Plus court chemin dans une grille avec A^*

On cherche dans ce sujet à générer une grille aléatoire formée de cases vides et d'obstacles, à l'afficher et à y chercher des plus courts chemins en utilisant l'algorithme A^*. Dans l'ensemble du sujet, le seul langage de programmation utilisé sera **Ocaml**.

1 Tracé de grille

On utilise le type `int array array` pour représenter les grilles, 0 codant une case vide et 1 un obstacle. Pour afficher une telle grille à l'écran, nous allons utiliser le module Graphics, dont on pourra consulter la documentation à l'aide de `Zeal`. Pour charger ce module, on pourra utiliser la commande

```
ocaml -I /home/candidat/.opam/default/lib/graphics graphics.cma
```

pour exécuter, et écrire la ligne `#load "graphics.cma";;` (ou `open Graphics;;`) dans le fichier .ml. On ajoutera également la ligne `#load "Unix.cma";;` pour disposer du module `Unix`.

1. Écrire une fonction `construire_grille` prenant en argument deux entiers n, m et un flottant p, et renvoyant une matrice de dimension $n \times m$ dont la case i, j vaut 1 avec probabilité
$$p\left(1 - \frac{4((i-(n-1)/2)^2 + (j-(m-1)/2)^2)}{(n-1)^2+(m-1)^2}\right)$$
et 0 sinon (la probabilité d'obstacle est maximale au centre, où elle vaut p).
2. Écrire une fonction `tracer_grille`, prenant en argument une telle matrice et un entier k, et affichant dans la fenêtre graphique cette matrice. Les cases libres seront représentées en blanc et les obstacles en noir, chaque case étant dessinée comme un carré $k \times k$.
3. Utiliser les deux fonctions précédentes pour afficher à l'écran une grille en utilisant les paramètres $n = 50, m = 50, p = 0.5$. On utilisera la plus grande taille k de case permettant que la grille tienne dans la fenêtre graphique. L'affichage prendra fin au bout de 10 secondes (On pourra utiliser la fonction `sleepf` du module `Unix`).

2 Déplacements horizontaux et verticaux

On considère dans cette partie que deux cases sont reliées par une arête de poids 1 si elles sont libres et partagent un côté en commun. Les déplacements dans la grille se font donc verticalement et horizontalement, comme une tour aux échecs.

1. Écrire une fonction `existe_chemin` prenant en argument une grille et deux cases, et testant s'il existe un chemin entre ces deux cases.
2. Quelle structure de données est nécessaire pour une implémentation efficace de l'algorithme A^* ?
3. Implémenter cette structure de données pour l'usage qui nous intéresse. On pourra ajouter les opérations nécessaires au fil du sujet. On mentionnera la complexité de chaque opération.
4. Implémenter l'algorithme $A*$, prenant en argument une grille, une case de départ, une case de destination, et une heuristique, et renvoyant le chemin obtenu (sous forme de liste de cases) et sa longueur. **Attention, l'heuristique pourra être non admissible**.

5. Tester sur une grille générée par `construire_grille 50 50 0.5`, de la case $(0, 0)$ à la case $(49, 49)$ en utilisant différentes heuristiques.
   a) Quel algorithme obtient-on en utilisant l'heuristique nulle ?
   b) Quelle heuristique vous semble la plus pertinente si on veut toujours trouver une solution optimale ?
   c) Quel type d'heuristique vous semble pertinent si on veut minimiser le temps de calcul, quitte à obtenir un résultat non optimal ?
6. Modifier l'implémentation de A^* pour obtenir un affichage graphique pas à pas de l'algorithme. On utilisera les couleurs suivantes (pour le premier cas se présentant) :
   - noir pour les obstacles ;
   - bleu pour les cases sur le chemin de la dernière case défilée ;
   - jaune pour les cases ayant déjà été défilées ;
   - cyan pour les cases ayant déjà été enfilées ;
   - blanc pour les cases libres.

   On choisira un pas de temps suffisamment court pour que l'affichage soit fluide. On testera avec les différentes heuristiques considérées à la question précédente.
7. La complexité spatiale d'un algorithme est la quantité de mémoire utilisée lors d'un appel (sans compter l'espace occupé par les entrées). Modifier la fonction précédente (et au besoin la structure de données associée) pour que la complexité spatiale soit linéaire en le nombre de cases explorées, et non en le nombre total de cases (on ne tiendra pas compte du coût de l'affichage graphique).

3 Déplacements de direction quelconque

On considère à présent que deux cases libres sont reliée si le segment reliant leurs centres n'intersecte aucun obstacle. Le poids de cette arête est alors la distance euclidienne entre ces centres.

1. Écrire une fonction :

   ```
   construire_graphe :
        int array array -> int -> (int * int) list array array
   ```

   prenant en argument une grille et un entier b, et renvoyant la matrice de listes d'adjacences représentant le graphe décrit. On ne testera la présence d'une arête qu'entre des cases dont les abscisses et les ordonnées ne diffèrent que d'au plus b.
2. Implémenter une version de A^* pour cette variante. On prendra en entrée la grille **et** la matrice de listes d'adjacences associée. Dans l'affichage graphique, à chaque étape le chemin sera dessiné par une suite de segments bleus reliant les cases de centre en centre. On utilisera les fonctions de synchronisation de `Graphics` pour améliorer la qualité visuelle de l'affichage.
3. Tester en utilisant différentes heuristiques pertinentes.

4 Coloriage des composantes connexes

1. Proposer et implémenter une méthode pour choisir n couleurs, dont noir, maximalement distinctes.
2. En déduire une fonction affichant une grille avec une couleur différente pour chaque composante connexe.