

6.- TÈCNIQUES ESPECÍFIQUES DE DISSENY (ESQUEMES).

Introducció a l'ús d'esquemes:

La idea d'esquema no resulta del tot desconeguda doncs ja fou vista a l'assignatura CP (Programació) quan al resoldre el problema; Donada una frase acabada amb un punt comptar quantes vegades conté la lletra A:

```
com:=0
iniciar_lectura
llegir(car)
fer car≠'.'    →   cas car='A'    →   com:=com+1
                    [] car≠'A'    →   res
                   fcas
                   llegir(car)
ffer
escriu(com)
```

S'en treia el següent model per al recorregut de seqüències:

```
iniciar_tractament
obtenir_el_primer
fer ¬final     →   tractar_l'element
                   obtenir_el_següent
ffer
acabar_tractament
```

De la solució del problema; Donada una frase acabada amb un punt cercar-ne la primera vocal, que podríem escriure:

```
iniciar_lectura
llegir(car)
fer (car≠'.') ∧ (¬vocal)   →   llegir(car) ffer
cas vocal      →   escriu(car)
 [] ¬vocal     →   escriu('no hi ha vocals')
fcas
          func vocal(c:car) ret (v:booleà) és
              ret ((c='A') ∨ (c='E') ∨ (c='I') ∨ (c='O') ∨ (c='U'))
          ffunc
```

S'en treia el següent model per a la recerca en seqüències:

```
obtenir_el_primer
fer (¬final) ∧ (¬trobat)   →   obtenir_el_següent ffer
cas trobat     →   tractament_d'èxit
 [] ¬trobat    →   tractament_de_fracàs
fcas
```

Aquests dos esquemes són la base de la major part, per no dir de tots, els exercicis d'aquella assignatura. Per poder-los aplicar el primer que cal és reconèixer el problema com pertanyent al tipus (recorregut o recerca) i en segon lloc fer una aplicació correcta de l'esquema. Per tenir èxit en aquesta tasca caldrà tenir doncs molta cura en aquesta segona fase.

Aquesta part de l'assignatura introdueix nous esquemes aplicables a diversos tipus de problemes. En tots els casos caldrà justificar, en primer lloc, que l'esquema és aplicable al problema. En segon lloc caldrà definir amb precisió l'estructura de dades amb que es representen diferents elements del problema. I per últim caldrà identificar de forma completa els elements de l'esquema amb els del problema.

Així per comptar les *A*s cal començar per raonar que la solució es pot trobar fent un recorregut total de la seqüència de caràcters que formen la frase. En segon lloc detallaríem les dades a usar: Un fitxer de caràcters (que conté un punt al final) i un natural per fer de comptador d'*A*s. Per últim identificarem esquema amb problema:

Iniciar_tractament	com:=0
Obtenir_el_primer	iniciar_lectura; llegir(car)
final	car='.'
tractar_l'element	cas car='A' → com:=com+1 [] car≠'A' → res fcas
Obtenir_el_següent	llegir(car)
Acabar_tractament	escriu(com)

De cara a l'examen és molt important detallar aquestes tres fases, escriure l'algorisme que en resulta i intentar, encara que sigui informalment, justificar-ne la correctesa.

Bibliografia:

- Data Structures and Algorithms/ A.V.Aho, J.E.Hopcroft i J.D.Ullman: Addison-Wesley, Reading (Mass.USA) 1983
- Algoritmos + Estructuras de datos = Programas/ N.Wirth: Castillo, Madrid 1980
- Fundamentals of Computer Algorithms/ Horowith i Sahni: Pitman
- Algorítmica: Concepción y Análisis/ G.Brassard i P.Bratley: Masson, Barcelona 1990

Traducció a Eiffel de les estructures de la notació algorísmica utilitzada: per a l'alternativa:

cas <condició> → <sentència>	**if** <condició> **then** <sentència>
[] ¬<condició> → res	
fcas	**end** --if

cas <condició> → <sentència a>	**if** <condició> **then** <sentència a>
[] ¬<condició> → <sentència b>	**else** <sentència b>
fcas	**end** --if

cas <condició a> → <sentència a>	**if** <condició a> **then** <sentència a>
[] <condició b> → <sentència b>	**elseif** <condició b> **then** <sentència b>
[] <condició c> → <sentència b>	**else** <sentència b>
fcas	**end** --if

En aquest darrer cas s'ha de complir: **precondició Ù ¬<condició a> Ù ¬<condició b> Þ <condició c>** perquè la sentència cas no avorti.

L'estructura iterativa seria:

inicialització	**from** inicialització
fer <condició> → <sentències>	**until** ¬<condició>
	loop <sentències>
ffer	**end** --loop

Tant acció com funció i var hauran de traduir-se **feature** (el cas de var haurà de traduir-se per **local** a les accions i funcions).

6.3.- MÈTODE D'ASSAIG I ERROR (BACKTRACKING).

- Tipus de problemes on és aplicable:
  - Mètode a aplicar a problemes d'heurística.
  - Destinat a obtenir solucions eficients a problemes complexos (complexitat elevada).
  - Quan la solució té forma de tupla de valors ($x_1, \ldots ,x_n$) ($\forall\ i{:}x_i \in S_i$), o es pot expressar en aquesta forma.
  - Quan es disposa d'algun criteri que permet determinar quan una tupla és solució.

- La solució "força bruta" (sense el mètode) $\Rightarrow$ Generar totes les solucions i destriar les que compleixen el criteri en qüestió $\Rightarrow$ Cost exponencial en n ($\Pi\ i{:}1 \leq i \leq n{:}S_i$).

- El mètode consisteix a anar construint la tupla solució a passos per un sistema "d'assaig i error" = Estructurar les possibles tuples en forma d'arbre; Les solucions s'obtenen per recorregut parcial d'aquest arbre.

  Aquest arbre és bàsic tant per determinar la complexitat de l'algorisme (nombre de nusos) com per fer la identificació amb l'esquema. D'aquest arbre s'en diu l'**arbre de recerca**.

- L'eficiència millora en funció del mètode que ens permeti determinar a partir de quin moment una tupla no pot ser solució (quin nus pertany a l'arbre).

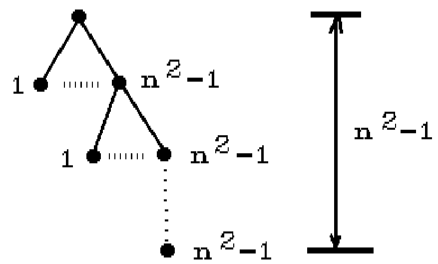
<u>El salt del cavall</u>:

Donat un tauler de $n \times n$ (n^2 quadres, $n \geq 3$) i un cavall, que es mou segons les regles dels escacs, situat al quadre x_0,y_0, ha de recórrer els n^2-1 quadres restants, visitant-los una sola vegada amb n^2-1 moviments.

{tupla=combinacions ordenades de n^2-1 quadres}

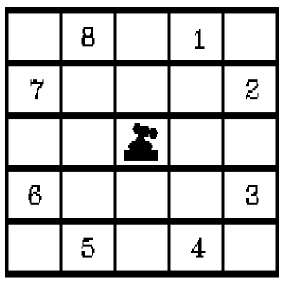
Solució "força bruta":

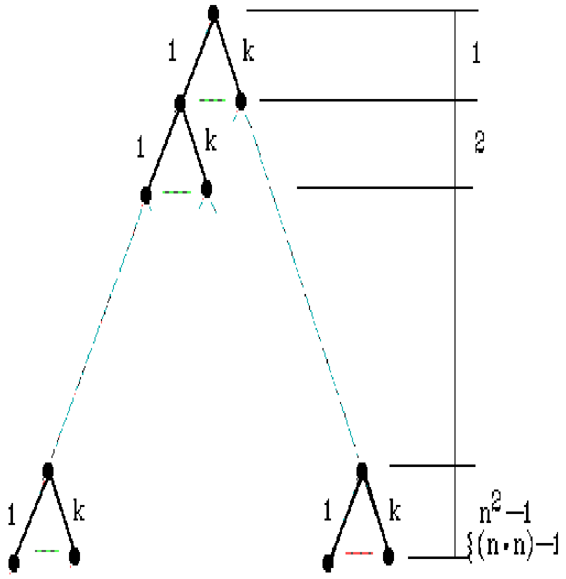
Construir totes les tuples possibles amb els n^2-1 quadres que cal recórrer, esbrinar si es poden recórrer amb els salts del cavall, determinant els que són solució. Complexitat=$(n^2\text{-}1)^{n^2\text{-}1}$



Solució "assaig i error":

Des d'un quadre x_i,y_i pot accedir a un màxim de 8 quadres, depenent dels límits del tauler.





A més d'aquestes possibilitats cal suprimir aquelles en que el quadre ja ha estat visitat, amb el que quedarà l'arbre de recerca:Amb $1 \le k \le 8 \Rightarrow k^{n^2-1} < 8^{n^2-1} = 2^{3 \cdot (n^2-1)}$ nusos possibles

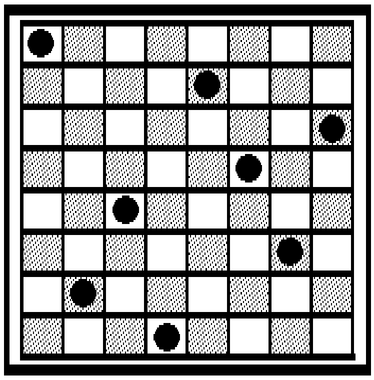
- Requisits:
  - Solució en forma de tupla de dades homogènies.
  - Existència d'un criteri de determinació tuples solució.
  - Existència d'un criteri per a eliminar tuples quan es preveu que no podran ser solució, abans de completar-les (¬*completable*).
- En aquests problemes la definició de l'estructura de les dades ha d'incloure una definició precisa de l'arbre de recerca i de la representació de la tupla solució, així com de les decisions que es prenen.
- Primer esquema: Genera totes les solucions (cas d'haver-n'hi més d'una), suposant que s'arranca en condicions d'obtenir el primer i que l'últim du la informació de que ho és:
  a) Recursiu: És una immersió amb la crida inicial *assaja(1);*

```
acció assaja(k:nat) és                                                      [1]
    prepara_recorregut                       {queda en condicions d'obtenir el primer}
    fer ∃_succ    →   següent_candidat
                      cas solució          →   tracta_solució
                       [] ¬solució         →   res
                      fcas
                      cas completable      →   assaja(k+1)
                       [] ¬completable     →   res
                      fcas
    ffer
facció
```

b) Iteratiu: [2]

```
k:=1
prepara_recorregut
fer k>0  →  cas ∃_succ   →  següent_candidat
                             cas solució        →  tracta_solució
                              [] ¬solució       →  res
                             fcas
                             cas completable    →  k:=k+1
                                                   prepara_recorregut
                              [] ¬completable   →  res
                             fcas
             [] ¬∃_succ  →  k:=k-1
            fcas
ffer
```

L'arbre de recerca és bàsic en dos aspectes ja que d'una banda dóna idea de la màxima complexitat de l'algorime, el cost serà funció del nombre de nusos de l'arbre que es visitin. Per l'altre banda és la base de la identificació, ja que l'algorisme no fa altre cosa que recorrer aquest arbre.



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

El problema de les vuit reines:

Col·locar vuit reines en un tauler d'escacs de manera que no es facin escac entre elles. Solució:

Fases:

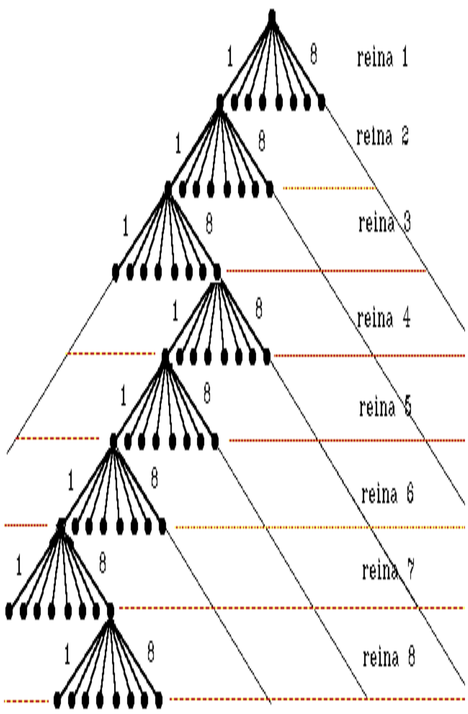
- Justificació, estructura de dades i arbre de recerca
- Identificació i escriptura de l'algorisme
- Raonament de correctesa

Compliment dels requisits:

a) Solució expressable en forma de tupla de dades homogènies, ja que a la solució cada reina estarà en una línia diferent ⇒ poden representar-se amb un vector de vuit elements on el subíndex és el número de la fila que ocupa i el valor la columna.

b) Es pot reconèixer una solució ja que no hi pot haver dues reines a la mateixa columna ni repetir les diagonals (la representació de la solució impedeix que hi hagi dues reines a la mateixa fila).

c) Es poden detectar les tuples no completables (no podem col·locar la reina següent).



Si només es consideren les columnes que no ocupen les reines anteriors llavors a la segona reina li queden 7 llocs, a la tercera 6, a la quarta 5, etc. ... i a la vuitena i darrera només 1.

Si es considera a més que també es poden suprimir les diagonals llavors la 2ª reina té 5 llocs, la 3ª ...

D'aquí surten dues interpretacions de l'esquema:

a) La iteració principal va de posició en posició.

b) La iteració principal va de posició lliure en posició lliure.

Versió a) Identificació:

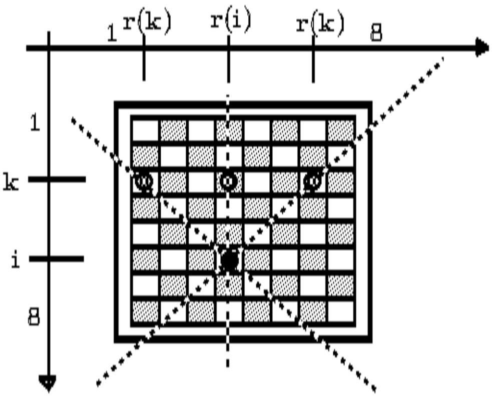
prepara_recorregut	r(k):=0
∃_succ	r(k)<8
següent_candidat	r(k):=r(k)+1
solució	(k=8) ∧ lliure(k,r)
tracta_solució	imprimir_solució
completable	(k<8) ∧ lliure(k,r)

tipus tauler:taula(1÷8) de nat ftipus

```
acció reines(k:nat;var r:tauler) és
    r(k):=0
    fer r(k)<8    →  r(k):=r(k)+1
                     cas (k=8) ∧ lliure(k,r)     →  imprimir_solució
                       [] (k<8) ∨ ¬lliure(k,r)   →  res
                     fcas
                     cas (k<8) ∧ lliure(k,r)     →  reines(k+1,r)
                       [] (k=8) ∨ ¬lliure(k,r)   →  res
                     fcas
    ffer
facció
```

Equival a:

```
acció reines(k:nat;var r:tauler) és
    r(k):=0
    fer r(k)<8    →  r(k):=r(k)+1
                     cas lliure(k,r)      →  cas k=8  →  imprimir_solució
                                               [] k<8  →  reines(k+1,r)
                                             fcas
                       [] ¬lliure(k,r) →
res
                     fcas
    ffer
facció
```



Quan volen posar una nova reina (*i*) només hi ha reines posades (*k*) més amunt (*i*>*k*). Dues reines estan a la mateixa columna si:

$$r(i)=r(k)$$

Dues reines estan a la mateixa diagonal descendent si:

$$r(i)-r(k) = i-k$$

Dues reines estan a la mateixa diagonal ascendent si:

$$r(k)-r(i) = i-k$$

Poden doncs resumir que dues reines es maten en diagonal si:

$$(i-k) = abs(r(i)-r(k))$$

```
{(1≤i≤8) ∧ (∀j: 1≤j≤i: 1≤r(j)≤8)}
func lliure(i:nat;r:tauler) ret (ll:booleà) és
    var k:nat fvar
    k:=1
    fer (r(k)≠r(i)) ∧ ((i-k)≠abs(r(i)-r(k)))   →  k:=k+1 ffer
    ret i=k
ffunc
```

Raonament de correctesa: Només s'ha posat una reina si la posició no està amenaçada ⇒ si es posen vuit reines s'ha assolit l'objectiu.

Iterativa:

```
k:=1
r(k):=0
fer k>0  →  cas r(k)<8   →  r(k):=r(k)+1
                             cas lliure(k,r)     →  cas k=8   →  imprimir_solució
                                                      [] k<8   →  k:=k+1
                                                                  r(k):=0
                                                    fcas
                               [] ¬lliure(k,r)   →  res
                             fcas
              [] r(k)=8   →  k:=k-1
            fcas
ffer
```

Anàlisi quantitativa: 1625 nusos visitats d'un arbre de 69.281, que és el 2,34% de la solució de les permutacions 8!=40.320 i el 0,0096853 de la solució bruta 8^8=16.777.216

Segon esquema: (màquina del tipus I de Scholl) arranquem amb el primer element obtingut i al demanar el següent de l'últim es detecta que ja no en queden més.

```
acció assaja_2a(k:nat) és                                                       [3]
    prepara_rec_2a                                             {torna amb el primer}
    fer ∃_cand     →  cas solució        →  tracta_solució
                        [] ¬solució      →  res
                      fcas
                      cas completable    →  assaja_2a(k+1)
                        [] ¬completable  →  res
                      fcas
                      següent_candidat
    ffer
facció
```

Iteratiu:

```
k:=1                                                                            [4]
prepara_rec_2a
fer k>0  →  cas ∃_cand       →  cas  solució        →  tracta_solució
                                  [] ¬solució       →  res
                                fcas
                                cas  completable    →  k:=k+1
                                                       prepara_rec_2a
                                  [] ¬completable   →  següent_candidat
                                fcas
              [] ¬∃_cand     →  k:=k-1
                                cas k>0   →  següent_candidat
                                  [] k=0  →  res
                                fcas
            fcas
ffer
```

Aplicació a les 8 reines:

```
acció reines_2a(k:nat;var r:tauler) és                                          [3]
    r(k)=1
    fer r(k)≤8  →  cas lliure(k,r)     →  cas k=8    →  impr_solució
                                            [] k<8   →  reines_2a(k+1)
                                          fcas
                     [] ¬lliure(k,r)   →  res
                   fcas
                   r(k):=r(k)+1
    ffer
facció
```

Iterativa:

```
k:=1                                                                            [4]
r(k)=1
fer k>0  →  cas r(k)≤8  →  cas lliure(k,r) ∧ (k=8)       →  impr_solució
                             [] ¬lliure(k,r) ∨ (k<8)     →  res
                           fcas
                           cas lliure(k,r) ∧ (k<8)       →  k:=k+1; r(k)=1
                             [] ¬lliure(k,r) ∨ (k=8)     →  r(k):=r(k)+1
                           fcas
              [] r(k)>8 →  k:=k-1
                           cas k>0   →  r(k):=r(k)+1
                             [] k=0  →  res
                           fcas
            fcas
ffer
```

Però aquesta solució és més complexa que l'anterior sense que tingui cap avantatge.

Versió b: La iteració principal va de posició lliure en posició lliure. Recordeu la precondició de la funció lliure {*(1£i£8) Ù ("j: 1£j£i: 1£r(j)£8)*}

```
    acció reines_3a(k:nat;var r:tauler) és                                              [3]
        r(k):=1
        fer (r(k)≤8) ∧c ¬lliure(k,r)      →  r(k):=r(k)+1 ffer
        fer r(k)≤8      →  cas k=8      →  impr_solució            |
                              [] k<8    →  res                     | cas k=8  →  impr_solució
                           fcas                                    |  []  k<8 →  reines_3a(k+1,r)
                           cas k<8      →  reines_3a(k+1,r)        | fcas
                              [] k=8    →  res                     |
                           fcas                                    |
                           r(k):=r(k)+1
                           fer (r(k)≤8) ∧c ¬lliure(k,r)      →  r(k):=r(k)+1 ffer
        ffer
    facció
```

Nota: Es podria suprimir els ∧c definint una nova funció *lliure* de forma que per *r(k)=9* retornés sempre *fals* sense avortar. Ara, tal com l'hem definida, podria retornar *cert*. De fet, aquest ∧c no deixa de ser un purisme.

- Cal reduir les comparances que són redundants.
- Cal transformar-lo a iteratiu.
- Provar els models [1] i [2].

Marcatges:

En ocasions pot millorar-se la determinació d'una solució si es procedeix a un marcatge (i desmarcatge) de la tupla en formació. Com solen ser costosos l'ús de marcatges es limitarà als casos en que es produeixi una millora real o en què siguin imprescindibles. Apareix una nova funció **acceptable**, fruit del marcatge, que permet desestimar un candidat *a priori*, equival a (*solució Ú completable*). Vol dir que:

acceptable Ù ¬solució Þ completable
acceptable Ù ¬completable Þ solució

Esquemes per solucions amb marcatges: al ser una immersió, cal la primera crida; *init_marcatge; assaja(1)*

```
    acció assaja(k:nat) és                                                              [5]
        prepara_recorregut
        fer ∃_succ      →  següent_candidat
                           cas acceptable      →  enregistra_candidat
                                                  cas solució        →  tracta_solució
                                                     [] ¬solució     →  res
                                                  fcas
                                                  cas completable        →  assaja(k+1)
                                                     [] ¬completable     →  res
                                                  fcas
                                                  desenregistra_candidat
                              [] ¬acceptable   →  res
                           fcas
        ffer
    facció
```

```
k:=1                                                                                    [6]
init_marcatge
prepara_recorregut
fer k>0  →  cas ∃_succ   →  següent_candidat
                            cas acceptable →  enregistra_candidat
                                                cas solució    →  tracta_solució
                                                 [] ¬solució   →  res
                                                fcas
                                                cas completable   →  k:=k+1;prepara_recorregut
                                                 [] ¬completable  →  desenregistra_candidat
                                                fcas
                             [] ¬acceptable   →  res
                            fcas
             [] ¬∃_succ  →  k:=k-1
                            cas k>0      →  desenregistra_candidat
                             [] k=0      →  res
                            fcas
            fcas
ffer
```

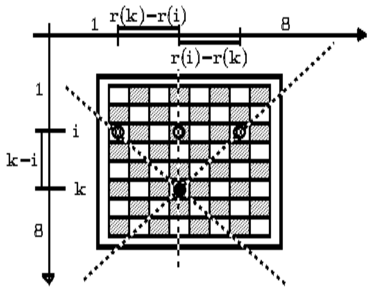
```
        acció assaja(k:nat) és                                                          [7]
            prepara_recorregut_2a
            fer hi_ha_candidats →
                cas acceptable      →  enregistra_candidat
                                        cas solució    →  tracta_solució
                                         [] ¬solució   →  res
                                        fcas
                                        cas completable    →  assaja(k+1)
                                         [] ¬completable   →  res
                                        fcas
                                        desenregistra_candidat
                 [] ¬acceptable     →  res
                fcas
                següent_candidat
            ffer
        facció
```

```
init_marcatge ; k:=1 ;  prepara_recorregut_2a                                           [8]
fer k>0  →  cas hi_ha_candidats    →  cas acceptable →  enregistra_candidat
                                                         cas solució    →  tracta_solució
                                                          [] ¬solució   →  res
                                                         fcas
                                                         cas completable    →  k:=k+1
                                                                               prepara_recorregut_2a
                                                          [] ¬completable   →  desenregistra_candidat
                                                                               següent_candidat
                                                         fcas
                                       [] ¬acceptable    →  següent_candidat
                                      fcas
             [] ¬hi_ha_candidats   →  k:=k-1
                                      cas k>0      →  desenregistra_candidat
                                                      següent_candidat
                                       [] k=0      →  res
                                      fcas
            fcas
ffer
```

Aplicació al problema de les 8 reines:

Una alternativa a la funció lliure és marcar tots els quadres que mata una reina, vol dir columnes i diagonals, en tal cas amb l'esquema [1]:

```
acció reines(k:nat) és                                                                        [5]
    r(k):=0
    fer r(k)<8→   r(k):=r(k)+1
                  cas lliure(k,r)     →   posa_reina(k)
                                          cas k=8      →   impr_tauler
                                           [] k<8      →   reines(k+1)
                                          fcas
                                          treu_reina(k)
                   [] ¬lliure(k,r)→   res
                  fcas
    ffer
facció
```



- La mateixa columna | (col): $r(i)=r(k)$
- La mateixa diagonal ascendent / (da): $(k-i=r(i)-r(k)) \Rightarrow r(k)+k=r(i)+i$
- La mateixa diagonal descendent \ (dd): $(k-i=r(k)-r(i)) \Rightarrow r(k)-k=r(i)-i$

Tres taules de booleans: col:taula(1÷8) de booleà
da:taula(2÷16) de booleà
dd:taula(-7÷7) de booleà

```
posar_reina:     (col(r(k)),da(r(k)+k),dd(r(k)-k)):=(fals,fals,fals)
treu_reina :     (col(r(k)),da(r(k)+k),dd(r(k)-k)):=(cert,cert,cert)
lliure:          col(r(k)) ∧ da(r(k)+k) ∧ dd(r(k)-k)
acció init_marcatge és
    var i:enter fvar
    i:=1;  fer i≤8   →   col(i):=cert;i:=i+1   ffer
    i:=2;  fer i≤16  →   da(i):=cert;i:=i+1    ffer
    i:=-7; fer i≤7   →   dd(i):=cert;i:=i+1    ffer
facció
```

- Transformar a iteratiu.
- Provar les altres alternatives.
- La solució de marcar una taula de *(1 ¸8,1 ¸8)* és molt més complicada al moment de marcar i desmarcar, ja que cal saber quantes reines maten un quadre determinat, vol dir usar una taula de naturals i comptar-les.

Esquemes per a generar una única solució:

La primera crida: èxit:=fals;assaja(1)

```
acció assaja(k:nat) és                                                                        [9]
    prepara_recorregut
    fer ¬èxit ∧ ∃_succ     →   següent_candidat
                               cas  solució    →   èxit:=cert
                                                   tracta_solució
                                [] ¬solució    →   cas  completable     →   assaja(k+1)
                                                    [] ¬completable     →   res
                                                   fcas
                               fcas
    ffer
facció
```

Iteratiu:

```
    k:=1                                                                                     [10]
    èxit:=fals
    prepara_recorregut
    fer ¬èxit ∧ k>0→   cas ∃_succ        →   següent_candidat
                                             cas solució     →   èxit:=cert
                                                                 tracta_solució
                                              [] ¬solució    →   cas completable     →   k:=k+1
                                                                                         prepara_recorregut
                                                                  [] ¬completable    →   res
                                                                 fcas
                                             fcas
                        [] ¬∃_succ    →   k:=k-1
                       fcas
    ffer
```

Per l'altre tipus de màquina:

```
    acció assaja(k:nat;var èxit:booleà) és                                                   [11]
        prepara_recorregut_2a
        fer ¬èxit ∧ hi_ha_candidat   →   cas solució     →   èxit:=cert
                                                             tracta_solució
                                          [] ¬solució    →   cas completable     →   assaja(k+1,èxit)
                                                              [] ¬completable    →   res
                                                             fcas
                                                             cas ¬èxit  →   següent_candidat
                                                              [] èxit   →   res
                                                             fcas
                                         fcas
        ffer
    facció
```

Iteratiu:

```
    k:=1                                                                                     [12]
    èxit:=fals
    prepara_recorregut_2a
    fer ¬èxit ∧ k>0→   cas hi_ha_candidat         →   cas solució     →   èxit:=cert
                                                                          tracta_solució
                                                       [] ¬solució    →   cas completable     →
                                                                                  k:=k+1
                                                                                  prepara_recorregut_2a
                                                                           [] ¬completable    →
                                                                                  següent_candidat
                                                                          fcas
                                                      fcas
                        [] ¬hi_ha_candidat     →   k:=k-1
                                                      cas k>0   →   següent_candidat
                                                       [] k≤0   →   res
                                                      fcas
                       fcas
    ffer
```

Amb marcatges: {recordant *acceptable Ù ¬solució Þ completable*}

```
    acció assaja(k:nat;var èxit:booleà) és                                                  [13]
        prepara_recorregut
        fer ¬èxit ∧ ∃_succ  →  següent_candidat
                               cas acceptable   →  enregistra_candidat
                                                   cas solució   →  èxit:=cert
                                                                    tracta_solució
                                                    [] ¬solució  →  assaja(k+1,èxit)
                                                                    desenregistra_candidat
                                                   fcas
                                [] ¬acceptable  →  res
                               fcas
        ffer
    facció
Iteratiu:
    init_marcatge                                                                           [14]
    k:=1
    èxit:=fals
    prepara_recorregut
    fer ¬èxit ∧ k>0→  cas ∃_succ    →  següent_candidat
                                       cas acceptable   →  enregistra_candidat
                                                           cas solució   →  èxit:=cert
                                                                            tracta_solució
                                                            [] ¬solució  →  k:=k+1
                                                                            prepara_recorregut
                                                           fcas
                                        [] ¬acceptable  →  res
                                       fcas
                       [] ¬∃_succ   →  k:=k-1
                                       cas k>0  →  desenregistra_candidat
                                        [] k≤0  →  res
                                       fcas
                      fcas
    ffer
```

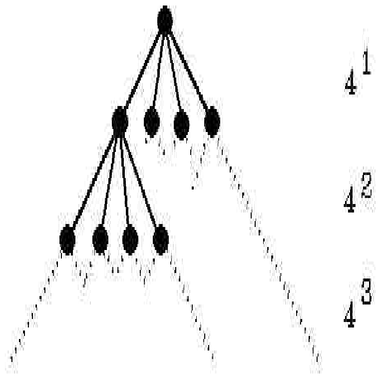
Exercici: Escriure els casos per l'altre tipus de màquina amb marcatges (corresponents als esquemes d'aquest capítol [7] i [8]).

Optimització:

- En alguns casos no es cerquen totes les solucions ni solament la primera, sinó aquella que és òptima en cert sentit (no totes les solucions tenen el mateix valor).
- Es generen totes les solucions, i es reté la millor fins el moment (Si és una funció f qui ens dóna el sentit d'optimitat):

```
    cas f(solució_actual)>f(òptim)     →  òptim:=solució_actual
     [] f(solució_actual)≤f(òptim)     →  res
    fcas
```

- Millorant l'algorisme si algun mecanisme ens permet saber si la solució actual pot ser o no millor que la seleccionada com a valor òptim provisional. ⇒ Aquesta és la idea base del mètode de brancament i poda.



Exemple 1; **Acolorit d'un mapa**: Donat un mapa, acolorir les regions de manera que dues regions adjacents siguin de color diferent, usant exclusivament quatre colors.

- Generarem totes les maneres possibles d'acolorir-lo.
- La solució és expressable en forma de tupla de dades homogènies amb tants components com regions. Cada element té el valor del color assignat.
- El mapa ve donat en forma d'una matriu de contigüitat.
- La solució bruta són variacions amb repetició de *4* elements presos de *n* en *n* vol dir que són 4^n, el que vol dir que són 4^n tuples de *n* elements, és a dir la complexitat és $n\cdot 4^n$.

  L'arbre té $(\Sigma i: 1\le i\le n: 4^i)=(4/3)\cdot(4^n-1)$ (ja que $a+a\cdot c+a\cdot c^2+\cdots+a\cdot c^{n-1}=a\cdot(c^n-1)/(c-1)$) però no es recorren tots els nusos.

```
{k≤n}
acció colora (k:nat) és
    c(k):=0
    fer c(k)<4   →  c(k):=c(k)+1
                    cas (k=n) ∧ compatible(k)      →  imprimir_solució  ⎤
                     [] (k<n) ∨ ¬compatible(k)     →  res               ⎥
                    fcas                                                ⎬  (1) ↓
                    cas (k<n) ∧ compatible(k)      →  colora(k+1)       ⎥
                     [] (k=n) ∨ ¬compatible(k)     →  res               ⎥
                    fcas                                                ⎦
    ffer
facció
```

```
(1) cas compatible(k)    →  cas k=n   →  imprimir_solució
                             [] k<n   →  colora(k+1)
                            fcas
     [] ¬compatible(k)   →  res
    fcas
```

```
func compatible(k:nat) ret (c:booleà) és
    var h:nat fvar
    h:=1
    fer ¬contigu(h,k) ∨ (c(h)≠c(k))   →  h:=h+1 ffer
    ret h=k
ffunc
```

⇒ Per simplificar s'ha considerat que contigu (la matriu de contigüitats) és una variable global de definició: *taula(1..n,1..n) de booleà*, que tindrà la diagonal principal de valor *cert* i *contigu(i,j)=cert* si *país(i)* és veí del *país(j)*.

- Millor eficiència amb una bona ordenació de les regions. Millor quan més prop de l'arrel podem descartar les solucions ⇒ Països en ordre descendent del nombre connexions amb veïns.

Exemple 2; **La motxilla**:

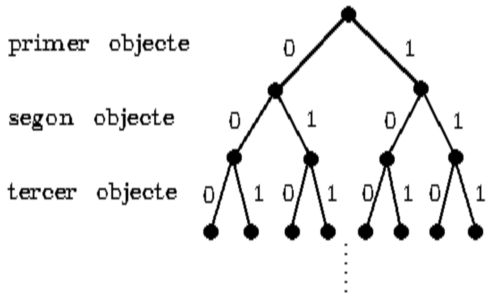
Donats *n* objectes de volums (o de pes) *v(i)* i de pes (o de valor) *p(i)* i una motxilla de capacitat *M* (aprofitable al 100%) (o que suporta un pes màxim *M*) elegir un subconjunt d'aquests objectes tal que es carregui el màxim pes (valor) possible. Equival a que si *S* és el subconjunt d'objectes seleccionat *i*:

$$M\ge(\Sigma i{:}i\in S{:}v(i)) \qquad (2)$$
$$P=(\Sigma i{:}i\in S{:}p(i))$$

No existeix un altre *S* que complint (2) faci més gran el valor *P*, s'escriurà:

$$(\forall S'{:}\ M\ge(\Sigma i{:}\ i\in S'{:}\ v(i))\wedge(S'\ne S){:}(\Sigma i{:}\ i\in S'{:}\ p(i))\le P)$$

- Es pot representar la solució com a tupla amb dues possibilitats:



a) Tupla de *n* elements de valors que indiquen si l'element pertany o no a la solució amb els valors 0 o 1 (que podrien ser *cert* o *fals*) (0 vol dir que l'element *i* no pertany a la solució *S*, 1 vol dir que sí hi pertany).

b) Tupla amb els *k* valors dels objectes que formen la solució (els *k* subíndex de *v* i *p*).

Versió I (representació a):

Variables globals:

```
var  t, t_òptim: taula(1÷n) de -1÷1
     v, p: taula(1÷n) de real
     M:nat
     profit_òptim:real
fvar
```

Usarem les funcions:

```
volum(k) =(Σi:1≤i≤k:v(i)·t(i))
profit(k)=(Σi:1≤i≤k:p(i)·t(i))
```

{Primera crida: *profit_òptim:=0; motxilla(1)*}

```
acció motxilla1(k:nat) és                                            [1]
    t(k):=-1                                                         {prepara_recorregut}
    fer t(k)<1 →                                                     {∃_succ}
        t(k):=t(k)+1                                                 {següent_candidat}
        cas (k=n)∧(volum(k)≤M)     →                                 {solució}
                cas profit(k)>profit_òptim   →  t_òptim:=t           {tracta_solució}
                                                profit_òptim:=profit(k)
                 [] profit(k)≤profit_òptim   →  res
                fcas                                                 {fi de tracta_solució}
         [] (k<n)∨(volum(k)>M)     →                                 {¬solució}
                res
        fcas
        cas (k<n)∧(volum(k)≤M)     →   motxilla1(k+1)                {completable}
         [] (k≥n)∨(volum(k)>M)     →   res                           {¬completable}
        fcas
    ffer
facció
```

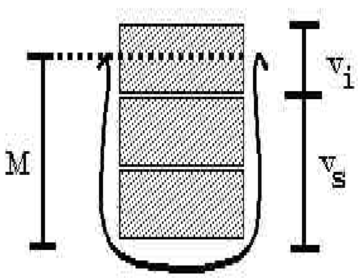
Simplificat:

```
acció motxilla1(k:nat) és
    t(k):=-1
    fer t(k)<1 →   t(k):=t(k)+1
|                  cas (k=n)∧(volum(k)≤M)∧(profit(k)>profit_òptim)  →
|                                                  t_òptim:=t
|                                                  profit_òptim:=profit(k)
|                   [] (k<n)∨(volum(k)>M)∨(profit(k)≤profit_òptim)  →
|                                                  res
|                  fcas
                   cas (k<n)∧(volum(k)≤M)   →  motxilla1(k+1)
                    [] (k≥n)∨(volum(k)>M)   →  res
                   fcas
    ffer
facció
```

Millores:

- Usar una immersió o marcatge que eviti el càlcul redundant de *volum(k)* i *profit(k)* (ús d'acumuladors).
- Evitar comparacions redundants.

```
Variables globals:                     var profit, volum: real fvar
    acció motxilla2(k:nat) és
        t(k):=-1
        fer t(k)<1 →
            t(k):=t(k)+1
            cas t(k)=1      →  profit:=profit+p(k)     ⎫
                               volum:=volum+v(k)       ⎬  profit:=profit+p(k)*t(k)
              [] t(k)≠1     →  res                     ⎪  volum:=volum+v(k)*t(k)
            fcas                                       ⎭
            cas volum≤M  →  cas k=n   →  cas profit(k)>profit_òptim   →  t_òptim:=t
                                                                         profit_òptim:=profit(k)
                                           [] profit(k)≤profit_òptim  →  res
                                         fcas
                              [] k<n    →  motxilla2(k+1)
                            fcas
              [] volum>M →  res
            fcas
            cas t(k)=1          →  profit:=profit-p(k)     ⎫
                                   volum:=volum-v(k)       ⎬  profit:=profit-p(k)*t(k)
              [] t(k)≠1         →  res                     ⎪  volum:=volum-v(k)*t(k)
            fcas                                           ⎭
        ffer
    facció
```



És possible obtenir una funció que permeti retallar més l'arbre que simplement amb *volum£M.*

Introduint el concepte de densitat {*p(i)/v(i)*} i ordenant els objectes en ordre decreixent d'aquest valor (primer els més densos) podem obtenir fàcilment la cota superior del màxim profit que es pot obtenir expandint un nus amb qualsevol dels seus fills o descendent, que és el profit que es pot obtenir amb el dret a fragmentar el darrer objecte, vol dir el profit ideal en el cas que poguéssim aprofitar els més densos a partir de la situació actual:

```
    func profit_límit(i:nat;vs,ps:real) ret (l:real) és
        fer (vs<M)∧(i<n)  →  i:=i+1
                              cas vs+v(i)≤M     →  ps:=ps+p(i)
                                                   vs:=vs+v(i)
                                [] vs+v(i)>M    →  ps:=ps+(M-vs)*p(i)/v(i)
                                                   vs:=M
                              fcas
        ffer
        ret ps
    ffunc
```

Sembla raonable esperar que siguin els més densos els més probables de pertànyer a la solució, pel que sembla millor seleccionar en ordre invers (els primers es més fàcil que hi siguin), ens cal modificar les definicions de *t* i *t_òptim*:

```
    var     t:taula(1÷n) de 0÷2
            t_òptim:taula(1÷n) de 0÷2
    fvar
```

```
acció motxilla3(k:nat) és
    t(k):=2
    fer t(k)>0 →
            t(k):=t(k)-1
            profit:=profit+p(k)*t(k)                    {el marcatge serveix per determinar si és acceptable}
            volum:=volum+v(k)*t(k)
            cas (volum≤M) ∧ profit_límit(k,volum,profit)>profit_òptim      →
                        cas k=n     →   cas profit>profit_òptim  →  t_òptim:=t
                                                                     profit_òptim:=profit(k)
                                         [] profit≤profit_òptim  →  res
                                        fcas
                         [] k<n     →   motxilla3(k+1)
                        fcas
             [] (volum>M) ∨ profit_límit(k,volum,profit)≤profit_òptim      →  res
            fcas
            profit:=profit-p(k)*t(k)
            volum:=volum-v(k)*t(k)
    ffer
facció
```

- Transformar-ho a iteratiu.

Fins aquí l'esquema. Pot semblar complex encara que no ho és, ja que es tracta simplement de recórrer l'arbre de recerca.