15/05/2022

Documentation Technique

Projet Takuzu

Cassiopée GHIZELLAOUI
Paul ZANOLIN

Table des matières

autosolve.h

Ce fichier gère tout ce qui concerne la résolution automatique, génération d'indice, et traitement des MOVE.

```
int giveHint(SIZEDGRID usergrid, int * x, int * y, int * val);
```

Prend en paramètre la grille, ainsi que les adresses de trois int qui permettront de stocker l'indice trouvés. Retourne 1 si un indice a été trouvé, 0 sinon. Pour trouver les indices, la fonction va appliquer les fonctions qui suivent afin de vérifier les différentes possibilités, dans l'ordre :

1) **Vérifier s'il y a des groupes de deux nombres qui se suivent dans la colonne et dans la ligne i. Si oui, calculer l'indice qui en ressort.**
2) **Vérifier s'il y a des nombres similaires séparés d'une case vide dans la colonne et dans la ligne i. Si oui, calculer l'indice qui en ressort.**
3) **Vérifier si une ligne/colonne est similaires à la ligne/colonne i, et que i possède deux cases vide quand l'autre est complètement remplie. Si oui, calculer l'indice qui en ressort.**
4) **Vérifier si une ligne/colonne i contient déjà size/2 symboles similaires. Si oui et qu'il reste des cases vides, calcule la valeur d'une des cases comme indice.**

```
int checkIfUnderIsTheSame(SIZEDGRID usergrid, int x, int y);
```

Vérifier si le nombre en (x+1, y) == celui en (x,y) (Si possible).

```
int checkIfRightIsTheSame(SIZEDGRID usergrid, int x, int y);
```

De même, avec (x, y) == (x, y+1).

```
int placeHintUnder(SIZEDGRID usergrid, int x, int y, int * outX, int *
outY, int * val);
```

Si des cases sont libres de part et d'autres du groupe de deux vertical, retourne 1 et met x, y et val à (x,y) de la case, et val a l'opposé de la valeur des cases du groupe de deux.

```
int placeHintRight(SIZEDGRID usergrid, int x, int y, int * outX, int *
outY, int * val);
```

Idem, mais horizontalement.

```
int checkIfSpaceBetweenTwoSameUnder(SIZEDGRID usergrid, int x, int y);
```

Retourne 1 s'il y a une case vide en (x+1, y) et que (x, y) == (x+2, y), sinon 0 (si possible).

```
int checkIfSpaceBetweenTwoSameRight(SIZEDGRID usergrid, int x, int y);
```

Retourne 1 s'il y a une case vide en (x, y+1) et que (x, y) == (x, y+2), sinon 0 (si possible).

```
void placeHintInSpace(int x, int y, int baseVal, int *outX, int *outY,
int *val);
```

Place la valeur opposée à baseVal en (x, y).

```
int checkIfLineHave2Empty(SIZEDGRID usergrid, int lineNum);
```

Vérifie si la ligne a deux cases vides.

```
int checkIfColumnHave2Empty(SIZEDGRID usergrid, int columnNum);
```

Idem pour la colonne.

```
int placeHintIfSameLine(SIZEDGRID usergrid, int lineNum, int * x, int *
y, int * val);
```

Si la ligne est similaire à une autre, place l'indice.

```
int placeHintIfSameColumn(SIZEDGRID usergrid, int columnNum, int * x,
int * y, int * val);
```

Idem pour les colonnes.

```
int isLineFull(SIZEDGRID usergrid, int lineNum);
```

Vérifie si la ligne est pleine, ne contient pas de cases vides.

```
int isColumnFull(SIZEDGRID usergrid, int columnNum);
```

Idem pour les colonnes.

```
int countSymbolInLine(SIZEDGRID usergrid, int lineNum, int * zero, int
* one, int * minusOne);
```

Compte les symboles sur la ligne. Met le nombre résultant de 0, respectivement 1 et -1, dans *zero, respectivement *one et *minusOne. Retourne 1 si (nbZero == size/2 ou nbUn == size/2) et nbminusOne != 0, sinon 0.

```
int countSymbolInColumn(SIZEDGRID usergrid, int columnNum, int * zero,
int * one, int * minusOne);
```

Idem en colonne.

```
int placeHintFillLine(SIZEDGRID usergrid, int lineNum, int * x, int *
y);
```

Place l'indice quand la ligne a déjà size/2 symboles d'un type.

```
int placeHintFillColumn(SIZEDGRID usergrid, int columnNum, int * x, int
* y);
```

Idem en colonne.

```
MOVE * allocMove();
```

Alloue la mémoire pour un maillon de la chaine de coups. Retourne l'adresse du nouveau maillon.

```
void freeMove(MOVE * moveToFree);
```

Libère la mémoire du maillon à l'adresse moveToFree.

```
MOVE * newMoveWithValues(int x, int y, int hypothesis, int changedOnce,
MOVE * previous);
```

Initialise un maillon avec les valeurs données en paramètre. Retourne l'adresse du nouveau maillon.

```
void rollbackGridToHypothesis(SIZEDGRID * grid, MOVE ** moveList);
```

Permet d'annuler les coups qui ont été joués sur la grille depuis la dernière hypothèse, cette dernière incluse. Ne retourne rien, modifie directement la grille et la liste actuelle de coups joués.

```
int getNextCaseToDo(SIZEDGRID usergrid, int rank, int * x, int * y, int
* val);
```

Permet de déterminer l'hypothèse n°rank à partir de la totalité des hypothèses possibles a un certain état de la grille. Contient une part d'aléatoire sur la valeur testée en premier. Retourne toujours 0, n'est pas sensée retourner 1 sauf si rank est > au nombre d'hypothèses possibles, ce qui n'est pas censé arriver. La nouvelle hypothèse est mise dans x, y, val.

```
int recursiveSolve(SIZEDGRID * usergrid, MOVE ** moveList, int
printSteps);
```

ALORS.

En gros recursiveSolve va prendre en paramètre la grille à résoudre, ainsi qu'un pointeur vers la liste des coups effectués (initialement NULL). Elle va d'abord tenter dans le while de compléter les cases forcées. Ensuite, lorsqu'il n'y a plus de cases forcées, elle vérifie si la grille est valide. Si non, elle retourne zéro, ce qui communique aux fonctions supérieures qu'une des hypothèses formulées était fausse. Si la grille est valide, on vérifie qu'elle ne soit pas complète. Si oui, retourne 1, et remonte la récursivité. Si non, on entre dans la boucle d'hypothèse, et la récursivité. On émet une hypothèse, on appelle la fonction sur ce nouveau tableau. Si la grille qui en sort est invalide, on annule l'hypothèse et on en émet une autre. Si valide, La grille est résolue ! On retourne 1 et on sort.

```
recursiveSolve ():
    Tant que Indices:
        Appliquer les indices;

    Si la grille est valide :
        Si la grille est complete :
            retourner 1;
        Sinon:
            Generer la liste des hypotheses possibles;
            i = 0;
            Faire:
                faire hypothese i
                result = recursiveSolve();
                Si result = 0:
                    annuler l'hypothese;
                i++;
            Tant que (i < longeureListeHypotheses ET result == 0);
            Si i = longeureListeHypotheses:
                retourner 0;
            Sinon Si result == 1:
                retourner 1;
    Sinon:
        retourner 0;

```

grid.h

Ce module gère les grilles, vérifie leur validité, le respect des règles du Takuzu, la gestion de mémoire pour les grilles, ainsi que les grilles par défaut.

```
SIZEDGRID genMask(int size);
```

Génère un masque aléatoire de la taille demandée, retourne le masque sous forme d'un SIZEDGRID. Le masque comporte taille^2/2 cases 1 et le même nombre de cases 0, aléatoires.

```
SIZEDGRID allocGrid(int size);
```

Alloue l'espace mémoire pour le tableau d'un SIZEDGRID de taille size.

```
void freeGrid(SIZEDGRID * grid);
```

Free l'espace mémoire à l'adresse grid.

```
int** convertToTakuzu4(int tab4[4][4]);
```

Converti la grille 4x4 implémentée en dur en tableau dynamique utilisable dans notre programme.

```
int** convertToTakuzu8(int tab8[8][8]);
```

Converti la grille 8x8 implémentée en dur en tableau dynamique utilisable dans notre programme.

```
int** convertToTakuzu16(int tab16[16][16]);
```

Converti la grille 16x16 implémentée en dur en tableau dynamique utilisable dans notre programme.

```
SIZEDGRID getGrid4();
```

Retourne un SIZEDGRID de la grille en dur 4x4.

```
SIZEDGRID getMask4(); // debug
```

Retourne un SIZEDGRID du masque en dur 4x4. N'est utilisé que pour des tests.

```
SIZEDGRID getGrid8();
```

Retourne un SIZEDGRID de la grille en dur 8x8.

```
SIZEDGRID getGrid16();
```

Retourne un SIZEDGRID de la grille en dur 16x16.

```
void fillWithInt(SIZEDGRID * gridToFill, int valueToFill);
```

Remplie la partie grid de gridToFill avec valueToFill (utile pour remplir de -1 une grille).

```
void addOneOnTwoUsingMaks(SIZEDGRID gridOne, SIZEDGRID * gridTwo,
SIZEDGRID mask);
```

Permet de remplacer certaines valeurs de gridTwo par celle de gridOne de la manière suivante :

If (mask[x][y] == 1) {*gridTwo[x][y] = gridOne[x][y]}.

```
int checkZeroEqualOne(SIZEDGRID usergrid, int x, int y, int val);
```

Vérifie que la première règle du Takuzu est respectée, c'est-à-dire qu'il est possible qu'il y ai autant de 1 que de zéro dans chaque ligne/colonnes, pour usergrid[x][y] = val. Retourne 1 si la règle est respectée, 0 sinon.

```
int checkMax2Following(SIZEDGRID usergrid, int x, int y, int val);
```

Vérifie que la seconde règle du Takuzu est respectée, soit pas plus de deux symboles similaires à la suite, pour usergrid[x][y] = val. Retourne 1 si la règle est respectée, 0 sinon.

```
int checkSimilarLinesOrColumns(SIZEDGRID usergrid, int x, int y, int
val);
```

Vérifie que la première règle du Takuzu est respectée, soit interdiction d'avoir deux lignes ou deux colonnes équivalentes, pour usergrid[x][y] = val. Retourne 1 si la règle est respectée, 0 sinon.

```
int isNewValValid(SIZEDGRID usergrid, int x, int y, int val, int
showErr);
```

Retourne 1 si les trois règles sont respectées, 0 sinon.

```
int isGridValid(SIZEDGRID usergrid);
```

Vérifie si une grille donnée est valide en appelant isNewValValid avec une des valeurs de la grille. 1 si valide, 0 sinon.

```
int checkEnded(SIZEDGRID usergrid);
```

Vérifie si toutes les cases de la grille sont remplie. /!\ Ne vérifie pas la cohérence de la grille, simplement si elle est remplie. 1 si oui, 0 sinon.

```
int countEmpty(SIZEDGRID usergrid);
```

Retourne le nombre de cases vides dans une grille (le nombre de -1 donc).

interaction.h

Ce module permet les interactions avec le programme, ainsi que la gestion de l'interface utilisateur.

```
void printGrid(SIZEDGRID grid, SIZEDGRID mask, int ignoreMask);
```

Affiche la grille grid. Si ignoreMask == 1, le masque donné en paramètre ne sera pas appliqué. Prend en charge les -1, affiche _.

```
void pickCoords(int* x, int* y, int size);
```

Demande a l'utilisateur de saisir des coordonnées de manière sécurisée. Size correspond à la taille de la grille, pour limiter la saisie. Les coordonnées retournées sont stockées dans *x et *y.

```
int getSize();
```

Demande à l'utilisateur de saisir une taille de grille, de manière sécurisée. Retourne cette taille. Possibilité de répondre au choix par 1, 2, 3, ou simplement par 4, 8, 16.

```
int getValue(int x, int y);
```

Demande à l'utilisateur de saisir une valeur pour la case en x, y, de manière sécurisée. Retourne la valeur saisie (1 ou 0).

```
SIZEDGRID getMask(int size);
```

Retourne un masque. L'utilisateur peut choisir de le générer de manière automatique ou de le saisir de manière sécurisée.

```
void play();
```

Fonction principale pour le jeu, permet à l'utilisateur de jouer.

```
void autoSolveInterface();
```

Fonction principale pour la résolution auto, permet au joueur de faire résoudre une grille au programme

```
void genGridInterface();
```

Permet de générer une grille 4x4. (marche aussi avec du 8x8, pas implémenté)

```
int mainMenu();
```

Menu Principal.

Autres précisions

Grid vs usergrid

Au début du code, nous utilisions majoritairement grid dans nos fonctions. Par la suite, nous avons plutôt utilisé usergrid. La différence notable entre ces deux termes est que, par convention, grid correspond la plupart du temps a une grille solution, c'est-à-dire totalement remplie de 1 et 0. A l'inverse, usergrid correspond à une grille en cours de résolution. Elle comporte des -1 dans les cases qui ne sont pas encore complétées. Cela s'applique la plupart du temps, bien qu'il y ait quelques exceptions.

Les structures

On a utilisé deux structures principales, SIZEDGRID et MOVE. Nous n'avons pas vraiment vu l'intérêt d'utiliser d'autres structures, mais après réflexion une structure CASE {int x ; int y ; int val ;} aurait pu être intéressante sur la résolution automatique.

SIZEDGRID correspond simplement à un tableau dynamique auquel on a attaché une taille.

MOVE contient les coordonnées x et y du coup, ainsi que le fait que le coup soit ou non une hypothèse. Le paramètre changedOnce n'a finalement pas été utilisé. Contient aussi l'adresse du coup précédent, permet de faire des listes chainées comme un stack, last in first out.

Pourquoi retourner des int quand on peut retourner des bool ?

Ça c'est une bonne question ça.

...

Merci de l'avoir posée.