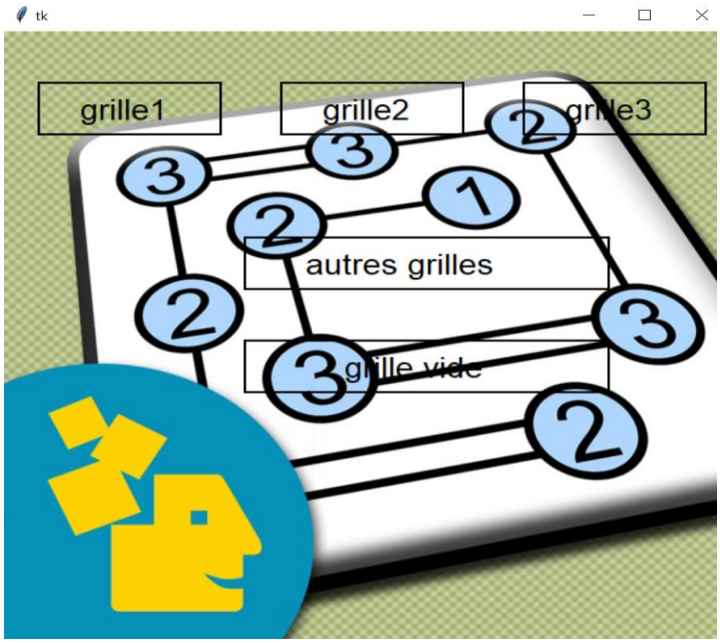
BOHOU CHRIST

RAGOUNATH LAKSHMAN

TP A-B

RAPPORT DU PROJET SLITHERLINK

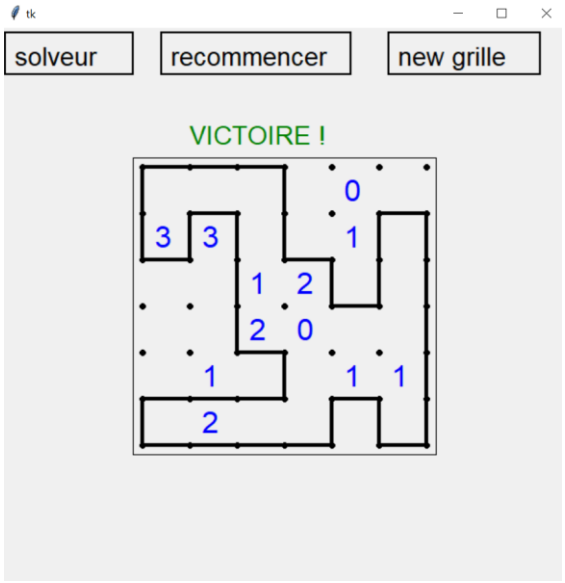
Présentation du menu



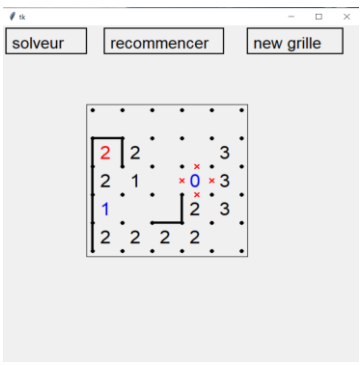
Guide d'utilisation

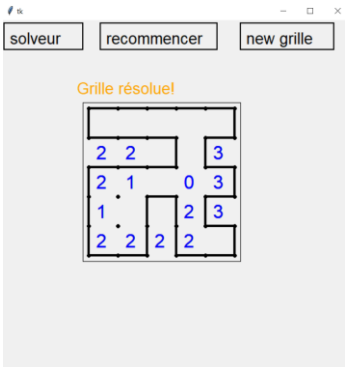
Le jeu est lancé par le programme python "**Slitherlink.py "**. On aperçoit le menu du jeu qui est dans une fenêtre 600*600 pixels dont on a la possibilité d'agrandir ou de réduire et de fermer après avoir choisi une des grilles : la grille 1, la grille 2, la grille 3. Cette dernière correspond à la grille-triviale, il y a également la grille vide qui nous a été donné dans le sujet et les "autres grilles" qui sont des grilles bonus que nous avons créé dont une est choisi aléatoirement après le clic du joueur. Ces grilles sont de différentes tailles. Le joueur doit choisir parmi les quelques grilles pour commencer à jouer. Une fois la grille choisie, il peut alors commencer à jouer, c'est-à-dire effectuer des clics droits ou clics gauches à l'aide de la souris pour compléter la grille et tenter de gagner en formant une boucle et en satisfaisant les indices qui apparaitront en bleu lorsque c'est le cas. Le clic gauche permet de tracer un segment et le clic droit permet d'interdire un segment par une croix entre deux sommets.

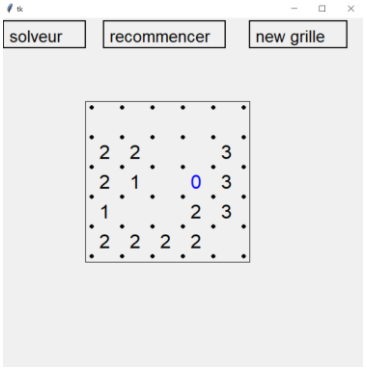
A la fin de la partie, le message “Victoire ! “ est affiché en vert si le joueur a rempli toutes les conditions pour gagner.



Le joueur a, ensuite, le choix de recommencer (pendant la partie également) en faisant un clic gauche sur le bouton “recommencer” ou de relancer une nouvelle partie avec également un clic gauche sur le bouton “new grille”. Il y a également un bouton solveur qui donne la réponse au joueur au cas où il n’aurait pas trouvé la solution ou qu’il aurait abandonné. Le solveur peut être aussi appelé avec la touche “s” du clavier. Une fois la grille résolue par le solveur le message “grille résolue !” est affiché en orange. Les images suivantes montrent l’utilisateur en galère, lorsqu’il appelle solveur et quand il recommence.







Comment tout cela marche ?

Clique (clique gauche)

Au début d’une partie, ”etat ” est le dictionnaire qui stocke tous les segments tracés, ou interdits, par le joueur. “ etat1 ” représente l’état du solveur. Si on clique sur le bouton “recommencer”, les deux états sont immédiatement vidés (etat= {}, etat1= {}), si le joueur clique sur le bouton “new grille” les deux états sont vidés et indices devient celui d’une nouvelle grille (différente de la précédente) parmi celles disponible.

Etat d'avancement

Pour la tache 1, nous avons simplement remplit les fonctions prédéfinis, c'est à dire les fonctions de base du jeu : les fonctions d'accès. Ce sont celles qui vont nous être utiles pour la suite du projet car on les utilisera dans d'autres fonctions. Dans cette même tache nous avons écrit une fonction "statut_case" qui fait appel aux fonctions d'accès pour indiquer le statut de la case à partir de son indice et des côtés tracés.

Pour la tache 2, nous avons principalement programmé trois fonctions, la fonction "longueur_boucle" qui vérifie toutes les conditions de victoires, la fonction cases_satisafaites qui vérifie si chaque indice est satisfait, c'est à dire si les cases contenant 0,1,2 ou 3 sont bien entouré du bon nombre de segments. Enfin la fonction longueur boucle qui compte le nombre de segments que contient la boucle à l'aide des sommets : on prend le sommet "départ" comme point de départ qu'on va ensuite appeler "précédent", puis à l'aide du second sommet qui appartient au même segment que le sommet "précédent" qu'on va appeler "courant", on va chercher le segment tracé entre "courant" et l'autre sommet qui n'est pas "départ" (ou precedent). Ce nouveau sommet s'appellera "courant" et l'ancien sommet "courant" se nommera "precedent". On déterminera, au fur et à mesure, grâce à cette technique le nombre de tous les segments de la boucle

Pour la tache 3, on s'est occupé de l'interface graphique, nous avons utilisé l'interface ergonomique développée avec Fltk. Nous pouvons bien charger la grille du choix du joueur. La liste de liste indices est créer par les deux fonctions " recupere_indices_grille(fichier_text)" et "cst_indices(liste_indices_grille)" qui fonctionne en parallèle, si la grille est invalide un message d'erreur sera afficher dans la console. Nous avons ensuite fait en sorte que les clics soient détectés sur les "boutons". Nous avons donc fait une boucle qui délimite une zone pour chaque bouton.

Pour la tache 4, nous avons principalement implémenter un algorithme de recherche automatique de solution, c'est à dire un solveur. Celui-ci donne la solution à une grille donnée, c'est à dire de satisfaire tous les indices par une unique boucle fermée et, si c'est le cas, de dessiner la grille résolue.

Les parties obligatoires du projet ont été entièrement traité. Il n'y a pas de bug mais le solveur prend un peu de temps, en fonction de la taille de la grille choisie, à afficher la solution. On remarque également que lorsque l'on Stream en direct à l'aide de discord, et que l'on teste le solveur, celui-ci a tendance à prendre un peu plus de temps à afficher la réponse car le Stream utilise beaucoup de mémoire RAM. Les autres boutons tel que "Recommencer" et "Autres parties" présentent aussi le même problème.

Description des améliorations éventuelles

Comme nous l'avons dit précédemment, une amélioration éventuelle serait celle de la rapidité du solveur en réduisant la complexité, car celui-ci est un peu lent. Pour cela il aurait fallu programmer des fonctions n'ayant pas de boucles (ou moins de fonctions avec boucles), réduire les instructions simples mais c'est évident plus compliqué et cela nous demande plus de temps.

Fonctions importantes

statut_case(indices, etat, case) :

Permet de vérifier le statut d'une case, si l'indice de la case est None elle renvoie None on peut tracer tous les segments adjacents à la case, si le nombre de segments tracés dans Etat est égale à l'indice de la case, elle renvoie 0 et la case est satisfaite. Si l'on peut encore tracer des segments adjacents à la case dans Etat on renvoie 'positif'. S'il n'est plus possible de satisfaire l'indice ou trop de segments ont été tracé ou interdits autour de la case, elle renvoie 'négatif'.
On pose (i, j) = case avec i<m et j<n pour une grille m*n. Et on construit la liste segments_adjacents= [((i, j), (i+1, j)), ((i, j), (i, j+1)), ((i, j+1), (i+1, j+1)), ((i+1, j), (i+1, j+1))] si l'indice vaut None on retourne directement None sinon, on crée deux listes : segments_adjacents_trace=[] et segments_adjacents_interdit=[]. Si on trouve un segment tracé ou interdit dans etat on l'ajoute à l'une des listes à la fin et on compare l'indice de la case avec la taille des différentes listes obtenues pour avoir son statut.

longueur_boucle(etat, segment) :
On vérifie si le segment appartient à une boucle pour cela on utilise la fonction "ordonner(segment)" qui sert à arranger les sommets d'un segment dans l'ordre croissant, départ = segment [0] precedent = segment, courant = segment [1] comme on a un seul segment, on initialise la longueur à 1 tant que courant est différant de départ "while courant ! = depart :" lst= segments_traces(etat, courant) lst1=segments_traces(etat, depart) si la taille de lst1 ou celle de lst2 est différente de 2 on renvoie None puis on parcourt la liste des segments adjacent au sommet depart for i in range(len(lst)) :
On sait que len(lst) vaut nécessairement 2 pour chaque segment de lst si le segment est différent Du segment de départ correspond à courant on cherche le nouveau sommet différent de courant celui si devient courant et courant devient le precedent. A chaque tour de boucle la longueur augmente de 1 et à la fin, on retourne sa longueur.

cases_satisfaites(indices,etat):

Vérifie si chaque indice est satisfait : chaque case contenant un nombre k compris entre 0 et 3 a exactement k cotés tracés. Si oui elle renvoie True sinon False. Elle réunit les deux conditions de victoire

r_solutions(sommet, etat1, indices) :

Fonction récursive qui permet de trouver la solution de la grille et la stocké dans etat1, le dictionnaire est initialement vide selon l'algorithme du solveur naïve.

Explication de l'implémentation :
lst1 représente la liste des segments adjacents au sommet, n représente la taille de lst1.
Si n vaut 1 et les cases de la grille sont satisfaites dans etat1 alors on return true sinon return False
If n est supérieur à 2 on return False, si n == 0 ou n == 1, on créé une nouvelle liste lst qui contient tous les autres segments adjacents au sommet. Si l'indice des cases voisines permettent de tracer (**if**

seg_à_tracer(segment, etat1)) le segment, on l'ajoute dans etat1 et on détermine l'autre sommet du segment, celui qui est différents du sommet de départ et on appelle récursivement la fonction sur le nouveau sommet (som) résultat = r_solutions(som_suiv, etat1, indices) si le résultat est True on return True sinon on efface le segment qui vient d'être ajouté à etat1 et si aucune solution passe par le sommet on renvoie False.