Problème de Correspondance de Post

Chejjari Adnane Cordeiro Fonseca Loïc Leloup Loïc

Noubissi Kamgang Allan Perraudin Benjamin

Taheri Amirsaeed

Février 2022

1 Introduction

Dans le cadre du cours "INFO-F308 - Projet transdisciplinaire" et de ce Printemps des Sciences, nous nous sommes penchés sur le Problème de Correspondance de Post, ou PCP, un problème introduit par Emile Louis Post en 1946.
Soit un alphabet Σ de symboles fini et un ensemble de blocs $(w_i, x_i) : w_i, x_i \in \Sigma$ $i = 1, ..., n$, existe-t-il une séquence d'entiers $j_1, ..., j_l$ telle que
$w_{j1}, w_{j2}, ...w_{jl} = x_{j1}, x_{j2}, ...x_{jl}$?

Évidemment il arrive que pour une certaine instance (une liste de blocs), il n'existe aucune solution. Par exemple l'instance suivante : (10, 0), (100,011), (1,01), (11, 10) ne possède aucune solution (on peut le voir facilement car aucun bloc ne peut être placé en premier). Nous utilisons la notation en matrice pour représenter les instances.

$$\begin{pmatrix} 10 & 100 & 1 & 11 \\ 0 & 011 & 01 & 10 \end{pmatrix} \tag{1}$$

Ce qui rend PCP particulièrement intéressant est que ce problème est indécidable, c'est à dire qu'il n'existe pas d'algorithme qui peut déterminer avec certitude si cette solution existe.
Dans le cadre du Printemps des Sciences nous avons tâché de présenter cette problématique relativement complexe et abstraite, à une audience de différents âges et sans connaissances à priori sur le sujet. Pour rendre cela didactique et intuitif, nous avons développé un mini jeu en ligne jouable sur navigateur à l'adresse suivante : *https* : *//pcp.digitelstudios.lu/*, permettant à une personne d'affronter différentes instances de PCP et de tenter de les résoudre en déplaçant des blocs.

2 Critères pour un bon jeu

La première question est de savoir quel type d'instance nous voulons que nos joueurs affrontent dans ce jeu qui fera pâlir les plus grands titres du monde du jeu vidéo.

Le but est de placer le joueur face à des instances de plus en plus difficiles à résoudre. Certains critères sont de mise pour considérer une instance comme "intéressante" pour le joueur, de plus ces critères peuvent permettre de favoriser nos chances de trouver une solution, et ainsi de générer plus d'instances.

2.1 Nombre d'instances par niveau

3 niveaux de difficultés sont présents dans le jeu, facile intermédiaire et difficile. Ces trois niveaux diffèrent par la solution minimale des instances qui les composent. De 3 à 4 pour le niveau facile, intermédiaire de 4 à 7 et 8 à 16 pour le

dernier. Exemple d'instance "facile":

$$\begin{pmatrix} 11 & 0 & 001 \\ 1 & 101 & 0 \end{pmatrix} \tag{2}$$

$\rightarrow$ solution de longueur 3 : (1)(2)(1)

Le but est que le joueur ne retombe pas sur une même instance, c'est pourquoi nous voulions générer une base de donnée suffisamment remplie pour permettre au jeu une certaine durée de vie. La répartition des tailles de solution est détaillée dans la section 4.2.

2.2 Réutilisation de bloc

L'un des aspects importants du jeu est de devoir utiliser plusieurs fois le même bloc. L'intérêt est d'amener le joueur à penser aux différentes combinaisons possibles en pensant aux "coups" suivant lorsqu'il place un bloc. Un nombre trop important de blocs dans une instance réduit cette possibilité et rajoute une difficulté jugée inintéressante.

3 Méthodologie

Comme mentionné plus haut il n'existe pas d'algorithme nous permettant de résoudre le PCP, or afin de rendre le jeu correct, il nous fallait des instances dont on sait qu'elles sont résolubles. Pour cela il suffit de trouver au moins une solution pour une instance donnée. Ajoutons à cela le fait qu'il faut connaître la longueur minimale de la solution.

3.1 Découpe manuelle

La première approche que nous avons testée, une des plus naturelle, consiste à prendre une string aléatoire sur notre alphabet, de la dupliquer et de placer les deux strings identiques l'une au dessus de l'autre comme on pourrait voir pour n'importe quelle solution. A partir de cette solution, il suffit de "découper" des blocs de tailles variables, et nous voilà avec une instance et sa solution. Plusieurs problèmes se posent malheureusement avec cette méthode.

Tout d'abord, il devient difficile de trouver plusieurs blocs identiques, en effet cela dépend beaucoup de la solution de départ, si celle ci n'a pas de schéma qui se répète alors on ne pourra pas découper plusieurs fois le même bloc. De même si on veut "forcer" des répétitions en modifiant légèrement la solution à la main, on se retrouve très souvent avec des blocs triviaux. Si on veut pouvoir générer un grand nombre d'instances, cette méthode ne convient pas car trop coûteuse en temps et offre trop peu de contrôle sur ses résultats, particulièrement pour des instances difficiles.

3.2 Backtracking itératif

La seconde approche est celle d'un backtracking itératif à partir d'instances générées aléatoirement. L'idée est simplement de résoudre une instance pour une certaine profondeur (16 maximum) et de parcourir l'arbre des possibilités en plaçant les blocs à la suite des autres, jusqu'à trouver (ou non) une solution. Le backtracking a l'avantage d'être assez simple à implémenter (voir algorithme 1), le reste du travail s'est concentré sur des optimisations décrites dans la section 4 suivante.

Algorithm 1 solve(depth, instance)

```
1:  if isSolution() then
2:      return true
3:  end if
4:  for bloc in instance do
5:      if bloc can be push then
6:          push(bloc)
7:          if solve(depth − 1, instance) == true then
8:              return true
9:          end if
10:         pop(bloc)
11:     end if
12: end for
```

4 Optimisation

Certaines optimisations, pour la résolution des instances, ont vu le jour au fil de notre travail pour gagner du temps de recherche et ainsi générer plus d'instances.

4.1 Filtres

4.1.1 Aucune solution

Directement inspiré de l'article [Lor01] sur la génération d'instances PCP, nous avons implémenté dans notre code 3 filtres qui permettent d'éviter de chercher une solution sur des instances qui ne peuvent en avoir aucune.

prefix filter : un bloc possède un préfixe si il possède une correspondance, donc il peut être placé en premier dans la solution (ex : (110, 11)). Si aucun bloc ne possède de préfixe, aucun bloc ne peut être placé.

length balance filter: ce filtre élimine les instances dans lesquelles les longueurs ne peuvent être contrebalancées. Exemple dans l'instance (3) :

$$\begin{pmatrix} 100 & 010 & 11 & 010 \\ 1 & 10 & 01 & 1 \end{pmatrix} \tag{3}$$

On ne peut commencer que par le bloc (1) qui possède un préfixe, or comme celui-ci est plus long en haut qu'en bas, il faut un bloc plus long en bas pour pouvoir avoir une solution avec deux strings de la même taille. Dans cette instance cela ne sera jamais possible avec aucun des autres blocs. Donc il n'existe aucune solution.

element balance filter : dans ce filtre-ci on ne regarde que le nombre de 0 et de 1. Pour chaque bloc qui peut être posé en premier, puisque le nombre de 0 et 1 dans les deux strings de la solution doivent être égaux, si une des strings du haut ou du bas à un nombre de 0 ou de 1 qui ne peuvent pas être compensé plus tard par d'autres blocs, il n'y a pas de solution.

$$\begin{pmatrix} 100 & 01 & 10 & 0 \\ 1 & 111 & 01 & 1 \end{pmatrix} \tag{4}$$

Dans cet exemple, la solution devra commencer par le bloc (1), il y aura donc deux 0 en trop dans la string du haut. Aucun autre bloc ne pourra rattraper ce nombre de 0 en bas, il n'existe donc pas de solution.

4.1.2 Solution trop courte

La génération des blocs est aléatoire, cependant nous souhaitons éviter la création de certains blocs qui forment des solutions trop courtes pour notre jeu (taille 1 et 2).

- Un bloc ne doit pas être "trivial", on entend par la un bloc constitué des deux mêmes strings (ex : $(01, 01)$). Il est évident qu'un bloc trivial forme à lui seul une solution de taille 1.

- Nous évitons d'avoir deux blocs symétriques. Si deux blocs sont symétriques alors il existe une solution de taille 2 en les plaçant à la suite, ainsi qu'une solution de taille 4, puis 6 etc...

$$\begin{pmatrix} 10 & 1 & 001 \\ 1 & 01 & 0 \end{pmatrix} \tag{5}$$

$\rightarrow$ solution de longueur 2 : (1)(2) (Le bloc 1 est le symétrique du bloc 2)

Nous portons évidemment notre attention sur la plus petite solution possible, dans ce cas il en existera toujours une de taille 2, jugée trop facile pour le joueur. Ces instances vont nous faire perdre du temps de résolution et ne se retrouveront dans aucun niveau, mieux vaut s'en débarrasser dès le début.

4.2 Profondeur itérative

Le choix des paliers est lié à notre volonté d'obtenir suffisamment d'instances pour chaque niveau. En pratique, les instances aléatoires ont statistiquement plus de chances d'avoir une petite solution qu'une grande, d'où la difficulté de trouver des solutions de grande taille. L'article [Lor01] énonce clairement cet état de fait. Pour garantir un nombres raisonnable d'instances difficiles

(plusieurs centaines), nous avons donc élargi le champ de recherche pour le troisième palier (entre 8 et 16), et réduit les premiers.

C'est pourquoi nous avons implémenté un backtracking itératif sur la profondeur maximale de recherche, et ceci par palier de difficulté. Sans cela, une instance qui possède une solution de 3 blocs et également de 15 blocs, n'ira pas jusqu'à la profondeur 15, puisque lors de la génération de la première itération du palier 1, une solution est déjà trouvée. Nous évitons donc de parcourir une profondeur trop élevé pour des instances avec une petite solution (minimale). La disparité du nombre d'instances trouvées par niveau nous a convaincu de diviser notre recherche dans le backtracking.

4.3 Programmation dynamique

Cette optimisation n'a pas vu le jour dans ce projet, elle reste néanmoins intéressante et mérite d'être mentionné.

Lors du parcours de l'arbre dans le backtracking, certains agencements de blocs ne mènent à aucune solution, or il se peut que la même configuration se présente une nouvelle fois dans la résolution. Alors nous pourrions garder en mémoire ces configurations pour éviter de devoir reparcourir une partie de l'arbre qui a déjà été explorée, et dont on sait qu'elle ne mène à rien.

5 Résultats

Notre base de donnée finale se compose actuellement de 38687 instances pour le niveau 1 (solution max 4), 13533 pour le niveau 2 (max 7) et 3835 pour le niveau 3 (max 16). Ce chiffre varie puisque notre de base de donnée augmente en temps réel au fur et à mesure que des instances sont trouvées.

```
+-------+--------------+
| level | COUNT(pairs) |
+-------+--------------+
|  NULL |            2 |
|     0 |        38687 |
|     1 |        13533 |
|     2 |         3835 |
+-------+--------------+
4 rows in set (0,05 sec)
```

Figure 1: Base de donnée des instances

References

[Lor01] Richard J. Lorentz. “”Creating Difficult Instances of the Post Correspondence Problem””. In: (2001).