Programmation Fonctionnelle

Projet: Automates finis déterministes

Group 01

Authors
Mehdi Boustani - s221594
Abdelkader Albashityalshaier - s211757
Alexandre Bourguignon - s211885

Professor
Christophe Debruyne

Assistant
Baptiste Vergain

2024-2025

Contents

1 Introduction

Ce rapport technique présente le travail réalisé dans le cadre du projet du cours INFO0094. Le document détaille les choix conceptuels et techniques effectués pour concevoir les différentes composantes du projet.

2 Automate fini déterministe (`dfa.scala`)

Dans cette section nous décrirons les traits et les fonctions utilisées pour représenter l'automate fini déterministe.

2.1 Trait StateDFA

Ce trait représente un état dans le DFA. Il est abstrait et peut être implémenté par des types concrets. Ce trait est utilisé pour permettre la généralisation et l'abstraction des états pour adapter le DFA à différents contextes d'utilisation.

2.2 Type Symbol et Word

`Symbol[+A]` : Définit un symbole appartenant à l'alphabet. Par défaut, il est typé comme A, mais il est générique pour permettre l'utilisation d'alphabets variés (ex. : Int, Char, String).

`Word[+A]` : Représente un mot, c'est-à-dire une séquence ordonnée de symboles.
Ces types rendent le DFA flexible pour gérer différents alphabets et simplifient la gestion des mots en utilisant les structures de données Scala comme Seq.

2.3 Trait DFA

Ce trait est l'interface principale pour définir un automate fini déterministe. Il encapsule les éléments (Q, Σ, δ, s, F).

- `states: Set[S]` : Représente l'ensemble des états possibles (Q).
- `alphabet: Set[Symbol[A]]` : Définit l'ensemble des symboles formant l'alphabet (Σ).
- `transition(state: S, symbol: Symbol[A]): Option[S]` : Implémente la fonction de transition (δ). Retourne un nouvel état si une transition valide existe, ou None sinon.
- `initialState: S` : Définit l'état initial (s).
- `acceptingStates: Set[S]` : Définit l'ensemble des états acceptants (F).

Ce trait fournit une structure claire pour définir un DFA tout en étant générique grâce aux types S (pour les états) et A (pour les symboles).

2.4 Les fonctions définies dans l'extension du trait DFA

Les extensions définies dans le fichier permettent d'ajouter des fonctionnalités supplémentaires aux objets implémentant le trait `DFA`.

2.4.1 Fonction `getAdjacentStates`

La méthode `getAdjacentStates` retourne l'ensemble des transitions possibles à partir d'un état donné. Cela correspond à l'ensemble des couples $(\sigma, q') \in \Sigma \times Q$ tels que $q' = \delta(q, \sigma)$, où q est l'état actuel.

Utilité Cette fonction facilite l'exploration des transitions disponibles depuis un état spécifique. Elle est utile, par exemple, pour la visualisation de l'automate ou pour des algorithmes exploitant les transitions (comme des recherches ou des tests).

Implémentation Voici le fonctionnement de la méthode étape par étape :

1. **Parcours de l'alphabet** : On parcourt l'ensemble des symboles de l'alphabet `dfa.alphabet` pour tester chaque symbole en tant que transition potentielle.
2. **Application de la fonction de transition** : Pour chaque symbole, on applique la fonction de transition `dfa.transition` à l'état donné et au symbole.
3. **Filtrage des transitions valides** : La fonction de transition retourne une `Option[S]`, ce qui signifie qu'elle peut renvoyer `None` (pas de transition définie) ou `Some(q')` (transition valide vers un état q'). Seules les transitions valides (`Some(q')`) sont conservées.
4. **Création des paires** : Pour chaque transition valide, un couple (symbole, nouvel état) est créé.
5. **Construction de l'ensemble** : Tous les couples (symbole, nouvel état) sont collectés dans un ensemble (`Set`).

2.4.2 Fonction `isAccepted`

Description La méthode `isAccepted` vérifie si un état donné appartient à l'ensemble des états acceptanteurs de l'automate (`dfa.acceptingStates`).

Utilité Cette fonction est utilisée en interne pour déterminer si l'état final atteint par un mot est un état accepté.

Implémentation Elle repose simplement sur l'ensemble des états acceptants `dfa.acceptingStates`

2.4.3 Fonction `accept`

Description La fonction `accept` détermine si un mot (`Word[A]`) conduit l'automate à un état acceptant lorsqu'on démarre depuis l'état initial.

Utilité Cette méthode est essentielle pour tester si un automate reconnaît un mot donné.

Implémentation La fonction utilise un parcours itératif du mot (`word`) à l'aide de `foldLeft`. À chaque étape, elle applique la fonction de transition `dfa.transition` sur le symbole courant. Si une transition n'est pas définie, elle retourne immédiatement `false`. Sinon, elle vérifie si l'état final atteint appartient à l'ensemble des états acceptants (`dfa.acceptingStates`).

Méthodes `solve`, `lazySolve`, et `heuristicSolve`

Utilité

Ces trois méthodes permettent de trouver les mots qui mènent de l'état initial à un état acceptant dans un automate déterministe fini (DFA) tout en évitant les chemins cycliques. Grâce à une File (Immutable Queue), nous stockons les chemins pour garder trace des mots et de leurs chemins. Cette approche nous permet d'avoir plus d'efficacité car nous simulons un bfs (breadth-first-search).

- `solve` : Explore tous les chemins de manière exhaustive et retourne une liste complète de solutions.

- `lazySolve` : Génère les solutions *à la demande* sous forme de `LazyList`, permettant une exploration paresseuse.
- `heuristicSolve` : Explore les chemins en suivant une approche guidée par heuristique[1], en priorisant les chemins avec un coût plus faible.

Implémentation commune

Toutes les méthodes reposent sur une fonction auxiliaire récursive (terminale) qui :

- Suit un ensemble de chemins (stockés dans une File "FIFO"), chaque chemin est représenté par:
  - `currentState` : L'état actuel.
  - `word` : La séquence de symboles menant à l'état actuel.
  - `visited` : Les états déjà visités pour éviter les cycles.
  - `adjacent` : Les transitions possibles depuis l'état actuel.
- Évalue si l'état actuel est acceptant :
  - Si oui, ajoute le mot correspondant à la solution.
- Explore les transitions non visitées depuis l'état actuel.
- Ajoute de nouveaux chemins à explorer en fonction des transitions restantes.

Étapes principales (`solve`)

1. Si aucun chemin n'est disponible (la File est vide), la recherche est terminée et les solutions sont retournées.
2. Pour chaque chemin, si l'état actuel est accepté, le mot correspondant est ajouté aux solutions.
3. Les transitions non visitées sont explorées :
   - Si aucune transition n'est possible, le chemin est abandonné.
   - Sinon, les nouveaux chemins sont ajoutés à la file à explorer.

Particularités des méthodes

- `solve` :
  - Retourne une liste complète (`List[Word[A]]`) contenant tous les mots possibles.
  - Explore de manière exhaustive tous les chemins disponibles.
- `lazySolve` :
  - Retourne les solutions sous forme de `LazyList`, permettant une génération *paresseuse*.
  - Utilise une fonction `LazyList.unfold` pour produire les mots un par un, réduisant les besoins en mémoire.
  - Sa fonction auxiliaire (`lazyHelper`) retourne un mot et met à jour la liste des chemins restants.
- `heuristicSolve` :
  - Intègre une fonction heuristique (`heuristic : S => Double`) pour guider l'exploration.
  - Privilégie les transitions vers des états avec le coût heuristique le plus faible.
  - Utilise une structure similaire à `lazySolve`, mais en priorisant les états prometteurs.

[1] On définit une fonction heuristique $H : Q \to \mathbb{R}^+$ telle que $H(q) = 0 \iff q \in F$ et $0 \leq H(q) \leq \varphi(q)$, l'admissibilité de q, est la longueur du plus petit mot accepté par l'automate au départ de l'état q.

Résultats

- `solve` : Produit immédiatement tous les mots possibles.
- `lazySolve` et `heuristicSolve` : Génèrent des solutions dynamiquement, optimisant les performances en fonction des besoins ou de l'heuristique.

3 Automate des chaînes binaires impaires (`binaire.scala`)

3.1 Description générale

Cet automate reconnaît des chaînes binaires où le nombre de '1' est impair. Les états, transitions et autres composantes sont spécifiquement définis pour répondre à cette contrainte. L'implémentation repose sur le trait `DFA`.

3.2 Choix des implémentations et explications

3.2.1 États (`BinaryState`)

Les états sont représentés par la classe `BinaryState`, un sous-type de `StateDFA`. 2 états sont définis :

- `even` : Représente les configurations où le nombre de '1' dans la chaîne parcourue est pair.
- `odd` : Représente les configurations où le nombre de '1' dans la chaîne parcourue est impair.

Utilité : L'utilisation de deux états est suffisante pour capturer la parité du nombre de '1', rendant l'automate simple et efficace.

3.2.2 Alphabet

L'alphabet est défini comme un ensemble contenant les symboles binaires '0' et '1'.

3.2.3 État initial

L'état initial est défini comme `even`, car une chaîne vide ou nulle contient zéro '1' (et zéro est pair).

3.2.4 États accepteurs

L'ensemble des états accepteurs est constitué uniquement de l'état `odd`, car l'automate doit accepter uniquement les chaînes ayant un nombre impair de '1'.

3.2.5 Fonction de transition

La fonction de transition détermine l'état suivant en fonction de l'état actuel et du symbole lu :

- Depuis l'état `even` :
  - Lire '0' ne modifie pas la parité (rester dans `even`).
  - Lire '1' rend le nombre de '1' impair (passer à `odd`).
- Depuis l'état `odd` :
  - Lire '0' ne modifie pas la parité (rester dans `odd`).
  - Lire '1' rend le nombre de '1' pair (passer à `even`).

3.3 Question

L'utilisation de la fonction solve doit renvoyer tous les mots sans cycles. Combien de mots de ce type doivent être renvoyés ?
Nous avons un seul mot sans cycle "1". En effet, nous avons deux états, et nous commençons par un état paire et avec une seule transition nous passons à l'état impaire qui est un état accepteur.

4 Automate du problème du fermier (`PMLC.scala`)

4.1 Description générale

Cet automate représente le problème classique où un fermier doit traverser une rivière avec une chèvre, un loup et un chou, tout en respectant des contraintes de sécurité. Les classes, états et fonctions définis permettent de modéliser et résoudre ce problème sous forme d'un automate fini déterministe (DFA).

4.2 Définition des composantes principales

4.2.1 `enum Item`

Utilité : L'énumération `Item` représente les différents éléments impliqués dans le problème : le fermier (`Farmer`), le loup (`Wolf`), la chèvre (`Goat`) et le chou (`Gabbage`). La méthode `toChar` permet de convertir chaque élément en un caractère correspondant pour simplifier les transitions.

4.2.2 `case class State[Q]`

Utilité : La classe `State` représente un état du problème, caractérisé par deux ensembles (`left` et `right`) contenant les éléments présents de chaque côté de la rivière.

4.2.3 Fonction `isSink`

Utilité : Cette fonction détermine si un état donné est un "état piège", c'est-à-dire une configuration où les contraintes de sécurité sont violées. Un état est considéré comme un piège si :

- Le loup (`Wolf`) et la chèvre (`Goat`) se trouvent sur le même côté de la rivière sans le fermier (`Farmer`).
- La chèvre (`Goat`) et le chou (`Gabbage`) se trouvent sur le même côté de la rivière sans le fermier (`Farmer`).

Pour vérifier cela, la fonction utilise un test sur les ensembles des deux côtés de la rivière (`left` et `right`). Un conflit est détecté si deux éléments incompatibles (par exemple, le loup et la chèvre) sont présents sur un même côté et que le fermier est absent. Une fois détecté, un état piège est marqué comme invalide et ne peut pas être exploré davantage.
—

4.2.4 Fonction `isValid`

Utilité : Cette fonction vérifie si une transition est valide pour un état donné en fonction du symbole de déplacement. Chaque symbole correspond à une action du fermier, soit seul (`'p'`), soit en transportant un autre élément (`'l', 'm', 'c'`). Une transition est considérée comme valide si :

- L'élément spécifié par le symbole se trouve sur le même côté que le fermier (`Farmer`).
- Le déplacement de cet élément (et du fermier) respecte les contraintes de sécurité.

Pour effectuer cette vérification, la fonction utilise une méthode auxiliaire qui teste si l'élément mentionné par le symbole est bien sur le même côté que le fermier. Si ce n'est pas le cas, la transition est immédiatement rejetée.

4.2.5 Fonction `moveNext`

Utilité : Cette fonction génère un nouvel état en déplaçant le fermier, seul ou avec un autre élément, de l'un des côtés de la rivière vers l'autre. Le fonctionnement se déroule en plusieurs étapes :

- Identifie le symbole correspondant à l'action (par exemple, `'p'` pour le fermier seul, `'l'` pour le fermier avec le loup, etc.).
- Vérifie sur quel côté de la rivière se trouve le fermier et l'élément à transporter.
- Met à jour les ensembles des deux côtés de la rivière en retirant les éléments déplacés du côté initial et en les ajoutant au côté opposé.

Cette fonction repose sur une méthode auxiliaire qui effectue les modifications des ensembles de manière immuable, garantissant que l'état est correctement mis à jour après chaque déplacement. Si le symbole est invalide (par exemple, un élément non présent avec le fermier), l'état reste inchangé.

4.2.6 `case class plmcDFA`

Utilité : Cette classe implémente un DFA spécifique pour le problème PLMC. Elle définit les composantes principales de l'automate :

- Les `states`, qui regroupent toutes les configurations possibles du problème.
- L'`alphabet`, contenant les symboles `'p', 'l', 'm', 'c'` (fermier seul ou fermier transportant un élément).
- L'`initialState`, correspondant à la configuration où tous les éléments sont du côté gauche.
- Les `acceptingStates`, où tous les éléments ont traversé la rivière.
- La `transition`, qui applique les règles de déplacement en vérifiant leur validité.

5 Automate du problème du taquin (`taquin.scala`)

5.1 Description générale

Le problème du taquin consiste à déplacer les tuiles d'une grille jusqu'à atteindre une configuration finale (grille résolue) en respectant les règles de déplacement. L'automate du taquin est modélisé comme un automate fini déterministe (`DFA`), où chaque état représente une configuration de la grille, et chaque transition représente un mouvement valide du vide (tuile `0`).

5.2 Définition des composantes principales

5.2.1 `type Grid`

Utilité : Le type `Grid` représente la grille du taquin sous forme d'une liste d'entiers (`List[Int]`), où chaque entier correspond à une tuile numérotée (ou `0` pour l'espace vide).

5.2.2 `case class TaquinState`

Utilité : Cette classe définit un état du taquin, caractérisé par une configuration particulière de la grille. Elle comprend :

- `grid`: une instance de `Grid` représentant la configuration actuelle.
- `emptyPos`: une position calculée (`lazy val`) identifiant l'index de l'espace vide dans la grille (calculée sur demande pour une question d'efficacité).

5.2.3 `class TaquinDFA`

Utilité : Cette classe modélise l'automate pour résoudre le problème du taquin. Les principales composantes sont les suivantes :

- `initialState`: l'état initial de la grille.
- `states`: une méthode récursive générant tous les états possibles accessibles depuis l'état initial.
- `alphabet`: l'ensemble des mouvements possibles (`'r', 'l', 'u', 'd'`) correspondant à droite, gauche, haut et bas.
- `acceptingStates`: l'état final où la grille est résolue.
- `transition`: une méthode permettant de passer d'un état à un autre en appliquant un symbole (mouvement), tout en vérifiant sa validité.

5.3 Les fonctions principales de l'extension du trait `DFA`

5.3.1 `calculateSize`

Utilité : Cette fonction calcule la taille (côté) de la grille à partir du nombre total de tuiles, en prenant la racine carrée de la longueur de la liste.

5.3.2 `getMoves`

Utilité : Cette fonction génère un dictionnaire associant chaque mouvement (`'r', 'l', 'u', 'd'`) à son décalage respectif dans la liste (`+1`, `-1`, `-size`, `+size`).

5.3.3 `states`

Utilité : Cette méthode explore récursivement tous les états atteignables à partir de l'état initial en appliquant toutes les transitions possibles jusqu'à stabilisation.

5.3.4 `acceptingStates`

Utilité : Cette méthode définit l'état final comme une grille triée dans l'ordre croissant (`[1, 2, ..., n, 0]`), où `0` est à la dernière position.

5.3.5 `transition`

Utilité : Cette méthode effectue une transition depuis un état donné en appliquant un mouvement. Elle :

- Vérifie la validité du mouvement via `isValidMove`.
- Calcule la nouvelle position du vide.
- Retourne une nouvelle grille après avoir échangé les positions (`swap`).

5.3.6 `swap`

Utilité : Cette méthode effectue un échange entre deux positions dans une grille pour générer une nouvelle configuration.

5.3.7 `isValidMove`

Utilité : Cette méthode détermine si un mouvement est valide en fonction de la position du vide (`emptyPos`) et des limites de la grille (rangée et colonne).

5.4 `h1` et `h2` : Fonctions heuristiques

Les fonctions heuristiques permettent d'évaluer un état intermédiaire de la grille en estimant sa proximité avec l'état final. Elles sont utilisées pour guider les algorithmes de recherche, en fournissant une estimation du *coût* restant pour atteindre la solution.

5.4.1 `h1`: Comptage des tuiles mal placées

Description : La fonction `h1` compte le nombre de tuiles qui ne se trouvent pas à leur position finale prévue.

Implémentation :

- On compare chaque tuile de la grille courante (`state.grid`) avec la configuration cible (`goalGrid`).
- Retourne le nombre de tuiles différentes.

5.4.2 `h2`: Distance de Manhattan

Description : La fonction `h2` calcule la somme des distances de Manhattan de chaque tuile par rapport à sa position finale.

Implémentation :

- Pour chaque tuile, on identifie sa position actuelle (`currentIndex`) et sa position cible (`goalIndex`).
- On calcule ensuite la distance de Manhattan, définie comme la somme des écarts absolus en ligne et en colonne :

$$\texttt{distance} = |\texttt{currentRow} - \texttt{goalRow}| + |\texttt{currentCol} - \texttt{goalCol}|$$

- Additionne les distances pour toutes les tuiles.

Comparaison entre `h1` et `h2` :

- `h1` pourrait conduire à une recherche moins efficace, car elle peut sous-estimer la difficulté du problème. En effet, elle ne prend en compte que le nombre de tuiles mal placées. Elle ne donne aucune information sur l'estimation de la distance réelle entre l'état actuel et l'état final.
- `h2` guide mieux l'algorithme de recherche, ce qui réduit le nombre de nœuds explorés et accélère la résolution du puzzle, car elle mesure la distance totale que chaque tuile doit parcourir pour atteindre sa position correcte.

  En conclusion, `h1` est plus simple à implémenter mais mois efficace que `h2`. `h2` donne plus d'informations pour l'état actuel et permet d'arriver plus rapidement sur une solution.