Projet de programmation: Slitherlink

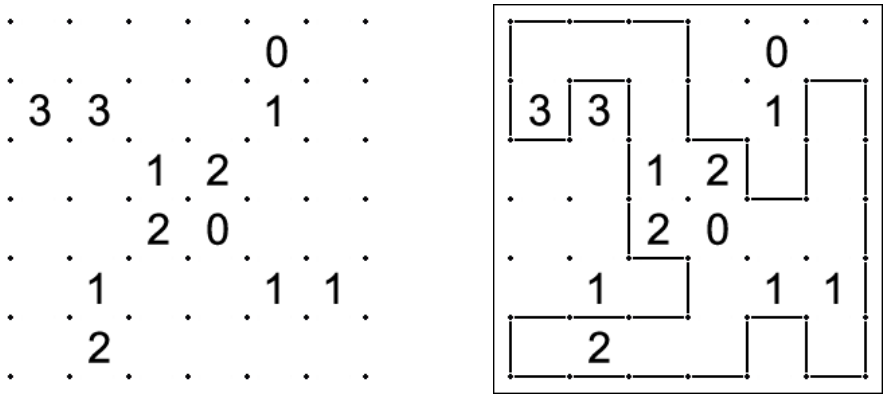
L'objectif de ce projet est d'implémenter un petit jeu de type "puzzle" et un algorithme de recherche automatique de solutions.

Le jeu Slitherlink

Slitherlink est un jeu de type casse-tête, se jouant sur une grille de nombres et d'inconnues, à la manière d'un Sudoku ou d'un démineur. Le but du jeu est de tracer les côtés des cases en respectant les règles suivantes :

- Un numéro dans une case indique le nombre de côtés de la case qui doivent être tracés : ni plus, ni moins ;
- Une case vide est une absence d'information, le joueur pourra tracer autant de côtés qu'il le souhaite ;
- L'ensemble des côtés tracés doit former une unique boucle fermée.

Voici un exemple de grille et sa solution (source : https://en.wikipedia.org/wiki/Slitherlink) :



La solution est bien constituée d'une unique boucle fermée, et on voit que chaque indice de case est respecté : autour des cases contenant un 0 aucun côté n'est tracé, un seul côté pour les cases contenant un 1, et ainsi de suite. On peut aussi se convaincre que c'est la seule solution possible : changer ne serait-ce qu'un segment dans le tracé conduirait à une impossibilité.

Pour résoudre une grille, on doit à l'aide de déductions logiques tracer certains des côtés, ou bien marquer des côtés comme ne devant pas être tracés. Quand on a déterminé qu'un côté doit ou ne doit pas être tracé, cela permet en général de faire de nouvelles déductions, et ainsi de suite jusqu'à compléter la grille. Un problème bien conçu ne doit avoir *qu'une seule solution*.

Pour bien comprendre le principe du jeu et vous familiariser avec sa présentation et ses règles, vous pouvez jouer quelques parties sur un site de jeu en ligne, par exemple celui-ci.

Objectif

L'objectif principal de ce projet est de réaliser un programme (graphique) permettant de jouer à ce jeu. L'utilisateur pourra tracer les côtés des cases ou les effacer par un simple clic gauche. Un clic droit permettra d'indiquer qu'une ligne ne doit pas être tracée.

L'interface devra présenter de façon claire, par un changement de couleur par exemple, quelles sont les indices qui sont satisfaits, et quels sont ceux qui ne le sont pas encore. Elle doit également afficher un message quand une grille est entièrement résolue.

Outre l'implémentation du jeu lui-même, un deuxième objectif du projet réside dans la programmation d'un *solveur*, ou générateur automatique de solutions. Un appui sur une touche du clavier déclenchera le calcul d'une solution du jeu, et son affichage.

Vocabulaire et notations

On donne ici un nom précis aux différents éléments d'une grille de jeu.

Grille. Une grille de slitherlink est présentée sous la forme d'un tableau rectangulaire de cases, ou encore d'une matrice rectangulaire.

Une grille à m lignes et n colonnes est appelée "grille $m \times n$", m est la *hauteur* de la grille et n sa *largeur*.

Case. Une *case* est l'un des "carrés" d'une grille, autrement dit l'un des éléments du tableau ou de la matrice qui le représente.

On notera $c_{i,j}$ la case se situant à l'intersection de la i-ème ligne et de la j-ème colonne (en comptant à partir de 0). La case tout en haut à gauche est donc la case $c_{0,0}$, celle tout en bas à gauche est $c_{m-1,0}$, et la case tout en bas à droite est $c_{m-1,n-1}$.

Indice. Un indice est un nombre compris entre 0 et 3 porté par une case.

Chaque case d'une grille peut soit porter un nombre compris entre 0 et 3, appelé *indice*, soit ne porter aucune information. Une case portant un indice indique combien de ses côtés (ou *segments adjacents*) doivent être tracés dans toute solution du problème.

Par exemple, si une case porte l'indice 2 alors deux exactement de ses segments adjacents devront être tracés pour que le problème soit résolu.

Une case ne comportant pas d'indice n'impose rien sur le nombre de ses segments adjacents qui peuvent être tracés.

Sommet. Un *sommet* correspond à un coin de l'une des cases de la grille.

Tout comme les cases, les sommets sont représentés par leurs coordonnées, cependant dans une grille $m \times n$ il y a $m + 1$ lignes de sommets (numérotées de 0 à m) et $n + 1$ colonnes (numérotées de 0 à n).

On note $s_{i,j}$ le sommet situé sur la i-ème ligne et la j-ème colonne de sommets. Ce sommet correspond au coin en haut à gauche de la case $c_{i,j}$ si $i < m$ et $j < n$, sinon c'est le coin en bas à droite de la case $c_{i-1,j-1}$ (sommets de la dernière ligne ou de la dernière colonne).

Chaque case possède quatre sommets adjacents. Les sommets adjacents à la case $c_{i,j}$ sont les sommets $s_{i,j}$, $s_{i,j+1}$, $s_{i+1,j+1}$ et $s_{i+1,j}$.

Segment. On appelle *segment* le côté d'une case de la grille.

Chaque segment est repéré par le couple de *sommets* qui sont à ses extrémités. Chaque case est bien sûr adjacente à quatre segments qui sont ses côtés.

Par exemple, le segment $[s_{1,4}, s_{2,4}]$ correspond au côté gauche de la case $c_{1,4}$.

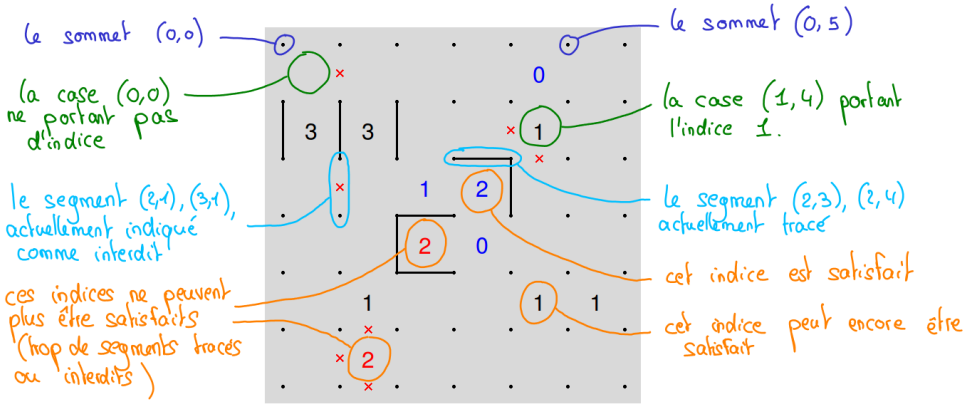
Un segment possède trois états possibles : *vierge*, *tracé* ou *interdit*. Un segment vierge n'est simplement pas représenté graphiquement. Un segment tracé est représenté par un trait plein, dans une couleur quelconque. Un segment interdit est représenté par une croix rouge (×) dessinée au milieu du segment.

Boucle. Une *boucle* est un ensemble de segments formant un chemin fermé.

Plus précisément, on appelle boucle un ensemble E de segments dont chaque élément est adjacent à exactement deux autres éléments de l'ensemble E (ni plus, ni moins). Intuitivement une boucle forme donc un "chemin" permettant de passer de sommet en sommet en suivant des segments et de revenir à son point de départ.

État. L'*état* d'une grille est constitué de l'état de chacun de ses segments.

Il s'agit d'indiquer, pour chaque segment de la grille, s'il est vierge, tracé ou interdit. On verra dans la section sur la représentation de l'état comment cette information peut être représentée en Python.



Réalisation

Le projet se décompose en quatre tâches principales (toutes obligatoires).

Tâche 1: Structures de données

Chargement de la grille

Une grille de slitherlink peut être représentée par un fichier texte comme suit:

```
$ cat grille1.txt
____0_
33__1_
__12__
__20__
_1__11
_2____
```

(La ligne `$ cat grille1.txt` ne fait pas partie du fichier.)

Ce contenu correspond à la grille utilisée comme exemple plus haut. La grille est décrite case par case, de la gauche vers la droite puis du haut vers le bas. Chaque caractère du fichier décrit le contenu d'une des cases de la grille : `_` représente l'absence d'information, et les nombres correspondent aux indices.

Rappel : une grille n'est pas forcément carrée, elle peut aussi avoir une forme rectangulaire.

Le programme réalisé doit être capable de lire des fichiers écrits dans le format spécifié ci-dessus. Les grilles proposées sur la page du projet respectent ce format et vous pourrez donc vous en servir pour tester votre programme.

Le programme doit également détecter d'éventuelles erreurs dans le format d'un fichier (par exemple : présence d'un caractère interdit, lignes de longueurs différentes, etc.), et les signaler par un message d'erreur sur la console.

La représentation des grilles dans la logique interne du programme est décrite dans la section suivante.

Représentation de l'état du jeu

L'état du jeu est représenté par deux informations :

- Les valeurs et positions des indices, représentées par une liste de listes d'entiers ;
- Les déductions faites par le joueur pour chaque segment : tracé, vide ou interdit.

Par exemple, les indices de la grille donnée en introduction seront représentés comme suit :

```
indices = [[None, None, None, None, 0   , None],
           [3   , 3   , None, None, 1   , None],
           [None, None, 1   , 2   , None, None],
           [None, None, 2   , 0   , None, None],
           [None, 1   , None, None, 1   , 1   ],
           [None, 2   , None, None, None, None]]
```

Les sommets de la grille (c'est-à-dire les coins des cases), seront représentés par leurs coordonnées dans la grille, donc par des *couples d'entiers*. Par exemple, le sommet $s_{i,j}$ sera représenté en Python par le couple `(i, j)`.

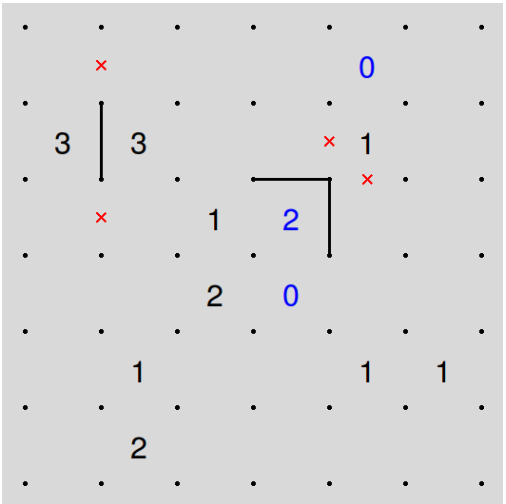
Un segment étant défini par la donnée de deux sommets adjacents, on utilisera donc un *couple de sommets* pour les représenter en Python, c'est-à-dire un *couple de couples d'entiers*. Par exemple, le segment $[s_{1,2}, s_{1,3}]$ sera représenté en Python par le couple `((1, 2), (1, 3))`.

Pour représenter l'état (vierge, tracé ou interdit) des segments au cours de la partie, on utilise un dictionnaire dont les clés sont les segments et dont la valeur associée à chaque segment est `1` si le segment est tracé et `-1` s'il est interdit. Les segments vierges ne figureront tout simplement pas dans le dictionnaire.

Remarque importante: Chaque segment peut être représenté de deux façons différentes. Par exemple, les deux écritures $[s_{1,2}, s_{1,3}]$ et $[s_{1,3}, s_{1,2}]$ font référence au même segment : celui qui se trouve entre les sommets $s_{1,2}$ et $s_{1,3}$. Afin de ne pas dédoubler l'information, on choisira toujours de représenter les segments dans le dictionnaire en donnant en premier le sommet le plus petit dans l'ordre lexicographique.

Par exemple, le dictionnaire `etat` représente l'état de la grille ci-dessous :

```
etat = {
    ((0, 1), (1, 1)) : -1
    ((1, 1), (2, 1)) : 1
    ((2, 1), (3, 1)) : -1
    ((1, 4), (2, 4)) : -1
    ((2, 4), (2, 5)) : -1
    ((2, 3), (2, 4)) : 1
    ((2, 4), (3, 4)) : 1
}
```



Fonctions d'accès

Pour faciliter le traitement de cette partie du projet, il est fortement recommandé d'écrire (au moins) les groupes de fonctions suivantes.

Le premières fonctions demandées reçoivent toutes en paramètre un dictionnaire `etat` décrivant l'état de la grille (selon la structure décrite ci-dessus) et un couple de sommets `segment` :

- `est_trace(etat, segment)` renvoyant `True` si `segment` est tracé dans `etat`, et `False` sinon ;
- `est_interdit(etat, segment)` renvoyant `True` si `segment` est interdit dans `etat`, et `False` sinon ;
- `est_vierge(etat, segment)` renvoyant `True` si `segment` est vierge dans `etat`, et `False` sinon ;
- `tracer_segment(etat, segment)` modifiant `etat` afin de représenter le fait que `segment` est maintenant tracé ;
- `interdire_segment(etat, segment)` modifiant `etat` afin de représenter le fait que `segment` est maintenant interdit ;
- `effacer_segment(etat, segment)` modifiant `etat` afin de représenter le fait que `segment` est maintenant vierge. **Attention,** effacer un segment revient à *retirer* de l'information du dictionnaire `etat`.

On implémentera également les fonctions suivantes, recevant en paramètre un dictionnaire `etat` comme précédemment, ainsi qu'un couple `sommet` représentant un sommet :

- `segments_traces(etat, sommet)`, renvoyant la liste des segments tracés adjacents à `sommet` dans `etat`;
- `segments_interdits(etat, sommet)`, renvoyant la liste des segments interdits adjacents à `sommet` dans `etat`;
- `segments_vierges(etat, sommet)`, , renvoyant la liste des segments vierges adjacents à `sommet` dans `etat`.

Enfin, on suggère d'implémenter la fonction suivante :

- `statut_case(indices, etat, case)` recevant le tableau d'indices, l'état de la grille et les coordonnées d'une *case* (pas d'un sommet !) et renvoyant `None` si cette case ne porte aucun indice, et un nombre entier sinon :
  - 0 si l'indice est satisfait ;
  - positif s'il est encore possible de satisfaire l'indice en traçant des segments autour de la case ;
  - négatif s'il n'est plus possible de satisfaire l'indice parce que trop de segments sont déjà tracés ou interdits autour de la case.

Ainsi le travail de manipulation des segments et de l'état de la grille sera confiné à **ces fonctions** et devra être invisible au reste du programme. En particulier, il est de la responsabilité de ces fonctions d'ordonner correctement les segments passés en paramètre (avec le plus petit sommet en premier) avant de lire ou d'écrire dans le dictionnaire `etat`. Ainsi, du point de vue du reste du programme, il n'y aura pas de différence entre consulter l'état du segment `((a, b), (c, d))` et celui du segment `((c, d), (a, b))`.

Tâche 2 : conditions de victoire

La deuxième tâche du projet consiste à détecter automatiquement la fin de la partie. La grille est résolue et le joueur a gagné lorsque les deux conditions suivantes sont réunies :

1. Chaque indice est satisfait : chaque case contenant un nombre k compris entre 0 et 3 a exactement k côtés tracés.
2. L'ensemble des segments tracés forme une unique boucle fermée.

La première condition ne pose pas de difficultés particulières. Pour tester la deuxième condition, une possibilité est, en partant d'un segment tracé quelconque, de tenter de suivre un chemin à partir d'une des extrémités du segment, tout en comptant le nombre de segments traversés. Si l'on parvient à revenir au point de départ, il suffit ensuite de déterminer si l'on a bien compté tous les segments tracés ou non. Si l'on ne peut pas revenir au point de départ, c'est que le segment de départ n'appartient pas à une boucle.

On peut donc par exemple programmer une fonction `longueur_boucle(etat, segment)` renvoyant `None` si le segment n'appartient pas à une boucle, et la longueur de la boucle à laquelle il appartient sinon. La fonction pourra implémenter l'algorithme suivant :

1. Appeler `depart` l'une des extrémités de `segment`.
2. Appeler `precedent` le sommet de départ, et `courant` l'autre extrémité du segment.
3. Tant que le sommet `courant` n'est pas identique au sommet `depart` :
    a. Déterminer les segments tracés autour du sommet `courant`.
    b. S'il n'y en a pas exactement deux, renvoyer `None`.
    c. Sinon, déterminer le sommet adjacent à `courant` qui ne soit pas égal à `precedent`.
    d. Appeler `precedent` l'actuel sommet courant, et `courant` le nouveau sommet, et recommencer.
4. Si l'on sort naturellement de la boucle, renvoyer le nombre de segments parcourus.

Pour déterminer si *tous* les segments tracés appartiennent à la même boucle, il pourra être utile de maintenir dans une variable globale du programme le nombre total de segments tracés, et comparer ce nombre au résultat de la fonction `longueur_boucle`.

Affichage des conditions de victoire

À chaque coup joué, le programme devra afficher un message sur la console indiquant le statut de chacune des deux conditions de victoires (indices tous satisfaits ou non, segments tracés formant une boucle unique ou non).

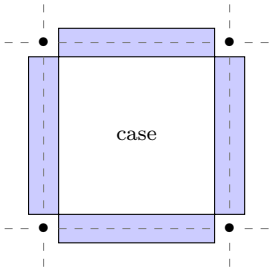
Tâche 3: Interface graphique

Une interface ergonomique développée avec `fltk` est attendue pour cette étape. Il faudra au minimum permettre au joueur de charger la grille de son choix parmi un ensemble de grilles disponibles et afficher la grille et ses indices. Le joueur pourra ensuite par un simple clic gauche tracer un segment, et par un clic droit marquer un segment comme interdit. N'importe quel clic (gauche ou droit) sur un segment déjà tracé ou interdit aura pour effet d'effacer la déduction.

Pour aider le joueur au cours de la partie, les indices devront être colorés selon le nombre de segments tracés les entourant : en bleu s'il y a exactement le bon nombre de segments, en noir s'il n'y en a pas encore assez, et en rouge s'il y en a trop. Le jeu devra automatiquement détecter la victoire, et afficher un message félicitant le joueur, accompagné d'un menu proposant de quitter, de recommencer au début ou de charger une autre grille.

Détection des clics

La partie la plus délicate de cette tâche est la détection des clics sur les segments. Pour plus de facilité, on considérera qu'un clic de souris tombe sur un segment quand ses coordonnées appartiennent à une certaine zone rectangulaire autour du segment, comme représenté sur la figure suivante (zones de clic en bleu) :



Comme il est hors de question de créer une longue liste de coordonnées (on ne connaît pas à l'avance la taille de la grille), il faut utiliser des opérations arithmétiques de manière judicieuse (en particulier des divisions et modulos) afin de déterminer si les coordonnées d'un clic tombent à proximité d'un segment.

Une stratégie possible est la suivante.

Si l'on considère que les cases de la grille font toutes la même taille `taille_case`, et que la grille est dessinée avec une marge en haut et une marge à gauche de taille `taille_marge`, alors les coordonnées du sommet $s_{i,j}$ sont exactement `(taille_marge + j * taille_case, taille_marge + i * taille_case)`.

Ainsi, un clic de souris à la position `(x, y)` est proche d'un segment vertical si `x` est proche de `taille_marge + j * taille_case` pour un certain `j` compris entre 0 et n, mais `y` n'est *pas* proche d'une valeur de la forme `taille_marge + i * taille_case` pour un certain `i` compris entre 0 et m.

Pour déterminer cette proximité, on pourra par exemple calculer `dx = (x - taille_marge) / taille_case`, et regarder si la valeur obtenue est «proche d'un nombre entier» ou pas (et de même pour `y`), c'est à dire par exemple si `-0.2 < dx - round(dx) < 0.2`.

Tâche 4: Recherche de solutions

La quatrième tâche du projet consiste à implémenter un algorithme de recherche automatique de solution, ou *solveur*, pour le jeu Slitherlink. Le rôle du solveur est de déterminer s'il est possible de résoudre 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.

Le solveur de base ne raisonne pas comme un être humain. Plutôt que d'essayer de déduire des informations sûres, le solveur va simplement essayer de construire la boucle en énumérant toutes les possibilités dans l'ordre, et annuler son dernier coup pour en essayer un autre à chaque fois qu'il arrive à une incohérence. Cet algorithme (appelé algorithme de recherche *par backtracking*) est très similaire à celui vu en TD pour résoudre le *problème du sac à dos*.

Supposons que le solveur a déjà tracé un chemin qui l'a emmené jusqu'à un sommet (i, j). Pour poursuivre ce chemin, le solveur essaie de tracer l'un des trois segments sortant de ce noeud, puis effectue un appel récursif sur le nouveau sommet (i', j') atteint. Si l'appel récursif répond vrai, alors une solution a été trouvée. Sinon, on efface le dernier segment tracé, et on recommence avec un des segments restants. Si les trois segments ont été essayés sans succès, c'est qu'il n'existe pas de solution à partir du chemin déjà tracé.

L'algorithme peut être décrit de la même manière que le parcours de boucle de la tâche 2 à la différence que la boucle à parcourir **n'est pas encore tracée** et que l'algorithme va simplement la deviner au fur et à mesure du calcul. L'algorithme procède donc comme suit, à partir d'un `sommet` de départ et d'un ensemble de `etat` initialement vide :

- Si `sommet` est adjacent à deux segments tracés dans `etat`, la boucle est bouclée. Il ne reste plus qu'à vérifier que tous les indices sont satisfaits. Si oui, la grille est résolue et on renvoie `True`, sinon il n'est plus possible de compléter la solution et on renvoie `False`.
- Si `sommet` est adjacent à plus de deux segments tracés dans `etat`, on a créé un branchement. La solution est erronée et on renvoie `False`.
- Si `sommet` n'est adjacent à aucun (ce qui n'arrivera qu'au tout début) ou à un seul segment tracé dans `etat`, pour chacun des autres segments adjacents à `sommet` :
  - On trace le segment dans `etat` en vérifiant qu'il n'enfreint pas les indices par excès (c'est à dire ne dépasse pas le nombre de segments autorisés par un indice voisin).
  - On rappelle récursivement l'algorithme sur l'autre extrémité du segment rajouté.
  - Si l'appel récursif renvoie `True`, alors une solution a été trouvée, et on renvoie `True`.
  - Si l'appel récursif renvoie `False`, alors il n'existe pas de solution à partir du chemin déjà tracé. On retire de `etat` le segment que l'on vient d'ajouter et on passe au segment suivant.
- Si aucun segment partant de `sommet` ne permet de continuer, alors il n'est plus possible de compléter la solution déjà tracée, et on renvoie `False`.

Si l'algorithme renvoie `True`, alors une solution a été trouvée et est représentée dans `etat`. Il ne reste plus qu'à l'afficher.

Remarque : cette technique de recherche part d'un sommet en particulier. Il se peut que la solution de la grille **ne passe pas par ce sommet**. Dans ce cas, l'algorithme renverra `False` et il faudra relancer la recherche à partir d'un autre sommet, jusqu'à les avoir énumérés tous. Si

on s'y prend de façon naïve, cela peut représenter un travail supplémentaire conséquent, mais il suffit de raisonner un peu pour se rendre compte que :

- Le chemin passe forcément par tous les sommets autour d'un indice "3", donc n'importe quel sommet de ce type sera forcément un bon point de départ.
- Le chemin passe au minimum par trois des quatre sommets autour d'un indice "2", donc si on ne trouve pas d'indice "3", un sommet autour d'un "2" sera probablement un bon point de départ, et si ce sommet-là échoue, n'importe quel autre sommet autour du même indice sera le bon.
- Le chemin passe au minimum par une des deux extrémités des diagonales des indices "1", donc si on ne trouve ni de sommet "3", ni de sommet "2", un sommet autour d'un "1" aura une chance sur deux d'être un bon point de départ, et sinon, le sommet opposé sera le bon.
- Si la grille ne contient ni indice 1, ni indice 2, ni indice 3, ce n'est vraisemblablement pas une grille très intéressante !

Tâches complémentaires (optionnelles)

Cette section propose des améliorations du projet, à n'aborder que si la partie principale du projet est *entièrement terminée*.

Solveur graphique : Cette amélioration consiste à réaliser un mode graphique du solveur, traçant et effaçant à nouveau les segments dans la fenêtre de jeu au fur et à mesure de la recherche. Les étapes de recherche peuvent être animées automatiquement (à une vitesse plus ou moins grande) ou s'afficher pas à pas par appui sur une touche.

Attention, l'affichage ralentit bien sûr énormément la recherche, c'est pourquoi le mode graphique est seulement une **option** qu'il doit être facile de désactiver au besoin.

Solveur sur une grille partielle et fonction "Indice" : Le solveur proposé dans la partie obligatoire suppose que la grille de départ est vide. On peut améliorer le solveur en permettant de lancer une recherche de solution sur une grille déjà partiellement remplie. Dans ce cas, le solveur doit essayer de compléter la grille en cours, *sans jamais retirer un segment déjà tracé par le joueur*.

Cette amélioration permet d'implémenter une fonction "Aide" pour aider un joueur qui se retrouverait bloqué sur une grille. Lorsque le joueur demande de l'aide, le programme essaie de résoudre la grille en cours. S'il y parvient, il indique au joueur l'un des segments qu'il lui manque pour arriver à une solution. Sinon, un message s'affiche pour indiquer au joueur qu'il a commis une erreur et que la grille actuelle n'admet plus de solution.

Astuces de résolution et amélioration du solveur : Une première amélioration simple du solveur consiste à marquer comme "interdit" tout segment qui produirait un branchement lors de la construction de la boucle. Cette amélioration permet d'arrêter le calcul lorsqu'un indice a trop de segments interdits pour être encore satisfait. Par exemple lorsqu'un indice "3" a déjà deux segments interdits, il est inutile de continuer la recherche.

Malgré tout, le solveur de base décrit dans le sujet peut tout de même être très lent. Une amélioration plus fine consiste à utiliser des stratégies de résolution comme celles décrites sur cette page Wikipedia. Le but de cette amélioration est d'intégrer ces stratégies dans

le calcul du solveur, pour pouvoir fixer sans aucun doute le statut (tracé ou interdit) de certains segments. On peut évidemment ajouter d'autres stratégies si l'on en découvre…

On ne lancera alors le calcul exhaustif du solveur qu'une fois que toutes les stratégies auront été épuisées, pour compléter les dernières zones restantes, que l'on espère alors peu nombreuses ou déjà très contraintes. Cette deuxième approche nécessite d'avoir traité l'amélioration *solveur sur une grille partielle*.

Points de sauvegarde : technique du "changement de couleur" : Une façon de progresser dans une grille de Slitherlink difficile consiste à tester une hypothèse, généralement dans le but d'arriver à une incohérence et de conclure que l'hypothèse inverse devait être juste. Quand ils ont recours à cette méthode, les joueurs de Slitherlink ont l'habitude de "changer de couleur" afin de pouvoir distinguer les conséquences de leurs hypothèses : en général, les segments sûrs sont tracés au stylo, et les hypothèses au crayon à papier afin de pouvoir être effacées lorsque l'incohérence est trouvée.

On veut améliorer cette technique en permettant au joueur de faire un point de sauvegarde : toutes les déductions faites après le point de sauvegarde seront tracées dans une nouvelle couleur. Ensuite le joueur pourra en une seule commande soit effacer toutes les nouvelles déductions faites depuis le point de sauvegarde (lorsqu'il s'est convaincu que l'hypothèse est fausse), soit oublier le point de sauvegarde et convertir toutes les déductions dans la couleur principale (lorsqu'il s'est convaincu que l'hypothèse est juste).

Rien n'empêche de raffiner **encore** cette amélioration en permettant d'imbriquer des points de sauvegarde dans des points de sauvegarde, autant que nécessaire ! Ainsi, le joueur pourra à chaque moment oublier toutes les déductions du point de sauvegarde en cours, ou bien convertir les déductions en cours vers la couleur du point de sauvegarde parent, chaque point de sauvegarde étant tracé avec une couleur différente.

Autres améliorations suggérées : Voici quelques autres suggestions d'améliorations moins détaillées. Attention, certaines sont plutôt faciles tandis que d'autres sont assez difficiles. N'hésitez pas à demander conseil à vos enseignants avant de vous lancer.

- Implémenter une fonction "Annuler", permettant au joueur d'annuler son dernier coup. Cette fonctionnalité nécessite de stocker l'historique des coups du joueur, afin de permettre l'annulation de plusieurs coups successifs : n annulations succéssives doivent annuler les n derniers coups joués.
- Permettre au joueur d'enregistrer une partie en cours ou de charger une partie enregistrée. L'enregistrement devra être fait dans un fichier de manière à pouvoir être récupéré lors d'une autre session de jeu. Il faut pour cela modifier la représentation des données dans le fichier afin de permettre l'enregistrement des déductions du joueur.
- Modifier le solveur pour qu'il renvoie la liste de toutes les solutions possibles à une grille donnée (en principe une grille bien faite n'a qu'une seule solution, mais cela n'est pas garanti a priori).
- Implémenter un générateur de grilles aléatoires résolubles (c'est-à-dire pour lesquelles il existe une solution, si possible unique).
- Implémenter un système de scores en nombre de coups et temps passé sur la grille. Le score du joueur devra être annoncé à la fin de la partie et la liste des meilleurs scores pour chaque grille sera conservée dans un fichier.

Toute autre amélioration est envisageable selon vos idées et envies, à condition d'en discuter au préalable avec un de vos enseignants.