Résolution de problème & back-tracking

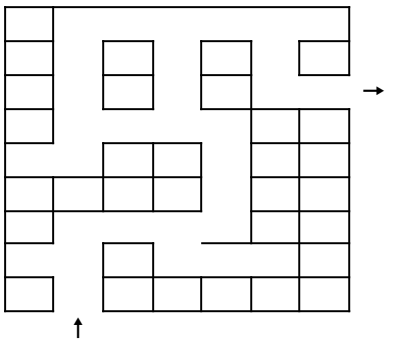
Bruno Mermet

Décembre 2021

Résoudre un problème

- Différents types de problème
  - 1 ou plusieurs solutions
  - Solution = situation ou chemin
  - Des solutions plus ou moins bonnes
- Problème initial = représenter une situation
- Démarche possible
  - Générer les différentes situations possibles

    => Pb de la finitude/du cardinal de l'ensemble des solutions
  - Vérifier la validité de chacune



Exemples

- Cache-cache Pirates
- Problèmes des gratte-ciel
- Le compte est bon
- Sudoku
- L’Arche de Noé
- Columbus’ Egg
- Le problème des *n* reines
- …

Smart Games
https://www.smartgames.eu/fr

Think Fun
https://www.thinkfun.fr/

Fun Games
https://www.fungamesnet.fr/logiquest

Comment générer les solutions potentielles

- Première version :

  Générer complètement chaque solution, et vérifier si elle est valide ou non

- Deuxième version :

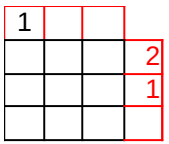
  Générer un début de solution

  - s'il est valide, générer la suite
  - Sinon, passer au début suivant (et ignorer toutes les solutions de même préfixe)
  - Si rien n'a été trouvé, revenir sur la décision de niveau antérieur

  => Approche récursive

Exemple : gratte-ciel (version 1)

- Énoncé



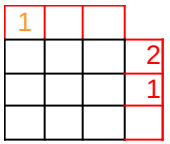
- Génération naïve

  3^9 = 19 683 possibilités → 177 147 valeurs (1 possibilité = 9 valeurs)

- Génération pré-contrainte (chaque ligne = permutation de [1, 2, 3]) :

  $(3!)^3 = 6^3$ = 216 possibilités → 1944 valeurs

Exemple : gratte-ciel (version 2*)



37 * 3 = 111 valeurs générées

[1, 2, 3] [1, 3, 2] [2, 1, 3] [2, 3, 1] [3, 1, 2] [3, 2, 1]

[1, 2, 3] [1, 3, 2] [2, 1, 3] [2, 3, 1] [3, 1, 2] [3, 2, 1]

[1, 2, 3] [1, 3, 2] [2, 1, 3] [2, 3, 1] [3, 1, 2] [3, 2, 1]

[1, 2, 3] [1, 3, 2] [2, 1, 3] [2, 3, 1] [3, 1, 2] [3, 2, 1]

[1, 2, 3] [1, 3, 2] [2, 1, 3] [2, 3, 1] [3, 1, 2] [3, 2, 1]

[2, 3, 1]

Complexité ?

Complexité : Cache-cache Pirates

- 4 ! = 24 placements des pièces
- Pour chaque placement
  - 4 x 4 x 4 x 2 = 128 orientations possibles

  => 3 072 configurations à examiner

Complexité : gratte-ciel

- $\frac{n^2!}{n \cdot n!}$ Possibilités
  - Pour un carré de côté 2 : 6 possibilités
  - Pour un carré de côté 4 : 217 945 728 000 configurations
  - Pour un carré de côté 6 : 86 109 566 386 551 207 747 222 094 479 360 000 000

Complexité : Columbus’ Egg

- $\frac{16!}{4 \cdot 4!}$ Possibilités = 217 945 728 000

Complexité : Sudoku

- $\frac{81!}{9 \cdot 9!}$ Possibilités = 1 775 036 137 060 114 144 216 555 895 912 364 389 027 397 255 515 023 625 424 341 395 293 329 346 614 090 993 121 749 624 684 544 000 000 000 000 000 000

Complexité : le Compte est bon

Comment procéder ?

Enlever 2 plaques, leur appliquer 1 opération et ajouter le résultat comme nouvelle plaque

$$(C_6^2 \cdot 4) \cdot (C_5^2 \cdot 4) \cdot (C_4^2 \cdot 4) \cdot (C_3^2 \cdot 4) \cdot (C_2^2 \cdot 4) = \dots$$

$$4^5 \cdot \sum_{i=2}^{6} C_n^2 = \dots$$

$$1024 \cdot (15 \cdot 10 \cdot 6 \cdot 3 \cdot 1) = \dots$$

$$1024 \cdot 2700 = \dots$$

$$2\,764\,800$$

Génération de solutions potentielles

- Utiliser la récursivité
  - On peut voir les problèmes comme des problèmes récursifs
  - Au niveau i :
    - on génère successivement toutes les possibilités à partir de la configuration reçue
    - Pour chaque possibilité, on enrichie la configuration reçue avec le choix fait, puis on passe au niveau i+1
    - Si la configuration est complète, on vérifie sa validité pour renvoyer Vraie (c'est une solution) ou Faux (ce n'est pas une solution)
    - Si aucune possibilité au niveau i n'a donné de solution, on revient sur le choix de niveau i-1
  - Attention aux références : l'appel au niveau i+1 ne doit pas modifier les variables du niveau i => principe de base du back-tracking

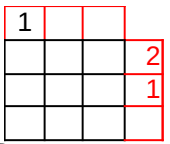
Vision récursive des problèmes
1. Les gratte-ciel (a)

- Version récursive 1 :
  - On numérote les cases de 1 à n^2
  - Pour chaque case, on envisage les valeurs de 1 à n
  - Quand on a terminé (case n^2 remplie), on vérifie la validité
- Version récursive 2 :
  - On numérote les lignes de 1 à n
  - Pour chaque ligne, on envisage les n! permutations des nombres de 1 à n
  - Quand on a terminé (ligne n remplie), on vérifie la validité
- Version récursive 3 :
  - Au niveau de récursivité i, on génère une valeur pour la i-ème case (sur les n^2)
  - On élimine tout de suite les valeurs qui ne conviennent pas

Vision récursive des problèmes
1. Les gratte-ciel (b)

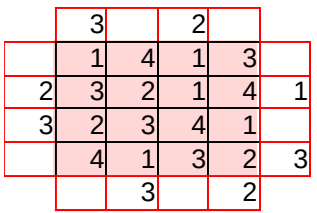
- Analyse empirique de la complexité
  - Problème 1 :

    temps = 0.0035s



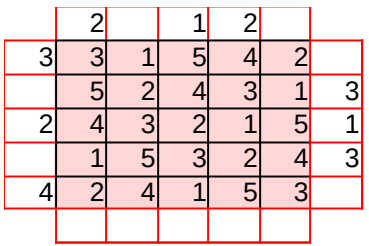
  - Problème 2 :

    temps = 1,7s



  - Problème 3 :

    temps =



Vision récursive des problèmes
2. Le compte est bon

- Résoudre le compte C avec n plaques revient à résoudre le compte C après avoir effectué une des 4 opérations sur les n plaques de départ
- On envisage toutes les paires (non ordonnées) de plaques parmi les n disponibles
  - Pour chaque paire, on envisage les 4 opérations
  - Pour chaque opération possible, on remplace les plaques ayant servi par le résultat obtenu (en tête) et on recommence
  - À tout instant, si la valeur en tête est égale au compte à obtenir, on a une solution valide
- Attention ! Ici, la solution, c’est la trace de ce qui a été fait !

Vision récursive des problèmes
3. Cache-cache Pirates

- Pour la i-ème case, choisir une plaque parmi les plaques restantes
- Envisager toutes ses positions (2 ou 4)
- Si on a placé la 4è plaque, évaluer la solution trouvée

Vision récursive des problèmes
4. Columbus' Egg

- Pour la i-ème case :
  - Placer un des œufs restant disponible
  - Passer à la case suivante
  - Si la 16è case est atteinte, évaluer la solution

Vision récursive des problèmes
5. Sudoku

- Procéder par case ou par ligne, comme pour le problème des gratte-ciel

Vision récursive des problèmes
6. Rush Hour

- Pour tous les mouvements possibles depuis la situation courante :
  - Calculer la position suivante et recommencer
- Problèmes :
  - suite de mouvements infinie (notamment avec les avant/arrière)
  - Pas de profondeur maximale a priori

Optimiser les recherches

- Principe général :

  À chaque nouveau bout de la solution généré, vérifier si ce bout de solution est valide

1. Les gratte-ciel optimisé
a. Principe

- Lorsqu'une ligne est générée, on peut vérifier si elle respecte les contraintes gauches et droits
- Lorsqu'une ligne est générée, si on respecte la règle de base sur les colonnes (pas de doublon)
- Lorsqu'une ligne est générée, on peut vérifiée si la visibilité haute n'est pas dépassée
- Il reste à vérifier les validités des colonnes à la fin

  => Important : faire les vérifications dès que possible pour élaguer au maximum l'arbre (c.f. ici la vérification de la visibilité haute, par exemple)

1. Les gratte-ciel optimisé
b. Analyse

	Version de base	Version optimisée
Cas 1	0,0035s	0,0001s
Cas 2	1,1s	0,0015s
Cas 3	1j 5h	0,08s

1. Les gratte-ciel optimisé
c. Encore mieux

- Générer case par case

  On rejette plus de solutions d’un coup

- Générer case par case en place

  Comme à l’itération n, on ne revient pas sur ce qui a été généré à l’itération n-1, on peut éviter une copie

- Comparaison sur le problème 6b
  - Version optimisée par ligne :1582s (26m22s)
  - Version optimisée par case : 202s (3m22s)
  - Version optimisée par case en place : 84s (1m24s)

2a. Cache-cache Pirates : Optimiser ?

- 3 072 configuration « seulement »
- Génération et validation rapides

=> optimisation non nécessaire

vraiment ??

2b. Comparatif Cache-cache Pirates force brute vs optimisation

- Sur l’ensemble des problèmes du jeu (48)
  - Force brute : 0,57s
  - Optimisation : 0,005s

3. Le compte est bon optimisé

- Analyse
  - Additions et multiplications commutatives
  - Soustractions et divisions faisables que dans un sens (on reste dans $\mathbb{N}$)

    => N’envisager qu’un sens pour chaque couple (division par 2 du nombre de cas)
  - 0 ne sert à rien (et pourrait même engendrer des problèmes)

    => ignorer les soustractions générant 0 (seules opération pouvant générer un 0)
  - Éviter de produire une valeur consommée

  => solutions en 3s environ

4. Columbus’ Egg optimisé

- Ne pas générer des configurations équivalentes
- Tester les lignes et blocs de ligne au fur et à mesure
- Tester les colonnes et blocs de colonne au fur et à mesure

5. Problème des *n* reines

- Générer une reine pour chaque ligne
- Sauter les colonnes déjà utilisées
- Ne faire les vérifications que vers le nord-ouest et le nord-est

En guise de conclusion

- Principe général
  - Recherche de solution par parcours exhaustif
  - Mécanisme récursif, avec retour-arrière si la voie prise mène à un échec (analogie du labyrinthe)
  - Nécessité de détecter au plus vite les voies menant à un échec pour rester efficace
- Difficultés
  - Maîtriser la récursivité
  - Faire attention aux potentielles modifications de l'état lors des appels récursifs avec le partage de référence
  - Surveiller la profondeur max possible
  - Savoir évaluer la complexité
- Et sinon… vous avez pensé à Prolog ?