PROGRAMACIÓN POR VUELTA ATRÁS

PROGRAMACIÓN POR VUELTA ATRÁS

Los algoritmos de **vuelta atrás** o retroceso se basan en recorrer el espacio completo de las soluciones posibles al problema planteado

El algoritmo básico de vuelta atrás es el siguiente:

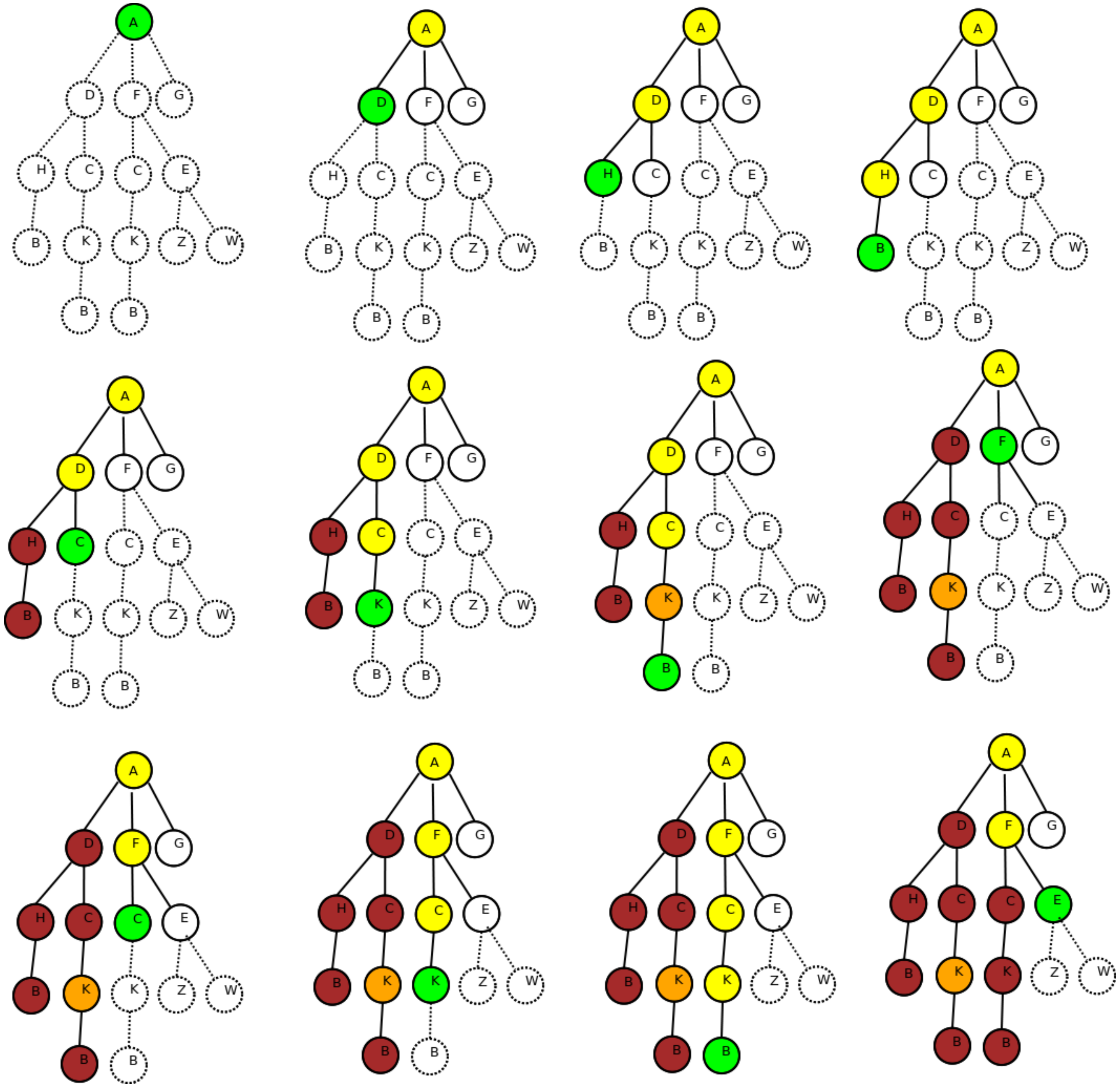
- Tomar una opción de entre las posibles
- Para cada elección, considerar toda opción posible recursivamente
- Devolver la mejor solución encontrada

Esta metodología es lo suficientemente genérica como para ser aplicada en la mayoría de los problemas.

Por el contrario, incluso teniendo cuidado en la implementación, es muy probable que un algoritmo de vuelta atrás sea de tiempo exponencial y no polinómico.

Diseño e implementación

- Esencialmente, la idea es encontrar la mejor combinación posible en un momento determinado.
- Durante la búsqueda, si se encuentra una alternativa incorrecta, la búsqueda retrocede hasta el paso anterior y toma la siguiente alternativa.
- Cuando se han terminado las posibilidades, se vuelve a la elección anterior y se toma la siguiente opción.
- Si no hay más alternativas la búsqueda falla.
- De esta manera, se crea un árbol implícito, en el que cada nodo es un estado de la solución (solución parcial en el caso de nodos interiores o solución total en el caso de los nodos hoja).
- Normalmente, se suele implementar este tipo de algoritmos como un procedimiento **recursivo**.
- Así, en cada llamada al procedimiento se toma una variable y se le asignan todos los valores posibles, llamando a su vez al procedimiento para cada uno de los nuevos estados.


A D F G H C C E B K K Z W B B
A D F G H C C E B K K Z W B B
A D F G H C C E B K K Z W B B
A D F G H C C E B K K Z W B B
A D F G H C C E B K K Z W B B
A D F G H C C E B K K Z W B B
A D F G H C C E B K K Z W B B
A D F G H C C E B K K Z W B B
A D F G H C C E B K K Z W B B
A D F G H C C E B K K Z W B B
A D F G H C C E B K K Z W B B
A D F G H C C E B K K Z W B B

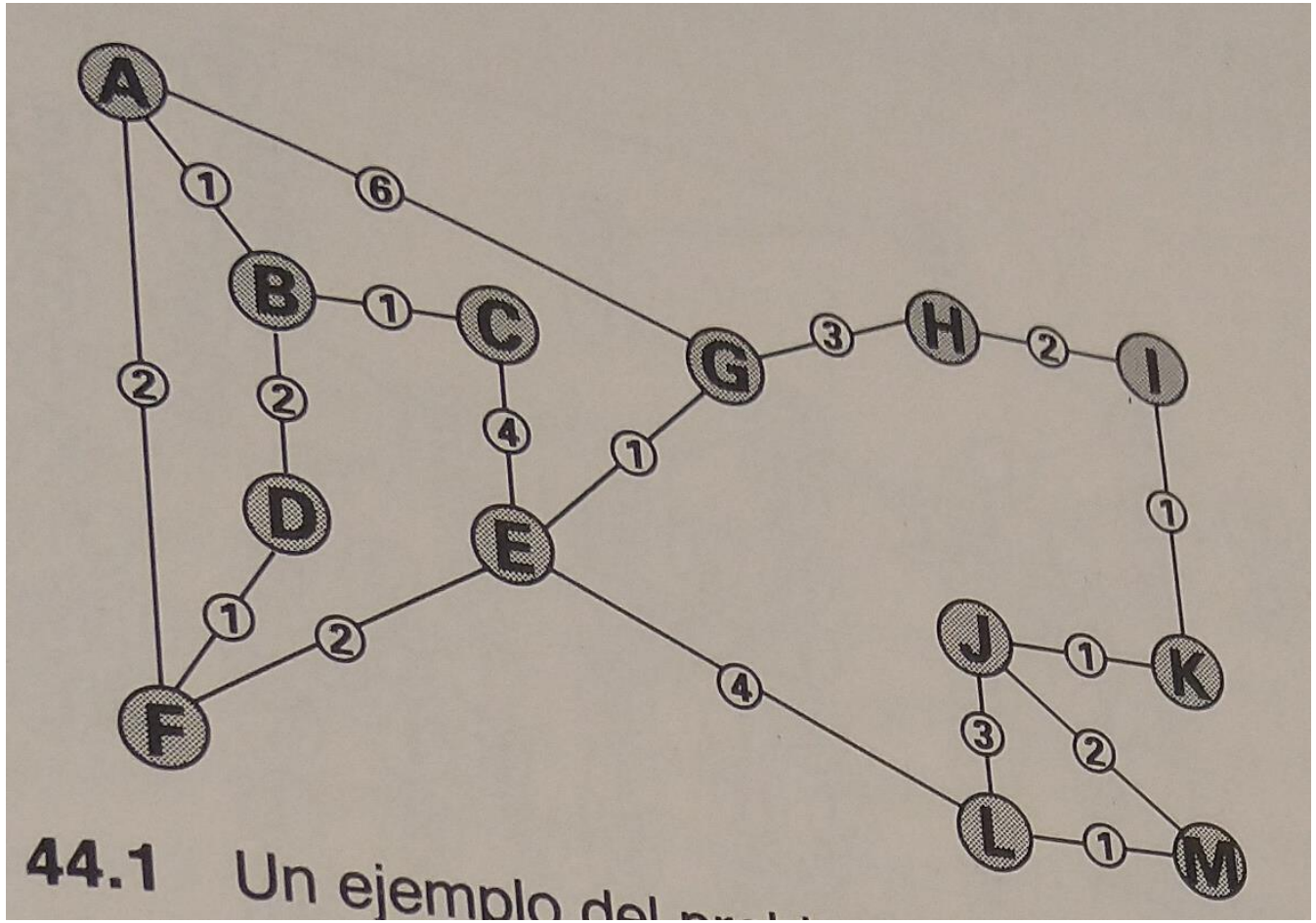
Heurísticas.

- Algunas heurísticas son comúnmente usadas para acelerar el proceso.
- Como las variables se pueden procesar en cualquier orden, generalmente es más eficiente intentar ser lo más restrictivo posible con las primeras.
- Este proceso poda el árbol de búsqueda antes de que se tome la decisión y se llame a la subrutina recursiva.
- Cuando se elige qué valor se va a asignar, muchas implementaciones hacen un examen hacia delante (FC - Forward Checking), para ver qué valor restringirá el menor número posible de opciones, de forma que se anticipa en:
  - Preservar una posible solución
  - Hacer que la solución encontrada no tenga restricciones destacadas.

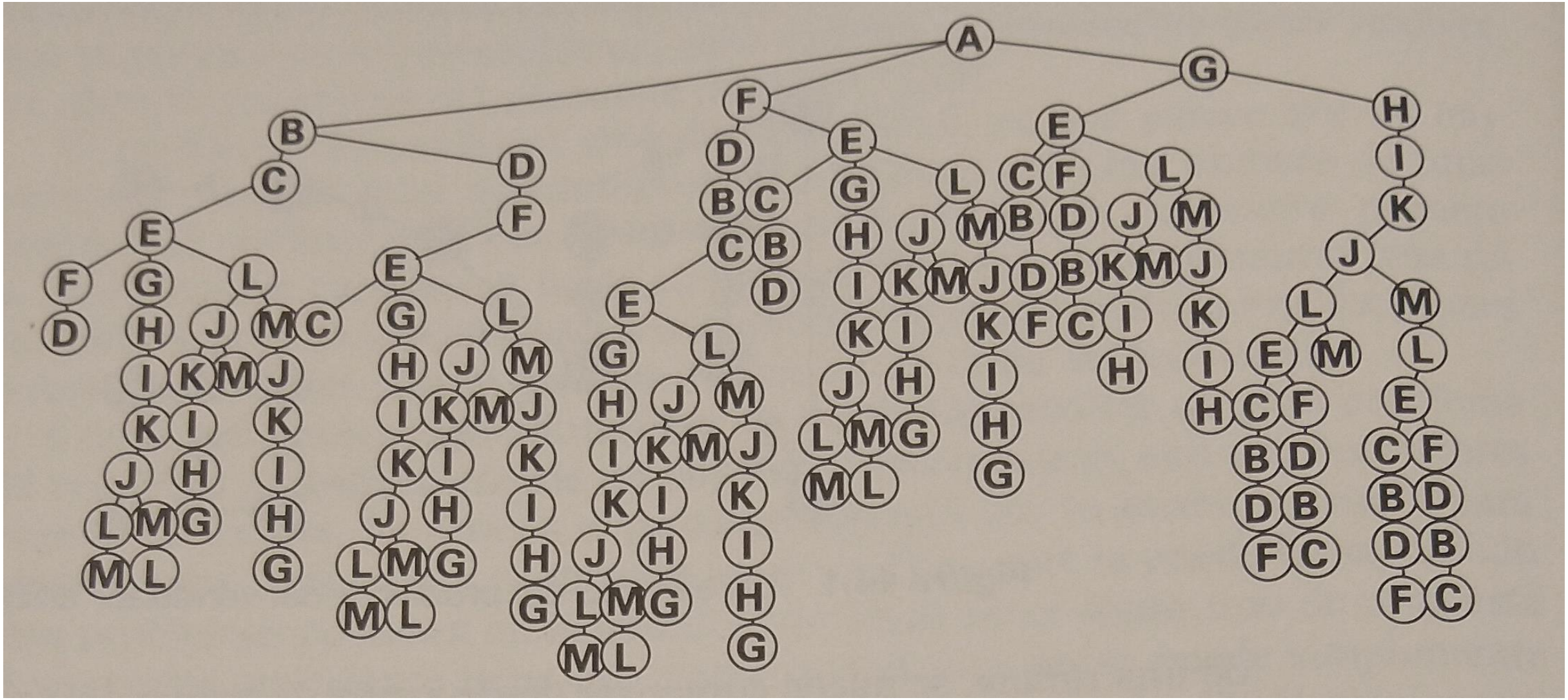
Ramificación y poda

- Este método busca una solución como en el método de backtracking, pero cada solución tiene asociado un costo y la solución que se busca es una de mínimo o máximo costo llamada solución óptima.
- Además de ramificar una solución padre (ramificación) en hijos se trata de eliminar de consideración aquellos hijos cuyos descendientes tienen un costo que supera al óptimo buscado acotando el costo de los descendientes del hijo (límite).
- La forma de acotar es un arte que depende de cada problema.
- La acotación reduce el tiempo de búsqueda de la solución óptima al "podar" los subárboles de descendientes costosos.

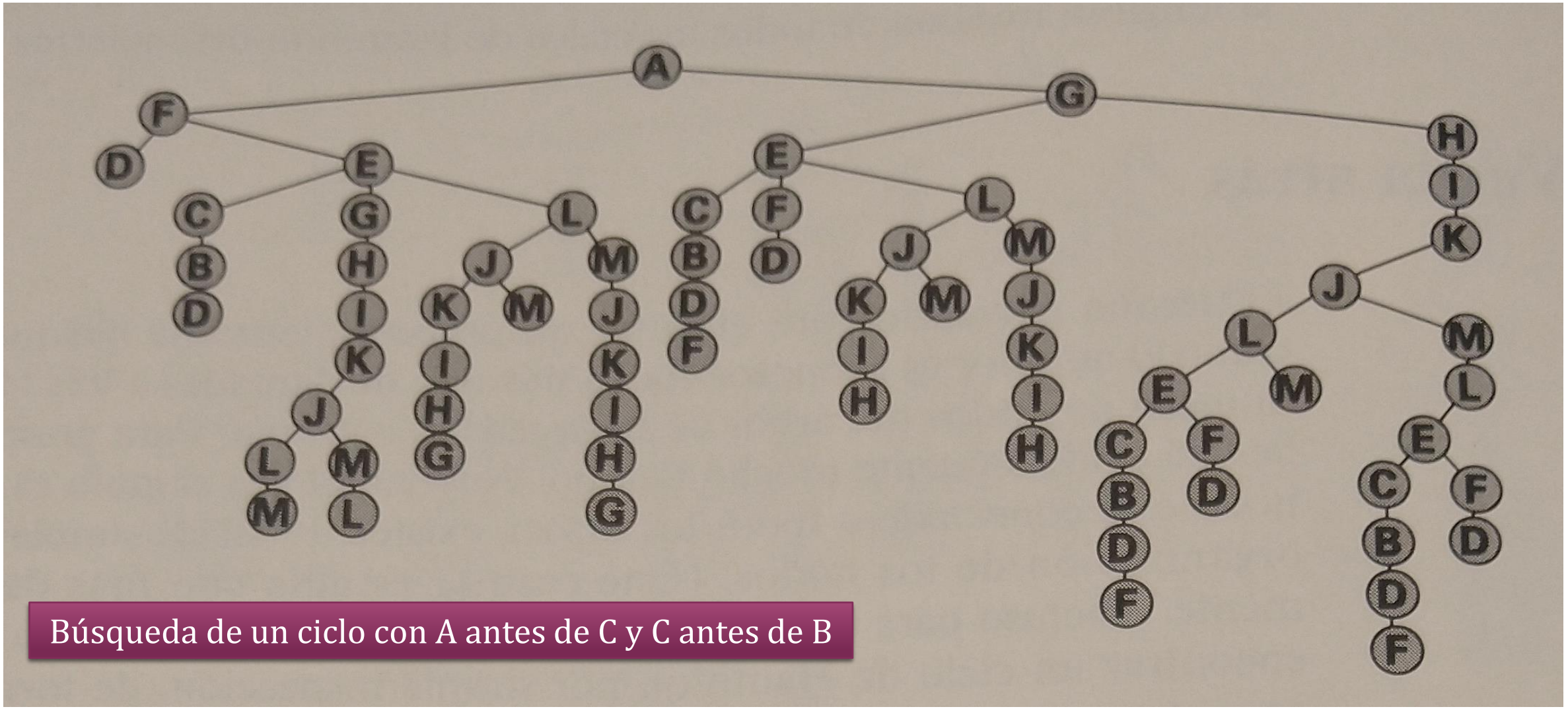
EL PROBLEMA DEL VENDEDOR AMBULANTE



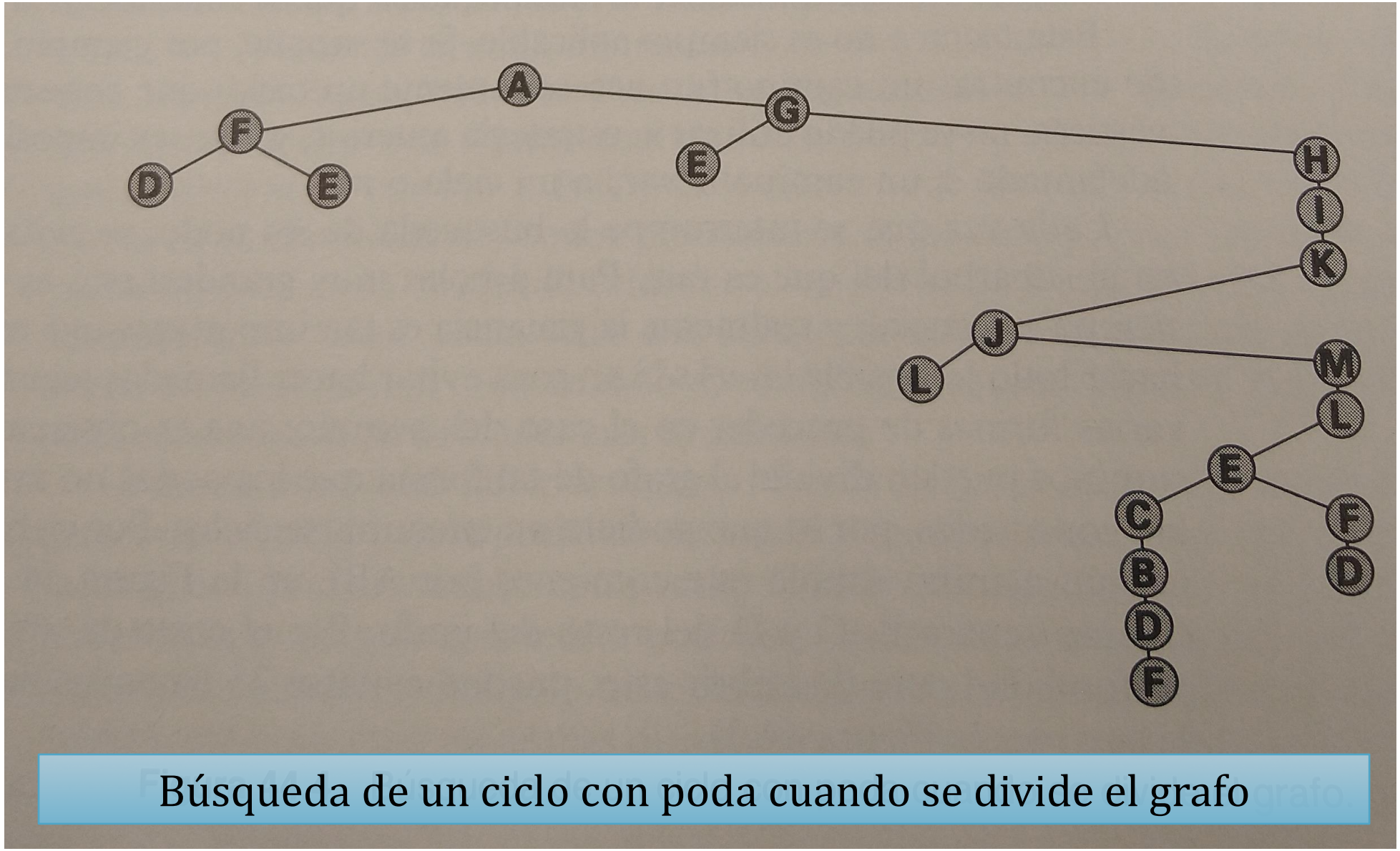
44.1 Un ejemplo del



Búsqueda exhaustiva



Poda del árbol de búsqueda exhaustiva. Cortando ciertas ramas y suprimiendo todo lo conectado a ellas.



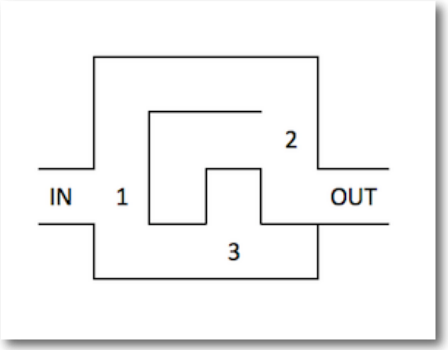
Búsqueda de un ciclo con poda cuando se divide el grafo

Ramificación y poda del problema del vendedor ambulante

- Cuando se busca el mejor camino (en el problema del vendedor ambulante), podemos podar la búsqueda tan pronto se determina que es imposible tener éxito.
- Supóngase que se ha encontrado un camino de coste ***x*** a través del grafo. Entonces es inútil continuar a lo largo de cualquier otro camino cuyo coste es mayor que ***x***.
- La poda es más efectiva si durante la búsqueda se descubre pronto el camino de menor coste; una forma de provocar esto es visitar los nodos adyacentes al actual por orden creciente del coste.

PROBLEMA DEL LABERINTO

- El nombre vuelta atrás *(backtracking)* viene del hecho de que en la búsqueda de la solución se va **volviendo a un punto anterior para probar alternativas**.



- **Imaginemos un laberinto:** al llegar a una encrucijada (1, 2, 3) se prueba con una dirección. Si con eso se llega a la solución, problema resuelto. Si no, se vuelve atrás a la encrucijada anterior y se prueba con otra, repitiendo el proceso cuantas veces sea necesario. (Si se agotan todas las opciones y no se ha llegado, es que no existe solución: *no hay salida*)
- Normalmente las combinaciones de este tipo de problemas son muchas, pero se aplican ciertas restricciones, lo que suele hacer el total de opciones a probar algo computable en un tiempo razonable. También se puede buscar revisar todas planteando obtener «una solución mejor»; de este modo se puede llegar a la **solución óptima.**

PROBLEMA DEL LABERINTO

- El problema del laberinto busca una ruta a través de un laberinto, y devuelve verdadero si hay una posible solución al laberinto.
- La solución implica recorrer el laberinto un paso a la vez, en donde los movimientos pueden ser hacia abajo, a la derecha, hacia arriba o a la izquierda (no se permiten movimientos diagonales).
- De la posición actual en el laberinto (empezando con el punto de entrada), se realizan los siguientes pasos:
  - para cada posible dirección, se realiza el movimiento en esa dirección y se hace una llamada recursiva para resolver el resto del laberinto desde la nueva ubicación.
  - Cuando se llega a un punto sin salida (es decir, no podemos avanzar más pasos sin pegar en la pared), retrocedemos a la ubicación anterior y tratamos de avanzar en otra dirección.
  - Si no puede elegirse otra dirección, retrocedemos de nuevo.
  - Este proceso continúa hasta que encontramos un punto en el laberinto en donde *puede realizarse un movimiento en* otra dirección.
  - Una vez que se encuentra dicha ubicación, avanzamos en la nueva dirección y continuamos con otra llamada recursiva para resolver el resto del laberinto.

PROBLEMA DEL LABERINTO

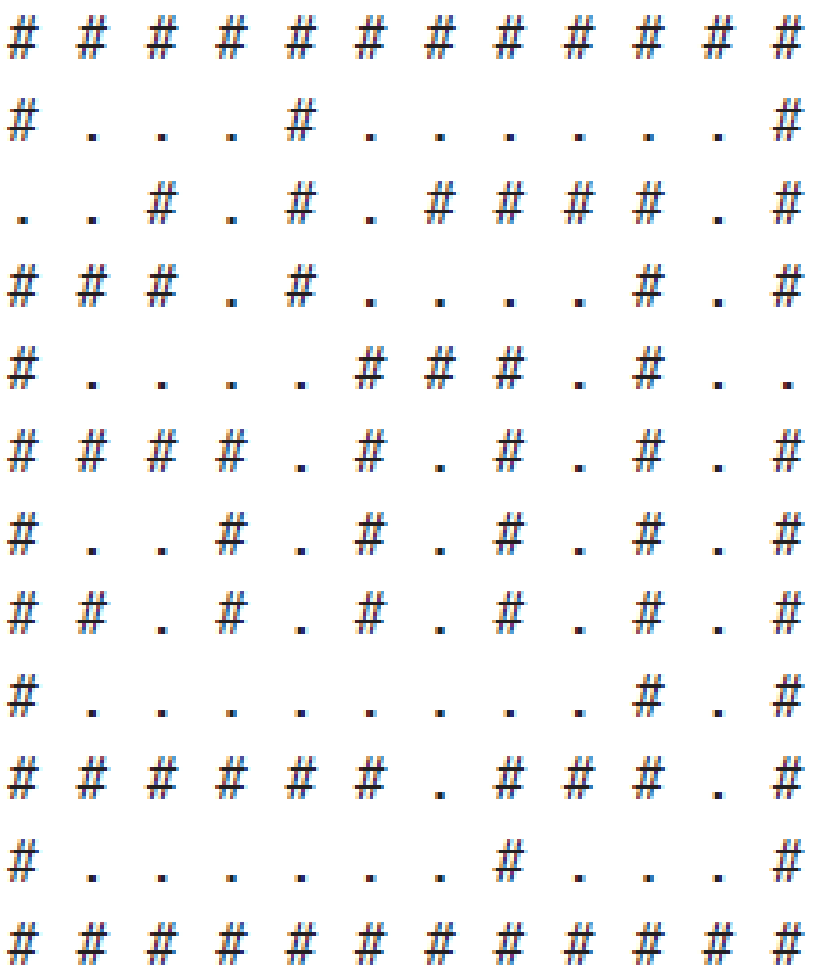
- Para retroceder a la ubicación anterior en el laberinto, nuestro método recursivo simplemente devuelve falso, avanzando hacia arriba en la cadena de llamadas a métodos, hasta la llamada recursiva anterior (que hace referencia a la ubicación anterior en el laberinto).

- A este proceso de utilizar la recursividad para regresar a un punto de decisión anterior se le conoce como "**vuelta atrás" recursiva** o ***backtracking.***

- Si un conjunto de llamadas recursivas no resulta en una solución para el problema, el programa retrocede hasta el punto de decisión anterior y toma una decisión distinta, lo que a menudo produce otro conjunto de llamadas recursivas.

- Si la vuelta atrás llega a la ubicación de entrada del laberinto y se han recorrido todas las direcciones, entonces el laberinto no tiene solución.

Representación de un laberinto mediante un arreglo bidimensional

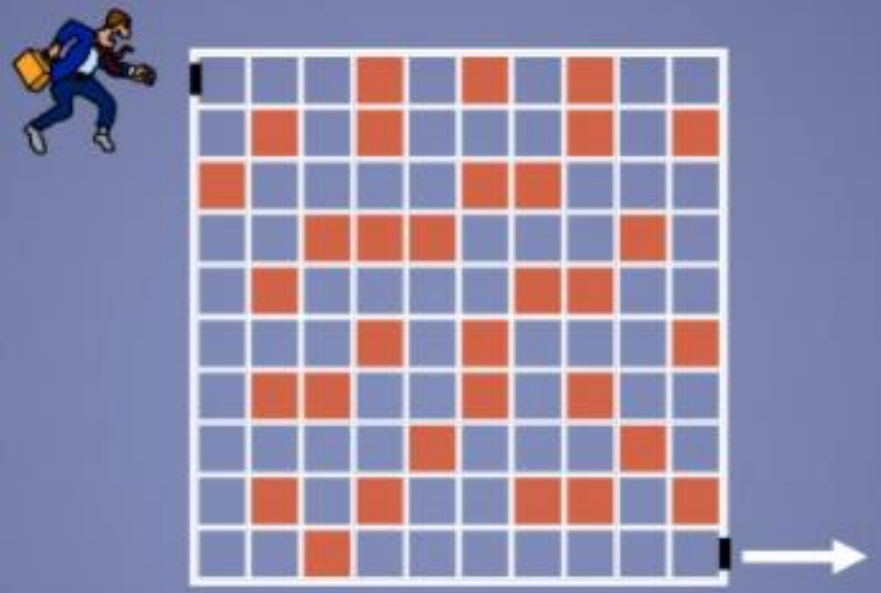


Los caracteres # representan las paredes del laberinto, y los puntos representan las ubicaciones en las posibles rutas a través del laberinto.

Sólo pueden realizarse movimientos hacia una ubicación en el arreglo que contenga un punto.

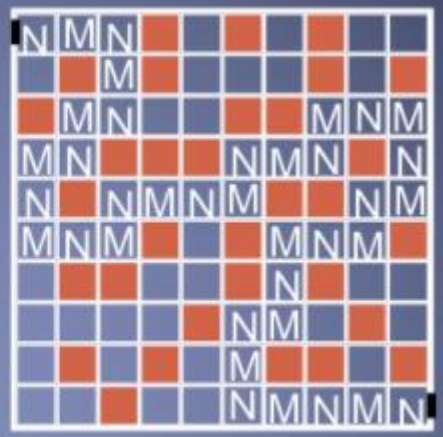
PROBLEMA DEL LABERINTO

- Nos encontramos en una entrada de un laberinto y debemos intentar atravesarlo.

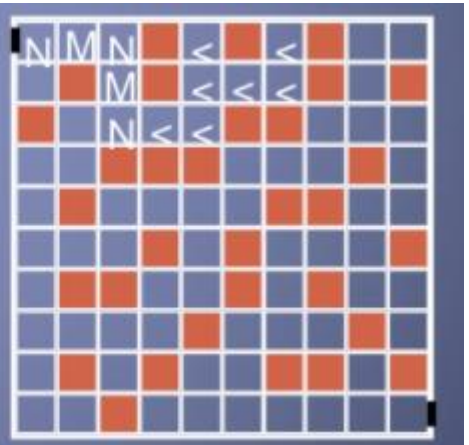


- Representación: matriz de dimensión $n \times n$ de casillas marcadas como *libre* u *ocupada* por una pared.
- Es posible pasar de una casilla a otra moviéndose sólamente en vertical u horizontal.
- Se debe ir de la casilla (1,1) a la casilla (n,n).

- Diseñaremos un algoritmo de búsqueda con retroceso de forma que se marcará en la misma matriz del laberinto un camino solución (si existe).



- Si por un camino recorrido se llega a una casilla desde la que es imposible encontrar una solución, hay que volver atrás y buscar otro camino.



- Además hay que marcar las casillas por donde ya se ha pasado para evitar meterse varias veces en el mismo callejón sin salida, dar vueltas alrededor de columnas…

Tipos de estructuras de datos

```
Casilla = (libre, pared, camino, imposible)
Laberinto = vector[1..n, 1..n] de casilla
```

Llamada inicial, solución de búsqueda con retroceso

```
HayCamino(1, 1, lab)
```

```
funcion HayCamino(x,y:entero, lab:laberinto)
//Indica si se ha encontrado un camino desde (1,1) hasta (x,y)
//Devuelve cierto si y solo si se puede extender hasta (n,n)
```

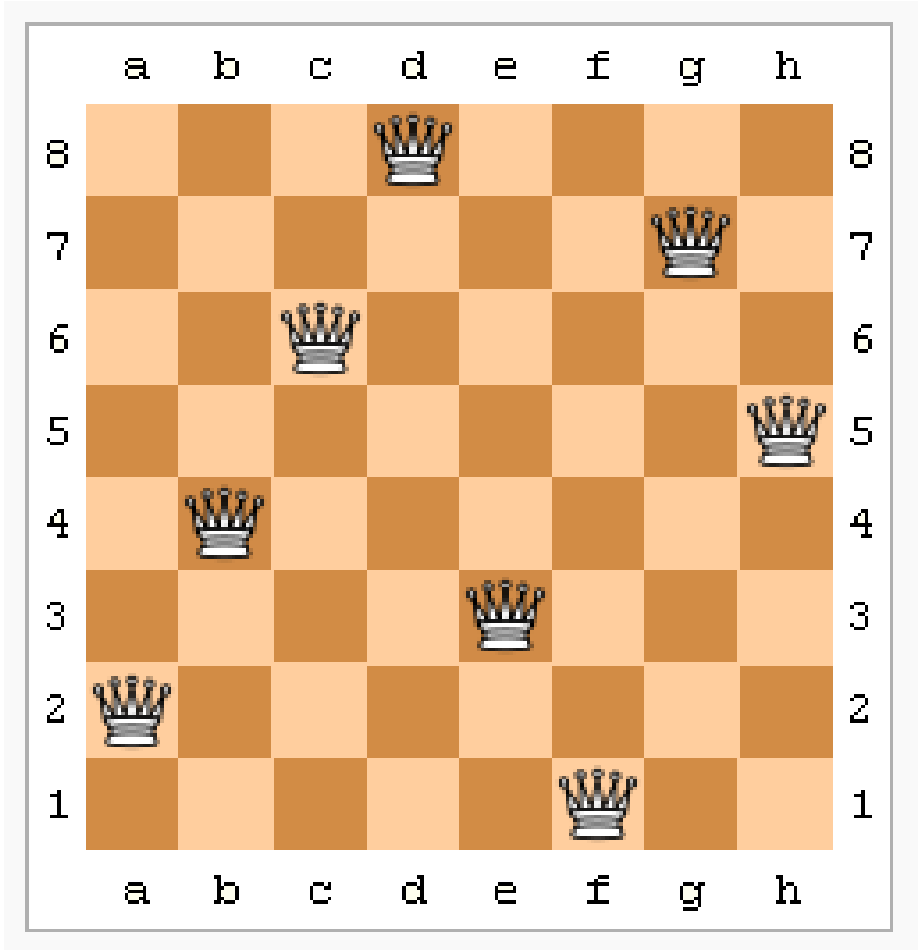
```
funcion HayCamino(x,y:entero, lab:laberinto)
    si (x<1) ˇ (x>n) ˇ (y<1) ˇ (y>n) ˇ lab[x,y]≠libre entonces
        devuelve falso;
    sino
        lab[x,y] = camino;
        si (x=n)^(y=n) entonces
            mostrar(lab);
            devuelve cierto;
        sino
            b = HayCamino(x+1,y,lab) ˇ
                HayCamino(x,y+1,lab) ˇ
                HayCamino(x-1,y,lab) ˇ
                HayCamino(x,y-1,lab);
            si ¬b entonces
                lab[x,y] = imposible;
            fin_si
            devuelve b;
        fin_si
    fin_si
fin_funcion
```

EL PROBLEMA DE LAS OCHO REINAS

- El **problema de las ocho reinas** es un pasatiempo que consiste en poner ocho reinas en el tablero de ajedrez sin que se amenacen.

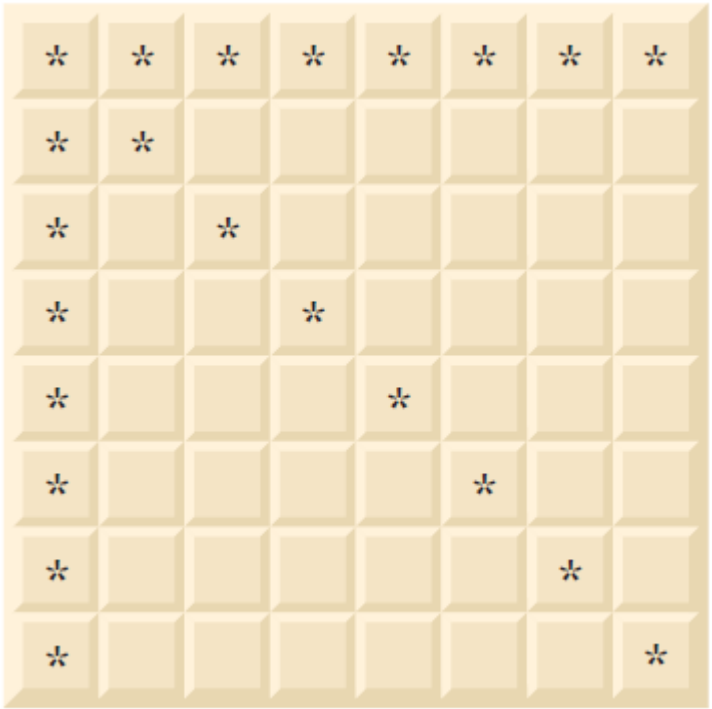
- En el juego del ajedrez la reina amenaza a aquellas piezas que se encuentren en su misma fila, columna o diagonal.

- Para resolver este problema se puede emplear un esquema vuelta atrás (o Backtracking).

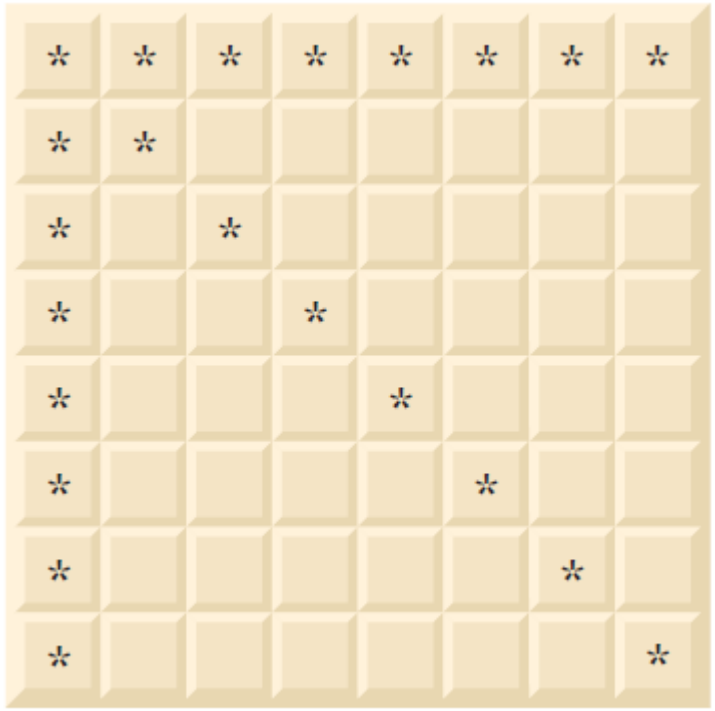


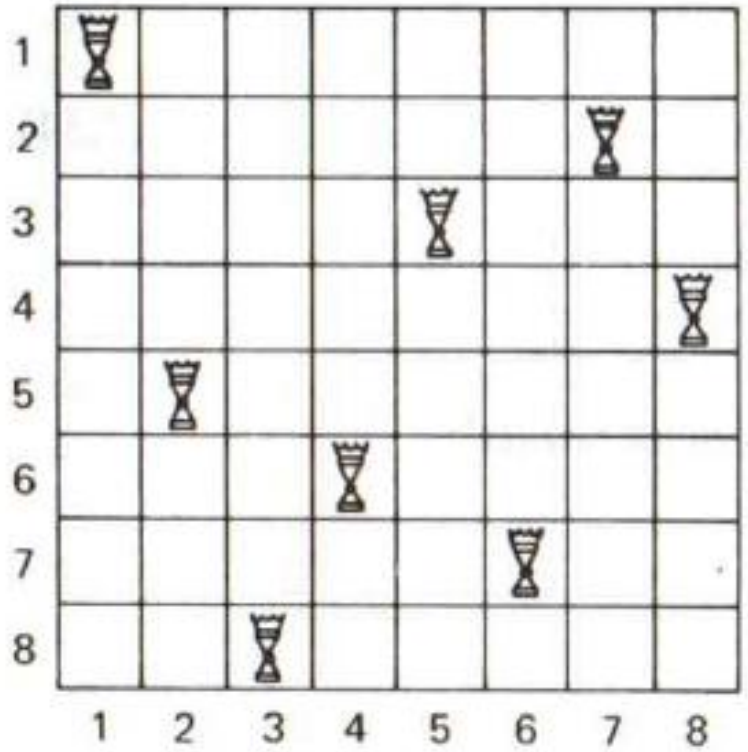
EL PROBLEMA DE LAS OCHO REINAS

- Su solución debe empezar con la primera columna y buscar una ubicación en esa columna, en donde pueda colocarse una reina; al principio, coloque la reina en la primera fila.
- Después, la solución debe buscar en forma recursiva el resto de las columnas.
- En las primeras columnas, habrá varias ubicaciones en donde pueda colocarse una reina. Tome la primera posición disponible.
- Si se llega a una columna sin que haya una posible ubicación para una reina, el programa deberá regresar a la columna anterior y desplazar la reina que está en esa columna hacia una nueva fila.
- Este proceso continuo de retroceder y probar nuevas alternativas es un ejemplo de la ***"vuelta atrás"*** recursiva.



EL PROBLEMA DE LAS OCHO REINAS





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