Rapport Projet 2024

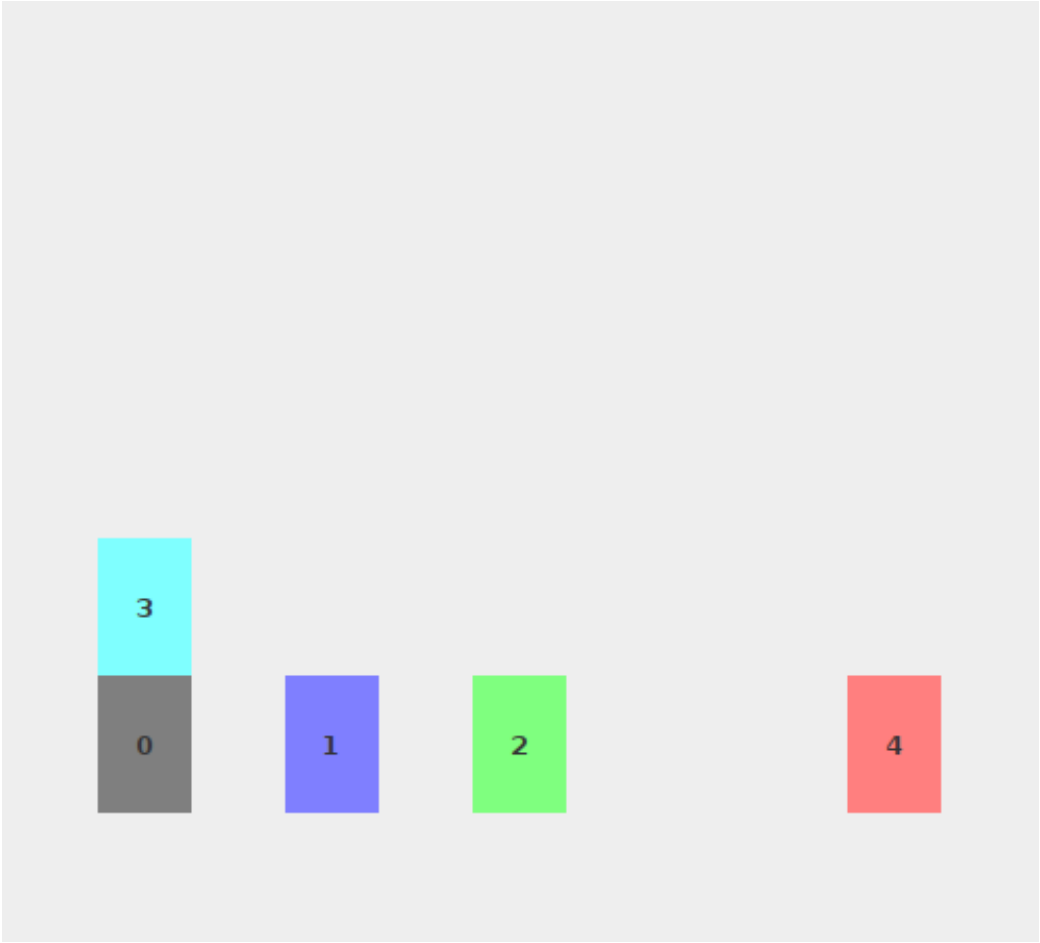
Romain MOALIC Thomas DOGUET

19 novembre 2024

Table des matières

1 Page d'Accueil

Bienvenue sur la page d'accueil de ce document. Ce document est le rapport pour le projet de fin de semestre "fil rouge" dans la matière "Aide à la décision et intelligence artificielle".



2 Mise en place primaire

Nous avions pour consigne de mettre en place un monde de blocs avec tout d'abord une mise en place de base, divisée en quatre sections.

2.1 Modélisation Attribut-Valeur

Cette partie du TP nous a permis de mettre en place les "bases" :

- Nous avons créé les variables, c'est-à-dire chaque élément de notre monde de blocs :
  - Elles sont composées d'un nom et d'un domaine.
  - Les méthodes créées sont des méthodes d'accession au domaine ou au nom, et une autre méthode "equals" permet de vérifier si deux variables sont les mêmes.
- Nous avons créé les variables booléennes, un type de variable dont le domaine est un ensemble contenant les deux booléens.
- Nous avons créé les contraintes, divisées en plusieurs types :
  - Les contraintes de différences, elles prennent en compte deux variables et vérifient grâce à la méthode "IsSatisfied" si elles sont bien différentes.
  - Les contraintes unaires, elles prennent en compte une variable et un sous-ensemble de leur domaine, et vérifient grâce à la méthode "IsSatisfied" si la variable existe.
  - Les contraintes d'implication, elles prennent en compte deux variables (`v1` et `v2`) et deux sous-ensembles de leurs domaines (`s1` et `s2`) :
    - Elles sont satisfaites si ce n'est pas une valeur de `s1` qui est affectée à `v1`, quelle que soit la valeur affectée à `v2`.

2.2 Planification

Cette partie du TP consistait à résoudre des problèmes, c'est-à-dire partir d'un état initial pour arriver à un état final. Pour cela :

- Nous avons d'abord créé les actions :
  - Une action a une précondition (liste de variables nécessaires) et une liste d'effets (état souhaité).
  - Une action vérifie si elle est applicable, c'est-à-dire si ses préconditions sont respectées dans l'état actuel.

— Nous avons créé les buts, qui servent à vérifier que les préconditions pour appliquer une action sont bien respectées.
— Nous avons ensuite créé une série de `Planners` pour résoudre les problèmes selon différents algorithmes :
  — Le parcours en largeur (`BFS`).
  — Le parcours en profondeur (`DFS`).
  — Le parcours suivant l'algorithme de Dijkstra.
  — Le parcours suivant l'algorithme `A-star`.

  Chaque classe contient une méthode `plan`, qui retourne une liste d'actions à effectuer pour résoudre le problème. Elles sont équipées de compteurs pour mesurer leur efficacité en différents cas.
— Enfin, nous avons créé une interface heuristique pour estimer des quantités comme un nombre d'actions ou des blocs avec une propriété particulière.

2.3 Problème de satisfaction des contraintes

Ici, nous nous sommes intéressés à transformer un état invalide en un état respectant toutes les contraintes :

— Nous avons créé un `Solver` :
  — Un solver transforme un état invalide en un état valide en suivant différents algorithmes :
    — L'algorithme de `Backtracking`.
    — L'algorithme `MAC`.
  — Chaque solver a une méthode `solve` qui retourne un état "résolu".
— Nous avons ensuite créé une classe `ArcConsistency` :
  — `enforceNodeConsistency` : Supprime les valeurs non valides dans les domaines des variables. Retourne `false` si une contrainte n'est pas satisfaite.
  — `revise` : Supprime les valeurs impossibles pour une variable, en fonction des contraintes avec d'autres variables.
  — `ac1` : Applique l'algorithme AC1 pour filtrer les domaines.
— Enfin, nous avons créé deux interfaces d'heuristiques :
  — `Heuristiques de Variable`.
  — `Heuristiques de Valeur`.

  Ces heuristiques ont été étendues dans des classes concrètes :
  — `RandomValueHeuristic` : Mélange la liste de valeurs de façon aléatoire.

— `NbConstraintsVariableHeuristic` : Classe les variables en fonction de leur occurrence dans les contraintes.

2.4 Extraction des connaissances

Nous avons maintenant mis en place l'extraction des connaissance en plusieurs étapes

— la mise en place de la classe itemset, qui représente un ensemble d'items (Variables booléennes et leurs fréquence.

— la mise en place de la classe AssociationRule, qui représente une règle d'association, qui est composé d'ensembles d'items, une prémisse et une conclusion, une fréquence et une confiance

— Maintenant notre objectif était de d'extraire ces règles et ensembles d'items, grâce à des miners, les miners sont équipés d'une méthode extract, qui renvoient des ensembles d'ensemble d'items ou de règles d'association, en fonction de condition (les fréquences et confiances minimales pour les règles d'association et les fréquences minimales pour les ensembles d'items)
Nous avons par exemple implémenté un miner d'items set en fonction de l'algorithme Apriori, ou bien un miner de règle d'association de manière brute(BruteForce)

— Nous avons pour finir créer la représentation d'une base de donnée de d'itemset, qui contient les itemset "actuels" et la liste de transactions effectuées auparavant.

3 Mise en place concrète du Monde de Bloc

Afin de mettre en place un monde de blocs, nous avons réutilisé toutes les étapes précédentes de la mise en place :

3.1 Attribut-Valeurs

— Nous avons créé les variables des blocs (`on`, `fixed`, `free`) en suivant une logique systématique.

3.2 Contraintes

— Chaque couple de blocs différents ne peut pas occuper la même position.
— Si un bloc est supporté par un autre, celui-ci devient fixe.
— Si une pile contient un bloc, elle n'est pas libre.

3.3 Contraintes de Régularité

— Les écarts entre blocs dans une pile doivent être réguliers.

3.4 Contraintes de Croissance

— Chaque bloc supérieur a une valeur plus grande que celui situé en dessous.

3.5 Planification

: Nous avons mi en place la planification afin de résoudre notre monde de block, pour cela
— Nous avons défini les différentes actions possibles dans ce monde. Il existe 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'.

  Pour cela, nous avons mis en place la classe `BWAction`, qui parcourt les différents blocs ou piles et génère toutes les actions possibles.
— Nous avons également implémenté trois heuristiques, qui servent à fournir des informations ou à évaluer l'efficacité de notre planification :
  — 1. Une heuristique comptant le nombre de blocs mal placés,
  — 2. Une heuristique comptant le nombre de blocs bloqués sous un autre,

— 3. Une heuristique calculant l'union des blocs bloqués et mal placés.

3.6 Problème de satisfaction des contraintes

Nous n'avons pas eu besoin de créer de représentation spécifique pour cette partie. Nous avons simplement créé des exemples montrant que nos solveurs implémentés précédemment suffisent.

3.7 Extraction de connaissances

Pour représenter l'extraction de connaissances, nous avons tout d'abord créé d'autres types de variables :

— 1. `On_b_b'` : Pour chaque couple de blocs différents $\{b, b'\}$, une variable prenant la valeur `true` lorsque le bloc b est directement sur le bloc b' (et `false` sinon).

— 2. `Ontable_b_p` : Pour chaque bloc b et chaque pile p, une variable prenant la valeur `true` lorsque le bloc b est sur la table dans la pile p (et `false` sinon).

Nous avons créé une classe `BWDatamining` qui récupère les différentes variables, en génère de nouvelles et construit une base de données grâce aux bibliothèques fournies.

3.8 Vue

Pour créer la vue, nous avons développé une classe `Vue` qui utilise le code et les bibliothèques données dans l'énoncé. Ce code est divisé en trois méthodes :

— Une méthode permettant de créer un monde de blocs à partir de l'état donné en paramètre,

— Une méthode pour afficher un état,

— Une méthode pour afficher le déroulé d'un plan de résolution.

3.9 Mise en place de la vue

Pour intégrer la vue, nous avons utilisé la classe `Thread`. En utilisant ses méthodes, il suffit de créer un thread et d'appeler sa méthode `run()` pour lancer l'affichage.

4 Explication des executables

La commande pour tous compiler est la suivante : javac -cp "lib/*" -d build src/**/*.java src/**/**/*.java

4.1 Executable des contraintes

La commande pour lancer l'executable des tests des contraintes est : java -cp build blockWorld.modelling.BWmodellingExecutable . Cette classe permet des tester les contraintes basiques, regulières, croissante ansi que regulière et croissante en même temps. Pour vérifier chaque contraintes, on définit un état qui seras lui à vérifier :

BWmodellingExecutable.java

```
1         int[][] preState0 = {
2           {0, 1},
3           {2, 3, 4, 5},
4           {6, 7},
5           {}
6         };
```

Cette matrice est converti en une map qui représente l'état de cette matrice pour ensuite parcourir chaque élement de cette état pour vérifier les contraintes :

BWmodellingExecutable.java

```
1     for (Constraint constraint : blockWorld.
         getBasiconstrains()) {
2         if (!constraint.isSatisfiedBy(state0)) {
3           allSatisfied = false;
4         }
5       }
```

Un affichage est ensuite fait si toutes les contraintes sont satisfaites
Les images montrent le test pour les contraintes basiques, les mêmes test sont effectués pour les contraintes régulières, croissante et régulières + croissante.
Voici le résultat ! :

```
toutes les contraintes 'basiques' ont étés validées pour [[I@629f0666
toutes les contraintes 'regulieres' ont étés validées pour [[I@1bc6a36e
toutes les contraintes 'croissantes' ont étés validées pour [[I@1ff8b8f
toutes les contraintes 'regulieres et croissantes' ont étés validées pour [[I@387c703b
```

4.2 Executable des Planner

La commande pour lancer l'executable des tests des planner est : java -cp build :lib/blocksworld.jar blockWorld.planning.BWplanningExecutable .

Cette classe permet de tester les planner avec le dfs, bfs, astar et c'est différentes heuristiques et dijkstra. Pour ça on définit un état de base et un autre état qui vaut au but a atteindre.

BWplanningExecutable.java

```
1         int[][] preEtat = {
2                 {0},
3             {1},
4             {2},
5             {3,4},
6         };
7
8         int[][] preBut = {
9             {0},
10            {1},
11            {2, 4},
12            {3},
13        };
```

En utlisant le dfs et autres, on peut récuperer le plan nécessaire avec une liste d'actions, le temp d'execution du planner utilisé et le nombre de noeuds visité pour obtenir le résultat.

BWplanningExecutable.java

```
1         DFSPlanner ActionDFS = new DFSPlanner(etat,
              actions, but);
2         ActionDFS.activateBooleanCount(true);
3         long debut = System.nanoTime();
4         List<Action> planDFS = ActionDFS.plan();
5         System.out.println(planDFS);
6         long fin = System.nanoTime();
7         float temp = (float) (fin - debut) /
              1000000000;
```

Un affichage est ensuite fait, montrant le temps d'execution, le nombre de noeuds visité et la taille du plan d'actions trouvé .
Les mêmes test sont faient pour le dfs, bfs, astar avec les différents heuristiques ([1], [2], [1,2]) et de même pour djikstra.

```
[de base = {On_4[-1, 0, -2, 1, -3, 2, -4, 3]=3, fixed_2[true, false]=false, fixed_4[true, false]=false}apres action {On_4[-1, 0, -2, 1, -3, 2,
 -4, 3]=2, fixed_2[true, false]=true, fixed_3[true, false]=false} avec un cout de = 1]
Le Dijkstra a pris : 48.906322s, a visité 6720
```

4.3 Executables des Solvers

Les commandes pour lancer les executables des tests des solvers sont :

- java -cp build :lib/blocksworld.jar blockWorld.planning.BWRegularityConstraintsExecutable
- java -cp build :lib/blocksworld.jar blockWorld.planning.BWIncreasingConstraintsExecutable
- java -cp build :lib/blocksworld.jar blockWorld.planning.BWRegularityAndIncreasingConstraintsExecutable

Ces classes permettent de tester les solvers pour les différentes contraintes, donc regulières, croissante et les deux en mêmes temps. Pour ça on recuperes les différentes contraintes pour les appliquer.

BWRegularityAndIncreasingConstraintsExecutable.java

```
1  RegularityConstraints reguliere = new
       RegularityConstraints(world);
2  IncreasingConstraints croissante = new
       IncreasingConstraints(reguliere);
3  Set<Constraint> touteContrainte = new HashSet
       <>();
4  touteContrainte.addAll(reguliere.getConstraints
       ());
5  touteContrainte.addAll(croissante.
       getConstraints());
```

En utilisant nos différents solvers : backtrack, mac et macHeuristique on a réussie à recuperer les résultat de nos solvers ainsi que le temps d'éxecution :

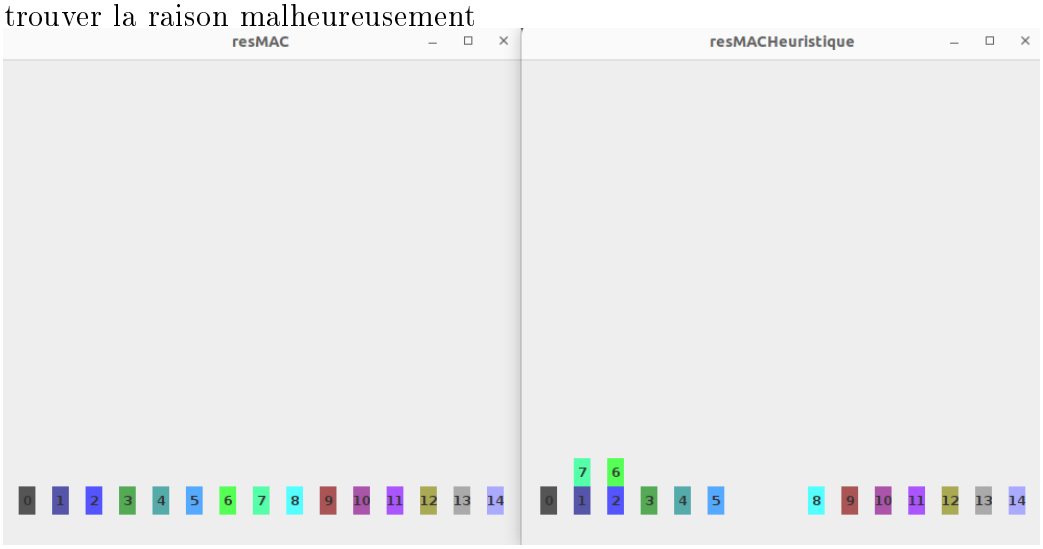
BWRegularityAndIncreasingConstraintsExecutable.java

```
1        Solver backtrack = new BacktrackSolver(
             BwVariable, touteContrainte);
2        long debut = System.nanoTime();
3        Map<Variable, Object> resBacktrack = backtrack.
             solve();
4        long fin = System.nanoTime();
5        float temp = (float) (fin - debut) /
             1000000000;
```

Voici le résultat :

```
Le backtrack a pris : 0.009383994s et avec un resultat de taille : 40
Le MAC a pris : 6.5174656s et avec un resultat de taille : 40
Le MACHeuristiquea a pris : 5.944252s et avec un resultat de taille : 40
```

Le resultat du macHeuristique n'est pas bon et nous n'avons pas réussi à trouver la raison malheureusement



4.4 Executable datamining

La commande pour lancer l'executable datamining est : java -cp build :lib/bw-generator.jar blockWorld.datamining.BWDataminingExecutable
Cette classe permet d'afficher les fréquences chercher des différentes variables (blocs et piles) :

```
 fixed_0 fixed_1 frequence = 0.7632
 fixed_0 frequence = 0.7892
 fixed_0 fixed_1 on_1_0 on_2_1 frequence = 0.7161
Premisse =  on_1_0 confidence = 0.96082115 frequency 0.7161
Premisse =  on_1_0 confidence = 0.96082115 frequency 0.7161
Premisse =  on_1_0 on_2_1 confidence = 1.0 frequency 0.7161
Premisse =  on_1_0 confidence = 0.97155505 frequency 0.7241
```