Rapport fil rouge monde des blocs

Auteurs: KOTIN Mahudo(22211719)
ADOUMA Hassan(21901741)

Introduction

Ce rapport décrit le choix de nos implémentations ainsi que les démonstrations réalisées dans le projet fil rouge : monde des blocs.

Partie I: Variables et contraintes

L'objectif de cette partie était de représenter le monde des blocs sous forme de variables et d'implémenter les contraintes liées à ce monde.

La classe `BWVariables` du package `blocksworld` permet de créer toutes les variables correspondant au monde des blocs. Son constructeur prend en paramètres le nombre de blocs et le nombre de piles pour un monde donné.

Trois types de variables sont créées telles que décrites dans le sujet (onb, freep, fixedb). Le principe est approximativement le même pour chaque type. On itère sur le nombre de blocs ou le nombre de piles selon le type de variable. On crée une instance de la classe `Variable` ou `BooleanVariable` en fonction du domaine de la variable, que l'on stocke ensuite dans un `Set<Variable>`. Le nom des variables est sous le format : "on" — "fixed" — "free" + index de bloc — pile. Ex: (on0, free0, fixed4).

NB: Bien que les valeurs de piles soient représentées par des valeurs négatives dans le domaine des variables "on", nous avons choisi de conserver leur index réel dans la nomenclature des variables free, d'autant plus que le sujet n'y était

pas contre. Ainsi la valeur de la pile 0 dans le domaine d'une variable onb sera -1 alors que la variable free liée à cette pile sera free0 et pas free-1.

Pour ce qui est des contraintes, elles sont créées dans la classe `BlocksWorld` du même package. Son constructeur prend en paramètres le nombre de blocs et le nombre de piles, et elle possède un attribut de type `BWVariables` pour obtenir les variables à utiliser dans les contraintes. Trois types de contraintes sont demandées:

- $onb \neq onb'$
- $onb = b' \Rightarrow fixedb' = \text{true}$
- $onb = -p - 1 \Rightarrow freep = \text{false}$ avec b et b' deux blocs différents, p une pile.

L'implémentation consiste à créer pour chaque couple de blocs différents ou de (bloc, pile) selon le type de contrainte une instance de `DiffrenceConstraint` ou de `ImplicationConstraint`.

Des contraintes additionnelles (contraintes régulières et croissantes) sont implémentées pour éviter les configurations circulaires du monde des blocs.

La classe `BWRegularyConstraints` permet d'obtenir en plus des contraintes basiques citées plus haut toutes les contraintes régulières. Ces dernières sont des contraintes de type implication (`ImplicationConstraint`) sous la forme: $onb = b' \Rightarrow onb' \in \{-1, \ldots, -nbp, b' - (b - b')\}$ avec b et b' deux blocs différents et nbp le nombre total de piles.

De même, la classe `BWAscContraints` permet d'obtenir les contraintes croissantes en plus des contraintes basiques. Ici il s'agit de contrainte unaire (`UnaryConstraint`) sous la forme: $onb \in \{-1, \ldots, -nbp, 0, b - 1\}$ avec b un bloc et nbp le nombre total de piles.

Démo

La démo de cette partie est réalisée dans la classe `MainConstraints` du package `main`. Ici, nous avons créé un monde de blocs de 8 blocs et trois piles avec toutes les variables et correspondantes. Ensuite, nous avons créé 4 états différents pour vérifier la satisfaction des contraintes basiques, régulières, croissantes et enfin régulières et croissantes. Par ailleurs, la méthode `getState` de la classe `BWVariables` permet de convertir un tableau 2D d'entiers (où la deuxième dimension représente les piles et les entiers les blocs) en une instance du monde des blocs d'un état donné.

Partie II: Planification

Dans cette partie, la classe `BWActions` du package `blocksworld` permet de créer toutes les actions possibles énoncées dans le sujet. Quatre types d'actions sont demandées. Pour chacune de ces actions, nous obtenons les triplets de

blocs et/ou de piles impliqués dans l'action. Ensuite, nous récupérons les variables liées à ces blocs et/ou piles utiles pour la création de l'action grâce à la classe `BWVaraibles` qui est une composition de `BWActions`. La méthode `helperbuildMap` permet après de créer les préconditions et les effets de chaque action en associant chaque variable à une valeur. Pour finir, nous créons une instance de `BasciAction` avec les effets et les préconditions que nous ajoutons à un `Set<BasicAction>`.

Nous avons implémenté ici deux heuristiques pour la planification: le nombre de blocs sur une mauvaise position et le nombre de bloc en bas de la pile sur une mauvaise position. Chacune d'entre elles implémente l'interface `Heuristic`.

Démo

La démo de cette partie se trouve dans la classe `MainPlanner` du package `main`. Ici, nous avons créé un monde des blocs avec 5 blocs et 3 piles, puis un état initial et un état final. Ensuite, nous avons créé une instance de chaque planificateur puis récupéré le plan de chacun d'eux. Enfin, nous affichons les plans grâce à la méthode `displayPlan` de la classe `View` du package `view`.

Afin de visualiser tous les plans simultanément (parce que certains plans étant plus longs que d'autres, attendre leur fin avant de montrer le prochain était long), nous avons créé un thread dans la classe `DisplayPlanThread` du package `blocksworld`. Pour chaque plan, on affiche dans le terminal le nombre de nœuds visités, la taille du plan, et la durée de calcul.

Partie III: Problèmes de satisfaction de contraintes

Démo

La démo de cette partie se trouve dans la classe `MainCp` du package `main`. Ici, nous avons créé un monde avec 5 blocs et deux piles. Nous avons ensuite lancé tous les solveurs avec les contraintes régulières d'abord, puis avec les contraintes croissantes, puis avec les contraintes régulières et croissantes et affiché leurs solutions grâce à la classe `View`. Dans le terminal, on affiche le temps de calcul de chaque solveur pour chaque type de contraintes.

Partie IV: Extraction de connaissances

L'extraction de connaissances est gérée dans la classe `BWDatamining`, du package `blocksworld`. Ici, trois types de variables booléennes sont demandées pour représenter un état ($on_{bb'}$, $ontable_{bp}$, $fixed_b$). Pour créer ces variables, nous avons ajouté comme composition de la classe `BWDatamining` la classe `BWVariables`, puis nous nous sommes inspirés des mêmes méthodes que dans la partie I, bien sûr en utilisant les variables déjà disponibles. Ensuite, la méthode `generateTransactionsVariables` nous permet d'obtenir les variables liées à

une transaction qui ont une valeur `true`, et la méthode `generateDatabase` génère une base de données pour un nombre donné de transactions.

Démo

La démo de cette partie se trouve dans la classe `MainDatamining` du package `main`. Ici, on génère une base de données d'états de monde des blocs, puis on extrait les items fréquents avec une fréquence minimale de $\frac{2}{3}$ et les règles avec une confiance minimale de $\frac{95}{100}$.

Conclusion

En résumé, ce projet, centré sur l'intelligence artificielle dans le contexte du monde des blocs, nous a permis de mettre en œuvre les principes clés de variables et contraintes, planification, problèmes de satisfaction de contraintes et datamining vus en cours. L'application cohérente de ces concepts a donné naissance à un programme robuste capable de résoudre des défis complexes.