UNIVERSITÉ CAEN NORMANDIE

UFR des Sciences

Université de Caen Normandie
Département mathématiques-informatique
2024–2025
Bruno Zanuttini

L3 informatique

TP fil rouge. Le monde des blocs

Unité YSINL5D2 : Aide à la décision et intelligence artificielle

---

Le monde des blocs (*BlocksWorld*) est un exemple couramment utilisé en intelligence artificielle, en particulier pour illustrer ou tester des algorithmes de planification. Dans un monde des blocs, on a n blocs, par exemple numérotés de 0 à $n-1$, et on s'intéresse à leurs empilements. Par exemple, on peut chercher un ensemble de déplacements de blocs, un par un, permettant de passer de la configuration de gauche à celle de droite sur la figure 1.

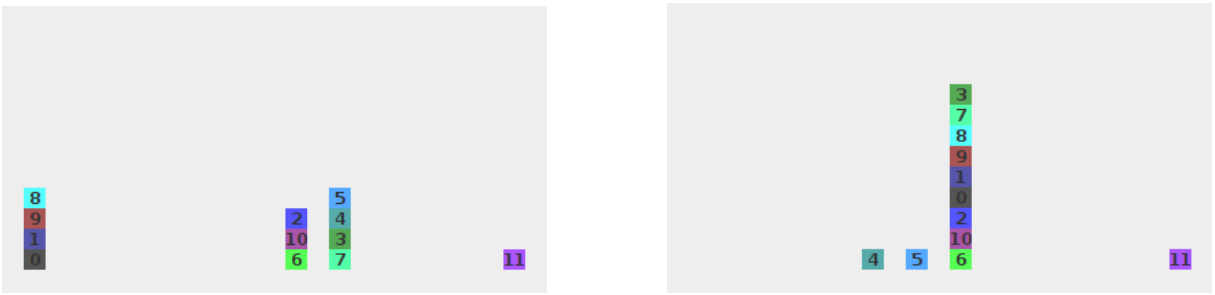

FIGURE 1 – Deux configurations d'un monde des blocs

De façon générale, on a donc n blocs organisés en m piles (certaines pouvant être vides). On peut décrire une configuration en spécifiant les couples (b, b') tels que le bloc b est posé sur le bloc b', et les couples (b, p) tels que le bloc b est le plus bas de la pile p (on dit aussi qu'il est « sur la table » dans la pile p). Enfin, même si c'est redondant, on peut spécifier les blocs « indéplaçables », c'est-à-dire les blocs b qui sont sous au moins un autre bloc, et les piles « libres », c'est-à-dire les piles qui ne contiennent aucun bloc.

Une librairie est fournie pour faciliter la visualisation des configurations (voir la partie 5). Par ailleurs, par souci de lisibilité, créer toutes les classes correspondant au monde des blocs dans un *package* dédié, nommé par exemple `blocksworld`.

1 Variables et contraintes

Dans ce fil rouge, on propose d'utiliser trois ensembles de variables pour représenter les configurations d'un monde de n blocs et m piles :

- pour chaque bloc b, une variable on_b, prenant pour valeur soit un autre bloc b' (signifiant que b est posé sur b'), soit une pile p (signifiant que b est sur la table dans la pile p) ; la variable on_b pourra par exemple avoir pour domaine l'ensemble d'entiers $\{-p, \ldots, -1, 0, \ldots, n-1\} \setminus \{b\}$, les entiers strictements négatifs codant les p piles, et les entiers positifs ou nuls, les blocs (on exclut b du domaine car un bloc ne peut pas être sur lui-même) ;
- pour chaque bloc b, une variable booléenne fixed_b, prenant la valeur `true` lorsque le bloc est indéplaçable ;
- pour chaque pile p, une variable booléenne free_p, prenant la valeur `true` lorsque la pile est libre.

Pour ce qui est des contraintes, on se restreint à des contraintes binaires ; les contraintes suivantes sont respectées par toute configuration valide d'un monde des blocs :

— pour chaque couple de blocs différents $\{b, b'\}$, les variables on_b et $\text{on}_{b'}$ ne peuvent pas prendre la même valeur (cela signifierait que les deux blocs sont posés sur un même troisième bloc, ou qu'ils sont sur la table dans la même pile) ;
— pour chaque couple de blocs $\{b, b'\}$, si la variable on_b a la valeur b', alors la variable $\text{fixed}_{b'}$ doit avoir la valeur `true` (car s'il y a un bloc juste au-dessus de b', celui-ci est indéplaçable) ;
— pour chaque bloc b et pour chaque pile p, si la variable on_b a la valeur $-(p+1)$ (c'est-à-dire la valeur représentant la pile p), alors la variable free_p doit avoir la valeur `false` (car s'il y a un bloc dans p, celle-ci n'est pas libre).

Remarquons que ces contraintes ne suffisent pas à définir une configuration valide : elles n'empêchent pas une configuration circulaire (le bloc 1 sur le bloc 2, le bloc 2 sur le bloc 3, le bloc 3 sur le bloc 1, et les trois blocs indéplaçables). Par ailleurs, elles ne forcent pas la variable fixed_b à être à `false` s'il n'y a aucun bloc au-dessus de b ; un bloc pourra donc être codé comme « indéplaçable » même s'il est en haut d'une pile. Enfin, elles ne forcent pas la variable free_p à être à `true` s'il n'y a aucun bloc dans p.

Dans la suite, nous nous assurerons (par des contraintes additionnelles, ou par construction des états initiaux et des actions) que toutes les configurations manipulées sont au minimum non circulaires.

Exercice 1. *Créer une classe que l'on pourra instancier en spécifiant un nombre de blocs et un nombre de piles, et à laquelle on pourra demander l'ensemble de variables correspondant.*

Exercice 2. *Implémenter le monde des blocs avec la modélisation ci-dessus, en créant une nouvelle classe que l'on pourra instancier avec un nombre de blocs et un nombre de piles, et à laquelle on pourra demander l'ensemble de toutes les contraintes spécifiées ci-dessus.*

Pour assurer que les configurations manipulées ne sont pas circulaires, on peut s'intéresser à des mondes des blocs particuliers. On appellera *régulière* une configuration dans laquelle, au sein de chaque pile, l'écart entre les numéros de deux blocs consécutifs est toujours le même, par exemple une configuration avec les piles $(1, 3, 5, 7)$ (écart de 2), $(8, 4, 0)$ (écart de -4), $(2, 9)$ et (6).

Exercice 3. *Proposer un ensemble de contraintes de type* `Implication`*, permettant d'assurer qu'une configuration est régulière. Créer une classe avec une méthode permettant d'obtenir l'ensemble de ces contraintes pour un monde des blocs donné, tel que défini par une instance de la classe de l'exercice 2.*

Une autre manière de s'assurer que les configurations manipulées ne sont pas circulaires est de les contraindre à être *croissantes*, c'est-à-dire qu'un bloc ne peut être positionné que sur un bloc de numéro plus petit (ou directement sur la table).

Exercice 4. *Proposer un ensemble de contraintes permettant d'assurer qu'une configuration est croissante. Créer une classe permettant d'obtenir l'ensemble de ces contraintes, sur le même modèle que pour l'exercice 3.*

De façon optionnelle, on peut également s'intéresser à d'autres types de contraintes (empêchant ou non à elles seules les configurations circulaires). On peut par exemple considérer des configurations n'ayant qu'une pile, des configurations qui alternent piles vides et piles non vides, etc.

Exercice 5. *Dans une classe exécutable, créer un petit exemple, quelques configurations, et afficher pour chacune un message indiquant si elle satisfait toutes les contraintes d'une configuration régulière et/ou croissante. Faire de même une démonstration des autres contraintes implémentées, s'il y en a.*

2 Planification

Pour la planification, on s'intéressera à des problèmes constitués d'un état initial totalement spécifié, et respectant la définition « intuitive » des variables fixed_b et free_p : pour tout bloc b, fixed_b sera à `true` si et seulement s'il y a un bloc sur b (et donc à `false` s'il n'y en a pas), et pour toute pile p, free_p sera à `true` si et seulement s'il n'y a pas de bloc dans p (et donc à `false` s'il y en a un).

Les actions à considérer sont celles consistant à déplacer un bloc vers une autre position, à condition que le bloc déplacé et la position de destination soient libres ; il y a donc quatre types d'actions :

1. déplacer un bloc b du dessus d'un bloc b' vers le dessus d'un bloc b'',
2. déplacer un bloc b du dessus d'un bloc b' vers une pile vide p,
3. déplacer un bloc b du dessous d'une pile p vers le dessus d'un bloc b',
4. déplacer un bloc b du dessous d'une pile p vers une pile vide p'.

Les contraintes impliquent donc que si l'on déplace un bloc b depuis le dessous d'une pile p, cette pile était constituée seulement de b.

Toutes ces actions devront donc être munies de trois préconditions atomiques :

— le bloc b doit être à la place correspondant à l'action (par exemple, on_b doit être égal à b' pour les actions du type 1),
— le bloc b doit être déplaçable (fixed_b doit être à `false`),
— la destination doit être libre ($\text{fixed}_{b''}$ doit être à `false` pour les actions du type 1, free_p doit être à `true` pour les actions du type 2, etc.).

Pour ce qui est des effets, à nouveau chaque action devra être munie de trois effets atomiques :

— la nouvelle position du bloc b doit être spécifiée (par exemple, on_b doit passer à la valeur b'' pour les actions du type 1),
— le nouveau statut de la position initiale doit être spécifié (par exemple, $\text{fixed}_{b'}$ doit passer à `false` pour les actions du type 1, et free_p doit passer à `true` pour les actions du type 3),
— le nouveau statut de la position de destination doit être spécifié (par exemple, $\text{fixed}_{b'}$ doit passer à `true` pour les actions du type 1).

Exercice 6. *Écrire une classe que l'on pourra instancier avec un nombre de blocs et un nombre de piles, et à laquelle on pourra demander toutes les actions spécifiées ci-dessus (pour toutes les combinaisons de blocs et de piles).*

Exercice 7. *Implémenter au moins deux heuristiques admissibles pour le monde des blocs.*

Exercice 8. *Dans une classe exécutable, créer au moins un exemple d'état initial (totalement spécifié), au moins un exemple d'état final (pouvant être partiellement spécifié), et lancer tous les planificateurs implémentés en affichant leur temps de calcul, le nombre de nœuds qu'ils ont explorés, et le plan trouvé (s'il en existe un).*
Indication : *On pourra s'aider d'une méthode prenant en argument une liste de piles (chacune représentée comme une liste de blocs), et retournant l'instanciation correspondant à cette liste et respectant la définition intuitive des variables* fixed_b *et* free_p*.*

Les plans trouvés pourront être simulés en utilisant la librairie fournie (voir la partie 5).

3 Problèmes de satisfaction de contraintes

Exercice 9. *Dans une classe exécutable, créer au moins une instance du monde des blocs (en spécifiant le nombre de blocs et le nombre de piles), en utilisant les contraintes demandant une configuration* régulière*, et lancer tous les solveurs de contraintes implémentés en affichant leur temps de calcul et la solution trouvée (s'il en existe une).*

Exercice 10. *Même exercice avec les contraintes demandant une configuration* croissante*, puis avec celles demandant une configuration régulière* et *croissante, puis avec d'autres combinaisons exploitant les contraintes additionnelles implémentées dans la partie 1, s'il y en a.*

On pourra afficher les solutions trouvées en utilisant la librairie fournie (voir la partie 5).

4 Extraction de connaissances

Pour l'extraction de connaissances, on s'intéressera à des bases de données d'états du monde des blocs, bases desquelles on cherchera à extraire des motifs et des règles d'association. Pour simplifier l'utilisation des algorithmes, on traduira ces bases avec des variables booléennes, issues de la « propositionnalisation » des variables proposées dans la partie 1 :

— pour chaque couple de blocs différents $\{b, b'\}$, une variable $\text{on}_{b,b'}$, 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 $\text{on-table}_{b,p}$, 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 fixed_b et free_p, avec la même signification que dans la partie 1.

Pour la génération de bases de données, on pourra utiliser la librairie `bwgenerator` fournie, dont la documentation peut être consultée à l'URL `https://zanuttini.users.greyc.fr/code/bwgenerator/`. Dans cette librairie, un état du monde des blocs est représenté par une liste de piles, chaque pile étant représentée par une liste d'entiers : l'entier à l'indice 0 représente le bloc du bas de la pile, l'entier à l'indice 1, le bloc posé directement sur lui, etc.

Exercice 11. *Écrire une classe qui pourra être instanciée avec un nombre de blocs et un nombre de piles, et à laquelle on pourra demander l'ensemble de toutes les variables booléennes (de type* `BooleanVariable`*) correspondant à ces paramètres, ainsi que l'instance (de type* `Set<BooleanVariable>`*) correspondant à un état donné (de type* `List<List<Integer>>`*). Pour ce dernier point, on pourra instancier les variables* fixed_b *et* free_p *avec leur signification intuitive (*fixed_b *à* `true` *pour tous les blocs sauf ceux en haut d'une pile, et* free_p *à* `true` *exactement pour les piles vides).*

Pour l'utilisation de la librairie, on pourra s'inspirer du code suivant, qui crée une base de `n` instances :

```
BooleanDatabase db = new BooleanDatabase(...);
for (int i = 0; i < n; i++) {
  // Drawing a state at random
  List<List<Integer>> state = Demo.getState(random);
  // Converting state to instance
  Set<BooleanVariable> instance = ...;
  // Adding state to database
  db.add(instance);
}
```

Exercice 12. *Dans une classe exécutable, créer une base de données (de type* `BooleanDatabase`*) d'états du monde des blocs, puis lancer une extraction de motifs et de règles d'association, et afficher les résultats.* Indication : *En utilisant la librairie fournie, les paramètres suivants fournissent des résultats intéressants : base de données de* 10 000 *instances, extraction de motifs de fréquence au moins* 2/3*, et de règles d'association de fréquence au moins* 2/3 *et de confiance au moins* 95/100*.*

5 Librairie

La librairie `blocksworld` fournie permet essentiellement de

- représenter des configurations, via le type `BWState` ; la représentation ne tient pas compte des numéros des piles, ni du fait que les blocs soient déplaçables ou non, ni du fait que les piles soient libres ou non ;
- construire des instances à partir des spécifications « le bloc b est au-dessus du bloc b' » ; implicitement, un bloc b pour lequel aucune telle spécification n'est fournie est placé sur la table ;
- représenter les configurations par des chaînes de caractères ou par des composants graphiques Swing (comme sur la figure 1).

La documentation de la librairie est disponible à l'URL `https://zanuttini.users.greyc.fr/code/blocksworld/`. À titre d'illustration, le code suivant permet d'afficher une représentation graphique (dans une `JFrame`) d'une instanciation des variables pour un monde de n blocs :

```
// Building state
BWStateBuilder<Integer> builder = BWStateBuilder.makeBuilder(n);
for (int b = 0; b < n; b++) {
  Variable onB = ...; // get instance of Variable for "on_b"
  int under = (int) instanciation.get(onB);
  if (under >= 0) { // if the value is a block (as opposed to a stack)
    builder.setOn(b, under);
  }
}
BWState<Integer> state = builder.getState();

// Displaying
BWIntegerGUI gui = new BWIntegerGUI(n);
JFrame frame = new JFrame(title);
frame.add(gui.getComponent(state));
frame.pack();
frame.setVisible(true);
```

Le code suivant permet de représenter l'exécution d'un plan (en supposant qu'`init` est l'état initial, de type `Map<Variable, Object>`, et que la méthode `makeBWState` produit une instance de `BWState` à la manière du code étiqueté « *Building state* » ci-dessus) :

```
BWIntegerGUI gui = new BWIntegerGUI(n);
JFrame frame = new JFrame(title);
BWState<Integer> bwState = makeBWState(init...);
BWComponent<Integer> component = gui.getComponent(bwState);
frame.add(component);
frame.pack();
frame.setVisible(true);

// Playing plan
for (Action a: plan) {
  try { Thread.sleep(1_000); }
  catch (InterruptedException e) { e.printStackTrace(); }
  state=a.successor(state);
  component.setState(makeBWState(state...));
}
System.out.println("Simulation of plan: done.");
```