Rapport Fil Rouge

Membres du groupe :

- Karim MEDJDOUB 22212237

Introduction :

Le monde des blocs (BlocksWorld) est un problème classique en intelligence artificielle, utilisé pour étudier et tester des algorithmes de planification, de satisfaction de contraintes et d'extraction de connaissances. Il consiste à manipuler un ensemble de blocs empilés sur plusieurs piles afin de passer d'une configuration initiale à une configuration finale en respectant des contraintes spécifiques.

Ce projet s'inscrit dans le cadre du cours d'Aide à la Décision et Intelligence Artificielle, et vise à explorer les différents aspects liés à la modélisation, la résolution et l'optimisation dans un tel environnement. À travers la création d'une modélisation efficace, la définition de contraintes, et la mise en œuvre d'algorithmes de planification et de résolution, ce projet offre une approche pratique pour mieux comprendre les concepts fondamentaux de l'IA.

Dans ce rapport, nous décrivons d'abord la modélisation du monde des blocs et les contraintes associées. Ensuite, nous présentons les algorithmes utilisés pour la planification et les heuristiques développées. Enfin, nous discutons des résultats obtenus et des perspectives d'amélioration.

I. Modélisation

une instance du monde des blocs représente une configuration de n blocs et de m piles, chaque bloc portera un identifiant qui prendra sa valeur dans {0, …, n-1} et chaque pile portera également un identifiant qui prendra sa valeur dans {-m, … , -1}.
Une instance pourra être décrite en utilisant 3 types de variables :

- **Variables On :** chaque bloc b possède une variable On_b qui prend ses valeurs dans {-m,…,0,...n-1}\{b}, cette variable représente le support du bloc b. Ce type sera représenté par la classe **`OnVariable`**.
  exemples :
  $On_7 = 3$ signifie que le bloc d'identifiant 7 est posé sur le bloc d'identifiant 3.
  $On_0 = -2$ signifie que le bloc d'identifiant 0 est posé sur la pile d'identifiant -2 (on peut aussi dire sur la table).

- **Variables Fixed :** chaque bloc b possède une variable booléenne $Fixed_b$ qui représente l'état de fixation du bloc (s'il peut se déplacer ou pas). Un bloc sera fixé s'il est support d'un autre bloc. Ce type sera représenté par la classe **`FixedVariable`**.
  exemples :
  $Fixed_4$ = false signifie que le bloc d'identifiant 4 n'est pas support d'un autre bloc, et donc qu'il peut se déplacer.
  $Fixes_6$ = true signifie que le bloc d'identifiant 5 est support d'un autre bloc, et donc qu'il ne peut pas se déplacer.

- **Variables Free :** chaque pile p possède une variable booléenne $Free_p$ qui représente l'état de liberté d'une pile (s'il y a un bloc posé sur cette pile). Une pile sera donc libre si aucun bloc n'est posé dessus. Ce type sera représenté par la classe **`FreeVariable`**.
  exemple :
  $Free_{-1}$ = true signifie que la pile d'identifiant -1 est libre, et donc qu'il n'y a aucun bloc dessus.
  $Free_{-4}$ = false signifie que la pile d'identifiant -4 n'est pas libre, et donc qu'il y a un bloc posé dessus.

Une instance du monde des blocs doit respecter un certain nombre de contraintes basiques pour être valide, ces dernières sont au nombre de 3 et sont décrites comme suit :

- **Non-Chevauchement :** pour chaque couple de blocs {b, b'}, On_b et $On_{b'}$ ne peuvent pas avoir la même valeur (ce qui signifierait que b et b' sont posés sur le même bloc ou sur la même pile). Ce type sera représenté par la classe **`NonOverlappingConstraint`**.
  Elle est traduite par la relation : $On_b \neq On_{b'}$

- **Fixation des supports :** pour chaque couple de blocs {b, b'}, si On_b a la valeur b' alors $Fixed_{b'}$ doit avoir la valeur true, donc si le bloc b est posé sur le bloc b' alors b' ne doit pas être déplaçable. Ce type sera représenté par la classe **`FixedSupportConstraint`**.
  Elle est traduite par la relation : $On_b = b' \Rightarrow Fixed_{b'} = true$

- **Occupation des piles :** pour chaque bloc b et chaque pile p, si On_b a la valeur p alors $Free_p$ doit avoir la valeur false, donc si le bloc b est posé sur la pile p alors p n'est pas libre. Ce type sera représenté par la classe **`PileOccupancyConstraint`**.
  Elle est traduite par la relation : $On_b = p \Rightarrow Free_p = false$

Ces contraintes basique n'étant pas suffisantes a définir une configuration valide (elles n'empêchent pas de configuration circulaire), nous y ajouterons les deux contraintes suivantes qui pourront être utilisées ensemble ou séparément pour assurer que la configuration n'est pas circulaire.

- **Écart constant :** cette contrainte s'assure que la configuration manipulée est régulière (au sein d'une même pile, l'écart des identifiants de deux blocs consécutifs est toujours le même), donc pour chaque couple de blocs {b, b'} si $On_b = b'$ et que $On_{b'} = b''$ alors l'écart entre les identifiants de b et b' doit être le même que l'écart entre les identifiants de b' et b''. Ce type sera représenté par la classe **`ConstantGapConstraint`**.
  Elle est traduite par la relation : $On_b = b' \wedge On_{b'} \geq 0 \Rightarrow b - b' = b' - On_{b'}$

- **Supériorité :** cette contrainte s'assure que la configuration manipulée est croissante (Un bloc ne peut être posé que sur un bloc d'identifiant plus petit), donc pour chaque bloc b, b doit être supérieur à On_b. Ce type sera représenté par la classe **`SuperiorityConstraint`**.
  Elle est traduite par la relation : $b > On_b$

Pour récupérer les ensembles de ces variables et contraintes pour un monde donné, nous utiliserons 4 fournisseurs aux quels on spécifiera le nombre de blocs n et le nombre de piles m souhaités et qui nous retourneront ces ensembles.

- **Fournisseur de variables :** nous retourne les $2n+m$ variables correspondant a la configuration (classe **`VariablesProvider`**).

- **Fournisseur de contraintes basiques** : nous retourne les $\frac{3n}{2}(n-1)+nm$ contraintes basiques correspondant à la configuration(classe **`GeneralConstraintsProvider`**).

- **Fournisseur de contraintes de régularité :** nous retourne les $n(n-1)$ contraintes de régularité correspondant à la configuration (classe **`RegularityConstraintsProvider`**).

- **Fournisseur de contraintes de croissance :** nous retourne les n contraintes de supériorité correspondant à la configuration (classe **`GrowingConstraintsProvider`**).

le fonctionnement de toutes ces classes est illustré dans le première démonstration (classe **`Demo1`**) qui s'utilise comme suit :

```
java blocksworld.demonstrations.Demo1 n m
```

avec :

- n : nombre de blocs.
- m : nombre de piles.

ainsi que dans la deuxième (classe **`Demo2`**) qui montre quelques exemples d'instanciations et vérifie si les configurations sont régulières et croissantes, et qui s'utilise comme suit :

```
java blocksworld.demonstrations.Demo2
```

II. Planification

Pour changer l'état d'une configuration du monde des blocs, on considérera les 4 types d'actions suivantes :

- déplacer un bloc b du dessus d'un bloc b' vers le dessus d'un bloc b'' représentée par la classe **`BlocToBlocMovement`**.
- déplacer un bloc du dessus d'un bloc b' vers une pile vide p représentée par la classe **`BlocToStackMovement`**.
- déplacer un bloc b du dessous d'une pile p vers le dessus d'un bloc b' représentée par la classe **`StackToBlocMovement`**.
- déplacer un bloc b du dessous d'une pile p vers une pile vide p' représentée parla classe **`StackToStackMovement`**.

Un fournisseur sera utilisé pour récupérer l'ensemble des $n(n+m-2)(n+m-1)$ actions possibles pour un monde de n blocs et m piles (classe **`MovementsProvider`**).

On utilisera les deux heuristiques qui portent sur le nombre de blocs mal placés suivantes :

- une heuristique dont l'estimation augmente de 1 pour chaque bloc mal placé (classe **`MissplacedHeuristic`**).
- une heuristique dont l'estimation augmente de 1 pour chaque bloc s'il est simplement mal placé, ou de 2 s'il est mal placé et a au moins un bloc au dessus de lui (classe **`BlockedMissplacedHeuristic`**).

La 3éme démonstration (classe **`Demo3`**) compare les performances des planificateurs (DFS, BSF, Dijkstra et A star)

```
java blocksworld.demonstrations.Demo3
```

et la 4éme (classe **`Demo4`**) simule le plan calculé par l'algorithme choisi

```
java blocksworld.demonstrations.Demo4 a
```

avec :

- a : algorithme de recherche (1 pour DFS, 2 pour BFS, 3 pour Dijkstra et 4 pour A*).

Remarque :
le nombre d'actions possibles pour une configuration d'un monde des blocs étant très grand (1820 actions à combiner à chaque étape de la recherche pour 10 blocs et 5 piles), on se heurte rapidement à des situations intraitables (énorme complexité en temps). Quelques approches que nous proposons pour y remédier serait par exemple d'utiliser des techniques de réduction des espaces combinatoires, paralléliser l'exploration, recherche multi-niveaux etc..

III. Résolution de problèmes de satisfaction de contraintes

Cette partie consiste a implémenter les solveurs vus en cours.
Cependant, nous avons modifié le solveur qui se base sur l'algorithme **Maintaining Arc Consistency**.

```
Fonction MAC(I, V, ED)
    // conditions d'arrêt de la récurrsivité
    si V = ∅ alors
        retourner I
    sinon
        // Réduction des domaines des variables par l'arc-cohérence
        si ¬ AC1(X, ED, C) alors
            retourner Nul
        // choisir une variable non encore instanciée
        x_i ← Retirer(V)
        // choisir une valeur dans le domaine de x_i
        pour v_i ∈ ED(x_i) faire
            N ← I ∪ (x_i, v_i)
            si IsConsistent(N) alors
                R ← MAC(N, V, ED)
                si R ≠ Nul alors
                    retourner R
        Mettre(V, x_i)
        retourner Nul
```

On remarque que cet algorithme relance une réduction des domaines à chaque appel récursif, ce qui a peu de sens car les contraintes ne sont pas modifiées du début à la fin de la recherche, ce qui implique que les domaines ne seront pas réduits à nouveau. Donc il est préférable de lancer la réduction par arc-cohérence une seule fois avant le début de la recherche effective pour avoir de meilleures performances.

La 5éme démonstration (classe **Demo5**) lance les 3 solveurs (backtrack, maintaining arc consistency, et maintaining arc consistency avec l'heuristique de variable «qui apparaît dans le plus de contraintes» et l'heuristique de valeurs «aléatoire».

```
java blocksworld.demonstrations.Demo5 n m
```

avec :

- n : nombre de blocs.
- m : nombre de piles.

La 6éme démonstration (classe **Demo6**) permet de lancer un solveur au choix et afficher la solution trouvée pour la configuration voulue.

```
java blocksworld.demonstrations.Demo6 n m c s
```

avec :

- n : nombre de blocs.
- m : nombre de piles.
- c : configuration voulue (1 pour régulière, 2 pour croissante et 3 pour les deux).
- s : solveur utilisé (1 pour Backtrack, 2 pour MAC, 3 pour MAC avec heuristiques.

Remarque :
Dans la plupart des cas, la configuration du monde des blocs est déjà localement consistante, donc il n'y a pas de réduction des domaines par arc-cohérence. Ce qui se reflète dans des performances similaires pour les solveurs Backtrack et MAC dans ces cas.

IV. Extraction de connaissances

Cette partie consiste a implémenter l'extraction de connaissances vue en cours. Nous utiliserons la classe **`BooleanVariablesProvider`** pour récupérer l'ensemble des variables booléennes ainsi que l'instance correspondant à un état donné.

La 7éme démonstration (classe **`Demo7`**) affiche les règles d'association extraites d'une base de données de 10000 transactions aléatoires ayant une fréquence et une confiances minimales.

```
java blocksworld.demonstrations.Demo7 f c
```

avec :

- f : fréquence minimale.
- c : confiance minimale.

Conclusion :

Ce projet a permis d'explorer des concepts clés de l'intelligence artificielle, notamment la planification, la satisfaction de contraintes et l'extraction de connaissances. En modélisant le monde des blocs et en mettant en œuvre des algorithmes, nous avons acquis une meilleure compréhension de ces techniques.

Les résultats obtenus sont satisfaisants, et ce travail ouvre des perspectives d'amélioration, comme l'optimisation des heuristiques utilisées ou l'ajout de nouvelles contraintes.