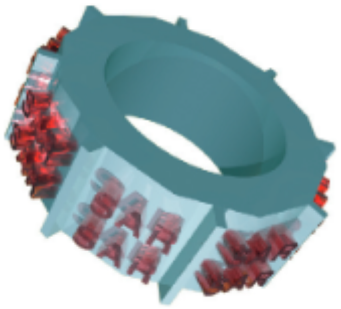

Université Pierre et Marie Curie

Système et Application Répartis

Sécurité et Fiabilité

BMC & Leap-Frog Game

Enseignant :
Souheib BAARIR

Étudiant :
Athmane BENTAHAR
(3410322)

Plan

Introduction

Dans le cadre de l'UE Sécurité et Fiabilité, ce mini projet permet la pratique des concepts théoriques vus en cours et l'apprentissage de la modélisation et l'utilisation d'outils pour la résolution automatisée de problèmes SAT.

Objectif

L'objectif de ce mini projet est de modéliser le jeu *Leap-Frog* et de le résoudre en utilisant le SAT solver **minisat**[2].

Minisat

Minisat est un résolveur SAT de problèmes a satisfiabilité booléenne. Minimaliste et open-source, il a été développé pour les chercheurs ainsi que les développeurs. Il est distribué sous la "MIT license".

Un résolveur SAT peut déterminer s'il est possible de trouver une affectation aux variables booléennes d'une formule pour que la valeur de cette dernière soit vraie. Cette formule doit être écrite à l'aide des opérateurs booléens (AND, OR et NOT) exclusivement, des parenthèses pour délimiter les clauses et des variables booléennes.

Si la formule est satisfiable, la plupart des résolveurs SAT, dont **minisat** peuvent fournir un ensemble d'affectations aux variables qui font que la formule soit vraie.

1 Configurations de base

1.1 configuration initiale

Deux types de grenouilles de deux couleurs différentes, positionnées sur des pierres et alignées en deux files séparées par une pierre vide au milieu. Étant donné le nombre de grenouilles en entrée du programme, une illustration de la configuration initiale pour $N = 3$ (trois grenouilles de chaque couleur) :

FIGURE 1 – Configuration initiale

1.2 configuration finale

La configuration finale est la modélisation de la configuration gagnante du jeu. Celle ci consiste en l'échange de places entre les grenouilles des deux couleurs en passant de l'autre coté de la pierre vide, comme l'illustre la figure suivante pour $N = 3$:

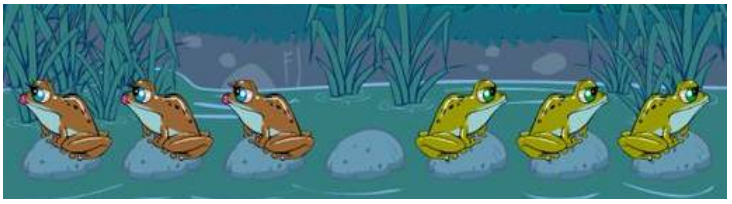

FIGURE 2 – Configuration finale

2 Déplacements possibles (transition)

Comme le précise le texte de l'énoncé, selon la position de la pierre vide (destination de chaque déplacement), il ne peut y avoir que quatre déplacements possibles.

Une pierre est modélisée par une variable du type Stone :

```
1 public class Stone {
2     boolean x;
3     boolean y;
4     Stone () {}
5 }
```

Cette classe ne joue aucun rôle dans l'implémentation, elle sert juste a visualiser la structure d'une pierre. Dans les formules générées pour chacun des déplacements

possibles, les variables sont sous la forme suivante :

```
1 (x_i_s & !y_i_s) // qui represente la pierre i ou est positionnee une
     ↪ grenouille de couleur x a l'etape s
2 (!x_j_s & y_j_s) // qui represente la pierre j ou est positionnee une
     ↪ grenouille de couleur y a l'etape s
3 (!x_k_s & !y_k_s) // qui represente la pierre k vide a l'etape s
```

2.1 Vers la droite

— Une grenouille ne peut se déplacer vers la droite que si elle est de couleur "x".

— Une grenouille de couleur "x" ne peut se déplacer vers la droite que si la pierre adjacente est vide.

Donc, dans la formule générée, nous allons trouver les deux tests.

```
( (x0_0 & !y0_0) & // grenouille de couleur "x"
  (!x1_0 & !y1_0) & // pierre adjacente (cote droit) vide
  (x0_1 = x1_0) & (y0_1 = y1_0) & // permutation
  (x1_1 = x0_0) & (y1_1 = y0_0) & // permutation
  (x2_1 = x2_0) & (y2_1 = y2_0) & (x3_1 = x3_0) & (y3_1 = y3_0) &
  (x4_1 = x4_0) & (y4_1 = y4_0) & (x5_1 = x5_0) & (y5_1 = y5_0) &
  (x6_1 = x6_0) & (y6_1 = y6_0) ) // les autres pierres gardent leur
     ↪ contenu
```

2.2 Vers la gauche

— Une grenouille ne peut se déplacer vers la gauche que si elle est de couleur "y".

— Une grenouille de couleur "y" ne peut se déplacer vers la gauche que si la pierre adjacente est vide.

Donc, dans la formule générée, nous allons trouver les deux tests.

```
( (!x1_0 & y1_0) & // grenouille de couleur "y"
  (!x0_0 & !y0_0) & // pierre adjacente (cote gauche) vide
  (x0_1 = x1_0) & (y0_1 = y1_0) & // permutation
  (x1_1 = x0_0) & (y1_1 = y0_0) & // permutation
  (x2_1 = x2_0) & (y2_1 = y2_0) & (x3_1 = x3_0) & (y3_1 = y3_0) &
  (x4_1 = x4_0) & (y4_1 = y4_0) & (x5_1 = x5_0) & (y5_1 = y5_0) &
  (x6_1 = x6_0) & (y6_1 = y6_0) ) // les autres pierres gardent leur
     ↪ contenu
```

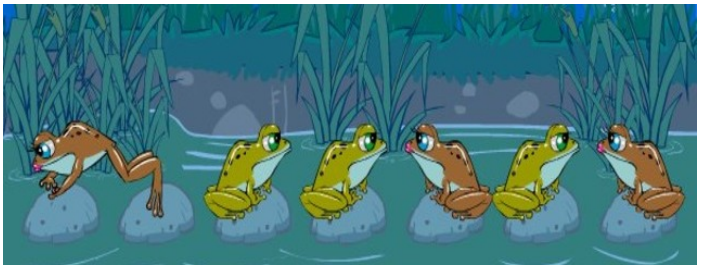

FIGURE 3 – Illustration d'un déplacement vers la gauche

2.3 Saut vers la droite

— Une grenouille ne peut se déplacer vers la droite que si elle est de couleur "x".

— Une grenouille de couleur "x" ne peut sauter vers la droite que si la pierre adjacente contient une grenouille de couleur "y".

— Une grenouille de couleur "x" ne peut sauter vers la droite que si deux pierres plus loin (sur la droite) la pierre est vide.

Donc, dans la formule générée, nous allons trouver les trois tests.

```
( (x1_0 & !y1_0) & // grenouille de couleur "x"
  (!x2_0 & y2_0) & // pierre adjacente (cote droit) de couleur "y"
  (!x3_0 & !y3_0) & // pierre deux pas plus loin (cote droit) vide
  (x3_1 = x1_0) & (y3_1 = y1_0) & // permutation
  (x1_1 = x3_0) & (y1_1 = y3_0) & // permutation
  (x0_1 = x0_0) & (y0_1 = y0_0) & (x2_1 = x2_0) & (y2_1 = y2_0) &
  (x4_1 = x4_0) & (y4_1 = y4_0) & (x5_1 = x5_0) & (y5_1 = y5_0) &
  (x6_1 = x6_0) & (y6_1 = y6_0) ) // les autres pierres gardent leur
     ↪ contenu
```

2.4 Saut vers la gauche

— Une grenouille ne peut se déplacer vers la gauche que si elle est de couleur "y".

— Une grenouille de couleur "y" ne peut sauter vers la gauche que si la pierre adjacente contient une grenouille de couleur "x".

— Une grenouille de couleur "y" ne peut sauter vers la gauche que si deux pierres plus loin (sur la gauche) la pierre est vide.

Donc, dans la formule générée, nous allons trouver les trois tests.

```
( (!x5_0 & y5_0) & // grenouille de couleur "y"
  (x4_0 & !y4_0) & // pierre adjacente (cote gauche) de couleur "x"
  (!x3_0 & !y3_0) & // pierre deux pas plus loin (cote gauche) vide
  (x3_1 = x5_0) & (y3_1 = y5_0) & // permutation
  (x5_1 = x3_0) & (y5_1 = y3_0) & // permutation
  (x0_1 = x0_0) & (y0_1 = y0_0) & (x1_1 = x1_0) & (y1_1 = y1_0) &
  (x2_1 = x2_0) & (y2_1 = y2_0) & (x4_1 = x4_0) & (y4_1 = y4_0) &
  (x6_1 = x6_0) & (y6_1 = y6_0) ) // les autres pierres gardent leur
     ↪ contenu
```

FIGURE 4 – Illustration d'un saut vers la gauche

2.5 Formule générique de déplacement

Pour chacune des grenouilles de la configuration, les quatre déplacements sont proposés dans la formule. Comme la formule doit être de la forme CNF, donc, une suite de conjonctions de mouvements pour chacune des configurations et une suite de conjonction de configurations entre la configuration initiale et la finale.

Il n'y a que les déplacements des grenouilles aux deux extrémités de la configuration qui sont gérés de manière programmatique dans le code. Ces déplacements sont ceux qui provoquent un débordement par rapport au nombre de pierres disponibles dans la configuration.

La formule générique que nous obtenons énumère tous les déplacements possibles a travers les étapes d'évolution des configurations.

Répétition de cette formule pour toutes les pierres du système.

```
1 ((grenouille de couleur "x") & (a droite vide) & (permutation) & (les
    ↪ autres gardent leur valeur) |
2 (grenouille de couleur "x") & (a droite de couleur "y") & (deux pas a
    ↪ droite vide) & (permutation) & (les autres gardent leur valeur) |
3 (grenouille de couleur "y") & (a gauche vide) & (permutation) & (les
    ↪ autres gardent leur valeur) |
4 (grenouille de couleur "y") & (a gauche de couleur "x") & (deux pas a
    ↪ gauche vide) & (permutation) & (les autres gardent leur valeur))
```

3 Sûreté en i

3.1 Pas deux grenouilles sur la même pierre à l'instant i

La formule qui nous permet de vérifier que dans une configuration nous n'avons pas deux grenouilles sur la même pierre, se traduit formellement pas un test lors de la configuration "i" que pour toutes les pierres du système :

```
(pas((grenouille de couleur "x") & (grenouille de couleur "y")))
```

Sous la forme booléenne pour l'étape 1 :

```
( !(x0_1 & y0_1) & !(x1_1 & y1_1) & !(x2_1 & y2_1) & !(x3_1 & y3_1) &
   ↪ !(x4_1 & y4_1) & !(x5_1 & y5_1) & !(x6_1 & y6_1) )
```

4 Sûreté généralisée

4.1 Pas deux grenouilles sur la même pierre jusqu'à l'instant i

La formule généralisée à toutes les étapes jusqu'à l'étape "i" se traduit par une conjonction de conditions de la forme précédente à chacune des étapes effectuées.

```
(pas((grenouille de couleur "x"_0) & (grenouille de couleur "y"_0))) &
(pas((grenouille de couleur "x"_1) & (grenouille de couleur "y"_1))) &
...
(pas((grenouille de couleur "x"_i) & (grenouille de couleur "y"_i)))
```

cette formule va s'ajouter à la formule de déplacement décrite à l'étape précédente.

```
((deplacements possibles_0) & (pas deux grenouilles_0)) &
((deplacements possibles_1) & (pas deux grenouilles_1)) &
...
((deplacements possibles_i) & (pas deux grenouilles_i))
```

5 Pas de répétition de configuration jusqu'à i

La formule qui permet de vérifier qu'il n'y a pas de répétition de configuration, donc l'élimination d'éventuels cycles, se traduit par deux boucles imbriquées pour comparer chaque configuration atteinte avec toutes ses précédentes.

La comparaison se fait avec des disjonctions de différences, c'est à dire, il suffit de trouver une pierre d'une configuration dont le contenu est différent de la même pierre mais dans la configuration actuelle.

```
1 ( !(x0_1 = x0_0) | !(y0_1 = y0_0) | !(x1_1 = x1_0) | !(y1_1 = y1_0) |
    ↪ !(x2_1 = x2_0) | !(y2_1 = y2_0) | !(x3_1 = x3_0) | !(y3_1 =
    ↪ y3_0) | !(x4_1 = x4_0) | !(y4_1 = y4_0) | !(x5_1 = x5_0) |
    ↪ !(y5_1 = y5_0) | !(x6_1 = x6_0) | !(y6_1 = y6_0) )
```

6 Nombre de pas minimal pour atteindre la solution

6.1 Par calcul

Lorsque nous analysons le jeu, nous remarquons que pour atteindre la solution, le chemin le plus court suit le pattern suivant :

Pour 3 grenouilles de chaque couleur il faut déplacer les grenouilles de cette manière :

```
1          "x" "y""y" "x""x""x" "y""y""y" "x""x""x" "y""y" "x"
2          // ou
3          "y" "x""x" "y""y""y" "x""x""x" "y""y""y" "x""x" "y"
```

Ce qui nous donne un nombre de déplacements qui se calcule par la formule :

$$N * (N + 2) \text{ où, } N = \text{nombre de grenouilles de chaque couleur}$$

6.2 Par tâtonnement

Une deuxième manière de calculer le nombre minimal de déplacements nécessaires pour arriver à la solution, qui est de le faire par tâtonnement. C'est à dire, ré-exécuter le programme avec le même nombre de grenouilles et en incrémentant le nombre d'étapes à chaque fois, jusqu'à trouver la première étape "SATISFIABLE".

Le fait de le faire de cette manière peut mener à une boucle infinie. Comme solution je plafonne le nombre d'essais par le calcul avec la formule précédente plus une marge de trois étapes.

7 Généralisation

Puisque le programme n'est pas intelligent, il fait juste ce qu'on lui dit de faire. Donc il ne différencie pas les configurations fournies en entrée ni en sortie. Alors, le programme se comportera de la même manière quelles que soient les configurations fournies, il va les tester pas à pas.

La seule différence réside dans le fait de trouver une étape "SATISFIABLE" qui n'est pas toujours possible dans le cas de configuration aléatoires.

Dans ce programme je laisse à l'utilisateur la liberté de saisir deux chaînes de caractères "in" et "out" qui représentent les deux configurations initiale et finale a tester. Ces deux configurations sont testées, nombre de grenouilles dans chacune, nombre de pierres vides et compatibilité en nombre de grenouilles.

Conclusion

Toutes les formules exposées dans ce rapport sont rédigées dans un format naturel et aussi dans le format booléen. Sauf que le résolveur SAT **Minisat**[2] ne reconnaît pas ces formats.

C'est dans cette phase qu'intervient **bool2cnf**[1], qui est un traducteur de formules du format booléen vers le format CNF comme son nom l'indique.

Son utilité est grande, car elle nous évite de devoir faire la conversion des formules personnellement et aussi permet d'introduire quelques optimisations en entrée du SAT solver.

Outils

[1] bool2cnf. `https://github.com/tkren/bool2cnf`. Dernier accès : 12-02-2018.

[2] minisat. `https://github.com/niklasso/minisat`. Dernier accès : 12-02-2018.