INFO0054 - Programmation Fonctionnelle

Projet 2024-2025 (inspiré par l'énoncé de 2018-2019 de Jean-Michel Begon)

Introduction

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 ;
- $\delta: Q \times \Sigma \rightarrow Q$ 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 :

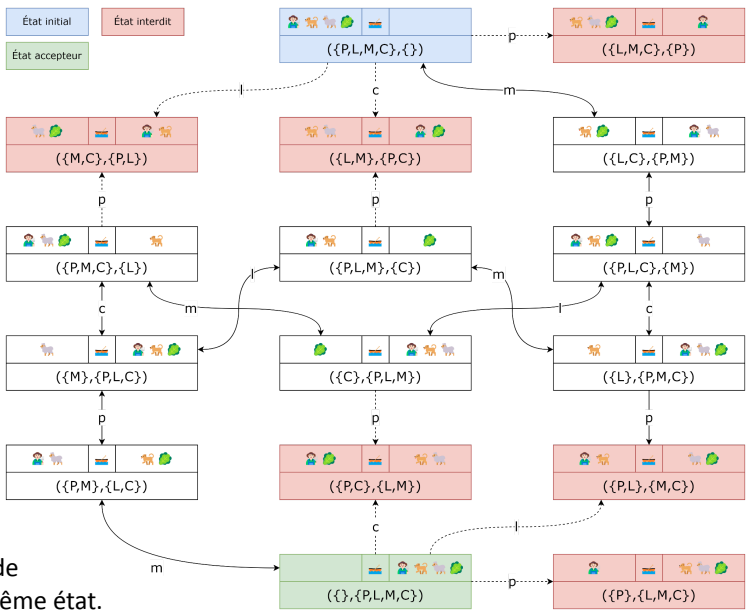
- $q_1 = s$
- $q_{i+1} = \delta(q_i, \sigma_i)\ 1 \leq i \leq N$
- $q_{N+1} \in F$

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 problèmes (e.g., 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[1] est dénoté $L_\emptyset$.

Exemple : Problème du loup, ~~de la chèvre~~ du mouton et du chou

> *"Il était une fois un fermier qui alla au marché et acheta un loup, [un mouton] et un chou. Pour rentrer chez lui, le fermier loua une barque pour traverser une rivière. Mais la barque ne pouvait contenir que le fermier et l'un de ses achats: le loup, [le mouton] ou le chou. S'ils étaient laissés sans surveillance, le loup allait dévorer la chèvre ou [le mouton] allait manger le chou. Le défi du fermier fut d'arriver de l'autre côté de la rivière avec ses achats intacts. Comment a-t-il fait ?"* (Source: Wikipédia)

Ce problème est illustré ci-dessous. Chacun du passeur (le fermier) 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 16 états, on ajoute un état particulier, le sink, tel que $sink = \delta(sink, \sigma)\ \forall \sigma \in \Sigma$; c'est-à-dire qu'il s'agit d'un état dont on ne peut sortir. Les transitions manquantes dans la figures sont toutes dirigées vers un sink--i.e., un état interdit.

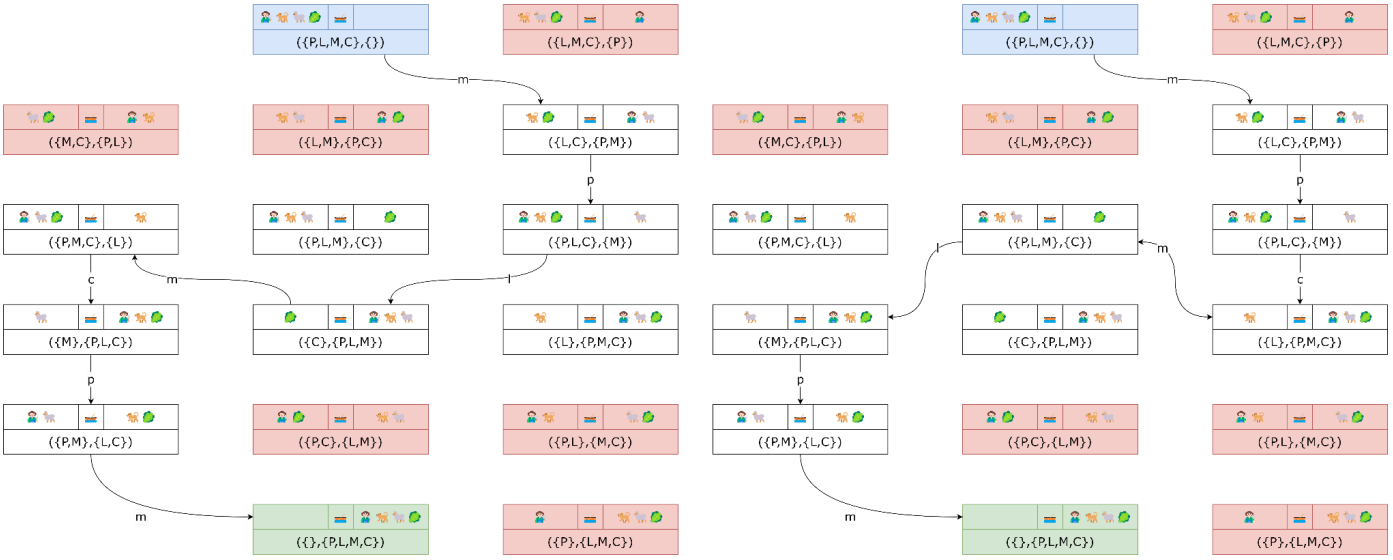


L'état initial est $s = (\{P, L, M, C\}, \{\})$: tous les quatre sont sur la rive initiale. L'ensemble des états accepteurs est le singleton $F = \{(\{\}, \{P, L, M, C\})\}$. L'alphabet est composé de quatre symboles $\Sigma = \{p, l, m, c\}$ qui représentent respectivement :

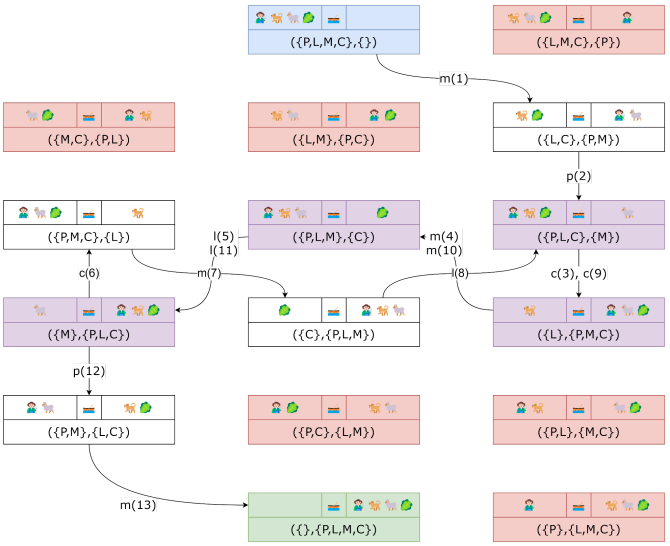
- Le passeur traverse seul.
- Le passeur traverse avec le loup.
- Le passeur traverse avec le mouton.
- Le passeur traverse avec le chou.

[1] 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.

Le problème admet deux solutions (i.e. mots) sans cycle de 7 symboles : *mplmcpm* et *mpcmlpm*.



La solution à 13 symboles *mpcmlcmlcmlpm* contient un cycle car elle repasse par les états en violet.



Question : dans le cadre de ce problème, définissez : Q, Σ, s, F, δ, L et $L_{\emptyset}$.

L'énoncé du projet

L'objectif de ce projet est de développer un solveur *générique* pour de tels automates puis, via des abstractions que vous devez concevoir et implémenter, résoudre une série de puzzles.

Partie	Description	Points
Abstractions	Utilisez efficacement les fonctions d'ordre supérieur, les classes, les traits et les synonymes de type pour réduire la duplication du code, augmenter la modularité et rendre le code plus compréhensible.	3

La fonction *accept*.	La fonction *accept* est une fonction qui génère un booléen à partir d'un mot. Elle génère true si le mot conduit l'automate fini déterministe à l'un des états acceptés.
	
	Vous pouvez choisir si accept est une fonction qui appartient à un objet d'un type ou d'un trait particulier, ou comme une fonction qui prend en entrée de tels objets. Vous trouverez ci-dessous quelques exemples.[2]
	
	`ex.accept(List(1,1,1,1)) → true`
	`ex.accept(List(1,1,1,2)) → false`
	`ex.accept(List(1,1,1))  → false`
	`accept(ex,List(1,1,1,1)) → true`
	`accept(ex,List(1,1,1,2)) → false`
		3

La fonction *solve*.	La fonction *solve* calcule tous les mots sans cycles qui conduisent à des états acceptés à partir de l'état initial. Vous ne devez considérer que les mots sans cycles, car vous risquez de vous retrouver avec un ensemble de mots infiniment grand.
	
	`solve(ex) → List(`
	`List(2,2),List(1,1,2),List(1,2,1),List(2,1,1),List(1,1,1,1))`
	
	`scala> ex.solve() → // idem`
	
	Tip 1 : Cette fonction s'appuiera sur la fonction *accept* définie précédemment. Vous aurez *très probablement* besoin de fonctions utilitaires et auxiliaires telles que *la recherche des états adjacents*. Une telle fonction renvoie, à partir d'un état, toutes les paires de $\Sigma \times Q$ menant aux états adjacents autorisés.
	
	I.e., $adjacence(q) = \{(\sigma, q') \in \Sigma \times Q | q' = \delta(q, \sigma)\}$
	
	Tip 2 : Vous aurez très probablement besoin de garder une "trace" des *mots* et de leurs "*chemins*" correspondants dans les automates finis déterministes.
		6

L'évaluation non stricte	Certains automates finis déterministes peuvent être très complexes, et la recherche de tous les mots sans cycles peut prendre un temps déraisonnable (ce que vous découvrirez probablement en travaillant sur le taquin). Certains peuvent être intéressés par une seule solution ou demander des solutions individuellement, la solution suivante étant recherchée « à la demande ». Cela peut être réalisé avec une évaluation non stricte (par exemple, avec des listes paresseuses). Dans cette partie du projet, vous devez développer une version de *solve*, *lazysolve*, qui repose sur l'évaluation non stricte.	4
Chaînes binaires impaires	Voici une description d'un automate fini (FSA) qui accepte les chaînes binaires impaires. Les chaînes binaires sont les mots. L'automate possède deux états, souvent appelés 'pair' (even) et 'impair' (odd). L'état 'pair' est l'état initial, car une chaîne vide (contenant zéro '1') est considérée comme paire. En lisant la chaîne d'entrée, il passe d'un état à l'autre en fonction de la rencontre d'un '0' ou d'un '1'. Si l'état final après la lecture de la chaîne entière est l'état 'impair', la chaîne est acceptée car elle contient un nombre impair de '1'. Sinon, la chaîne est rejetée. L'utilisation de la fonction solve doit renvoyer tous les mots sans cycles. Combien de mots de ce type doivent être renvoyés ? - Montrez le résultat de solve. - Montrez que 10101011 est accepté. - Montrez que 10101010 n'est pas accepté *Vous perdrez des points si cette partie ne peut pas être appliquée à solve ou accept.*	2

[2] Cet exemple est un automate finis déterministe qui, à partir de 0, nous donne toutes les séquences de +1 ou +2 qui nous donnent la somme de 4.

<table>
<tr><td>Problème du loup, du mouton et du chou</td><td>Implémentez un automate fini déterministe pour le problème du loup, du mouton, et du chou.

- Montrez le résultat de solve.
- Montrez que mpcmlcmlcmlpm est accepté.
- Montrez que ppp n'est pas accepté.

Vous perdrez des points si cette partie ne peut pas être appliquée à solve ou accept.</td><td>4</td></tr>
<tr><td>Le taquin</td><td>Un jeu de taquin 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 au plus quatre actions possibles, représentées 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, 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. Il n’y a donc qu’un seul état accepteur.

Pour l’exemple ci-dessous, le mot accepté serait :

ruldlurdlurrdlurdlurddlurdlurdllurdlurdrulldrrulldrruldruldlurrdlurdluldrurdlurdllurrdllurdlurdr

<table><tr><td>2</td><td>3</td><td>6</td></tr><tr><td>1</td><td></td><td>5</td></tr><tr><td>7</td><td>8</td><td>4</td></tr></table> ⇒ <table><tr><td>1</td><td>2</td><td>3</td></tr><tr><td>4</td><td>5</td><td>6</td></tr><tr><td>7</td><td>8</td><td></td></tr></table>

Tip 1 : L'état initial doit être fourni. Assurez-vous d'éviter l'utilisation de générateurs de nombres aléatoires dans votre programme fonctionnel.

Tip 2 : Si vous examinez tous les états possibles, un seul d'entre eux a une solution. En trouver un peut également prendre beaucoup de temps. Essayez de résoudre le problème pour les puzzles 2x2.

Tip 3 : La pièce manquante peut être représentée par un zéro.

- Montrez le résultat de solve [[_ 2][1 3]]
- Montrez le résultat de solve [[2 1][3 _]]
- Montrez que dudr est accepté pour [[_ 2][1 3]]
- Montrez le résultat de solve pour l’exemple

Vous perdrez des points si cette partie ne peut pas être appliquée à solve ou accept.</td><td>4</td></tr>
<tr><td>Spécifications</td><td>Toutes les fonctions ont une spécification (même les fonctions auxiliaires et utilitaires). Vous devez spécifier ce que la fonction fait, et non comment elle le fait.</td><td>2</td></tr>
<tr><td>Rapport technique</td><td>Votre rapport technique peut être rédigé en français. Dans ce rapport, vous décrirez et motiverez brièvement vos ADT et les choix d'implémentation de vos fonctions. Il vous sera également demandé d’y répondre aux questions posées et de réfléchir de manière critique à votre travail.

Le rapport doit être structuré comme un rapport technique : page de garde, résumé, introduction, « milieu », conclusions, références et – si nécessaire – annexes.</td><td>2</td></tr>
</table>

Bonus	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 $H: Q \rightarrow R^+$ telle que 1) $H(q) = 0 \Leftrightarrow q \in F$ et $0 \leq H(q) \leq \phi(q)$ où $\phi(q)$, l'admissibilité de 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 de la sorte, il est raisonnable de penser que si H(q1) < H(q2), l'état q1 est plus prometteur que l'état q2. Voici deux heuristiques : - H1(q) : le nombre de cases mal placées de l'état q ; - H2(q) : la distance de Manhattan entre chaque case de l'état q et sa position finale. Étendez le programme de façon à ce que les heuristiques puissent être appliquées *(à n'importe quel automate)*. Implémentez les deux heuristiques et analysez leur comportement pour des puzzles de taille 3 (et de taille 4 si vous vous sentez aventureux).	3

Nous évaluerons également l'efficacité de vos fonctions, que vous devrez évaluer ou critiquer *brièvement* dans votre rapport. Par exemple, évitez d'utiliser **append** si vous pouvez bénéficier de **cons** et **reverse**, visez la récursivité terminale autant que possible, etc.

Nous vous encourageons également à consulter la bibliothèque standard de Scala pour toute fonction, abstraction, etc. utile. Vous n'êtes pas obligé d'utiliser les ADT que nous allons développer dans ce cours ; à la place, vous pouvez utiliser ceux proposés par Scala.

Nous vous encourageons à rechercher des informations sur des livres ou des articles sur le Web. Si vous en tirez des informations utiles ou importantes, assurez-vous de les citer et de les référencer de manière appropriée dans votre rapport.

Modalités

Le projet sera réalisé en **groupes de trois étudiants** et devra être remis au plus tard le **5 décembre 2024 à 23h59**. Votre code devra être remis sous forme de fichier ZIP et votre rapport sous forme de PDF. Veillez à remettre les deux séparément. Toutes les soumissions se feront via eCampus.

Pour anticiper les éventuels problèmes liés à la formation des groupes, vous devrez constituer ceux-ci sur eCampus avant le **24 octobre 2024**. Passé cette date, il ne sera plus possible de former un groupe et donc de remettre le projet.

La participation au projet est obligatoire. Les étudiants qui ne soumettent rien pour le projet, qui comprend un rapport vierge et/ou un code source vide, recevront une note d'absence (A) pour le cours. Attention : les fichiers vides, le code source avec uniquement des noms, etc., ne sont pas considérés comme une soumission et entraîneront un A.

N'attendez pas la dernière minute pour signaler un problème.

Pour rappel, le plagiat est sévèrement puni. Je rappelle également aux étudiants de consulter la « Charte d'utilisation des outils d'intelligence artificielle par l'étudiant » (français) ou « Charter for the use of artificial intelligence tools by students » (anglais). Vous êtes encouragés à discuter d'idées et d'approches avec vos pairs, mais vous n'êtes pas autorisé à partager votre code.

Activité d'auto-évaluation collective

Une activité d'auto-évaluation collective sera organisée la semaine suivant la date limite de remise. En groupe, vous évaluerez et donnerez votre feedback sur votre projet. Les groupes dont les auto-évaluations se rapprocheront de nos évaluations obtiendront un point bonus. L'activité n'est pas obligatoire, mais vous devez être présent physiquement pour bénéficier d'un point bonus.