Programación y Estructuras de Datos Avanzadas

Capítulo 6: Vuelta atrás

Capítulo 6: Vuelta atrás

6.1 Planteamiento y esquema general

- La **vuelta atrás** (retroceso o backtracking) es un esquema que hace una búsqueda exhaustiva y sistemática de todas las posibles soluciones $\rightarrow$ **muy lento** por lo que sólo se usará si no hay otro esquema algorítmico**.**
- Tiene una diferencia fundamental con los algoritmos voraces:
  - **Algoritmo voraz:** añade/descarta elementos a la solución y no deshace ninguna decisión tomada.
  - **Vuelta atrás:** añade y quita todos los elementos a la posible solución. Prueba todas las combinaciones hasta encontrar dicha solución.
- La ejecución del algoritmo se representa en un **árbol o un grafo**:
  - Suele ser **implícito (*no se genera, se almacena y luego se recorre*)**, sino que se van construyendo sólo las partes a las que llega el recorrido.
  - El nodo raíz es la situación inicial $\rightarrow$ solución vacía.
  - El esquema de vuelta **atrás realizará un recorrido en profundidad del grafo implícito de un problema**.
  - **¿Cómo es este grafo?**

Descripción del grafo de búsqueda (o de soluciones)

- Una **solución** se puede expresar como una tupla, **(x_1, x_2,..., x_n)**, que satisfaga las restricciones impuestas por el problema.
- **El grafo suele ser acíclico y en, la mayoría de los casos un árbol.**
- Las soluciones **se construyen de forma incremental**: en cada momento, el algoritmo se encontrará en cierto nivel **k** del árbol, con una solución parcial (x_1, ..., x_{k-1}), $k \leq n$ $\leftarrow$ solución/secuencia k-prometedora (*INTERPRETACIÓN*)
  - **Si se puede** añadir un nuevo elemento a la solución $\mathbf{x_{k+1}}$, se genera y se **avanza al nivel k+1**, extendiendo la solución $\rightarrow$ **solución k+1-prometedora**
  - **Si no se puede**, decimos que estamos ante un ***nodo de fallo*** $\rightarrow$ en este caso el algoritmo **retrocede** al nivel anterior **k-1** probando otros valores para $\mathbf{x_k}$ si es posible (si no retrocede hasta el nodo y nivel en el que queden ramas pendientes de ser exploradas).
  - Se sigue hasta que la solución parcial sea una solución completa del problema, o hasta que no queden más posibilidades por probar.

- **Esquema general (construcción incremental de las soluciones):**

```
fun VueltaAtras (v: Secuencia, k: entero)
    {v es una secuencia k-prometedora}
    IniciarExploraciónNivel(k)
    mientras OpcionesPendientes(k) hacer
        extender v con siguiente opción
        si SoluciónCompleta(v) entonces
            ProcesarSolución(v)
        sino
            si Completable (v) entonces
                VueltaAtras(v, k+1)
            fsi
        fmientras
    fsi
ffun
```

v secuencia k-prometedora (solución parcial con **k-1 elementos** asignados que cumplen las restricciones) → exploraremos el nivel k del árbol

Inicializaciones para comenzar la exploración del nivel k

Comprueba que quedan opciones por explorar en el nivel k (compleciones)

Asigna la siguiente opción a v[k]

Comprueba si se ha completado una solución al problema, es decir, si v es solución (suele incluir k=n)

Representa las operaciones que se desean hacer con la solución como imprimirla, almacenarla o devolverla al punto de llamada.

Comprueba si tras añadir ese elemento al vector éste sigue siendo factible para la solución (poda por factibilidad)

Puesto que v ya cumple las restricciones hasta el nivel k, invoco recursivamente el algoritmo para seguir extendiéndolo

- Normalmente se invocará con VueltaAtras(nodo_raiz,1)=VueltaAtras([_,_,_,_,..,_],1)
- Este algoritmo no se detiene al encontrar la primera solución → exploración completa

- **Si sólo se necesita una solución → añadimos *encontrado***

```
fun VueltaAtras (v: Secuencia, k: entero, encontrado: booleano)
    {v es una secuencia k-prometedora}
    IniciarExploraciónNivel(k)
    mientras OpcionesPendientes(k) ∧ ¬ encontrado hacer
        extender v con siguiente opción
        si SoluciónCompleta(v) entonces
            ProcesarSolución(v)
            encontrado ← cierto
        sino
            si Completable (v) entonces
                VueltaAtras(v, k+1, encontrado)
            fsi
        fsi
    fmientras
ffun
```

← Se detiene al encontrar la primera solución

Llamada inicial:
VueltaAtras([_,_,_,_,..,_],1,falso)

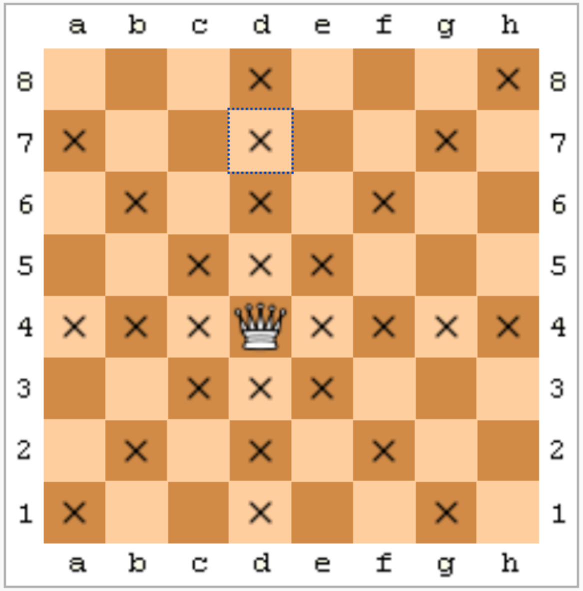
Coste de los algoritmos vuelta atrás:

- Algoritmos de búsqueda exhaustiva.
- Coste en el caso peor → del orden del tamaño del espacio de búsqueda.
- No se puede estimar cuánto se poda el árbol (depende de los datos de entrada)

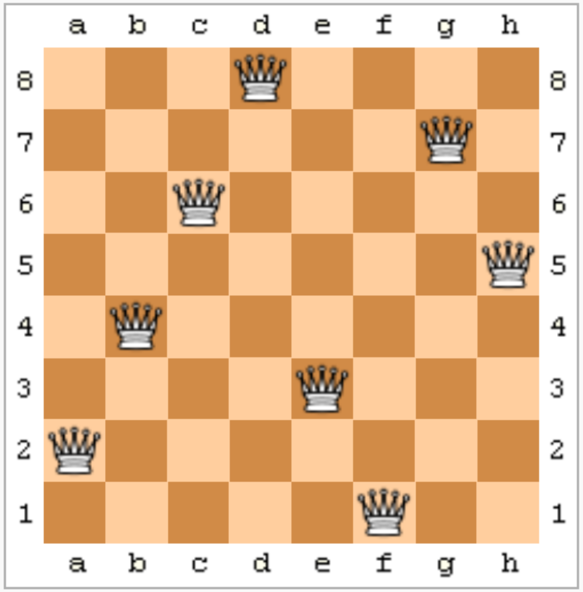
→ **Daremos como cota superior al coste el tamaño máximo del árbol de soluciones**

• Ejemplo: Problema de las n reinas de ajedrez

- El problema de las 8 reinas (n=8) consiste en colocar 8 reinas en un tablero de ajedrez sin que se amenacen entre ellas.



Dama. Movimientos posibles de la pieza.



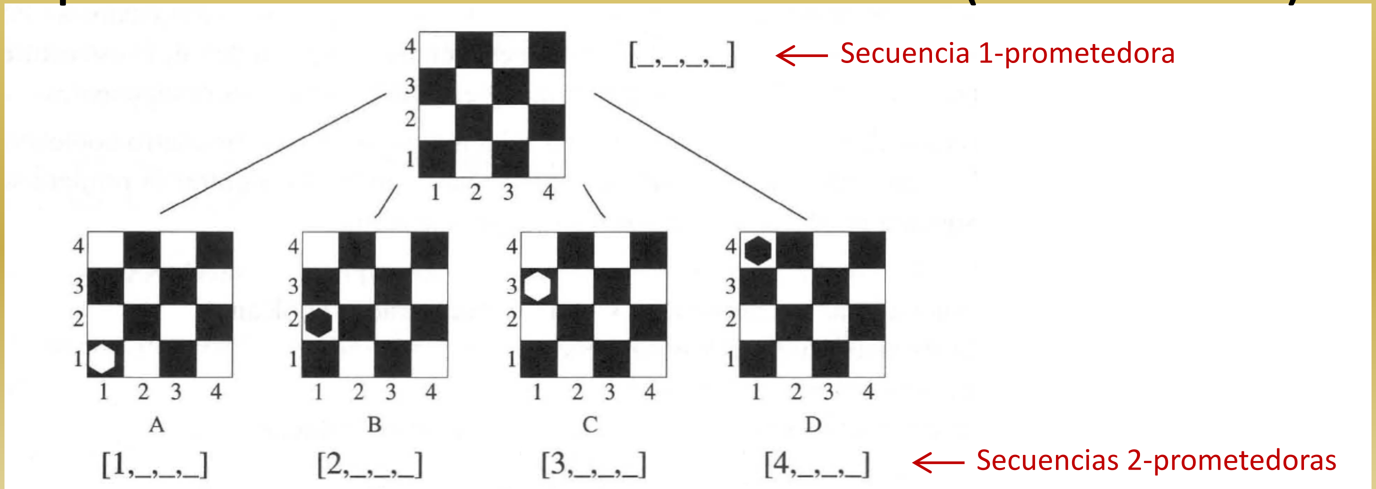
Una posible solución entre las 92 posibles soluciones en un tablero de 8x8

– **No existe un algoritmo eficiente para resolver este problema → búsqueda exhaustiva (algoritmo de vuelta atrás)**

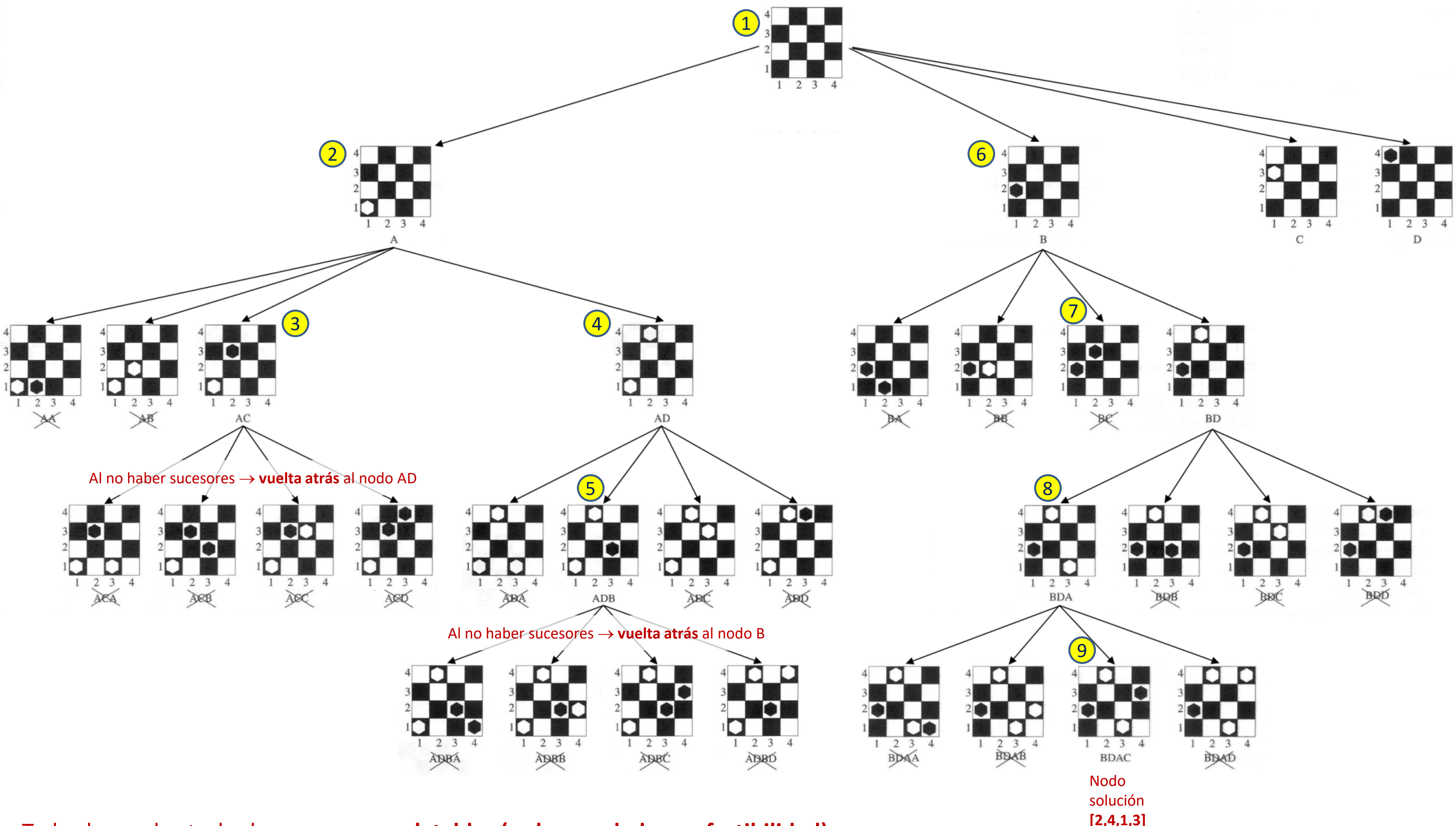
- **Planteamiento del problema:**
  - **n**: número de reinas a colocar en un tablero n×n.
  - No puede haber 2 reinas ni en la misma fila, columna o diagonal (*poda por factibilidad*).

- **Descripción de la solución:**
  - **v: matriz [1..n] de entero**, donde **v[i]=nº de fila en la que se colocará la reina i-ésima** (necesariamente estarán en distinta fila).
  - Solución inicial: v=[_,_,…,_] o v=[0,0,…,0] (_ ó 0 indica que todavía no se ha asignado dicha reina).
  - *Un vector será k+1-prometedor si ninguna de las k primeras reinas asignadas se amenazan → Así está el pseudocódigo (lo que implicaría que la solución es un vector n+1-prometedor).*

- **Ejemplo para n=4 → 4 reinas a colocar en tablero 4x4 (situación inicial):**



Árbol de soluciones/búsqueda del problema de las 4 reinas (sólo busca 1 solución)



Todos los nodos tachados son **no completables (se han podado por factibilidad).**

- **No se usará exactamente el pseudocódigo general:**

Algoritmo original

```
fun VueltaAtras (v: Secuencia, k: entero)
    {v es una secuencia k-prometedora}
    IniciarExploraciónNivel(k)
    mientras OpcionesPendientes(k) hacer
        extender v con siguiente opción
        si SoluciónCompleta(v) entonces
            ProcesarSolución(v)
        sino
            si Completable (v) entonces
                VueltaAtras(v, k+1)
            fsi
        fmientras
    fsi
ffun
```

Algoritmo "adaptado" al problema

```
fun VueltaAtras (v: Secuencia, k: entero)
    {v es una secuencia k-prometedora}
    IniciarExploraciónNivel(k)
    mientras OpcionesPendientes(k) hacer
        extender v con siguiente opción
        si Completable (v) entonces
            si SoluciónCompleta(v) entonces
                ProcesarSolución(v)
            sino
                VueltaAtras(v, k+1)
            fsi
        fsi
    fmientras
ffun
```

Es <

Algoritmo principal

```
función Reinas(s:Vector[1..n] de entero, n,k: entero)
    s[k] ← 0
    mientras s[k] ≤ n hacer        ← n posibles teóricos sucesores
                                     (n filas posibles donde
                                     colocar la reinas)
        s[k] ← s[k] + 1
        si Completable(s,k) entonces        ← Descarta/poda los no factibles
            si k = n entonces        ← Se completa la solución si
                                       están las n reinas colocadas
                escribir(s)
            sino
                Reinas(s,n,k+1)
            fsi
        fsi
    fmientras
ffun
```

Esquema general

```
fun VueltaAtras (v: Secuencia, k: entero)
    {v es una secuencia k-prometedora}
    IniciarExploraciónNivel(k)
    mientras OpcionesPendientes(k) hacer
        extender v con siguiente opción
        si Completable (v) entonces
            si SoluciónCompleta(v) entonces
                ProcesarSolución(v)
            sino
                VueltaAtras(v, k+1)
            fsi
        fsi
    fmientras
ffun
```

- El algoritmo encuentra todas las soluciones posibles.

Llamada inicial:
Reinas([_,_,_,_,..,_],n,1)

Función Completable: comprueba que no haya ninguna reina en la misma fila ni diagonal (por construcción ya no está en la misma columna)

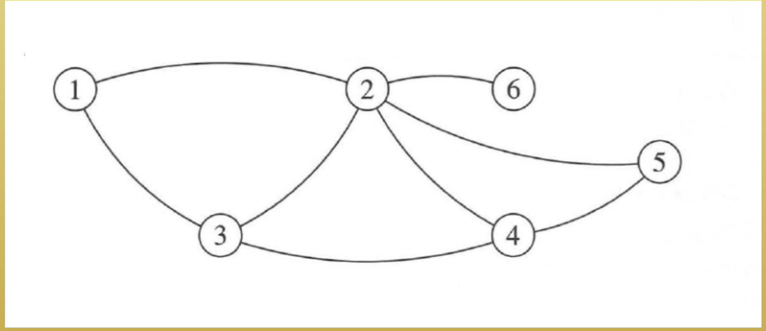
```
fun Completable(s:vector[1..n] de entero, k: entero): booleano
    var
        i: entero
    fvar
    para i ← 1 hasta k-1 hacer
        si s[i] = s[k] ∨ (abs(s[i]-s[k]) = abs(i-k)) entonces
            dev falso
        fsi
    fpara
    dev cierto
ffun
```

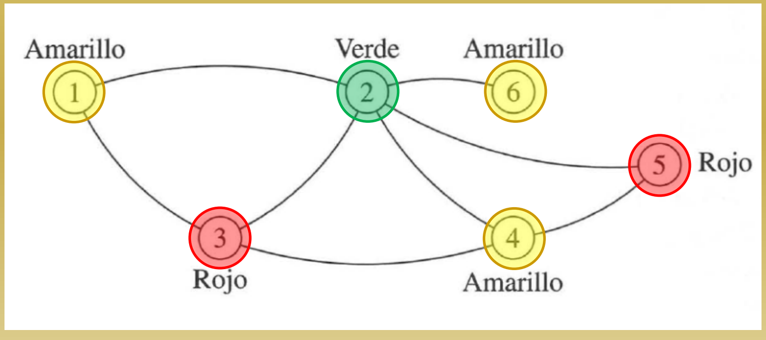
Coste (caso peor):

– Difícil de contabilizar poda. Una cota superior del coste es el nº máximo de nodos del árbol: n nodos=n niveles y cada nodo se expande en las n-k nodos restantes (descartamos 1 fila de cada vez porque no puede repetirse)→ n! nodos (realmente éste el nº de nodos en el último nivel) → O(n!). Aunque en la práctica es mucho mejor por la poda.

6.2 Coloreado de grafos

- **Datos de entrada:**
  - **Grafo conexo de n nodos** $\rightarrow$ matriz de adyacencia [1..*n*,1..*n*].
  - **m colores** distintos: 1..m
- **Objetivo:** asignar 1 color a cada nodo de forma que no haya 2 nodos adyacentes con el mismo color.
  - Es posible colorear cualquier mapa en el plano con 4 colores, pero se pueden necesitar menos.
- **Ejemplo:** grafo con 6 nodos y 3 colores (Rojo, Verde y Amarillo):

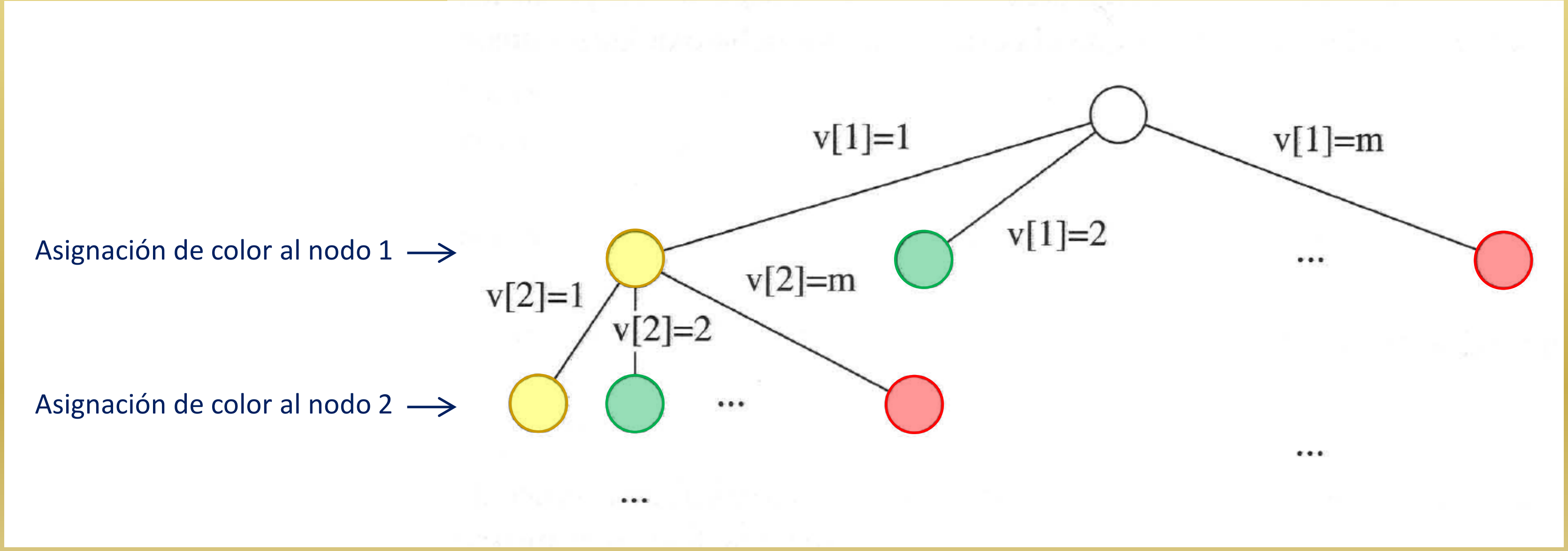




Posible solución

- **Descripción de la solución:**
  - **v: matriz [1..n] de entero**, donde **v[i]=número de color del nodo i-ésimo (∈[1..m])**.
  - Solución inicial: v=[_,_,…,_] o v=[0,0,…,0]

- **Árbol de exploración de las soluciones:**
  - **Cada nivel** → asignación de color a un nodo del grafo.
  - **Cada nodo** → tiene m hijos posibles.



Algoritmo principal

```
tipo Grafo = matriz[1..N,1..N] de entero
tipo Vector = matriz[0..N] de entero
fun ColoreaGrafo (g:Grafo, m: entero, k: entero, v: Vector, exito:booleano
    {v es un vector k-prometedor}
    v[k] ← 0
    exito ← falso
    mientras v[k] ≤ m ∧ ¬ exito hacer
        v[k] ← v[k] + 1
        si Completable (v) entonces
            si k = N entonces
                Procesar(v)
                exito ← cierto
            sino
                ColoreaGrafo(g,m,k+1,v,exito)
            fsi
        fsi
    fmientras
ffun
```

Voy a asignar en el nodo k-ésimo (los anteriores ya están asignados)

m posibles teóricos sucesores (m colores posibles para el nodo k)

Descarta/poda los no factibles

Se completa la solución si están las n nodos coloreados

Llamada inicial:
ColoreaGrafo(g,m,1,v,falso)

Esquema general adaptado al problema

```
fun VueltaAtras (v: Secuencia, k: entero)
    {v es una secuencia k-prometedora}
    IniciarExploraciónNivel(k)
    mientras OpcionesPendientes(k) hacer
        extender v con siguiente opción
        si Completable (v) entonces
            si SoluciónCompleta(v) entonces
                ProcesarSolución(v)
            sino
                VueltaAtras(v, k+1)
            fsi
        fsi
    fmientras
ffun
```

```
fun Completable(g:Grafo, v:Vector, k: entero): booleano
    i: entero
    para i ← 1 hasta k-1 hacer
        si g[k,i] = 1 ∧ v[k] = v[i] entonces
            dev falso
        fsi
    fpara
    dev cierto
ffun
```

Comprueba que no hay 2 nodos adyacentes con el mismo color en lo asignado hasta ahora.

Coste (caso peor):

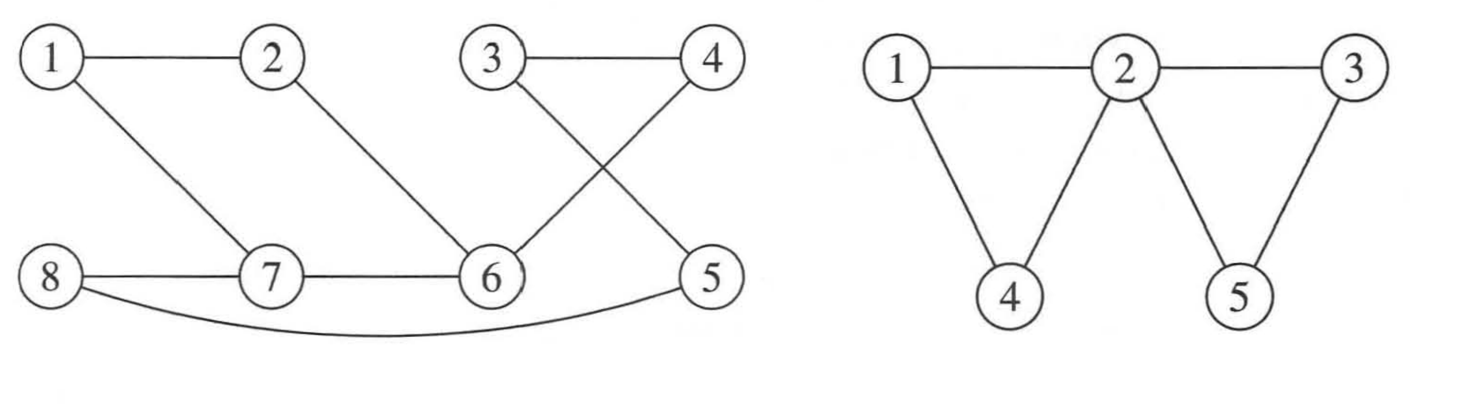
– Tamaño del árbol en el orden de m^n (realmente es el número de hojas del árbol) → cota superior del coste $O(m^n)$.

6.3 Ciclos Hamiltonianos

- Tenemos un grafo no dirigido de n nodos → g: matriz de adyacencia [1..*n*,1..*n*].
- **Objetivo:** encontrar un ciclo Hamiltoniano (camino que pasa una sola vez por cada nodo y terminan en el nodo inicial)

  → Si el grafo es valorado y busco un ciclo Hamiltoniano de coste mínimo: problema del *Viajante de Comercio* (algoritmos de Ramificación y Poda)

  → No está garantizado que haya solución.
- **Ejemplos:**

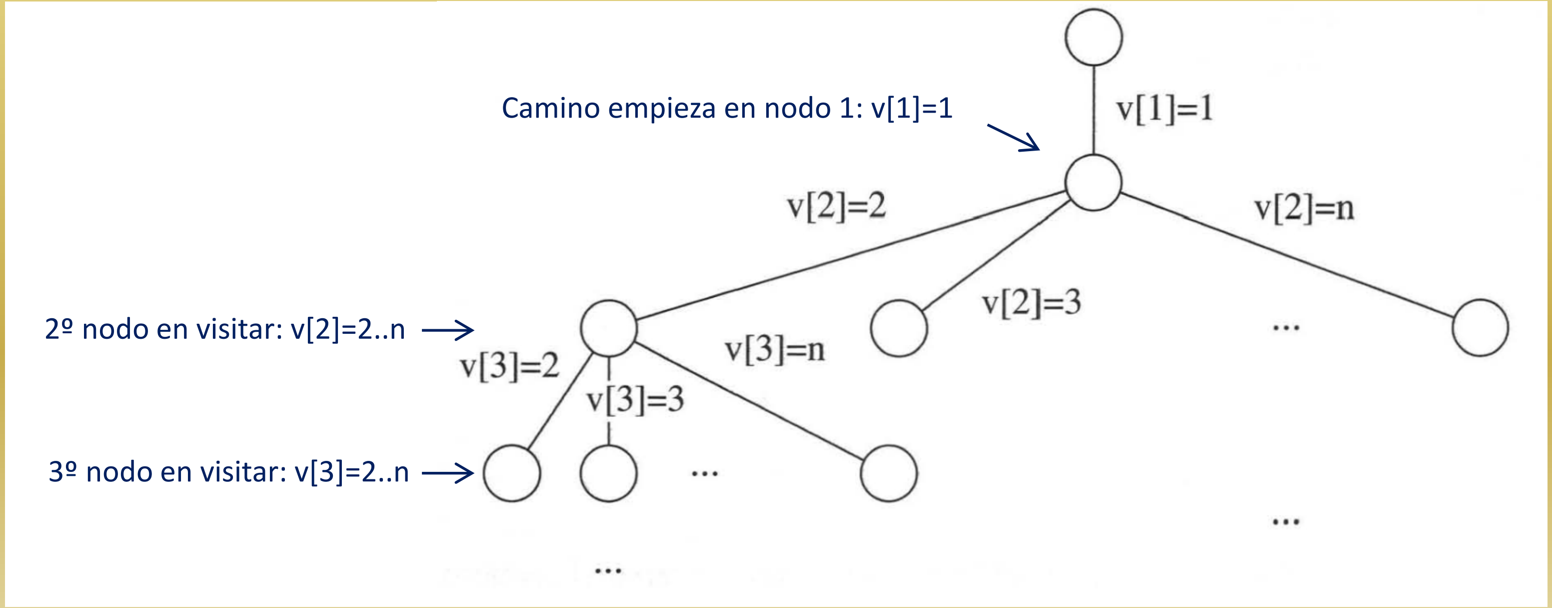


ciclo Hamiltoniano 1-2-6-4-3-5-8-7-1

ningún ciclo Hamiltoniano

- **Descripción de la solución:**
  - **v: matriz [1..n] de entero → contiene el orden de recorrido de los nodos**.
  - Solución inicial: v=[1,_,…,_] o v=[1,0,…,0] → empezamos en nodo 1
  - Usamos **incluidos: matriz [1..n] de booleano →** ***incluido[i]=V≡nodo i visitado***

- **Árbol de exploración de las soluciones:**
  - **Cada nivel** → añado un nodo al recorrido.
  - **Cada nodo** → tiene tantos hijos posibles como sucesores del nodo anterior (como máximo n).



Algoritmo principal

```
tipo Grafo = matriz[1..N,1..N] de entero
tipo Vector = matriz[1..N] de entero
tipo VectorB = matriz[1..N] de booleano
fun CiclosHamiltoniano (g:Grafo, k: entero, v: Vector, incluidos: VectorB)
  var
    i: entero
  fvar
  para i = 2 hasta n hacer
    si g[v[k-1],i] ∧ ¬ incluidos[i] entonces
      v[k] ← i
      incluidos[i] ← cierto
      si k = n entonces
        { se comprueba que se cierra el ciclo }
        si g[v[n],1] entonces
          PresentarSolución(v)
        fsi
      sino
        CiclosHamiltoniano(g,k+1,v, incluidos)
      fsi
      incluidos[i] ← falso
    fsi
  fpara
ffun
```

n-1 posibles sucesores (todos los nodos menos el de salida)

Comprueba si se puede extender la solución Completable(v): si hay camino del nodo anterior (k-1) al nodo i y el nodo i no ha sido visitado

Siguiente nodo a visitar: el i

Solución(v) requiere 2 condiciones

Marca el nodo i como no visitado para la siguiente (se cambiará v[k]=i por v[k]=i+1 si es posible)

***Llamada externa:** inicializa variables*

```
fun PresentarHamiltonianos(g)
  var
    v: Vector
    incluidos: VectorB
    i: entero
  fvar
  v[1] ← 1
  incluidos[1] ← cierto
  para i = 2 hasta n hacer
    incluidos[i] ← falso
  fpara
  CiclosHamiltoniano(g,2,v,incluidos)
ffun
```

Coste (caso peor):

– Tamaño del árbol: n niveles y cada uno se expande en n-k nodos → cota superior del coste O(n!)

6.4 Subconjuntos de suma dada

- **Datos:**
  - Conjunto *A* de *n* números enteros no repetidos.
  - Valores de *m* y *C*, con *m*<*n*.
- **Objetivo:** encontrar todos los subconjuntos de *A* con *m* elementos y cuyos valores sumen exactamente *C*.
- **Posibilidades:**
  a) Representar la solución como un vector de enteros: chequear todas las permutaciones de m elementos $\rightarrow$ O(n(n-1)..(n-m+1)) [Si m$\rightarrow$n O(n!)].
  b) Representar la solución como un vector de n booleanos que representan si se asigna o no un cada elemento del vector inicial $\rightarrow$ O(2^n).
- **Descripción de la solución (caso b):**
  - **v: matriz [1..n] de bool** $\rightarrow$ v[i]=V $\leftrightarrow$ elemento i de A incluido en la solución.

    (solución inicial: v=[F,F,,…,F])
  - **k: nº de bit chequeado de v**
  - **Sumandos:** nº de elementos incluidos en la solución parcial ($\leq m$)
  - **Suma:** suma de los valores de los elementos incluidos ($\leq C$)

Algoritmo principal

```
tipo Vector = matriz[1..N] de entero
tipo VectorB = matriz[1..N] de booleano
fun SubconjuntosSumaDada(datos: Vector, k: entero, v: VectorB,
                  sumandos: entero, suma: entero)
   var
      i: entero
   fvar
   si sumandos = m entonces
      si suma = C entonces
         PresentarSolución(v)
      fsi
   sino
      si k < n entonces
         v[k] ← falso
         SubconjuntosSumaDada(datos,k+1,v, sumandos, suma)
         v[k] ← cierto
         si suma + datos[k] ≤ C entonces
            suma ← suma + datos[k]
            sumandos ← sumandos + 1
            SubconjuntosSumaDada(datos,k+1,v, sumandos, suma)
         fsi
      fsi
   fsi
ffun
```

Compruebo solución al principio: v es solución si tiene m sumandos y suman C

Quedan bits por asignar si k<n

Cada nodo tiene 2 posibles sucesores v[k]=V y v[k]=F

Completable (faltaría por comprobar si sumandos≤m)

Esquema general adaptado al problema

```
fun VueltaAtras (v: Secuencia, k: entero)
   {v es una secuencia k-prometedora}
   si SoluciónCompleta(v) entonces
      ProcesarSolución(v)
   sino
      IniciarExploraciónNivel(k)
      mientras OpcionesPendientes(k) hacer
         extender v con siguiente opción
         si Completable (v) entonces
            VueltaAtras(v, k+1)
         fsi
      fmientras
   fsi
ffun
```

Llamada inicial:
SubconjuntosSumaDada(datos,1,[F,F,..F],0,0)

Coste (caso peor):

– Árbol de n niveles y como máximo 2 hijos por nodo (árbol binario) → cota superior del coste $O(2^n)$.

6.5 Reparto equitativo de activos

- **Datos:**
  - n activos de una sociedad, con valores=$\{v_1,v_2,..,v_n\}$.
  - 2 socios.
- **Objetivo:** repartir los activos a medias $\rightarrow$ 2 subconjuntos disjuntos cada uno con el mismo valor entero.
- **Ejemplo:** n=10 activos

Activo	1	2	3	4	5	6	7	8	9	10
Valor	10	9	5	3	3	2	2	2	2	2

¿Voraz? Trato de asignar al primer socio el mayor activo de los que quedan mientras no se supere la mitad del valor total de los activos $\rightarrow$ en el ejemplo no sería posible completar el reparto. **No aplicable esquema voraz**.

Socio1	Socio2
10	5
9	3
	3
	2
	2
	2
	2
	2

Voraz: no equitativo

Socio1	Socio2
10	9
2	5
2	3
2	3
2	
2	

Una solución (reparto ya equitativo)

- **Descripción de la solución:**
  - **v: matriz [1..n] de entero** $\rightarrow$ v[i]=1 ó 2 (árbol binario): indica si el activo i-ésimo va al socio 1 ó 2.
  - **k:** nº de activos ya repartidos.
  - **suma1, suma2:** valor de los activos del socio 1 y 2, respectivamente.
  - **sumaTotal:** suma de los valores de todos los activos.

Algoritmo principal

```
tipo Vector = matriz[0..N] de entero
fun DividirSociedad(x: Vector, suma1, suma2, sumaTotal, k: entero, v: Vector)
    {v es un vector k-prometedor}
    si k = N entonces
        si suma1 = suma2 entonces
            Procesar(v)
        fsi
    sino
        v[k+1] ← 1
        si Completable(x, suma1, sumaTotal, k+1) entonces
            suma1 ← suma1 + x[k+1]
            DividirSociedad(x, suma1, suma2, sumaTotal, k+1, v)
        fsi
        v[k+1] ← 2
        si Completable(x, suma2, sumaTotal, k+1) entonces
            suma2 ← suma2 + x[k+1]
            DividirSociedad(x, suma1, suma2, sumaTotal, k+1, v)
        fsi
    fsi
ffun
```

Compruebo solución al principio: v es solución si ya no quedan activos por repartir y lo que se lleva cada uno suma lo mismo

Comprueba si esa asignación es factible (el valor del socio 1 no supera sumaTotal/2)

Asigna el activo k+1 al socio nº 1

Asigna el activo k+1 al socio nº 2

Cada nodo tiene 2 posibles sucesores, asignar al socio 1 o al 2

```
fun Completable(x:Vector, sumaParcial, sumaTotal, k: entero): booleano
    si sumaParcial + x[k] ≤ sumaTotal div 2 entonces
        dev falso
    sino
        dev cierto
    fsi
ffun
```

***Llamada externa:** inicializa variables*

```
fun ResolverSeparacionSocios (x:Vector)
    var
        i, suma1, suma2, sumaTotal: entero
        v: Vector
    fvar
    sumaTotal ← 0
    suma1 ← 0
    suma2 ← 0
    para i ← 1 hasta N hacer
        sumaTotal ← sumaTotal + x[i]
    fpara
    si sumaTotal mod 2 = 0 entonces
        DividirSociedad(x, v, 1, suma1, suma2, sumaTotal)
    fsi
ffun
```

Coste (caso peor):

– Árbol de n niveles y como máximo 2 hijos por nodo (árbol binario, según asigne al socio 1 o al 2) → cota superior del coste $O(2^n)$.

6.6 El robot en busca del tornillo

- **Datos de entrada:**
  - Edificio cuyas habitaciones se representan en un tablero de $n \times m$ casillas con tres posibles valores:
    - L: “paso libre”, E: “paso estrecho” o T: “tornillo”.
  - Hay un robot que, en cada punto, puede moverse a la casilla Norte, Sur, Este, Oeste si el paso está libre (por las casillas de paso estrecho no puede moverse).
- **Objetivo:** saliendo el robot de la casilla (1,1) $\rightarrow$ encontrar la casilla ocupada por el tornillo.
  - El algoritmo debe devolver la secuencia de casillas que componen el camino de regreso desde la casilla ocupada por el tornillo hasta la casilla (1,1)
- **Esquema a utilizar:**
  - No hay ninguna función de selección adecuada que evite deshacer decisiones tomadas.
  - Si nos pidiesen camino más corto (o grafo infinito) $\rightarrow$ búsqueda en anchura
  - No es necesario devolver el camino más corto, sino el camino lo antes posible $\rightarrow$ búsqueda en profundidad.
    - El robot puede llegar a una casilla en la que no pueda avanzar más $\rightarrow$ habrá que deshacer decisiones tomadas: vuelta atrás adecuado (aunque es totalmente equivalente a la búsqueda en profundidad).
    - Usaremos el esquema vuelta atrás que se detiene al encontrar la primera solución y devuelve la secuencia de casillas desde el tornillo hasta la salida.
    - Marcaremos los nodos ya visitados (espacio de exploración $\rightarrow$ grafo con ciclos).

Algoritmo principal (k no se utiliza explícitamente)

```
tipo TEdificio = matriz[1..LARGO, 1..ANCHO] de caracter
tipo TEdificioB = matriz[1..LARGO, 1..ANCHO] de booleano        ← Para marcar las casillas ya visitadas
tipo TCasilla =
    registro
        x,y: entero                                              } Utilizado para indicar una posición en el tablero
    fregistro
tipo TListaCasillas = Lista de TCasilla      ← Lista de casillas (utilizada por solución y compleciones)
fun BuscaTornillo (edificio: TEdificio, casilla: TCasilla,       ← Casilla actual
        exploradas: TEdificioB, solucion: TListaCasillas, exito: booleano)
    exploradas[casilla.x, casilla.y] ← cierto      ← Marco la casilla actual como visitada
    si edificio[casilla.x, casilla.y] = T entonces
        solución ← CrearLista()
        solución ← Añadir(solución, casilla)       } Se crea una lista para incluir todas las casillas
        exito ← cierto                               desde la solución hasta la salida
    sino
        hijos ← Caminos(edificio, casilla)   ← Función compleciones, calcula los posibles sucesores de la casilla
                                               actual (4 como máximo: ir al Norte, Sur, Este y Oeste). Los
                                               devuelve como una lista de casillas
        exito ← falso
        mientras ¬ exito ∧ ¬ ListaVacia?(hijos) hacer
            hijo ← Primero (hijos)
            si ¬ exploradas [hijo.x, hijo.y] entonces
                BuscaTornillo (edificio,hijo,exploradas,solución,exito)
            fsi
        fmientras
        si exito entonces
            solución ← Añadir(solución, casilla)
        fsi
    fsi
ffun
```

Algoritmo Caminos/Compleciones

```
fun Caminos (edificio: TEdificio, casilla: TCasilla): TListaCasillas
    var
        hijos: TListaCasillas
        casilla_aux: TCasilla
    fvar
    hijos ← CrearLista()
    si casilla.x+1 ≤ LARGO entonces
        si edificio[casilla.x+1,casilla.y] ≠ E entonces
            casilla_aux.x ← casilla.x+1
            casilla_aux.y ← casilla.y
            hijos ← Añadir (solución, casilla_aux)
        fsi
    fsi
    si casilla.x-1 ≥ 1 entonces
        si edificio[casilla.x-1,casilla.y] ≠ E entonces
            casilla_aux.x ← casilla.x-1
            casilla_aux.y ← casilla.y
            hijos ← Añadir (solución, casilla_aux)
        fsi
    fsi
    si casilla.y+1 ≤ ANCHO entonces
        si edificio[casilla.x,casilla.y+1] ≠ E entonces
            casilla_aux.x ← casilla.x
            casilla_aux.y ← casilla.y+1
            hijos ← Añadir (solución, casilla_aux)
        fsi
    fsi
    si casilla.y-1 ≥ 1 entonces
        si edificio[casilla.x,casilla.y-1] ≠ E entonces
            casilla_aux.x ← casilla.x
            casilla_aux.y ← casilla.y-1
            hijos ← Añadir (solución, casilla_aux)
        fsi
    fsi
    dev hijos
ffun
```

Devuelve una lista de casillas (pares de coordenadas) posibles sucesores de la casilla actual

Condiciones de poda

Movimiento a la derecha (Este)

Movimiento a la izquierda (Oeste)

Movimiento hacia abajo (Sur) (suponiendo que la coordenada -1,1- está en la esquina superior izquierda de la matriz)

Movimiento hacia arriba (Norte)

Coste (caso peor):

– Árbol de n^2 niveles (=nº casillas matriz) y como máximo 4 hijos por nodo (4 posibles movimientos) → cota superior del coste $O(4^{n^2})$

6.7 Asignación de cursos en una escuela

- **Datos de entrada:**
  - Escuela en donde se impartirán *n* cursos en *n* aulas y *n* profesores.
  - Funciones booleanas conocidas:
    - *válida(aula,curso)* $\rightarrow$ indicará si una determinada aula tiene capacidad para un curso.
    - *especialidad(prof,curso)* $\rightarrow$ indicará si un determinado profesor está preparado para un curso.

- **Objetivo:** diseñar un algoritmo para encontrar una forma de asignar a cada curso un aula y un profesor apropiados.

- **Descripción de la solución de los datos utilizados:**
  - **tipo** Tcurso = **registro**

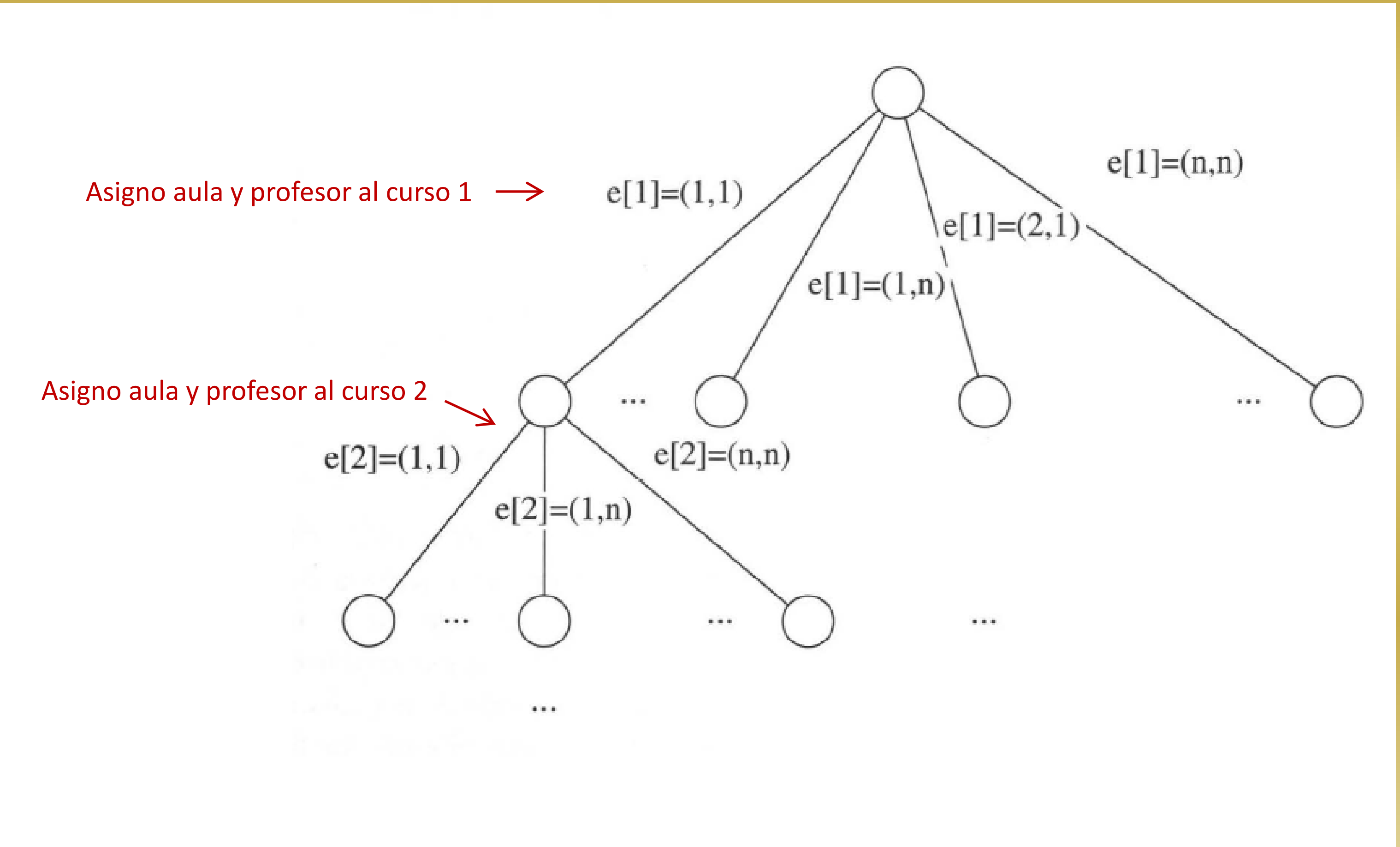
    aula: entero

    profesor: entero

    fregistro *(contendrá la asignación a un curso de un aula y un profesor)*
  - **v: matriz [1..n] de Tcurso** *(v[i] contiene la asignación para el curso i-ésimo)*
  - **k:** nº de cursos ya asignados.
  - **asigAula,asigProf: matriz [1..n] de booleano** *(se utilizan para ir marcando las aulas y los profesores ya utilizados/asignados)*

Árbol de búsqueda:

- **Cada nivel** → asigno aula y profesor a un nuevo curso.
- **Cada nodo** → tiene n^2 hijos que resultan de la asignación a un curso de n posibles aulas y profesores. Recorro desde (1,1), (1,2), .., (1,n), (2,1), (2,2), .. (n,n), siendo (aula,profesor)



Algoritmo principal

```
tipo TCurso =
    registro
        aula: entero
        profesor: entero
    fregistro
tipo TEscuela = matriz[0..n] de TCurso
tipo TVectorB = matriz[0..n] de booleano
fun CursosEscuela (escuela: TEscuela, k: enteros, asigAula: TVectorB,      ← Asigna al aula k el curso y el profesor
            asigProf: TVectorB, exito: booleano)
    var
        es_solucion: booleano
        aula: entero
        prof: entero
    fvar
    aula ← 1     ← Comienza probando aula 1 hasta aula n
    mientras aula ≤ n ∧ ¬ exito hacer
        si ¬ asigAula[aula] ∧ válida(aula,k) entonces     ← Condiciones de poda para aula (no asignada y de tamaño suficiente)
            prof ← 1     ← Comienza probando profesor 1 hasta profesor n
            mientras prof ≤ n ∧ ¬ exito hacer
                si ¬ asigProf[prof] ∧ especialidad(prof,k) entonces     ← Condiciones de poda para profesor (no asignado y de la especialidad adecuada)
                    escuela[k].aula ← aula
                    escuela[k].prof ← prof
                    asigAula[aula] ← cierto
                    asigProf[prof] ← cierto
                    si k = n entonces
                        PresentarSolucion(escuela)     ← Solución completa, asigno éxito a cierto para terminar la búsqueda
                        exito ← cierto
                    sino
                        CursosEscuela(escuela,k+1,asigAula,asigProf,exito)
                    fsi
                    asigAula[aula] ← falso
                    asigProf[prof] ← falso
                fsi
                prof ← prof + 1
            fmientras
        fsi
        aula ← aula + 1
    fmientras
ffun
```

Llamada externa: *inicializa variables*

```
fun HorarioValido(n:entero)
    var
        asigAula: TvectorB
        asigprof: TvectorB
    fvar
    para i = 1 hasta n hacer
        asigAula ← falso
        asigProf ← falso
    fpara
    exito ← falso
    CursosEscuela(escuela,1,asigAula,asigProf,exito)
ffun
```

Coste (caso peor):

- Árbol de n niveles (=nº casillas matriz) y como máximo n^2 hijos por nodo ($n \times n$ posibles asignaciones para 1 curso) → cota superior del coste $O\left(\left[n^2\right]^n\right) = O(n^{2n})$