Rapport

Projet Fil Rouge : Monde de Blocs

Module : Intelligence Artificielle

Contributeurs :

Nom et prénom	**Numéro d'étudiant**
Salah Eddine ELOUARDI	22110344
Mohammed Aidi	22309377

Université de Caen Normandie
UFR des Sciences
Département Informatique
2ème année de licence d'informatique

Table des matières

0.1 Introduction

Le projet **Fil Rouge** s'inscrit dans le cadre du module d'Intelligence Artificielle et porte sur la résolution d'un problème complexe, le *Monde des Blocs*. Ce dernier consiste à manipuler des piles de blocs pour atteindre un objectif donné, en respectant des contraintes spécifiques et en utilisant diverses techniques d'intelligence artificielle.

Ce projet combine trois concepts clés :

— **Satisfaction de contraintes (CSP)** : Générer des états qui satisfait l'ensemble des contraintes.

— **Planification** : Générer des séquences d'actions menant à un état final.

— **Extraction de connaissances** : Analyser les données pour découvrir des motifs fréquents et des règles d'associations.

L'objectif est d'implémenter un système modulaire et robuste capable de résoudre différents scénarios, tout en assurant la validité des configurations.

0.2 Rôles des classes principales

Le projet repose sur plusieurs classes principales, chacune ayant un rôle bien défini dans la gestion des états, des actions et des contraintes.

0.2.1 BlockWorld

La classe `BlockWorld` est le noyau du projet. Selon un nombre de blocks et un nombre de pile elle prépare 3 ensembles de variables qui permet de représenter un état :

— **Les variables "onB" :** ces variables sont utilisés pour représenter le numéro de block (un entier null ou positive) ou de pile (un entier négative) sur lequel chaque block est posé.

— **Les variables "fixedB" :** ces valriables sont utilisés pour représenter par un booléan l'état de chaque block ("true" si le block n'est pas déplaçable "false" sinon).

— **Les variables "freeP" :** ces valriables sont utilisés pour représenter par un booléan l'état de chaque pile ("true" si la pile est libre "false" sinon).

0.2.2 Classes pour les contraintes

Trois classes pour gérer trois types de contraintes :

— `BWAvecConstraints` : Empêche les états invalides (exemple : 2 variables **"onb"** ont la mème valeur, **"onb"** à la valeur **"b'"** et **"fixedb'"** à la valeur **false**, ..).

— `BWCroissantConstraints` : Assure que les blocs sont empilés dans un ordre croissant.
— `BWRegulieretConstraints` : Assure que l'écart entre les blocks est identique pour chaque pile .

0.2.3 BWActions

Cette classe définit les actions possibles dans le système, comme déplacer un bloc d'une pile à une autre. Chaque action est caractérisée par :
— **Préconditions :** Par exemple, un bloc doit être libre pour être déplacé.
— **Effets :** Les changements qui surviennent après l'action (e.g., un bloc devient libre).

0.2.4 BWVariablesBoolean

Les variables booléennes modélisent les relations entre les blocs. Cette classe simplifie la représentation des états et permet l'extraction des connaissances d'une manière plus simple.

0.3 Démonstrations réalisées

Pour valider les fonctionnalités, plusieurs tests ont été réalisés, couvrant les contraintes, la planification et l'extraction de connaissances.

0.3.1 Tests des contraintes (CSP)

Les contraintes ont été testées sur des configurations variées pour valider leur efficacité. Par exemple :
— **État initial :** Un bloc empilé de manière incorrecte.
— **Résultat attendu :** État invalide signalé par le système.

0.3.2 Tests de planification

Des algorithmes comme BFS, DFS et A* ont été utilisés pour générer des plans atteignant des états finaux valides. Exemple :

```
Etat initial : [A, B], [C]
Plan :
1. D placer B vers C.
2. D placer A vers B.
```

Etat final : [C], [A, B].

0.3.3 Tests d'extraction de connaissances

L'extraction de connaissances vise à découvrir des motifs fréquents et des règles d'association à partir d'une base de données de 10 000 états générés aléatoirement avec la librairie `bwgenerator`. Chaque état est représenté par des variables booléennes (`onB,b'`, `on-tableB,p`, `fixedB`, `freeP`), facilitant l'analyse.

Algorithmes utilisés :

- **Apriori :** Extraction de motifs fréquents avec une fréquence minimale de 2/3.
- **BruteForceAssociationRuleMiner :** Extraction de règles d'association avec une fréquence minimale de 2/3 et une confiance minimale de 95%.

Résultats obtenus :

- Motifs fréquents extraits : 150 (par exemple, "Si A est sur B, alors C est libre").
- Règles d'association extraites : cohérentes avec la logique du monde des blocs.

Ces résultats valident l'approche et démontrent la pertinence des techniques d'extraction pour analyser des configurations complexes.

0.4 Choix d'implémentation

0.4.1 Contraintes personnalisées

Les contraintes croissantes et régulières assurent des configurations valides en réduisant le nombre d'états invalides générés.

0.4.2 Actions planifiées

La classe `BWActions` définit l'ensemble des actions possibles dans un monde de blocs, en héritant de la classe `BWAvecConstraints`. Chaque action correspond au déplacement d'un bloc d'une position source vers une destination, sous certaines conditions.

Définition des actions Les actions sont générées en respectant les contraintes suivantes :

- Le bloc source doit être libre (`fixedB[i] = false`).
- La destination doit être libre :
  - Si la destination est une pile, `freeP[k] = true`.
  - Si la destination est un autre bloc, `fixedB[k] = false`.

Préconditions et effets Chaque action est définie par :

- **Préconditions :** - Le bloc est sur une position donnée (`onB[i] = j`). - Le bloc est libre (`fixedB[i] = false`). - La destination est libre (`freeP[k] = true` ou `fixedB[k] = false`).
- **Effets :** - Le bloc est déplacé à la nouvelle position (`onB[i] = k`). - La position source devient libre. - La position destination devient occupée.

Ensemble des actions Le constructeur de la classe génère toutes les combinaisons possibles d'actions pour chaque bloc, en itérant sur les positions sources et destinations, tout en évitant les déplacements invalides (par exemple, déplacer un bloc vers lui-même). Les actions générées sont stockées dans un ensemble (`Set<Action>`) accessible via la méthode `getActions()`.

Avantages Cette approche garantit que :

- Toutes les actions valides sont préparées à l'avance.
- Les contraintes du monde des blocs sont respectées, évitant la génération d'états invalides.

0.4.3 Heuristiques

Les heuristiques `BWHeuristic1` et `BWHeuristic2` sont utilisées pour guider la recherche vers des solutions optimales.

BWHeuristic1 Elle calcule le **nombre de blocs mal placés** par rapport à l'état but, en comparant les variables `on` de l'état actuel et de l'état but. **Avantages :** Rapide et simple, elle fournit une estimation minimale du coût. **Limitation :** Ne considère pas les dépendances entre blocs.

BWHeuristic2 Elle améliore la première en ajoutant le **nombre de blocs au-dessus du premier bloc mal placé** dans chaque pile, reflétant les déplacements nécessaires pour libérer les blocs. **Avantages :** Plus précise pour les configurations complexes. **Limitation :** Plus coûteuse en calcul.

Ces heuristiques équilibrent simplicité et précision, guidant efficacement la recherche tout en minimisant les états explorés.

0.5 Résultats et analyse

Les tests montrent que le système est robuste et efficace :
— Les contraintes sont respectées dans tous les cas.
— Les plans générés atteignent systématiquement les objectifs.
— Les motifs extraits sont cohérents avec les données initiales.

0.6 Exécution du projet

Pour exécuter les différentes parties du projet, un script `script.sh` est fourni dans le dossier `scripts`. Ce script permet de naviguer facilement entre les tests des différentes fonctionnalités via un menu interactif.

0.6.1 Préparation avant l'exécution

Avant d'exécuter le fichier `script.sh`, assurez-vous de :
— Ajouter les fichiers `bwgenerator.jar` et `blocksworld.jar` dans le dossier `lib`.
— Compiler les fichiers du projet en exécutant `script.sh`, ce qui déclenche automatiquement la compilation et l'exécution.

0.6.2 Menu principal

Lors de l'exécution du script, un menu s'affiche pour vous permettre de choisir les tests à exécuter :

```
Choisissez une option :
1) Test constraints
2) Test plannificateurs
3) Test CSP
4) Test Extraction des connaissances
5) Quitter
```

Chaque option permet d'explorer une partie du projet :
— **Test constraints :** Vérifie la validité des états en appliquant les contraintes.
— **Test plannificateurs :** Lance les algorithmes BFS, DFS et A* (avec les heuristiques 1 et 2) en modes console et graphique.
— **Test CSP :** Génère des états satisfaisant les contraintes croissantes et régulières.
— **Test Extraction des connaissances :** Extrait des motifs fréquents et des règles d'association.

0.6.3 Partie graphique

Pour la partie graphique (disponible dans les tests de planificateurs), assurez-vous que le fichier `blocksworld.jar` est correctement placé dans le dossier `lib`, sinon des erreurs d'exécution peuvent survenir.

Ce script fournit un moyen simple et efficace pour naviguer entre les différentes fonctionnalités du projet. En combinant les modes console et graphique, il permet une validation complète des résultats tout en assurant une expérience utilisateur fluide.

0.7 Conclusion

Ce projet a permis de modéliser un problème complexe en combinant plusieurs techniques d'intelligence artificielle. Les résultats obtenus montrent la robustesse et la validité des solutions proposées.