Tema 6. Cerca exhaustiva

Estructures de Dades i Algorismes

FIB

Transparències d' Antoni Lozano
(amb edicions menors d'altres professors)

Q1 2019 – 20

Tema 6. Cerca exhaustiva

Tema 6. Cerca exhaustiva

Algorismes de força bruta

- Molts problemes consisteixen en, donat un conjunt de restriccions, trobar un objecte que les satisfà (una solució)
- Per exemple, resoldre un sudoku.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

- Hi ha variacions:
  - trobar/comptar totes les solucions
  - trobar la millor de totes les solucions (solució òptima)
  - etc.

Algorismes de força bruta

Sovint l'única forma de resoldre aquests problemes és provar totes les possibilitats. D'això en diem força bruta o cerca exhaustiva:

- Acostuma a ser exponencial.
- Pot ser lenta, però millor que res...
- Pot arribar a ser pràctica amb entrades petites.
- Es pot ajudar d'altres tècniques
  (com dividir i vèncer, algorismes voraços, etc.).

Algorismes de força bruta

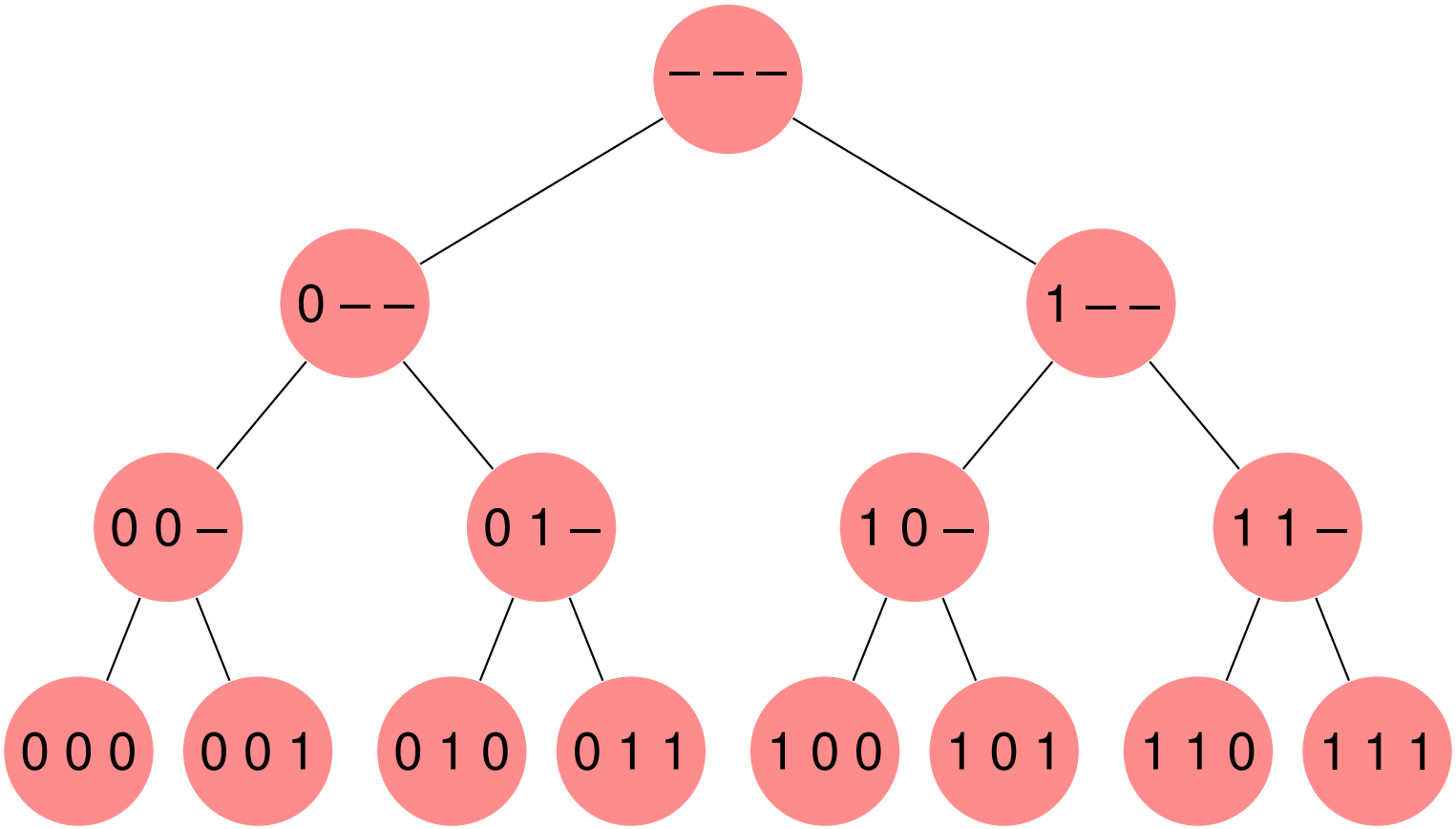
Suposem que volem processar (p. ex. escriure, comptar, o afegir a una llista) totes les cadenes de zeros i uns de mida n.

Tenim un procediment `processa(vector<int>& A)` que tracta el vector `A`. Llavors es crida `binari(0, A)`, on $n =$ `A.size()` i `binari` es defineix així:

```
// i es el nombre de valors ja assignats
// i es tambe la seguent posicio del vector A que assignarem
void binari(int i, vector<int>& A) {
  if (i == A.size()) processa(A);     // cas base
  else {                              // cas inductiu
    A[i] = 0; binari(i+1);
    A[i] = 1; binari(i+1);
} }
```

Algorismes de força bruta

Per a $n = 3$, s'obté el següent arbre de recursió:



Les fulles són solucions.

Les arestes indiquen com estenem cada solució parcial.

Els nodes interns són solucions parcials.

Algorismes de força bruta

Quin cost té la cerca exhaustiva?

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

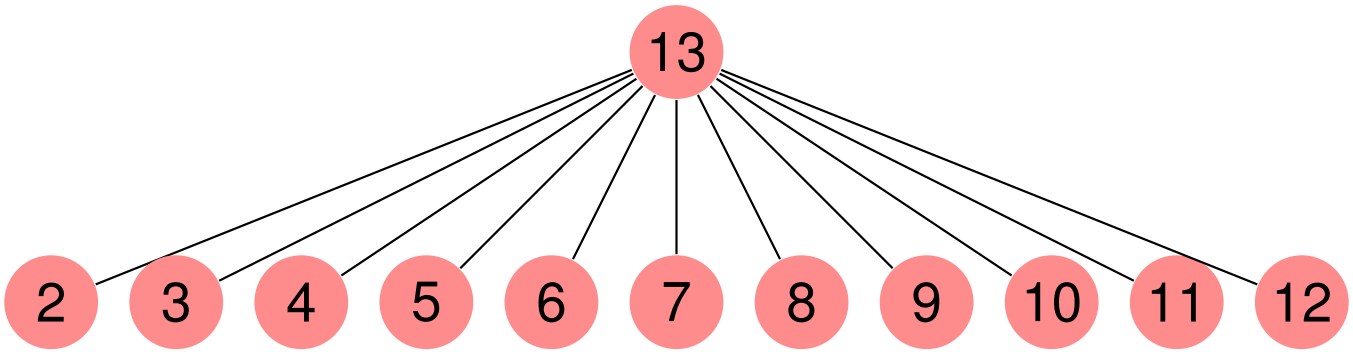
  Per exemple, les cerques en profunditat i amplada en grafs també són cerques exhaustives

Algorismes de força bruta

Exemple: primers

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

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



Arbre implícit per a $x = 13$

Backtracking

Un algorisme de backtracking funciona com una cerca exhaustiva, però no continua quan veu que una solució parcial no es pot estendre a una solució

Els algorismes de backtracking són més eficients que una simple cerca exhaustiva, però el cost és sovint encara exponencial.

En català, *backtracking* es tradueix per:

- tornada enrere
- cerca amb retrocés

Exemple: moblar un pis

- Estratègia de força bruta: provar totes les configuracions dels mobles en tots els espais.
- L'estratègia de backtracking usa 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)

Backtracking

Volem totes les cadenes de zeros i uns de mida n que contenen k uns.

Podem modificar l'algorisme vist abans de manera que eviti la recursió dels subarbres que contenen més de k uns i més de $n - k$ zeros

Crida inicial: `binari(0, A, 0, 0)`, on $n =$ `A.size()` i

```
// u es el nombre de 1's que porto
// z es el nombre de 0's que porto
void binari(int i, vector<int>& A, int u, int z) {
  if (i == A.size()) processa(A);
  else {
    if (z < n-k) {
      A[i] = 0; binari(i+1, u, z+1);
    }
    if (u < k) {
      A[i] = 1; binari(i+1, u+1, z);
} } }
```

Backtracking

Volem totes les cadenes de zeros i uns de mida n que contenen $\leq k$ uns.

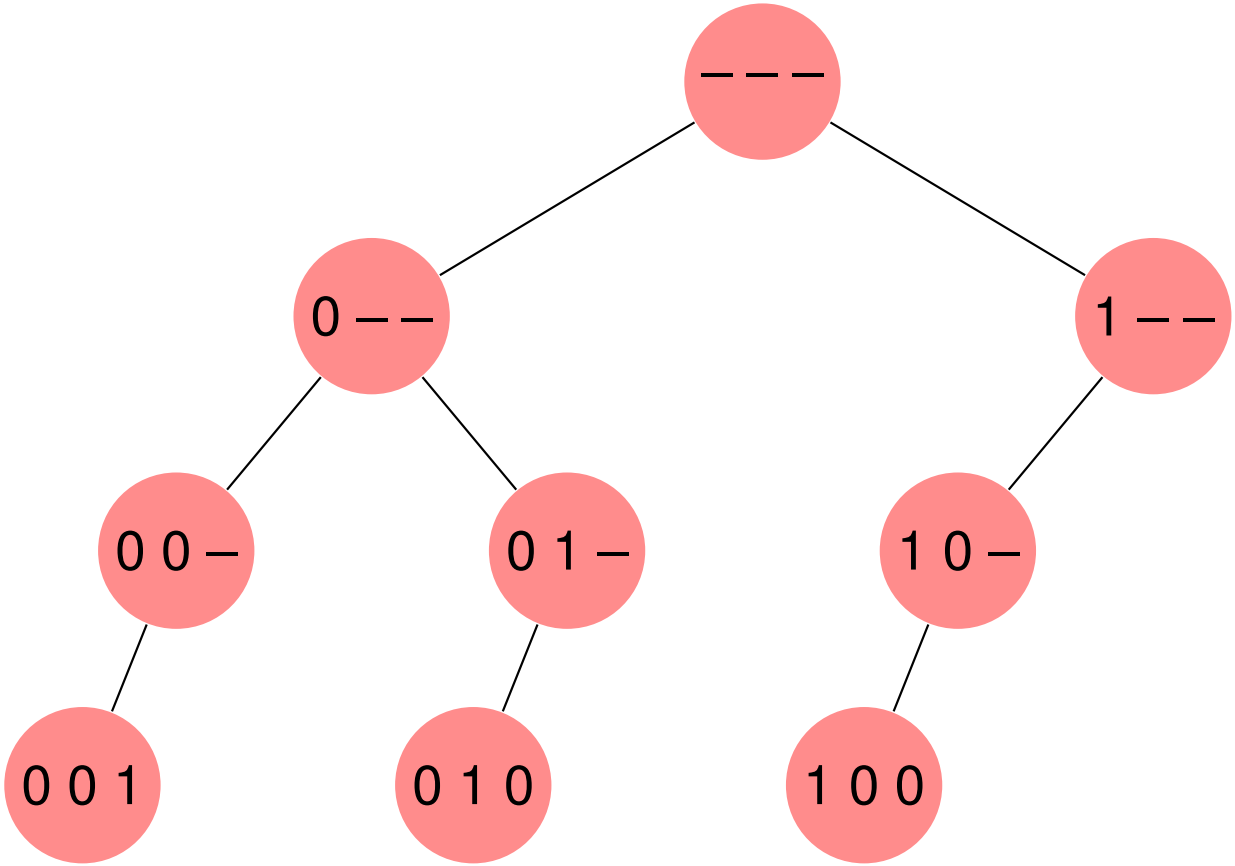
Podem modificar l'algorisme vist abans de manera que eviti la recursió dels subarbres que contenen més de k uns.

Crida inicial: `binari(0, A, 0, 0)`, on $n =$ `A.size()` i

```
// u es el nombre de 1's que porto
// z es el nombre de 0's que porto
void binari(int i, vector<int>& A, int u, int z) {
  if (u > k or z > n-k) return;
  if (i == A.size()) processa(A);
  else {
    A[i] = 0; binari(i+1, u, z+1);
    A[i] = 1; binari(i+1, u+1, z);
} }
```

Backtracking

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.

Però encara hi ha un nombre exponencial de nodes.

Algorisme genèric

Es pot definir un algorisme genèric de tornada enrere:

- L'espai de solucions (parcials) 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

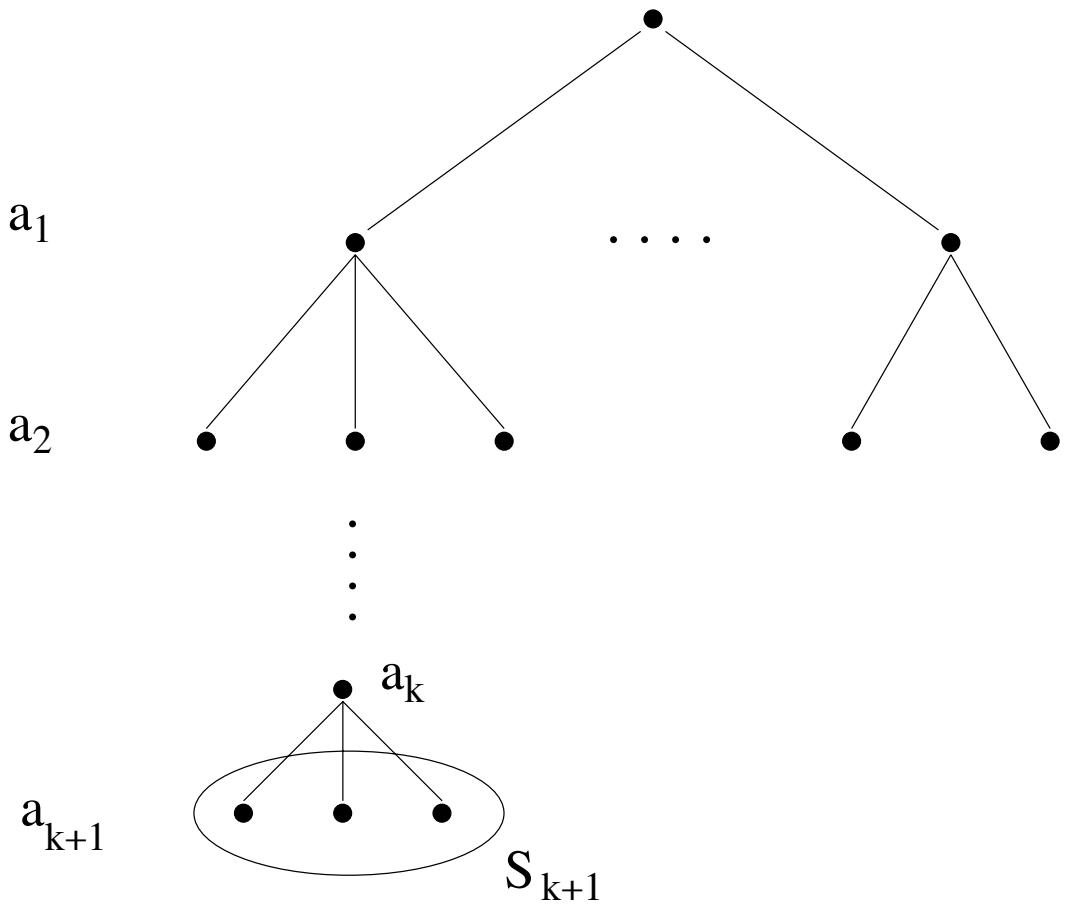
$$A = (a_1, a_2, \ldots, a_k)$$

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).

Algorisme genèric

Un algorisme de tornada enrere és sovint una cerca en profunditat en un arbre de configuracions:



Algorisme genèric

```
// i es el nombre de valors ja assignats
// i es tambe la seguent posicio de A que assignarem
void tornada_enrere(int i, vector<T>& A) {
  if (not es_pot_estendre_a_solucio(i, A)) return;
  if (i == A.size()) {
    if (es_solucio(A)) processa(A);
  }
  else {
    for (T v : S_i) {
      A[i] = v;
      tornada_enrere(i+1, A);
    }
  }
}
```

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

Algorisme genèric

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

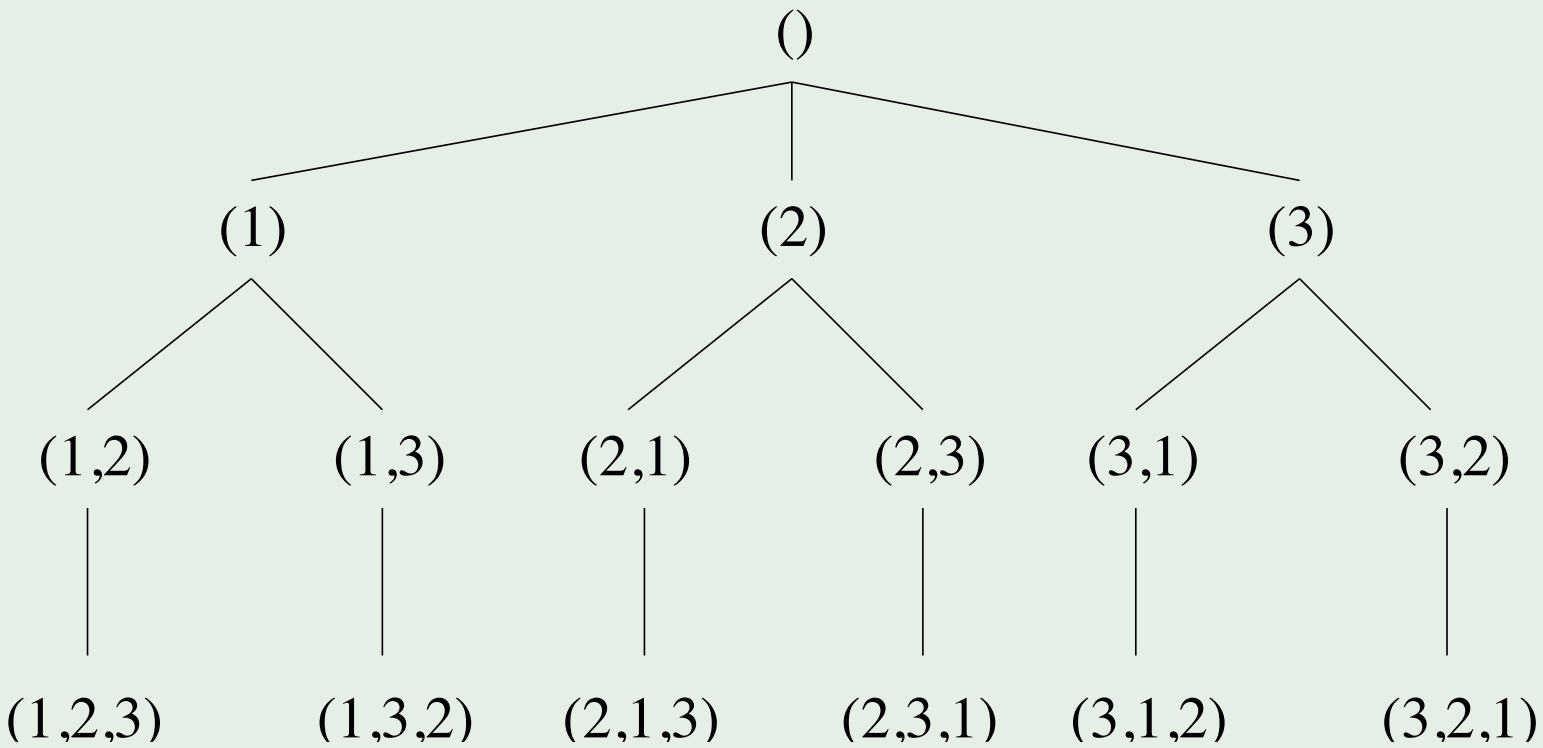
$$\prod_{k=1}^{n} k = n!.$$

Adaptem l'algorisme genèric amb

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

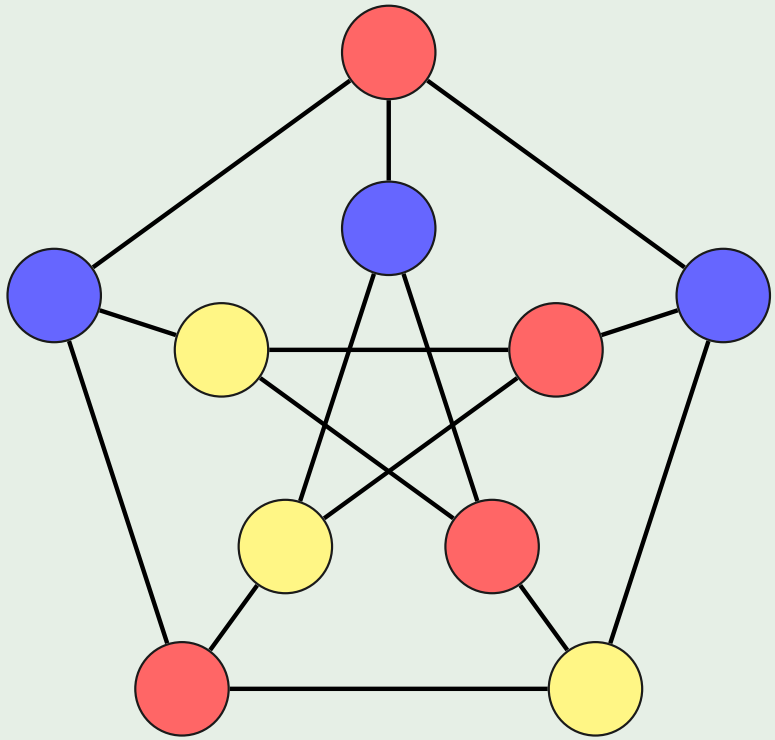
Algorisme genèric

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



Exemple: 3-Colorabilitat

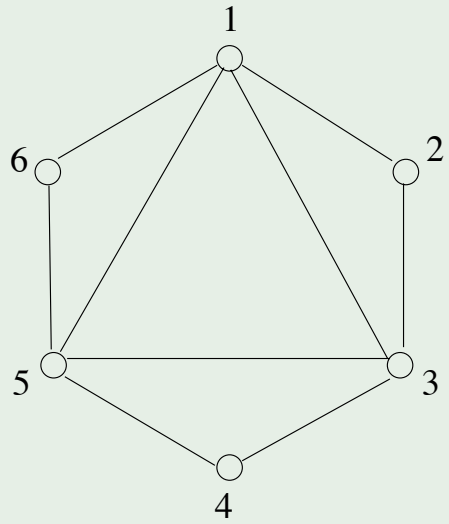
El problema de la 3-colorabilitat consisteix en
decidir si es pot assignar un color a cada vèrtex (d'un total de 3)
de manera que els adjacents tinguin colors diferents.



Una 3-coloració del graf de Petersen

3-colorabilitat

Donat un graf, per exemple



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

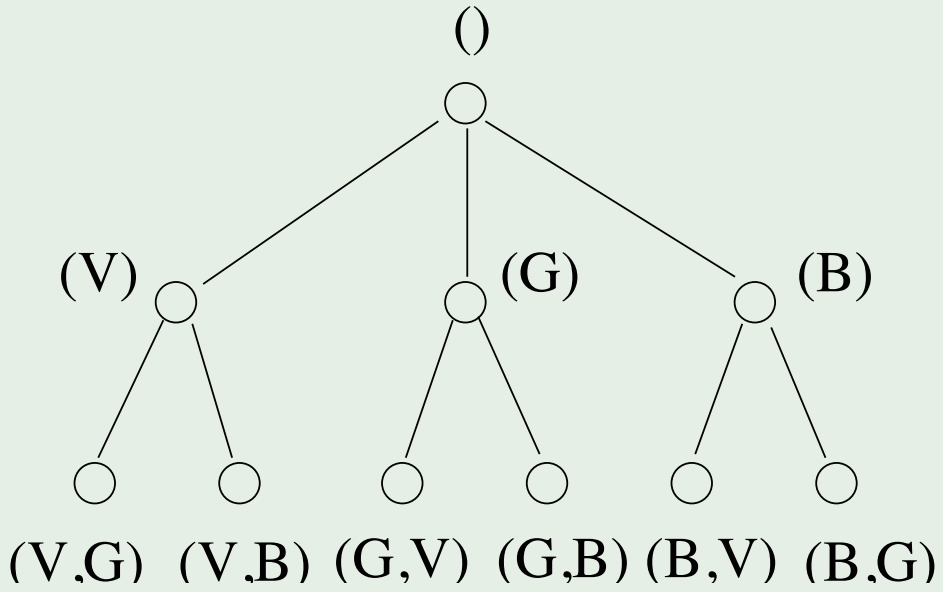
$$A = (a_1, a_2, \ldots, a_k)$$

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.

Algorisme genèric

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



Però si el que volem és trobar només una solució,

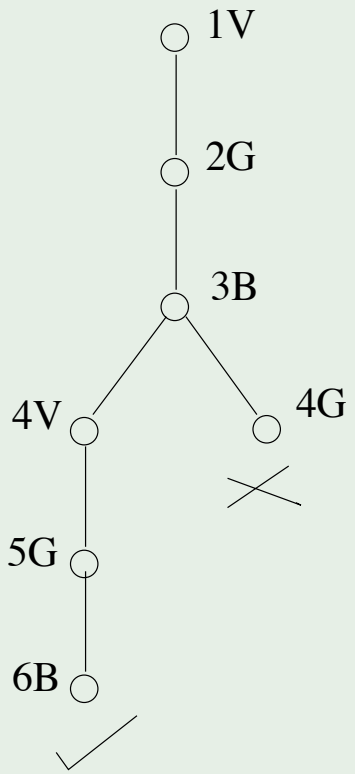
- es pot fixar un color per al vèrtex 1
- es pot fixar també un color per al vèrtex 2 sempre que sigui diferent
- qualsevol altra solució serà simètrica (ha d'assignar colors diferents)

Algorisme genèric

Fent la tria

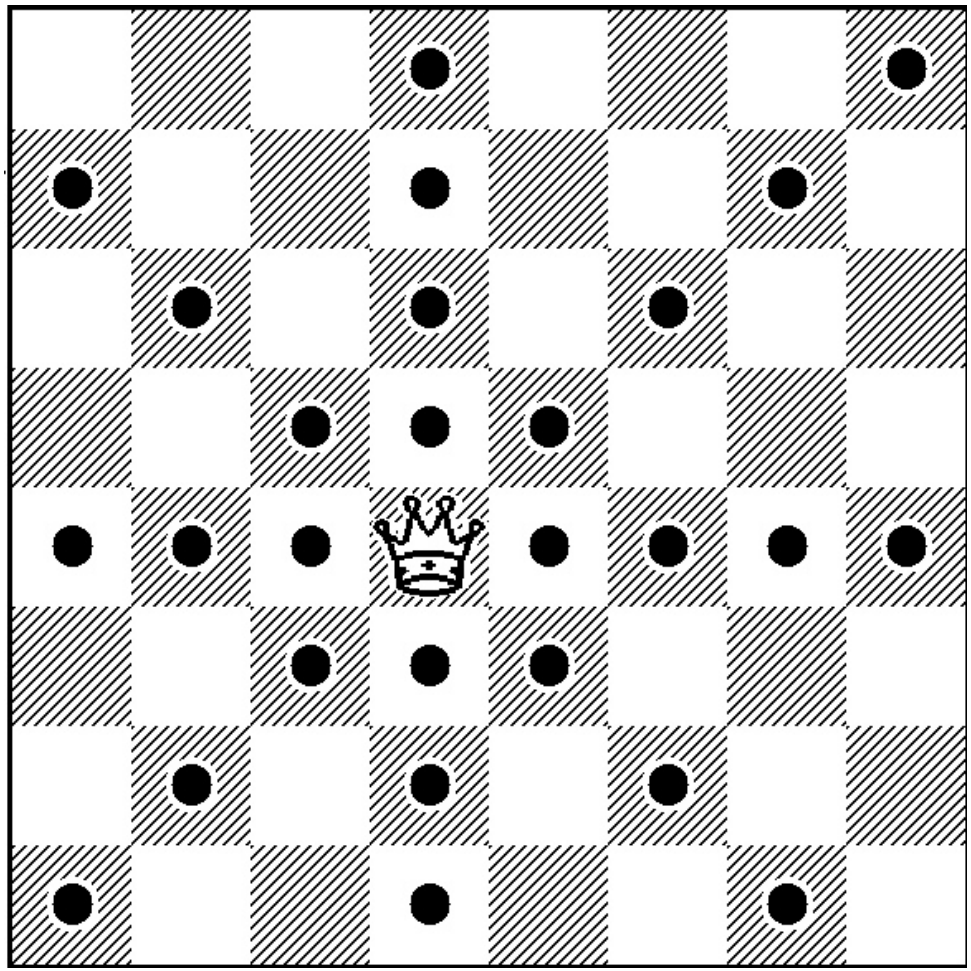
- $S_1 = \{V\}$
- $S_2 = \{G\}$

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



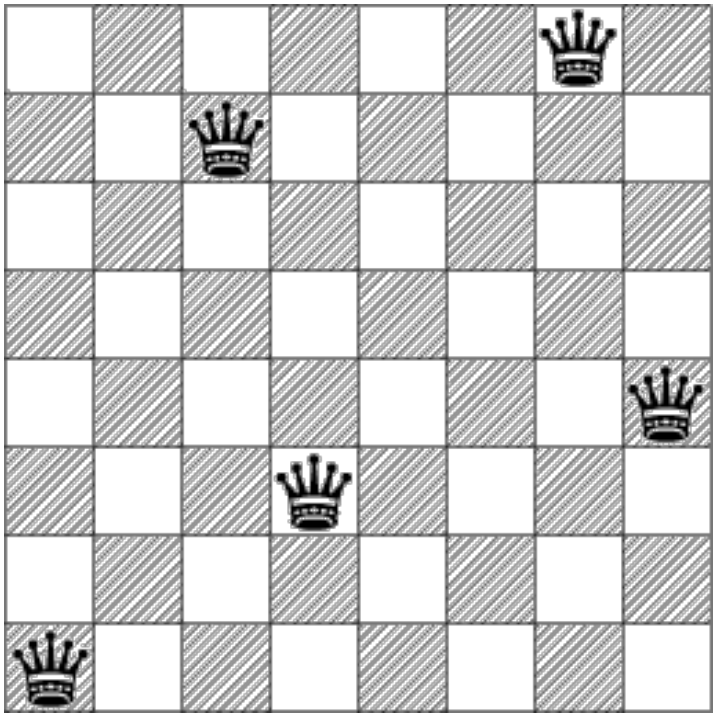
Les *n* reines

Moviments de la reina en el joc dels escacs:



Les *n* reines

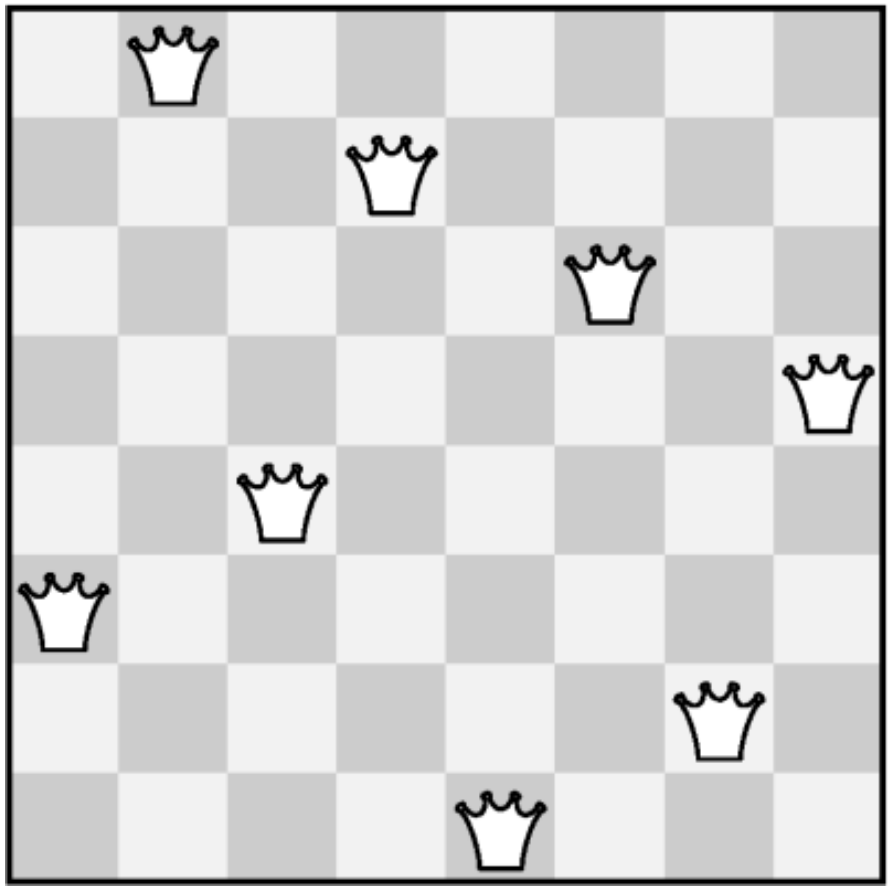
Quantes reines podem col·locar sobre un tauler sense que s'amenacin?
5? 6? 7? 8?



Les *n* reines

Problema de les 8 reines

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



Les n reines

Estratègies de resolució per força bruta:

1. Triar 8 posicions diferents del tauler.

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

2. Triar 8 posicions en files diferents.

$$8^8 = 16.777.216 \text{ configuracions}$$

3. Triar 8 posicions en files i columnes diferents.

$$8! = 40.320 \text{ configuracions}$$

Amb estratègies de backtracking més sofisticades encara es pot millorar més.

Les n reines

Considerarem el problema generalitzat de les n reines.

Problema de les n reines

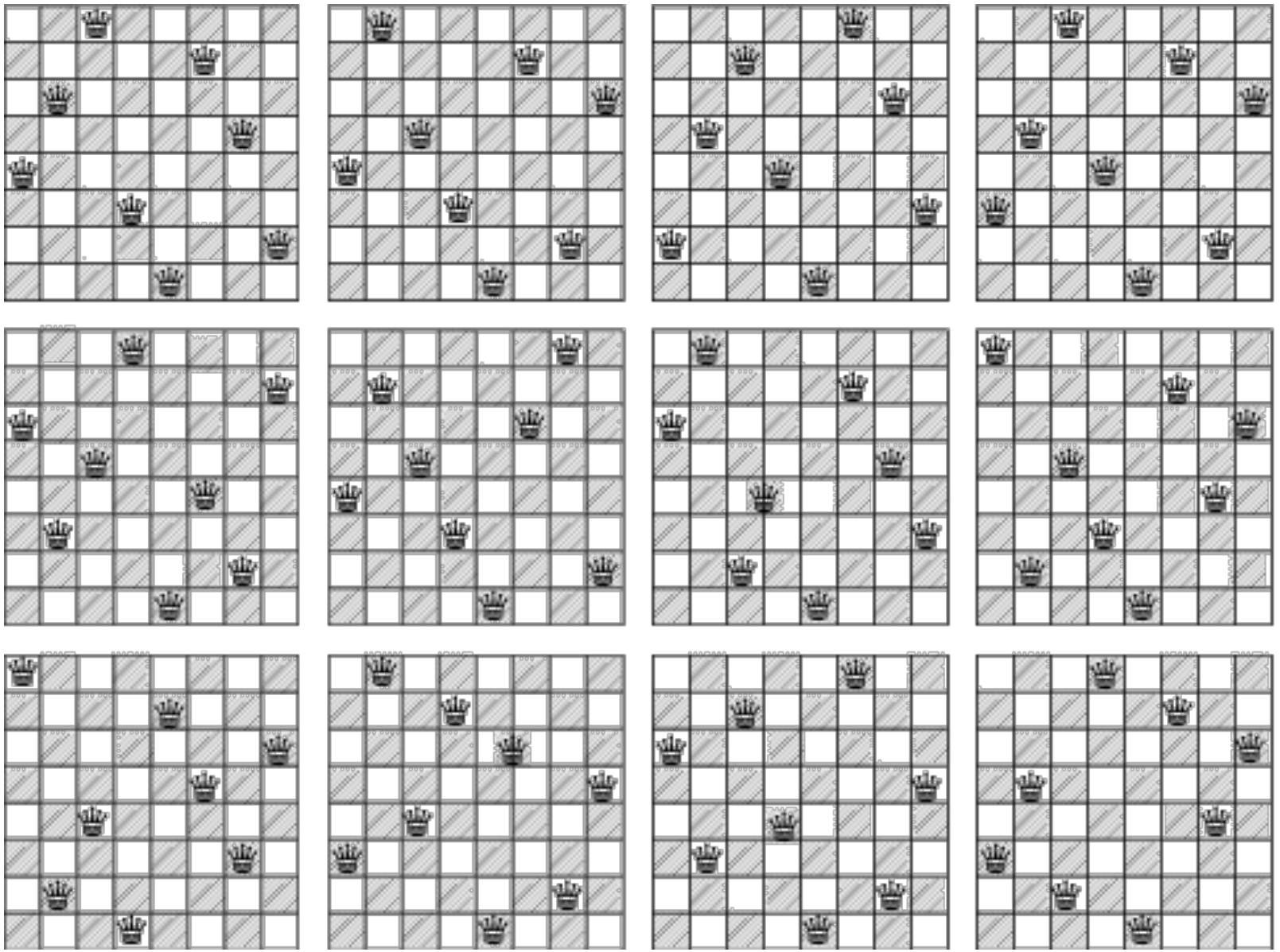
Col·locar n reines en un tauler $n \times n$ sense que cap n'amenaci cap altra.

Les n reines

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

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

Les 12 solucions no isomorfes per a $n = 8$



Les n reines

Primera implementació:

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

Les *n* reines

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 $t[i]$.

Les *n* reines

```
#include <iostream>
#include <vector>

using namespace std;

int n;           // Mida del tauler
vector<int> t;   // Configuracio de les reines

// Completa de totes les formes possibles t,
// despres d'haver assignat ja i reines
void reines(int i);

int main() {
  cin >> n;
  t = vector<int>(n);
  reines(0);
}
```

Les *n* reines

```
void escriu() {
  for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j)
      cout << (t[i] == j ? "Q" : ".");
    cout << endl;
  }
  cout << endl;
}
```

Les n reines

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

- columna, comprovem si $t[i] = t[k]$
- diagonal descendent ($\searrow$), comprovem si $t[i] - i = t[k] - k$
- diagonal ascendent ($\nearrow$), comprovem si $t[i] + i = t[k] + k$

Les *n* reines

```
bool legal(int i) {
  for (int k = 0; k < i; ++k)
    if (t[k]     == t[i]     or
        t[k] - k == t[i] - i or
        t[k] + k == t[i] + i)
      return false;
  return true;
}

void reines(int i) {
  if (i == n) escriu();
  else
    for (int j = 0; j < n; ++j){
      t[i] = j;
      if (legal(i))
        reines(i+1);
    }
}
```

Les n reines

Segona implementació:

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

Les *n* reines

```
#include <iostream>
#include <vector>

using namespace std;

int n;
vector<int> t;
// mc[j]  si ja hi ha reina a la columna j,
// md1[k] si ja hi ha reina a la diagonal i+j = k, etc.
vector<int> mc, md1, md2;

void reines(int i);

int main() {
  cin >> n;
  t = vector<int>(n);
  mc = vector<int>(n, false);
  md1 = md2 = vector<int>(2*n-1, false);
  reines(0);
}
```

Les *n* reines

```
int diag1(int i, int j) { return i+j; }
int diag2(int i, int j) { return i-j + n-1; }

void reines(int i) {
  if (i == n) escriu();
  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] = md1[diag1(i, j)] = md2[diag2(i, j)] = true;
        reines(i+1);
        mc[j] = md1[diag1(i, j)] = md2[diag2(i, j)] = false;
      }
}
```

Les *n* reines

Si només volem una solució, podem parar quan trobem la primera:

```
 // Diu si hi ha una solucio completant la solucio parcial
bool reines(int i) {
  if (i == n) {
    escriu();
    return true;
  }
  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] = md1[diag1(i, j)] = md2[diag2(i, j)] = true;
        if (reines(i+1)) return true;
        mc[j] = md1[diag1(i, j)] = md2[diag2(i, j)] = false;
      }
    return false;
  }
}
```

Les *n* reines

Si volem comptar solucions:

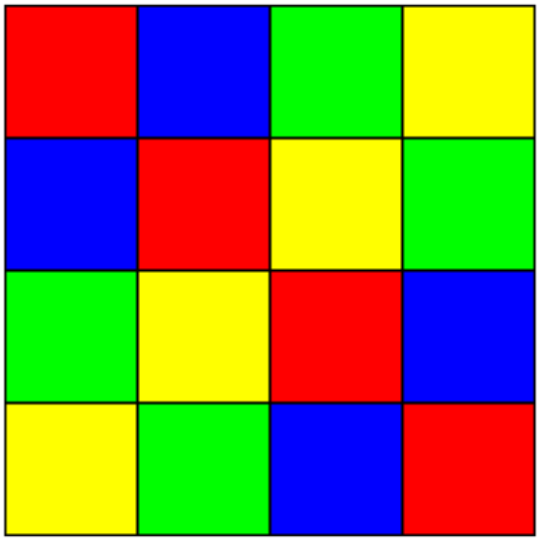
```
 // Diu quantes solucions hi ha completant la solucio parcial
int reines(int i) {
  if (i == n) {
    return 1;
  }
  else {
    int res = 0;
    for (int j = 0; j < n; ++j)
      if (not mc[j] and
          not md1[diag1(i, j)] and
          not md2[diag2(i, j)]) {
        t[i] = j;
        mc[j] = md1[diag1(i, j)] = md2[diag2(i, j)] = true;
        res += reines(i+1);
        mc[j] = md1[diag1(i, j)] = md2[diag2(i, j)] = false;
      }
    return res;
  }
}
```

Quadrats llatins

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	3
2	3	1
3	1	2

A	B
B	A



Quadrats llatins

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

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

Quadrats llatins

Problema dels quadrats llatins

Donat un n, trobar tots els quadrats llatins d'ordre n.

Quadrats llatins

Solució per tornada enrere amb marcatges.
Cost: $\mathcal{O}(n^{n^2})$.

```
#include <iostream>
#include <vector>

using namespace std;

int n;

// q[i][j] == valor a la fila i, columna j
vector<vector<int>> q;

// f[i][v] si la fila i ja usa el valor v
vector<vector<bool>> f;

// c[j][v] si la columna j ja usa el valor v
vector<vector<bool>> c;
```

Quadrats llatins

```
void escriu() {
  for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
      cout << q[i][j] << '\t';
    }
    cout << endl;
  }
  cout << endl;
}
```

Quadrats llatins

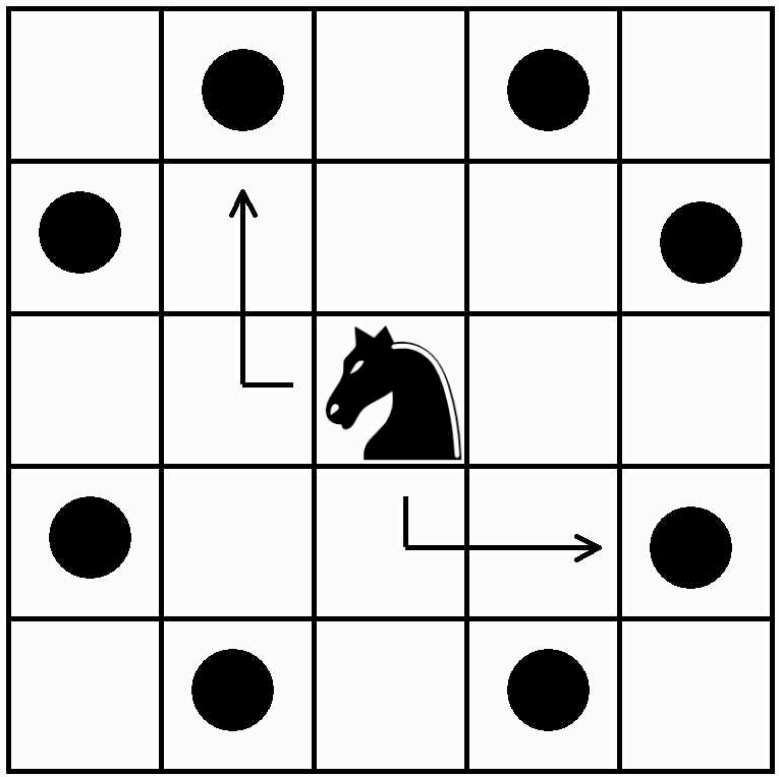
```
// Troba tots els quadrats llatins completant des de (i, j)
void quadrats_llatins(int i, int j) {
  if (i == n) return escriu();
  if (j == n) return quadrats_llatins(i+1, 0);
  for (int v = 0; v < n; ++v) {
    if (not f[i][v] and not c[j][v]) {
      f[i][v] = c[j][v] = true;
      q[i][j] = v;
      quadrats_llatins(i, j+1);
      f[i][v] = c[j][v] = false;
    }
  }
}

int main () {
  cin >> n;
  q = vector<vector<int>>(n, vector<int>(n));
  f = c = vector<vector<bool>>(n, vector<bool>(n, false));
  quadrats_llatins(0, 0);
}
```

Els salts de cavall

Salts de 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.



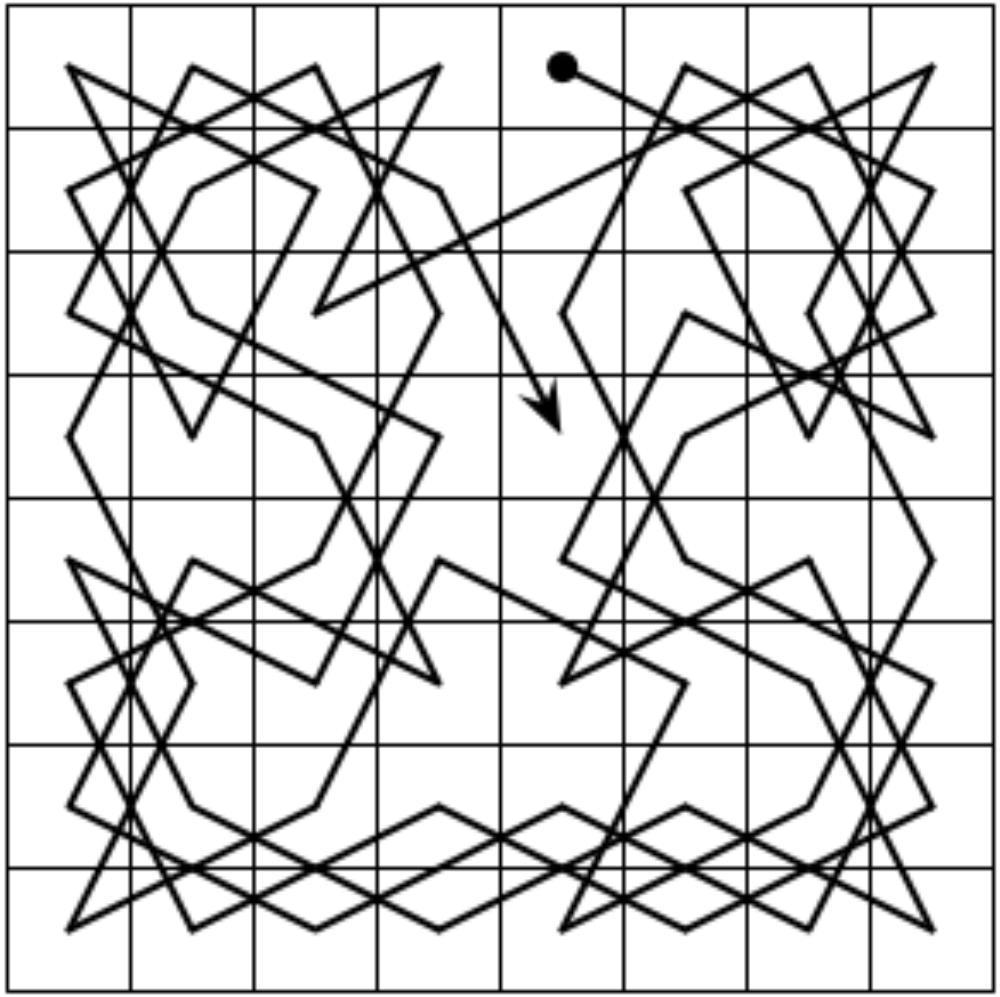
Els moviments del cavall

Els salts de cavall

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

- Es considera en versió
  - tancada: l'inici i final estan 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
  - 13.267.364.410.532 tours tancats no dirigits en un tauler 8×8
- En alguns casos es poden trobar tours en temps polinòmic amb l'estratègia de dividir i vèncer.

Els salts de cavall



Una solució oberta per al tauler d'escacs

Els salts de cavall

Problema dels salts de cavall

Donat un tauler $n \times n$ i una casella (i, j),
volem trobar un tour obert que comenci en (i, j).

Solució per tornada enrere.

Els salts de cavall

```
#include <iostream>
#include <vector>

using namespace std;

int n;

// t[i][j] == k quan en el salt k-esim arribem a (i,j)
// -1 si encara no hi hem arribat
vector<vector<int>> t;

// Si podem omplir el tauler des de (i, j) havent fet s salts
bool es_pot(int i, int j, int s);

int main() {
  int i, j;
  cin >> n >> i >> j;
  t = vector<vector<int>>(n, vector<int>(n, -1));
  cout << es_pot(i, j, 0) << endl;
}
```

Els salts de cavall

```
bool es_pot(int i, int j, int s) {
  if (i >= 0 and i < n and j >= 0 and j < n and t[i][j] == -1){
    t[i][j] = s;
    if (s == n*n-1 or
        es_pot(i+2, j-1, s+1) or es_pot(i+2, j+1, s+1) or
        es_pot(i+1, j+2, s+1) or es_pot(i-1, j+2, s+1) or
        es_pot(i-2, j+1, s+1) or es_pot(i-2, j-1, s+1) or
        es_pot(i-1, j-2, s+1) or es_pot(i+1, j-2, s+1))
      return true;
    t[i][j] = -1;
  }
  return false;
}
```

Els salts de cavall

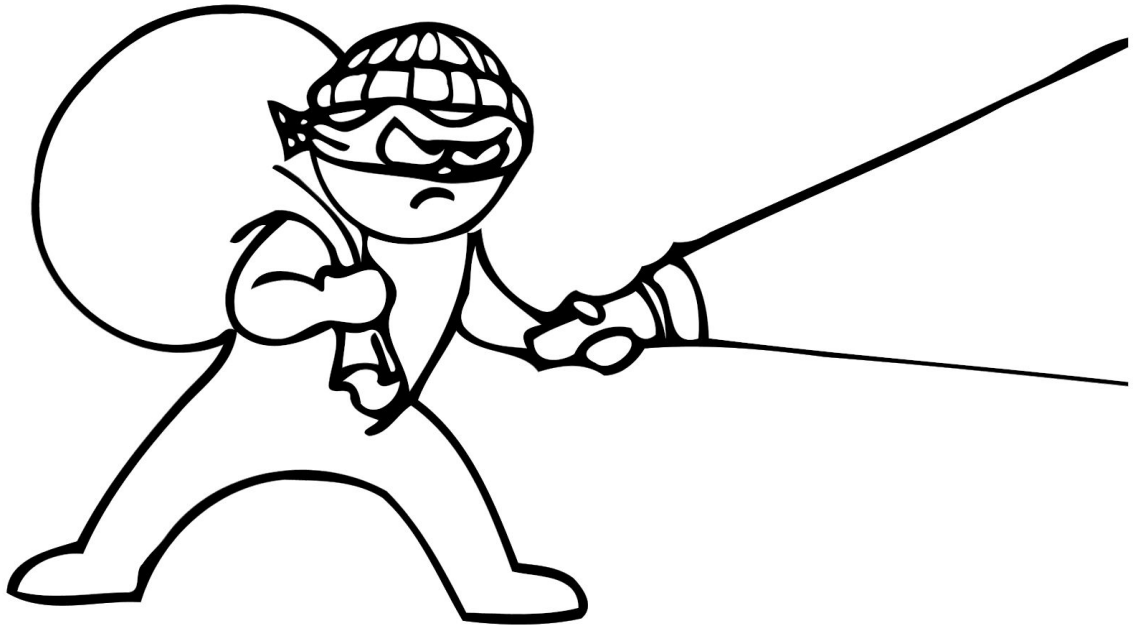
```
// Implementacio alternativa

vector<int> di = {1,  1, -1,  -1,  2,  2, -2,  -2};
vector<int> dj = {2, -2,  2,  -2,  1, -1,  1,  -1};

bool es_pot(int i, int j, int s) {
  if (i >= 0 and i < n and j >= 0 and j < n and t[i][j] == -1){
    t[i][j] = s;
    if (s == n*n-1) return true;
    for (int k = 0; k < 8; ++k)
      if (es_pot(i + di[k], j + dj[k], s+1))
        return true;
    t[i][j] = -1;
  }
  return false;
}
```

La motxilla

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?

La motxilla

En primer lloc cal:

- fer una llista amb els pesos i els valors dels objectes
- estimar fins a quin pes pot carregar en total com a màxim.

Ara només necessita un algorisme per actuar de pressa.

La motxilla

Problema de la motxilla

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

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

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

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

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

La motxilla

Primera solució: podem quan superem la capacitat

```
#include <iostream>
#include <vector>

using namespace std;

int c;            // Capacitat
int n;            // Nombre d'objectes
vector<int> p;    // Pesos
vector<int> v;    // Valors
vector<int> s;    // Solucio

int bv = -1;      // Millor valor   fins ara
vector<int> bs; // Millor solucio fins ara
```

La motxilla

```
void opt(int k, int spp, int svp) {
  if (spp > c) return; // Capacitat excedida: no continuem
  if (k == n) {
    if (svp > bv) { // Millorem la solucio que teniem fins ara
      bs = s;
      bv = svp;
    }
    return;
  }
  s[k] = 0; opt(k+1, spp,         svp        ); // Deixem obj. k
  s[k] = 1; opt(k+1, spp + p[k], svp + v[k]); // Agafem obj. k
}

int main() {
  cin >> c >> n;
  p = v = s = vector<int>(n);
  for (int& x : p) cin >> x;
  for (int& x : v) cin >> x;
  opt(0, 0, 0);
  cout << bv << endl; }
```

La motxilla

Segona solució: podem quan superem la capacitat,
i quan ja no podem superar el millor cost trobat fins ara (branch & bound)

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int c;            // Capacitat
int n;            // Nombre d'objectes
vector<int> p;    // Pesos
vector<int> v;    // Valors
vector<int> s;    // Solucio

int bv = -1;      // Millor valor   fins ara
vector<int> bs; // Millor solucio fins ara
```

La motxilla

```
void opt(int k, int spp, int svp, int svr) {
  if (spp > c  or  svp + svr <= bv) return;
  if (k == n) {
    bs = s;
    bv = svp;
    return;
  }
  s[k] = 0; opt(k+1, spp,          svp,          svr - v[k]);
  s[k] = 1; opt(k+1, spp + p[k], svp + v[k], svr - v[k]);
}

int main() {
  cin >> c >> n;
  p = v = s = vector<int>(n);
  for (int& x : p) cin >> x;
  for (int& x : v) cin >> x;
  opt(0, 0, 0, accumulate(v.begin(), v.end(), 0));
  cout << bv << endl;
}
```

El viatjant de comerç

El problema del viatjant de comerç (travelling salesman) consisteix en, donada una xarxa de ciutats, trobar l'ordre en què visitar-les de forma que:

- es comença i s'acaba a la ciutat del viatjant
- es passa per la resta de ciutats exactament un cop, i
- la distància total recorreguda és la més curta possible

Ruta òptima d'un viatjant passant per les 15 ciutats més grans d'Alemanya.

Font: `upload.wikimedia.org/wikipedia/commons/c/c4/TSP_Deutschland_3.png`

El viatjant de comerç

- És un dels problemes d'optimització combinatòria més estudiats
- És important en informàtica teòrica
- Té aplicacions pràctiques en
  - planificació
  - logística
  - fabricació de microchips
  - seqüenciació d'ADN
  - astronomia
  - ...

El viatjant de comerç

```
class Viatjant {

  int n;                  // nombre de ciutats
  matriu<int> M;          // la matriu de distancies
  vector<int> s;          // seguent de cada vertex
                          // (-1 si encara no utilitzat)
  vector<int> sol;        // millor solucio fins ara
  double millor;          // cost de la millor solucio fins ara
```

El viatjant de comerç

```
void recursiu (int v, int t, double c) {
    // v = darrer vertex del cami
    // t = talla del cami
    // c = cost fins ara
    if (t == n) {
        c += M[v][0];
        if (c < millor) {
            millor = c;
            sol = s;
            sol[v] = 0;
        }
    } else {
        for(int u = 0; u < n; ++u)
          if (u != v and s[u] == -1) {
            if (c + M[v][u] < millor) {
                s[v] = u;
                recursiu(u, t+1, c+M[v][u]);
                s[v] = -1;
}   }   }   }
```

El viatjant de comerç

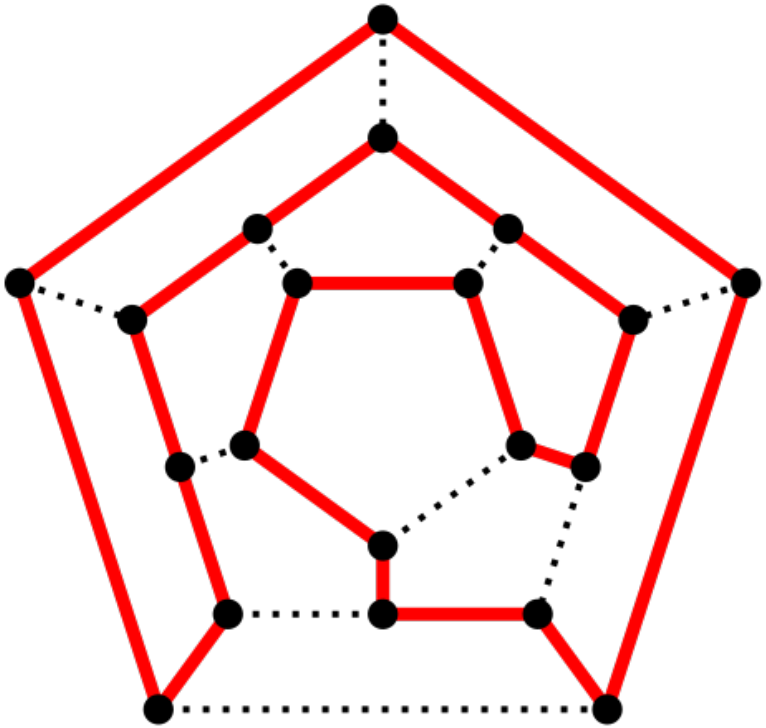
```
public:

    Viatjant (matriu M) {
        this->M = M;
        n = M.rows();
        s = vector<int>(n, -1);
        sol = vector<int>(n);
        millor = infinit;
        recursiu(0, 1, 0);
    }

    vector<int> solucio (     ) { return sol;     }
    int         seguent (int x) { return sol[x]; }
    double         cost (     ) { return millor; }
};
```

Graf Hamiltonià

- Un cicle Hamiltonià és un cicle que visita cada vèrtex exactament un cop



Font: `https://en.wikipedia.org/wiki/Hamiltonian_path`

- Si un graf té un cicle Hamiltonià, llavors diem que el graf és Hamiltonià.
- Donat un graf, volem saber si és Hamiltonià.

Graf Hamiltonià

- Suposem que el graf és connex
- Suposem que el graf està representat amb llistes d'adjacència

```
typedef vector< vector<int> > Graf;
typedef list<int>::iterator iter;


class GrafHamiltonia {

    Graf G;                 // el graf
    int n;                  // nombre de vertexs
    bool trobat;            // indica si ja s'ha trobat un cicle
    vector<int> s;          // seguent de cada vertex
                            // (-1 si encara no utilitzat)
    vector<int> S;          // solucio (si trobat)
```

Graf Hamiltonià

```
void recursiu (int v, int t) {
    // v = darrer vertex del cami, t = talla del cami
    if (t == n) {
        // cal assegurar-nos que el cicle es pugui tancar
        if (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;
}   }   }   }
```

Graf Hamiltonià

```
public:

    GrafHamiltonia (Graf 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;
    }
};
```

La reconstrucció Turnpike

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 $i > j$ (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.

La reconstrucció Turnpike

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 $i > j$ (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.

La reconstrucció Turnpike

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 $i > j$ (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.

La reconstrucció Turnpike

Els problemes de reconstrucció solen 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
- es pot aplicar un algorisme de backtracking que normalment funciona en temps $\mathcal{O}(n^2 \log n)$ però que, en el cas pitjor, és exponencial

La reconstrucció Turnpike

Els problemes de reconstrucció solen 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
- es pot aplicar un algorisme de backtracking que normalment funciona en temps $\mathcal{O}(n^2 \log n)$ però que, en el cas pitjor, és exponencial

La reconstrucció Turnpike

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 $x_1 = 0$.
- Com que $x_6 - x_1 = \max(D) = 10$, tenim $x_6 = 10$
- Eliminem 10 de D. Ara,

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

La reconstrucció Turnpike

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

- La distància més gran que queda és 8.

  Per tant, o bé $8 = x_6 - x_2$ o bé $8 = x_5 - x_1$. De forma que

  $$x_2 = 2 \text{ o } x_5 = 8.$$

  Si tenen solució, seran simètriques.

  Sense pèrdua de generalitat, triem $x_5 = 8$.

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

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

La reconstrucció Turnpike

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

- Com que $\max(D) = 7$, o bé $7 = x_6 - x_2$ o bé $7 = x_4 - x_1$.
  Per tant, $x_2 = 3$ o $x_4 = 7$.
  - Si $x_4 = 7$, les distàncies
    $$x_6 - 7 = 3 \text{ i } x_5 - 7 = 1$$
    han de ser a D, però veiem que hi són.
  - Si $x_2 = 3$, les distàncies
    $$3 - x_1 = 3 \text{ i } x_5 - 3 = 5$$
    han de ser a D, però 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,
  $$D = \{2, 2, 3, 3, 4, 5, 5, 5, 6\}$$

La reconstrucció Turnpike

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

- Com que $\max(D) = 6$, o bé $6 = x_3 - x_1$ o bé $6 = x_6 - x_2$.

  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

$$x_2 - x_1 = 4 \text{ i } x_5 - x_2 = 4$$

    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,

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

La reconstrucció Turnpike

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

- Com que $\max(D) = 6$, o bé $6 = x_4 - x_1$ o bé $6 = x_6 - x_3$.

  Per tant, $x_4 = 6$ 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.

La reconstrucció Turnpike

- Aquest mètode dóna lloc a un algorisme que, si no es produeix cap tornada enrere, té cost $\mathcal{O}(n^2 \log n)$.
- 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.