15/05/2022

Projet Takuzu

Année 2021-2022

Cassiopée GHIZELLAOUI
Paul ZANOLIN

Table des matières

Introduction

Nous avons dû réaliser un jeu nommé « Takuzu » en utilisant notre dernier langage tout récemment appris : le C. Un Takuzu qu'est- ce que c'est ? C'est un jeu de réflexion qui consiste à remplir une grille avec les chiffres 0 et 1 en suivant certaines règles dont on parlera plus tard.

L'objectif principal était de nous familiariser avec l'utilisation du langage C en l'appliquant sur un projet concret. On a notamment pu approfondir trois principales notions. Le passage par référence pour passer une grille à une fonction. Les structures pour stocker les grilles. Et pour finir, bien évidemment, les fonctions pour nous permettre d'avoir un code organisé.

Présentation fonctionnelle

L'objectif final de ce projet était de créer un jeu nommé Takuzu, nous devions proposer au minima deux niveaux différents, l'un avec une grille 4x4 et l'autre avec une grille 8x8 pour lesquelles le joueur aurait la possibilité de les remplir. Pour cela nous avons procédé en trois grandes étapes que nous allons développer ci-dessous après une brève présentation des règles :

Règles du Takuzu

- Il doit avoir le même nombre de 1 et de 0 sur une ligne ou une colonne
- Il ne peut avoir deux lignes ou deux colonnes identiques
- On peut voir maximum deux 1 ou deux 0 à la suite

Et, découlant de ces règles :

- Entre deux 0 il ne peut y avoir uniquement un 1
- Entre deux 1 il ne peut y avoir uniquement un 0
- Une série de deux 0 est entourée par un 1 à ses extrémités
- Une série de deux 1 est entourée par un 0 à ses extrémités

Laisser un joueur résoudre une grille

Le joueur doit pouvoir résoudre une grille de Takuzu. Cette dernière est constituée uniquement de 1 et de 0, mais, lorsqu'elle est présentée au joueur, seulement quelques cases sont visibles. Pour cela nous allons générer un masque qui servira à savoir quelle case l'ordinateur doit montrer ou non. (Voir ci-contre).

```
Voici le masque :
    0  1  2  3

0   0  1  0  0
1   1  1  0  0
2   0  0  1  1
3   1  1  0  1
```

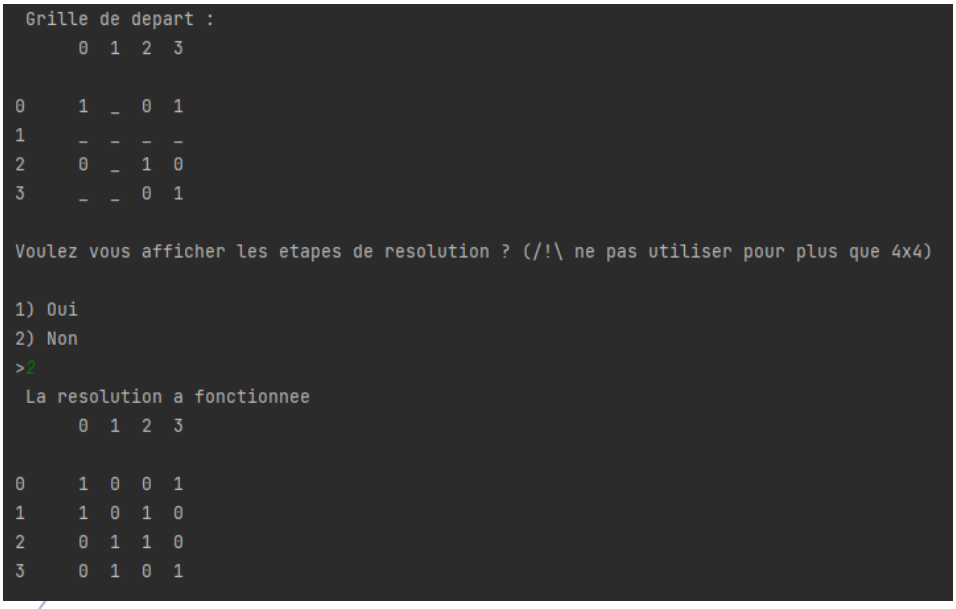
```
Voici la grille :
    0  1  2  3

0   _  0  _  _
1   1  0  _  _
2   _  _  1  0
3   0  1  _  1
```

Revenons à l'utilisateur, il doit pouvoir choisir une case et dire s'il désire la remplir par un 0 ou par un 1. Soit le coup est valide soit il est invalide. Dans le dernier cas, le programme propose à l'utilisateur un indice, pour une grille, afin de la compléter. Il y a deux bons coups à différencier : un coup valide respecte les règles mais n'est pas celui attendu par le programme, a l'inverse un coup correct est la bonne réponse et celle attendue. De plus l'utilisateur possède 3 vies et lorsqu'un mauvais coup est effectué il en perd une.

Résoudre automatiquement une grille

Dans cette partie on veut que l'ordinateur résolve lui-même la grille en prenant la place de l'utilisateur. Il faut aussi que chaque étape de la résolution soit affichée à l'écran. Pour cela, nous avons conçu une fonction qui demande à l'utilisateur de choisir entre une grille préfabriquée ou une grille qu'il entre lui-même. Ensuite, le programme tentera de résoudre la grille en appliquant les conseils ou en faisant des hypothèses. Cette fonction est capable de résoudre des grilles de 4x4, et de 8x8, cependant pour certaines grilles de 8x8, elle prendre un temps conséquent, ou au contraire sera quasi instantanée. Ci-dessous une illustration de la résolution d'une grille de 4x4



Générer une grille

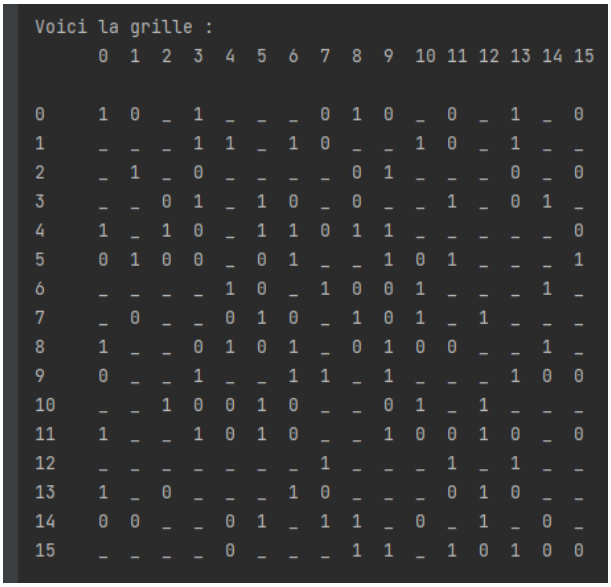
Nous n'avons pas eu le temps de terminer cette partie. Néanmoins, l'algorithme de résolution étant fonctionnel, il est possible de l'appliquer a une grille vide pour en générer une nouvelle.

Nous avons fait des essais sur une grille de 4x4 et cela fonctionne sans problème, mais nous n'avons pas encore pu obtenir de temps raisonnables pour la génération de grille 8x8. On peut penser, pour améliorer les performances, a pré-générer quelques lignes/colonnes puis laisser l'algorithme résoudre la fin de la grille. De plus, cette prégénération permettrai d'obtenir une part d'aléatoire, ce qui pour l'instant n'est pas le cas.

Fonctionnalités additionnelles

Nous avons pu ajouter une fonctionnalité au niveau des niveaux justement. On avait comme obligation de mettre une grille 4x4 et une 8x8. Nous avons pris la liberté de rajouter une grille 16x16.

De plus, nous avons ajouté un indice possible. Pour les lignes qui possède déjà taille/2 exemplaires d'un chiffre, les cases restantes auront forcément la valeur de l'autre chiffre. Nous avons donc implémenté cela pour donner des indices supplémentaires à l'utilisateur et aider d'avantage notre résolution automatique.



Présentation technique

Passons maintenant à une présentation plus technique du projet. Nous allons nous focaliser sur quelques-uns des algorithmes de notre code, qui, pour nous, reflètent bien la complexité et les difficultés principales de ce projet. Nous montrerons ensuite la structure que nous avons choisi pour nos données, et enfin nous parlerons des difficultés que nous avons rencontré.

Description des principaux algorithmes réalisés

Les algorithmes qui étaient les plus complexes à développer était sans hésiter celui qui vérifie la validité d'une grille, même partiellement remplie, l'algorithme de suggestion d'indices, ainsi que l'algorithme de résolution automatique. Nous allons donc les passer en revue.

isValid (), le gardien.

isValid (plus tard renommé isNewValValid) est en quelque sorte le gardien de la cohérence du Takuzu. C'est cet algorithme qui permet de vérifier si une grille, après que le joueur a placé une nouvelle case, sera toujours valide ou non.

```
245  int isNewValValid(SIZEDGRID usergrid, int x, int y, int val, int showErr) {
246      int zeroEqualOne, twoSameMax, notSameColOrLines;
247      // rule 1 : zeroEqualOne
248      zeroEqualOne = checkZeroEqualOne(usergrid, x, y, val);
249      // rule 2 : deux max a la suite
250      twoSameMax = checkMax2Following(usergrid, x, y, val);
251      // rule 3 : pas de colone ou de ligne pareilles
252      notSameColOrLines = checkSimilarLinesOrColumns(usergrid, x, y, val);
253      if (showErr) {
254          if (!zeroEqualOne) {
255              printf( format: "Le nombre de zero par ligne/colonne doit etre egal au nombre de un\n")
256          } else if (!twoSameMax) {
257              printf( format: "Maximum deux memes valeurs a la suite\n");
258          } else if (!notSameColOrLines) {
259              printf( format: "Deux colonnes/lignes ne peuvent pas etre similaires\n");
260          }
261      }
262      return zeroEqualOne && twoSameMax && notSameColOrLines;
263  }
```

Il fonctionne d'une manière finalement assez simple : il prend en paramètre une copie de la grille actuelle, ainsi que les coordonnées et la valeur de la nouvelle case insérée par l'utilisateur (on peut aussi lui donner les coordonnées et valeur d'une case existante pour vérifier la cohérence de la grille entière.). Il passe ensuite ces paramètres dans trois fonctions qui vérifient chacune une règle du Takuzu. Après cela, si showErr est à 1, le programme affiche une des règles qui n'est pas respectée par le coup joué. Cela permet de réutiliser la fonction, en mettant showErr à 0, notamment dans l'algorithme de résolution. Enfin, la fonction retourne une valeur vraie si les trois règles sont respectées.

Cet algorithme est sans conteste le plus important du programme.

giveHint (), l'altruiste.

Cet algorithme est celui qui va observer la grille de Takuzu sous toutes ses coutures et donner un indice à l'utilisateur s'il trouve une valeur forcée pour une case.

Il existe 4 types d'indices :

- Si deux chiffres similaires se suivent, ils ont à leurs extrémité le chiffre opposé.
- Si deux chiffres similaires sont séparés par une case vide, le chiffre au centre est le chiffre opposé.
- Si une ligne à laquelle il manque deux cases est similaire à une ligne déjà terminée, alors on met les valeurs opposées a celle de la ligne pleine dans les cases restantes.
- Si une ligne a déjà taille/2 exemplaires d'un chiffre, les cases restantes ont la valeur de l'autre chiffre.

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

    // premiers indices, quand ya deux chiffres pareils à coté

    for (int i = 0; i < usergrid.size; ++i) {
        for (int j = 0; j < usergrid.size; ++j) {
            if (checkIfUnderIsTheSame(usergrid, x: i, y: j)) {
                if (placeHintUnder(usergrid, x: i, y: j, outX: x, outY: y, val)) {
                    //printf("a%d, %d\n", *x, *y);
                    return 1;
                }
            }
            if (checkIfRightIsTheSame(usergrid, x: i, y: j)) {
                if (placeHintRight(usergrid, x: i, y: j, outX: x, outY: y, val)) {
                    //printf("b%d, %d\n", *x, *y);
                    return 1;
                }
            }
        }
    }
    // second indice, le chiffre entre deux chiffres
    for (int i = 0; i < usergrid.size; ++i) {...}
    // troisieme indice : les lignes avec deux trous, différentes
    for (int i = 0; i < usergrid.size; ++i) {...}
    // quatrieme indice : si une ligne a deja size/2 d'une valeur dedans
    int zero, one, minusOne;
    for (int i = 0; i < usergrid.size; ++i) {...}
    return 0;
}
```

Il fonctionne comme suit :

Pour chacun des indices, on vérifie sur chaque colonne et chaque ligne si les conditions sont vérifiées. Si c'est le cas et qu'un indice peut être donné, on met les valeurs aux adresses x et y à la position de la case, et la valeur à val a la valeur de la case. La fonction retourne ensuite 1. Si elle ne trouve pas d'indice, elle retourne 0. Cela permet de l'utiliser pour la résolution automatique.

recursiveSolve (), le puissant.

Nous y voilà. Cette fonction nous a donné pas mal de fil à retordre, même si elle est en fait assez simple à comprendre. Etant beaucoup trop grosse pour tenir sur une image, nous l'avons réécrite ici en pseudo-code. Cette fonction va donc prendre en paramètre une grille, ainsi que le premier élément d'une liste chainée qui va permettre de stocker les coups joués. Le principe est le suivant : quand l'on entre dans la fonction, on va déjà

```
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;

```

récupérer tous les coups certains grâce a giveHint. Ensuite, on vérifie la cohérence de la grille. Si elle n'est pas valide, on retourne 0, afin d'informer les fonctions de niveau supérieur qu'une suggestion effectuée est mauvaise, ou que la grille n'est pas solvable. Si elle est valide, on vérifie qu'elle n'est pas terminée, auquel cas on retournerai 1 et toute la récursion remonterai. Si elle n'est pas terminée, cela signifie que l'on doit faire une suggestion. On effectue donc la suggestion et on envoie la grille, ainsi que la liste de coup précédents, à recursiveSolve () qui va effectuer les mêmes étapes. Nous allons maintenant voir pourquoi nous avons utilisé une telle structure pour nos données.

Structure de données

Nous avons principalement deux structures de données dans le programme : SIZEDGRID et MOVE. SIZEDGRID correspond simplement à un tableau 2D carré avec sa taille, ce qui nous évite d'ajouter des paramètres a nos fonctions. MOVE est une LLC qui va stocker les coups joués lors de la résolution. Cela permet de pourvoir revenir en arrière lorsqu'une hypothèse n'est pas valide, comme on peut le voir ci-contre.

```
struct move {
    int x;
    int y;
    int hypothesis;  // 1 or 0
    int changedOnce; // 1 or 0 // not used
    struct move * previous;
};
typedef struct move MOVE;
```

Difficultés

Les difficultés rencontrées lors de ce projet étaient différentes de celles rencontrées pour le projet python du premier semestre ainsi que pour le projet transverse. En effet, ici, la difficulté principale était aussi la particularité du C, avec les structures et la gestion de la mémoire. En effet, Ça a plutôt été un casse-tête de choisir comment écrire nos structures de données, notamment la structure **MOVE** et **SIZEDGRID**, qui sont les plus utilisées dans ce programme. Nous avons aussi eu quelques difficultés à concevoir l'algorithme de résolution, car nous voulions un algorithme qui serait totalement utilisable pour créer des grilles de zéro, et il fallait donc introduire une part d'aléatoire tout en ne sautant pas de possibilités pour résoudre la grille. Ici aussi, c'était compliquer de découper certaines grosses fonctions comme **giveHint** afin de les rendre atomiques, mais une fois fait, on se plaisait à les réutiliser dans d'autres parties du programme.

Présentation des résultats

Menu principal

Ici le menu principal nous permet de faire un choix entre quatre possibilités.

```
Bienvenue !
 Que souhaitez vous faire ?
1) Jouer
2) Resoudre automatiquement une grille
3) Generer une grille
4) Quitter
>
```

Jouer

Partie 1

Si l'utilisateur choisi le choix 1 c'est-à-dire jouer, on lui propose alors de choisir la taille de la grille qu'il désire.

```
 Quelle taille de grille souhaitez vous ?
1) 4
2) 8
3) 16
>2
```

On voit ici sur les deux captures, que l'utilisateur peut soit cliquer sur l'indice du choix, c'est-à-dire 1) 2) ou 3) Ou bien il a la possibilité de directement saisir la taille qu'il souhaite, c'est-à-dire 4, 8 ou 16.

```
 Quelle taille de grille souhaitez vous ?
1) 4
2) 8
3) 16
>8
```

Partie 2

Une fois la taille sélectionnée on nous avons trois nouveaux choix qui s'offrent à nous. Ici l'utilisateur a sélectionné l'option « générer un masque », le masque à donc été généré et affiché.

```
 Voulez-vous :
1) Generer un masque
2) Saisir un masque
3) Jouer
>1
 Voici le masque :
     0  1  2  3  4  5  6  7

0    0  0  1  1  1  0  1  0
1    0  1  0  1  0  1  1  0
2    0  0  1  1  1  0  0  0
3    1  0  1  1  0  0  1  0
4    1  1  0  1  1  0  1  0
5    1  0  1  1  1  1  1  0
6    1  1  1  0  0  1  0  0
7    1  0  1  0  0  0  0  0
```

Partie 3

Cela permet ensuite à l'utilisateur de jouer.

Il sélectionne d'abord la ligne puis la colonne dans laquelle il souhaite rentrer une valeur et pour finir il rentre donc la valeur désirée. On peut remarquer que le programme l'informe de la règle non respectée, lui donne ensuite son nombre de vies restantes et donne en plus un indice. Nous avons pu effectuer nos tests à l'aide des grilles qui nous ont été fournis

```
Voici la grille :
      0  1  2  3  4  5  6  7

0     1  _  _  _  _  _  0  _
1     _  0  _  0  _  _  _  _
2     _  1  _  _  _  0  1  0
3     _  1  _  _  0  1  _  0
4     _  _  1  0  _  _  _  _
5     0  _  _  0  _  0  _  _
6     _  0  _  _  _  _  1  _
7     1  _  _  _  _  _  0  _

Choisissez une case.
Ligne:
>4
 Colonne:
>7
 Valeur de la case en (4, 7):
>0
 Maximum deux memes valeurs a la suite
Il vous reste 2 vies
Et pourquoi pas mettre un petit 0 en (4, 1)
```

Résoudre automatiquement une grille

Voici un aperçu de ce qui se produit lors de la résolution auto d'une grille, avec les étapes affichées. (ci-contre)

```
0 en (5, 0) grace au calcul d'indice.
1 en (5, 6) grace au calcul d'indice.
0 en (0, 6) grace au calcul d'indice.
1 en (0, 2) grace au calcul d'indice.
Aucune case certaine restante
La grille est valide
12 Valeurs possibles pour les cases restantes
Hypothese: 0 en (1, 0)    -> 0/12
1 en (1, 2) grace au calcul d'indice.
0 en (2, 2) grace au calcul d'indice.
1 en (2, 0) grace au calcul d'indice.
0 en (1, 3) grace au calcul d'indice.
0 en (2, 3) grace au calcul d'indice.
Aucune case certaine restante
La grille n'est pas valide, retour a la dernier hypothese
Annulation 0 en (1, 0)
Hypothese: 1 en (1, 0)    -> 1/12
0 en (2, 0) grace au calcul d'indice.
Aucune case certaine restante
La grille est valide
8 Valeurs possibles pour les cases restantes
Hypothese: 1 en (1, 2)    -> 0/8
0 en (2, 2) grace au calcul d'indice.
0 en (1, 3) grace au calcul d'indice.
1 en (2, 3) grace au calcul d'indice.
Aucune case certaine restante
La grille est valide
La grille est resolue. Sortie.
```

Générer une grille

Pour le moment, il n'y a que la démonstration (voir ci-dessous).

Conclusion

Ce projet nous as permis de coder un projet de A à Z et donc d'utiliser à une importante fréquence le langage C. On a pu se familiariser avec ce dernier et approfondir nos compétences. Le fait d'être en groupe nous a aussi permis d'un peu plus équilibrer nos niveaux qui étaient très différents. La communication au sein de notre groupe était très homogène en passant notamment par Discord et GitHub, nous avons pu rester constamment en contact et au même niveau sur le projet.