UNIVERSITÉ
CAEN
NORMANDIE

Aide à la décision en Intelligence Artificielle

Document explicatif - Fil Rouge

LE MONDE DES BLOCS

Binôme

KACED	Louheb	22111744
DJEHA	Wassim	22208244

• Introduction :

Le projet "Fil Rouge" que nous présentons ici est une implémentation complète du monde des blocs (BlocksWorld), un exemple classique utilisé en Intelligence Artificielle. Réalisé en Java, ce travail englobe plusieurs aspects cruciaux, de la modélisation des configurations du monde des blocs à l'extraction de connaissances à partir de bases de données. Chaque phase du projet est accompagnée d'une classe exécutable dédiée, offrant ainsi une exploration approfondie du monde des blocs, tant du point de vue de la modélisation que de la planification des actions. Nous avons également abordé la vérification de la satisfaction des contraintes à l'aide de solveurs, tout en exploitant la librairie blocksworld pour représenter graphiquement les configurations.

• Les parties du projet :

1. Modélisation :

Pour la modélisation, nous avons choisi d'utiliser les classes "Variable" et "BooleanVariable" que nous avons déjà implémentées lors des premiers TPs afin de créer et représenter nos différentes variables dans le monde des blocs.

Pour les variables "on", nous utilisons la classe "Variable" car elle prend un nom de variable que nous représentons par "on + numéro du bloc" et un domaine qui prend toutes les valeurs de nos blocs et piles, représentées par des entiers (strictement négatifs pour les piles et positifs pour les blocs).

Pour les variables "fixed" et "free", la classe "BooleanVariable" est utilisée car ces deux variables prennent comme valeur soit "true" ou "false", et la classe "BooleanVariable" permet par défaut d'initialiser ce domaine lors de la création d'une variable. La classe qui permet de récupérer toutes ces variables est "BlocksWorld".

En ce qui concerne la représentation des contraintes, nous avons également utilisé les classes déjà implémentées dans nos TPs telles que "Constraint", "DifferenceConstraint", "Implication" et "UnaryConstraint".

La classe "BlocksWorldConstraints" permet de créer trois types de contraintes : des contraintes de différence entre deux positions de blocs, des contraintes d'implication

liant la position d'un bloc à l'état de sa fixation "fixedb" et à la libération de sa pile "freep".

La classe "BlocksWorldRegular" nous permet de générer des contraintes assurant que nous avons le même écart entre les blocs qui sont posés sur chaque pile. La classe "Croissante" génère des contraintes garantissant que notre configuration est croissante, c'est-à-dire qu'un bloc ne peut être positionné que sur un bloc de numéro plus petit (ou directement sur la table).

Enfin, la classe "DemoModelling" est notre classe exécutable où nous avons créé des exemples de configurations et vérifié si elles respectent nos contraintes de régularité et de croissance.

2. Planification :

Pour la partie planification, nous avons commencé par créer une classe nous permettant d'avoir toutes les actions possibles sur un monde des blocs en lui fournissant le nombre de blocs et de piles, ainsi qu'une référence vers notre BlocksWorld pour éviter de le recréer à chaque fois dans le constructeur. Nous utilisons nos variables pour créer des actions.

Chaque action est une instance de notre classe "BasicAction" que nous avons implémentée pendant les TP.

Pour créer l'action, nous établissons une map de préconditions qui doivent être respectées pour l'application de l'action, ainsi qu'une map d'effets qui représente les changements qui seront appliqués sur notre monde après l'exécution de l'action.

En ce qui concerne notre monde, il est représenté par une "map<Variable,Object>". Lors de la création de notre monde initial et de notre monde final pour tester les différents planificateurs, nous parcourons toutes nos variables et attribuons une valeur à chacune selon la position où nous souhaitons la placer.

Dans notre classe "DemoPlanning", nous créons un état initial et un état final, puis nous lançons tous nos planificateurs tels que "DFS", "BFS", "Dijkstra" et "Astar". Nous affichons le plan trouvé pour chacun avec le nombre de nœuds explorés et le temps d'exécution.

Cette partie inclut aussi deux heuristiques significatives. La première heuristique repose sur le calcul du nombre de blocs mal placés à chaque étape du

processus de planification. Cette approche vise à identifier et à minimiser les erreurs de placement, contribuant ainsi à des solutions plus précises.

La deuxième heuristique adoptée est axée sur l'estimation de la distance totale que chaque bloc doit parcourir pour atteindre sa position cible. En évaluant la distance des déplacements nécessaires, cette heuristique offre des informations cruciales pour guider la planification vers des solutions efficaces. Ces ajouts renforcent notre capacité à explorer et à évaluer les plans de manière plus nuancée, contribuant ainsi à des résultats de planification plus optimisés.

En ce qui concerne la visualisation graphique des états et des actions, nous utilisons la bibliothèque fournie "blocksworld.jar". Nous avons implémenté la méthode "makestate" qui permet de créer un état du monde pour la visualisation graphique

Dans notre interface graphique, nous affichons le plan trouvé par le planificateur Astar ; il est possible de visualiser d'autres plans en changeant la variable que nous passons dans notre boucle, par exemple, pour visualiser le plan de Dijkstra, nous utilisons dans la boucle "resultatDijkstra".

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 qui permet de modéliser les relations entre les différentes entités du problème. Trois classes exécutables, DemoReguliere, DemoCroissante, et DemoCroissanteReguliere, représentent différentes configurations du problème en utilisant des ensembles de contraintes distincts pour guider les solveurs vers des solutions efficaces.

La classe DemoReguliere constitue le point de départ du processus, créant une instance de BlocksWorld et récupérant l'ensemble des variables décrivant l'état du monde des blocs. Ensuite, elle instancie les classes BlocksWorldConstraints et BlockWorldRegular pour obtenir les ensembles de contraintes de base et régulières, respectivement. Ces contraintes servent de fondement aux différents solveurs utilisés pour explorer les solutions possibles.

La classe DemoCroissante suit le même principe que DemoReguliere, mais introduit la classe BlockWorldCroissant pour implémenter des contraintes d'une configuration croissante en plus des contraintes de base.

Concernant la classe DemoCroissanteReguliere, les contraintes de base, régulières, et croissantes sont implémentées dans la même configuration, offrant ainsi une perspective complète sur la résolution de problèmes plus complexe, incorporant des exigences régulières tout en guidant la planification vers une croissance spécifique.

Chaque démo utilise différents solveurs, tels que BacktrackSolver, MACSolver, et des versions améliorées avec des heuristiques spécifiques (HeuristicMACSolver), exploitant les contraintes pour trouver des solutions optimales tout en mesurant le temps de calcul nécessaire à chaque approche.

4. Extractions des connaissance :

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 en fonction des relations entre les blocs dans un monde donné. Les relations prises en compte incluent la présence de blocs "on" d'un bloc sur un autre et la position "on-table" d'un bloc par rapport à la table.

Pour créer ces variables, nous utilisons les informations fournies par le modèle du monde des blocs (`BlocksWorld`). En parcourant les différentes configurations possibles, la classe `BooleanBlocksWorld` génère des variables booléennes représentant les relations "on" et "on-table" entre les blocs et la table.

Ces variables booléennes sont ensuite utilisées pour construire une base de données booléenne (`BooleanDatabase`).

Pour n instances , on récupère un état généré grâce à la méthode getState() se trouvant dans la librairie fournie “bwgenerator” , puis on ajoute les “BooleanVariable” correspondant à cet état dans notre base de données.

Donc, cette base de données est remplie avec des instances générées du monde des blocs, où chaque instance est représentée par un ensemble de ces variables booléennes.

Finalement, nous avons appliqué deux algorithmes distincts pour extraire des connaissances à partir de cette base de données. Tout d'abord, l'algorithme Apriori a été utilisé pour découvrir des motifs fréquents, révélant ainsi des configurations récurrentes dans le monde des blocs. Ensuite, l'algorithme de force brute a été

employé pour extraire des règles d'association, mettant en évidence des relations significatives entre les variables.

L'exécution du programme, initiée par la classe "**DemoDataMining**" , offre une vue d'ensemble des motifs fréquents et des règles d'association dans le contexte du modèle du monde des blocs, ouvrant ainsi des perspectives intéressantes pour l'analyse des relations au sein de ce monde simulé.

- ## 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**
- ## Commandes de compilation et d'exécution:

Lorsque vous souhaitez travailler sur notre projet, suivez ces étapes pour la compilation et l'exécution du code.

1. Tout d'abord, ouvrez un terminal dans le répertoire source /**src** de votre projet.

2. Ensuite, pour compiler toutes les classes nécessaires et prendre en compte les bibliothèques externes **blocksworld.ja**r et **bwgenerator.jar**, utilisez la commande suivante:

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

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 :

4.

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