Projet **Blocksworld « Monde des Blocs »**

Réalisé par :

ANNOU Rayane 22312297 GROUPE : 2A

Emam Mohamed Elmamy 22019076 GROUPE 2A

# Table des matières

# 1 Introduction

La résolution de problèmes complexes est au cœur de nombreux défis en intelligence artificielle. Parmi ces problèmes, celui du monde des blocs est emblématique, car il combine des éléments de planification, de gestion de contraintes, et d'extraction de connaissances. Dans ce projet, nous avons exploré et implémenté différentes approches pour modéliser et résoudre des problèmes dans le contexte du monde des blocs, en nous appuyant sur des concepts tels que les problèmes de satisfaction de contraintes (CSP), les algorithmes de planification (comme BFS, DFS, et A*), ainsi que des heuristiques pour optimiser la recherche de solutions.

L'objectif principal de ce travail est de démontrer comment différentes techniques d'IA peuvent être appliquées pour résoudre des problèmes de manipulation de blocs dans un environnement simulé. Nous avons également veillé à illustrer visuellement les solutions générées à travers une librairie d'affichage spécifique.

Ce document décrit en détail les choix d'implémentation effectués, le rôle des différentes classes développées, et les démonstrations réalisées. Nous commencerons par une présentation des variables et des contraintes définies, suivie d'une exploration des algorithmes de planification, des approches pour résoudre des CSP, des méthodes pour extraire des connaissances, et enfin une description de la librairie utilisée pour simuler et afficher les résultats.

# 2 Variables et contraintes

## 2.1 Variables

Dans cette section, nous allons décrire la gestion des variables, qui représente une composante essentielle dans la modélisation du problème du monde des blocs. La classe **MondeDesBlocs** a été conçue pour gérer les différentes variables nécessaires pour décrire l'état du système. Ces variables permettent de définir la position des blocs, les contraintes sur les piles, et les états de fixation des blocs.

La classe MondeDesBlocs est responsable de l'initialisation et de la gestion des variables nécessaires à la représentation du problème du monde des blocs. Elle est conçue pour manipuler trois types principaux de variables : les variables Onb, les variables Fixedb, et les variables Freep. Ces variables sont stockées dans des listes respectivement appelées variablesOnb, variablesFixedb, et variablesFreep.

Le constructeur **MondeDesBlocs(int nombreDeBlocs, int nombreDePiles)** est chargé de l'initialisation des variables. À la création d'une instance de cette classe, le constructeur procède à la création et à l'initialisation des variables **Onb**, **Fixedb**, et **Freep**.

Création des variables **Onb :**
Pour chaque bloc, une variable **Onb** est créée. Cette variable a un domaine qui correspond à l'ensemble des autres blocs et des piles. Par exemple, pour le bloc On0, le domaine contient les

indices des autres blocs ainsi que les piles disponibles avec le signe ( - ). Il est essentiel de s'assurer que chaque bloc ne peut pas être placé sur lui-même, d'où l'ajout de la condition if (j != i) dans la création du domaine de chaque variable Onb.

Création des variables **Fixedb :**
Pour chaque bloc, une variable booléenne **Fixedb** est créée. Ces variables représentent si un bloc est fixe (ne peut pas être déplacé) ou libre (peut être déplacé). Par exemple, une variable **Fixed0** peut être vraie si le bloc 0 est fixé, ou fausse s'il est libre.

Création des variables **Freep :**
Pour chaque pile, une variable **Freep** est créée. Ces variables indiquent si une pile est libre pour recevoir un bloc. Une pile est libre si aucune variable **Onb** n'indique qu'un bloc y est posé. Par exemple, **Free1** peut être vraie si la pile 1 est libre.

# 2.2 Contraintes de Base

Dans cette section, nous définissons les contraintes qui régissent les relations entre les variables du problème des blocs. Ces contraintes sont essentielles pour modéliser les interactions et garantir la validité des configurations du monde des blocs, notamment en ce qui concerne la position des blocs et leur placement sur les piles.

La classe **MondeDesBlocsContraintes** étend la classe **MondeDesBlocs** pour ajouter des règles et des relations supplémentaires sous forme de **contraintes** entre les variables. Ces contraintes sont essentielles pour définir un espace de recherche cohérent où les blocs doivent être placés en respectant certaines règles logiques.

Le constructeur **MondeDesBlocsContraintes** initialise la liste des contraintes et des piles et appelle la méthode **definirContraintes** pour définir un ensemble de règles logiques. Les principales contraintes définies sont :

**Contrainte 1** : Cette contrainte stipule que deux blocs ne peuvent pas être placés dans la même position. Concrètement, pour chaque paire de variables onb (position d'un bloc), une contrainte de différence est ajoutée. Cela empêche deux blocs d'occuper la même position.

```
contraintes.add(new DifferenceConstraint(variablesOnb.get(i), variablesOnb.get(j)));
```

**Contrainte 2** : Si un bloc est placé sur un autre bloc (c'est-à-dire que onb = b'), alors le bloc sur lequel le premier est placé doit être fixé (fixedb' = true). Cela garantit que si un bloc est sur un autre, ce dernier doit être immobile (fixe).

```
Constraint implicationConstraint = new Implication(variablesOnb.get(i), domaineB,
variablesFixedb.get(j), domaineFixedB);

contraintes.add(implicationConstraint);
```

**Contrainte 3** : Si un bloc est placé sur une pile (représentée par une valeur négative pour onb), alors cette pile doit être marquée comme occupée, ce qui est assuré par une contrainte d'implication entre onb et freep.

Constraint implicationPileLibre = new Implication(variablesOnb.get(i), domainep, variablesFreep.get(p), domainefree);

contraintes.add(implicationPileLibre);

**getContraintes()** : Cette méthode retourne la liste de toutes les contraintes définies dans le monde des blocs. Elle permet d'accéder à l'ensemble des règles logiques appliquées au problème.

# 2.3 Contraintes Configuration Régulière

Les contraintes régulières définissent une relation répétitive ou symétrique entre les blocs dans la configuration. Ces contraintes permettent de structurer les blocs dans un motif logique, où des écarts réguliers entre leurs indices sont maintenus.

La classe **ContraintesConfigurationRegulière** étend la classe **MondeDesBlocsContraintes** pour introduire des règles spécifiques qui imposent une régularité dans les positions des blocs. Par exemple, elle garantit que si deux blocs ont une certaine différence entre leurs indices, un troisième bloc peut être positionné pour maintenir cette différence.

Principales Caractéristiques :

**Attributs** :

**contraintesRegulieres** : Ensemble des contraintes définies pour maintenir une configuration régulière des blocs.

**Méthodes :**

**definirContraintesRegulieres :** Cette méthode génère des contraintes d'implication entre les blocs, en s'assurant que la différence entre leurs indices est maintenue à travers toute la configuration.

**Exemple :** Si le bloc 1 et le bloc 2 ont un écart d'indice de 2, un troisième bloc pourra être positionné pour conserver cet écart avec le bloc 2.

**Étapes de Définition :**

- Pour chaque paire de blocs (variable1 et variable2), on calcule l'écart entre leurs indices.
- Ensuite, on cherche un troisième bloc qui pourrait maintenir cet écart avec la même logique.
- Une contrainte d'implication est créée pour garantir cette régularité.

Implication imp = new Implication(variable1, Set.of(Integer.parseInt(variable2.getName().substring(2))), variable2, domain);

contraintesRegulieres.add(imp);

**Méthode getContraintesRegulieres :** Cette méthode retourne l'ensemble des contraintes régulières définies.

# 2.4 Contraintes Configuration Croissante

Les contraintes de configuration croissante imposent un ordre logique entre les blocs, où un bloc ne peut être placé que sur un bloc d'indice inférieur ou sur une pile libre. Ces règles garantissent une structure hiérarchique, facilitant l'organisation des blocs.

La classe **ContraintesConfigurationCroissante** hérite également de **MondeDesBlocsContraintes** pour ajouter des règles spécifiques visant à organiser les blocs dans un ordre croissant.

**Principales Caractéristiques :**

Attributs :

**contraintesCroissantes** : Ensemble des contraintes définies pour garantir une configuration croissante des blocs.

**Méthodes :**

**Méthode definirContraintesCroissantes() :** Cette méthode crée des contraintes pour chaque bloc afin de restreindre les positions possibles selon un ordre croissant.

- **Exemple :** Un bloc bloc1 d'indice 3 peut être placé sur un bloc bloc2 d'indice 2, mais pas l'inverse.

**Étapes de Définition :**

**Contraintes unaires :** Pour chaque bloc, on définit un domaine qui inclut les blocs d'indice inférieur ou les piles libres.

UnaryConstraint unaryConstraint = new UnaryConstraint(bloc1, domaineBloc1);

contraintesCroissantes.add(unaryConstraint);

**Contraintes d'implication :** Si un bloc est placé sur un autre, une contrainte est créée pour assurer que le bloc inférieur respecte également un ordre croissant.

Implication implicationConstraint = new Implication(bloc1, Set.of(numBloc2), bloc2, domaineImplication);

contraintesCroissantes.add(implicationConstraint);

**Méthode getContraintesCroissantes() :** Cette méthode retourne toutes les contraintes de configuration croissante.

# 2.5 Classe exécutable App

Pour compiler les classes du package blocksworld :

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

Et pour executer le Code de la classe App :

java -cp ../build blocksworld.App

**Explication de la classe :**

**Étape 1 : Création du Monde des Blocs**

ContraintesConfigurationReguliere bw = new ContraintesConfigurationReguliere(11, 6);

Un monde avec **11 blocs** et **6 piles** est créé.

**Étape 2 : Initialisation des Variables `Onb`, `Fixes`, et `Freep`**

```java
Map<Variable, Object> assignment = new LinkedHashMap<Variable, Object>();

// Initialisation des variables Onb
List<Variable> onbList = new ArrayList<>(bw.getVariablesOnb());
for (int i = 0; i < onbList.size(); i++) {
    System.out.println("get(" + i + ") : " + onbList.get(i));
}

assignment.put(onbList.get(0), -2);
assignment.put(onbList.get(1), -1);
assignment.put(onbList.get(2), 0);
assignment.put(onbList.get(3), 1);
assignment.put(onbList.get(4), 2);
assignment.put(onbList.get(5), -4);
assignment.put(onbList.get(6), 4);
assignment.put(onbList.get(7), 5);
assignment.put(onbList.get(8), 6);
assignment.put(onbList.get(9), -3);
assignment.put(onbList.get(10), 9);
```

Figure1 : « Initialisation des Variables Onb »

```java
// Initialisation des variables Fixes
List<BooleanVariable> fixedList = new ArrayList<>(bw.getVariablesFixes());
for (int i = 0; i < fixedList.size(); i++) {
    System.out.println("get(" + i + ") : " + fixedList.get(i));
}

assignment.put(fixedList.get(0), true);
assignment.put(fixedList.get(1), true);
assignment.put(fixedList.get(2), true);
assignment.put(fixedList.get(3), false);
assignment.put(fixedList.get(4), true);
assignment.put(fixedList.get(5), false);
assignment.put(fixedList.get(6), true);
assignment.put(fixedList.get(7), false);
assignment.put(fixedList.get(8), false);
assignment.put(fixedList.get(9), true);
assignment.put(fixedList.get(10), false);
```

Figure 2 : « Initialisation des variables fixedb »

```java
// Initialisation des variables Freep (Piles libres)
List<BooleanVariable> freepList = new ArrayList<>(bw.getPilesLibres());
for (int i = 0; i < freepList.size(); i++) {
    System.out.println("get(" + i + ") : " + freepList.get(i));
}

// Affectation des valeurs aux variables Freep
if (freepList.size() == 6) {  // Assurez-vous qu'il y a bien 6 éléments

    assignment.put(freepList.get(0), false);
    assignment.put(freepList.get(1), false);
    assignment.put(freepList.get(2), false);
    assignment.put(freepList.get(3), false);
    assignment.put(freepList.get(4), true);
    assignment.put(freepList.get(5), true);
} else {
    System.out.println("Erreur : Nombre de piles libres inattendu");
}
```

Figure 3 : « Initialisation des variables freep »

**Étape 3 : Vérification des Contraintes Régulières**

La méthode `definirContraintesRegulieres()` génère un ensemble de contraintes basées sur des motifs réguliers.

**Vérification des Contraintes Régulières** :

- Le programme vérifie si l'assignation satisfait toutes les contraintes régulières.
- Si une contrainte échoue, elle est affichée et le résultat devient "non régulière".

Dans cet **exemple**, toutes les contraintes régulières sont satisfaites, et le **résultat** est :

**La configuration est régulière.**

**Étape 4 : Vérification des Contraintes Croissantes**

Un objet de type ContraintesConfigurationCroissante est créé pour générer les contraintes croissantes.

**Vérification des Contraintes Croissantes** :

- Le programme vérifie si l'assignation satisfait toutes les contraintes croissantes.
- Si une contrainte échoue, elle est affichée et le résultat devient "non croissante".

Dans cet **exemple**, toutes les contraintes croissantes sont satisfaites, et le **résultat** est :

**La configuration est croissante.**

# 3 Planification

## 3.1 Actions Monde des blocs

La classe ActionsMondeDesBlocs étend la classe MondeDesBlocs pour introduire un ensemble d'**actions possibles** dans le contexte du monde des blocs. Elle représente toutes les actions qui peuvent être effectuées pour déplacer les blocs entre différentes positions, tout en respectant les conditions spécifiques associées à chaque action. Voici une vue d'ensemble de cette classe et de son rôle dans cette partie.

Donc son **Objectif principal :** Générer toutes les actions possibles dans un monde des blocs donné, défini par un certain nombre de blocs et de piles.

**Actions générées :**

1. Déplacement d'un bloc depuis un bloc source vers un autre bloc.
2. Déplacement d'un bloc depuis un bloc source vers une pile vide.
3. Déplacement d'un bloc depuis une pile vers un bloc cible.
4. Déplacement d'un bloc d'une pile vers une autre pile vide

Ces actions représentent les moyens de modifier la configuration du monde des blocs pour résoudre un problème ou atteindre un objectif.

**Structure de la Classe**

1. **Attributs :**

- actions: Un ensemble (Set) qui stocke toutes les actions possibles.

2. **Constructeur :**
   - Prend en entrée le nombre de blocs et de piles, initialise le monde des blocs, puis appelle la méthode **genererActions**() pour créer toutes les actions possibles.
3. **Méthode genererActions() :**
   - Génère toutes les actions en fonction des 4 types décrits ci-dessus. Pour chaque type, des boucles imbriquées sont utilisées pour examiner toutes les combinaisons possibles de blocs et de piles.
4. **Méthodes de création des actions :** Chaque méthode génère un type d'action spécifique avec des conditions préalables (préconditions) et des effets associés :

   **Préconditions** : Elles définissent les conditions qui doivent être vraies pour que l'action puisse être exécutée. Par exemple :

   - Un bloc doit être sur un autre bloc ou une pile.
   - Une pile doit être libre si un bloc doit y être placé.

   **Effets** : Ils définissent les changements dans l'état du monde après l'exécution de l'action. Par exemple :

   - Mettre à jour la position d'un bloc.
   - Libérer ou occuper un bloc ou une pile.

# 3.2 Classe exécutable App2

Commande pour compiler toutes les classes du package blocksworld :

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

Commande pour exécuter App2 :

java -cp ../build/.:../lib/blocksworld.jar blocksworld.App2

App2 est une application simulant le **monde des blocs**, un problème classique d'intelligence artificielle et de planification. L'objectif est de manipuler des blocs (empilés sur des piles) pour atteindre une configuration de but, à partir d'une configuration initiale. L'application explore plusieurs algorithmes de recherche pour résoudre le problème et visualise les solutions.

Une explication générale de App2 :

1. **Création du Monde des Blocs**

- Le monde est modélisé avec un nombre donné de blocs (**nbBolc**) et de piles (**nbPile**). Ici, 4 blocs et 3 piles.
- La classe ActionsMondeDesBlocs gère les variables et actions possibles :
  - **Variables** : Position des blocs (quel bloc est sur quel autre bloc ou pile) et informations sur les piles libres.
  - **Actions** : Mouvements possibles comme déplacer un bloc d'une pile à une autre ou sur un autre bloc.

2. **Définition des États**

- **État initial (etatInitiale)** :
  - Spécifie la position initiale des blocs (par exemple, bloc 0 est sur une pile, bloc 1 est sur une autre pile, etc.).
  - Indique si un bloc est "fixe" ou non, et quelles piles sont libres.
- **État de but (etatGoal)** :
  - Spécifie où les blocs doivent se trouver à la fin (par exemple, bloc 1 doit être sur bloc 3, etc.).
  - Utilisé pour évaluer si une solution satisfait le problème.

3. **Heuristiques dans A*** :

- **Blocs mal placés** :
  - Compte combien de blocs ne sont pas dans leur position de but.
- **Distance de Manhattan** :
  - Somme des distances entre les positions actuelles des blocs et leurs positions de but (en termes de "déplacement vertical" ou "horizontal").

Ces heuristiques aident A* à prioriser les chemins les plus prometteurs.

4. **Planification (Résolution du Problème)**

- Plusieurs algorithmes de recherche sont utilisés pour trouver un plan d'actions permettant de passer de l'état initial à l'état de but :
  - **Recherche BFS (Breadth-First Search)** : Exploration en largeur.
  - **Recherche DFS (Depth-First Search)** : Exploration en profondeur.
  - **Dijkstra** : Recherche de chemin le plus court avec un coût uniforme.
  - **A*** :
    - Avec heuristique 1 : Nombre de blocs mal placés.
    - Avec heuristique 2 : Distance de Manhattan entre les positions actuelles et les positions de but.

Ces algorithmes génèrent un plan (une séquence d'actions) pour atteindre l'objectif.

5. **Résultats et Comparaisons**

- Pour chaque algorithme :
  - Affichage du plan trouvé (séquence d'actions).
  - Nombre de nœuds explorés pendant la recherche.
  - Temps de calcul nécessaire.

6. **Simulation et Visualisation**

- **Visualisation graphique** :
  - Une interface graphique (BWIntegerGUI) affiche l'état des piles et des blocs.
  - Le plan trouvé par chaque algorithme est simulé action par action, avec des pauses, pour montrer le déplacement des blocs.
- **Méthode displaySolution** :
  - Gère la simulation graphique du plan trouvé par un solveur donné (par exemple, BFS ou A*).

- Met à jour l'état des blocs après chaque action et reflète ce changement dans l'interface.

# 4 Problèmes de satisfaction de contraintes

## 4.1 Classe exécutable App3

Pour compiler les classes du package blocksworld :

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

Pour exécuter la classe App3 :

```
java -cp ../build/.:../lib/blocksworld.jar blocksworld.App3
```

App3 est une classe exécutable Java pour résoudre un problème de configuration de blocs, en utilisant plusieurs techniques de résolution de contraintes, notamment le backtracking, la propagation des contraintes (MAC), et des solveurs heuristiques (HeuristicMACSolver). Le problème semble impliquer des configurations de piles de blocs avec différentes contraintes, comme la régularité (contraintes de configuration régulière), la croissance (contraintes de configuration croissante), et la combinaison des deux.

**Structure générale du code**

Le code est divisé en plusieurs parties, chacune correspondant à un type spécifique de configuration et à l'application de différentes méthodes de résolution de contraintes.

***Création des contraintes et des mondes des blocs*** *:*

- Dans chaque partie du code, il existe une création d'un "monde" des blocs, représenté par des instances de classes comme **ContraintesConfigurationReguliere** et **ContraintesConfigurationCroissante**. Ces classes vont définir des contraintes spécifiques sur la manière dont les blocs doivent être organisés.
- **Exemple de contraintes régulières** : Ce sont des contraintes qui exigent une disposition particulière des blocs.
- **Exemple de contraintes croissantes** : Ces contraintes imposent que les blocs doivent être disposés dans un ordre croissant.

***Définition des contraintes et des variables*** *:*

- Pour chaque type de configuration, le code définit des **variables** représentant les différentes parties de la configuration des blocs.
- Ensuite, des **contraintes** sont définies pour chaque monde de blocs, et elles sont regroupées (par exemple, **contraintesDeBase** et **contraintesRegulieres** ou **contraintesCroissantes** ).

***Combinaison des contraintes régulières et croissantes** :*

- Dans une autre partie du code, des **contraintes régulières** et **croissantes** sont combinées. Cela permet de créer des configurations plus complexes où les blocs doivent respecter à la fois les règles de régularité et les règles de croissance.
- Une fois les contraintes combinées, le processus de résolution est à nouveau exécuté à l'aide des solveurs (**BacktrackSolver**, **MACSolver**, **HeuristicMACSolver**).

***Solvers de contraintes** :*

- Le programme utilise plusieurs solveurs pour résoudre le problème sous différentes configurations. Voici les principaux solveurs utilisés :
    - **BacktrackSolver** : Un solveur de backtracking standard qui explore toutes les solutions possibles de manière récursive. Cela permet de trouver une solution en parcourant les configurations possibles tout en respectant les contraintes.
    - **MACSolver** : Un solveur basé sur la propagation des contraintes, qui réduit les domaines des variables pour éliminer des options incompatibles, augmentant ainsi l'efficacité du processus de recherche.
    - **HeuristicMACSolver** : Un solveur qui combine la propagation des contraintes avec des heuristiques pour choisir la meilleure variable et la meilleure valeur à explorer. Des heuristiques sont utilisées pour optimiser l'ordre de traitement des variables et des valeurs, améliorant ainsi la performance du solveur.

Chaque solver est mesuré en termes de **temps de calcul**, et le programme affiche le temps nécessaire pour trouver une solution à l'aide de chaque solver.

***Mesure du temps d'exécution** :*

- Chaque fois qu'un solver est utilisé, le code mesure le temps qu'il faut pour trouver une solution en utilisant System.currentTimeMillis(). Le temps d'exécution est ensuite affiché à l'écran.

***Vérification des solutions** :*

- Après que chaque solver ait trouvé une solution, le programme vérifie si la solution satisfait toutes les contraintes définies. Si une contrainte est violée, le programme signale un problème. Sinon, il confirme que la configuration est valide.

**Affichage des résultats de la solution :**

- Après l'exécution du solveur, le programme affiche non seulement le temps d'exécution, mais également la configuration finale des blocs (c'est-à-dire la disposition des blocs sur les piles) si une solution est trouvée. Cela permet de visualiser la solution finale et de vérifier si elle respecte les contraintes définies.

# 5 Extraction de connaissances

# 5.1 Gestionnaire Variables

La classe **GestionnaireVariables** gère les **variables booléennes** qui permettent de décrire l'état du monde des blocs et qui seront utilisées dans l'extraction des connaissances. Elle prend en entrée des informations de base telles que le nombre de blocs et de piles, puis crée et structure des variables représentant les différentes relations possibles dans cet environnement.

Le **constructeur** de la classe initialise l'objet en prenant en entrée le **nombre de blocs** et le **nombre de piles**. Il crée un monde des blocs, ainsi que des listes pour gérer les variables booléennes qui décrivent la configuration de ce monde.

- **Création des variables onB,B' :** Les variables de type **onB,B'** sont générées pour chaque **pair de blocs distincts**. Cela représente une relation entre les blocs : si un bloc **b** est posé sur un bloc **b'**.
- **Création des variables on-tableB,P :** Ces variables sont générées pour chaque bloc et chaque pile, et indiquent si un bloc **B** est posé sur la table dans la pile **P**.

**Méthodes** :

***getToutesLesVariables****() :*

Cette méthode permet de récupérer **toutes les variables** qui décrivent l'état du monde des blocs. Cela inclut :

- **Les variables liées aux blocs fixes** : Cela permet de savoir quels blocs sont fixes, c'est-à-dire ne peuvent pas être déplacés ou modifiés dans l'état donné.
- **Les variables de piles libres** : Cela permet de savoir quelles piles sont vides ou disponibles pour ajouter des blocs.
- **Les relations entre blocs** : Ces relations sont décrites par les variables onB,B', qui nous indiquent si un bloc **B** est posé sur un autre bloc **B'**.
- **Les relations entre blocs et piles** : Ces relations sont décrites par les variables on-tableB,P, qui nous indiquent si un bloc **B** est posé sur la table ou dans une pile spécifique **P**.

**getInstance() :**

fait bien plus que simplement vérifier l'état des blocs. Elle **crée des variables booléennes** qui représentent l'état actuel du monde des blocs sous plusieurs aspects :

- **Bloc fixé** : Si un bloc est fixé (c'est-à-dire pas empilé sur un autre bloc), la méthode crée une variable booléenne pour ce bloc.
- **Pile libre** : Si une pile est vide, la méthode crée une variable booléenne pour cette pile.
- **Relation entre blocs** : Pour chaque paire de blocs, si un bloc est posé sur un autre, une variable booléenne est créée pour cette relation.
- **Relation entre blocs et la table/pile** : La méthode crée des variables pour savoir si un bloc est posé sur la table ou dans une pile.

Ainsi, la méthode **getInstance** permet de formaliser l'état d'un système de blocs sous forme de **variables booléennes**, qui représentent les connaissances relatives à l'état actuel du système.

# 5.2 Classe exécutable App4

Pour compiler les classes du packages blocksworld :

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

Pour exécuter la classe exécutable App4 :

```
java -cp ../build/.:../lib/bwgenerator.jar blocksworld.App4
```

La classe exécutable **App4** est un exemple d'application de techniques de **data mining** (extraction de connaissances) sur un problème du type **Blocksworld**. L'objectif de ce programme est de générer des états aléatoires du monde des blocs, de construire une base de données booléenne représentant ces états, puis d'extraire des motifs fréquents et des règles d'association à partir de cette base de données.

Le processus se décompose en plusieurs étapes :

*1.* ***Création du Gestionnaire des Variables***

Tout d'abord, un objet **GestionnaireVariables** est créé avec un **monde de 5 blocs** et **3 piles**. Ce gestionnaire génère les variables booléennes représentant des informations sur l'état des blocs et des piles. Il crée des variables comme :

- **Variables "onb,b'"** : Indiquent si un bloc **b** est posé sur un autre bloc **b'**.
- **Variables "on-tableb,p"** : Indiquent si un bloc **b** est posé sur la table dans la pile **p**.
- **Variables "fixedb"** : Représentent si un bloc **b** est fixé sur la table.
- **Variables "freep"** : Indiquent si une pile **p** est libre.

*2.* ***Génération des États Aléatoires***

Le programme utilise la **librairie bwgenerator.jar** pour générer des états aléatoires du monde des blocs. Cette librairie est particulièrement utile pour simuler une variété de configurations du problème sans avoir à les définir manuellement.

La génération des états se fait à l'aide de la méthode **getState** de la classe **Demo** (de la librairie bwgenerator). Cette méthode génère des configurations aléatoires, où chaque configuration représente un état du monde des blocs, sous forme de **listes de piles**, chaque pile contenant des indices de blocs.

*3.* ***Création de la Base de Données Booléenne***

Une fois que les états sont générés, chaque état est transformé en une instance booléenne représentant les relations entre les blocs et les piles. Ces instances sont ajoutées à une **base de données booléenne**.

- **Base de données de 10 000 instances** : Le programme génère **10 000 états aléatoires**, transformés en instances booléennes et ajoutés à la base de données. Cette base de données est ensuite utilisée pour extraire des motifs fréquents et des règles d'association.

Le code correspondant à la création de la base de données est le suivant :

```
int nombreInstances = 10000;

for (int i = 0; i < nombreInstances; i++) {

   List<List<Integer>> state = Demo.getState(random);

   Set<BooleanVariable> instance = gestionnaire.getInstance(state);

   base.add(instance);

}
```

*4.* ***Extraction des Motifs Fréquents avec l'Algorithme Apriori***

Une fois la base de données remplie, l'algorithme **Apriori** est appliqué pour extraire les **motifs fréquents**. Le critère de fréquence minimale est défini à **2/3**, ce qui signifie que seuls les motifs qui apparaissent dans au moins **2/3 des instances** seront extraits.

L'algorithme Apriori fonctionne en détectant les ensembles de variables booléennes qui apparaissent fréquemment ensemble dans les transactions de la base de données. Cela permet d'identifier des relations récurrentes entre les blocs et les piles.

Exemple de l'extraction des motifs fréquents :

```
float frequenceMinimale = 2.0f / 3.0f;  // Fréquence minimale de 2/3

Set<Itemset> motifsFrequents = apriori.extract(frequenceMinimale);
```

Les motifs extraits sont affichés, et chaque motif est une combinaison de variables qui apparaissent fréquemment dans les états du monde des blocs.

*5.* ***Extraction des Règles d'Association***

Après l'extraction des motifs fréquents, des **règles d'association** sont générées. Ces règles prennent la forme **"Si X alors Y"**, où X est une condition et Y est une conclusion.

- Le critère de fréquence minimale pour les règles est également **2/3**, ce qui signifie que seules les règles dont les prémisses apparaissent dans **au moins 2/3 des transactions** seront considérées.
- Le critère de **confiance minimale** est défini à **95%**, ce qui signifie que les règles extraites doivent être fiables à un degré de **95%**.

Le code correspondant pour l'extraction des règles d'association est :

```
float minFrequency = 2.0f / 3.0f;  // Fréquence minimale de 2/3

float minConfidence = 0.95f;  // Confiance minimale de 95%
```

Set<AssociationRule> regles = miner.extract(minFrequency, minConfidence);

Chaque règle extraite est affichée avec ses conditions (prémisse et conclusion), ainsi que la **fréquence** et la **confiance** associées.

*6.* ***Affichage des Résultats***

Enfin, les résultats des deux étapes d'extraction sont affichés :

- Les **motifs fréquents** qui ont une fréquence supérieure ou égale à **2/3**.
- Les **règles d'association** qui satisfont les critères de **fréquence** (>= 2/3) et de **confiance** (>= 95%).

# 6 Librairie d'affichage

Dans ce projet, nous avons utilisé la librairie **Blocksworld** dans deux contextes distincts afin de visualiser et simuler les solutions proposées dans différentes parties de l'étude.

***Visualisation des Plans en Planification***

*Dans la partie dédiée à la* ***planification*** *(Partie 2), la librairie* ***Blocksworld*** *a été utilisée pour simuler et visualiser les plans générés par les planificateurs. Après avoir défini un état initial et un objectif, les algorithmes de planification ont trouvé des séquences d'actions menant de l'état initial à l'objectif. Ces plans ont ensuite été simulés à l'aide de la librairie, qui permet de représenter graphiquement l'évolution du monde des blocs au fil des actions, rendant les résultats plus compréhensibles et intuitifs.*

***Affichage des Solutions en Problème de Satisfaction de Contraintes***

*Dans la partie dédiée aux* ***problèmes de satisfaction de contraintes*** *(Partie 3), nous avons également utilisé la librairie* ***Blocksworld*** *pour visualiser les solutions proposées par les solveurs de contraintes. Ces solveurs génèrent des états finaux respectant les contraintes spécifiées. La librairie a permis de représenter les solutions de manière claire, facilitant leur interprétation et leur validation.*

# 7 Conclusion

Ce projet a exploré différentes approches de résolution de problèmes dans le domaine du **Monde des Blocs**, en combinant des techniques issues de la **planification automatique**, des **problèmes de satisfaction de contraintes (CSP)**, et de l'**extraction de connaissances**. Chaque partie a permis de mettre en œuvre des méthodes variées pour modéliser, résoudre et analyser les problèmes dans ce cadre.