Hitori

PROJET 2018 - 2019
PYTHON SEMESTRE 2

1 1 Présentation du projet.....2
1.1 Menu des niveaux.....2
1.2 En jeu.....2

2 États d'avancement du projet.....2
2.1 Les tâches.....2
2.2 Les améliorations éventuelles.....2

3 Les fonctions principales.....3
3.1 Les fonctions "graphiques".....3
3.2 Les fonctions "règles du jeu".....3
3.3 Le solveur.....4

4 Organisation.....5
4.1 Première approche du sujet.....5
4.2 Le codage.....5

VILAYVANH Johnson TP 5
MONTEIRO Justin TP 4

1 Présentation du projet

1.1 Menu des niveaux

Lors du lancement du programme, une fenêtre se lance et on arrive dans le menu principal, dans lequel on peut :
- cliquer sur un niveau pour lancer une grille
- lire les règles et le fonctionnement du programme, résumé en 2 pages.

1.2 En jeu

Le clique **gauche** sur la grille permet de noircir la cases, ou les remettre "blanche".
Le clique **droit** permet d'entourer d'un cercle rouge une case, pour permettre au joueur de marquer une case afin de se repérer.

On a ensuite 5 boutons qui se présente à nous :
"**Résoudre**" va lancer le solveur et résoudre la grille, s'il y a 1 solution (pas fonctionnel à 100%, et très lent). On peut décider d'afficher l'avancé du solveur graphiquement, ou non ;
"**Recommencer**" vous ramène au menu principal ;
"**Quitter**" vous propose de quitter le jeu ;
"←" permet d'annuler l'action précédente ;
"→" nous ramène à l'action qui suivait.

2 États d'avancement du projet

2.1 Les tâches

Toutes les tâches ont été effectuées, mais la tâche 4, celle du solveur, n'est pas fonctionnel à 100%, et reste très basique donc très lente.
Malheureusement nous ne sommes pas parvenu à résoudre ce qui n'allait pas, ni a l'améliorer au-delà de la base qui était demandé.

2.2 Les améliorations éventuelles

En dehors du solveur qui reste dans un état très simpliste qui ne fonctionne pas parfaitement non plus, d'autres améliorations auraient pu être faites, mais il nous semblait plus pertinent d'arriver à faire fonctionner le solveur en priorité, en vain malheureusement.
On aurait pu par exemple faire en sorte de permettre au joueur de sauvegarder ou charger une partie. Pour cela, il aurait fallu différencier les cases noircies sur le fichier texte sur lequel on allait sauvegarder notre grille, que ce soit par un symbole pour chaque case (noircie, ou non), ou par une simple ligne qui représenterait la liste des cases noircies, qui se mettrait en marche au lancement de la grille.

Également durant la partie, permettre au joueur d'avoir un indice, pour savoir si une case quelconque doit être noircie, ou au contraire ne présente aucun conflit. Ou bien indiquer quelles cases sont en conflits, montrer les cases noires voisines, ou afficher les cases qui gênent pour permettre aux cases de former une zone tenante.
Mais on a été trop entêté à vouloir perfectionner le solveur que nous sommes passés à côtés de simples améliorations comme celles citées.

3 Les fonctions principales

3.1 Les fonctions "graphiques"

- **menu_niveaux()** : 1ère fonction qui démarre le jeu, en affichant la fenêtre de sélection des niveaux, ainsi que les 2 pages de règle.
  En cliquant sur un niveau, on lance la fonction qui exécute lire_grille(), permettant de lire le fichier texte correspondant au niveau choisi.
- **afficher_grille()** : c'est la fonction principale qui affiche toute la partie graphique du jeu en lui-même, en allant de la grille et toutes ses lignes et les chiffres, jusqu'aux boutons situés en-dessous la grille.
  La fonction est fait en sorte de pouvoir agrandir la taille les cases de la grille en modifiant la valeur de taille_case, au tout début du programme.

3.2 Les fonctions "règles du jeu"

Durant la partie, on a 3 fonctions principales qui régissent les règles du jeu :

- **sans_conflit()** : 1ère des 3 fonctions principales nécessaires au bon déroulement du jeu. Dans cette fonction, on recrée une liste de listes composé non plus des lignes de la grille, mais des colonnes. On étudie par la suite les lignes et les colonnes, en prenant soin d'enlever les cases noircies à l'aide de la liste noircie. Pour ce faire, on a changé les cases qui correspondaient à des cases noircies en None, puis on efface tous les None des listes.
  Il ne reste plus qu'à comparé chaque liste de ligne et colonne avec son homologue set() : si la taille des listes différent, au moins 1 élément est présent 2 fois, et donc la condition n'est pas vérifiée.
- **sans_voisines_noircies()** : sans doute la plus facile des 3 fonctions.
  On va simplement étudier pour toutes les cases noircies, tous les voisins de chacune de celles-ci.
  Si une de ces cases voisines se trouve être dans la liste des cases noircies, alors on a des cases noires qui se touchent.
  Remarque cependant, nous n'avons pas utilisé l'argument "grille" comme il était indiqué dans l'énoncé, car cela ne nous semblait pas du tout utile au vu de notre méthode.

- **connexe()** : la fonction qu'on craignait le plus, qui s'est finalement trouvée être bien plus simple que ce qu'on pensait.
  On s'est énormément inspiré l'algorithme de coloration de zone vu en cours, et on l'a simplement adapté pour qu'elle convienne à la situation.
  Ainsi, au lieu de colorer une zone en changeant tous les éléments qui sont de couleur/d'élement identiques à la case de départ, on change tous les éléments qui sont de type "int". Cependant, on prend bien soin au préalable de changer dans la liste de la grille toutes les cases qui sont noircies par des None.
  On applique ensuite la fonction récursive de coloration pour tous les types "int" à partir de la 1ère case non noircie (en partant du coin haut gauche de la grille), et on les change tous par la chaîne de caractère "OK".
  Il ne reste plus qu'à additionner le nombre de "OK" dans cette copie de la grille, avec le nombre de cases noircies, qui doit correspondre au nombre totale de case de la grille, Si ils sont égaux, alors les cellules visibles forment une zone entre elles.

3.3 Le solveur

- **solveur()** : notre version du solveur étant très basique, elle était plus simple que prévue également. On s'est contenté de suivre à la lettre la consigne qui nous indiquait les étapes à suivre dans le programme :

- Si la règle 2 ou 3 était enfreinte, on annulait la récursivité sur l'avancement actuel ;
- Si les 3 règles sont vérifiées, alors la liste noircie qu'on a est une solution de la grille : on la renvoie ;

On a ensuite 2 branches de possibilités qui s'offraient à nous, comme indiqué dans le sujet :

1 - La cellule étudiée n'est pas en conflit, on continue d'étudier la prochaine case, sans noircir celle actuelle ;

2 - Si elle est en conflit, on noircit alors la case, et on étudie la prochaine case en répétant ce même processus. Et dès que la règle 2 ou 3 est enfreinte, on revient à cette étape en étudiant un autre cas : le cas où on ne noircit pas la case.

En répétant ce processus de récursivité, on obtient une solution.

On propose au joueur d'afficher l'avancé du solveur graphiquement, mais cela prend considérablement plus de temps.

Le solveur semble fonctionner sur de nombreuses grilles, même si elle prend énormément de temps, mais certaines grilles qui pourtant bien une solution, ne sont pas résolvables par notre solveur.

4 Organisation

4.1 Première approche du sujet

Avant de se répartir le travail, on a posé sur une feuille de papier ce que l'on voulait faire, de la même manière que pour notre projet du 1er semestre : on écrit les fonctions importantes, les règles du jeu, on schématiser un exemple d'à quoi le jeu pourrait ressembler dans sa version finale ; les grosses lignes du programme.

4.2 Le codage

On a décidé dans un premier temps d'accomplir la tâche 1 ensemble, qui nous permettra d'avoir la grille en visuel, pour ensuite nous répartir le travail :
- l'un de nous se concentrerait sur la partie graphique (la tâche 3) ;
- tandis que l'autre sur les fonctions nécessaires au déroulement du jeu (la tâche 2).

Étant donné que nous ne pensions pas arriver à faire fonctionner un solveur dans un premier temps, nous voulions nous rejoindre sur la tâche 4, qui semblait être le plus gros obstacle du projet.

Nous accordions également une grande importance à ne pas faire les mêmes erreurs que pour le 1er projet, comme les lignes de codes trop superflu et trop étouffantes. Le code reste tout de même très long, mais par rapport à notre premier projet, on est bien plus satisfait quant à la lisibilité et la partie graphique avec laquelle nous étions bien plus à l'aise grâce à l'expérience du 1er projet.