Projet « Trajet de Poursuite »

L'idée est d'implémenter l'**algorithme de Dijkstra** (voire **A*** ensuite) afin d'établir un **chemin « le moins coûteux »** possible **pour relier deux cases d'un quadrillage**, sur lequel on peut se déplacer horizontalement, verticalement ou en diagonale.

On pourra considérer qu'il s'agit d'un trajet choisi par un **« personnage non jouable » ou « *bot* » à la poursuite d'un joueur**, sur un plateau de jeu bi-dimensionnel, et que **le coût est le temps de parcours**, à vitesse constante (sauf sur certaines cases spéciales qui ralentissent le déplacement). On illustrera graphiquement à la fois ce parcours, mais également les nœuds du graphe (cases) « évalués » pour le déterminer, afin notamment de comparer les deux algorithmes dans ce contexte. Des éléments « clé en main » sont fournis, afin de se concentrer sur l'implémentation de l'algorithme (ou des 2) de *path finding*. Le code source complète l'information fournie ici.

Paramètres et organisation générale

Le module `param.py` définit différentes « **constantes** », notées comme selon la convention courante **en majuscules**, qui sont reprises dans les deux autres fichiers python qui l'importent.

On peut ainsi personnaliser différents paramètres, notamment quant aux aspects cosmétiques du rendu graphique, mais aussi et surtout

- pour le ralentissement du déplacement à travers les cases codées `SLOW` (coefficient multiplicateur de la durée `COEFF_SLOW`, à 3 par défaut), et
- l'importance relative de l'heuristique « distance à vol d'oiseau » dans A* (`COEFF_HEURISTIC`, à 0,9 dans le fichier fourni).

Les deux autres fichiers Python fournis sont

- `model.py` **qu'il faudra compléter**, consacré aux parties de modélisation détaillées dans la suite, ainsi que
- `window.py` qui assure la **représentation graphique** et qui est utilisable tel quel.

Plateau de jeu

Il s'agit d'une **grille rectangulaire**, modélisée dans un **fichier texte**, où les cases de bordure ainsi que celles constituant les obstacles infranchissables sont représentées par tout caractère différent de l'espace `' '` et du point `'.'`. Ces derniers désignent **respectivement une case libre** (franchie à vitesse normale) et une **case franchie à une vitesse plus lente** (*ex. eau ou terrain meuble, buissons, etc. dans un jeu*).

Toutes les lignes ont le même nombre de caractères (on termine par un retour à la ligne final).

Par exemple :

```
********************
*            ....    *
*    ###     ..      *
*     ##      ...    *
*                    *
********************
```

Quelques exemples de telles « *maps* » sont fournis (fichiers `.txt`).

Modélisation à l'aide d'un graphe

Un tel **plateau** est **converti en un graphe** (éventuellement orienté, *cf.* *****), dans lequel les nœuds sont les cases et les arêtes représentent les déplacements possibles entre ces cases.

La fonction `graph_from_file` fournie dans le module `model.py` renvoie un graphe qui modélise ce plateau de jeu, à partir du fichier texte dont l'URL est passée en paramètre. Plus précisément, la représentation du graphe qui est renvoyée est un **dictionnaire**, dans lequel **les clés sont les coordonnées** des cases, servant d'étiquette pour les nœuds***** (tuples de deux entiers partant de zéro, numéros respectifs de ligne puis de colonne). Les **valeurs** associées sont à leur tour des **dictionnaires**, où la clé `'cat'` donne une constante (`FREE`, `SLOW`, `WALL`) indiquant la **catégorie**, la nature de la case associée au nœud (*seconde partie de l'étiquette du nœud en somme*), et où `'neighb'` donne accès à une **liste des nœuds voisins**.

***** *On insère également dans ce dictionnaire deux éléments indiquant les dimensions du plateau*.

La valeur renvoyée par `graph_from_file` pourrait donc être par exemple :

```
{'length':7, 'height':5,
 (0,0):{'cat':WALL}, (0,1):{'cat':WALL},     …
 (1,1):{'cat':FREE, 'neighb':[(0,1), (1,1), (1,0)]},
 (1,2):{'cat':SLOW, 'neighb':[(1,1), (1,3), (2,1), (2,2)]},
          …                                  …                 }
```

Les **arêtes** sont **pondérées** pour tenir compte de la nature du terrain. Ici, on a choisi de le faire dynamiquement : pour des cases voisines horizontalement ou verticalement (ayant un *côté* commun), on attribue la valeur de base 1 à l'arête. Quand elles sont en contact seulement par un *sommet* (passage de l'une à l'autre en diagonale) on donne une valeur de base de $\sqrt{2}$ environ, en cohérence avec les distances à parcourir. Enfin, si la case *dont on sort****** est de catégorie `SLOW`, la valeur de base de la pondération est multipliée par un coefficient noté `COEFF_SLOW` supérieur à 1, pour prendre en compte la lenteur relative du déplacement. Le calcul de ces pondérations est **pris en charge par la fonction** `extra_cost` de `model.py`. *Cette approche dynamique est rapide à mettre en œuvre, mais ne brille guère au niveau de l'optimisation des temps de calcul, évidemment. N'hésitez pas à modifier cela…*

(***** *On a opté ici pour ce choix arbitraire afin de simplifier, le graphe est donc de fait orienté même si c'est peu important ici et qu'on utilise d'ailleurs plutôt un vocabulaire de graphe non orienté*)

La dernière fonction de `model.py`, à savoir `get_path`, est **celle que vous devrez compléter**, celle dans laquelle l'algorithme de *path finding* est mis en œuvre (voir sa *docstring*). On commencera par implémenter Dijkstra, avant de permettre éventuellement l'utilisation d'A* en l'adaptant légèrement la définition de la fonction, afin de prendre alors en compte le paramètre `algo`.

Visualisation

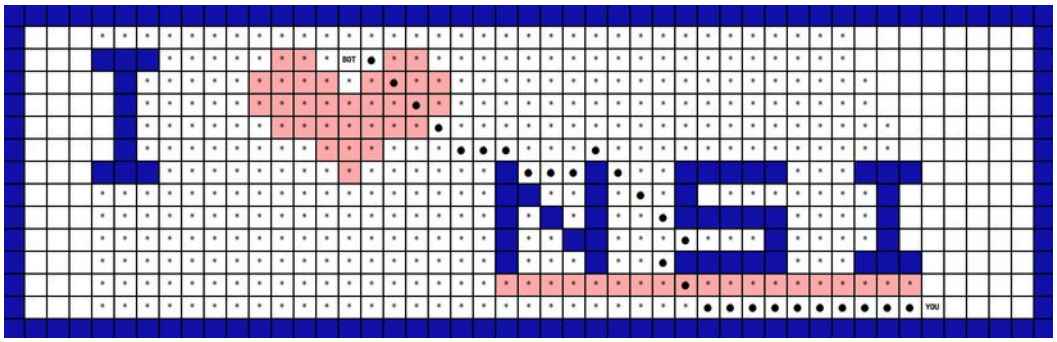
Le module `window.py` fournit une classe `Fen`, dont on crée un objet unique (utilisable tel quel). C'est cette instanciation qui **permet d'illustrer graphiquement le terrain** l'aide d'une fenêtre `Tkinter`, ainsi que **le trajet calculé** du point de départ (« *bot* ») au point d'arrivée (« joueur » poursuivi).

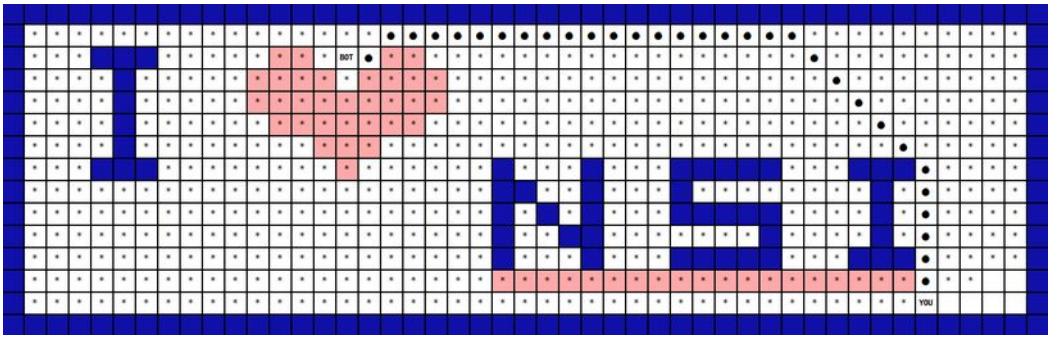
On pourra **déplacer au clavier** ces deux éléments pour étudier diverses dispositions (*on les dispose où l'on souhaite, en les déplaçant uniquement horizontalement ou verticalement mais pas en diagonale, même si le « bot » et le personnage poursuivi sont ensuite supposés pouvoir se déplacer en biais quand le trajet de l'un à l'autre est calculé*).

Il est également possible de **choisir l'algorithme** utilisé (soit Dijkstra soit A*), pour comparer les résultats obtenus et prendre la mesure des calculs effectués (dans `get_path`, vous pourrez tenir compte de ce paramétrage, indiqué par le paramètre algo qui vaut l'une des « constantes » prédéfinies `DIJKSTRA` ou `A_STAR`).

Le **trajet** renvoyé est **indiqué par des** `•` (*paramètre par défaut, modifiable*). Les cases qui ont fait l'objet d'une évaluation « définitive » du coût d'un trajet issu du point de départ sont indiquées par un `*`, ce qui permet d'apprécier les différences entre le parcours du graphe impliqué par l'un ou l'autre des algorithmes.

Voici à titre d'illustration deux exemples pour lesquels les résultats sont clairement différents (A* avec `COEFF_HEURISTIC` à 0,9 pour le premier), où l'on notera par exemple l'exploration « moins profonde » avec A* ainsi que la priorité à un parcours qui tend à rapprocher du but « à vol d'oiseau » (*cellules `SLOW` en rose ici*).





Derniers conseils

Pour rester efficace et prendre plaisir à programmer la partie manquante, il est indispensable d'**avoir bien compris l'algorithme** de Dijkstra. Prenez le temps de le **mettre en œuvre « à la main »** sur les exemples proposés.

Cette phase permet d'**identifier les données nécessaires**. On est alors en mesure de choisir **sous quelle forme on choisit de les représenter** (à définir clairement *avant* de « coder »).

Le terrain de jeu le plus réduit (**`test_map.txt`**) permet de vérifier facilement si besoin l'évolution des différentes variables mises en jeu.

Le **principe de A*** est le suivant : lors de l'évaluation du coût du trajet vers un nœud, **on ajoute une valeur sensée donnée un ordre de grandeur du coût du trajet restant** pour aller à la destination ciblée, on parle d'heuristique. Ici, cette valeur sera proportionnelle (cf. `COEFF_HEURISTIC`) à la **distance « à vol d'oiseau »** entre le centre de la case associée au nœud et celui de la case d'arrivée. Cela a tendance à guider l'exploration du graphe en priorité dans la direction du nœud visé, comme on peut l'observer sur la représentation graphique...