AIDE A LA DECISION ET INTELLIGENCE ARTIFICIELLE

RAPPORT FIL ROUGE

FOFANA Abdoulaye,
L3 info,
Groupe 3B,

November 19, 2024

UNIVERSITÉ
CAEN
NORMANDIE

Le projet "Fil Rouge" présenté ici consiste en une réalisation complète du monde des blocs (BlocksWorld) en Intelligence Artificielle. Développé en Java, ce projet couvre divers aspects fondamentaux. Le monde des blocs (BlocksWorld) est un modèle emblématique en intelligence artificielle, souvent utilisé pour illustrer ou tester des algorithmes de planification. Dans un monde de block, nous avons un certain nombre de blocs, numérotés de 0 à n - 1, organisés en plusieurs piles, dont certaines peuvent être vides.
Pour représenter les différentes configurations d'un monde de n blocs et m piles, nous utilisons trois ensembles de variables. Chaque bloc b est associé à une variable onb, qui indique s'il est posé sur un autre bloc ou sur une pile. Nous avons également une variable booléenne fixedb pour déterminer si un bloc est indéplaçable, et une autre variable booléenne freep pour indiquer si une pile est libre. Les contraintes qui régissent ces configurations sont principalement binaires et garantissent que les blocs ne peuvent pas être empilés de manière circulaire, tout en respectant les relations de dépendance entre les blocs et les piles.
Dans ce rapport j'explique le choix des implémentations ainsi que les démonstration réalisées.

Sommaire

1 Modélisation : Variables et Contraintes

Dans cette section, nous allons modéliser le monde des blocs à l'aide de variables et appliquer toutes les contraintes qui y sont associées. Voici les classes qui ont été développées pour cela.

1.1 Classe BlocksVariable

La classe BlocksVariable est une structure centrale pour modéliser les variables d'un monde des blocs avec un constructeur qui prend en paramètre un nombre de blocs et de piles. Elle permet de gérer trois types de variables :

- Onb Variables : Représentent la position d'un bloc (sur une pile ou un autre bloc).
- Fixedb Variables : Indiquent si un bloc est fixé (non mobile).
- Freep Variables : Indiquent si une pile est libre (vide).

Les méthodes buildOnbVariables, buildfixedbVar, et buildfreepVar génèrent respectivement les ensembles de variables onbVar, fixedbVar et freepVar en utilisant les classes Variables et BooleanVariable du package modelling.
Les méthodes getOnbVariableByIndexe, getFixedbVariableByIndexe, et getFreepVariableByIndexe permettent de récupérer directement une variable par son index. La méthode getVariablesState génère une représentation des états des variables à partir d'un tableau.

1.2 Classe BlocksWord

La classe BlocksWorld modélise les contraintes de base du monde des blocs. Pour cela son constructeur initialise un objet BlocksVariable pour définir les variables associées aux blocs et piles.Les contraintes spécifiques (DifferenceConstraint et Implication) sont générées automatiquement lors de l'initialisation.

1.2.1 Contrainte de différence

- Garantit que pour chaque couple de blocs différents b , b' les variables onb et onb' ne peuvent pas prendre la même valeur. La classe DifferenceConstraint de modelling a été utilisée pour ça.

1.2.2 Contraintes d'implication

- Si un bloc est positionné sur un autre bloc (onb1 = b'), alors ce dernier devient fixe (fixedb = true).
- Si un bloc est placé sur une pile spécifique (onb = -(p+1)), cette pile n'est plus libre (freep = false). La classe Implication de modelling a été utilisé pour ça.

1.3 Classe BlocksRegularConfigConstraint

La classe BlocksRegularConfigConstraint représente une contrainte de régularité pour les configurations du monde des blocs. Elle se sert des contraintes déjà présentes dans BlocksWorld et ajoute des règles supplémentaires pour garantir une disposition régulière des blocs. La méthode buildRegularyContraintes génère des contraintes supplémentaires en utilisant les indices des variables on.

1.4 Classe BlocksIncreasingConfigConstraint

La classe BlocksIncreasingConfigConstraint génère des contraintes de croissance pour les configurations. Elle s'assure que les blocs respectent une structure croissante, c'est-à-dire que certaines variables de position des blocs doivent être inférieures à celles des blocs suivants. La méthode buildIncreasingContraintes ajoute des contraintes de croissance sur les variables on du monde des blocs.

1.5 Classe DemoOfContraintes

La classe DemoOfContraintes sert de démonstration pour tester les contraintes de régularité et de croissance dans le monde des blocs. J'ai crée des configurations avec 8 blocs et 3 piles et vérifier si elles respectent les contraintes définies par les classes BlocksIncreasingConfigConstraint (pour les contraintes de croissance) et BlocksRegularConfigConstraint (pour les contraintes de régularité).

2 Planification

Cette partie consistait à implémenter toutes les actions du mondes des blocs. D'implémenter des heuristiques et de lancer des planificateurs pour voir les résultats. Pour ce faire je me suis servi des classes déjà implémentées dans le Tp sur la planification.

2.1 Classe BlocksAction

La classe BlocksAction génère toutes les actions possibles dans le monde des blocs. Elle crée des actions permettant de déplacer des blocs entre différentes positions, comme de blocs à blocs ou de blocs à des piles.

- buildActionToMoveB1FromB2ToB3 crée l'action pour déplacer un bloc b1 de b2 à b3. Cela implique de vérifier les préconditions (par exemple, b1 doit être sur b2) et les effets (par exemple, b1 sera déplacé sur b3).
- buildActionToMoveB1FromB2ToP crée l'action pour déplacer un bloc b1 de b2 vers une pile p. Cette action n'est possible que si la pile est libre et aura comme effet fixedp = true.
- buildActionToMoveB1FromPToB2 crée l'action pour déplacer un bloc b1 d'une pile p à un bloc b2. On aura comme préconditions b1 est sur la pile p, fixedb2 = false et aura comme effet freep = true et fixedB2 = false.
- buildActionToMoveBFromP1ToP2 crée l'action pour déplacer un bloc b d'une pile p1 à une autre pile p2. On aura comme préconditions fixedb = false, freep2 = true et on aura comme effet freep1 = true, freep2 = false.

Dans cette partie j'implémente également deux heuristiques, à savoir une heuristique estimant le nombre de blocs qui ne sont pas dans la position correcte et une heuristique qui évalue la distance totale que chaque bloc doit parcourir pour atteindre sa position finale.

2.2 Classe DemoPlanification

La classe DemoPlanification simule la planification dans un monde de blocs de 4 blocs et 3 piles à l'aide des algos que nous avons implémentés durant le Tp de la planification qui sont

DFS, BFS, Djikstra et A*.Je définis un état initial et un état final, puis utilise les planificateurs pour générer des plans. Chaque planificateur est chronométré et les résultats sont affichés, y compris le nombre de nœuds visités, la longueur du plan et le temps de calcul. Enfin, les résultats sont visualisés graphiquement en utilisant la classe Vue. Certains plans étant plus long, j'ai crée dans une classe (DisplayPlanActions) un thread pour pouvoir visualiser tous les plans au moment.

3 Problèmes de satisfaction de contraintes

Cette partie consistait à créer une instance du monde des blocs en spécifiant un nombre de blocs et de piles en utilisant les contraintes déjà implémentées et de lancer tous les solveurs de contraintes en affichant leur temps de calcul et la solution trouvée s'il en existe une.

3.1 Classe DemoCSP

Dans cette classe DemoCSP j'initialise un état de 7 blocs et de 3 piles et crée ensuite les différentes contraintes pour les solveurs.Les résultats sont affichés visuellement à l'aide de la classe Vue, permettant ainsi de comparer les performances des différentes approches de résolution. Le temps de calcul de chaque solveur pour chaque type de contraintes est affiché en secondes.

4 Extraction de connaissances

Pour cette partie consiste à extraire des motifs et des règles d'association dans une base de données. Pour cela trois types de variables sont demandées :

- pour chaque couple de blocs différents b, b', une variable onbb', prenant la valeur true lorsque le bloc b est directement sur le bloc b' et false sinon
- pour chaque bloc b et pour chaque pile p, une variable on-tablebp prenant la valeur true lorsque le bloc b est sur la table dans la pile p et false sinon
- pour chaque bloc b et pour chaque pile p, les variables fixedb et freep, avec la même signification que dans la partie 1

4.1 Classe BlocksExtraction

La classe BlocksExtraction permet de générer et gérer un ensemble de variables booléennes représentant les relations entre les blocs et les piles. Pour créer l'ensemble des variables de cette partie, j'ai ajouté un attribut de type BlocksVariable à la classe pour pouvoir créer l'ensemble des variables comme çà a été fait dans la classe BlocksVariable en utilisant les variables déjà disponible. La méthode buildOnBlock crée des variables booléennes pour indiquer si un bloc i est posé sur un autre bloc j (par exemple, on_{ij}).

4.2 Classe DemoExtraction

La classe DemoExtraction illustre l'extraction de motifs fréquents et de règles d'association dans le monde des blocs. J'initiale d'abord un monde de bloc de 8 blocs et 3 pile.Ensuite génère une base de données de 10000 transactions avec une confiance minimale de 95/100 et une fréquence minimale de 2/3. L'algo Apriori permet d'extraire les itemsets fréquents, puis

l'algo de brute force permet d'extraire des règles d'association. Les résultats sont affichés, montrant les itemsets fréquents et les règles d'association avec leur fréquence et confiance.

5 Conclusion

En conclusion, ce fil rouge sur le monde des blocs ainsi que l'ensemble des Tps passés m'a permis d'acquérir une compréhension approfondie des concepts clés dans le domaine de l'aide à la décision et intelligence artificielle, notamment dans les domaines de la représentation des variables et des contraintes, de la planification, de la programmation par contraintes et de l'extraction de connaissances.
Ces travaux m'ont permis de développer une vision globale des problématiques liées à l'aide à la décision et l'intelligence artificielle.

6 Commandes de compilation et d'exécution:

Pour compiler et exécuter les classes exécutables, j'ai fais un script.sh qui permet de lancer les classes exécutables et voir les démonstrations. Pour ce faire, ouvrir un terminal dans le répertoire du projet, lancer la commande ./script.sh et suivre les instructions sur le terminal.