x_n)$ representando una solución si x_i es la columna de la i-esima fila en la que la reina i esta colocada.
- Los x_i's serán todos distintos ya que no puede haber dos reinas en la misma columna.
- ¿Como comprobar que dos reinas no estén en la misma diagonal?

			(1,4)				
(2,1)		(2,3)					
	(3,2)						
(4,1)		(4,3)					
			(5,4)				(5,8)
(6,1)				(6,5)		(6,7)	
	(7,2)				(7,6)		
		(8,3)		(8,5)		(8,7)	

Solución para las 8 reinas

- Si las casillas del tablero se numeran como una matriz A(1..n,1..n), cada elemento en la misma diagonal que vaya de la parte superior izquierda a la inferior derecha, tiene el mismo valor "fila-columna".
- También, cualquier elemento en la misma diagonal que vaya de la parte superior derecha a la inferior izquierda, tiene el mismo valor "fila+columna".
- Si dos reinas están colocadas en las posiciones (i,j) y (k,l), estarán en la misma diagonal solo si,

$$i - j = k - l \text{ ó } i + j = k + l$$

- La primera ecuación implica que

$$j - l = i - k$$

- La segunda que

$$j - l = k - i$$

- Así, dos reinas están en la misma diagonal si y solo si

$$|j-l| = |i-k|$$

Solución para las 8 reinas

- El procedimiento COLOCA(k) devuelve verdad si la k-esima reina puede colocarse en el valor actual de X(k). Testea si X(k) es distinto de todos los valores previos X(1),...,X(k-1), y si hay alguna otra reina en la misma diagonal. Su tiempo de ejecución de O(k-1).

  Procedimiento COLOCA(K)

  {X es un array cuyos k primeros valores han sido ya asignados. ABS(r) da el valor absoluto de r}

```
Begin
  For i:=1 to k do
     If X(i) = X(k) or ABS(X(i)-X(k)) = ABS(i-k)
     Then return (false)
  Return (true)
end
```

Solución para las 8 reinas

Procedimiento NREINAS(N)

{Usando backtracking este procedimiento imprime todos los posibles emplazamientos de n reinas en un tablero nxn sin que se ataquen}

```
Begin
X(1) := 0, k := 1                              {k es la fila actual}
While k > 0 do                                 {hacer para todas las filas}
   X(k) := X(k) + 1                            {mover a la siguiente columna}
   While X(k) ≤ n and not COLOCA (k) do        {puede moverse esta reina?}
         X(k) := X(k) + 1
   If X(k) ≤ n                                 {Se encontró una posición}
      Then if k = n                            {Es una solución completa?}
              Then print (X)
              Else k := k + 1; X(k) := 0       {Ir a la siguiente fila}
      Else k := k - 1                          {Backtrack}
end
```

Solución para las 8 reinas

- Nótese que en un tablero 8x8 hay $C_{64,8}$ formas posibles de colocar 8 reinas, es decir 4.4 billones de 8-tuplas para examinar. Sin embargo, permitiendo solo emplazamientos de reinas en filas y columnas distintas, necesitamos examinar, a lo sumo, 8!, es decir, 40.320 8-tuplas.
- Para ver aplicaciones sobre ajedrez (Deep Blue, Kasparov, etc.)

  http://www.research.ibm.com/deepblue/home/html/clips.html (multimedia clip)

Solución para la suma de subconjuntos

- Tenemos n números positivos distintos (usualmente llamados pesos) y queremos encontrar todas las combinaciones de estos números que sumen M.
- Los anteriores ejemplos mostraron como podríamos formular este problema usando tamaños de las tuplas fijos o variables.
- Consideraremos una solución backtracking usando la estrategia del tamaño fijo de las tuplas.
- En este caso el elemento X(i) del vector solución es uno o cero, dependiendo de si el peso W(i) esta incluido o no.

Solución para la suma de subconjuntos

- Generación de los hijos de cualquier nodo en el árbol:
- Para un nodo en el nivel i, el hijo de la izquierda corresponde a $X(i) = 1$, y el de la derecha a $X(i) = 0$.
- Una posible elección de funciones de acotación es $B_k(X(1), \ldots, X(k))$ = true si y solo si,

$$\sum_{1..k} W(i)X(i) + \sum_{k+1..n} W(i) \geq M$$

- Claramente $X(1), \ldots, X(k)$ no pueden conducir a un nodo respuesta si no se verifica esta condición.

Solución para la suma de subconjuntos

- Las funciones de acotación pueden fortalecerse si suponemos los W(i)'s en orden creciente.
- En este caso, X(1),...,X(k) no pueden llevar a un nodo respuesta si

$$\sum_{1..k} W(i)X(i) + W(k+1) > M$$

- Por tanto las funciones de acotación que usaremos serán las definidas de la siguiente forma: B_k(X(1), ..., X(k)) es true si y solo si

$$\sum_{1..k} W(i)X(i) + \sum_{k+1..n} W(i) \geq M$$

y

$$\sum_{1..k} W(i)X(i) + W(k+1) \leq M$$

Solución para la suma de subconjuntos

- Ya que nuestro algoritmo no hará uso de B_n, no necesitamos preocuparnos por la posible aparición de W(n+1) en esta función.
- Aunque hasta aquí hemos especificado todo lo que es necesario para usar cualquiera de los esquemas Backtracking, resultaría un algoritmo mas simple si diseñamos a la medida del problema que estemos tratando cualquiera de esos esquemas .
- Esta simplificación resulta de la comprobación de que si X(k) = 1, entonces
  - $\sum_{1..k} W(i)X(i) + \sum_{k+1..n} W(i) > M$

Esquema de algoritmo recursivo

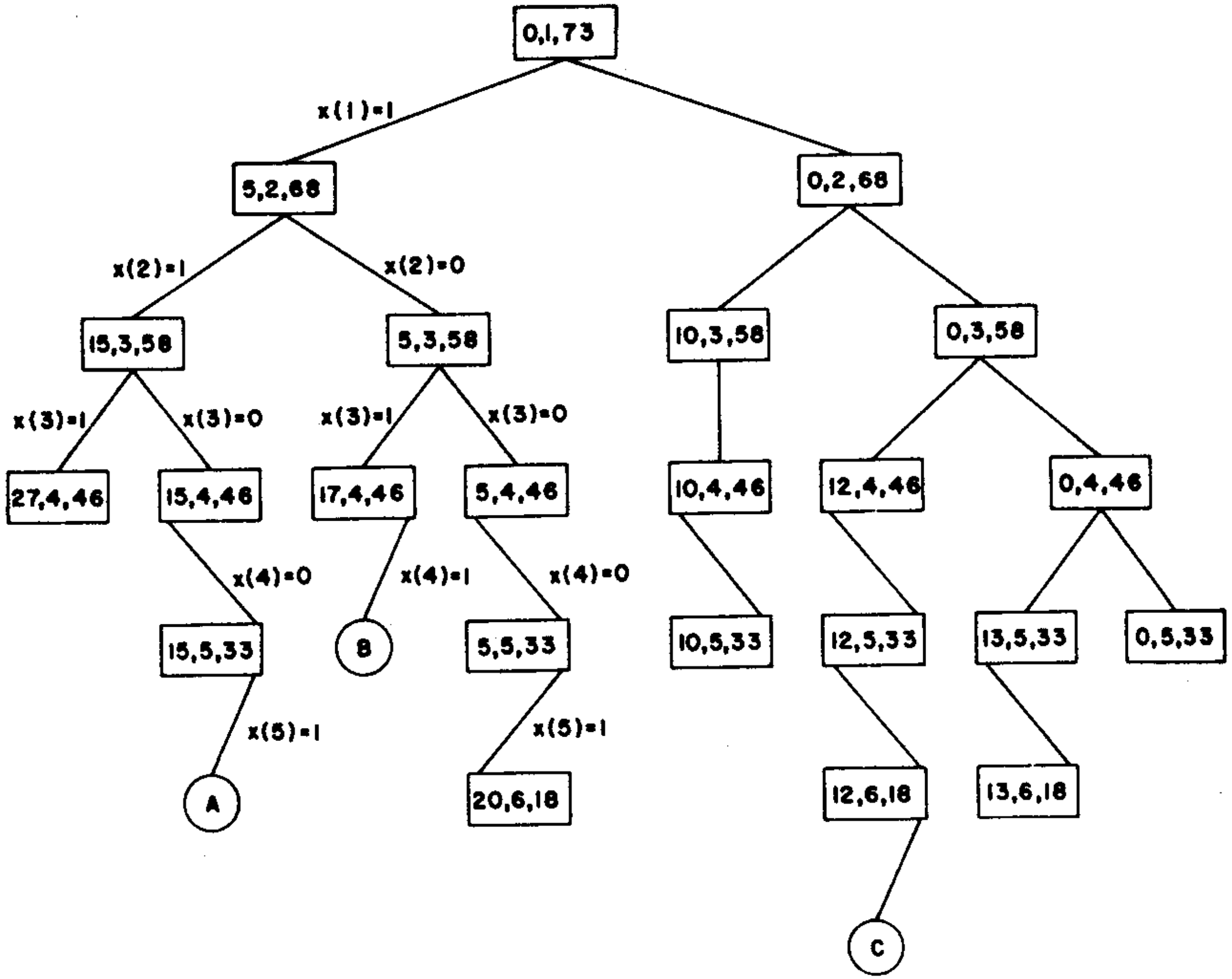
Procedimiento SUMASUB (s,k,r)

{Los valores de X(j), $1 \leq j < k$, ya han sido determinados. $s = \sum_{1..k-1} W(j)X(j)$ y $r = \sum_{k..n} W(j)$. Los W(j) están en orden creciente. Se supone que $W(1) \leq M$ y que $\sum_{1..n} W(i) \geq M$}

```
Begin
{Generación del hijo izquierdo. Nótese que s+W(k) ≤ M ya que B k-1 = true}
            X(k) = 1
{4}         If s + W(k) = M
{5}         Then For i = 1 to k print X(j)
            Else
{7}                   If s + W(k) + W(k+1) ≤ M
                      Then SUMASUB(s + W(k), k+1, r-W(k))
{Generación del hijo derecho y evaluación de B k}
            If s + r - W(k) ≥M and s + W(k+1) ≤ M
            Then X(k) = 0
            SUMASUB(S, K+1, R-w(K))
 end
```

Ejemplo

Como trabaja SUMASUB para el caso en que: W = (5, 10, 12, 13, 15, 18) y M = 30.



El problema del coloreo de un grafo

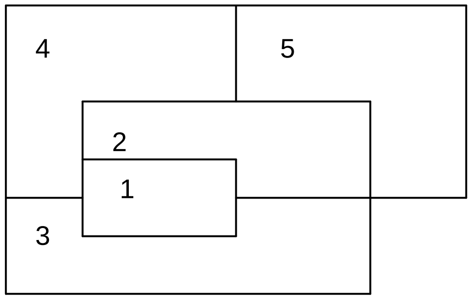
- Sea G un grafo y m un numero entero positivo. Queremos saber si los nodos de G pueden colorearse de tal forma que no haya dos vértices adyacentes que tengan el mismo color, y que solo se usen m colores para esa tarea.
- Este es el problema de la m-colorabilidad.
- El problema de optimización de la m-colorabilidad, pregunta por el menor numero m con el que el grafo G puede colorearse. A ese entero se le denomina Numero Cromático del grafo.

El problema del coloreo de un grafo

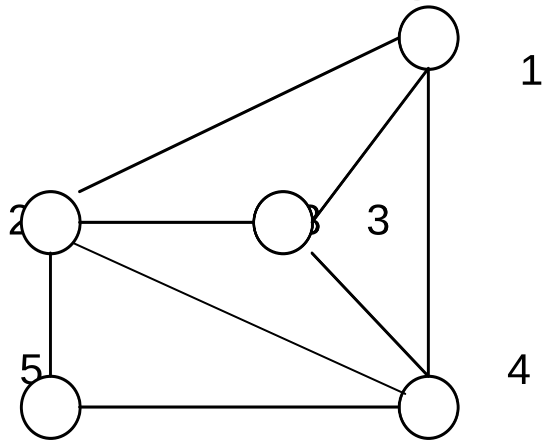
- Un grafo se llama plano si y solo si puede pintarse en un plano de modo que ningún par de aristas se corten entre si.
- Un caso especial famoso del problema de la m-colorabilidad es el problema de los cuatro colores para grafos planos que, dado un mapa cualquiera, consiste en saber si ese mapa podrá pintarse de manera que no haya dos zonas colindantes con el mismo color, y además pueda hacerse ese coloreo solo con cuatro colores.
- Este problema es fácilmente traducible a la nomenclatura de grafos

El problema del coloreo de un grafo

- El mapa



- puede traducirse en el siguiente grafo

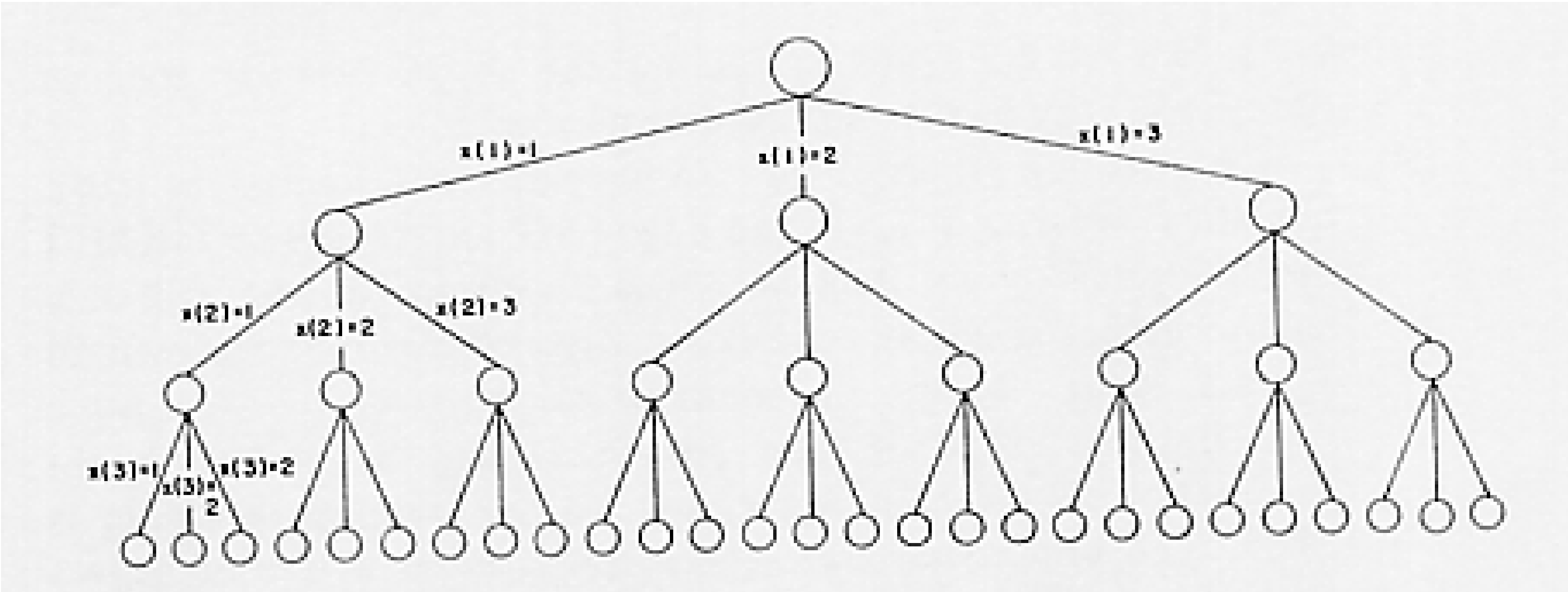


El problema del coloreo de un grafo

- Representamos el grafo por su matriz de adyacencia GRAFO(1:n, 1:n) siendo GRAFO(i,j) = true si (i,j) es una arista de G. En otro caso GRAFO(i,j) = false.
- Los colores se representan por los enteros 1, 2, ..., m
- Las soluciones vendrán dadas por n-tuplas (X(1),...,X(n), donde X(i) será el color del vértice i.
- Usando la formulación recursiva del procedimiento backtracking, puede construirse un algoritmo que trabaja en un tiempo $O(nm^n)$

El problema del coloreo de un grafo

- El espacio de estados subyacente es un árbol de grado m y altura n+1, en el que cada nodo en el nivel i tiene m hijos correspondientes a las m posibles asignaciones para X(i), $1 \leq i \leq n$, y donde los nodos en el nivel n+1 son nodos hoja.



El problema del coloreo de un grafo

```
Algoritmo M-Color (k)
while (true)
   SiguienteValor(k)
   if (color[k] = 0) then break                (1)
   if (k = n)
     then print este coloreo                   (2)
     else M-Color (k + 1)                      (3)
endWhile
```

(1) {no hay mas colores para k}
(2) {se encontró un coloreo valido para todos los nodos}
(3) {intenta colorear el siguiente nodo}

El problema del coloreo de un grafo

```
Algoritmo SiguienteValor (k)
    {Devuelve los posibles colores de X(k) dado que X(1)
    hasta X(k-1) ya han sido coloreados}
 while (true)
  color[k] = (color[k] + 1) mod (n + 1)
  if (color[k] = 0) then return                    (1)
  for i = 1 to n+1
    if (conec[i,k] and color[i] = color[k])
      then break
  endfor
  if (i = n+1) return                              (2)
endWhile
```

(1) no hay mas colores para probar

(2) Se ha encontrado un nuevo color (ningun nodo colisiona)

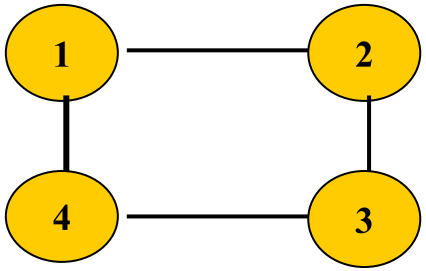
Eficiencia del algoritmo

- El numero de nodos internos en el espacio de estados es $\sum_{i=1..n-1} m^i$
- En cada nodo interno SiguienteValor invierte O(nm) en determinar el hijo correspondiente a un coloreo legal.
- El tiempo total esta acotado por

$$\sum_{i=1..n-1} m^i n = n(m^{n+1}-1)/(m-1) = O(n\ m^n)$$

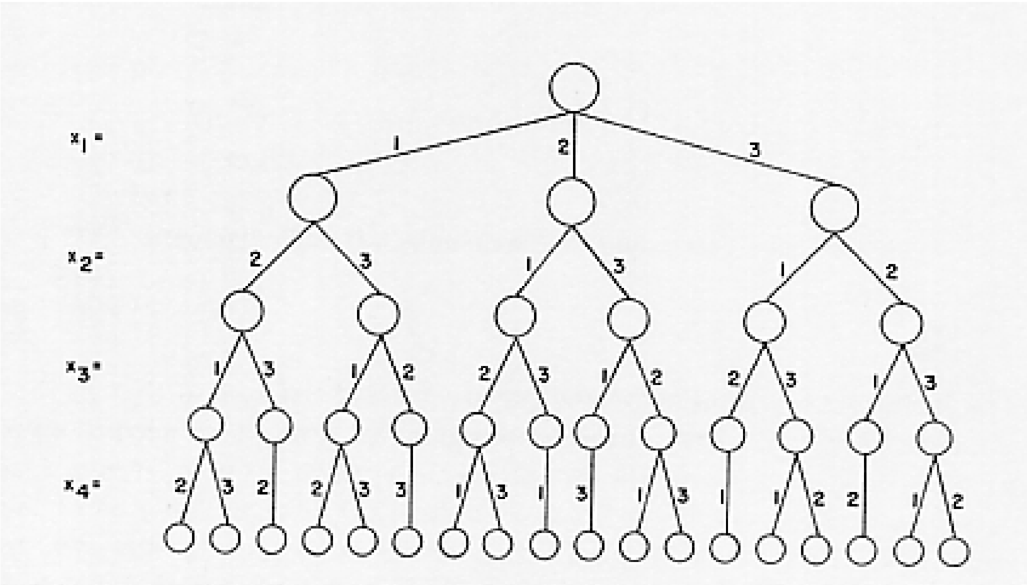
Ejemplo de coloreo

Si consideramos el siguiente grafo

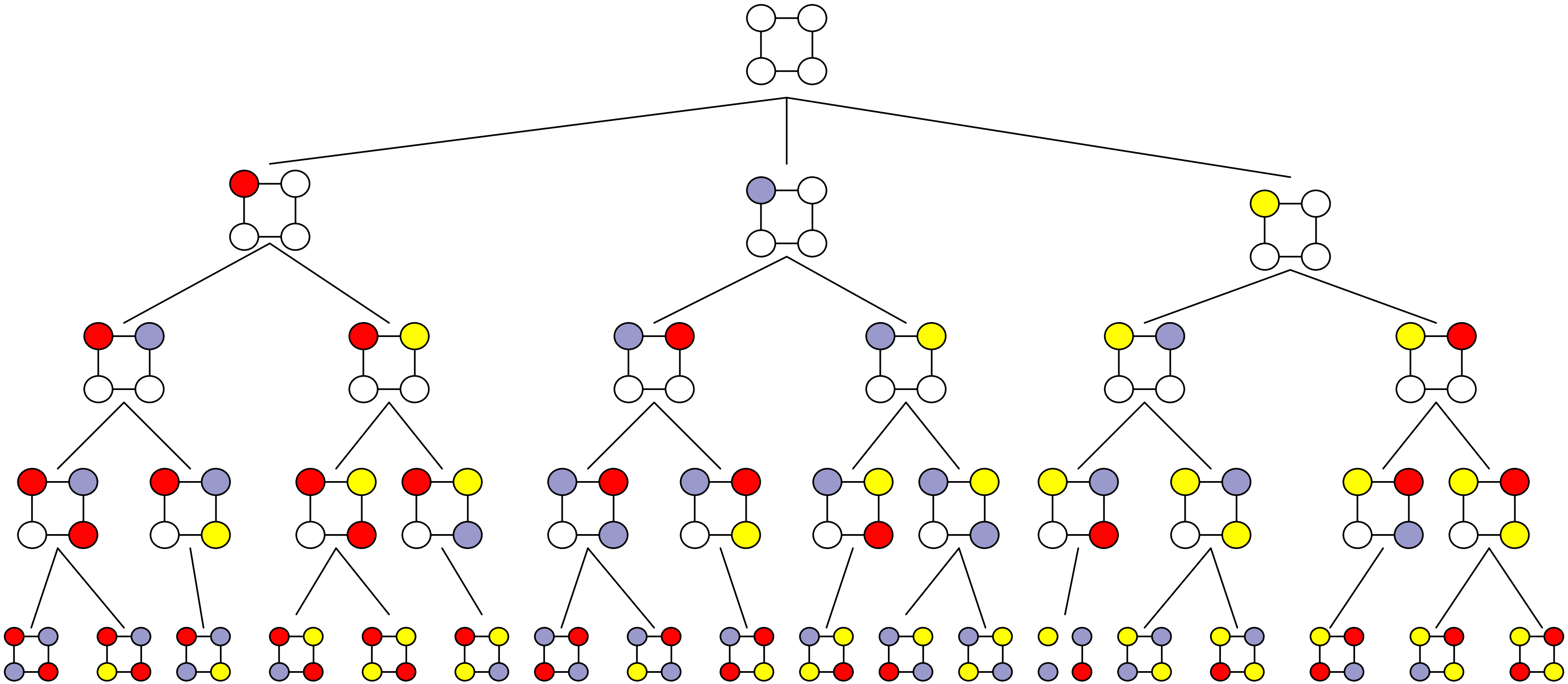


Cada camino a una hoja representa un coloreo usando a lo mas 3 colores

El arbol que genera M-Color es



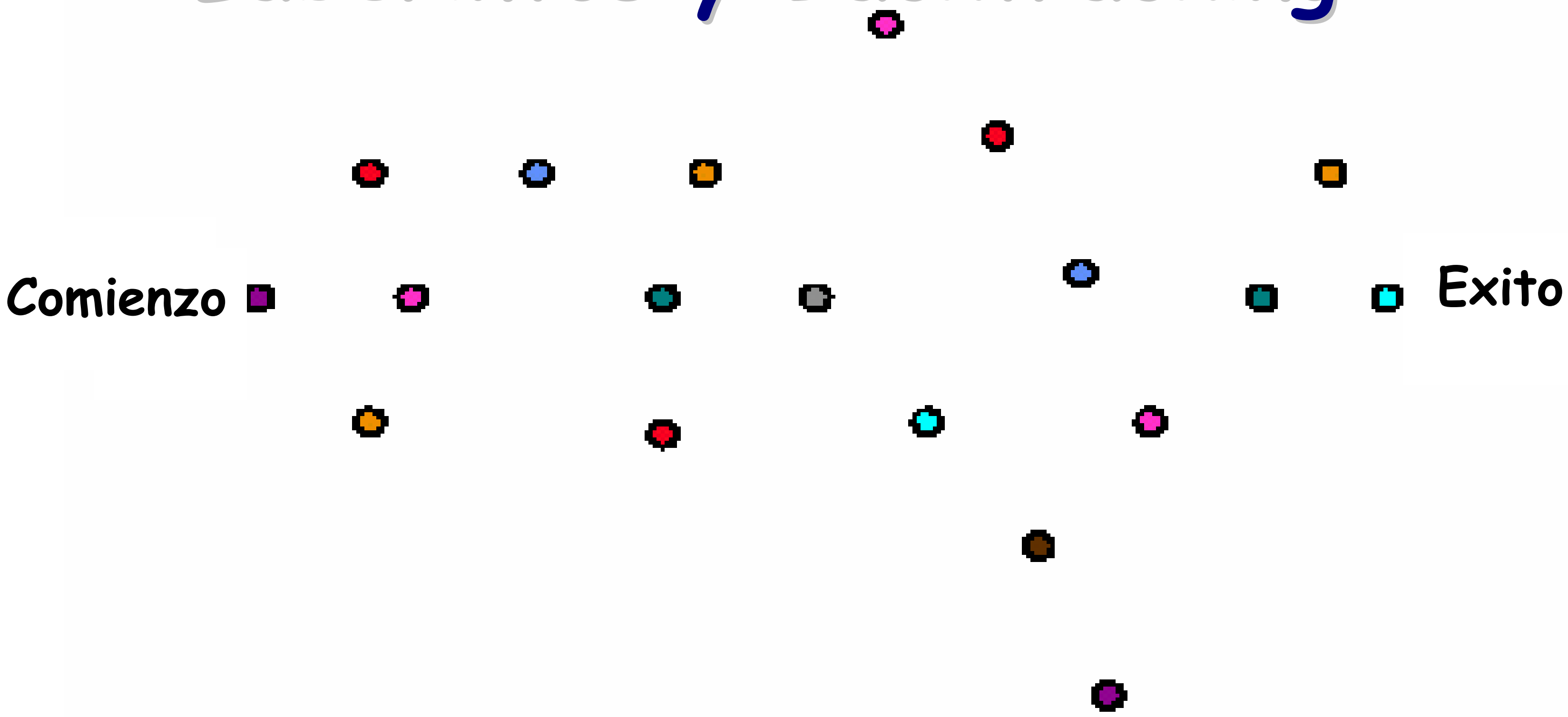
Otra representación del ejemplo



Laberintos y Backtracking

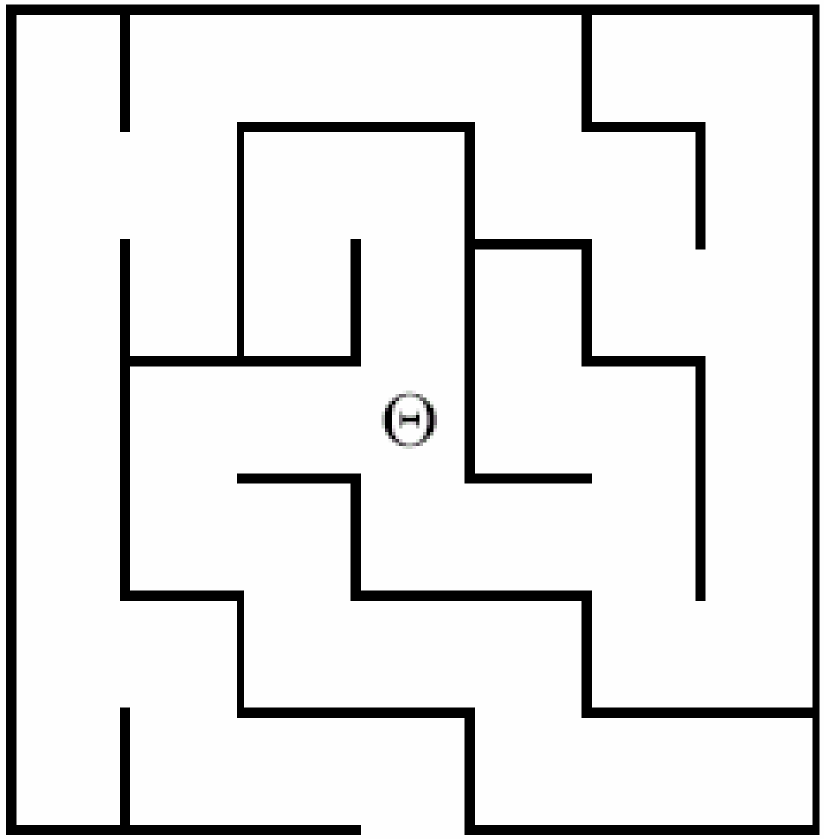
Este mosaico representa un laberinto, y esta en la Catedral de Chartres. Antes de estar alli, ya se conocia en Creta mil años antes. Tambien es conocido en otras culturas.

Laberintos y Backtracking



Un laberinto puede modelarse como una serie de nodos. En cada nodo hay que tomar una decision que nos conduce a otros nodos.

Un laberinto sencillo

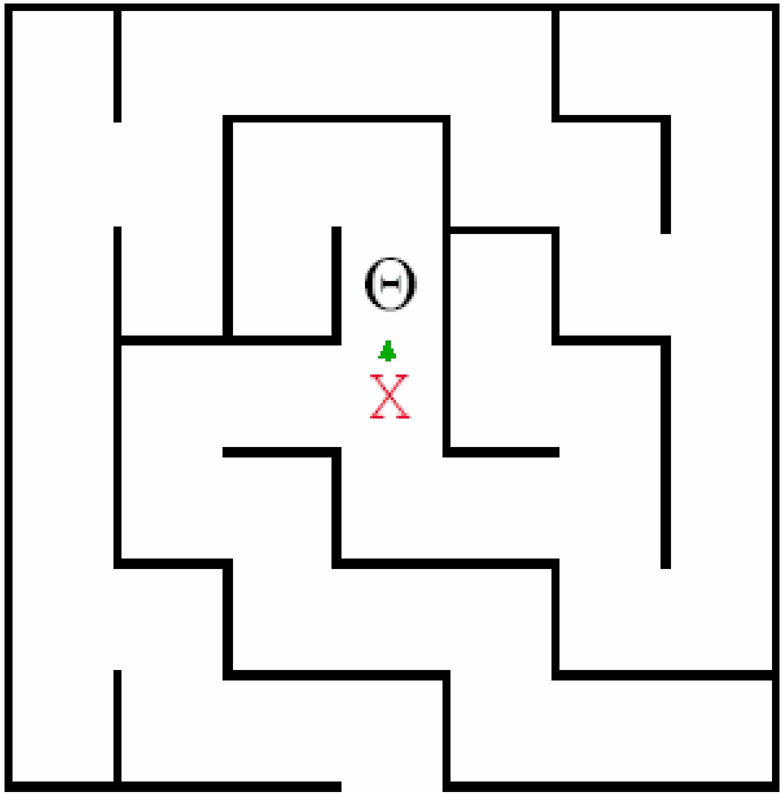


Buscar en el laberinto hasta encontrar una salida. Si no se encuentra una salida, informar de ello

Algoritmo Backtracking Modificado

- Si la posicion actual esta fuera, devolver TRUE para indicar que hemos encontrado una solucion.
  Si la posicion actual esta marcada, devolver FALSE para indicar que este camino ya ha sido explorado.
  Marcar la posicion actual.
  For (cada una de las 4 direcciones posibles **)**
  {   **Si (** Esta direccion no esta bloqueada por un muro **)**
          { Moverse un paso en la direccion indicada desde la posicion actual.
            Intentar resolver el laberinto desde ahi haciendo una llamada recursiva.
            Si esta llamada prueba que el laberinto es resoluble, devolver TRUE para indicar este hecho.
          }
  }
  Quitar la marca a la posicion actual.
  Devolver FALSE para indicar que ninguna de las 4 direcciones lleva a una solucion

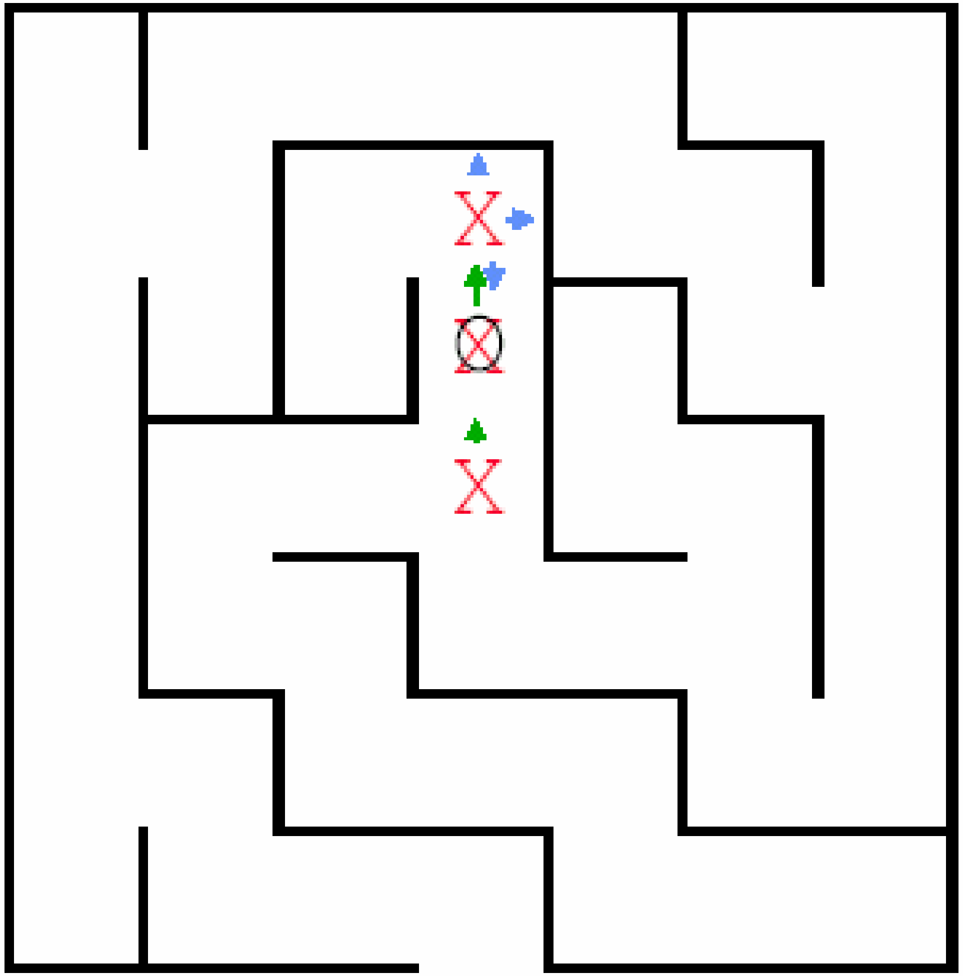
Backtracking en Acción



La parte crucial del algoritmo es el lazo FOR que nos lleva hacia las posibles alternativas que hay en un punto concreto. Aqui nos movemos hacia el norte.

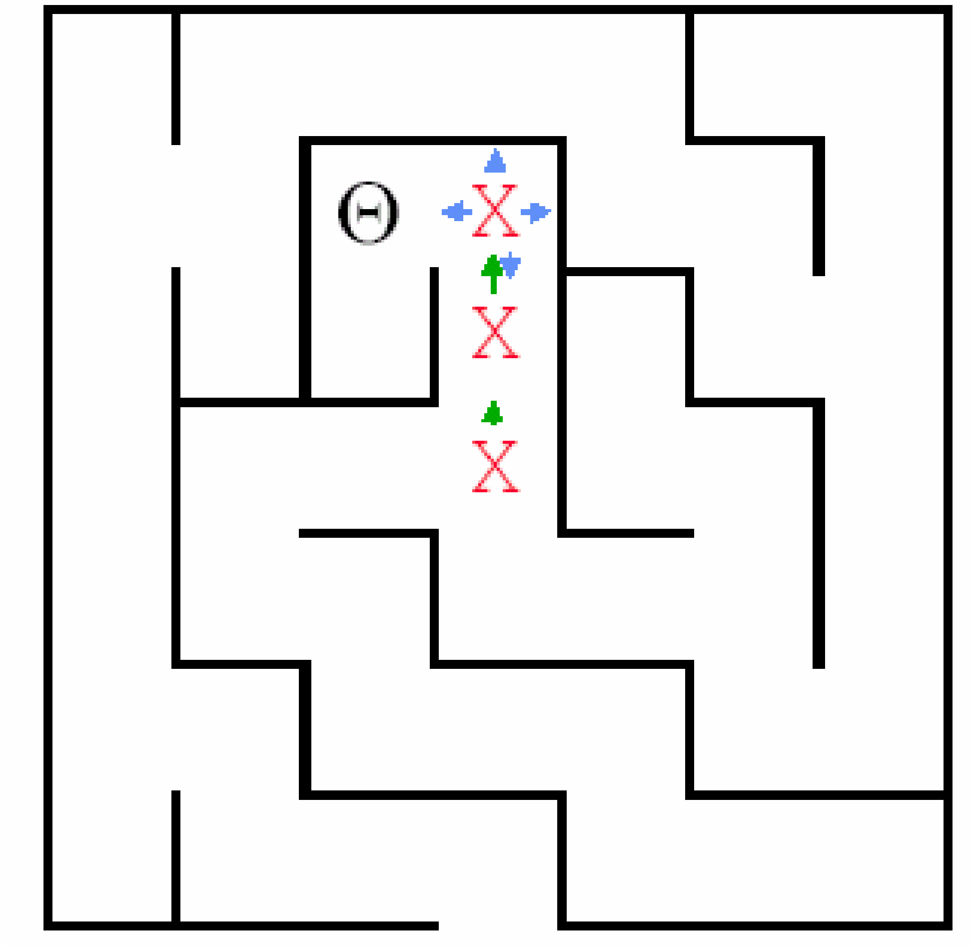
```
for (dir = North; dir <= West; dir++)
{     if (!WallExists(pt, dir))
      {if (SolveMaze(AdjacentPoint(pt, dir)))
            return(TRUE);
}
```

Backtracking in Acción



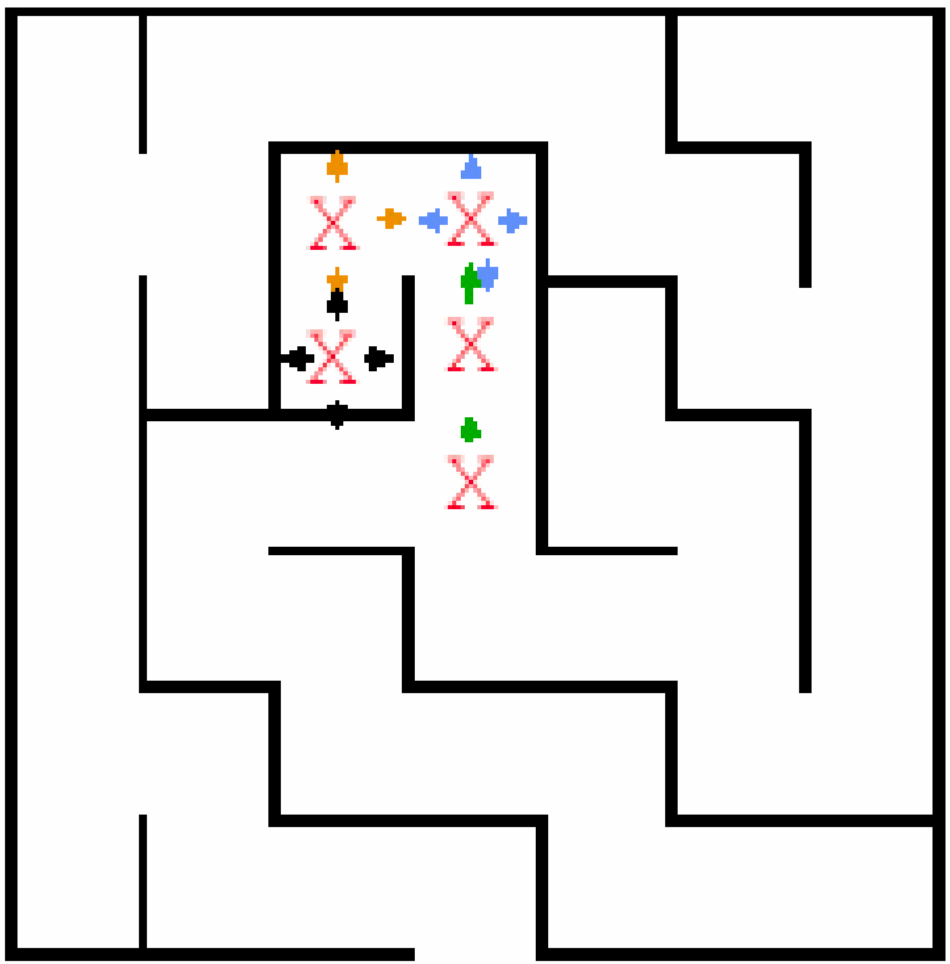
Aqui nos movemos hacia el Norte de nuevo, pero ahora la direccion Norte esta bloqueada por un muro. El Este tambien esta bloqueado, por lo que intentamos el Sur. Esa accion descubre que ese punto esta marcado, de modo que volvemos atrás.

Backtracking in Acción



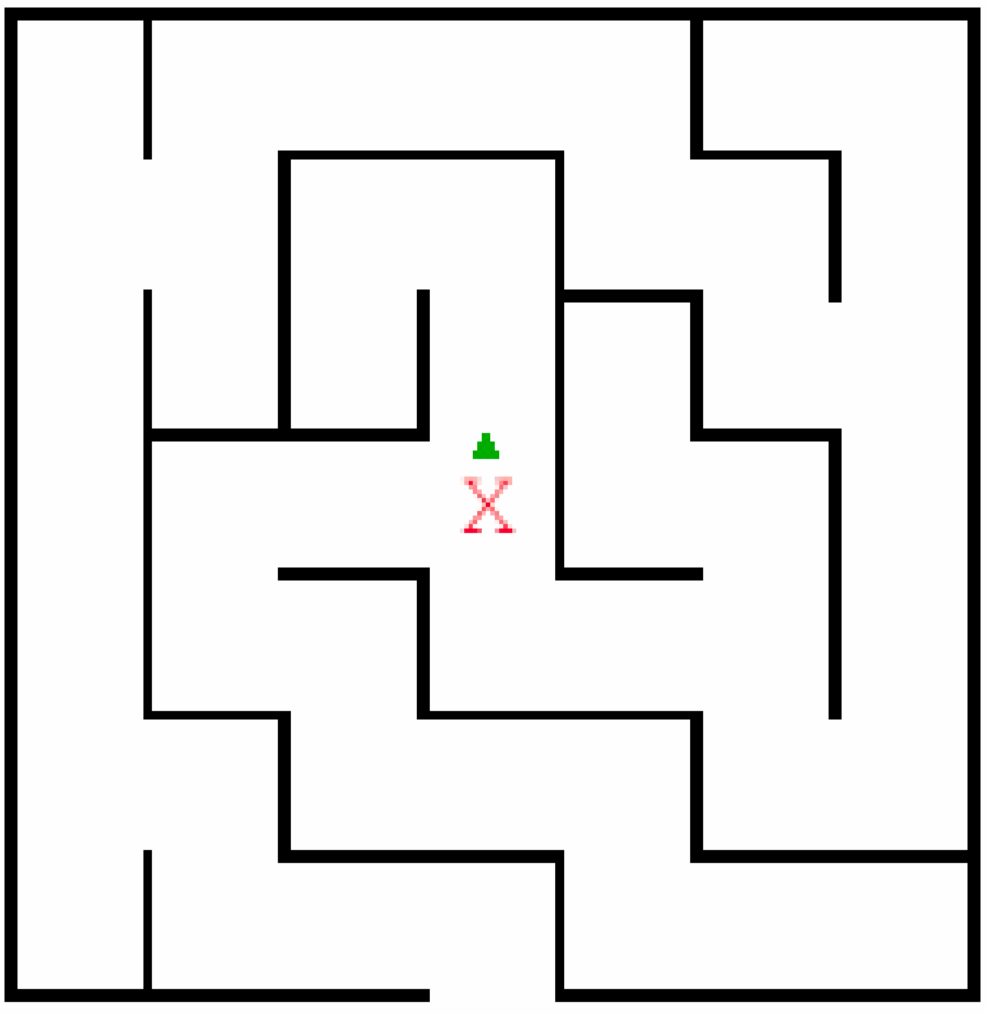
Por tanto el siguiente movimiento que podemos hacer es hacia el Oeste

Backtracking in Acción



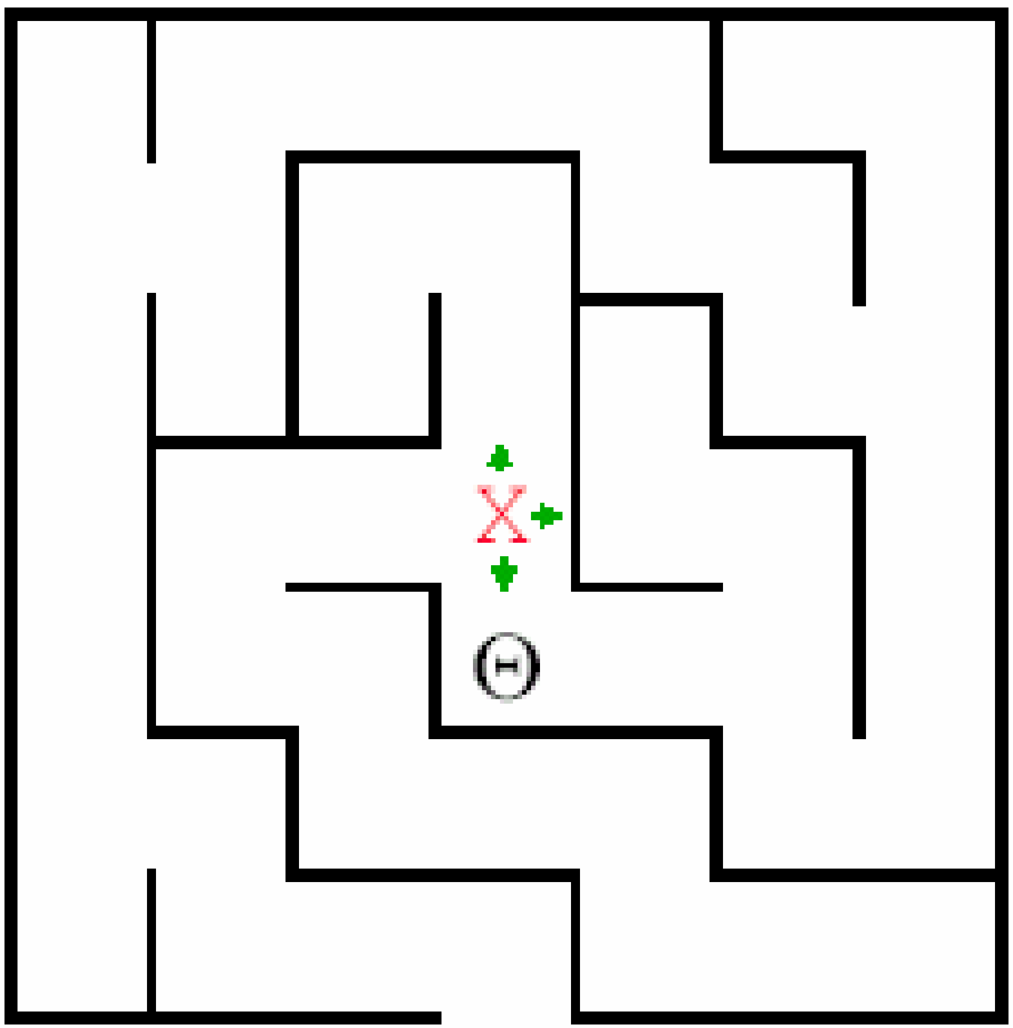
Este camino llega a un nodo (final) muerto .
¡Por tanto es el momento de hacer un backtrack!

Backtracking in Acción



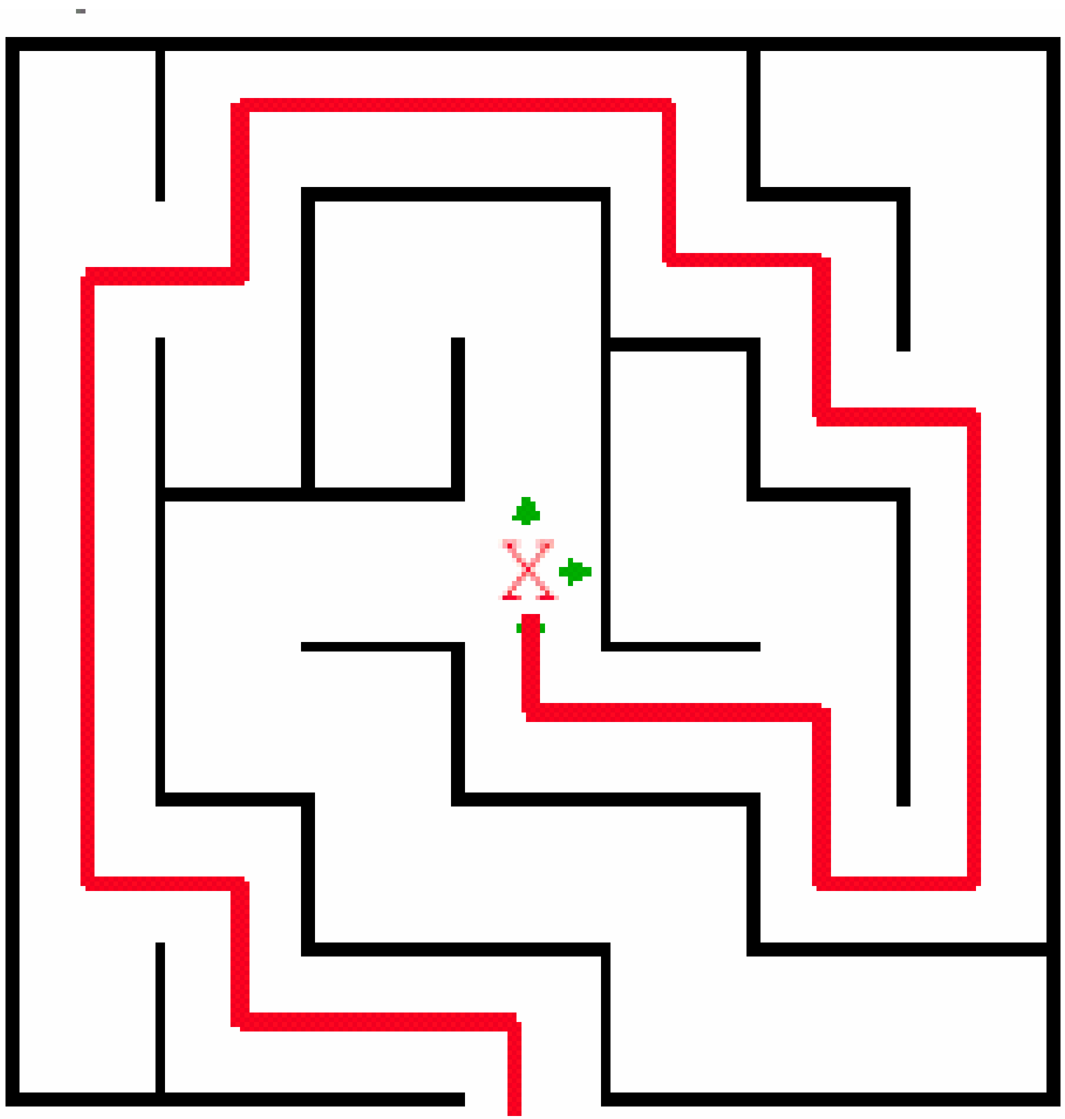
Se realizan sucesivas llamadas recursivas hasta volvernos a encontrar aqui

Backtracking in Acción



Intentamos
ahora el Sur

Primer camino que se encuentra



Ciclos hamiltonianos

- Sea G = (V, E) un grafo conexo con n vértices. Un ciclo Hamiltoniano es un camino circular a lo largo de los n vértices de G que visita cada vértice de G una vez y vuelve al vértice de partida, que naturalmente es visitado dos veces.
- Estamos interesados en construir un algoritmo backtracking que determine todos los ciclos Hamiltonianos de G, que puede ser dirigido o no.
- El vector backtracking solución $(x_1, \ldots, x_n)$ se define de modo que x_i represente el i-esimo vértice visitado en el ciclo propuesto.

Ciclos hamiltonianos

- Todo lo que se necesita es determinar como calcular el conjunto de posibles vértices para x_k si ya hemos elegido $x_1, \ldots, x_{k-1}$.
- Si k = 1, entonces X(1) puede ser cualquiera de los n vértices.
- Para evitar imprimir el mismo ciclo n veces, exigimos que X(1) = 1.
- Si 1 < k < n, entonces X(k) puede ser cualquier vértice v que sea distinto de X(1), X(2),…,X(k-1) que este conectado por una arista a X(k-1).
- X(n) solo puede ser el único vértice restante y debe estar conectado a X(n-1) y a X(1).

Ciclos hamiltonianos

```
Algoritmo Hamiltoniano (k)
while
   x[k] = SiguienteValor(k)
   if (x[k] = 0) then return
   if (k = N) then print solucion
     else Hamiltoniano (k+1)
endWhile
```

Uitlizando este algoritmo podemos particularizar el esquema backtracking recursivo que vimos para encontrar todos los ciclos hamiltonianos

```
Algoritmo SiguienteValor (k)
while
   value = (x[k]+1) mod (N+1)
   if (value = 0) then return value
   if (G[x[k-1],value])
      for j = 1 to k-1  if x[j] = value then break
      if (j=k) and (k < N or k = N and G[x[N],x[1]])
              then return value
endWhile
```