Partie B – Recherche du plus court chemin

Objectif :

Dans cet exercice, l'objectif est de trouver le plus court chemin sur une carte en 2 dimensions. Vous implémenterez l'algorithme de Dijkstra et A*, et comparerez leurs performances sur différentes cartes. Un squelette de code vous est fourni. Il comprend :

- Un fichier WeightGraph.java définissant une structure de graphe pondéré. Vous n'avez normalement pas à le modifier, mais vous pouvez choisir une autre représentation si vous préférez.
- Un fichier App.java définissant :
    - Une méthode « main » dans laquelle vous devez lire la carte encodée dans un fichier texte, la représenter par graphe (à l'aide de la structure donnée), et lancer la recherche du plus court chemin avec l'un des deux algorithmes complétés
    - Une méthode Dijkstra et une méthode AStar à compléter. Celles-ci doivent, à partir du graphe, trouver le chemin entre le sommet de départ et d'arrivé.
    - Une méthode drawBoard et une classe Board permettant de gérer l'affichage de la carte, du déroulement des algorithmes et du chemin trouvé. Vous n'avez, là-encore, pas besoin de toucher ces codes.

Après avoir complété le code, vous évaluerez et comparerez les performances de Dijkstra et d'A* (en proposant une ou plusieurs heuristiques) sur différentes cartes (pertinentes). Vous devez aussi proposer une réflexion sur des cas réels d'utilisation des deux algorithmes à partir de vos constatations.

Format des cartes :

Le graphe est fourni sous forme de fichier texte, et un exemple est donné avec le squelette. Le format à respecter est le suivant (les parties à remplacer sont indiquées par <>) :

==Metadata==

=Size=

nlines=<int : Le nombre de lignes>

ncol=<int: Le nombre de colonnes>

=Types=

G=<int: Le temps nécessaire pour parcourir verticalement ou horizontalement une case de ce type>

<string : La couleur de la case du type précédent (sont utilisables sans modification du code : « green », « gray », « blue » et « yellow »)>

W=<int: Le temps nécessaire pour parcourir une case de ce type>

<string : La couleur de la case du type précédent (sont utilisables sans modification du code : « green », « gray », « blue » et « yellow »)>

…

==Graph==

<string : Une succession de ncols lettres, selon les types définis précédemment.>

<string : Une succession de ncols lettres, selon les types définis précédemment.>

… (nlines fois)

==Path==

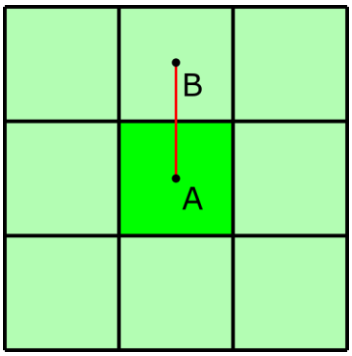
Start=<int,int: les coordonnées (ligne, colonne) du point de départ>

Finish=<int,int: les coordonnées (ligne, colonne) du point d'arrivée>

Vous pouvez donc définir vos propres cartes en modifiant celle proposée.

Définition du graphe :

Pour définir le graphe, vous devez considérer la connexité étendue aux 8 voisins (voir figure ci-dessous). Le poids d'une arrête allant d'une case A à B (connexes) doit être une combinaison du temps pour parcourir une case définie dans le fichier texte. Dans l'exemple ci-dessous, le poids de l'arrête rouge sera donc $\frac{t_A + t_B}{2}$ (où t_X est le temps nécessaire pour parcourir horizontalement ou verticalement la case X).
Attention à la définition des poids diagonaux.



La case verte foncée doit être connexe à toutes les cases vertes claires

Indications sur les algorithmes :

Dijkstra : l'algorithme de Dijkstra cherche à assigner la plus courte distance depuis le point de départ à tous les sommets d'un graphe, en utilisant une approche gloutonne : on part ainsi du point de départ (qui a une distance nulle depuis le point de départ), et ses voisins se voient attribuer la distance entre le point de départ et eux-mêmes comme distance temporaire. On considère alors le nœud courant (ici, le point de départ) comme visité et on ne le visite plus.

On repète cette procédure en choisissant comme nœud courant celui qui a la plus petite distance temporaire. On regarde pour ses voisins si la distance en passant par le nœud courant (donc distance temporaire plus distance du nœud courant au voisin) est plus petite que la distance temporaire. Si tel est

le cas, on mets à jour la distance temporaire, et on enregistre le nœud courant comme nœud parent du voisin dans le chemin.

On rèpte cette procédure jusqu'à avoir visité le nœud d'arrivé, où on peut alors dérouler le chemin.

Pour plus d'informations, vous pouvez consulter la page Wikipedia.

A* : l'algorithme A* (à pronnoncer A star ou A étoile) peut être vu comme une extension de l'algorithme de Dijkstra. La différence se fait dans le choix du nœud à explorer. Au lieu de choisir celui ayant la plus petite distance temporaire, on sélectionne en fonction de la somme entre la distance temporaire et une heuristique (ici, une distance estimée au nœud d'arriver) à définir.

Pour plus d'informations, vous pouvez consulter la page Wikipedia.