Université de Caen Normandie

Licence 3 Informatique

Aide à la décision en Intelligence Artificielle

Le Monde Des Blocks (blocksWorld)

Fil rouge

Réalisé par :
DIAB Joyce
BOUAMAR Lina

Année Universitaire
2024/2025

Table des matières

1 Introduction

Dans notre projet du "Fil Rouge", nous avons développé une version fonctionnelle du Monde des Blocs (BlocksWorld), un modèle classique en Intelligence Artificielle. Ce projet a été réalisé en Java et aborde plusieurs aspects fondamentaux, notamment la modélisation, la planification, la résolution de contraintes et extraction des connaissances. À chaque étape, nous avons mis en œuvre des classes exécutables pour explorer et tester les différentes fonctionnalités, en nous appuyant sur des librairies graphiques pour visualiser les configurations.

Notre travail s'appuie également sur des concepts vus en cours et en TPs, comme la vérification de contraintes via des solvers et l'extraction de connaissances à partir de bases de données. L'objectif principal était de mieux comprendre les défis de modélisation et de planification dans un environnement simulé tout en appliquant des techniques d'IA.

2 Les différentes parties du projet

2.1 Modélisation

Pour représenter le monde des blocs, nous avons utilisé des classes comme Variable et BooleanVariable qu'on a déjà implémenté lors de nos Tps précédents. Ces classes ont été adaptées pour représenter les différents éléments du projet :

Variables "onB" : Elles indiquent sur quel bloc ou pile chaque bloc est placé. Nous utilisons la classe Variable, avec des noms onBi où i représente le numéro du bloc (ex : "onB1") et un domaine qui inclut des entiers (positifs ou null pour les blocs, négatifs pour les piles) et exclut i.

Variables "fixedB" et "freeP" : Ces variables booléennes, créées à l'aide de BooleanVariable, indiquent si un bloc est fixé (pas déplaçable) ou si une pile est libre.

Pour gérer ces variables, nous avons conçu une classe principale appelée BlocksWorld, qui regroupe toutes les données nécessaires. Elle prend en entrée un nombre de blocs et un nombre de piles et dispose un Set<Variable>, ainsi que des Map<Integer, Variable> qui facilitent la recupération d'une variable concernée par son numéro.

Cette derniere possede une fonction instanciation qui crée une instance a partir d'une liste de piles(List<List<Integer»)

En complément, des classes spécifiques comme BlocksWorldConstraints qui extends de Blocksworld permettent de créer un monde avec des règles particulières ; les Contraintes de différence qui assurent que deux blocs ne partagent pas la même position, et les contraintes d'implication qui lient l'état d'un bloc (fixe ou libre) à sa position.

En termes de contraintes, nous avons réutilisé des classes telles que Constraint, DifferenceConstraint, Implication et UnaryConstraint.

La classe "BlocksWorldRegular" qui extends de BlocksWorldConstraints permet de définir des contraintes assurant que les blocs posés sur chaque pile respectent un écart constant. Quant à la classe "BlocksWorldIncreasing", elle génère des contraintes garantissant une organisation croissante des blocs. Autrement dit, un bloc ne peut être placé que sur un autre bloc ayant un numéro plus petit.

La partie modélisation a été testée via une classe exécutable, DemoModelling, où des configurations de blocs sont générées et vérifiées selon les règles définies.

2.2 Planification

Pour automatiser les actions possibles dans le monde des blocs, nous avons créé une classe regroupant toutes les actions réalisables (déplacer un bloc, le fixer, etc.). Cette classe fonctionne en prenant en entrée le nombre de blocs, le nombre de piles et extends BlocksWorld pour récupérer les differentes variables.

Les actions dans notre système sont modélisées en tant qu'instances de la classe "BasicAction" et pour les définir on a utilisé une map de préconditions qui, elle, liste les conditions qui doivent être satisfaites pour que l'action soit applicable. Par exemple, pour déplacer un bloc, il doit être déplaçable, libre et son emplacement cible doit également être disponible. Une map d'effets qui, elle, décrit les changements qui seront apportés au monde des blocs une fois l'action exécutée. Ces changements mettent à jour les relations entre les blocs et les piles (par exemple, un bloc passant de l'état "on-table" à "on-block").

Le monde des blocs lui-même est représenté par une structure "map<Variable, Object>", où chaque variable encode un aspect du système (comme les positions des blocs ou leur état "libre"/"fixe"), et l'objet associé représente la valeur actuelle de cette variable.(Integer ou boolean dans notre cas)

Dans notre classe "DemoPlanning", nous avons mis en place un processus qui commence par definir deux états, un etat initial et un etat final, la nous utilisons différents algorithmes de planification pour explorer et trouver un chemin entre ces deux états ; "DFS", "BFS", "Dijkstra" et "Astar". Pour chaque planificateur, nous affichons :

1. Le plan trouvé : une séquence ordonnée d'actions nécessaires pour atteindre l'état final.
2. Le nombre de nœuds explorés : un indicateur de l'efficacité de l'algorithme.
3. Le temps d'exécution : mesuré en millisecondes, reflétant la rapidité de chaque méthode.

Pour améliorer la performance de l'algorithme, nous avons implémenté deux heuristiques. La premiere calcule le nombre de blocs mal placés, son objectif est de minimiser ce nombre progressivement, orientant ainsi la planification vers des solutions plus précises. La deuxieme mise en œuvre repose sur une estimation précise du nombre minimal d'actions qui doit etre effectué pour atteindre l'état final. Elle calcule, pour chaque pile, le nombre de déplacements nécessaires en fonction de la position du plus profond bloc mal placé.

Pour la visualisation graphique des états et des actions, nous avons utilisé la bibliothèque fournie "blocksworld.jar", qui propose des outils dédiés à la représentation visuelle des configurations du monde des blocs. Cela permet d'observer l'évolution des états à mesure que les actions sont appliquées, rendant ainsi la compréhension et l'analyse des résultats plus intuitives.

C'est pour cela qu'on a créé une classe "BlocksWorldDisplay" qui prend en parametres un monde de blocs et un titre.

La méthode makeState() a été utilisée pour créer un état du monde, facilitant ainsi la visualisation graphique de l'évolution des configurations.

La méthode displayState() permet l'affichage d'une instance et la méthode simulatePlan() prend un état initial et un plan d'actions et simule l'évolution du monde suivant les actions.

3 Problème de satisfaction des contraintes

La résolution des problèmes de planification dans le monde des blocs repose sur le concept fondamental de satisfaction de contraintes. Cela permet de modéliser de manière formelle les relations entre les blocs et les piles, facilitant ainsi l'exploration et l'optimisation des solutions.

Pour tester les différentes configurations et contraintes, on a trois classes executables DemoSolverRegular, DemoSolverIncreasing, et DemoRegularIncreasing

La classe DemoSolverRegular initie le processus en créant une instance de BlocksWorld, représentant un état du monde des blocs. Elle extrait ensuite toutes les variables nécessaires pour décrire cet état (positions des blocs, états des piles, etc.) en utilisant BlocksWorldConstraints pour définir les contraintes de base et BlocksWorldRegular qui introduit des contraintes régulières garantissant des écarts fixes entre les blocs empilés sur les piles. Ces contraintes constituent un ensemble fondamental pour les solvers qui explorent les solutions.

La classe DemSolveroIncreasing prolonge le principe de DemoSolverRegular mais en ajoutant plutot des contraintes spécifiques à la configuration dite croissantes. Ces contraintes sont implémentées via la classe BlocksWorldIncreasing, qui ajoute une logique supplémentaire tout en maintenant les contraintes de base.

La classe DemoRegularIncreasing combine toutes les contraintes précédentes (base, régulières et croissantes) pour créer une configuration complète et complexe. Elle offre une perspective unifiée en intégrant les exigences de régularité tout en respectant les règles de croissance.

Chaque démonstration utilise plusieurs solveurs pour résoudre les cp, chacun ayant ses propres caractéristiques. Le BacktrackSolver effectue une exploration en profondeur en testant toutes les solutions possibles de manière systématique. Bien que robuste, il peut être inefficace pour des cp de grande taille. MACSolver s'appuie sur la propagation de contraintes pour réduire l'espace de recherche. Ce solver est souvent plus rapide que BacktrackSolver, car il élimine rapidement les solutions impossibles. HeuristicMACSolver c'est une version améliorée du MACSolver, intégrant des heuristiques spécifiques pour accélérer la recherche de solutions optimales.

4 Extractions des connaissance

Dans l'extraction de connaissances, nous avons conçu une classe appelée BooleanBlocksWorld pour modéliser le monde des blocs à l'aide de variables booléennes. Cette classe génère un ensemble de variables représentant les relations entre les blocs, notamment lorsqu'un bloc est positionné "on" un autre bloc ou "on-table" directement sur la table.

La génération de ces variables repose sur les concepts issues du modèle du monde des blocs. La classe BooleanBlocksWorld crée des variables booléennes reflétant les relations "on" et "on-table". Ces variables sont ensuite utilisées pour alimenter une base de données booléenne (BooleanDatabase).

Pour construire cette base, on a crée une méthode createInstance qui prend une liste de piles et nous renvoie l'instance qui contient toutes les boolean variables qui sont a true. Nous générons des états à l'aide de la méthode getState() de la bibliothèque fournie "bwgenerator". Les variables booléennes correspondant à chaque état sont ensuite ajoutées à la base de données. Ainsi, celle-ci se compose de multiples instances du monde des blocs, chaque instance étant définie par un ensemble de variables booléennes.

Deux algorithmes ont été appliqués pour extraire des connaissances de cette base. L'algorithme Apriori a permis d'identifier des motifs fréquents, révélant des configurations récurrentes dans le monde des blocs. En parallèle, un algorithme de force brute a été utilisé pour générer des règles d'association, mettant en évidence des relations significatives entre les variables.

Enfin, la classe DemoDataMining offre une vue d'ensemble des motifs fréquents et des règles d'association issues du modèle. Ce travail ouvre des perspectives prometteuses pour l'analyse des relations dans le monde des blocs et des systèmes similaires.

5 Conclusion

Ce projet a permis de combiner plusieurs approches pour explorer et analyser les dynamiques du monde des blocs. La modélisation via la classe BooleanBlocksWorld a facilité une représentation claire des relations entre les blocs, tandis que l'utilisation des algorithmes Apriori et de force brute a permis d'extraire des motifs fréquents et de mettre en évidence des règles d'association importantes.

Les connaissances ainsi obtenues ne se limitent pas à une compréhension théorique : elles apportent des perspectives concrètes pour améliorer les planificateurs ou résoudre des problématiques complexes basées sur des systèmes à contraintes similaires.

En résumé, ce travail illustre comment des outils d'extraction de connaissances peuvent enrichir l'analyse de systèmes complexes, tout en offrant des solutions pratiques et optimisées pour des applications futures.