Projet individuel d'algorithmique-programmation
IAP1 : groupe 2.2
Picross

octobre 2008

1 Informations générales

1.1 Travail à rendre

Le projet est à réaliser en OCaml **individuellement**. Il sera accompagné d'un **dossier** contenant impérativement la description des choix faits, la description des types et des fonctions. Pour chaque fonction, on donnera impérativement l'interface complète (dans le code en commentaire et dans le rapport pour les fonctions présentées).

Même si le sujet est décomposé en questions, en général chque question se résoud par l'écriture d'une ou plusieurs fonctions intermédiaires. Celles-ci doivent comporter une interface également.

Le dossier fournira également des cas de tests accompagnés des résultats attendus et retournés.

Sur le site du cours figure un petit document sur ce que l'on attend dans un rapport. Consultez le !

1.2 Calendrier et procédure de remise

Le projet (dossier + copie du listing de code) est à rendre au secrétariat **le 7 novembre à 16h30 au plus tard**. Le numéro du groupe, ainsi que le nom du chargé de TD, devront figurer en gros, et en rouge sur la page de garde.

Le fichier .ml contenant votre code devra être déposé electroniquement au plus tard **le 7 novembre à minuit**.

Les soutenances seront organisées dès la semaine suivante. Il vous faudra consulter les panneaux d'affichage et votre courrier electronique pour obtenir l'ordre de passage.

Enfin n'attendez pas pour vous mettre au travail ! Un projet se travaille dès la remise du sujet afin d'avoir le temps de laisser murir la solution et de poser des questions au client (dans votre cas, votre chargé de TP).

1.3 Procédure de dépot

Pour déposer le code du projet, voici la procedure (consultez les informations sur les machines de l'école, tout figure) :

1. faire un fichier archive compressé : taper la commande `tar zcvf IAP1-PJ-groupex_y.tar.gz toto.ml`
2. lancer une connexion ssh sur la machine de dépot. Le mot de passe qui vous est demandé est votre mot de passe habituel sur les machines de l'école.
3. Une interface graphique apparaît alors. Faire entrer une première fois pour choisir le projet puis taper 3 puis entree. Cela ferme l'interface graphique, votre projet est deposé. Un message vient confirmer votre dépot. C'est tout

Ne vous inquiétez pas l'interface fabrique un nom de projet déposé à partir de votre login. Votre projet ne sera pas confondu avec celui d'un autre.

Indiquez quand même en commentaire dans votre fichier `toto.ml` votre nom et votre groupe. Ce sera plus facile à lire pour le correcteur.

Vous pouvez déposer successivement plusieurs versions, le dernier dépot écrase l e précédent, seul le dernier dépot est pris en compte

Enfin tout cela peut se faire de l'extérieur. Il suffit de vous connecter avant de commencer au point 1 sur une machine visible de l'exterieur (par ssh).

Le code à déposer est un fichier.ml commenté avec les interfaces des fonctions. **Le fichier doit pouvoir être compilé (sans aucune intervention humaine) sur les machines Yaca de l'école**, n'oubliez pas de vérifier que tout fonctionne sur les machines de l'école avant de le dép oser - si vous l'avez développé sous un autre système d'exploitation oar exemple . A priori ce devrait être portable.

2 Présentation du sujet

Dans ce sujet, il s'agit d'implanter un solveur de puzzles Picross. Un Picross ou logigraphe ou hanjie ou nonogram est un jeu de réflexion solitaire, qui consiste à découvrir un dessin sur une grille en noircissant des cases, d'après des indices logiques laissés sur le bord de la grille.

Des jeux de ce genre ont fait leur apparition en 1925 avec notamment une grille appelée le "Stephenson's Train". Ce n'est qu'à la fin des années 80 que le Picross sous sa forme actuelle a vu le jour. En 1987, Non Ichida, graphiste japonaise, pour les besoins d'un concours qu'elle gagna, crée une image sur un immeuble en allumant et éteignant les fenêtres de ce dernier. En 1988 elle publie 3 puzzles qu'elle nomme "Window Art Puzzles" dans un magazine japonais.

Dans la même période, et sans connexion apparente, Mr Tetsuya Nishio invente plus ou moins le même jeu et est publié dans un autre magazine.

2.1 Jeu de Picross

Un Picross se présente sous la forme d'une grille de cases qu'il faudra noircir ou laisser blanche pour former une image. Les indices se trouvant à gauche de chaque ligne et en haut de chaque colonne permettent de résoudre le puzzle : chaque indice est une série de nombres qui signifient le nombre de cases noires contiguës qui constituent chaque bloc, sachant qu'il faut au moins une case blanche entre les blocs. On ne considérera que des Picross qui ont une solution unique.

2.2 Résolution

Il existe un ensemble de règles assez restreint pour résoudre la plupart des grilles de Picross. En effet, les grilles simples peuvent être résolues sans jamais faire de "raisonnement par l'absurde", *i.e.* en faisant l'hypothèse qu'une case doit être blanche ou noire et en déduire que le puzzle ne peut pas être résolu. Bien que ces règles fonctionnent à merveille lorsqu'elles sont appliquées par un cerveau humain, elles sont difficiles à modéliser en informatique et peu efficaces. Néanmoins il existe un point commun entre le raisonnement humain et les algorithmes utilisés sur machine : Ils procèdent tous deux une ligne ou colonne à la fois. Pour le reste il faudra partir de zéro.

3 Préliminaires

3.1 Types de données

Un Picross est défini par la taille de la grille, la liste des indices pour les lignes et celle pour les colonnes. Chaque indice est une liste d'entiers.

Question 1.
Définir le type `puzzle` regroupant la taille du Picross et ses indices.

Une solution est une grille de cases. Chaque case de la solution est soit noire soit blanche.

Question 2.
Définir le type somme `square` des cases d'une solution.

Pour une grille on prendra une matrice sous la forme d'une liste de listes.

3.2 Opérations sur les matrices

Les opérations suivantes pourront s'avérer très utiles pour la suite.

Question 3.
Ecrire la fonction `transpose` qui calcule la transposée de la matrice donnée.

Question 4.
Ecrire les fonctionelles `matrix_map_line` et `matrix_map_column` qui, étant donnée une fonction qui opère sur une liste, renvoient le résultat de cette fonction appliquée aux lignes ou aux colonnes d'une matrice donnée.

Question 5.
Ecrire les fonctionelles `matrix_forall_line` et `matrix_forall_column` qui, étant donnée un prédicat sur une liste *i.e.* une fonction du type `'a list -> bool`, renvoient `true` si toutes les lignes ou colonnes vérifient le prédicat et `false` sinon.

4 Vérification d'une solution

Dans cette partie, il s'agit d'écrire une fonction qui vérifie qu'une grille donnée est bien solution d'un puzzle.

Question 6.
Ecrire une fonction `is_line_solution` qui renvoie vraie si la ligne donnée est bien solution d'un indice donnée.

Indication. On pourra écrire deux fonctions auxiliaires :
- une fonction qui prend une liste de cases en argument et la renvoie sans les cases blanches en tête ;
- une autre fonction qui teste si le bloc en tête d'une liste de cases a la taille attendue.

Question 7.
Ecrire la fonction `is_solution` qui vérifie qu'une matrice de cases est bien solution du puzzle donné.

5 Solveur basique

Comme il a été dit au préallable, la meilleure façon de résoudre un Picross est de procéder une ligne ou colonne à la fois. Ainsi un solveur se divise en deux : une fonction qui trouve la plus grande partie possible de la solution pour une ligne à partir de l'indice et les cases qui ont déjà été fixées, et une fonction qui applique celle-ci successivement aux lignes et aux colonnes suivant une stratégie donnée.

Pour construire un solveur simple, il suffit d'adopter une stratégie naïve qui consiste à appliquer le solveur de ligne à toutes les lignes puis à toutes les colonnes. S'il reste des cases indéternimées on réitère le processus.

Définition 5.1 (ligne partielle, colonne partielle)**.**
Une ligne ou colonne partielle est une ligne ou colonne qui peut comporter des cases blanches ou noires ou de couleur indéterminée

Indication. On pourra se servir du type prédéfini `'a option` qui a pour constructeurs `None` et `Some` **of** `'a` pour modéliser les cases d'une ligne partielle.

5.1 Solveur de ligne

Question 8.
Ecrire la fonction `find_offset` qui trouve le plus petit décalage de la gauche où on peut insérer un bloc d'une taille donnée dans une ligne partielle donnée. Elle renvoie le triplet (`decalage`, `bloc`, `ligne_restante`) où

- `decalage` est le décalage à partir de laquelle le bloc a été inséré ;
- `bloc` est la partie de la ligne qui a été modifiée pour insérer le bloc ;
- `ligne_restante` est ce qu'il reste comme case non traitées.

Question 9.
Ecrire la fonction `skip_offset` qui prend une ligne partielle et un nombre n en argument et essaie de colorier les n premières cases en blanc. Elle renvoie le couple (`blancs`, `ligne_restante`) où `blancs` est une liste de n cases blanches et `ligne_restante` est la liste des cases non traitées.

Question 10.
Ecrire la fonction `find_positions` qui prend un indice et une ligne partielle et renvoie toutes les positions possibles pour les blocs donnés dans l'indice.

Indication. Pour trouver toutes les positions possibles pour les blocs décrits par un indice il suffit de partir de la position la plus à gauche de ces blocs et de les décaler progressivement vers la droite.

Question 11.
Ecrire la fonction `aggregate_positions` qui prend une liste de lignes et renvoie la ligne partielle qui contient les cases dont la couleur est la même dans toutes les lignes et des cases de couleur indéterminée partout ailleurs.

Question 12.
Ecrire la fonction `line_solver` qui réalise un solveur de ligne.

5.2 Solveur complet

Question 13.
Ecrire une fonction qui applique le solveur de ligne suivant la stratégie naïve donnée.

Remarque. Ce solveur ne doit pas renvoyer de ligne partielle.

5.3 Résoudre des Picross

Il est maintenant l'heure de voir le solveur en action. Voici quelque puzzles ; essayez de les résoudre à la main d'abord puis avec le solveur que vous avez implanté.

Étoile

			2				2
			2	2	1	2	2
	2	2					
	2	2					
	1	1					
1	1	1					

Cœur

		3	4	4	4	3
2	2					
	5					
	5					
	3					
	1					

Téléphone

				1		
			4	2	5	
2	1					
2	3					
1	1					
	3					
	3					

Musique

		2	2	5	1	3
	3					
1	1					
1	1					
	3					
	3					

6 Solveurs optimisés

Pour aller plus loin et résoudre dans un temps raisonnable des problèmes plus grands, il faut trouver des stratégies d'application du solveur de lignes différentes.

Question 14.
Déterminer un type de données dont les valeurs pourront décrire l'avancement de la résolution du puzzle. On se limitera à savoir pour chaque ligne et chaque colonne si elle est résolue ou non. Implanter un nouveau solveur qui tient compte de cette nouvelle donnée et n'applique le solveur de ligne que si la ligne ou colonne n'est pas résolue.

Question 15.
Trouver un autre type qui indiquera si une ligne ou colonne contient de nouvelles informations. Implanter un nouveau solveur qui repère les cases qui ont été modifiées sur une ligne (resp. colonne) par une passe du solveur de ligne et considère alors que la colonne (resp. ligne) correspondante contient une nouvelle information. Ce solveur n'appliquera le solveur de ligne que sur des lignes ou des colonnes non résolues qui contiennent des nouvelles informations.

Question 16.
Résumer les différences de performance en temps de calcul entre les solveurs qui ont été implantés. Se baser sur les puzzles donnés, des puzzles récupérés d'autres sources ou encore certains déterminés soi-même. Dans ce dernier cas, veiller à ce que la solution soit unique.

Remarque. Il est possible que certains puzzles ne puissent pas être résolus par les solveurs présentés. En effet, il en existe qui nécéssitent de raisonner par l'absurde.