Projet de programmation: Hitori

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

Le jeu de Hitori

Le descriptif qui suit est tiré de la page Wikipedia.

Hitori (diminutif de ひとりにしてくれ, *hitori ni shite kure*, littéralement « laissez-moi seul » ou « laissez-moi tranquille ») est un jeu de logique d'origine japonaise. Hitori a été publié pour la première fois dans *Puzzle Communication Nikoli #29*, en mars 1990.

Règles du jeu

Le jeu de Hitori se joue sur une grille rectangulaire dont chaque cellule contient une valeur (par exemple un nombre). Le but du jeu est de noircir certaines des cellules de sorte que:

1. Parmi les cellules visibles (non noircies), chaque nombre ne peut apparaître qu'une seule fois au plus sur chaque ligne et chaque colonne;
2. Deux cellules noircies ne peuvent se toucher par un côté (mais leurs coins peuvent se toucher);
3. L'ensemble des cellules visibles doit être d'un seul tenant (il ne peut pas y avoir deux zones visibles distinctes non reliées entre elles).

On dira que deux cellules sont *en conflit* si elles portent le même nombre et se trouvent sur la même ligne ou sur la même colonne. La règle 1 signifie donc qu'une grille est résolue si aucune cellule n'est en conflit avec une autre (et si les règles 2 et 3 sont également respectées).

Les figures 1 et 2 montrent un exemple de grille et sa solution.

2	2	1	5	3
2	3	1	4	5
1	1	1	3	5
1	3	5	4	2
5	4	3	2	1

Figure 1: Exemple de grille de hitori. Certaines cellules sont en conflit, par exemple les deux premières cellules de la première ligne, qui contiennent le nombre 2.

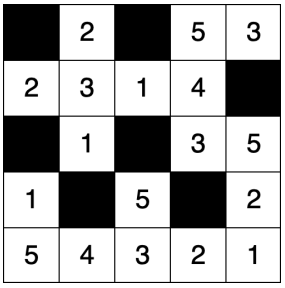

Figure 2: Solution de la grille de la figure 1. On voit que tous les conflits ont disparu, qu'aucune cellule noire n'est adjacente à une autre, et que les cellules visibles forment une seule zone.

Objectif du projet

L'objectif principal de ce projet est de réaliser un programme (graphique) permettant de jouer au jeu de Hitori. L'utilisateur pourra noircir les cellules ou les rendre à nouveau visibles par un simple clic gauche. Le programme doit pouvoir indiquer au joueur si la grille est entièrement résolue ou s'il reste des problèmes (élément en double sur une ligne ou une colonne, cellules noires voisines ou non-connexité des cellules visibles).

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.

Réalisation du projet

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

Tâche 1: Représentation et chargement des niveaux

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

```
$ cat grille.txt
2 2 1 5 3
2 3 1 4 5
1 1 1 3 5
1 3 5 4 2
5 4 3 2 1
```

Ce contenu correspond à la grille utilisée comme exemple plus haut. Chaque ligne consiste simplement en la suite des nombres contenus dans les cellules d'une rangée de la grille, séparés par des espaces.

Le programme réalisé doit être capable de lire des fichiers écrits dans le format spécifié ci-dessus. Il sera ainsi possible de charger les fichiers proposés sur la page du projet, d'échanger des grilles,

et pourquoi pas de consistuer collectivement une bibliothèque de grilles les plus difficiles ou intéressantes.

La première tâche du projet consiste à programmer les quatre fonctions suivantes : - `lire_grille(nom_fichier)` prenant en argument une chaîne de caractères `nom_fichier` et renvoyant une liste de listes décrivant les valeurs des cellules de la grille, - `afficher_grille(grille)` prenant en argument une liste de listes représentant une grille, et l'affichant joliment sur le terminal, - `ecrire_grille(grille, nom_fichier)` prenant en argument une grille sous forme de liste de listes de nombres et un nom de fichier, et sauvegardant la grille fournie dans la fichier indiqué, en respectant le même format.

Remarques:

- Une grille n'est pas forcément carrée, mais peut être rectangulaire;
- Si le fichier fourni n'est pas bien formé (par exemple s'il contient des valeurs inconnues ou des lignes de longueurs différentes), la fonction de lecture pourra provoquer une erreur ou renvoyer la valeur `None`.

Tâche 2: Réalisation du moteur de jeu

La seconde tâche du projet consiste à programmer la logique interne du jeu, c'est-à-dire la partie qui permet de manipuler la structure de donnée interne représentant l'état de la grille pour simuler le noircissement d'une cellule, déterminer si une grille est résolue, et vérifier si les règles ont bien été respectées.

Pour représenter l'ensemble des cellules noircies d'une grille, on utilisera un objet de type `set` (voir les rappels fournis en fin de sujet) dont les éléments sont des couples de coordonnées de cellules de la forme `(i, j)`, `i` étant le numéro de ligne et `j` le numéro de colonne de la cellule. Un tel objet peut décrire aussi bien l'état actuel d'une grille que sa solution. Par exemple, la solution à la grille de l'exemple donné plus haut pourrait être l'ensemble

```
{(2, 0), (0, 0), (3, 3), (2, 2), (3, 1), (0, 2), (1, 4)}
```

On devra dans cette partie réaliser (au moins) la liste de fonctions suivante. Chacune de ces fonctions reçoit en argument au moins une liste de listes `grille` décrivant le contenu des cellules, et un ensemble `noircies` décrivant l'ensemble des cellules noircies.

Il est bien sûr possible de créer d'autres fonctions, mais les fonctions demandées devront être parfaitement opérationnelles et respecter exactement les signatures indiquées (noms des fonctions, noms et types des paramètres, valeurs de retour demandées).

- Fonction `sans_conflit(grille, noircies)` renvoyant `True` si la règle du jeu numéro 1 est respectée, autrement dit si aucune des cellules visibles de la grille ne contient le même nombre qu'une autre cellule visible située sur la même ligne ou la même colonne, et `False` sinon.
- Fonction `sans_voisines_noircies(grille, noircies)` renvoyant `True` si la règle du jeu numéro 2 est respectée, autrement dit si aucune cellule noircie n'est voisine (par un de ses quatre bords) d'une autre cellule noircie, et `False` sinon.

- Fonction `connexe(grille, noircies)` renvoyant `True` si la règle du jeu numéro 3 est respectée, autrement dit si les cellules visibles de la grille forment une seule zone (ou région, ou composante connexe), et `False` sinon.

  On rappelle que la notion de zone est comprise au sens du coloriage de zone vu en cours (chapitre sur la récursivité), c'est-à-dire qu'au sein d'une zone on peut passer de toute cellule visible à toute autre en travsersant uniquement des cellules visibles, et en ne passant d'une cellule à l'autre que par l'un des quatre côtés (deux cellules se touchant par un coin ne sont pas considérées comme reliées).

  Pour cette fonction, il est donc conseillé d'adapter l'algorithme de coloration de zone.

Tâche 3: interface graphique

Une interface ergonomique 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, afficher un message de félicitations en cas de victoire, et afficher un menu proposant au joueur de quitter, de recommencer au début ou de charger une autre grille.

L'interface devra également proposer une fonction "Annuler", permettant de revenir en arrière d'un coup. Pour mettre cette fonction en place, il sera nécessaire de mémoriser (par exemple dans une liste) l'historique du jeu, c'est-à-dire la suite ordonnée des cellules noircies ou dévoilées depuis le début de la partie. Ainsi, lorsque le joueur veut annuler son dernier coup, il suffit d'effacer la dernière entrée de l'historique, et de rétablir la visibilité de la cellule correspondante.

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 de Hitori. On ne s'intéresse pour l'instant qu'à la réalisation d'un solveur simple, permettant de déterminer quelles sont les configurations *gagnantes*, c'est-à-dire celles à partir desquelles il existe une solution, et inversement, quelles sont les configurations *perdantes*, celles à partir desquelles il n'est pas possible de gagner.

Nous allons commencer par implémenter un algorithme de recherche naïf (du même type que l'algorithme du monnayeur vu en cours, appelé algorithme de *backtracking*, ou algorithme de recherche en profondeur), qui depuis un état donné du jeu (qu'on appelle une *configuration*) va considérer chaque cellule une par une, et explorer récursivement le reste de la grille, d'abord en supposant que cette cellule est noircie, puis qu'elle reste visible.

Commençons par remarquer que puisque le contenu des cellules n'est jamais modifié, une configuration du jeu est entièrement décrite par l'ensemble `noircies`.

> **Algorithme de recherche (pseudo-code).** L'algorithme s'exécute depuis une position `(i, j)` et avec un ensemble `noircies` de cellules déjà noircies. Il effectue les tâches suivantes:
>
> 1. Si la grille est invalide (une des règles est enfreinte), alors la partie est perdue: on ne pourra plus gagner en noircissant davantage de cellules. Le solveur renverra `None` pour signaler que cette hypothèse a échoué.

2. Si la grille est résolue, l'ensemble `noircies` est une solution, on le renvoie tel quel.

3. Sinon, on considère la cellule `(i, j)`. Dans toute solution de la grille, cette cellule sera soit visible, soit noircie.

   a. Si la cellule n'est en conflit avec aucune autre cellule (aucune autre cellule sur la ligne `i` ou la colonne `j` ne possède le même chiffre), alors on la laisse visible et on passe à la cellule suivante (par exemple la cellule `(i, j+1)`, ou `(i+1, 0)` si l'on est en bout de ligne);

   b. Sinon :

      - on noircit la cellule `(i, j)` et on cherche une solution à partir de la cellule suivante;
      - si une solution est trouvée, on la renvoie;
      - sinon on rend à nouveau visible la cellule `(i, j)` et on cherche une solution à partir de la cellule suivante.

4. Si aucune solution n'est trouvée (que la cellule `(i, j)` soit noircie ou visible), alors il ne peut exister de solution depuis la configuration actuelle, il faut donc renvoyer `None`.

Écrire une fonction `resoudre(grille, noircies, i, j)` qui implémente l'algorithme ci-dessus. La fonction renverra l'ensemble (type `set`) des cellules à noircir dans la solution, ou `None` si aucune solution n'existe étant donnés les arguments reçus.

Tâches optionnelles

Solveur graphique

Le solveur propose un mode graphique, noircissant et rendant à nouveau visibles dans la fenêtre de jeu les cellules visitées au fur et à mesure de la recherche. 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 orienté par les conflits

Plutôt que d'envisager le statut (noircie ou visible) de chaque cellule une par une, il est possible de modifier le solveur afin qu'il ne s'intéresse qu'aux cellules pour lesquelles un conflit existe (en effet, les autres cellules pourront toujours rester visibles dans la solution de la grille).

Afin de pouvoir faire fonctionner cet algorithme on pré-calcule, en passant en revue chaque ligne et chaque colonne, tous les ensembles de cellules en conflit entre elles, c'est-à-dire tous les ensembles de cellules se situant sur la même ligne ou colonne et possédant le même chiffre. Il faut bien sûr choisir un type de données Python approprié pour cela.

Une observation simple est que, dans un ensemble de cellules en conflit deux à deux, une seule cellule sera visible dans la solution, toutes les autres devant être noircies. On propose donc le nouvel algorithme suivant :

Algorithme de recherche orienté par les conflits (pseudo-code). L'algorithme exécuté depuis un ensemble `noircies` procède comme suit:

1. Si la grille est invalide (une des règles est enfreinte), alors la partie est perdue: on ne pourra plus gagner en noircissant davantage de cellules. Le solveur renverra `None` pour signaler que cette hypothèse a échoué.

2. Si la grille est résolue, l'ensemble `noircies` est une solution, on le renvoie tel quel.

3. Sinon, on choisit parmi les conflits pré-calculés un ensemble `doublons` de cellules en conflit, et pour chacune tour à tour:
   - on noircit toutes les cellules sauf celle-là et on cherche une solution depuis la nouvelle configuration obtenue;
   - si une solution est trouvée, on la renvoie;
   - sinon, on rétablit la configuration précédente puis on passe à la cellule en conflit suivante.

4. Si aucune cellule de l'ensemble `doublons` ne peut être noircie, alors il ne peut exister de solution depuis la configuration actuelle, il faut donc renvoyer `None`.

Implémenter ce nouvel algorithme de résolution, et comparer son efficacité avec l'algorithme de base.

Astuces de résolution

Le solveur de base décrit plus haut peut être assez lent. On peut accélérer les calculs en utilisant une des observations suivantes (liste d'astuces adaptée de Wikipedia):

1. Lorsqu'on détermine qu'une cellule doit rester visible, toutes les cellules en conflit avec cette cellule peuvent immédiatement être noircies.

2. Lorsqu'une cellule est noircie, toutes les cellules adjacentes doivent rester visibles. D'après la règle précédente, on peut donc noircir toutes les cellules avec lesquelles ces cellules sont en conflit, et ainsi de suite. Utilisées ensemble, ces deux astuces permettent de "propager" des contraintes dans la grille dès qu'on noircit une cellule.

3. Lorsqu'un chiffre est entouré de deux chiffres identiques, il ne peut être noirci. S'il l'était, il y aurait deux chiffres identiques dans la même ligne ou colonne, ou deux cellules noires adjacentes (ce qui est interdit).

4. Lorsqu'une ligne ou colonne contient deux chiffres identiques adjacents ainsi que le même chiffre seul, on peut en déduire que le chiffre seul peut être noirci (dans le cas contraire, les deux chiffres adjacents devraient être noircis, ce qui est interdit).

Le but de cette amélioration est d'intégrer ces astuces dans le calcul du solveur, pour pouvoir fixer sans aucun doute le statut (visible ou noircie) d'une cellule. On peut évidemment ajouter d'autres astuces si l'on en découvre…

Autres améliorations possibles

Voici quelques autres suggestions d'améliorations, à n'aborder que si la partie obligatoire est *entièrement terminée*. 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 sur une de ces pistes.

- Implémenter une fonctionnalité "Vérification" qui teste si les règles sont respectées, et met en surbrillance les cellules problématiques sinon (par exemple des cellules visibles en conflit, deux cellules noires adjacentes, ou une zone de cellules visibles non reliée aux autres).
- Implémenter une fonctionnalité "Indice" qui utilise le solveur pour indiquer au joueur une cellule à noircir si la partie peut encore être gagnée, ou bien annonce immédiatement la défaite s'il n'y a plus de solution possible.
- 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 le format de fichier afin de permettre la représentation de cellules noircies.
- 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 niveaux aléatoires résolubles (c'est-à-dire pour lesquels il existe une solution, si possible unique).
- (Difficile :) Cette amélioration reprend, en la poussant plus loin, l'idée de la partie "Astuces de résolution". Il s'agit d'implémenter un solveur de niveaux qui imite le raisonnement d'un être humain, et ne recourt pas à l'algorithme de backtracking. Le solveur ne devra pas "deviner" si une cellule doit être noircie et revenir en arrière en cas d'erreur, mais fonctionner uniquement par déductions à partir de règles.

  Des exemples de stratégies de résolution que l'on peut implémenter sont visibles sur ce lien. Il peut être intéressant de comparer les performances du solveur "déductif" et du solveur par *backtracking* implémenté plus haut. Un solveur de ce type peut également permettre d'évaluer la difficulté d'une grille, en donnant un "score de difficulté" à chacune des règles utilisées pour la résoudre.

Toute autre amélioration est envisageable selon vos idées et envies, à condition d'en discuter au préalable avec un de vos enseignants. En effet, il serait dommage que ce que vous considérez comme une amélioration ne présente en réalité que peu d'intérêt de programmation ou ne dénature complètement le reste du travail.

Annexe : rappels sur le type `set()`

Le type `set` est un type conteneur, non séquentiel, itérable et mutable. Une valeur de type `set` ressemble à un dictionnaire dans lequel il n'y aurait que des clés. Ils servent à représenter des ensembles au sens mathématique (au plus 1 occurrence de chaque objet : pas de doublons !).

On définit un ensemble ainsi :

```
>>> vide = set()  # ensemble vide
>>> len(vide)
0
>>> ensemble = {1, 2, 1, 1, 4}
>>> len(ensemble)
3
>>> ensemble
{1, 2, 4}
```

Note : on ne peut insérer dans un ensemble que des éléments de type immutable (comme pour les clés d'un dictionnaire).

La particularité de ce type est qu'il permet de rechercher une valeur de manière très efficace (plus efficace que dans une liste) à l'aide du mot-clé `in`. On peut ajouter un élément à l'aide de la méthode `add`, ou en retirer un à l'aide de la méthode `discard`.

```
>>> ensemble = {1, 2, 1, 1, 4}
>>> 1 in ensemble
True
>>> ensemble.add(3)
>>> 3 in ensemble
True
>>> ensemble.discard(3)
>>> 3 in ensemble
False
```

De nombreuses opérations prédéfinies sont possibles, en particulier les opérations ensemblistes de base. Pour plus de détails, consulter la documentation officielle.

Il existe une variante immutable du type `set` (appelée `frozenset`) pouvant servir de clé de dictionnaire (ou de valeur dans un autre ensemble).