IA02 - FINAL P2016 – 20 Juin

- Cours et TDs autorisés.

Picross (15 points)

Un fidèle lecteur du Fil (un peu tricheur) vous a contacté pour réaliser un programme en prolog qui permet de résoudre une grille de Picross. Une grille de picross est une grille dont il faut colorier certaines cases (en noir) et laisser les autres non coloriées en respectant des contraintes de blocs. Le coloriage final permet de découvrir un dessin. Un bloc est une suite de cases noires alignées sur une ligne ou une colonne. Pour chaque ligne et pour chaque colonne est fournie la liste des tailles de blocs qui doivent exactement apparaître sur la ligne ou sur la colonne.

	3	1	3		1	3	1		3	1, 1	1, 1
1,1,3,3											
1,1,1,1											
3,1,3											

Par exemple, la contrainte 3,1,3 indiquée avant la 3e ligne de la grille de la figure ci-contre signifie que sur la 3e ligne on doit trouver un bloc de 3 cases noires consécutives, puis 1 bloc de 1 case noire, puis un autre bloc de 3 case noires. La contrainte 3 sur la 1ère colonne signifie qu'il n'y aura qu'un seul bloc de 3 cases noires sur cette colonne.

Il n'y a pas de contrainte sur la 4e colonne signifiant qu'aucune case ne doit être coloriée sur cette colonne. Notons qu'au moins une case blanche doit séparer deux blocs de cases noires consécutifs.

Afin de résoudre un problème de picross en prolog, on représente une solution candidate par une liste de listes, chaque sous liste représentant une ligne de la grille. Chaque élément d'une liste représente une case de la grille qui peut être 0 (non colorié) ou 1 (colorié en noir). Ainsi, l'exemple suivant représente une solution de l'instance précédente :

```
[[1,0,1,0,1,1,1,0,1,1,1],
 [1,0,1,0,0,1,0,0,1,0,0],
 [1,1,1,0,0,1,0,0,1,1,1]]
```

Consignes : À part les prédicats de base (`write`, `nl`, `!`, `\+`, `=`, `\=`, `is`, `=:=`, `=\=`, `>=`, ...), on suppose qu'aucun prédicat n'est prédéfini. On suppose aussi que les prédicats `longueur` et `concat` tels que définis dans le poly sont disponibles. Tout autre prédicat utilisé devra être réécrit sauf indication contraire. Chaque question est indépendante : vous pouvez utiliser un prédicat d'une question (même sans y avoir répondu).

Partie I : Préliminaires (4 points)

1- Ecrire le prédicat **`element_n(N,L,X)`** qui unifie le `N`e élément (`N` donné) de la liste unifiée avec `L` (donnée) avec la variable `X`.

2- Ecrire le prédicat **`ligne_n(N,G,L)`** qui unifie la `N`e ligne (`N` donné) de la grille unifiée avec `G` (donnée) avec la variable `L`.
Ex : `ligne_n(2,[[1,0,1],[1,0,1],[1,1,1]],L).` unifie `L` avec `[1,0,1]`.

3- Ecrire le prédicat **`colonne_n(N,G,C)`** qui unifie la `N`e colonne (`N` donné) de la grille `G` (donnée) avec la variable `C`.
Ex : `colonne_n(3,[[1,0,1],[1,0,1],[1,1,1]],C).` unifie `C` avec `[1,1,1]`.

4- Ecrire le prédicat **`colonnes(G,C)`** qui unifie `C` avec la liste des colonnes de la grille unifiée avec `G` (donnée).
Ex : `colonnes([[1,0,1],[1,0,1],[1,1,1]],C).` unifie `C` avec `[[1,1,1],[0,0,1],[1,1,1]]`.

Partie II : Vérifier qu'une grille candidate est solution (5 points)

1- Ecrire le prédicat **`passe_bloc(L,NL,T)`**. La variable `L` est unifiée avec une liste (donnée) représentant une ligne ou une colonne (une composante) d'une grille de picross. Si `L` commence par un bloc (c'est-à-dire par une suite de `1`), ce prédicat unifie `NL` avec les éléments qui suivent le bloc et `T` est unifiée avec la taille du bloc qui a été passé. Si la liste `L` ne commence pas un bloc (c'est-à-dire `L` commence par 0), `NL` est unifié avec `L` et `T` est unifiée avec la valeur 0.

Ex : `passe_bloc([1,1,0,1,0],NL,T).` unifie `NL` avec `[0,1,0]` et `T` avec 2.
Ex : `passe_bloc([0,1,1,0,1,0],NL,T).` unifie `NL` avec `[0,1,1,0,1,0]`et `T` avec 0.

2- Ecrire le prédicat **`analyse_composante(C,B)`**. Ce prédicat s'efface si la composante (une liste représentant une ligne ou une colonne) unifiée avec `C` (donnée) est valide par rapport à la contrainte de bloc unifiée avec `B` (donnée) qui est la liste des tailles de bloc que l'on doit retrouver dans la composante (l'ordre est important).
Ex : `analyse_composante([0,1,1,0,1,0,1,1],[2,1,2]).` s'efface.
Ex : `analyse_composante([1,1,0,1,0,1,1,1],[3,1,2]).` ne s'efface pas.
Ex : `analyse_composante([0,0,0,0,0,0,0],[]).` s'efface.

3- Ecrire le prédicat **`analyse_l_composantes(LC,LB)`**. Ce prédicat s'efface si la liste de composantes (une liste de listes) unifiée avec `LC` (donnée) est valide par rapport à la liste de contraintes de blocs (qui est aussi une liste de listes) unifiée avec `LB` (donnée); chaque contrainte de bloc de `LB` s'appliquant à une des composantes de `LC` dans le même ordre.
Ex : `analyse_l_composantes(`
`[[1,0,1,0,1,1,1,0,1,1,1],[1,0,1,0,0,1,0,0,1,0,0],[1,1,1,0,0,1,0,0,1,1,1]],`
`[[1,1,3,3],[1,1,1,1],[3,1,3]]).` s'efface.
Ex : `analyse_l_composantes([[1,0,1],[1,0,1]], [[1,1],[1]]).` ne s'efface pas.
Ex : `analyse_l_composantes([[1,0,1],[0,0,0],[1,1,1]], [[1,1],[],[3]]).` s'efface.

Partie III : Générer une solution (5 points)

Dans cette partie, la stratégie qui va être utilisée consiste à générer un ensemble de lignes vérifiant les contraintes de bloc en ligne et ensuite à vérifier que les contraintes de colonnes sont vérifiées.

1- Ecrire le prédicat **`genere_bloc(X,T,B)`**. qui unifie `B` avec une liste de taille `T` ne contenant que des valeurs `X` (donnée qui sera unifié avec 0 ou 1).
Ex : `genere_bloc(0,3,B).` unifie `B` avec `[0,0,0]`.
Ex : `genere_bloc(1,4,B).` unifie `B` avec `[1,1,1,1]`.

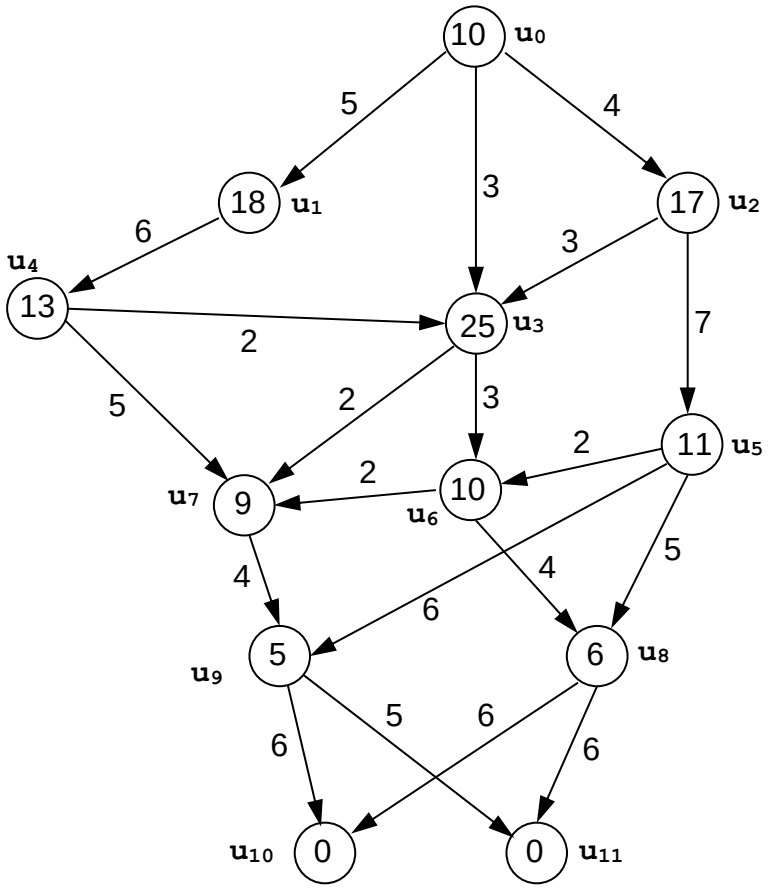
2- Ecrire le prédicat **`genere_ligne(N,LB,L)`** qui permet de générer et d'unifier une ligne avec la variable `L` dont la taille a été unifiée avec `N` (donnée). La ligne générée doit vérifier la contrainte de bloc unifiée avec `LB` (donnée). Ce prédicat doit être capable de générer tous les lignes possibles qui vérifie la contrainte de bloc.
Ex : `genere_ligne(3,[1,1],L).` unifie `L` avec `[1,0,1]` (c'est la seule solution)
Ex : `genere_ligne(6,[2,1],L).` unifie `L` avec successivement
`L = [1,1,0,1,0,0] ; L = [1,1,0,0,1,0] ; L = [1,1,0,0,0,1] ; L = [0,1,1,0,1,0] ;`
`L = [0,1,1,0,0,1] ; L = [0,0,1,1,0,1] ;`
Dans l'écriture des prédicats, on fera attention à gérer le cas où le dernier bloc termine une ligne et le cas où le dernier bloc est suivi de cases non coloriées. On fera aussi attention aux cas particuliers :
`genere_ligne(3,[],L).` unifie `L` avec `[0,0,0]` (c'est la seule solution)
`genere_ligne(0,[],L).` unifie `L` avec `[]` (c'est la seule solution)
`genere_ligne(0,[2],L).` échoue

3- Ecrire le prédicat **`genere_grille(N,BS,LS)`** qui permet de générer et d'unifier une liste de *M* lignes de taille `N` (donnée) avec la variable `LS` de manière à ce que les lignes générées vérifient les contraintes de bloc unifiées avec la liste de contraintes `BS` donnée (une par ligne et dans le même ordre) et où *M* est la taille de la liste `BS`. Ce prédicat doit être capable de générer toutes les listes de lignes possibles.
Ex : `genere_grille(3,[[1,1],[],[]],LS).` unifie `LS` avec `[[1,0,1],[0,0,0],[0,0,0]]` (c'est la seule solution) .
Ex : `genere_grille(4,[[1,1],[],[2,1]],LS).` unifie `LS` avec successivement
`LS = [[1,0,1,0],[0,0,0,0],[1,1,0,1]]; LS = [[1,0,0,1],[0,0,0,0],[1,1,0,1]];`
`LS = [[0,1,0,1],[0,0,0,0],[1,1,0,1]];`

4- Ecrire le prédicat **`solve_picross(CL,CC,G)`** qui permet de générer et d'unifier avec la variable `G` une grille de picross qui vérifie les contraintes en ligne de la liste `CL` (donnée) et les contraintes en colonne de la liste `CC` (donnée).
Ex : `solve_picross([[1,1,3,3],[1,1,1,1],[3,1,3]],`

`[[3],[1],[3],[],[1],[3],[1],[],[3],[1,1],[1,1]],G)`. unifie `G` avec `[[1,0,1,0,1,1,1,0,1,1,1], [1,0,1,0,0,1,0,0,1,0,0], [1,1,1,0,0,1,0,0,1,1,1]]` (c'est la seule solution) .

Trajectoire (5 pts)



Dans le graphe d'états ci-contre, u_0 est l'état ou nœud initial, u_{10} et u_{11} sont les états finaux ou nœuds solutions. La valeur de l'heuristique `h(n)` est précisée en chaque nœud `n`, le coût de chaque opérateur est indiqué directement sur chaque flèche. Par exemple, $h(u_3)=25$.

Question 1 :

1.1 Appliquer l'algorithme A à ce graphe, évidemment jusqu'à ce que la condition de terminaison soit satisfaite. Vous développerez donc simultanément un graphe de recherche et un arbre de recherche, en partant bien sûr du nœud u_0. Vous préciserez lors de chaque expansion, exactement comme en cours et TD, les valeurs de `d` et de `h`, ainsi que le numéro d'ordre de l'expansion `(2 points)`.
Remarque : pour les flèches de l'arbre de recherche, vous pouvez éventuellement utiliser une couleur différente. Par ailleurs, si vous êtes amené à changer la valeur de `d` ou à modifier l'emplacement d'une flèche de cet arbre, faites en sorte que ce changement soit bien visible.

1.2 Quel chemin avez-vous trouvé ? Etait-il possible de trouver d'autres chemins ? Quel est leur coût ? Sont-ils optimaux ? `(1 point)`

Question 2 :

2.1 Montrez que l'heuristique `h` n'est pas minorante. Indication : regardez en particulier les nœuds u_2 et u_3. Puis modifiez les valeurs de `h` aux nœuds u_2 et u_3 de façon à ce que `h` devienne minorante. On appelle `h'` cette nouvelle heuristique. `(1 point)`

2.2 Appliquez alors l'algorithme A* muni de `h'` au problème initial. Qu'obtenez-vous? `(1 point)`