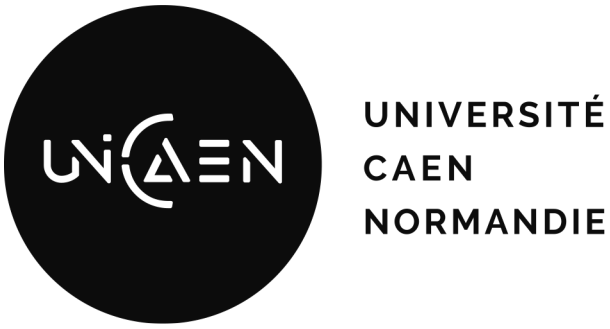
Rapport fil rouge

March 30, 2025



1 Introduction

Ce rapport présente le projet fil rouge réalisé en Java, qui consiste à implémenter le monde des blocs, un exemple couramment utilisé en intelligence artificielle, notamment pour illustrer et tester des algorithmes de planification. Au cours de ce projet, nous avons suivi les différentes étapes développées tout au long du semestre, en commençant par la modélisation, puis la planification, la programmation par contraintes, et enfin l'extraction de connaissances.

2 Structure du monde des blocks

Dans ce fil rouge, on propose d'utiliser trois ensembles de variables pour représenter les configurations d'un monde de n blocs et m piles : — pour chaque bloc b, une variable onb, prenant pour valeur soit un autre bloc b (signifiant que b est posé sur b), soit une pile p (signifiant que b est sur la table dans la pile p) ; la variable onb pourra par exemple avoir pour domaine l'ensemble d'entiers p, . . . , 1, 0, . . . , n 1 b, les entiers strictements négatifs codant les p piles, et les entiers positifs ou nuls, les blocs (on exclut b du domaine car un bloc ne peut pas être sur lui-même) ; — pour chaque bloc b, une variable booléenne fixedb, prenant la valeur true lorsque le bloc est indéplaçable ; — pour chaque pile p, une variable booléenne freep, prenant la valeur true lorsque la pile est libre. Pour ce qui est des contraintes, on se restreint à des contraintes binaires ; les contraintes suivantes

3 Les etapes du projet

3.1 Modélisation

Cette partie consiste essentiellement à créer une modélisation du monde des blocs, c'est-à-dire définir les différentes variables (`onb`, `fixedb`, `freep`) et les contraintes que ce monde doit respecter. Pour cela, nous avons utilisé les différentes classes du package `modelling`, développées lors du TP1.

Variables Pour modéliser les variables nécessaires, nous avons utilisé deux classes principales :

- `Variable` : utilisée pour générer les variables de type `onb`.
- `BooleanVariable` : utilisée pour générer les variables de type `fixedb` et `freep`.

Pour cela nous avons implémenté la classe `BlocksWorld`, qui permet de générer les différentes variables du monde des blocs. Grâce aux classes mentionnées, les variables sont définies avec des domaines adaptés aux configurations possibles du monde des blocs.

Contraintes La définition des variables ne suffit pas pour garantir qu'une configuration est valide. Il est nécessaire de définir des contraintes pour s'assurer que les règles du monde des blocs sont respectées. Pour cela, nous avons fait appel à plusieurs classes du package `modelling`.

Voici les classes utilisées pour générer et appliquer ces contraintes :

- `BlocksWorldConstraints` : génère les contraintes de base nécessaires pour assurer la validité de la configuration.
- `RegularBlocksWorld` : ajoute des contraintes pour garantir que la configuration est régulière.
- `IncreasingConstraints` : génère des contraintes qui garantissent que la configuration respecte un ordre croissant.

En combinant les variables et ces différentes contraintes, nous avons pu modéliser un monde des blocs cohérent et conforme aux règles définies.

3.2 Planification

Cette partie vise à créer les différentes **actions possibles** dans notre configuration et à générer des **plans** grâce à divers planificateurs. Ces plans permettent de passer d'un **état initial** de notre monde des blocs à un **état but** défini au préalable. Pour cela, nous avons utilisé les classes du package `planning`, réalisées dans le cadre du TP2.

3.2.1 Génération des actions

Dans notre monde des blocs, chaque action correspond à un déplacement possible (par exemple, déplacer un bloc d'une pile vers une autre ou sur un autre bloc). Ces actions sont modélisées comme des instances de la classe `BasicAction` du package `planning`.

La classe principale pour générer ces actions est :

- `BlocksWorldAction` : Cette classe génère toutes les actions de déplacement en créant des instances de `BasicAction`. Elle définit pour chaque action :

- Les **préconditions** nécessaires pour que l'action soit applicable (par exemple, le bloc à déplacer doit être libre).
- Les **effets** de l'action, qui modifient l'état actuel du monde des blocs.

3.2.2 Génération des plans

Pour trouver un plan permettant de passer de l'état initial à l'état but, nous avons utilisé les différents planificateurs étudiés dans le TP2. Ces planificateurs permettent d'explorer l'espace des états et de générer une séquence d'actions pour atteindre l'objectif.

Parmi les planificateurs utilisés, on retrouve :

- **DFS (Depth-First Search)**
- **BFS (Breadth-First Search)**
- **Dijkstra**
- **A**

3.2.3 Heuristiques pour A

Lors de l'utilisation du planificateur A, deux heuristiques admissibles ont été développées pour estimer le coût restant afin d'atteindre l'état but :

- `MisplacedBlocksHeuristic` : Cette heuristique calcule un coût basé sur le nombre de blocs mal placés dans la configuration actuelle par rapport à l'état but. Plus le nombre de blocs mal placés est élevé, plus le coût estimé est important.

- `DistanceHeuristic` : Cette heuristique calcule la distance entre la position actuelle des blocs et leur position dans l'état but. Pour chaque bloc, si sa position dans l'état actuel est différente de sa position dans l'état but, la distance entre ces deux positions est calculée. La distance totale est la somme des distances de chaque bloc.

Ces outils nous permettent de comparer les performances des différents planificateurs en termes de nombre d'actions générées, de nœuds explorés et de temps d'exécution.

3.3 Problèmes de satisfaction de contraintes

Dans cette partie, nous utilisons les différents solveurs du package `cp`, réalisés dans le cadre du TP3 (`BacktrackSolver`, `MACSolver`, `HeuristicMACSolver`), pour trouver une configuration qui satisfait les contraintes données (configuration régulière , configuration croissante et configuration regulier et croissante à la fois).

Pour cela, nous avons créé une classe exécutable qui fait appel aux différents solveurs afin de chercher une configuration respectant les contraintes spécifiées.

- `AppSolver` : Cette classe exécutable permet de trouver une configuration qui satisfait les contraintes en utilisant les différents solveurs mentionnés.

3.4 Extraction des connaissances

Dans cette partie, nous nous intéressons à des bases de données représentant les états du monde des blocs. Pour cela, nous utilisons les différentes classes du package `datamining`, étudié lors du TP4, afin d'extraire des motifs et des règles d'association.

- `BooleanVariablesFactory` : La classe `BooleanVariablesFactory` est instanciée avec un nombre de blocs et de piles. Elle fournit deux fonctionnalités principales :
  - Générer l'ensemble des variables booléennes ($On_{b,b}$, $OnTable_{b,p}$, $Fixed_b$ et $Free_p$)correspondant aux paramètres `nbBlocks` et `nbPiles`.
  - Traduire un état donné (sous forme d'une liste de listes d'entiers) en un ensemble d'instances de variables booléennes (`Set<BooleanVariable>`).

Les variables générées sont ensuite utilisées pour extraire les différentes connaissances de la base de données que nous avons déjà créée.

Pour illustrer le fonctionnement, nous avons également créé une classe exécutable :

- `AppDataMining` : Dans cette classe, une base de données transactionnelle d'états du monde des blocs est créée, et des motifs fréquents ainsi que des règles d'association sont extraits à.
  La classe `AppDataMining` exécute les étapes suivantes :
  - Création d'une base de données (`BooleanDatabase`) contenant **n états** générés aléatoirement.

- Extraction de motifs fréquents avec une **fréquence minimale**.
- Extraction de règles d'association avec une **fréquence minimale** et une **confiance minimale**.

4 Mode d'emploi et les différentes classes exécutables

4.1 Modélisation

Pour la partie planification, nous avons développé deux classes exécutables permettant de modéliser et de valider les contraintes du problème du monde des blocs :

- **AppBasicConstraints** : Cette classe teste si les trois contraintes de base sont satisfaites par une configuration donnée.
- **AppConstraints** : cette classe permet de tester si une configuration donnée respecte l'ensemble des contraintes avancées (regularite et croissante).

4.2 Planification

Pour la partie planification, nous avons une classe exécutable :

- `AppAction` : Cette classe permet de créer un état initial et un état but, puis de lancer les différents solveurs sur l'exemple donné.

Après l'exécution, il vous sera demandé de choisir le plan à visualiser selon l'algorithme utilisé. Pour cela, entrez un numéro entre 1 et 4 correspondant aux algorithmes suivants :

1. DFS
2. BFS
3. Dijkstra
4. A*

4.3 Problèmes de satisfaction de contraintes

Dans cette partie, nous avons développé une classe exécutable pour résoudre les problèmes de satisfaction de contraintes :

- **AppSolver** : Cette classe exécutable permet de trouver une configuration qui satisfait les contraintes définies, en utilisant différents solveurs. Les solveurs appliqués offrent des approches variées pour rechercher des solutions dans l'espace des configurations valides.

4.4 Problèmes de satisfaction de contraintes

Dans cette partie, nous avons développé une classe exécutable pour résoudre les problèmes de satisfaction de contraintes :

- **AppSolver** : Cette classe exécutable permet de trouver une configuration qui satisfait les contraintes définies, en utilisant différents solveurs. Les solveurs appliqués offrent des approches variées pour rechercher des solutions dans l'espace des configurations valides.

4.5 Extraction des connaissances

Dans cette partie, nous avons développé une classe exécutable pour l'extraction des connaissances :

- **AppDataMining** : Cette classe exécutable permet de réaliser des tâches d'extraction de connaissances, telles que la découverte de motifs fréquents et de règles d'association.

4.6 Commandes pour la compilation et l'exécution

Toutes les commandes suivantes doivent être exécutées depuis le répertoire 'src'.
Avant de compiler le package `blocksworld`, il est nécessaire de compiler d'abord les autres packages dont il dépend, à savoir `modelling`, `planning`, `cp` et `datamining`. Pour cela, il convient d'exécuter la commande suivante à partir du répertoire `src` :

```
javac -d ../build modelling/*.java planning/*.java cp/*.java datamining/*.java
```

Pour compiler les fichiers Java avec les bibliothèques externes (pour le package blocksworld) :

```
javac -d ../build -cp .:../lib/bwgenerator.jar:../lib/blocksworld.jar
./blocksworld/*.java
```

Pour exécuter une classe exécutable (remplacez 'NomClasseExecutable' par le nom de votre classe executable) :

```
java -cp .:../lib/bwgenerator.jar:../lib/blocksworld.jar:../build
blocksworld.NomClasseExecutable
```

Note : Il est recommandé de copier les commandes directement depuis le fichier ***readme.txt*** *pour éviter toute erreur de saisie et garantir un copier-coller correct dans le terminal.*