N7PLPA: Miniprojet SMT

Author : Rémi Delmas & Christophe Garion
Public : N7 3IN
Date : 20/11/17

Résumé

Le but de ce miniprojet est d'utiliser un solver SMT pour créer une labyrinthe avec des contraintes prédéfinies.

1 Présentation du problème

On s'intéresse ici à un problème de génération de labyrinthe sur une grille $n \times n$. Le labyrinthe est constitué de cases colorées et on dispose de quatre couleurs : rouge, vert, bleu et jaune. Les couleurs sont considérées comme étant placées sur un disque dans cet ordre.

Pour générer le labyrinthe, on impose les contraintes suivantes :

- deux cases adjacentes doivent avoir des couleurs différentes.
- lorsqu'une case est d'une certaine couleur, on ne peut aller sur une des cases adjacentes que si cette case a la couleur suivante dans la liste. Par exemple, depuis une case verte on ne peut aller que sur une case bleue.
- la case de départ du labyrinthe sera rouge et placée au hasard sur la première ligne.
- la case d'arrivée du labyrinthe sera placée sur la dernière ligne.
- l'utilisateur pourra fixer une distance minimale pour le chemin entre la case de départ et la case d'arrivée.
- il n'y a pas de chemin entre la première ligne et la dernière ligne du labyrinthe ne commençant pas par la case de départ et ne finissant pas par la case d'arrivée.

Un exemple de génération de labyrinthe est donné sur la figure 1.

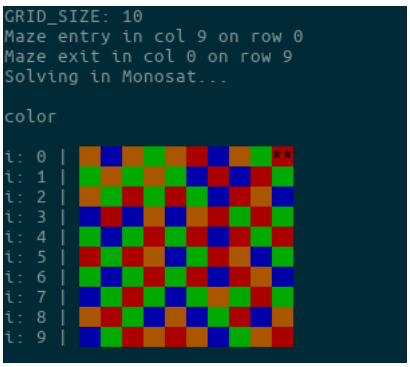

FIGURE 1 – Un exemple de labyrinthe

Le solveur SMT utilisé, MonoSAT [1] dans notre cas, doit donc affecter une couleur à chaque case de telle sorte que les contraintes soient respectées et que le chemin de l'entrée à la sortie soit de taille supérieure ou égale à la distance minimale fixée par l'utilisateur.

2 Les graphes dans MonoSAT

MonoSAT étant un solveur SMT, un certain nombre de théories sont disponibles. En particulier, on peut utiliser des prédicats sur les graphes dans MonoSAT : atteignabilité, plus court chemin, arbre couvrant minimal etc.

Le programme suivant construit le graphe présenté sur la figure 2 (les entiers sur les arcs sont leurs poids respectifs) et demande à MonoSAT de vérifier dans un premier temps si n_4 est atteignable depuis n_1, puis si n_1 est atteignable depuis n_4 (ce qui n'est pas le cas) :

```
from monosat import *

graph = Graph()

n1 = graph.addNode()
n2 = graph.addNode()
n3 = graph.addNode()
n4 = graph.addNode()
```

```
edge12 = graph.addEdge(n1, n2, 3)
edge13 = graph.addEdge(n1, n3, 1)
edge34 = graph.addEdge(n3, n4, 1)

Assert(graph.reaches(n1, n4))

result = Solve()

if result:
    print("SAT")
else:
    print("UNSAT")

Assert(graph.reaches(n4, n1))

result = Solve()

if result:
    print("SAT")
else:
    print("UNSAT")
```

Le programme est disponible sur Moodle via le fichier `graph_ex.py`.

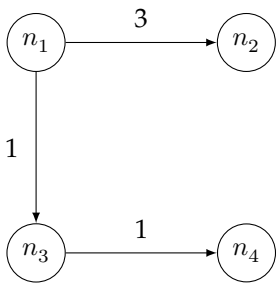

FIGURE 2 – Un graphe à 4 nœuds

L'appel à `graph.addEdge` ne construit pas un arc, mais demande à MonoSAT d'ajouter potentiellement un arc entre les deux nœuds. L'appel renvoie d'ailleurs une variable booléenne qui sera vraie si l'arc est effectivement ajouté dans le graphe.
On pourra donc utiliser ces variables d'arc dans des contraintes et le solveur va calculer quels sont les arcs nécessaires pour respecter ces contraintes.

Sur le même graphe, on peut également exprimer des contraintes sur la longueur des chemins entre les nœuds :

```
from monosat import *

graph = Graph()

n1 = graph.addNode()
n2 = graph.addNode()
n3 = graph.addNode()
n4 = graph.addNode()

edge12 = graph.addEdge(n1, n2, 3)
edge13 = graph.addEdge(n1, n3, 1)
```

```
edge34 = graph.addEdge(n3, n4, 1)

Assert(graph.distance_leq(n1, n2, 5))

result = Solve()

if result:
    print("SAT")
else:
    print("UNSAT")

Assert(graph.distance_leq(n1, n4, 1))

result = Solve()

if result:
    print("SAT")
else:
    print("UNSAT")
```

Le programme est disponible sur Moodle via le fichier `graph_ex_pcc.py`.

3 Les BitVectors dans MonoSAT

Parmi les théories disponibles dans MonoSAT, les BitVectors permettent de stocker des valeurs sous forme de tableaux de bits et d'imposer des contraintes sur les valeurs correspondantes.

```
from monosat import *

bv1 = BitVector(4) # vecteur de 4 bits
bv2 = BitVector(4)
bv3 = BitVector(4)
bv4 = BitVector(4)

Assert(bv1 == 0)
Assert(bv2 == 2)
Assert(bv3 == 6)
Assert(Not(Equal(bv1, bv4)))

result = Solve()

if result:
    print("SAT")
    print("bv1 = {0}".format(bv1.value()))
    print("bv2 = {0}".format(bv2.value()))
    print("bv3 = {0}".format(bv3.value()))
    print("bv4 = {0}".format(bv4.value()))
else:
    print("UNSAT")
```

Le programme est disponible sur Moodle via le fichier `bv_ex.py`.

4 Questions

1. définir une variable `grid_size` représentant la taille du labyrinthe, une variable `min_path_length` représentant la taille minimale du chemin entre l'entrée et la sortie. Ces deux variables seront récupérées via la ligne de commande, en utilisant le module `sys`. Attention, les arguments de la ligne de commande sont des chaînes de caractères et il faut utiliser par exemple la fonction **`int`** pour les transformer en entiers. Regardez le fichier `args.py` disponible sur Moodle.
2. définir quatre variables `RED`, `GREEN`, `BLUE`, `YELLOW` associées respectivement aux valeurs 0, 1, 2 et 3. Ces variables représenteront les couleurs des cases.
3. créer les variables suivantes :
   – une variable `graph` représentant le labyrinthe sous forme de graphe MonoSAT

- un dictionnaire `nodes` qui associera des coordonnées `(i, j)` à des nœuds
- un dictionnaire `edges` qui associera des paires de nœuds à des arcs
- un dictionnaire `color` qui associera à chaque nœud un BitVector de taille 2 (pour représenter la couleur)

4. créer les nœuds du graphe et initialiser `nodes` et `color` en conséquence. Créer les arêtes du graphe et initialiser `edges` en conséquence (on supposera que les arêtes ont toutes un poids égal à 1).
5. imposer les contraintes : deux nœuds voisins n'ont pas la même couleur, une arête entre deux nœuds impose que les couleurs des deux nœuds soient compatibles. On pourra choisir pour ordre des couleurs rouge → vert → bleu → jaune → rouge par exemple
6. placer le nœud de départ et celui d'arrivée et donner les contraintes d'atteignabilité et de taille de chemin.
7. résoudre le problème. Pour afficher le labyrinthe, on pourra utiliser la fonction suivante (il faut importer le module `colorama`) :

```python
def print_color_map():
    from colorama import Fore, Back, Style

    for i in range(grid_size):

        print( "\ni:%2d | " % i, end="" )

        for j in range(grid_size):
            node = nodes[i,j]
            # background color
            color_idx = color[node].value()
            bcolor = Back.RESET
            if color_idx == 0:
                bcolor = Back.RED
            elif color_idx == 1:
                bcolor = Back.GREEN
            elif color_idx == 2:
                bcolor = Back.BLUE
            else : # color_idx == 3
                bcolor = Back.YELLOW
            print( Fore.BLACK + bcolor + "  " , end="" )

        print( Style.RESET_ALL, end="" )
```

8. si vous n'aviez pas à disposition un prédicat d'atteignabilité, comment pourriez-vous l'encoder directement ? Quel est le nombre de clauses que vous devez générer ?

5 Documents à rendre

Vous devez renvoyer pour le **21/12/17 23h** le source d'un programme Python répondant aux questions ci-dessus. **Le fichier source devra impérativement s'appeler `maze-login.py` où `login` est votre login.** Les réponses aux questions pourront y être insérées sous forme de commentaires ou dans un fichier texte séparé (format texte simple, pas de OpenOffice ou PDF). Le fichier sera envoyé par mail à `garion@isae-supaero.fr` et **le sujet du mail devra être `[N7PLPA] projet SAT`**.

Références

[1] Sam Bayless. *MonoSAT*. 2015. URL : https://github.com/sambayless/monosat.

License CC BY-NC-SA 3.0