### Tema 6. Cerca exhaustiva

Estructures de Dades i Algorismes

FIB

Antoni Lozano

Q1 2019–2020 Versió de 27 de novembre de 2019

### Tema 6. Cerca exhaustiva

- 1 Força bruta i cerca amb retrocés
  - Algorismes de força bruta
  - Cerca amb retrocés
  - Algorisme genèric
- 2 Problemes
  - La reconstrucció Turnpike
  - Les n reines
  - El quadrat llatí
  - Els salts de cavall
  - Graf hamiltonià
  - Problema del viatjant
  - La motxilla
- 3 El principi minimax

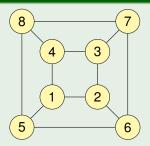
### Tema 6. Cerca exhaustiva

- 1 Força bruta i cerca amb retrocés
  - Algorismes de força bruta
  - Cerca amb retrocés
  - Algorisme genèric
- 2 Problemes
  - La reconstrucció Turnpike
  - Les *n* reines
  - El quadrat llatí
  - Els salts de caval
  - Graf hamiltonià
  - Problema del viatjant
  - La motxilla
- 3 El principi minimax

Hem tractat problemes combinatoris (per exemple, **cicle hamiltonià**) definits per una sèrie de regles o restriccions que defineixen un **espai de solucions**.

Tot sovint, l'única manera de resoldre un d'aquests problemes és provar totes les possibilitats per trobar una solució que satisfà les restriccions.

### Exemple: Cicle hamiltonià

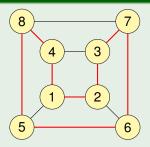


Les possibilitats són totes les permutacions de vèrtexs:
(12345678) (12345687) (12376584

Hem tractat problemes combinatoris (per exemple, **cicle hamiltonià**) definits per una sèrie de regles o restriccions que defineixen un **espai de solucions**.

Tot sovint, l'única manera de resoldre un d'aquests problemes és provar totes les possibilitats per trobar una solució que satisfà les restriccions.

### Exemple: Cicle hamiltonià



Les *possibilitats* són totes les permutacions de vèrtexs:

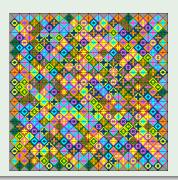
### Exemple: Puzzle Eternity II

- Dissenyat per Lord Monckton, que va oferir \$2M del 2007 al 2010 per una solució completa
- El puzzle era del tipus d'aparellament d'arestes
- El premi va quedar desert, però es van concedir \$10.000 a una solució parcial de 467 arestes aparellades del total de 480



### Exemple: Puzzle Eternity II

- Dissenyat per Lord Monckton, que va oferir \$2M del 2007 al 2010 per una solució completa
- El puzzle era del tipus d'aparellament d'arestes
- El premi va quedar desert, però es van concedir \$10.000 a una solució parcial de 467 arestes aparellades del total de 480



### Exemple: Puzzle Eternity II

- Dissenyat per Lord Monckton, que va oferir \$2M del 2007 al 2010 per una solució completa
- El puzzle era del tipus d'aparellament d'arestes
- El premi va quedar desert, però es van concedir \$10.000 a una solució parcial de 467 arestes aparellades del total de 480

Our calculations are that if you used the world's most powerful computer and let it run from now until the projected end of the universe, it might not stumble across one of the solutions.

Lord Monckton, 2005, The Times

De l'estratègia d'explorar sistemàticament un espai de solucions en diem força bruta o cerca exhaustiva:

- Acostuma a ser exponencial
- Pot ser lenta, però és millor que no tenir solució
- Pot arribar a ser pràctica amb entrades petites
- Es pot ajudar d'altres tècniques (com dividir i vèncer o algorismes voraços)
- Pot tenir com a objectiu trobar totes les solucions, trobat la solució òptima, determinar l'existència d'alguna solució, etc.

Suposem que volem processar totes les cadenes de zeros i uns de mida *n*.

Disposem d'un procediment PROCESSAR( $\mathcal{C}$ ) que fa el tractament desitjat del vector  $\mathcal{C}$ .

```
BINARI(n) (processar totes les cadenes binàries de mida n)

si n=0 llavors

PROCESSAR(C)

si no

C[n] \leftarrow 0; BINARI(n-1)

C[n] \leftarrow 1; BINARI(n-1)
```

El cost ve donat per  $T(n) = 2T(n-1) + \Theta(1)$ . Pel teorema mestre I:  $T(n) \in \Theta(2^n).$ 

Suposem que volem processar totes les cadenes de zeros i uns de mida *n*.

Disposem d'un procediment PROCESSAR( $\mathcal{C}$ ) que fa el tractament desitjat del vector  $\mathcal{C}$ .

```
BINARI(n) (processar totes les cadenes binàries de mida n)

si n=0 llavors

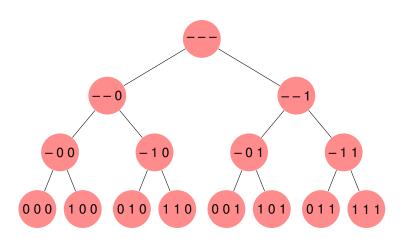
PROCESSAR(C)

si no

C[n] \leftarrow 0; BINARI(n-1)
C[n] \leftarrow 1; BINARI(n-1)
```

El cost ve donat per  $T(n)=2T(n-1)+\Theta(1)$ . Pel teorema mestre I:  $T(n)\in\Theta(2^n).$ 

Per a n = 3, s'obté l'arbre de recursió:



Podem considerar un algorisme genèric iteratiu de cerca exhaustiva que fa servir els procediments:

- PRIMER(x): genera un primer candidat
- SEGÜENT(x, c): genera el candidat següent a c
- VÀLID(x, c): comprova si el candidat c és solució
- NUL(c): diu si c és un candidat "nul"

### Algorisme genèric de força bruta

```
FORÇA BRUTA(x)
c \leftarrow \mathsf{PRIMER}(x)
mentre no (\mathsf{NUL}(c))
si \mathsf{VALID}(x,c) llavors
\mathsf{PROCESSAR}(c)
c \leftarrow \mathsf{SEGÜENT}(x,c)
```

#### Quin cost té la cerca exhaustiva?

Per exemple, les cerques en profunditat (DFS) i amplada (BFS) en grafs també són cerques exhaustives:

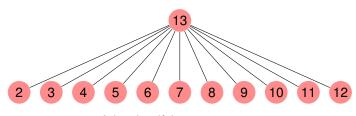
- si el graf ve donat a l'entrada, són cerques polinòmiques
- si hi ha un arbre o graf implícit, normalment són exponencials

### Exemple: nombres primers

```
bool es_primer (int x) {
  if (x <= 1) return false;
  for (int i = 2; i < x; ++i)
      if (x % i == 0) return false;
  return true; }</pre>
```

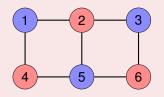
Nombre màxim d'iteracions: (x-1)-2+1=x-2. Cost en funció de x:  $\Theta(x)$ .

Cost en funció de n = |x|:  $\Theta(2^n)$ .



#### Exercici: Recobriment

Un *recobriment de vèrtexs* d'un graf G = (V, E) és un subconjunt  $U \subseteq V$  tal que per a cada aresta  $\{u, v\} \in E$ , almenys un dels extrems u o v pertany a U. La mida d'un recobriment U és |U|. Per exemple, el graf següent té dos recobriments de mida 3:  $\{1,3,5\}$  i  $\{2,4,6\}$ :



Dissenyeu i analitzeu un algorisme que, donat un graf G i un natural k, determini si G té un recobriment de vèrtexs de mida k.

#### Definició

Es defineix el nombre de Ramsey R(k, l) com el natural n més petit tal que tot graf de n vèrtexs conté un subgraf complet de k vèrtexs o un conjunt independent de l vèrtexs.

#### Definició

Es defineix el nombre de Ramsey R(k, l) com el natural n més petit tal que tot graf de n vèrtexs conté un subgraf complet de k vèrtexs o un conjunt independent de l vèrtexs.

### **Propietats**

Observeu que:

- $R(k, l) \ge \max\{k, l\}$
- $R(3,3) \ge 6$

#### Definició

Es defineix el nombre de Ramsey R(k, l) com el natural n més petit tal que tot graf de n vèrtexs conté un subgraf complet de k vèrtexs o un conjunt independent de l vèrtexs.

### Alguns nombres de Ramsey

- P(3,3) = 6
- P(4,4) = 18
- $43 \le R(5,5) \le 49$
- $102 \le R(6,6) \le 165$

#### Definició

Es defineix el nombre de Ramsey R(k, l) com el natural n més petit tal que tot graf de n vèrtexs conté un subgraf complet de k vèrtexs o un conjunt independent de l vèrtexs.

### Alguns nombres de Ramsey

- P(3,3) = 6
- P(4,4) = 18
- $43 \le R(5,5) \le 49$
- $102 \le R(6,6) \le 165$

#### Exercici: Cost del càlcul dels nombres de Ramsey

Dissenyeu a grans trets un algorisme per calcular R(k, l) donats k i l i estimeu una fita superior del seu cost.

Un algorisme de cerca amb retrocés (o tornada enrere) funciona com una cerca exhaustiva, però s'atura quan troba una solució parcial que no es pot estendre a una solució.

L'esquema de cerca amb retrocés:

- es pot veure com una implementació intel·ligent de la cerca exhaustiva amb un cost millorat, però sovint encara exponencial
- en anglès, s'anomena backtracking

Un algorisme de cerca amb retrocés (o tornada enrere) funciona com una cerca exhaustiva, però s'atura quan troba una solució parcial que no es pot estendre a una solució.

#### L'esquema de cerca amb retrocés:

- es pot veure com una implementació intel·ligent de la cerca exhaustiva amb un cost millorat, però sovint encara exponencial
- en anglès, s'anomena backtracking

### Exemple: moblar un pis

- Estratègia de força bruta: provar totes les configuracions dels mobles en tots els espais
- L'estratègia de cerca amb retrocés aprofita que:
  - cada moble acostuma a anar a un espai concret (no posarem el sofà a la cuina)
  - hi ha mobles que van junts (cadires i taula, llit i tauletes)
  - si una subdistribució no és satisfactòria, no considerarem la distribució que la conté (si no ens agrada posar un moble davant d'una finestra, ja no explorarem a partir d'aquí)

Eliminar un gran grup de possibilitats en un pas es coneix com a poda.

### Exemple: moblar un pis

- Estratègia de força bruta: provar totes les configuracions dels mobles en tots els espais
- L'estratègia de cerca amb retrocés aprofita que:
  - cada moble acostuma a anar a un espai concret (no posarem el sofà a la cuina)
  - hi ha mobles que van junts (cadires i taula, llit i tauletes)
  - si una subdistribució no és satisfactòria, no considerarem la distribució que la conté (si no ens agrada posar un moble davant d'una finestra, ja no explorarem a partir d'aquí)

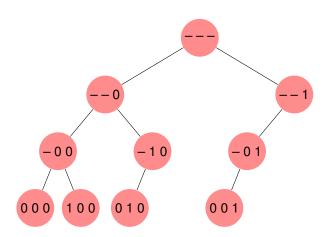
Eliminar un gran grup de possibilitats en un pas es coneix com a poda.

Suposem que volem totes les cadenes de zeros i uns de mida n que contenen un màxim de k uns. Podem modificar l'algorisme vist abans de manera que eviti la recursió dels subarbres que contenen més de k uns.

Crida inicial: BINARI(n, 0).

```
\begin{array}{l} \text{BINARI}(n,i) \\ \text{(processar totes les cadenes binàries de mida } n \text{ amb} \leq k \text{ uns)} \\ \textbf{si } n = 0 \text{ llavors} \\ \text{PROCESSAR}(C) \\ \textbf{si no} \\ C[n] \leftarrow 0; \text{BINARI}(n-1,i) \\ \textbf{si } i < k \text{ llavors} \\ C[n] \leftarrow 1; \text{BINARI}(n-1,i+1) \end{array}
```

Per a n = 3 i k = 1, s'obté l'arbre de recursió:



És millor que generar totes les possibilitats i després comprovar els uns.

Es pot definir un algorisme genèric de tornada enrere:

- L'espai de solucions d'un problema s'acostuma a organitzar en forma d'arbre de configuracions
- Cada node o configuració de l'arbre es representa amb un vector

$$A=(a_1,a_2,\ldots,a_k)$$

que conté les tries ja fetes

- El vector A s'amplia en la fase avançar triant un  $a_{k+1}$  d'un conjunt de candidats  $S_{k+1}$  (explorar en profunditat)
- A es redueix en la fase retrocedir (backtrack)

```
TORNADA ENRERE(x)
    calcular S_1 // conjunt de candidats per a a_1
    k \leftarrow 1
    mentre k > 0
        mentre S_k \neq \emptyset
            a_k \leftarrow \text{un element de } S_k
            S_k \leftarrow S_k - \{a_k\}
            A \leftarrow (a_1, a_2, \ldots, a_k)
            si solució(A) llavors
                 PROCESSAR(A)
            k \leftarrow k + 1 // avançar
            calcular S_k // candidats per a a_k
        k \leftarrow k - 1 // \text{ retrocedir}
```

- Cost en temps: mida de l'arbre (normalment exponencial)
- Cost en espai: profunditat de l'arbre (normalment polinòmic)

### Exemple: permutacions de *n* elements

Quines són les permutacions dels naturals  $\{1, \ldots, n\}$ ?

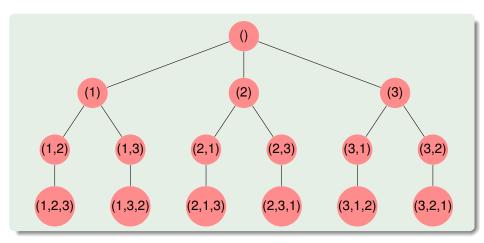
- Hi ha n possibilitats per al primer
- Fixat el primer, hi ha n-1 possibilitats per al segon
- Repetint el raonament, obtenim

$$\prod_{k=1}^{n} k = n!$$

### Adaptem l'algorisme genèric amb

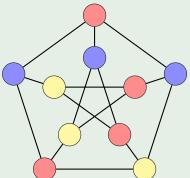
- $A = (a_1, ..., a_k)$ , tal que  $a_i$  és l'i-èsim element triat
- $S_1 = \{1, ..., n\}$  i  $S_{k+1} = \{1, ..., n\} A$  per a  $k \ge 1$

Amb n = 3, s'obté l'arbre de configuracions:



### Exemple: 3-Colorabilitat

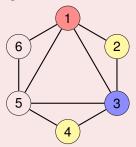
El problema de la **3-colorabilitat** consisteix a decidir si es pot assignar un color a cada vèrtex d'un total de 3 de manera que els vèrtexs adjacents tinguin colors diferents.



Una 3-coloració del graf de Petersen

#### Exercici

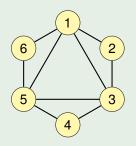
- Trobeu un algorisme polinòmic per decidir si un graf té una 2-coloració.
   (Jutge: Two colors)
- Què impedeix estendre l'algorisme anterior a la 3-colorabilitat? Concretament, donat el graf



què hauria de fer un algorisme que hagués assignat, per aquest ordre, el color vermell a 1, el groc a 2, el blau a 3 i el groc a 4?

#### 3-colorabilitat

Donat el graf

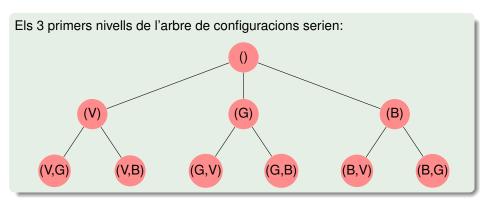


Les configuracions seran assignacions parcials de colors, és a dir,

$$A=(a_1,a_2,\ldots,a_k)$$

representarà el fet que el vèrtex i s'acoloreix amb el color  $a_i \in \{B, G, V\}$ 

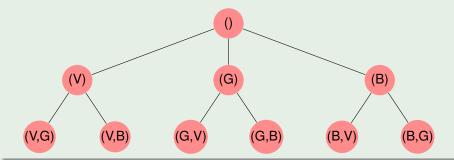
• El conjunt de candidats  $S_{k+1}$  per a  $a_{k+1}$  contindrà els colors compatibles amb els veïns que ja han estat acolorits



Però si el que volem és trobar només una solució,

- es pot fixar un color per al vèrtex 1
- es pot fixar un color diferent per al vèrtex 2
- qualsevol altra solució serà simètrica

Els 3 primers nivells de l'arbre de configuracions serien:



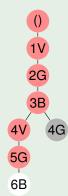
Però si el que volem és trobar només una solució,

- es pot fixar un color per al vèrtex 1
- es pot fixar un color diferent per al vèrtex 2
- qualsevol altra solució serà simètrica

#### Fent la tria

- $S_1 = \{V\}$
- $S_2 = \{G\}$

i definint  $S_{k+1} = \{c \in \{V, G, B\} \mid \forall i \leq k \ (\{i, k+1\} \in A(G) \Rightarrow c \neq a_i)\},$  s'obté l'arbre de configuracions



## Tema 6. Cerca exhaustiva

- Força bruta i cerca amb retrocés
  - Algorismes de força bruta
  - Cerca amb retrocés
  - Algorisme genèric
- 2 Problemes
  - La reconstrucció Turnpike
  - Les n reines
  - El quadrat llatí
  - Els salts de cavall
  - Graf hamiltonià
  - Problema del viatjant
  - La motxilla
- 3 El principi minimax

Suposem que hi ha *n* punts sobre l'eix *x* amb coordenades

$$x_1, x_2, \ldots, x_n$$
.

A més, suposem que tots els valors són naturals i

$$0=x_1\leq x_2\leq \cdots \leq x_n$$

Els n punts determinen n(n-1)/2 distàncies de la forma  $|x_i - x_j|$  per a  $i \neq j$  (no necessàriament diferents).

#### Observació

És fàcil construir el conjunt de distàncies a partir del conjunt de punts en temps  $\Theta(n^2)$  (ordenades, en temps  $\Theta(n^2 \log n)$ ).

#### Problema de la reconstrucció Turnpike

Donat el conjunt de distàncies, reconstruir el conjunt de punts.

Suposem que hi ha *n* punts sobre l'eix *x* amb coordenades

$$x_1, x_2, \ldots, x_n$$
.

A més, suposem que tots els valors són naturals i

$$0=x_1\leq x_2\leq \cdots \leq x_n$$

Els n punts determinen n(n-1)/2 distàncies de la forma  $|x_i - x_j|$  per a  $i \neq j$  (no necessàriament diferents).

#### Observació

És fàcil construir el conjunt de distàncies a partir del conjunt de punts en temps  $\Theta(n^2)$  (ordenades, en temps  $\Theta(n^2 \log n)$ ).

Problema de la reconstrucció Turnpike

Donat el conjunt de distàncies, reconstruir el conjunt de punts.

Suposem que hi ha *n* punts sobre l'eix *x* amb coordenades

$$X_1, X_2, \ldots, X_n$$
.

A més, suposem que tots els valors són naturals i

$$0=x_1\leq x_2\leq \cdots \leq x_n$$

Els n punts determinen n(n-1)/2 distàncies de la forma  $|x_i-x_j|$  per a  $i \neq j$  (no necessàriament diferents).

#### Observació

És fàcil construir el conjunt de distàncies a partir del conjunt de punts en temps  $\Theta(n^2)$  (ordenades, en temps  $\Theta(n^2 \log n)$ ).

#### Problema de la reconstrucció Turnpike

Donat el conjunt de distàncies, reconstruir el conjunt de punts.

Els problemes de reconstrucció acostumen a ser més complexos que els de construcció:

- multiplicar és més fàcil que factoritzar
- reconstruir un graf de n vèrtexs a partir dels seus subgrafs de n-1 vèrtexs és més complex que obtenir aquests subgrafs

Per al problema de la reconstrucció Turnpike

- o no es coneix cap algorisme polinòmic
- hi ha un algorisme que normalment funciona en temps  $O(n^2 \log n)$  però que en el cas pitjor és exponencial

Els problemes de reconstrucció acostumen a ser més complexos que els de construcció:

- multiplicar és més fàcil que factoritzar
- reconstruir un graf de n vèrtexs a partir dels seus subgrafs de n-1 vèrtexs és més complex que obtenir aquests subgrafs

Per al problema de la reconstrucció Turnpike

- no es coneix cap algorisme polinòmic
- hi ha un algorisme que normalment funciona en temps  $O(n^2 \log n)$  però que en el cas pitjor és exponencial

### Exemple amb $D = \{1, 2, 2, 2, 3, 3, 3, 4, 5, 5, 5, 6, 7, 8, 10\}$

Com que |D| = 15 = n(n-1)/2, obtenim n = 6.

- Comencem fent  $x_1 = 0$ .
- Clarament,  $x_6 = 10$ .
- Eliminem 10 de D. Ara,

$$D = \{1, 2, 2, 2, 3, 3, 3, 4, 5, 5, 5, 6, 7, 8\}.$$

### $D = \{1, 2, 2, 2, 3, 3, 3, 4, 5, 5, 5, 6, 7, 8\}, \text{ eix } x: 0 = 10$

La distància més gran que queda és 8. Per tant,

$$x_2 = 2$$
 o  $x_5 = 8$ .

Si tenen solució, seran simètriques. Triem  $x_5 = 8$ .

• Eliminem de *D* les distàncies  $x_6 - x_5 = 2$  i  $x_5 - x_1 = 8$ . Ara,

$$D = \{1, 2, 2, 3, 3, 3, 4, 5, 5, 5, 6, 7\}$$

### $D = \{1, 2, 2, 2, 3, 3, 3, 4, 5, 5, 5, 6, 7, 8\}, \text{ eix } x: 0 = 10$

• La distància més gran que queda és 8. Per tant,

$$x_2 = 2$$
 o  $x_5 = 8$ .

Si tenen solució, seran simètriques. Triem  $x_5 = 8$ .

• Eliminem de *D* les distàncies  $x_6 - x_5 = 2$  i  $x_5 - x_1 = 8$ . Ara,

$$D = \{1, 2, 2, 3, 3, 3, 4, 5, 5, 5, 6, 7\}.$$

## $D = \{1, 2, 2, 3, 3, 3, 4, 5, 5, 5, 6, 7\}, \text{ eix } x: 0_{---} 8 10$

- Com que 7 és el valor més alt a D,  $x_4 = 7$  o  $x_2 = 3$ .
  - Si  $x_4 = 7$ , les distàncies

$$x_6 - 7 = 3 i x_5 - 7 = 1$$

han de ser a D, i hi són.

• Si  $x_2 = 3$ , les distàncies

$$3 - x_1 = 3 i x_5 - 3 = 5$$

han de ser a *D*, i també hi són.

No tenim cap guia per triar. Provarem una opció ( $x_4 = 7$ ) i veurem si porta a una solució. Si no, tornarem enrere.

• Eliminem les distàncies  $x_4 - x_1 = 7$ ,  $x_5 - x_4 = 1$  i  $x_6 - x_4 = 3$ . Ara,

$$D = \{2, 2, 3, 3, 4, 5, 5, 5, 6\}$$

## $D = \{1, 2, 2, 3, 3, 3, 4, 5, 5, 5, 6, 7\}, \text{ eix } x: 0_{---} 8 10$

- Com que 7 és el valor més alt a D,  $x_4 = 7$  o  $x_2 = 3$ .
  - Si  $x_4 = 7$ , les distàncies

$$x_6 - 7 = 3 i x_5 - 7 = 1$$

han de ser a D, i hi són.

• Si  $x_2 = 3$ , les distàncies

$$3-x_1=3$$
 i  $x_5-3=5$ 

han de ser a D, i també hi són.

No tenim cap guia per triar. Provarem una opció ( $x_4 = 7$ ) i veurem si porta a una solució. Si no, tornarem enrere.

• Eliminem les distàncies  $x_4 - x_1 = 7$ ,  $x_5 - x_4 = 1$  i  $x_6 - x_4 = 3$ . Ara,

$$D = \{2, 2, 3, 3, 4, 5, 5, 5, 6\}.$$

### $D = \{2, 2, 3, 3, 4, 5, 5, 5, 6\}, \text{ eix } x: 0_{-} 7 8 10$

- La distància més gran és 6. Per tant,  $x_3 = 6$  o  $x_2 = 4$ .
  - Si  $x_3 = 6$ , llavors  $x_4 x_3 = 1$ , que no pertany a D.
  - Si  $x_2 = 4$ , llavors

$$x_2 - x_1 = 4$$
 i  $x_5 - x_2 = 4$ 

i això és impossible perquè 4 només apareix un cop a D.

Aquesta línia de raonament no porta a una solució. Tornem enrere i triem  $x_2 = 3$ .

En el conjunt de distàncies anteriors

$$D = \{1, 2, 2, 3, 3, 3, 4, 5, 5, 5, 6, 7\},\$$

eliminem  $x_2 - x_1 = 3$ ,  $x_5 - x_2 = 5$  i  $x_6 - x_2 = 7$ . Ara,

 $D = \{1, 2, 2, 3, 3, 4, 5, 5, 6\}.$ 

### $D = \{2, 2, 3, 3, 4, 5, 5, 5, 6\}, \text{ eix } x: 0_{-}7810$

- La distància més gran és 6. Per tant,  $x_3 = 6$  o  $x_2 = 4$ .
  - Si  $x_3 = 6$ , llavors  $x_4 x_3 = 1$ , que no pertany a D.
  - Si  $x_2 = 4$ , llavors

$$x_2 - x_1 = 4$$
 i  $x_5 - x_2 = 4$ 

i això és impossible perquè 4 només apareix un cop a D.

Aquesta línia de raonament no porta a una solució. Tornem enrere i triem  $x_2 = 3$ .

En el conjunt de distàncies anteriors

$$D = \{1, 2, 2, 3, 3, 3, 4, 5, 5, 5, 6, 7\},\$$

eliminem 
$$x_2 - x_1 = 3$$
,  $x_5 - x_2 = 5$  i  $x_6 - x_2 = 7$ . Ara,

$$D = \{1, 2, 2, 3, 3, 4, 5, 5, 6\}.$$

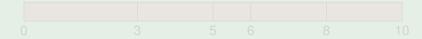
### $D = \{1, 2, 2, 3, 3, 4, 5, 5, 6\}, \text{ eix } x: 0.3 \_ 8.10$

- Com que 6 és el valor més alt a D,  $x_4 = 6$  o  $x_3 = 4$ .
  - Si  $x_3 = 4$ , tant  $x_3 x_1$  com  $x_5 x_3$  valdrien 4, però no és possible perquè D només conté un 4.

Per tant,  $x_4 = 6$  i obtenim

$$D = \{1, 2, 3, 5, 5\}.$$

• Només queda triar  $x_3 = 5$ . Com que ens queda  $D = \emptyset$ , tenim una solució:



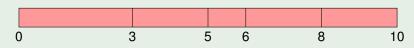
### $D = \{1, 2, 2, 3, 3, 4, 5, 5, 6\}, \text{ eix } x: 0.3 \_ 8.10$

- Com que 6 és el valor més alt a D,  $x_4 = 6$  o  $x_3 = 4$ .
  - Si  $x_3 = 4$ , tant  $x_3 x_1$  com  $x_5 x_3$  valdrien 4, però no és possible perquè D només conté un 4.

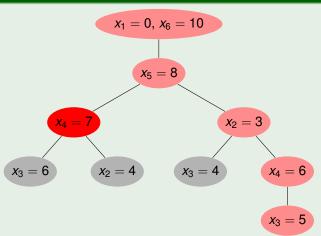
Per tant,  $x_4 = 6$  i obtenim

$$D=\{1,2,3,5,5\}.$$

Només queda triar x₃ = 5. Com que ens queda D = ∅, tenim una solució:



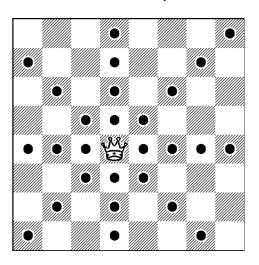
#### Arbre de decisió



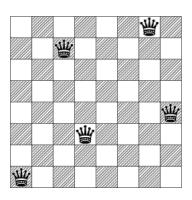
Els nodes grisos indiquen que els punts triats són inconsistents amb les dades. El node vermell no té nodes vàlids com a fills.

- Aquest mètode dona lloc a un algorisme que, si no es produeix cap tornada enrere, té cost O(n² log n)
- Per a punts aleatoris distribuïts de manera uniforme, es produeix com a molt una tornada enrere en tot l'algorisme
- Exemple, amb el codi en C++: Weiss, M.A., Data Structures and Algorithm Analysis in C++, 2a edició, Addison-Wesley, 1999

Moviments de la reina en el joc dels escacs:

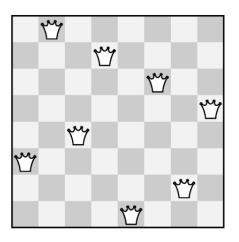


Quantes reines podem col·locar sobre un tauler sense que n'hi hagi dues que s'amenacin mútuament? 5? 6? 7? 8?



#### Problema de les vuit reines

Col·locar vuit reines en un tauler d'escacs sense que cap amenaci cap altra.



### Estratègies de resolució per força bruta:

Triar 8 posicions diferents del tauler:

$$\binom{64}{8} = 4.426.165.368$$
 configuracions

2 Triar 8 posicions en files diferents:

$$8^8 = 16.777.216$$
 configuracions

Triar 8 posicions en files i columnes diferents:

8! = 40.320 configuracions

Amb *backtracking* encara es pot millorar més.

#### Qüestió

Quin és el cost de qualsevol algorisme que resolgui el problema de les vuit reines?

Considerarem el problema generalitzat.

#### Problema de les *n* reines

Col·locar n reines en un tauler  $n \times n$  sense que cap amenaci cap altra.

#### Qüestió

Quin és el cost de qualsevol algorisme que resolgui el problema de les vuit reines?

Considerarem el problema generalitzat.

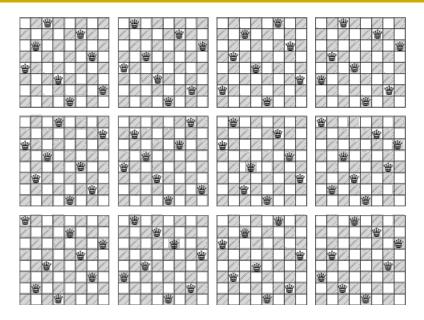
#### Problema de les n reines

Col·locar n reines en un tauler  $n \times n$  sense que cap amenaci cap altra.

Nombre de solucions no isomorfes (per rotació o reflexió) de les n reines per a  $n \in \{1, ..., 10\}$  :

	solucions
n	Solucions
1	1
2	0
3	0
4	1
5	2
6	1
7	6
8	12
9	46
10	92
	02

# Les 12 solucions no isomorfes per a n = 8



#### Exercici

Suposeu que hi ha una reina en la posició (1,2) (fila 1, columna 2) en un tauler  $4 \times 4$  i mostreu tots els escacs amenacats per la reina.

A partir d'aquí, deduïu regles per saber quines

- files
- columnes
- diagonals principals (\( \sigma \))
- diagonals secundàries ( )

estan amenaçades.

#### Primera implementació:

- amb tornada enrere
- amplia la solució parcial sempre que sigui "legal" (que es pugui estendre a una solució completa)
- cost en cas pitjor:  $\Theta(n^n)$

Implementarem la posició de les reines amb un vector

```
vector<int> T;
```

que indicarà que la reina de la fila i és a la columna T[i].

Per saber si les reines de les files *i* i *k* comparteixen

- columna, comprovem si T[i] = T[k]
- diagonal principal ( $\searrow$ ), comprovem si T[i] i = T[k] k
- diagonal secundària ( $\nearrow$ ), comprovem si T[i] + i = T[k] + k

```
class NReines {
   int n; // nombre de reines
   vector<int> T; // configuracio actual
   void recursiu(int i) {
       if (i == n) {
          escriure();
        } else {
           for (int j = 0; j < n; ++j) {
               T[i] = j;
               if (legal(i)) {
                   recursiu(i+1);
   } } }
```

```
bool legal(int i) {
// Indica si la config. amb les reines 0..i es legal
// sabent que la config. amb les reines 0..i-1 ho es
    for (int k = 0; k < i; ++k) {
        if (T[k] == T[i] or
            T[i]-i == T[k]-k or T[i]+i == T[k]+k) {
            return false;
    return true;
```

```
void escriure() {
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            cout << (T[i] == j ? "O" : "*") ;
        }
        cout << endl;
    }
    cout << endl;
}</pre>
```

```
public:
    NReines(int n) {
        this->n = n;
        T = vector<int>(n);
        recursiu(0);
    }
};
```

### Programa principal

```
int main() {
    int n = readint();
    NReines r(n);
}
```

### Segona implementació:

- amb tornada enrere
- amplia la solució parcial sempre que sigui "legal" (que es pugui estendre a una solució completa)
- amb marcatges
- cost en cas pitjor:  $\Theta(n^n)$

```
class NReines {
    int n; // nombre de reines
   vector<int> T; // congiguracio actual
   vector<br/>boolean> mc; // marca de les columnes
   vector<boolean> md1; // marca de les diagonals 1
   vector<boolean> md2; // marca de les diagonals 2
    inline int diag1(int i, int j) {
        return n-j-1 + i;
    inline int diag2(int i, int j) {
        return i+j;
```

```
void recursiu(int i) {
    if (i == n) {
        escriure();
    } else {
        for (int j = 0; j < n; ++j) {
            if (not mc[j] and not md1[diag1(i, j)]
                               and not md2[diag2(i, j)]) {
                T[i] = j;
                mc[j] = true;
                md1[diag1(i, j)] = true;
                md2[diag2(i, j)] = true;
                recursiu(i+1);
                mc[j] = false;
                md1[diag1(i, j)] = false;
                md2[diag2(i, j)] = false;
```

```
public:

NReines(int n) {
    this->n = n;
    T = vector<int>(n);
    mc = vector<boolean>(n, false);
    md1 = vector<boolean>(2*n-1, false);
    md2 = vector<boolean>(2*n-1, false);
    recursiu(0);
}
};
```

### Programa principal

```
int main() {
   int n = readint();
   NReines r(n);
}
```

#### Tercera implementació:

- amb tornada enrere
- amplia la solució parcial sempre que sigui "legal" (que es pugui estendre a una solució completa)
- amb marcatges
- amb un booleà per finalitzar la cerca
- cost en cas pitjor:  $\Theta(n^n)$

```
class NReines {
    int n: // nombre de reines
    vector<int> T; // configuracio actual
    bool trobat; // indica si ja s'ha trobat una solucio
    vector<br/>boolean> mc; // marca de les columnes
    vector<br/>
boolean> mdl; // marca de les diagonals 1
    vector<boolean> md2; // marca de les diagonals 2
    inline int diag1 (int i, int j) {
        return n-j-1 + i;
    inline int diag2 (int i, int j) {
        return i+j;
```

```
void recursiu(int i) {
    if (i == n) {
       trobat = true;
       escriure():
    } else {
        for (int j = 0; j < n and not trobat; ++j) {
            if (not mc[j] and not md1[diag1(i, j)]
                      and not md2[diag2(i, j)]) {
                T[i] = j;
                mc[j] = true;
                md1[diag1(i, j)] = true;
                md2[diag2(i, j)] = true;
                recursiu(i+1);
                mc[j] = false;
                md1[diag1(i, j)] = false;
                md2[diag2(i, j)] = false;
} } } }
```

```
public:

NReines(int n) {
    this->n = n;
    T = vector<int>(n);
    mc = vector<boolean>(n, false);
    md1 = vector<boolean>(2*n-1, false);
    md2 = vector<boolean>(2*n-1, false);
    trobat = false;
    recursiu(0);
} ;
```

#### Programa principal

```
int main() {
    int n = readint();
    NReines r(n);
}
```

#### Quadrat llatí

Un quadrat llatí és qualsevol quadrícula  $n \times n$  omplerta amb n símbols diferents cadascun dels quals apareix un cop a cada fila i cada columna.

1	2	2	3					
2	3	3	1					
3	1		2					
		4	Α					
		Е	3	Α		•		

Nombre de quadrats llatins  $n \times n$  per a  $n \in \{1, ..., 11\}$ :

n	solucions
1	1
2	2
3	12
4	576
5	161280
6	812851200
7	61479419904000
8	108776032459082956800
9	5524751496156892842531225600
10	9982437658213039871725064756920320000
11	776966836171770144107444346734230682311065600000

Solució per tornada enrere amb marcatges.

```
Cost: O(n^{n^2}).
```

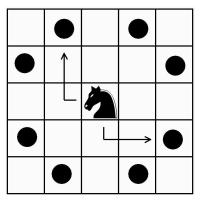
```
class QuadratLlati {
   int n; // nombre de files i columnes
   matrix<int> Q; // el quadrat llati
   matrix<boolean> F; // F[i][c]: c no apareix a la fila i
   matrix<boolean> C; // C[j][c]: c no apareix a la columna j
```

```
void recursiu(int cas) {
    if (cas == n*n) {
        cout << Q << endl;
    } else {
        int i = cas/n;
        int j = cas%n;
        for (int c = 0; c < n; ++c) {</pre>
            if (F[i][c] and C[j][c]) {
                Q[i][j] = c;
                F[i][c] = C[j][c] = false;
                recursiu(cas+1);
                F[i][c] = C[j][c] = true;
 } } }
```

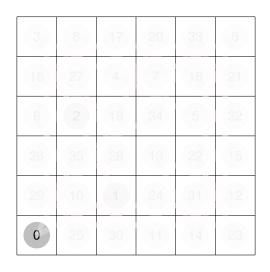
```
public:
    QuadratLlati(int n) {
        this->n = n;
        Q = matrix < int > (n, n);
        F = matrix<boolean>(n, n, true);
        C = matrix<boolean>(n, n, true);
        recursiu(0);
int main() {
    int n = readint();
    QuadratLlati ql(n);
```

#### Salts del cavall (knight's tour)

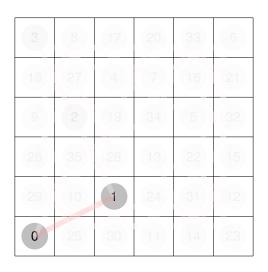
Donat un tauler  $n \times n$  i la posició d'una casella, trobar, si existeix, un recorregut del cavall d'escacs que visiti totes les caselles sense repeticions.



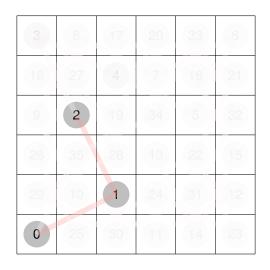
Els moviments del cavall



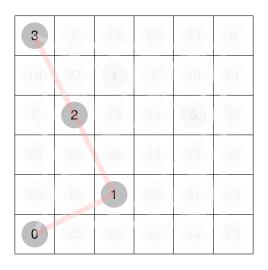
Una solució tancada per al tauler  $6 \times 6$ 



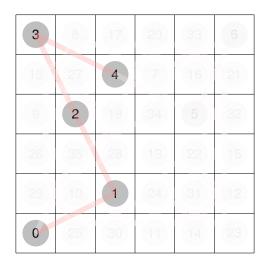
Una solució tancada per al tauler  $6 \times 6$ 



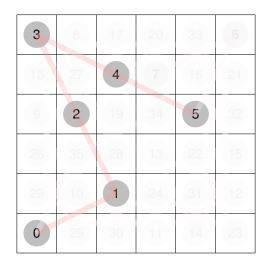
Una solució tancada per al tauler  $6 \times 6$ 



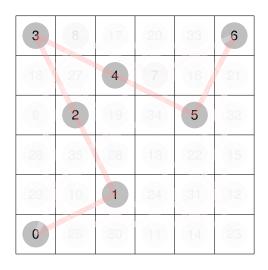
Una solució tancada per al tauler  $6 \times 6$ 



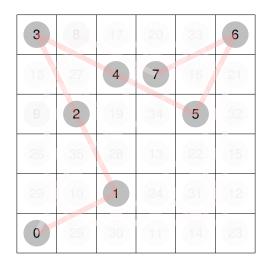
Una solució tancada per al tauler  $6 \times 6$ 



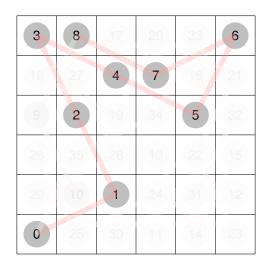
Una solució tancada per al tauler  $6 \times 6$ 



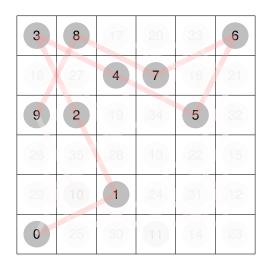
Una solució tancada per al tauler  $6 \times 6$ 



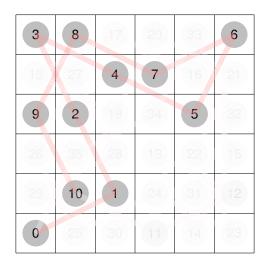
Una solució tancada per al tauler  $6 \times 6$ 



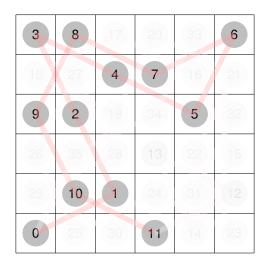
Una solució tancada per al tauler  $6 \times 6$ 



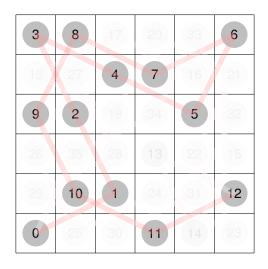
Una solució tancada per al tauler  $6 \times 6$ 



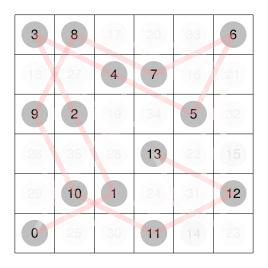
Una solució tancada per al tauler  $6 \times 6$ 



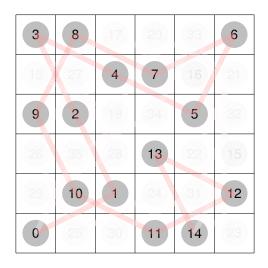
Una solució tancada per al tauler  $6 \times 6$ 



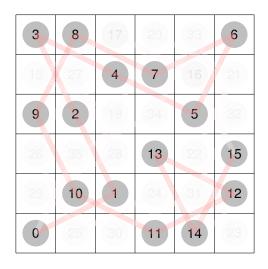
Una solució tancada per al tauler  $6 \times 6$ 



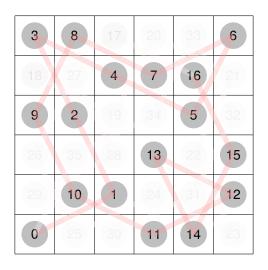
Una solució tancada per al tauler  $6 \times 6$ 



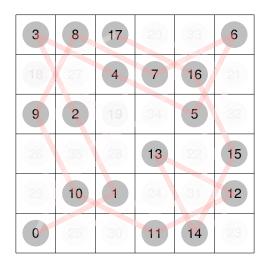
Una solució tancada per al tauler  $6 \times 6$ 



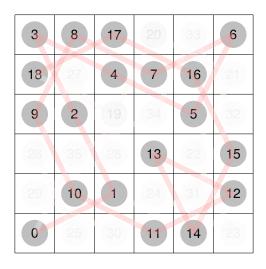
Una solució tancada per al tauler  $6 \times 6$ 



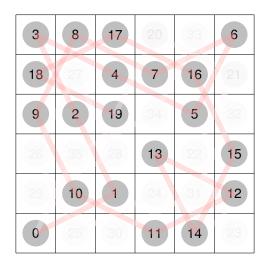
Una solució tancada per al tauler  $6 \times 6$ 



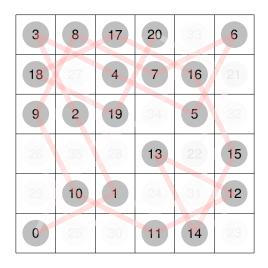
Una solució tancada per al tauler  $6 \times 6$ 



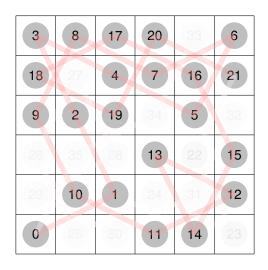
Una solució tancada per al tauler  $6 \times 6$ 



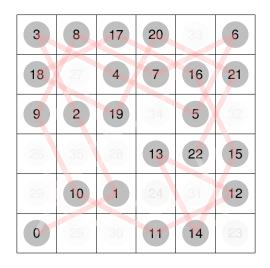
Una solució tancada per al tauler  $6 \times 6$ 



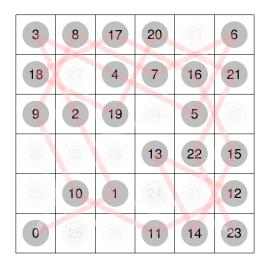
Una solució tancada per al tauler  $6 \times 6$ 



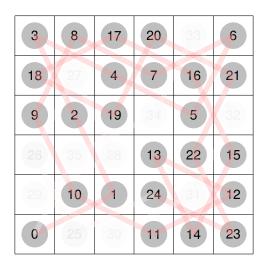
Una solució tancada per al tauler  $6 \times 6$ 



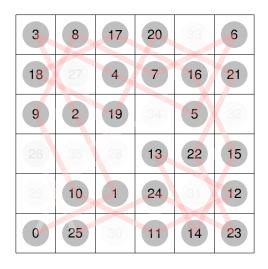
Una solució tancada per al tauler  $6 \times 6$ 



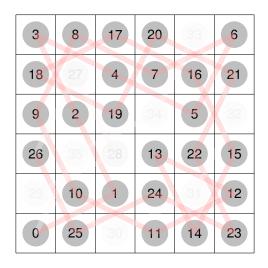
Una solució tancada per al tauler  $6 \times 6$ 



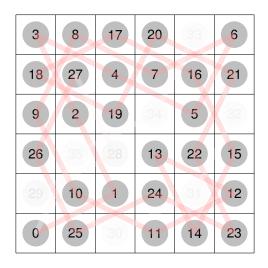
Una solució tancada per al tauler  $6 \times 6$ 



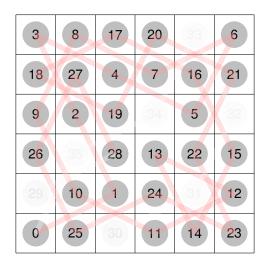
Una solució tancada per al tauler  $6 \times 6$ 



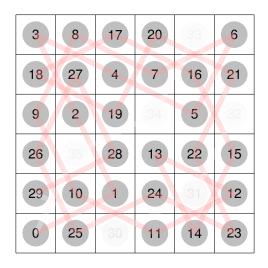
Una solució tancada per al tauler  $6 \times 6$ 



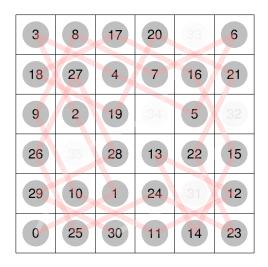
Una solució tancada per al tauler  $6 \times 6$ 



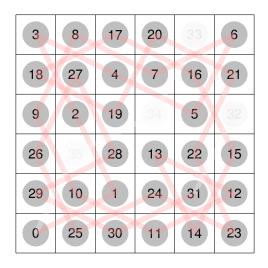
Una solució tancada per al tauler  $6 \times 6$ 



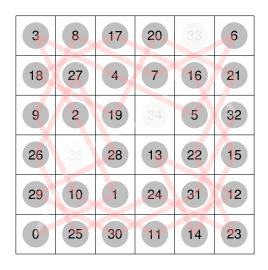
Una solució tancada per al tauler  $6 \times 6$ 



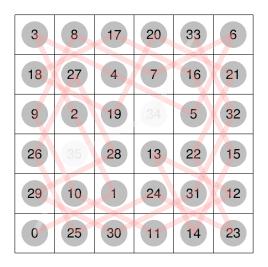
Una solució tancada per al tauler  $6 \times 6$ 



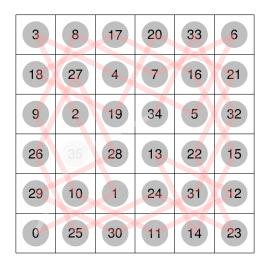
Una solució tancada per al tauler  $6 \times 6$ 



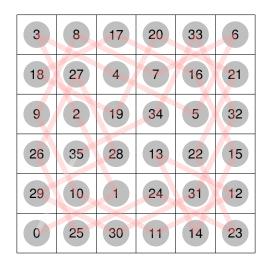
Una solució tancada per al tauler  $6 \times 6$ 



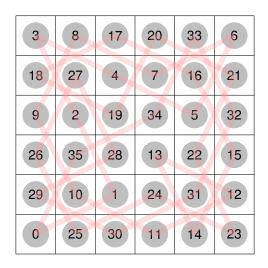
Una solució tancada per al tauler  $6 \times 6$ 



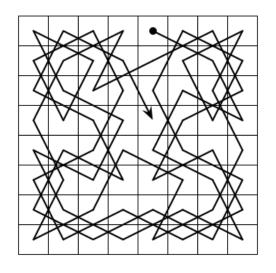
Una solució tancada per al tauler  $6 \times 6$ 



Una solució tancada per al tauler  $6 \times 6$ 



Una solució tancada per al tauler  $6 \times 6$ 



Una solució oberta per al tauler  $8\times 8$ 

El problema dels salts del cavall té una llarga tradició matemàtica. Prové de l'Índia (s. IX d.C.) i el va treballar Euler (s. XVIII).

- Hi ha dues versions:
  - tancada: inici i final a un salt de cavall
  - oberta: inici i final en posicions arbitràries
- Hi ha:
  - 9.862 tours tancats no dirigits en un tauler 6 × 6
  - ullet 13.267.364.410.532 tours tancats no dirigits en un tauler 8  $\times$  8
- En la majoria de taulers, es poden trobar tours en temps polinòmic amb l'estratègia de dividir i vèncer

Solució per tornada enrere. Cost  $O(2^{3n^2})$ .

Cada casella del tauler conté un enter:

- i > 0 si s'hi arriba en el salt i-èsim del cavall
- -1 si encara no s'ha considerat

```
class SaltsDeCavall {
    typedef matrix<int> matriu;

    int n; // nombre de files i columnes
    int ox,oy; // punt d'origen
    bool trobat; // ja s'ha trobat una solucio
    matriu M; // configuracio actual
    matriu S; // solucio (si trobat)
```

```
void recursiu(int pas, int x, int y) {
   if (pas == n*n-1) {
      trobat = true;
      S = M;
   } else {
      provar(pas, x+2, y-1); provar(pas, x+2, y+1);
      provar(pas, x+1, y+2); provar(pas, x-1, y+2);
      provar(pas, x-2, y+1); provar(pas, x-2, y-1);
      provar(pas, x-1, y-2); provar(pas, x+1, y-2);
}
```

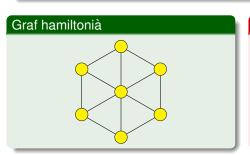
```
public:
    SaltsDeCavall(int n, int ox, int oy) {
        this->n = n;
        this->ox = ox;
        this->oy = oy;
        trobat = false;
        M = matriu(n, n, -1);
        M[ox][oy] = 0;
        recursiu(0, ox, oy);
    bool te solucio() {
        return trobat;
    matriu solucio() {
        return S;
```

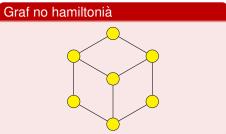
Programa principal. Una solució en  $6 \times 6$  es troba començant en (0,1).

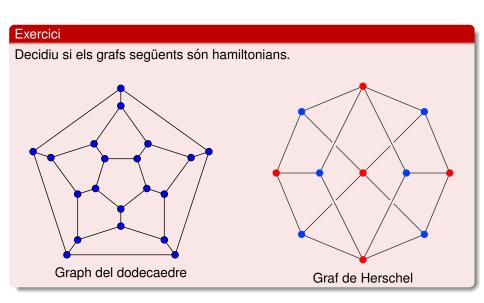
```
int main() {
    int n,ox,oy;
    cin >> n >> ox >> oy;
    SaltsDeCavall sc(n,ox,oy);
    if (sc.te_solucio()) cout << sc.solucio() << endl;
}</pre>
```

#### Definició

Un graf és hamiltonià si conté un cicle hamiltonià, és a dir, un cicle que visita cada vèrtex exactament un cop.







L'origen del terme *hamiltonià* aplicat a grafs i cicles és l'*Icosian game* inventat pel matemàtic William R. Hamilton el 1857.



#### Problema del Graf Hamiltonià

Donat un graf, determinar si és hamiltonià.

#### Solució per backtracking

La nostra solució assumeix:

- que el graf donat és connex
- que les llistes d'adjacència estan ordenades

#### Classe GraphHamiltonia

#### Problema del Graf Hamiltonià

Donat un graf, determinar si és hamiltonià.

#### Solució per backtracking

La nostra solució assumeix:

- que el graf donat és connex
- que les llistes d'adjacència estan ordenades

#### Classe GraphHamiltonia

#### Problema del Graf Hamiltonià

Donat un graf, determinar si és hamiltonià.

#### Solució per backtracking

La nostra solució assumeix:

- que el graf donat és connex
- que les llistes d'adjacència estan ordenades

#### Classe GraphHamiltonia

```
void recursiu(int v, int t) {
    // v = darrer vertex del cami
    // t = nombre de vertexs del cami
    if (t == n) {
        // comprova que es pot tancar el cicle
        if (not G[v].empty() and G[v][0] == 0) {
            s[v] = 0;
            trobat = true;
            S = s;
            s[v] = -1;
    } else {
        for (int u : G[v]) {
            if (s[u] == -1) {
                s[v] = u;
                recursiu(u, t+1);
                s[v] = -1;
                if (trobat) return;
    } } }
```

#### public:

```
GrafHamiltonia(Graph G) {
        this->G = G;
        n = G.size();
        s = vector < int > (n, -1);
        trobat = false;
        recursiu(0, 1);
    bool te_solucio() {
        return trobat;
    vector<int> solucio() {
        return S;
};
```

#### Programa principal

La funció llegir\_graf() llegeix i retorna el graf. El programa principal llegeix el graf, crea el solucionador i l'executa.

```
int main() {
    GrafHamiltonia ham(llegir_graf());
    if (ham.te_solucio()) {
        vector<int> s = ham.solucio();
        cout << 0 << "_";
        for (int u = s[0]; u != 0; u = s[u]) {
            cout << u << "_";
        }
        cout << endl;
    }
}</pre>
```

#### Problema del Viatjant (TSP, de l'anglès Traveling Salesman Problem)

Un viatjant visita els clients de *n* ciutats diferents. L'objectiu del viatjant és sortir de la seva ciutat, visitar exactament un cop cadascuna de les altres i tornar al punt de partida, tot minimitzant la distància total del viatge.

#### Distància mètrica

Donats n punts sobre el pla i una mesura de distància d on  $d_{ij} \ge 0$  representa la distància de i a j, diem que la distància és mètrica si:

- és *simètrica*, és a dir, per a tot  $i, j, d_{ij} = d_{ji}$  i
- compleix la *desigualtat triangular*, és a dir, per a tot  $i, j, k, d_{ik} \le d_{ij} + d_{jk}$

#### Problema del Viatjant Mètric (*Metric TSP*)

Resoldre el problema del viatjant quan la distància entre les ciutats és mètrica.

#### Problema del Viatjant (TSP, de l'anglès Traveling Salesman Problem)

Un viatjant visita els clients de *n* ciutats diferents. L'objectiu del viatjant és sortir de la seva ciutat, visitar exactament un cop cadascuna de les altres i tornar al punt de partida, tot minimitzant la distància total del viatge.

#### Distància mètrica

Donats n punts sobre el pla i una mesura de distància d on  $d_{ij} \ge 0$  representa la distància de i a j, diem que la distància és mètrica si:

- és simètrica, és a dir, per a tot  $i, j, d_{ij} = d_{ji}$  i
- compleix la *desigualtat triangular*, és a dir, per a tot  $i, j, k, d_{ik} \le d_{ij} + d_{jk}$

#### Problema del Viatjant Mètric (Metric TSP)

Resoldre el problema del viatjant quan la distància entre les ciutats és mètrica.





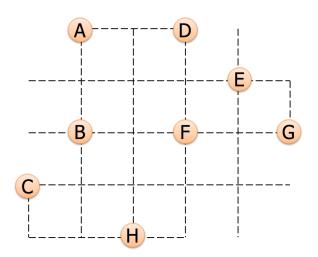
#### Aproximació per al Viatjant Mètric

És fàcil produir en temps polinòmic una solució amb cost el doble de l'òptim en el cas pitjor:

- 1 trobar un arbre d'expansió mínim
- g fer un recorregut de l'arbre
- Obtenir un cicle hamiltonià aplicant dreceres

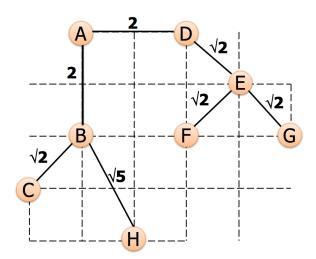
Mitjançant l'algorisme de Christofides es pot obtenir una solució el 50% més llarga que l'òptima en el cas pitjor. Els algorismes d'aquesta mena se'n diuen *d'aproximació*.

# Algorithm example Graph with N=8 nodes

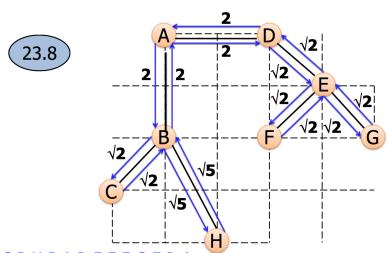


# Algorithm example Minimum spanning tree

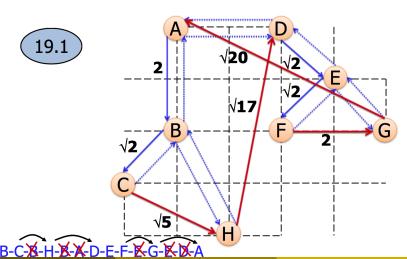
11.9



# Algorithm example Tree traversal

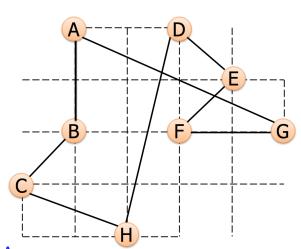


# Algorithm example Shortcuts



# Algorithm example Final result

19.1



### Cost de l'aproximació

- un arbre d'expansió mínim S
- fer un recorregut de l'arbre
- obtenir un cicle hamiltonià T aplicant dreceres

Sigui  $T^*$  un recorregut òptim del problema i e una aresta arbitrària de  $T^*$ . Llavors,

$$d(T) \leq 2d(S) \leq 2d(T^* - e) \leq 2d(T^*).$$

#### Qüestió

Trobeu una ruta més curta per a l'exemple anterior i comproveu que la solució de l'algorisme té un cost màxim del doble que la vostra.

#### Solució per tornada enrere

- troba una ruta òptima
- amb cost exponencial

```
typedef matrix<double> matriu_distancies;

class TSP {
   matriu_distancies M; // matriu de distancies
   int n; // nombre de ciutats
   vector<int> s; // seguent de cada ciutat (-1 si no usat)
   vector<int> sol_opt; // millor solucio fins ara
   double cost_opt; // cost de la millor solucio fins ara
```

```
void recursiu(int v, int t, double c) {
    // v = darrer vertex del cami
    // t = nombre de vertexs del cami
    // c = cost fins ara
    if (t == n) {
        c += M[v][0];
        if (c < cost_opt) {</pre>
            cost_opt = c;
            sol_opt = s;
            sol_opt[v] = 0;
    } else {
        for (int u = 0; u < n; ++u) {
            if (u != v and s[u] == -1) {
                if (c + M[v][u] < cost_opt) {
                     s[v] = u;
                     recursiu(u, t+1, c+M[v][u]);
                     s[v] = -1;
} } } }
```

#### public:

```
TSP (matriu_distancies M) {
    this->M = M;
    n = M.rows();
    s = vector<int>(n, -1);
    sol_opt = vector<int>(n);
    cost_opt = infinity;
    recursiu(0, 1, 0);
}
```

#### Programa principa

- llegeix n
- crea una matriu de distàncies M amb ciutats situades a l'atzar
- executa recursiu (definint TSP tsp(M))
- escriu cost\_opt (el cost de la solució òptima)

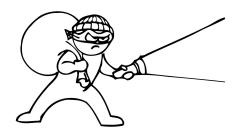
#### public:

```
TSP (matriu_distancies M) {
    this->M = M;
    n = M.rows();
    s = vector<int>(n, -1);
    sol_opt = vector<int>(n);
    cost_opt = infinity;
    recursiu(0, 1, 0);
}
```

#### Programa principal

- llegeix n
- o crea una matriu de distàncies M amb ciutats situades a l'atzar
- executa recursiu (definint TSP tsp (M))
- escriu cost\_opt (el cost de la solució òptima)

Suposem que un lladre vol entrar en una botiga i carregar al seu sac una combinació d'objectes amb el màxim valor total.



#### Com pot trobar la millor combinació fent ús de l'algorísmia?

- fent la llista dels pesos i valors dels objectes i sabent quant pot carregar
- definint bé el problema
- escrivint un algorisme que maximitzi el guany

Suposem que un lladre vol entrar en una botiga i carregar al seu sac una combinació d'objectes amb el màxim valor total.



Com pot trobar la millor combinació fent ús de l'algorísmia?

- fent la llista dels pesos i valors dels objectes i sabent quant pot carregar
- definint bé el problema
- 3 escrivint un algorisme que maximitzi el guany

El segon pas és definir bé el problema.

#### Problema de la motxilla (entera)

Donada una motxilla que pot carregar un pes C i n objectes amb

- $\bullet$  pesos  $p_1, p_2, \ldots, p_n$
- i valors  $v_1, v_2, \dots, v_n$

trobar una selecció  $S \subseteq \{1, ..., n\}$  dels objectes

- amb el màxim valor  $\sum_{i \in S} v_i$
- que no superi la capacitat de la motxilla:

$$\sum_{i\in\mathcal{S}}p_i\leq C$$

Tercer pas: escriure un primer algorisme en C++.

```
void recursiu (int i, double val, double pes) {
    // i = objecte que toca tractar
    // val = valor acumulat
    // pes = pes acumulat
    if (i == n) {
        if (val > millor) {
            millor = val;
            sol = s;
    } else {
        // la possibilitat: intentar agafar l'objecte i
        if (pes+p[i] <= C) {
            s[i] = true;
            recursiu(i+1, val+v[i], pes+p[i]);
        // 2a possibilitat: no agafar l'objecte i
        s[i] = false;
        recursiu(i+1, val, pes);
```

```
public:
  Motxilla (int n, vector<double> p, vector<double> v,
                 double C) {
      this->n = n;
      this->p = p;
      this -> v = v;
      this->C = C;
      s = sol = vector<boolean>(n);
      millor = 0;
      recursiu(0, 0, 0);
  vector<boolean> solucio () {
      return sol; }
  double cost () {
      return millor; }
};
```

El programa principal llegeix el nombre d'objectes, s'inventa els pesos, els valors i la capacitat, crea el solucionador, l'executa i n'escriu la solució.

```
int main() {
   int n = readint();
   vector<double> p = randvector(n);
   vector<double> v = randvector(n);
   double C = 0.4*n;
   cout << v << endl << p << endl << C << endl;

   Motxilla motx(n, p, v, C);
   cout << motx.cost() << endl;
   cout << motx.solucio() << endl;
}</pre>
```

Quart pas: millorar l'algorisme.

Solució amb fita superior. Aquest cop es té en compte la contribució màxima que podrien arribar a tenir tots els objectes a partir de l'i + 1 (encara que no càpiguen a la motxilla).

Si fins i tot agafant-los tots no es pogués superar el millor cost trobat fins ara, no cal seguir per aquell camí.

```
void recursiu (int i, double val, double pes) {
    // i = objecte que toca tractar
    // val = valor acumulat, pes = pes acumulat
    if (i == n) {
        if (val > millor) {
            millor = val;
            sol = s;
    } else {
        // la possibilitat: intentar agafar l'objecte i
        if (pes+p[i] <= C and val+sv[i] > millor) {
            s[i] = true;
            recursiu(i+1, val+v[i], pes+p[i]);
        // 2a possibilitat: no agafar l'objecte i
        if (val+sv[i+1] > millor) {
            s[i] = false;
            recursiu(i+1, val, pes);
```

```
Són com abans: solucio, cost, main.
public:
  Motxilla (int n, vector<double> p, vector<double> v,
                  double C) {
      this -> n = n;
      this->p = p;
      this->\forall = \forall;
      this->C = C;
      s = sol = vector<boolean>(n);
      millor = 0;
      sv = vector<double>(n+1);
      sv[n] = 0;
      for (int i = n-1; i >= 0; --i) {
          sv[i] = sv[i+1] + v[i];
      recursiu(0, 0, 0);
```

### Problema de la motxilla fraccional

Donada una motxilla que pot carregar un pes C i n productes amb

- $\bullet$  pesos  $p_1, p_2, \ldots, p_n$
- i valors  $v_1, v_2, \ldots, v_n$

trobar fraccions  $x_1, \ldots, x_n \in [0, 1]$  dels productes

- amb màxim valor  $\sum_{i=1}^{n} x_i v_i$
- que no superin la capacitat de la motxilla:

$$\sum_{i\in\mathcal{S}}x_ip_i\leq C$$

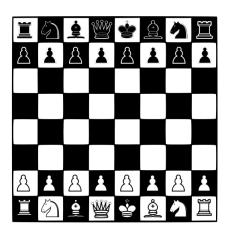
#### Exercici

- 1 Trobeu un algorisme polinòmic per al problema anterior
- Adapteu l'algorisme a la motxilla entera i digueu si és correcte

### Tema 6. Cerca exhaustiva

- 1 Força bruta i cerca amb retrocés
  - Algorismes de força bruta
  - Cerca amb retrocés
  - Algorisme genèrio
- 2 Problemes
  - La reconstrucció Turnpike
  - Les *n* reines
  - El quadrat llatí
  - Els salts de caval
  - Graf hamiltonià
  - Problema del viatjant
  - La motxilla
- 3 El principi minimax

En un joc com els escacs, no es pot pretendre fer una anàlisi exhaustiva de l'arbre de configuracions.



Es calculen 318.979.564.000 maneres de fer els 4 primers moviments

L'heurística minimax realitza una cerca parcial amb la qual no pot assegurar una estratègia guanyadora però...

- troba una de les millors jugades
- imitant el mètode dels jugadors humans

Suposem que tenim les blanques i que volem jugar una bona partida.

El primer pas és definir una funció estàtica d'avaluació AVALUA(u) que assigni un valor a cada situació u.

- Ha de donar valors més alts com més favorable és u per a les blanques
  - valors propers a 0 quan no hi ha avantatge definit
  - grans valors negatius si les negres tenen avantatge
- Ha de tenir en compte factors com
  - el nombre i tipus de peces blanques i negres que queden
  - qui amenaça més peces contràries
  - qui controla el centre
  - la llibertat de moviment
- Al final de la partida, ha de donar
  - $+\infty$  si hi ha escac i mat a les negres
  - 0 si acaba en taules
  - $-\infty$  si hi ha escac i mat a les blanques

### Exemple de funció d'avaluació AVALUA

- si hi ha escac i mat a les negres:  $+\infty$
- si hi ha escac i mat a les blanques:  $-\infty$
- si no, partint de 0 punts sumem
  - +1 (-1) punt per cada peó blanc (negre)
  - $+3\frac{1}{4}(-3\frac{1}{4})$  punts per cada alfil o cavall blanc (negre)
  - +5 (-5) punts per cada torre blanca (negra)
  - +10 (-10) punts per la reina blanca (negra)

Si la funció AVALUA(u) fos perfecta, només caldria calcular-la per cada moviment possible de les blanques (en 1 torn).

El millor moviment seria el que portés a una situació v on AVALUA(v) sigui màxim entre les posicions v successores de u.

```
MINIMAX(u)
val \leftarrow -\infty

per a cada posició w successora de u fer

si AVALUA(w) \geq val llavors
val \leftarrow \text{AVALUA}(w)
v \leftarrow w

retornar v
```

Aquesta funció sacrificaria la reina per capturar un peó!

Si la funció AVALUA(u) fos perfecta, només caldria calcular-la per cada moviment possible de les blanques (en 1 torn).

El millor moviment seria el que portés a una situació v on AVALUA(v) sigui màxim entre les posicions v successores de u.

```
MINIMAX(u)
val \leftarrow -\infty

per a cada posició w successora de u fer

si AVALUA(w) \geq val llavors
val \leftarrow \text{AVALUA}(w)
v \leftarrow w

retornar v
```

Aquesta funció sacrificaria la reina per capturar un peó!

Preveurem la resposta de les negres: suposarem que les negres voldran minimitzar AVALUA.

```
 \begin{array}{l} \operatorname{MINIMAX}(u) \\ val \leftarrow -\infty \\ \textbf{per a cada} \ \operatorname{posici\acute{o}} \ w \ \operatorname{successora} \ \operatorname{de} \ u \ \textbf{fer} \\ \textbf{si} \ w \ \operatorname{no} \ \operatorname{t\'e} \ \operatorname{successor} \ \textbf{llavors} \\ valw \leftarrow \operatorname{AVALUA}(w) \\ \textbf{si no} \\ valw \leftarrow \min \{ \ \operatorname{AVALUA}(x) \mid x \ \operatorname{\'es} \ \operatorname{successor} \ \operatorname{de} \ w \ \} \\ \textbf{si} \ valw \geq val \ \textbf{llavors} \\ val \leftarrow valw \\ v \leftarrow w \\ \textbf{retornar} \ v \end{array}
```

Aquesta funció mai no considerarà perdre una peça, tot i que sigui rendible.

Preveurem la resposta de les negres: suposarem que les negres voldran minimitzar AVALUA.

```
 \begin{array}{l} \mathsf{MINIMAX}(u) \\ \mathit{val} \leftarrow -\infty \\ \mathbf{per\ a\ cada\ } \mathsf{posicio}\ \mathit{w\ } \mathsf{successora\ } \mathsf{de\ } \mathit{u\ } \mathsf{fer} \\ \mathbf{si\ } \mathit{w\ } \mathsf{no\ } \mathsf{t\'e\ } \mathsf{successor\ } \mathsf{llavors} \\ \mathit{valw} \leftarrow \mathsf{AVALUA}(\mathit{w}) \\ \mathbf{si\ } \mathsf{no} \\ \mathit{valw} \leftarrow \mathsf{min} \{ \ \mathsf{AVALUA}(x) \ | \ \mathit{x\ } \mathsf{\acute{e}s\ } \mathsf{successor\ } \mathsf{de\ } \mathit{w\ } \} \\ \mathbf{si\ } \mathit{valw} \geq \mathit{val\ } \mathsf{llavors} \\ \mathit{val} \leftarrow \mathit{valw} \\ \mathit{v} \leftarrow \mathit{w} \\ \mathbf{retornar\ } \mathit{v} \\ \end{array}
```

Aquesta funció mai no considerarà perdre una peça, tot i que sigui rendible.

És preferible examinar més jugades per avançat! Examinarem n jugades.

```
MINIMAX(u, n)
val \leftarrow -\infty

per a cada posició w successora de u fer
B \leftarrow \text{NEGRES}(w, n)

si B \geq val llavors
val \leftarrow B
v \leftarrow w

retornar v
```

Les negres intenten minimitzar el guany de les blanques:

```
\begin{aligned} & \text{NEGRES}(w,n) \\ & \text{si } n = 0 \text{ o } w \text{ no t\'e successor llavors} \\ & & \text{retornar AVALUA}(w) \\ & \text{si no} \\ & & \text{retornar min} \{ \text{ BLANQUES}(x,n-1) \mid x \text{ \'es successor de } w \} \end{aligned}
```

Les blanques intenten maximitzar el seu guany:

```
\begin{aligned} \mathbf{BLANQUES}(w,n) \\ \mathbf{si} \ n &= 0 \ \mathbf{o} \ w \ \text{no t\'e successor Ilavors} \\ \mathbf{retornar} \ \mathbf{AVALUA}(w) \\ \mathbf{si no} \\ \mathbf{retornar} \ \max \{ \ \mathbf{NEGRES}(x,n-1) \mid x \ \acute{\mathbf{e}} \mathbf{s} \ \mathbf{successor} \ \mathbf{de} \ w \} \end{aligned}
```

