FACULTÉ DES SCIENCES
LICENCE 3 INFORMATIQUE

Aide à la Décision en Intelligence Artificielle

Rapport - Fil Rouge

Auteurs :
Anis IDRES **Ayoub REZALA**

Année Universitaire : 2024/2025

Table des matières

1 Introduction

Le projet nommé **Fil Rouge** représente une mise en œuvre complète du **monde des blocs (BlocksWorld)**, un exemple classique d'application en Intelligence Artificielle. Ce projet, développé en Java, aborde les principaux concepts d'IA, tels que la modélisation, la planification, et la résolution de contraintes.
Chaque composante du projet est associée à une classe exécutable, permettant une ex- ploration approfondie et une interaction simplifiée pour tester différentes fonctionnalités. Une bibliothèque graphique, blocksworld.jar, est également intégrée pour visualiser les configurations et les actions dans le monde des blocs.
Enfin, des solveurs sont exploités pour valider la satisfaction des contraintes, et des algorithmes spécifiques sont utilisés pour améliorer la performance et analyser les solutions possibles.

2 Parties du Projet

2.1 Modélisation

Dans notre modélisation, nous avons choisi d'utiliser les classes `Variable` et `BooleanVariable`, développées lors des premiers TPs, pour représenter les différentes variables du monde des blocs. Plus précisément, les variables `on`, qui indiquent la position des blocs, sont représentées par la classe `Variable`, tandis que les variables `fixed` (indiquant si un bloc est fixé) et `free` (indiquant si une pile est libre) utilisent la classe `BooleanVariable`. La gestion et la récupération de ces variables sont centralisées dans la classe `BlocksWorld`.

Pour représenter les contraintes, nous avons utilisé les différentes classes `Constraint`, `DifferenceConstraint`, `Implication`, et `UnaryConstraint`, également développées lors des TPs. La classe `BlocksWorldConstraints` est chargée de générer trois types de contraintes :

Les contraintes de différence, empêchant deux blocs d'occuper la même position. Les contraintes d'implication, qui lient la position d'un bloc à son état de fixation (`fixed`) ou à la disponibilité d'une pile (`free`). La classe `BlocksWorldRegular` ajoute des contraintes supplémentaires pour garantir que les blocs posés sur chaque pile respectent un écart constant entre leurs indices. De son côté, la classe `Croissante` génère des contraintes assurant que les blocs sont positionnés de manière croissante, c'est-à-dire qu'un bloc ne peut être placé que sur un bloc ayant un indice inférieur ou directement sur une pile.

Enfin, la classe `DemoModelling` sert de point d'entrée exécutable pour tester nos

configurations. Elle permet de vérifier si des configurations données respectent les contraintes de régularité et de croissance définies par notre modélisation.

2.2 Planification

Pour la partie planification, nous avons développé une classe permettant de gérer toutes les actions possibles dans un monde des blocs. Chaque action est représentée comme une instance de la classe `BasicAction`, que nous avons implémentée lors des TPs. Une action est définie par deux éléments essentiels :

— Une **map de préconditions**, regroupant les conditions nécessaires à son exécution.

— Une **map d'effets**, décrivant les modifications appliquées au monde des blocs après l'exécution de l'action.

Pour tester les différents planificateurs, nous avons créé des états initiaux et finaux. Ces états sont définis en parcourant l'ensemble des variables du monde des blocs et en leur attribuant des valeurs correspondant à la position souhaitée pour chaque bloc.

Dans la classe `DemoPlanning`, nous générons un état initial et un état final, puis exécutons les principaux algorithmes de planification, tels que :

— `DFS` (Depth-First Search),

— `BFS` (Breadth-First Search),

— `Dijkstra`,

— et `A*`.

Pour chacun de ces planificateurs, nous affichons le plan trouvé ainsi que des métriques comme le **nombre de nœuds explorés** et le **temps d'exécution**.

Deux heuristiques significatives ont été intégrées pour guider le processus de planification :

1. **Nombre de blocs mal placés** : Cette heuristique évalue le nombre de blocs qui ne sont pas à leur position cible à chaque étape. Elle permet de minimiser progressivement les erreurs de placement, contribuant à une planification plus précise.
2. **Distance totale des déplacements** : Cette heuristique calcule la distance totale que chaque bloc doit parcourir pour atteindre sa position finale. Elle fournit une estimation utile pour orienter la planification vers des solutions efficaces en réduisant les déplacements.

Ces heuristiques renforcent notre capacité à explorer et évaluer les plans de manière plus détaillée, aboutissant à des solutions de planification optimisées.

Enfin, pour la visualisation graphique des états et des actions dans le monde des blocs, nous utilisons la bibliothèque `blocksworld.jar`, qui permet d'afficher les configurations de manière intuitive et interactive.

2.3 Problème de satisfaction des contraintes

La résolution des problèmes de planification dans le monde des blocs repose sur la **satisfaction de contraintes**, un concept fondamental permettant de modéliser les relations entre les différentes entités du problème. Trois classes exécutables sont utilisées pour représenter des configurations distinctes du problème :

- `DemoReguliere` : Cette classe se concentre sur la validation des contraintes régulières dans les configurations du monde des blocs.
- `DemoCroissante` : Cette classe implémente des contraintes garantissant une organisation croissante, où chaque bloc est positionné uniquement sur un bloc ayant un indice inférieur ou sur une pile.
- `DemoCroissanteReguliere` : Cette classe combine les contraintes régulières et croissantes pour offrir une solution plus complète et stricte.

Ces classes utilisent différents solveurs pour résoudre les problèmes de planification :

- `BacktrackSolver` : Un solveur basé sur une recherche exhaustive avec retour en arrière.
- `MACSolver` : Un solveur optimisé exploitant la cohérence des arcs pour réduire l'espace de recherche.
- `HeuristicMACSolver` : Une version améliorée intégrant des heuristiques spécifiques pour accélérer la recherche de solutions optimales.

Chaque démo évalue les performances des solveurs en mesurant le **temps de calcul** nécessaire et en comparant l'efficacité des approches utilisées pour résoudre les configurations définies. Ces mesures permettent de mieux comprendre l'impact des contraintes et des heuristiques sur les solutions obtenues.

2.4 Extraction de connaissances

Dans la section dédiée à l'extraction de connaissances, nous avons développé une classe appelée `BooleanBlocksWorld` pour modéliser le monde des blocs sous forme de variables booléennes. Cette classe est conçue pour créer un ensemble de variables booléennes représentant les relations entre les blocs dans un monde donné. Ces variables booléennes sont utilisées pour construire une base de données booléenne (`BooleanDatabase`).

Pour n instances, nous récupérons un état généré à l'aide de la méthode `getState()` de la bibliothèque fournie `bwgenerator`. Ensuite, nous ajoutons les `BooleanVariable` correspondant à cet état dans notre base de données. Ainsi, cette base de données est remplie avec des instances générées du monde des blocs, chaque instance étant représentée par un ensemble de ces variables booléennes.

Pour extraire des connaissances à partir de cette base de données, deux algorithmes distincts ont été appliqués :

- **Algorithme Apriori** : Utilisé pour découvrir des motifs fréquents, révélant des configurations récurrentes dans le monde des blocs.
- **Algorithme de force brute** : Utilisé pour extraire des règles d'association, mettant en évidence des relations significatives entre les variables.

L'exécution de ce programme est initiée par la classe `DemoDataMining`, qui offre une vue d'ensemble des motifs fréquents et des règles d'association extraites. Cela permet d'analyser et de mieux comprendre les relations et les configurations typiques dans le contexte du modèle du monde des blocs.

3 Classes exécutables

- **Modélisation :** `DemoModelling`
- **Planification :** `DemoPlanning`
- **Problème de satisfaction de contraintes :**
  - **Configuration Régulière :** `DemoReguliere`
  - **Configuration Croissante :** `DemoCroissante`
  - **Configuration Régulière & Croissante :** `DemoCroissanteReguliere`
- **Extraction de connaissances :** `DemoDataMining`

4 Commandes de compilation et d'exécution

Pour assurer une utilisation optimale de notre projet, il est essentiel de suivre les bonnes pratiques de compilation et d'exécution. Ces étapes permettent de préparer et de tester efficacement les fonctionnalités développées. Voici les instructions détaillées :

1. Tout d'abord, ouvrez un terminal dans le répertoire source `/src` de notre projet.
2. Ensuite, pour compiler toutes les classes nécessaires et prendre en compte les bibliothèques externes `blocksworld.jar` et `bwgenerator.jar`, utilisez la commande suivante :

```
1    javac -d ../build -cp .:../lib/blocksworld.jar:../lib/
   bwgenerator.jar modelling/*.java planning/*.java cp/*.java
   datamining/*.java blocksworld/*.java
2
```

3. Une fois la compilation réussie, vous pouvez exécuter n'importe quelle classe exécutable de votre projet en utilisant la commande suivante. Assurez-vous de remplacer `NOM_CLASSE_EXECUTABLE` par le nom réel de la classe que vous souhaitez exécuter :

```
1    java -cp ../build:../lib/blocksworld.jar:../lib/
   bwgenerator.jar blocksworld.NOM_CLASSE_EXECUTABLE
2
```