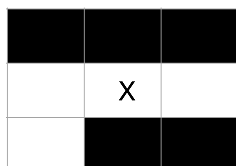
Projet IPI 2015-2016
Pierre Boutry & Pierre Tellier
pierre.tellier@ensiie.fr

Jeu de Merlin

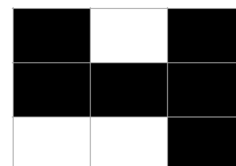
Description du jeu.
Le jeu comporte un tableau carré de 9 cases. Chaque case peut être noire ou blanche. Au départ les cases sont dans une initialement toutes blanches. Le but du jeu est de transformer toutes les cases en case noire. Pour cela le joueur peut cliquer sur des cases et les couleurs changent en suivant les règles suivantes :

- Si le joueur clique sur la case centrale les couleurs des cases de la colonne et de la ligne du milieu sont inversées ;
- Si le joueur clique sur l'un des coins ce sont les 4 cases proches de ce coin qui sont inversées ;
- Si le joueur clique sur une case d'un côté qui n'est pas un coin ce sont les cases de ce côté qui sont inversées.

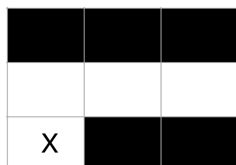
Par exemple, après avoir cliqué au milieu de :



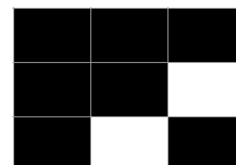
on obtient :



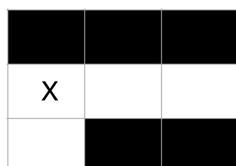
Par exemple, après avoir cliqué sur le coin :



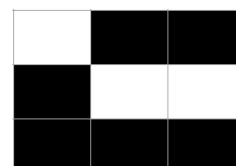
on obtient :



Par exemple, après avoir cliqué au milieu du côté :



on obtient :



1) Modélisation du jeu.

- Définir un type `plateau` qui permet de représenter un plateau comme un tableau de 9 booléens.
- Définir un type `coord` qui permet de représenter les coordonnées comme un couple de 2 entiers (chacun entre 1 et 3);
- Définir un type `partie` qui permet de représenter une partie sous la forme d'une liste chainée de coordonnées de clics.
- Définir une fonction `applique_clic` qui prend en paramètres un plateau et les coordonnées d'une case, et qui délivre (sous la forme d'un paramètre modifiable) le plateau obtenu si on clique sur cette case.
- Définir une fonction `applique_partie` qui prend en paramètres un plateau initial et une partie et qui génère la plateau obtenu si on joue la partie sur le plateau initial.
- Définir une fonction `plateau_init_test` qui initialise un plateau ne comprenant que des cases blanches.
- Définir une fonction `plateau_gagnant` qui permet de vérifier qu'un plateau passé en paramètre n'est constitué que de cases noires.

- Définir une fonction `partie_gagnante` qui permet de déterminer si une partie est gagnante sur le tableau initial.
- Ecrire un programme qui permet à un utilisateur de jouer, c'est à dire saisir des clics jusqu'à abandon ou victoire.

2) Solution automatique du jeu

L'objectif du travail est d'étendre votre programme afin qu'il calcule à la demande une suite de clics à réaliser pour obtenir un plateau gagnant. Il existe des façons de trouver « intelligemment » la solution au problème, nous allons nous intéresser à une approche plus « brutale », la résolution par essais successifs.
L'idée est d'essayer toutes possibilités, jusqu'à obtenir un plateau gagnant. Pour être sûr d'aboutir, on mémorisera toutes les configurations de plateaux déjà produites, afin de s'interdire de les produire à nouveau lors d'une nouvelle action de jeu. On marquera aussi les configurations entièrement développées, afin de ne pas essayer de repartir de ces configurations. On s'appuie sur le fait que le jeu ne comporte que 9 cases, chacune pouvant ne revêtir que 2 couleurs, il y a donc un nombre fini de plateaux, 2^9=512 combinaisons.

- Une première version gèrera les configuration sous la forme d'une table comportant les 512 configurations possibles et d'indicateurs permettant de savoir si la configuration a déjà été produite, ou entièrement explorée.
- Une autre version de la méthode de résolution par essais successifs stockera les configurations visitées dans une liste. A chaque développement d'une configuration non marquée comme déjà développée, les configurations qu'il sera possible de générer (c'est à dire non déjà présentes dans la liste) seront ajoutées à la liste.
- Enfin une troisième version stockera les configurations produites dans un arbre, en partant de la configuration de départ. L'algorithme de résolution itérative consistera à répéter la recherche dans l'arbre d'une configuration non déjà développée, et d'enraciner toutes les configurations issues de celle-ci qui ne soient pas déjà dans l'arbre. Vous proposerez plusieurs variantes à l'algorithme de recherche d'une configuration :
  - 1ère configuration trouvée en profondeur d'abord
  - parcours aléatoire
  - parcours avec heuristique : parmi toutes les configurations candidates à être développées, choisir celle qui semble être la plus proche de la solution, selon un critère que vous proposerez.

Toutes ces versions sont à réaliser.

3) Livrables (à déposer sur scoledge, dépôt Projet IPI, jusqu'au 5/1/2016) :

- Les sources de votre programme (fichiers.c et .h), documentés format Doxygen.
- Un jeu de tests unitaires
- Des traces d'exécution avec Valgrind (prévoir un mode « batch »).
- Les fichiers Makefile, et README avec les instructions d'installation et utilisation
- La documentation générée par Doxygen.
- Une courte documentation technique précisant le périmètre couvert, et les parties non fonctionnelles (cas non couverts, fonctions non écrites ou non mises au point).

4) Annexes

```
typedef struct t_noeud {
      plateau p;
      // + autres champs : indicateurs, heuristique ?, lien vers père ?
      struct t_noeud *fils[9]; // initialisés à NULL en absence de fils.
} Noeud, *Arbre;
```

```
Arbre ajouterFils(plateau p, Arbre arbre, int numFils)
{
	Noeud *noeud;

	// ajouter autres paramètres et initialisations
	noeud = (Noeud *)malloc(sizeof(Noeud));
	noeud -> plateau = p;
	for(i=0;i<9;i++)
		noeud->fils[i]=NULL;
	if (arbre != NULL)
		arbre->fils[numFils]=noeud;
	else arbre = noeud;

	return arbre;
}

// premier appel de ajouterFils (dans main()):
plateau plt = init_plateau_test();
Arbre a = ajouterFils(plt, NULL, 0);
```