ENSAE

Projet C++

Recherche de chemin optimal par l'algorithme A*

Robin Monthulé

10/02/2015

Sommaire

Description du programme

1. Cahier des charges

L'objectif de ce projet est de proposer une résolution du problème de plus cours chemin. Considérons un environnement en 2D subdivisé en cases carrées praticable ou non, ainsi qu'une entrée et une sortie. Nous cherchons alors le trajet le plus cours allant de l'entrée vers la sortie en ne passant que par des cases praticables.

Pour parvenir à ce résultat j'ai préféré proposer une interface utilisateur, rendant ainsi le programme plus manipulable et vivant qu'une simple ligne de commande. Les conditions fixées sont alors les suivantes:

- Le projet sera présenté sous forme d'un exécutable.
- L'environnement 2D devra être créé par l'utilisateur, l'importation d'environnement via paramètre est inutile lorsque nous avons à faire à une interface utilisateur.
- L'environnement sera de taille libre.
- Les déplacements possibles seront les suivants : haut, bas, gauche, droite. Les déplacements diagonaux ne sont pas autorisés.
- Le projet aura pour seule vocation d'être un widget exportable en sa globalité, en ce sens l'ensemble du programme dédié à la résolution sera solidaire de l'interface, pour des questions pratiques, et ne pourra donc pas être utilisé séparément.
- L'algorithme implémenté sera l'algorithme A*, rappelons ce dernier succinctement. Partant de l'entrée que nous stockons dans une liste courante. Pour tout élément de la liste courante, soit cet élément est la sortie et l'algorithme s'arrête, sinon cet élément sera considéré comme traité et ses éléments contigus seront ajoutés à la liste courante. À noter que l'on examine les éléments de la liste courante par coût estimé (coût de l'entrée à ce point ajouté à la distance du point à la sortie) croissant.

Pour réaliser ce projet, j'ai utilisé la bibliothèque Qt (voir http:// qt-project.org/) ainsi que son IDE afin d'implémenter l'interface utilisateur. De ce fait le fichier projet (qui est ici un .pro) ne peut être ouvert qu'avec Qt, et la compilation n'est possible que sous cet IDE.

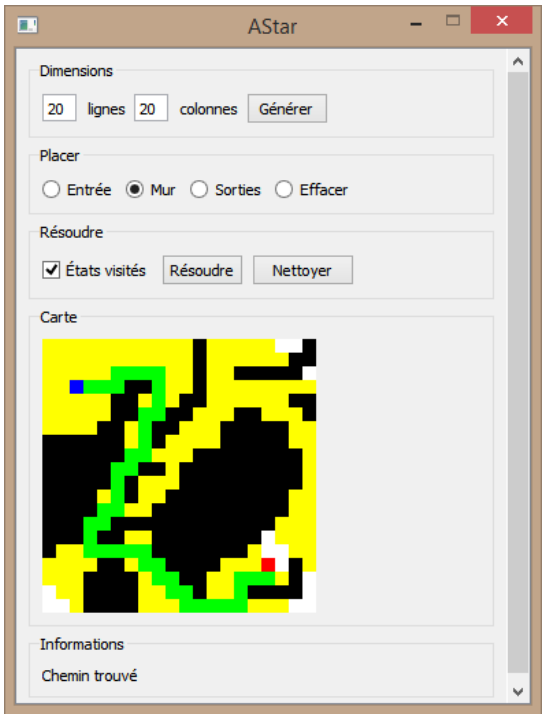


Figure 1 - Interface utilisateur

2. Interface utilisateur

L'interface se présente comme suit:

L'ensemble, qui est le widget, principal possède un layout vertical comprenant 5 widgets, chacun de ces derniers possède son propre layout (en général horizontal) Les 5 widgets sont les suivants:

a. Dimensions

Permet de choisir la taille de la carte (environnement) dans l'ensemble $[\![1;\ 99]\!]\times[\![1;\ 99]\!]$. Lors du clique sur le bouton "Générer" une nouvelle carte vierge est créée, supprimant ainsi l'ancienne.

b. Placer

Cet ensemble de boutons radio permet de sélectionner le type d'élément qui sera placé sur la carte lors d'un clique sur cette dernière.

c. Résoudre

Résout la carte créée par l'utilisateur en affichant ou non les cases visitées par l'algorithme. Permet aussi de nettoyer la carte (i.e. les sorties de l'algorithme).

d. Carte

Représente l'environnement créé par l'utilisateur ainsi que les sorties de l'algorithme.

e. Informations

Informations relatives au déroulement du programme (ceci a surtout été utilisé pour débugger le code).

Pour être plus précis, le layout vertical est contenu par un scroller, lui-même possédé par le widget principal. Ce scroller est présent sur la droite du widget et permet de faire défiler le widget verticalement (sachant que la hauteur de ce dernier est limitée à 700px, ceci afin de permettre à des écrans de toutes tailles de supporter le programme).

3. Utilisation

L'interface utilisateur est relativement intuitive même si épurée au maximum de texte. Cette dernière s'utilise dans l'ordre des widgets qui apparaissent:

1- L'utilisateur **choisit les dimensions** de la carte, cette dernière doit avoir une longueur et une hauteur maximale de 99 (sachant qu'une case sera représenté par un carré de 10 pixels de côté). Les dimensions rentrée, il ne reste plus qu'à cliquer sur le bouton "Générer" pour initialiser et faire apparaitre une carte vierge.
On notera qu'un deuxième clique sur le bouton "Générer" provoquera si nécessaire un redimensionnement de la fenêtre (le code devant redimensionner la fenêtre dès le premier clique, ceci provient d'un élément indésirable du scroller sur lequel nous reviendrons en fin de rapport).

2. L'utilisateur **remplit sa carte**. Pour ce faire, ce dernier doit tout d'abord choisir le type d'élément qu'il veut placer sur la carte via les boutons radio puis cliquer sur la carte pour y placer l'élément choisi. Les éléments sont :

- L'entrée : C'est l'origine, elle sera dessinée en **bleu** et tout nouveau placement d'entrée supprimera la précédente (une seule entrée par carte).
- Le Mur : Ce sont les cases impraticables de la carte, ils seront dessinés en **noir** et sont évidemment non-unique. Il ne faut pas oublier ici que les déplacements diagonaux sont non-autorisée lorsque que l'on place ses murs. Notons aussi que l'ajout d'un mur sur une entrée ou une sortie supprimera cette dernière.
- La sortie : C'est l'objectif, elle sera dessinée en **rouge** et sera unique, comme l'entrée.
- La case blanche ("Effacer") : C'est la case par défaut de la carte, dessinée en blanc, elle représente l'ensemble des cases praticables.

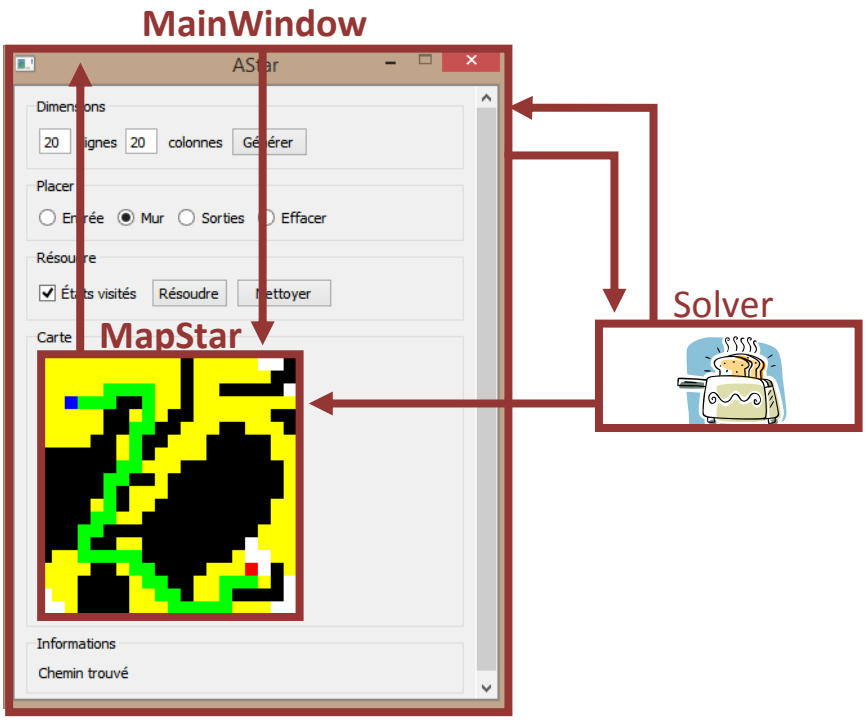
De plus il est possible de modifier la carte après la résolution du problème : rien n'empêche de supprimer un mur ou de déplacer n'importe quel élément sur la carte sans avoir à en regénérer une nouvelle.

3. L'utilisateur **résout le problème** en cliquant sur le bouton "Résoudre". En faisant ainsi l'algorithme de recherche de chemin optimal va se mettre en marche, et lorsqu'il aura terminé sa sortie (le chemin optimal) sera dessiné en vert sur la carte. De plus, en cochant la case "États visités" l'algorithme dessinera en plus en jaune l'ensemble des états visités lors de son exécution.
Enfin si l'utilisateur clique sur le bouton "Nettoyer" l'ensemble des sorties de l'algorithme (les cases vertes et jaunes) seront redessinées en blanc sur la carte, ceci afin d'obtenir une meilleur visibilité si l'on souhaite remodifier la carte.

Architecture

1. Architecture logicielle

L'ensemble du programme est décomposé en trois classes: MainWindow, MapStar et Solver.



Sur ce schéma, les flèches représentent les inclusions par passage en attribut. Par exemple dans notre cas la classe MapStar ne peut accéder au solver que par la classe MainWindow. J'ai choisi dans la structure des classes de rendre privés tous les attributs, cela fait partie des bonnes pratiques, ainsi les classes possèderont toutes des accesseurs pour certains attributs. Ces méthodes sont sans intérêt et ne seront donc pas expliquée ici.

a. MainWindow

Cette classe est le widget principal (la fenêtre ici). Ses seuls attributs sont l'ensemble des éléments (widgets, layouts et boutons) qui le compose ainsi qu'une instance de la classe MapStar et une instance de la classe Solver. Cette classe est l'élement principal de l'interface graphique et la plus grande partie de son code consiste en la création et le placement des différents éléments. Cependants quelques méthode présente un caractère non-trivial:

- updateSize() : Cette fonction est chargée de modifier la taille de la fenêtre lorsqu'une nouvelle carte est générée.
- solve() : Bien que cette fonction ne consiste qu'en l'appel à la fonction solve() du solver, sa particularité vient de sa nature un "slot". Cet élément, sous Qt, consiste en une méthode réaction activée après une action (appuie sur le bouton "Résoudre"), et un tel élément ne peut exister que dans une classe héritant de QWidget (la classe Solver ne peut donc pas posséder de slot).

- *resizeEvent* (QResizeEvent * event): Cette fonction est appelée lors d'un redimensionnement de la fenêtre afin de redimensionner le scroller (et donc les éléments qu'il contient).

b. MapStar

Cette classe gère la carte, de son affichage jusqu'à ses intéractions avec l'utilisateur. Elle comprend une image, un objet permettant de modifier l'image (un dessinateur), la couleur actuelle, le dernier point dessiné, l'échelle de la carte (ici 10) ainsi que le widget principal (instance de MainWindow). Ses principales méthodes sont:

- draw(const QPoint & position) : Cette fonction dessine sur la carte la case donnée avec la couleur courante. Cette fonction appelle la fonction de dessin pixel() et gère en plus l'enregistrement de ces modifications pour le solver.
- print() : Cette fonction actualise l'image de la carte.
- generate() : Ce slot, connecté au bonton "Générer" créé une carte vierge (donc une image image blanche et une matrice vide pour le solver)
- changeColor() : Ce slot, connecté aux boutons radio, actualise la couleur lors d'un clique sur les boutons radio.
- clear() : Ce slot, connecté au bouton "Nettoyer", efface de la carte les cases vertes et jaunes.
- *mouse[...]Event*(QMouseEvent *event) : Ces fonctions sont activée lors d'un clique, mouvement ou relâchement de la souris sur la carte et implémentent les actions qui conviennent.

c. Solver

Cette classe gère la partie algorithmique du programme. C'est elle qui possède les entrée et les sorties de l'algorithme et qui gère son exécution. Ses attributs sont un peu plus fournis:

- MainWindow *window, MapStar *map : Les instances de MainWindow et MapStar qui lui sont associés.
- int width, int length : Les largeurs et longueur de la matrice (carte).
- vector< vector<int> > mat : Cette matrice représente la carte avec les conventions (0=vide), (1=entrée), (-1=sortie), (2=mur).
- vector< vector<int> > process : Cette matrice représente l'état des cases au cours de l'algorithme avec les conventions (0=non visitée), (1=visitée), (2=traitée).
- vector< vector<int> > cost : Cette matrice contient le coût réel de l'entrée jusqu'à la case considérée.
- vector< vector<int> > ecost : Cette matrice contient le coût estimé allant de l'entrée à la sortie en passant par la case considérée. À noter que son utilité n'est pas flagrante sachant qu'elle est combinaison de ecost et dist(QPoint p,QPoint q) .
- vector< vector<QPoint> > back : Cette matrice associe à chaque case son parent dans le chemin allant de l'entrée à la sortie en passant par cette case.
- list<QPoint> todo : Cette liste contient l'ensemble des états visités en attente de traitement (trié par coût estimé croissant).
- QPoint in, QPoint out : Ce sont simplement l'entrée et la sortie.

Hormis ses accesseur la classe possède un petit nombre de fonctions digne d'intérêt:

- void solve() : Cette fonction implémente l'algorithme A* puis en affiche le résultat sur la carte.
- void insert(QPoint pos) : Cette fonction insère un élément dans la liste todo en laissant cette dernière triée.
- dist(QPoint p,QPoint q), adm(QPoint q) : Ces fonctions géométriques donnent respectivement la distance entre deux points et le fait qu'un point soit bien dans les limites de la carte.

2. Pistes d'améliorations

a. Redimensionnement

On remarque que lorsque l'on souhaite générer une carte dont les dimensions sont supérieures aux dimensions actuelles de la fenêtre cette dernière ne se redimensionne pas correctement. C'est en effet après un second clique sur le bouton générer que la fenêtre se redimensionne normalement.
Bien que ne sachant pas exactement d'où vienne se problème, il semble qu'il soit lié à un problème d'intrication et d'actualisation. Pour une certaine raison, lorsque la fonction de redimensionnement est appelée, l'image n'a pas encore été chargée et le widget n'a donc pas atteint sa taille finale. Un moyen de contrer ce problème dans un souci du détail aurait été de calculer directement la taille future de la fenêtre (sans avoir à la récupérer via les attributs du widget).
Cependant on notera aussi que la gestion de la taille des différents widgets est parfois rudimentaire et pourrais être améliorer. Certains attributs (entre taille de la fenêtre, taille minimale de la fenêtre, taille du scroller, taille minimale du scroller) ont parfois été fixé de façons à ce que l'interface se comporte correctement, sans pour autant s'attarder sur la pertinence de ces attributions (par exemple, des redondances sont à prévoir).

b. Structuration du code

L'inconvénient de ce programme reste néanmoins la non-portabilité de son algorithme (A*) et la non structuration des données. Comme expliqué précédemment, cela provient d'une priorité donnée à l'interface utilisateur et à l'aspect pratique et visuel du programme. Cependant avec plus de temps il est possible de faire un programme plus universel.

La structuration passe principalement (si ce n'est uniquement) par l'instanciation de la carte, qui dans notre cas est simplement une matrice d'entiers. L'idéal étant ici de créer une classe virtuelle Graphe afin de représenter les connections entre les différentes cases accessibles. Pour qu'une telle classe convienne il lui faudra impérativement trois méthodes (virtuelles):

- Une première qui a un nœud associe la liste des nœuds contigus.
- Une deuxième qui donne la distance entre deux nœuds contigus (poids de l'arrête)
- Une troisième qui donne la distance estimée entre deux nœuds (dans notre cas nous avons pris la distance de Manhattan)

Cette classe devra aussi posséder comme attribut une liste d'éléments de type libre (les nœuds). Ainsi nous n'obtenons pas la structure conventionnelle de graphe (i.e. liste de nœuds et liste

d'arrêtes) mais une interface permettant d'adapter n'importe quel type de données, que ce soit un véritable graphe ou une matrice d'entiers. Dans ce cadre il faudra alors remplacer la matrice process par les habituelles liste ouverte (process=1) et liste et liste fermée (process=2), qui sont des liste de nœuds.