INFO0054 - Programmation fonctionnelle
Projet: Puzzles réguliers

Jean-Michel Begon

2018-2019

Un automate fini déterministe est défini par un tuple $(Q, \Sigma, \delta, s, F)$, où

Q est un ensemble fini d'états ;

Σ est un alphabet fini ;

$\delta : Q \times \Sigma \to Q$ est la fonction de transition ;

s est l'état initial ;

$F \subseteq Q$ est l'ensemble des états accepteurs.

En outre, on appelle "mot" une suite finie de symboles $\{\sigma_i\}_{i=1}^{N}$ $(\sigma_i \in \Sigma)$.

On dit qu'un mot $\{\sigma_i\}_{i=1}^{N} = \sigma_1 \ldots \sigma_N$ est accepté par l'automate si la propriété suivante est satisfaite :

— $q_1 = s$

— $q_{i+1} = \delta(q_i, \sigma_i) \quad 1 \leq i \leq N$

— $q_{N+1} \in F$

Autrement dit, la suite de symboles permet de passer de l'état initial vers un des états accepteurs.

L'ensemble des mots acceptés par l'automate forme un langage. La classe des langages reconnaissables par des automates finis déterministes s'appelle la classe des langages réguliers.

Dans le cadre de ce projet, nous allons nous intéresser à des puzzles qui peuvent se mettre sous la forme de tels automates. L'ensemble des solutions forme alors un langage, dénoté L. Le sous-langage des mots acycliques est dénoté $L_{\not\circlearrowleft}$.

Par mot acyclique, on entend un mot tel que la séquence de transitions qui lui est liée ne passe jamais deux fois par le même état.

1 Exemple — Le loup, le mouton et le chou

Soit le puzzle suivant :

> "Vous êtes accompagné d'un loup, d'un mouton et d'un chou. Vous devez les faire traverser une rivière mais vous n'avez à votre disposition qu'une barque ne pouvant transporter qu'un seul des trois avec vous. Si le loup est seul avec le mouton, il va le manger. Si le mouton est seul avec le chou, il va le manger. Est-il possible faire traverser le loup, le mouton et le chou ?"

Son automate est illustré à la figure 1. Chacun du passeur (P), du loup (L), du mouton (M) ou du chou (C) peut se trouver du côté initial ou de l'autre côté de la rivière. A ces seize états, on ajoute un état particulier, le *sink*, tel que $sink = \delta(sink, \sigma) \quad \forall \sigma \in \Sigma$; c'est-à-dire qu'il s'agit d'un état dont on ne peut sortir.

L'état initial est $s = (\{\}, \{P, L, M, C\})$: tous les quatre sont sur la rive initiale. L'ensemble des états accepteurs est le singleton $F = \{(\{P, L, M, C\}, \{\})\}$.

L'alphabet est composé de quatre symboles $\Sigma = \{p, l, m, c\}$ qui représentent respectivement "le passeur traverse seul", "le passeur traverse avec le loup", "le passeur traverse avec le mouton", "le passeur traverse avec le chou".

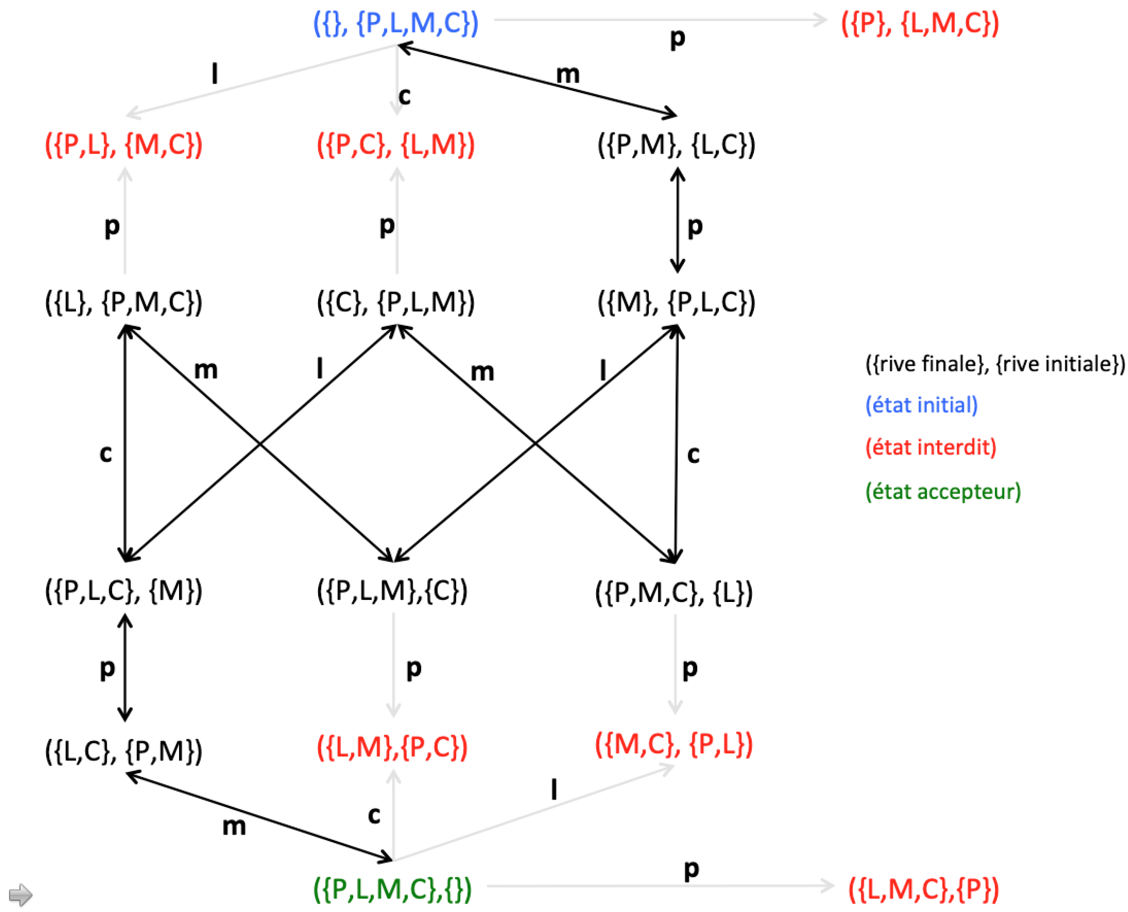


FIGURE 1 – Graphe du loup, du mouton et du chou. Les transitions manquantes sont toutes dirigées vers le *sink*.

Le problème admet deux solutions (*i.e.* mots) sans cycle de sept symboles : *mplmcpm* et *mpcmlpm*. La première se traduit par :

s **:** On démarre de la rive initiale : $(\{\}, \{P, L, M, C\})$.

$\rightarrow_m$ On transporte le mouton de l'autre côté : $(\{P, M\}, \{L, C\})$.

$\rightarrow_p$ On revient en laissant le mouton : $(\{M\}, \{P, L, C\})$.

$\rightarrow_l$ On transporte le loup de l'autre côté : $(\{M, P, L\}, \{C\})$.

$\rightarrow_m$ On revient avec le mouton : $(\{L\}, \{P, M, C\})$.

$\rightarrow_c$ On transporte le chou de l'autre côté : $(\{L, P, C\}, \{M\})$.

$\rightarrow_p$ On revient en laissant le loup et le chou : $(\{L, C\}, \{P, M\})$.

$\rightarrow_m$ On transporte le mouton de l'autre côté : $(\{P, L, M, C\}, \{\}) \in F$.

La solution à treize symboles *mpcmlcmlcmlpm* contient un cycle car elle repasse par les états $(\{M\}, \{P, L, C\})$, $(\{P, M, C\}, \{L\})$, $(\{C\}, \{P, L, M\})$, $(\{P, L, C\}, \{M\})$.

2 Enoncé

2.1 Le *solver* /10

Le but du projet est d'implémenter un *solver* générique de puzzles réguliers. La première fonction à implémenter est :

```
(rp-solve s adj acc-state?)
```

Elle prend trois arguments, tous liés à un même puzzle $P = (Q, \Sigma, \delta, s, F)$:

1. `s` : la représentation de l'état initial ;
2. `adj` : la fonction d'adjacence de P : $adj(q) = \{(\sigma, q_j) \in \Sigma \times Q | q_j = \delta(q, \sigma)\} \quad \forall q \in Q$. On représentera une paire (σ, q_j) par une paire pointée et l'ensemble de ces paires par une liste.
3. `acc-state?` : le prédicat unaire défini sur l'ensemble Q et valant $True$ si et seulement si son argument est issu de F.

Elle renvoie un itérateur paresseux du sous-langage de P constitué des mots sans cycle ($L_{\not\circlearrowleft}$) par ordre croissant de longueur. Un mot est représenté par la liste de ses symboles.

Un itérateur paresseux pour une suite $m_1, \ldots, m_p$ est une suite $f_1, \ldots, f_p, f_{p+1}$ telle que les p premiers éléments sont des fonctions vérifiant $(f_k) = (m_k.f_{k+1})$ et $(f_{p+1}) =' ()$.

En mots, l'évaluation de f_k pour $1 \leq k \leq p$ est une paire pointée dont le `car` est m_k et le `cdr` est f_{k+1}. Le programme fonctionne donc même dans le cas d'un langage infini.

La fonction `rp-solve` renvoie bien une *fonction* et non une paire ! C'est l'évaluation de cette fonction qui produit la paire.

Toutes les informations relatives à un puzzle donné doivent être encapsulées dans les trois paramètres de la fonction.

2.2 Le taquin /7

Un jeu de taquin (figure 2) est constitué d'une grille de $N \times N$ cases, contenant des pièces numérotées de 1 à $N^2 - 1$. La pièce manquante permet de faire coulisser une des pièces adjacentes. Pour une configuration donnée, il y a donc au plus quatre actions possibles, cristallisées par l'alphabet :

u **:** la pièce au-dessus du trou vient prendre sa place, le trou se déplace donc vers le haut ;

d **:** la pièce en-dessus du trou vient prendre sa place, le trou se déplace donc vers le bas ;

l **:** la pièce à gauche du trou vient prendre sa place, le trou se déplace donc vers la gauche ;

r **:** la pièce à droite du trou vient prendre sa place, le trou se déplace donc vers la droite.

A partir d'une configuration donnée (figure 2a), le but du jeu est de remettre les pièces dans l'ordre de 1 à $N^2 - 1$ de gauche à droite et de haut en bas (figure 2b). Il n'y a qu'un seul état accepteur. Pour l'exemple de la figure 2, un mot accepté serait

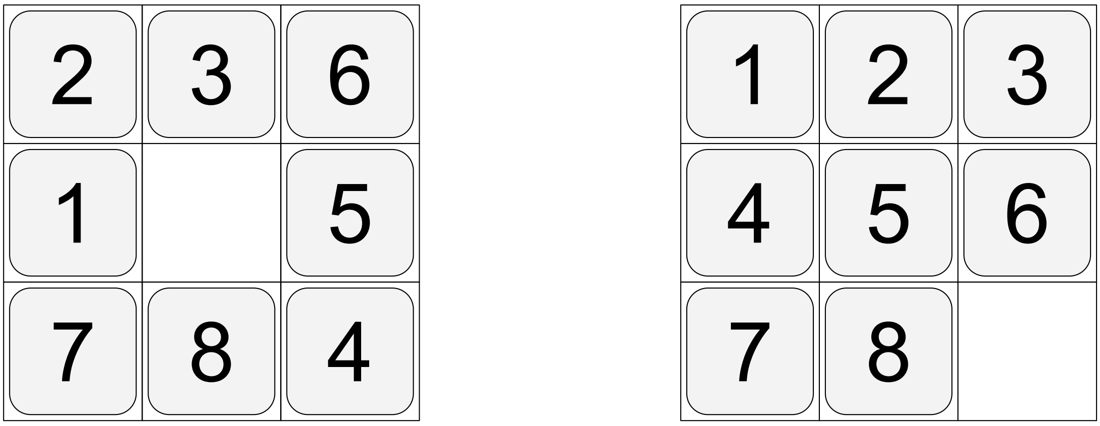
$$ruldlurdlurrdlurdlurddlurdlurdllurdlurdrulldrrulldrruldruldlurrdlurdluldrurdlurdllurrdllurdlurdr$$

Concernant cette partie du projet, vous devez fournir un fichier `taquin.scm` qui expose les trois fonctions suivantes :

`taquin-make-state` une fonction qui prend en entrée la représentation d'un taquin telle qu'expliqué ci-après et qui renvoie l'état de l'automate correspondant ;

`taquin-adj-states` la fonction d'adjacence du jeu de taquin ;

`taquin-acc-state?` le prédicat d'acceptation.



(a) Taquin mélangé

(b) Taquin résolu

FIGURE 2 – Jeu de taquin

La fonction `taquin-make-state` prend en entrée une liste de N listes. La $i$ème sous-liste représente la $i$ème ligne du taquin, en commençant à compter depuis le haut. Les nombres sont numérotés à partir de 1 et le trou est symbolisé par le symbole `x`. Par exemple, la liste `'((2 3 6) (1 x 5) (7 8 4))` correspond au taquin de la figure 2a. Vous pouvez adopter une autre représentation interne des états.

Notez qu'il n'y a pas de restriction sur la taille du taquin ; les fonctions `taquin-make-state`, `taquin-adj-states` et `taquin-acc-state?` doivent gérer le cas général de $N \times N$ cases ($N \in \mathbb{N}_0$).

2.3 Heuristique ________________________________ /3

Dès que $N \geq 3$, le taquin est un puzzle dont l'espace d'états Q est suffisamment large pour qu'une exploration naïve ne donne pas de résultats dans un temps raisonnable. Une manière de contourner ce problème est d'explorer les états les plus prometteurs d'abord. Pour ce faire, on définit une fonction heuristique $H : Q \to \mathbb{R}_+$ et qui est telle que

— $H(q) = 0 \iff q \in F$;

— $0 \leq H(q) \leq \phi(q)$ (admissibilité)

où $\phi(q)$ est la longueur du plus petit mot accepté par l'automate au départ de l'état q. Une bonne heuristique fournit une borne inférieure raisonnable à $\phi(n)$. De la sorte, il est raisonnable de penser que si $H(q_1) < H(q_2)$, l'état q_1 est plus prometteur que l'état q_2.

Deux heuristiques communément utilisées pour le taquin sont

$H_1(q)$: le nombre de cases mal placées de l'état q ;

$H_2(q)$: la distance de Manhattan entre chaque case de l'état q et sa position finale.

Pour cette partie du projet, il faut ajouter la fonction `taquin-heuristic` au fichier `taquin.scm` et la rendre visible. Il faut également ajouter et rendre visible une fonction `rp-solve-heuristic` dans le fichier `solver.scm`. Cette fonction prend, en plus des trois arguments de `rp-solve`, la fonction heuristique et renvoie un itérateur paresseux des solutions dans l'ordre d'exploration de l'heuristique.

3 Rapport

Il n'est pas nécessaire de rendre un rapport pour ce projet. Si toutefois vous souhaitez défendre vos choix d'implémentation, vous pouvez rendre un court rapport. Dans tous les cas, il ne s'agit pas d'expliquer le détail de votre implémentation.

4 Soumission et remarques

Le projet est à réaliser par groupes de deux. Il doit être rendu pour le 23 avril 2019, 23h59 via la plateforme de soumission (`http://submit.montefiore.ulg.ac.be/`) sous la forme d'une archive au format `tar.gz` contenant :

— un fichier intitulé `solver.scm` contenant les fonctions `rp-solve` et `rp-solve-heuristic`,

— un fichier intitulé `taquin.scm` contenant tous les éléments relatifs au puzzle,

— un éventuel rapport au format PDF.

Veillez à rendre les différentes fonctions publiques en ajoutant l'instruction `(provide rp-solve)`, etc. en tête du fichier correspondant.

Utilisez vos identifiants ULiege comme nom de groupe (sxxxxxx-sxxxxxx).

N'oubliez pas de spécifier toutes les fonctions que vous définissez (même à l'aide d'un `letrec`).

Si nécessaire, vous pouvez utiliser l'implémentation de base des ensembles avec les commandes `set`, `set-member?` et `set-add` (à ne pas confondre avec `set-add!`).

Bon travail !