Exercice de programmation - Boggle

Posté par Nicolas Zermati dans les catégories actualité

21/08
2012

Dans cet article, je vous propose de réaliser générateur de grilles et de solutions pour le jeu Boggle. C'est un problème que j'ai du résoudre il y a quelques années pour le jeu BoggleDroid. Aujourd'hui je souhaite partager cette expérience avec vous.

L'intérêt n'est pas le jeu en lui même mais c'est d'examiner les algorithmes et les structures de données à mettre en place pour le résoudre. Ce sera l'occasion de (re)découvrir ce que sont les Tries et de les voir à l'action. Vous assisterez au refactoring d'un algorithme de force brute vers un algorithme plus subtil. Et, je l'espère, vous aurez la satisfaction de voir un problème être résolu de plus en plus rapidement d'un paragraphe à l'autre.

Générer une grille

Une grille de Boggle est carrée et comporte 16 cellules. Dans chaque cellule se trouve une lettre de l'alphabet.

Les lettres sont choisies en fonction de leur fréquence dans une langue donnée. Une fonction : `Boogle::Language.letter(lang_sym)` permet d'obtenir une lettre au hasard dans un alphabet donné. Les langues disponibles sont : le français, l'anglais, l'allemand, l'espagnol et l'italien. On représente chaque langue par un symbole Ruby : `:fr`, `:en`, `:de`, `:es` et `:it`. Pour obtenir une lettre aléatoire dans l'alphabet français on écrira :

```
> Boogle::Language.letter(:fr)
=> "E"
```

Dès lors qu'on dispose de cette fonction la génération de la grille est triviale.

```
module Boggle
  class Grid
    def initialize(lang)
      @matrix = (0...4).map{(0...4).map{Language.letter(lang)}}
    end
  end
end
```

Résoudre une grille

Maintenant que l'on peut générer une grille il faut être capable d'en donner la solution. La solution d'une grille se compose de tous les mots présents dans la grille. Voici la définition de Wikipédia pour les *mots présents dans la grille* :

> *[...] mots pouvant être formés à partir de lettres adjacentes du plateau. Par « adjacentes », il est sous-entendu horizontalement, verticalement, ou en diagonale. Les mots doivent être de 3 lettres au minimum, peuvent être au singulier ou au pluriel, conjugués ou non, mais ne doivent pas utiliser plusieurs fois le même dé pour le même mot.*

On décide qu'une chaine de lettres forme un mot lorque celle çi est présente dans un dictionnaire. Voici une première implémentation assez naive d'un dictionnaire :

```
module Boggle
  class Dict
    def initialize(filepath)
      @filepath = filepath
      @words = File.readlines(filepath).map(&:chomp).delete_if{|w| w.size < 3}
    end

    def exists?(word)
      @words.include?(word)
    end
  end
end
```

Le fichier d'entrée doit contenir un mot, en majuscule et non accentué, par ligne. Conformément à la règle du jeu on filtre les mots de moins de 3 lettres.

Vous trouverez sans mal des listes de mots sur Internet.

Bruteforce

Pour trouver les solutions des grilles, commencons par étudier un algorithme utilisant la force brute.

On explore tous les chemins possibles dans la grille dans `Boggle::Algos::BruteForce` :

```
1/ Pour chaque case C de la grille faire :
  2/ Créer un chemin P au départ de C
  3/ Pour chaque case adjacente A à la case courrante de P faire :
    4/ Simuler un déplacement sur A
    5/ Ajouter le mot formé à la solution s'il existe
    6/ Réinstancier l'algorithme à partir de l'étape 3
7/ Retourner la solution
```

Soit en Ruby (github) :

```ruby
module Boggle
  module Algos
    class BruteForce
      attr_reader :solutions

      def initialize(grid, dict)
        @grid      = grid
        @dict      = dict
        @solutions = Set.new
        @cursor    = []
        @word      = ""
      end

      def solve
        init_steps = (0...GRID_SIZE * GRID_SIZE).map do |n|
          [n / GRID_SIZE, n % GRID_SIZE]
        end
        explore!(init_steps)
      end

      private

      def explore!(steps)
        steps.each do |coord|
          move_to!(coord)
          @solutions << @word if @word.size >= 3 && @dict.exists?(@word)
          explore!(next_steps)
```

```
          move_back!
        end
      end

      def next_steps
        i, j   = @cursor.last
        coords = [[i-1, j], [i, j-1], [i-1, j-1], [i+1, j],
                  [i, j+1], [i+1, j+1], [i-1, j+1], [i+1, j-1]]
        coords.delete_if do |coord|
          i, j = coord
          i >= GRID_SIZE || i < 0 || j >= GRID_SIZE || j < 0 || @cursor.includ
        end
      end

      def move_to!(coord)
        @cursor << coord
        @word += @grid.matrix[coord[0]][coord[1]]
      end

      def move_back!
        @cursor.pop
        @word = @word[0..-2]
      end
    end
  end
end
```

L'algorhitme est simple, on fait exactement ce qu'on aurait fait avec ses yeux, un crayon ou son doigt.

Je lance un petit script de test :

```
require "./algos"

grid = Boggle::Grid.new(:fr)
dict = Boggle::Dict.new("~/ods6.txt")
algo = Boggle::Algos::BruteForce.new(grid, dict)

puts grid
algo.solve
puts "#{algo.solutions.size} solutions found"
```

Et là bien sûr pas de réponse avant un très très long moment. En utilisant une grille 3x3 et l'Officiel du Scrabble 6 (avec ~ 380 000 mots) ; il m'a fallu 270 secondes pour trouver les solutions de la grille. Je n'ai pas eu la patience de laisser terminer l'exécution en 4x4.

Tout cela n'a rien de surprenant puisqu'il y a une dizaine de millions de chemins possibles en 4x4. On a donc autant de requètes `Dict#exists?`,

`@cursor.include?(coord)`, etc.

Optimisation du dictionnaire

La recherche dans le dictionnaire est coûteuse. C'est une recherche linéaire. Une comparaison de la chaine recherchée et de chaque mot du dictionnaire est effectuée ; soit 380 000 comparaisons dans notre cas.

On a besoin d'une structure où les recherches sont le plus rapide possible. On va donc utiliser un Hash dont les clés seront les mots du dictionnaire :

```
module Boggle
  class Dict
    def initialize(filepath)
      @filepath = filepath
      @words = {}
      f = File.open(filepath, "r")
      f.each_line { |line| @words[line.chomp] = nil if line.size > 3 }
      f.close
    end

    def exists?(word)
      @words.has_key?(word)
    end
  end
end
```

Ce changement de structure de donnée change la donne : on passe à moins d'une seconde pour une grille `3x3`; c'est mieux ! Par contre la résolution d'une grille `4x4` dure ~ 430 secondes, c'est toujours long.

```
$ time ruby ./app.rb
UAUO
CUDH
IENU
LESD
119 solutions found
AUCUN
AUCUNE
AUCUNES
[...]
SUD
DUNDEE
DUNDEES
ruby ./app.rb  432.80s user 0.96s system 99% cpu 7:16.74 total
```

Recherche des cases adjacentes

Pour rechercher les cases adjacentes on utilise à nouveau une recherche linéaire : `@cursor.include?(coord)`. Lorsque les chemins formés sont courts cela n'est pas gênant. Mais, plus les chemins sont longs et plus cette recherche est coûteuse.

On va donc échanger de la mémoire contre du temps de calcul en créant un masque sur la grille. Ce dernier mémorise les cases appartenant au chemin permettant de savoir si des coordonnées sont sur le chemin en cours ou non.

On ajoute le masque `@used` à la classe `Boggle::Algos::BruteForce` dans le constructeur, dans la recherche des cases adjacentes et dans le déplacement du curseur. Voici les ajouts et changements (github) :

```
def initialize(grid, dict)
  ...
  @used = (0...GRID_SIZE).map{ (0...GRID_SIZE).map{ false } }
end

def next_steps
  ...
    i >= GRID_SIZE || i < 0 || j >= GRID_SIZE || j < 0 || @used[i][j]
  ...
end

def move_to!(coord)
  ...
  @used[coord[0]][coord[1]] = true
end

def move_back!
  i, j        = @cursor.pop
  @word       = @word[0..-2]
  @used[i][j] = false
end
```

Une fois encore le gain est notable : ~ 100 secondes pour résoudre une grille `4x4`.

Autres optimisations

On peut imaginer encore beaucoup d'optimisations de cet algorithme de bruteforce. Dans un premier temps on déplace le masque `@used`, le curseur `@cursor` et la notion de déplacement `move_to!` et `move_back!` dans `Boggle::Grid`. Ensuite on peut calculer une fois les voisins de chaque case et réutiliser ceux ci par la suite. En effet, jusqu'à maintenant on calculait ces coordonnées à chaque appel à `Boggle::Algos::BruteForce::next_steps`.

On créé une constante : `NEIGHBORS` qui contiendra les positions des voisins de chaque case (github). On utilisera ce masque dans `next_steps`. C'est encore le même échange : *mémoire contre temps de calcul*.

```
NEIGHBORS = (0...GRID_SIZE).map do |i|
  (0...GRID_SIZE).map do |j|
    coords = [[i-1, j], [i, j-1], [i-1, j-1], [i+1, j],
              [i, j+1], [i+1, j+1], [i-1, j+1], [i+1, j-1]]
    coords.delete_if do |coord|
      a, b = coord
      a >= GRID_SIZE || a < 0 || b >= GRID_SIZE || b < 0
    end
  end
end
```

```
def next_steps
  i, j = @cursor.last
  NEIGHBORS[i][j].reject { |coord| @used[coord[0]][coord[1]] }
end
```

Avec ce changement j'obtiens un temps de ~ 50 secondes pour résoudre les grilles.

Un algorithme plus subtil

Bien que 50 secondes soit une amélioration très importante, si l'on veut générer des milliers de grilles il faudra des dixaines d'heures.

Notre algorithme précédent testait tous les chemins possibles. Les solutions comportent rarement plus de quelques centaines de mots, pourtant on cherchait des millions de mots dans le dictionnaire. L'objectif est donc de réduire le nombre d'interrogation du dictionnaire.

On va arrêter l'exploration d'un chemin lorsque ses premières lettres ne forment pas un préfixe d'un mot du dictionnaire. Pour cela, la methode `Dict#exists?` doit permettre de savoir s'il existe un préfixe dans le dictionnaire (github).

```
module Boggle
  class Dict
    def initialize(filepath)
      @filepath = filepath
      @words = File.readlines(filepath).map(&:chomp)
    end

    def exists?(word)
```

```
      @words.each do |w|
        return (w == word ? :found : :prefix) if w.start_with?(word)
      end
      return :not_found
    end
  end
end
```

Voici la méthode `explore!` de l'algorithme `Boggle::Algos::Subtil` (github) :

```
def explore!(steps)
  steps.each do |coord|
    grid.move_to!(coord)
    case @dict.exists?(grid.word)
    when :found
      @solutions << grid.word
      explore!(grid.next_steps)
    when :prefix
      explore!(grid.next_steps)
    else
      # do not expore deeper
    end
    grid.move_back!
  end
end
```

Avec une grille de `3x3` il m'a fallu ~ 50 secondes pour obtenir toutes les solutions. L'algorithme utilisé semble plus intelligent, pourtant l'implémentation de `Boggle::Dict` rend cet algorithme moins performant que la force brute.

Des structures des données à la rescousse

Dans cette situation on va chercher une structure de données adaptée à la recherche de préfixes. Le hachage des chaines de caractères a accéléré la version précédente (diff). Et, maintenant, les Tries vont nous permettre de faire la même chose : *adapter la structure de donnée à l'algorithme*.

Plutôt que d'implémenter une version dédiée de cette structure de donnée, je vais utiliser l'implémentation de la gem algorithms. Il faut penser à faire le `bundle install` et le `bundle exec ...` si nécessaire.

La version de `Containers::Trie` de la gem ne permet pas de vérifier si une chaine est un préfixe d'un des mots contenus dans le Trie. Pour combler ce manque, j'ai du monkey-patcher la classe en question (github).

Après cette modification il devient très rapide de vérifier si un préfixe existe dans le dictionnaire :

```
module Boggle
  class Dict
    def initialize(filepath)
      @trie = Containers::Trie.new
      f = File.open(filepath, "r")
      f.each_line { |line| @trie[line.chomp] = true if line.size > 3 }
      f.close
    end

    def exists?(word)
      return @trie.has_key?(word)? :found : :prefix if @trie.prefix?(word)
      :not_found
    end
  end
end
```

J'obtiens alors un temps d'exécution de ~ 20 secondes en `3x3` comme en `4x4`. Le temps d'exécution sert principalement à construire la structure `Boggle::Dict`. Dans les précédentes situations, la construction du dictionnaire était négligeable, à présent c'est ce qui prend le plus de temps.

Le dictionnaire peut être réutilisé d'une résolution à une autre. On va donc générer 100 grilles 4x4 pour mieux apprécier le gain (github). Cette exécution a duré ~ 40 secondes. À elle seule, la construction du dictionnaire me prend ~ 20 secondes (temps d'exécution sans résolution de grille). On a donc une génération de 100 grilles en ~ 20 secondes soit ~ 5 grilles par seconde.

Conclusion

On a vu que le problème n'a pas pu être résolu par la force brute en un temps acceptable et ce malgré nos tentatives d'optimisation.

Il a fallu identifier l'origine de la complexité du problème : le grand nombre de chemins possibles et le temps d'interrogation du dictionnaire. Dans un premier temps on a du modifier notre algorithme pour examiner moins de chemins. Enfin on a choisi une structure de donnée adaptée aux opérations effectuées par ce nouvel algorithme.

Pour avoir un point de comparaison, la version OCaml que j'utilise est 200 fois plus rapide (1000 grilles par seconde avec 1 thread). Mais, le résultat obtenu avec Ruby est très correct selon moi. Sur un quad-core on peut générer ~ 20 grilles par seconde. Il faudra donc moins d'une minute pour générer 1000 grilles.

J'espère que l'article vous a plu et que ce cheminement vous a rappelé l'importance des algorithmes et des structures de données.

L'équipe Synbioz.

Libre d'être ensembles.

Les QRcodes ont la côte ! ECMAScript 6

Articles connexes

Si le prélèvement à la source m'était conté

30/11/2018

Il était une fois, en l'an 2018, au cœur d'un joli pays nommé France, un peuple qui s'apprêtait à vivre un petit bouleversement. Son chef, qui était fort influençable, avait hérité de par son...

Ruby on Rails en 2018

29/03/2018

Vous me voyez sans doute venir... Utiliser Ruby on Rails en 2018 ? Et puis quoi encore ? Lire la suite...

Elm Europe 2017

22/06/2017

Nous sommes de plus en plus nombreux à découvrir de nouvelles technologies et de nombreux langages de programmation. Chez Synbioz, nous tentons quotidiennement d'explorer cette Voie Lactée numérique...

Retour sur dotJS 2016

13/12/2016

Cette année, dotJS a failli ne pas pouvoir arriver car le théâtre prévu pour accueillir les quelques 1200 participant a brûlé 3 mois avant l'évènement. C'est finalement une autre salle qui a été...

Votre commentaire*

Prénom*

Email*

Submit Comment

Contactez-nous :

03 59 82 80 99

Newsletter

S'abonner

Rejoignez-nous !

Poursuivons la conversation

Nos derniers posts

Linux sur mobile, enfin ?

WebRTC Partie 1 : Signalement par pigeon voyageur

Duplication ou coïncidence ?

Pilotez vos tests Elixir avec des scénarios

SYNBIOZ
Solutions web & mobile

336 avenue de Dunkerque
59130
Lambersart

5 rue de douai 75009 ***Paris***

1 rue du Guesclin 44000
Nantes