PREDA - UNED

Programación y Estructuras de Datos Avanzadas

Capítulo 6

Vuelta atrás

Planteamiento

- Aplicación
  - problemas en los que sólo podemos recurrir a una búsqueda exhaustiva, recorriendo el espacio de todas las posibles soluciones hasta encontrar una o hasta que hayamos explorado todas las opciones
- Debido al coste
  - aplicar el conocimiento disponible sobre el problema para abandonar un camino no válido lo antes posible
  - pensar antes si es posible usar un algoritmo voraz
- Recorrido del grafo representa el espacio de búsqueda
  - si el grafo es infinito (o muy grande) se trabaja con un grafo implícito, porque no tiene sentido que el programa lo construya para luego aplicar las técnicas de búsqueda
  - Recorrido en profundidad podando las ramas no válidas
  - Vamos generando una solución parcial (secuencia k-prometedora) y si encontramos un “nodo de fallo” retrocedemos y exploramos otras ramas

Elementos

- IniciarExploraciónNivel():
  recoge todas las opciones posibles en que se puede extender la solución k-prometedora
- OpcionesPendientes():
  comprueba que quedan opciones por explorar en el nivel
- SoluciónCompleta():
  comprueba que se haya completado una solución al problema
- ProcesarSolución():
  representa las operaciones que se quieran realizar con la solución, como imprimirla o devolverla al punto de llamada
- Completable():
  comprueba que la solución k-prometedora se puede extender con la opción elegida cumpliendo las restricciones del problema hasta llegar a completar una solución

Esquema

```
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
        fsi
    fmientras
ffun
```

Esquema

```
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
        fsi
    fmientras
ffun
```

Esquema

```
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
        fsi
    fmientras
ffun
```

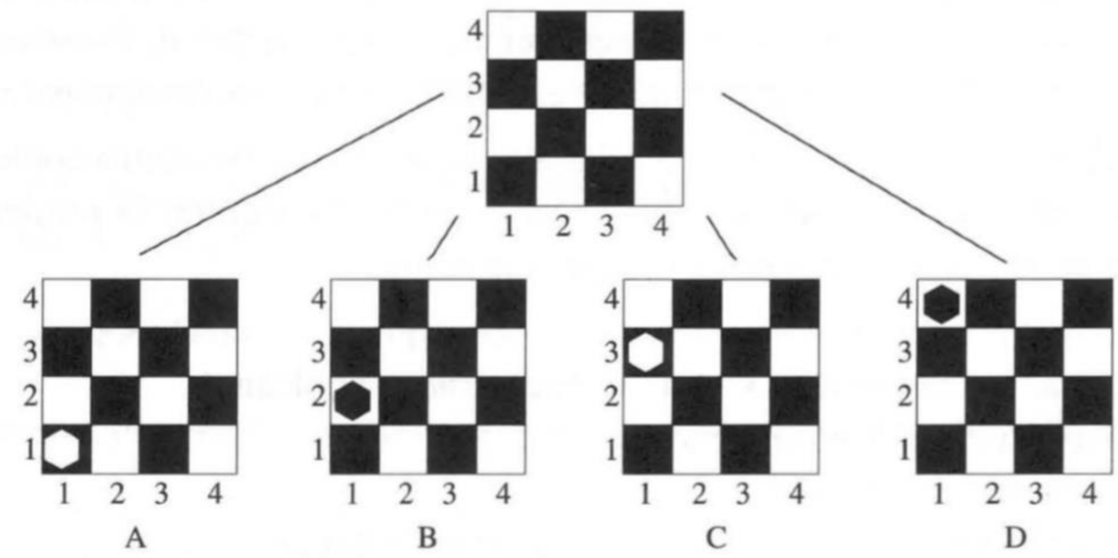
Coste

- El coste del caso peor es del orden del tamaño del espacio de búsqueda
- Las funciones de poda que utilicemos reducen el coste, aunque muchas veces no es posible saber cuanto (depende de los datos), así que se da una cota superior

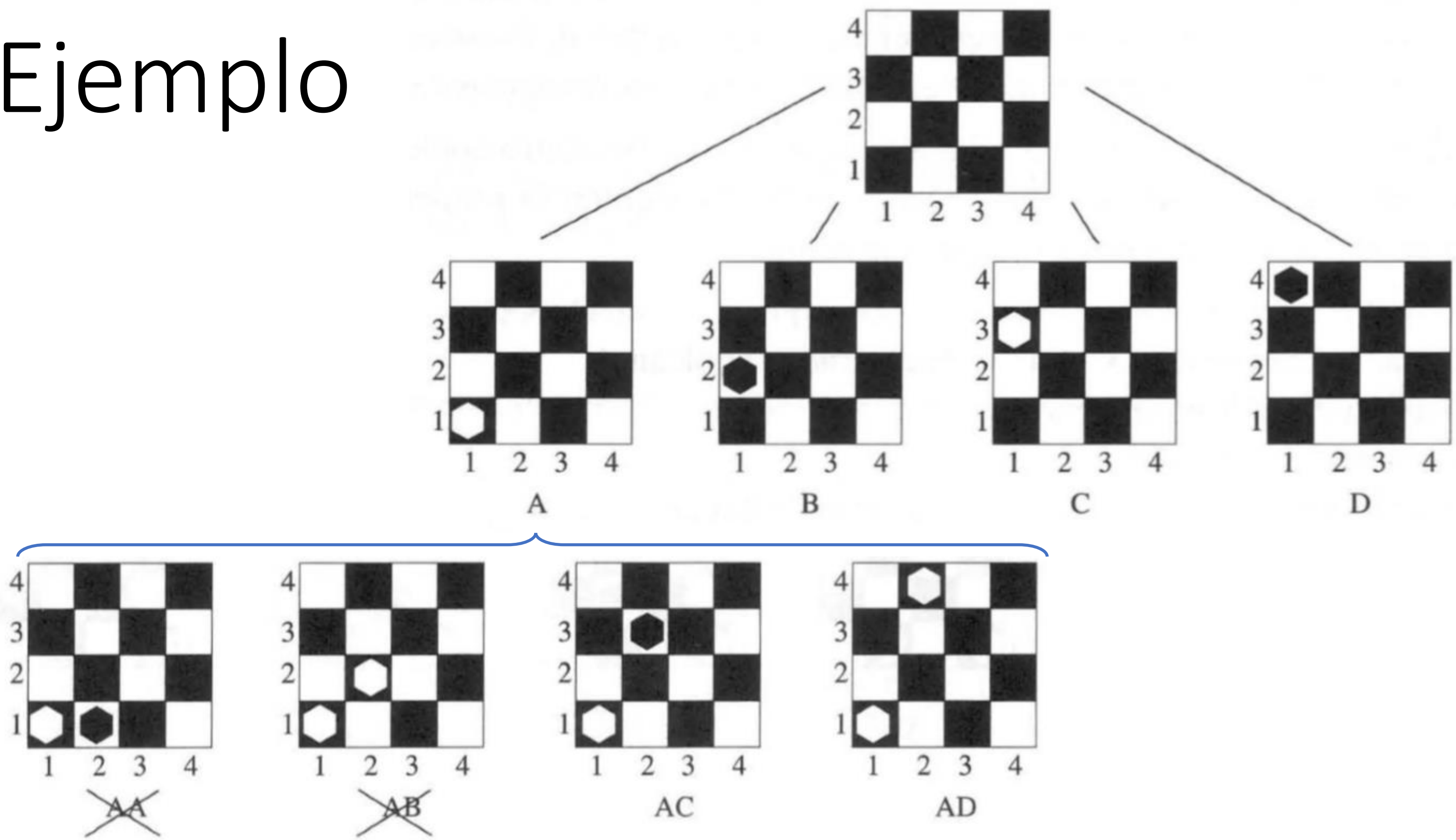
Ejemplo

- Colocar N reinas en un tablero de ajedrez N x N de forma que ninguna reina ataque a otra
- La columna de la reina c_i va a ser i, y la solución es una asignación de la fila que corresponde a cada reina (un vector donde f_i es la fila asignada a c_i)

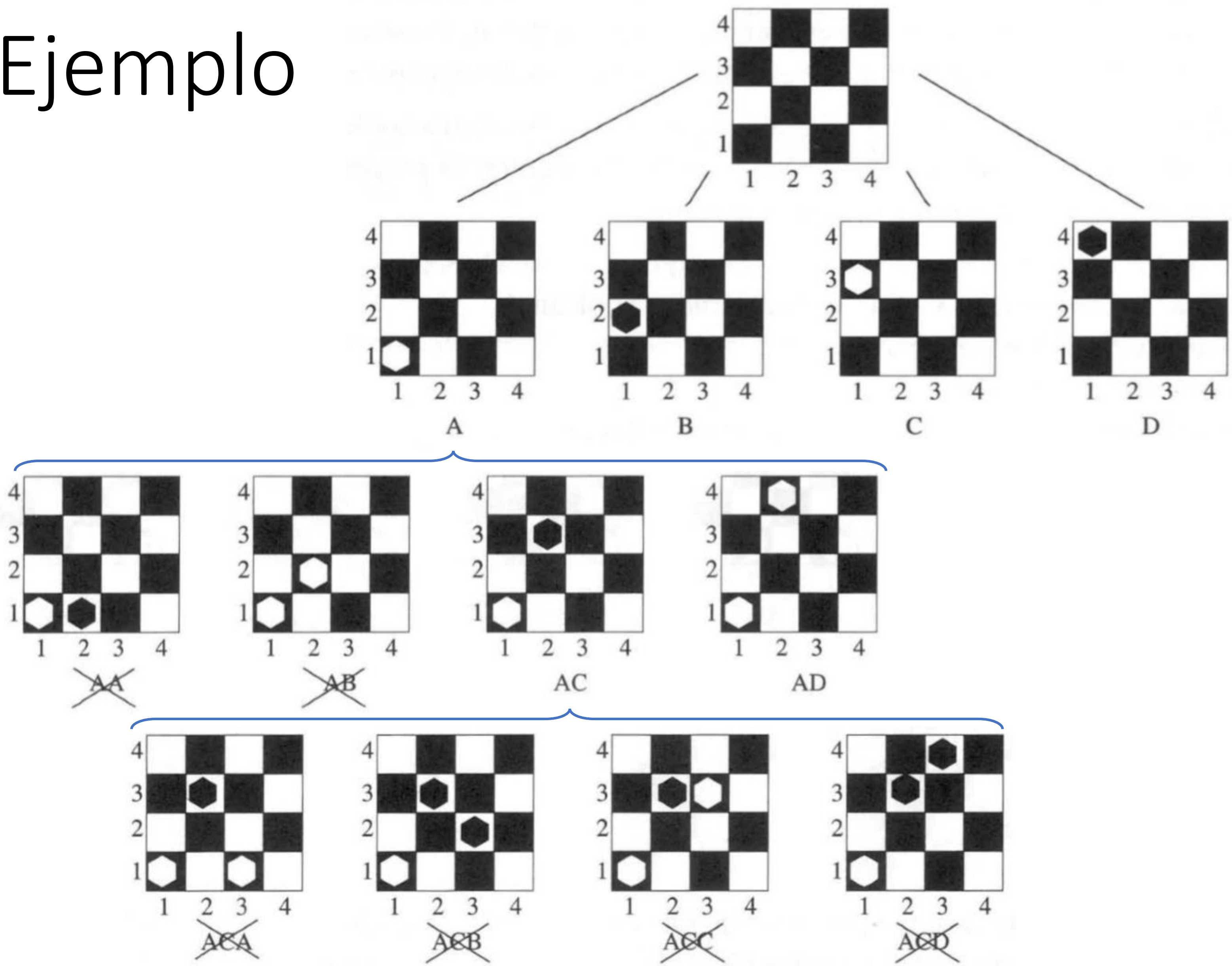
Ejemplo



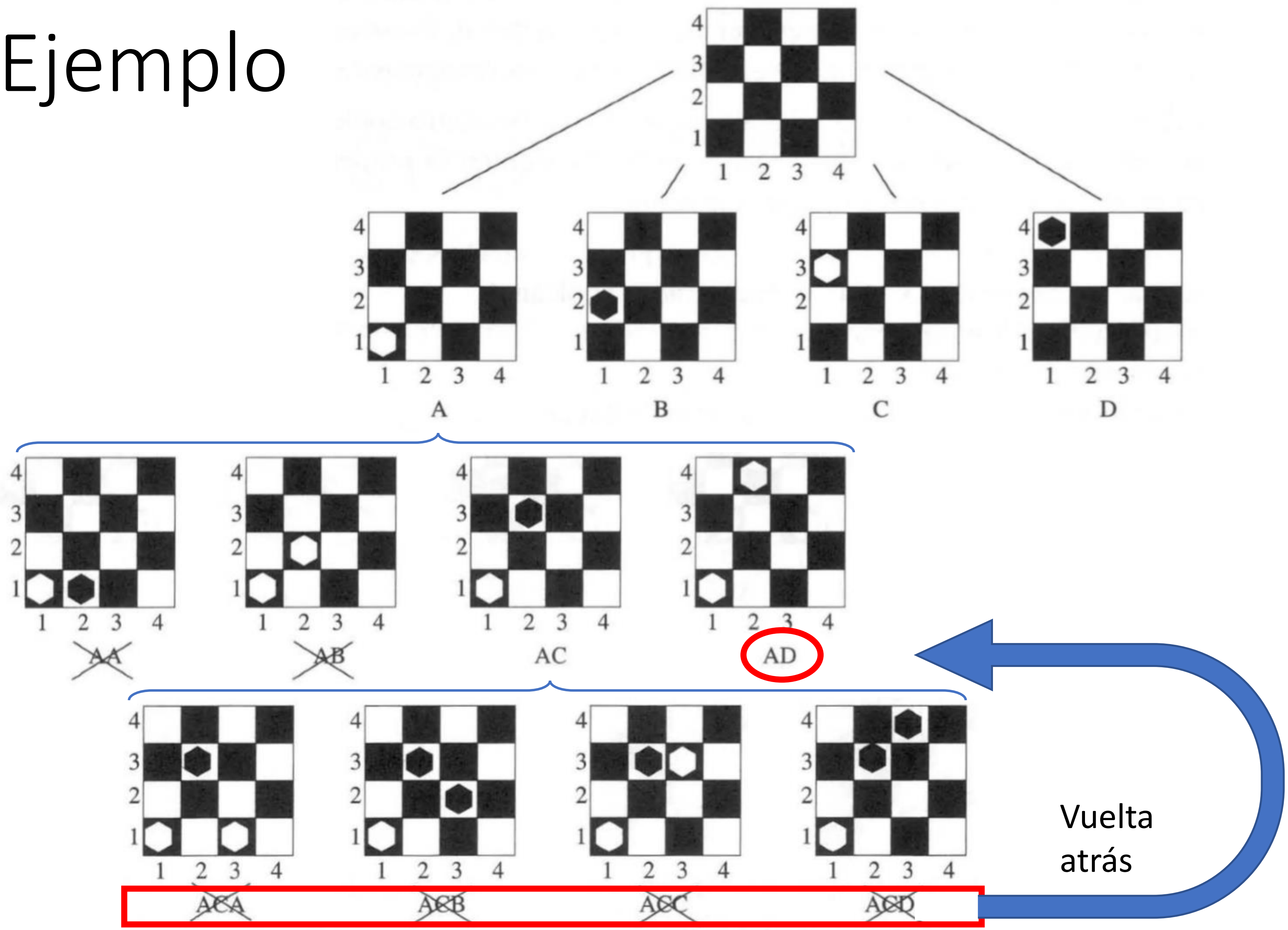
Ejemplo



Ejemplo

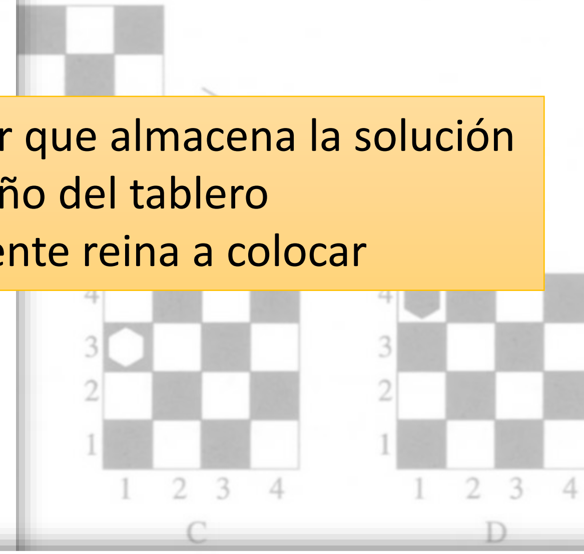


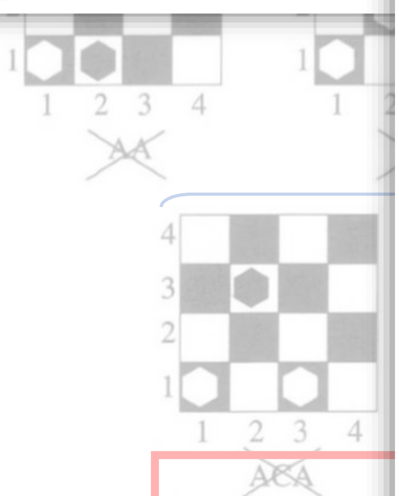
Ejemplo



```
función Reinas(s:Vector[1..n] de entero, n,k: entero)
    s[k] ← 0
    mientras s[k] ≤ n hacer
        s[k] ← s[k] + 1
        si Completable(s,k) entonces
            si k = n entonces
                escribir(s)
            sino
                Reinas(s,n,k+1)
            fsi
        fsi
    fmientras
ffun
```

- **s**: vector que almacena la solución
- **n**: tamaño del tablero
- **k**: siguiente reina a colocar



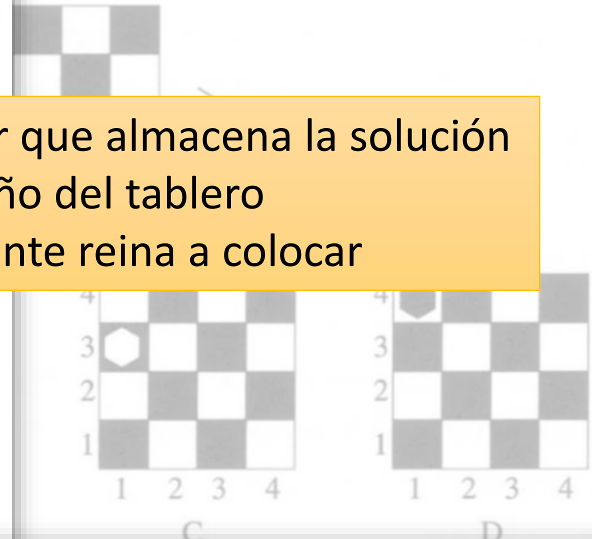


```
fun Completable(s:vector[1..n] de entero, k: entero): booleano
    var
        i: entero
    fvar
    para i ← 1 hasta k-1 hacer
        si s[i] = s[k] ∨ (abs(s[i]-s[k]) = abs(i-k)) entonces
            dev falso
        fsi
    fpara
    dev cierto
ffun
```

Comprueba si una extensión de la solución parcial anterior es completable (no tiene reinas en la misma fila, ni en la misma diagonal)

```
función Reinas(s:Vector[1..n] de entero, n,k: entero)
    s[k] ← 0
    mientras s[k] ≤ n hacer
        s[k] ← s[k] + 1
        si Completable(s,k) entonces
            si k = n entonces
                escribir(s)
            sino
                Reinas(s,n,k+1)
            fsi
        fsi
    fmientras
ffun
```

- **s**: vector que almacena la solución
- **n**: tamaño del tablero
- **k**: siguiente reina a colocar



```
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
```

Comprueba si una extensión de la solución parcial anterior es completable (no tiene reinas en la misma fila, ni en la misma diagonal)

Cota superior de coste: tamaño del árbol = n^n, pero considerando la restricción de que no hay reinas en la misma fila, la cota superior del coste se reduce a n!

Ejercicio de examen

6. En relación al esquema algorítmico de vuelta atrás, ¿cuál de las siguientes afirmaciones es **cierta**?:

   (a) Implementa un recorrido en amplitud de forma recursiva sobre el grafo implícito del problema.
   (b) Aun existiendo una solución al problema, puede darse el caso de que el esquema de vuelta atrás no la encuentre.
   (c) Siempre es más eficiente que el esquema voraz, de forma que, si ambos son aplicables, nos decantaremos por utilizar el esquema de vuelta atrás.
   (d) Todas las afirmaciones anteriores son falsas.

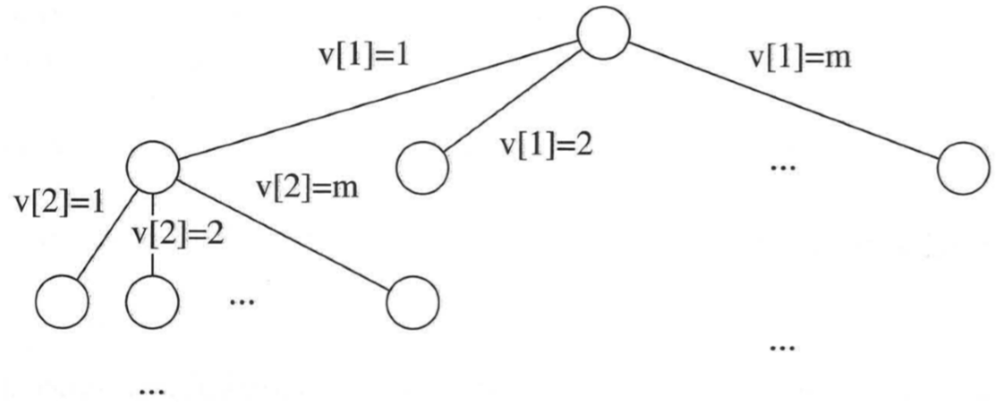
Ejercicio de examen

6. En relación al esquema algorítmico de vuelta atrás, ¿cuál de las siguientes afirmaciones es **cierta**?:

   (a) Implementa un recorrido en amplitud de forma recursiva sobre el grafo implícito del problema.
   (b) Aun existiendo una solución al problema, puede darse el caso de que el esquema de vuelta atrás no la encuentre.
   (c) Siempre es más eficiente que el esquema voraz, de forma que, si ambos son aplicables, nos decantaremos por utilizar el esquema de vuelta atrás.
   (d) Todas las afirmaciones anteriores son falsas.

Coloreado de grafos

- Objetivo: asignar uno de m colores a cada vértice de un grafo conexo sin que haya dos vértices adyacentes con el mismo color



- **v**: vector de soluciones donde v[i]=j es el color j asignado al nodo i
- **n**: número de nodos
- **m**: número de colores

```
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+1] ← 0
    exito ← falso
    mientras v[k+1] <  m ∧ ¬ exito hacer
        v[k+1] ← v[k+1] + 1
        si Completable(g, v, k) entonces
            si k = N entonces
                Procesar(v)
                exito ← cierto
            sino
                ColoreaGrafo(g,m,k+1,v,exito)
            fsi
        fsi
    fmientras
ffun
```

Usar v[k] en lugar de v[k+1]

da vértice
rtices

ctor de soluciones
le v[i]=j es el color j
ado al nodo i

```
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
```

```
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+1] ← 0
    exito ← falso
    mientras v[k+1] <  m ∧ ¬ exito hacer
        v[k+1] ← v[k+1] + 1
        si  Completable(g, v, k) entonces
            si k = N entonces
                Procesar(v)
                exito ← cierto
            sino
                ColoreaGrafo(g,m,k+1,v,exito)
            fsi
        fsi
    fmientras
ffun
```

Cota superior de coste: $O(n \cdot m^n)$
Tamaño del árbol multiplicado por el coste n de Completable (se llama para cada nodo)

Usar v[k] en lugar de v[k+1]

ctor de soluciones
le v[i]=j es el color j
ado al nodo i

```
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
```

Ejercicio de examen

6. En el problema de colorear un grafo con *n* nodos utilizando *m* colores, de manera que no haya dos vértices adyacentes que tengan el mismo color, una cota superior ajustada del coste de encontrar una solución utilizando un esquema adecuado es del orden de:

   (a) $O(n^{m/2} \log n)$.
   (b) $O(m \log n)$.
   (c) $O(n^m)$.
   (d) $O(nm^n)$.

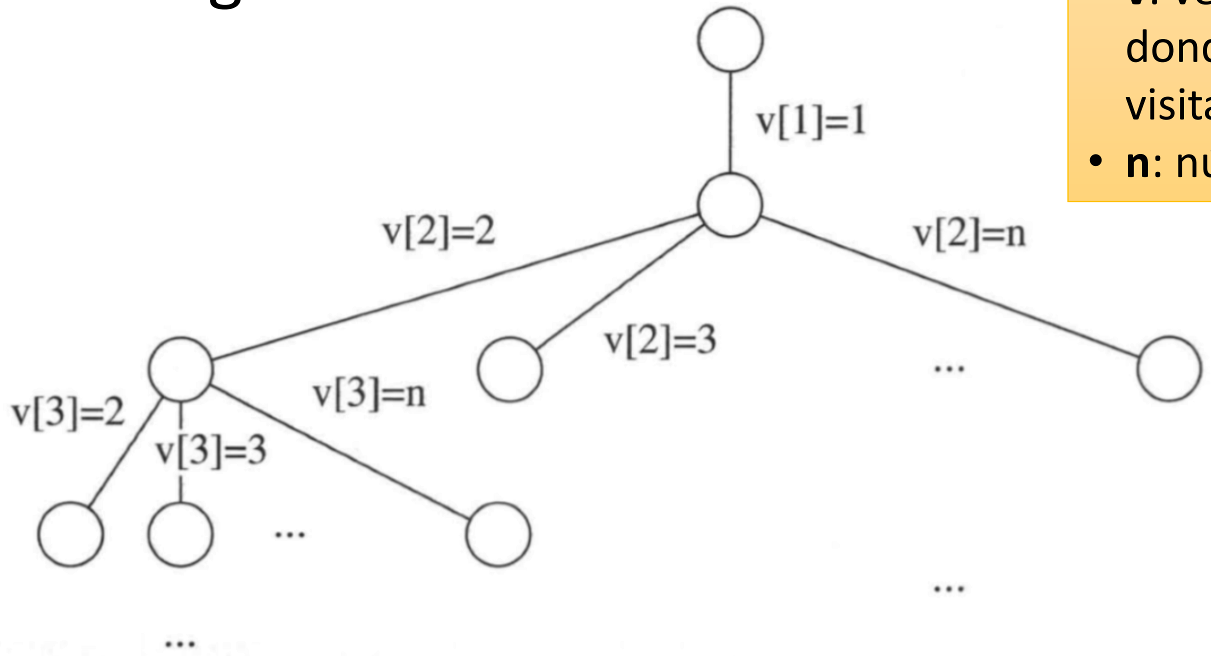
Ejercicio de examen

6. En el problema de colorear un grafo con *n* nodos utilizando *m* colores, de manera que no haya dos vértices adyacentes que tengan el mismo color, una cota superior ajustada del coste de encontrar una solución utilizando un esquema adecuado es del orden de:

   (a) $O(n^{m/2} \log n)$.
   (b) $O(m \log n)$.
   (c) $O(n^m)$.
   (d) $O(nm^n)$.

Ciclos Hamiltonianos

- Definición: caminos que pasan por cada vértice una sola vez y terminan en el vértice inicial
- Objetivo: encontrar todos los ciclos Hamiltonianos de un grafo



- **v**: vector de soluciones donde v[i]=j es el nodo j visitado en el orden i
- **n**: número de nodos

```
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
```

rtice una

onianos

```
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
```

```
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
```

Cota superior de coste:
forma del árbol = $O((n-1)^n)$
o O(n!) para ser más precisos

onianos

```
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
```