# Rapport de Projet : Aide à la décision et intelligence artificielle

Abdoul-bassit Anouar, Haili Rayane

20 novembre 2025

UNIVERSITÉ
CAEN
NORMANDIE

# Table des matières

# 1 Introduction

## 1.1 Présentation générale du projet

Dans le cadre de notre premier semestre de la troisième année du cycle licence en Informatique, il nous a été proposé un projet durant l'entièreté du semestre en aide à la décision et intelligence artificielle. Ce dernier consiste à implémenter des algorithmes de planification, de résolution de contraintes et d'extraction de connaissances sur le monde des blocs.

## 1.2 Objectifs pédagogiques

Ce projet fil rouge vise à explorer quatre domaines fondamentaux de l'intelligence artificielle appliqués au problème du monde des blocs. Le premier objectif consiste à `modéliser` le problème en définissant des variables et des contraintes pour représenter les configurations valides. Le deuxième objectif porte sur la `planification`, où il s'agit de trouver des séquences d'actions permettant de passer d'un état initial à un état final en implémentant des algorithmes de recherche et des heuristiques. Le troisième objectif concerne la `résolution de problèmes de satisfaction de contraintes (CSP)`, permettant de générer des configurations respectant un ensemble de contraintes données. Enfin, le quatrième objectif vise l'`extraction de connaissances (Data Mining)` à partir de bases de données d'états, en découvrant des motifs fréquents et des règles d'association.

## 1.3 Technologies utilisées

Ce projet a été implémenté en `Java` et utilise plusieurs algorithmes classiques : `DFS`, `BFS`, `Dijkstra` et `A*` pour la planification, ainsi que `AC-1` pour la programmation par contraintes. Pour l'extraction de connaissances, nous avons utilisé l'algorithme `Apriori` pour l'extraction de motifs et `BruteForce` pour l'extraction des règles. Pour la visualisation des configurations du monde des blocs, nous avons utilisé la bibliothèque graphique `Swing` avec `blocksworld` fournie avec le projet. L'extraction de connaissances s'appuie sur la bibliothèque `bwgenerator` pour générer les bases de données d'états.

# 2 Rôle des principales classes

## 2.1 Package modelling

— `BlocksWorld` : Représente un monde de blocs avec un nombre de blocs et de piles donnés.
— `BwWithConstraints` : Extension de BlocksWorld avec gestion des contraintes.
— `GrowingConstraint` : Contrainte de croissance imposant que les blocs empilés respectent un ordre croissant.
— `GrowingConfigurationConstraint` : Gère la création de contraintes de croissance pour le monde de blocs.
— `ImplicationOntoFixed` : Contrainte d'implication entre une variable onb et une variable fixedb'.
— `ImplicationOntoFree` : Contrainte d'implication entre une variable onb et une variable freep.
— `RegularityConstraint` : Contrainte de régularité imposant que trois blocs consécutifs empilés respectent une différence arithmétique constante.
— `RegularConfigurationConstraints` : Gère la création et la vérification des contraintes de régularité pour le monde de blocs.

## 2.2 Package planning

— `BwWithConstraintsAndActions` : Extension de BwWithConstraints avec gestion des actions de planification.
— `MisplacedBlocksHeuristic` : Heuristique basée sur le comptage des blocs mal placés.
— `WeightedBlocksHeuristic` : Heuristique pondérée pour le problème du monde des blocs.
— `PlanVisualizer` : Classe permettant de visualiser l'exécution d'un plan dans le monde des blocs.

## 2.3 Package cp

— `Main` : Classe principale pour tester différents solveurs CSP sur le problème du monde des blocs avec différentes combinaisons de contraintes.

### 2.4 Package datamining

— `BlocksWorldMiner` : Classe permettant de générer une base de données d'instances du monde des blocs pour le data mining.

## 3 Choix techniques importants

### 3.1 Modélisation

Pour la modélisation, nous avons opté pour une classe `BlocksWorld` qui centralise la gestion des blocs, des piles et des variables. Les variables sont organisées en trois catégories : `onb` (indique sur quoi est posé chaque bloc), `fixedb` (indique si un bloc est fixé) et `freep` (indique si une pile est libre). Chaque variable `onb` possède un domaine incluant piles et autres blocs, permettant de représenter toutes les configurations possibles.

Les variables sont stockées dans un `Map`, facilitant leur accès à travers leur nom et la mise en place de contraintes comme `ImplicationOntoFixed` ou `ImplicationOntoFree`.

### 3.2 Planification

Pour la planification, nous avons implémenté deux heuristiques pour guider la planification dans le monde des blocs :

— `WeightedBlocksHeuristic` : elle attribue un poids différent selon le type de variable : **3** pour les variables `on`, **2** pour les blocs `fixed` et **1** pour les piles `free`. Cela permet de privilégier les actions les plus déterminantes pour atteindre le but.

— `MisplacedBlocksHeuristic` : elle compte simplement le nombre de blocs dont la position actuelle diffère de l'état final. C'est une estimation simple du progrès vers le but.

Comparaison et choix : l'heuristique pondérée est plus performante car elle tient compte de l'importance relative des variables, ce qui permet de prioriser les déplacements critiques et d'atteindre le but plus rapidement et efficacement.

# 4 Configuration du Build avec Ant

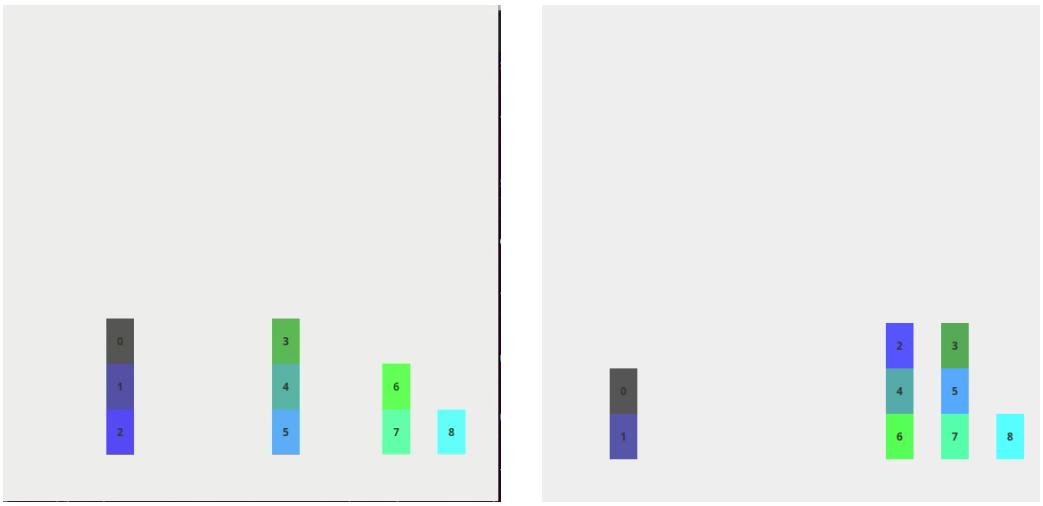
Pour automatiser la compilation et l'exécution du projet, nous avons utilisé un fichier build.xml. Cette configuration permet de :

1. `Compiler` l'ensemble des sources Java avec `ant compile`.
2. `Exécuter` et lancer la classe principale `Launcher` avec `ant run`. Lors de l'exécution, l'utilisateur est invité à choisir entre le lanceur de modelling, planning, csp ou datamining. En fonction de son choix, la classe correspondante est lancée.
3. Générer la `javadoc` dans le dossier `docs` avec `ant javadoc`.

# 5 Démonstrations

## 5.1 Planification

Les figures suivantes montrent l'évolution du système entre l'état initial et l'état final obtenu après planification.




État initial État final

FIGURE 1 – Illustration des résultats obtenus dans la partie Planning

## 5.2 Programmation par Contraintes

La figure ci-dessous présente une configuration générée par notre solveur CSP, respectant l'ensemble des contraintes implémentées : régularité, croissance, implications et différences.

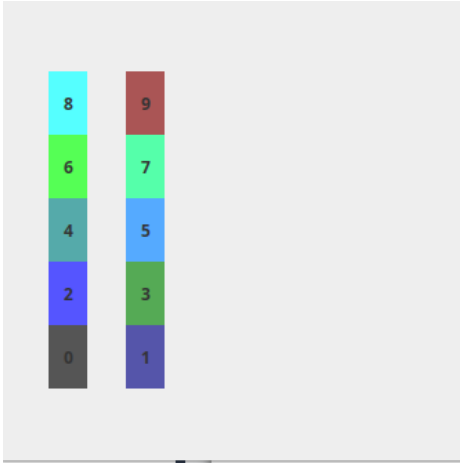



FIGURE 2 – Illustration des résultats obtenus dans la partie CSP

## 5.3 Data Mining

L'illustration suivante résume les principaux motifs et règles extraits lors de l'analyse.

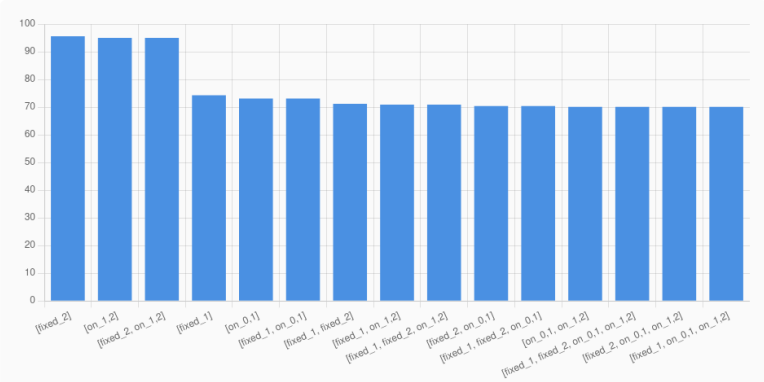



FIGURE 3 – Résultats obtenus dans la partie extraction des motifs

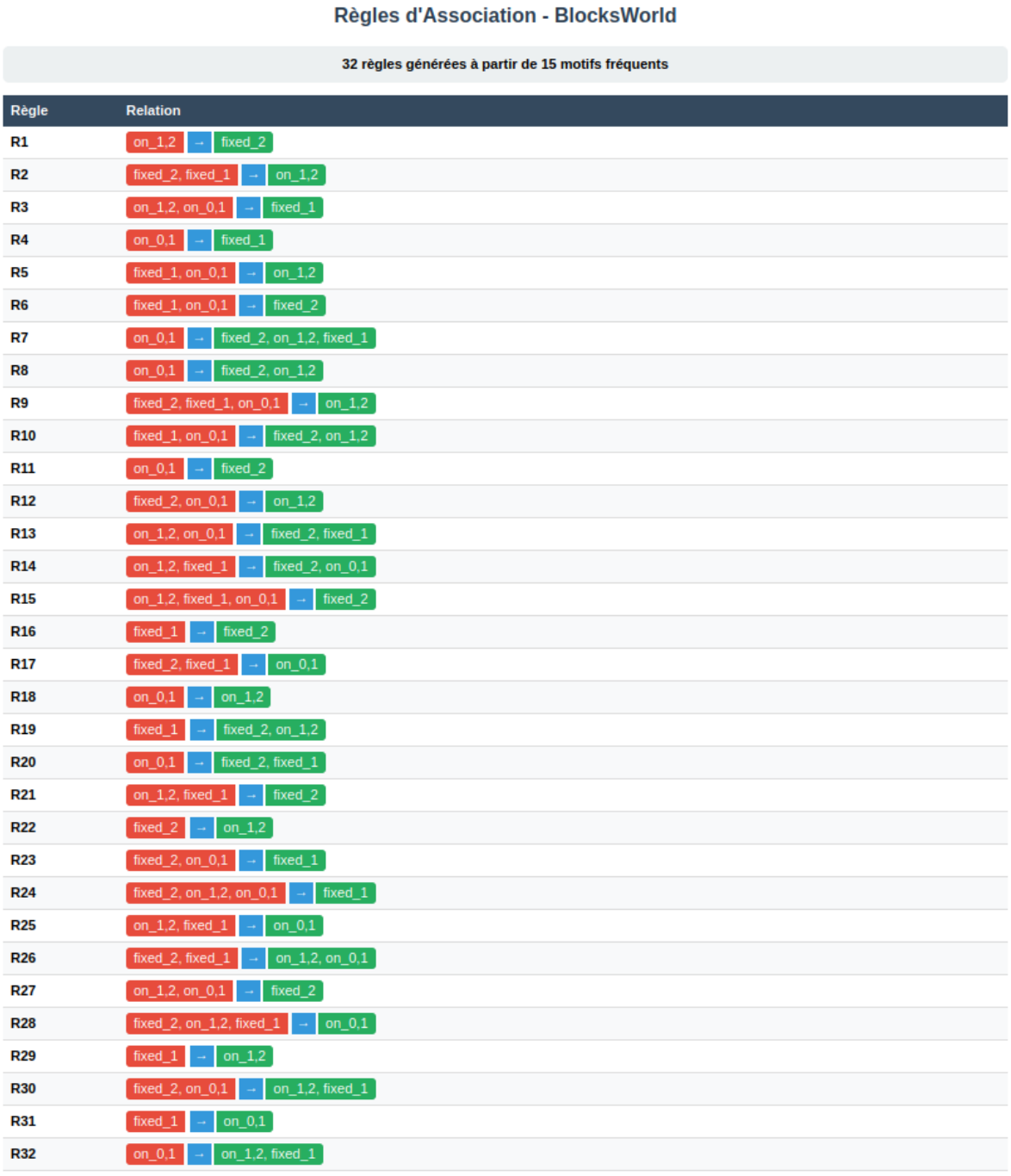



FIGURE 4 – Résultats obtenus dans la partie extraction des règles

# 6 Analyse des Performances

## 6.1 Planification

### 6.1.1 Performance des planners

Dans cette partie, nous avons comparé les performances de quatre algorithmes de recherche : A*, Dijkstra, BFS et DFS. Cette analyse nous permet d'évaluer leur efficacité en termes de temps d'exécution et d'exploration de nœuds.

| **Critère** | **A*** | **Dijkstra** | **BFS** | **DFS** |
|---|---|---|---|---|
| Nœuds explorés | 425 | 510 | 480 | 650 |
| Temps d'exécution | 207 ms | 326 ms | 295 ms | 180 ms |
| Longueur du plan | 18 actions | 18 actions | 18 actions | 24 actions |
| Optimalité | ✓Oui | ✓Oui | ✓Oui | ×Non |

TABLE 1 – Comparaison des performances des algorithmes de planification

### 6.1.2 Performance des heuristiques

Dans cette partie, nous avons évalué deux heuristiques pour l'algorithme A* : l'heuristique des blocs mal placés et l'heuristique pondérée, afin de mesurer leur impact sur les performances de la planification.

| **Critère** | **Misplaced Blocks** | **Weighted Blocks** |
|---|---|---|
| Nœuds explorés | 580 | 425 |
| Temps d'exécution | 411 ms | 207 ms |
| Longueur du plan | 18 actions | 18 actions |

TABLE 2 – Comparaison des performances des heuristiques pour A*

## 6.2 Programmation par Contraintes

Cette analyse compare les performances de trois solveurs CSP appliqués avec l'ensemble complet des contraintes implémentées : croissance, régularité, implications et différences.

| **Solveurs** | **Backtrack** | **MACSolveur** | **HeuristiqueMac** |
|---|---|---|---|
| Temps d'exécution | 30124 ms | 3541 ms | 131 ms |

TABLE 3 – Comparaison des performances des solveurs CSP

## 6.3 Data Mining

Sur une base de 10.000 instances avec 5 blocs et 5 piles, le processus a extrait 15 motifs (fréquence 2/3) et 32 règles (confiance 95/100) en moins de 5 secondes. Cette performance démontre l'efficacité des algorithmes sur des données de taille modérée. Cependant, `BruteForce` et `Apriori` pourraient rencontrer des limites avec des configurations plus complexes ou des bases de données plus volumineuses, leur complexité exponentielle risquant d'affecter les temps d'exécution.

# 7 Problèmes rencontrés et Solutions

| **Problèmes** | **Solutions** |
|---|---|
| Modélisation initiale trop complexe du BlocksWorld | Simplification avec 3 types de variables : onb, fixedb, freep |
| Contrainte de régularité ternaire difficile à utiliser sur nos solveurs implémentés, car ils fonctionnent uniquement avec des contraintes unaires ou binaires | Transformation en contraintes binaires avec paires (b1,b2) et recréation de b3 |

TABLE 4 – Problèmes et solutions lors du développement

# 8 Conclusion

Ce projet nous a permis d'explorer quatre domaines essentiels de l'intelligence artificielle à travers le problème du monde des blocs. La phase de modélisation a posé des bases claires pour représenter les différentes configurations. La planification, elle, a montré comment des algorithmes guidés par des heuristiques peuvent trouver des solutions efficaces. La programmation par contraintes a mis en valeur la puissance des solveurs modernes, notamment MAC associé à une bonne heuristique, pour traiter des problèmes plus complexes. Enfin, l'extraction de connaissances nous a aidés à mieux comprendre les configurations en faisant ressortir des motifs et des relations qui ne sont pas toujours visibles au premier regard.