Évolution du code

Dans un premier temps, nous avons choisi de créer nous même une classe graphe et de ne pas utiliser les structures existantes. Nous avons ainsi défini un graph comme un ensemble de points et d'arêtes, il a fallu créer ces deux objets.
Nous avons réutilisé le modèle fait au cours semestre 5 en TP pour M312, nous avons défini un point comme un tableau de double de taille 2, le premier élément représente la coordonnée x et le deuxième élément représente la coordonnée y d'un point dans R2 noté (x,y).
Nous avons aussi créé l'objet arête qui est un tableau de Point de taille 2, une arête est donc simplement une liaison entre 2 points.
Une fois ces objets créés nous avions tout ce dont nous avons besoins pour créer un graph.
Nous avons ainsi ajouter en premier le point source à l'ensemble des points du graphe, puis le corps du graphe (points allant du point (1,1) au point (n,n) pour une graphe de taille n), et pour finir on ajoute le puits.
Ensuite afin de créer les arêtes nous avons décidé de parcourir deux fois notre ensemble de point puis d'utiliser la formule fourni dans le sujet : on regarde pour des points (i,j) et (i',j') si abs(i-i') + abs(j-j') = 1 alors on créer une arête entre le point (i,j) et (i',j') pour des graphes carrés. Pour des graphes triangulaires nous avons rajouter la condition i = i'-1 et j = j'-1.
Ensuite on rajoute les arêtes entre la source et tous les points ayant x = 1, et les arêtes entre le puits et tous les points ayant x = n. Toutes ces arêtes sont créées dans les deux sens c'est-à-dire que pour une arêtes entre le point (i,j) et (i',j') on crée l'arête (i,j)-(i',j') et l'arête (i',j')-(i,j). Cela est nécessaire pour l'implémentation de l'algorithme de Dijsktra.
Une fois l'objet graphe créé nous avons essayé de l'afficher, nous avons essayé de l'afficher dans le terminal en surchargeant l'opérateur chevron cependant l'affichage n'étais pas agréable visuellement et illisible pour des grandes tailles de graphe. Nous avons donc cherché une méthode pour afficher ces graphes de manière lisible et agréable.
Nous avons ainsi trouvé la classe bitmap_image qui permet de créer une image BMP, nous nous sommes documentés pour comprendre l'utilisation des différentes fonctions et nous avons réussi à représenter un graphe. Dans un premier temps nous avons décider d'une distance fixe entre chaque point, cependant à partir d'une certaine taille de graphe notre graphe ne rentrais plus dans l'image. C'est pourquoi nous avons améliorer le code de cette fonction en définissant une distance entre chaque point proportionnel à la taille du graphe afin que nos points ne sortent plus de l'image.
Après avoir créé un graphe et être capable de l'afficher nous pouvions nous pencher sur l'implémentation de l'algorithme de Dijkstra.

Pour l'implémentation de cet algorithme nous nous sommes documentés et il s'est avéré qu'il était beaucoup plus simple de raisonner avec les indices des points et d'utiliser la structure PriorityQueue.
Pour ce faire nous avons tout d'abord implémenter cet algorithme en parcourant pour chaque occurrence les vecteurs de sommet et d'arêtes pour trouver les indices des points voisins, cet algorithme marche mais est très lent pour des grandes tailles de graphe. Dans cet algorithme nous avons utilisé une fonction distance qui calcule la distance euclidienne entre deux points pour trouver la plus courte distance. Nous avons dans un premier temps garder cet algorithme et essayer de l'améliorer comme demandé. Nous avons rajouté une condition dans notre

boucle qui permet à notre algorithme de s'arrêter lorsque le puits ne fait plus partie des sommets restants à visiter, cela a permis une réduction du temps d'exécution peu significative. Nous avons aussi ajouté un vecteur qui stock le prédécesseur de chaque point.
Cette amélioration a permis ensuite de créer la fonction PlusCourtChemin qui par backtracking retrouve le prédécesseur de chaque point en partant du puits et en remontant jusqu'au point source, cela permet donc de trouver le chemin le plus court entre la source et le puits.
En reprenant le modèle d'affichage du graphe nous avons été capable de créer une fonction qui retrace le graphe et repassant en rouge le chemin le plus court entre la source et le puits.

Nous avons ensuite implémenté un second algorithme de Dijkstra en utilisant cette fois la méthode de Fast Marching pour la fonction coût.

Une fois ceci fini nous avons essayé d'adapter notre code fait pour le graphe et le calcul du chemin le plus court à l'image du musée. Pour ce faire nous avons réutiliser la classe bitmap_image en extrayant les informations de chaque pixel, nous avons associé à chaque pixel de l'image un point. Cependant nous avions besoins de pouvoir différencier un point associé à un pixel bleu d'un point associé à un pixel rouge ou noir, c'est pourquoi nous avons rajouter des arguments booléen blue et red dans la classe Point qui valent 1 si le pixel associé à ce point est respectivement bleu ou rouge. Nous avons ainsi relié les points les points bleu voisins entre eux, les points rouges voisins entre eux et les points bleu et rouge voisins entre eux.
Une fois que cela était fait nous avons essayé d'utiliser notre algorithme de Dijkstra mais le calcul était beaucoup trop long et n'aboutissait pas. Il s'est avéré que le fait de parcourir à chaque occurrence les vecteurs de sommet et d'arête étais trop couteux surtout pour des tailles élevées comme pour notre image qui est une image de 800 par 546 pixels.
Nous nous sommes alors penchés sur l'améliorations de notre code pour le graphe dans le but de pouvoir l'adapter à notre image du musée.

Nous avons d'abord pensé à optimiser l'algorithme de Dijkstra en évitant de parcourir plusieurs fois les vecteurs sommets et arêtes à chaque occurrence, nous avons donc construit dans la fonction de l'algorithme de Dijkstra un vecteur qui pour chaque indice de point associe ses voisins en parcourant une seule fois les vecteurs sommet et arêtes au lieu de les parcourir à chaque occurrence. Cette amélioration à permet une réduction significative du temps d'exécution pour des graphes de grandes tailles cependant ce n'étais pas encore suffisant pour pouvoir l'adapter à notre image de musée.
Nous avons alors rajouté un argument à notre graphe qui servirai uniquement à l'algorithme de Dijkstra, cet argument est un vecteur qui pour chaque indice de point associe l'indice de ses voisins. Cette amélioration est celle qui nous a permis de pouvoir réutiliser notre algorithme pour notre musée car en créant ce vecteur nous n'avons pas besoins de parcourir plusieurs fois nos vecteurs déjà créé car ce vecteur est initialisé lors de la création de notre graphe.

Afin que l'algorithme nous donne le chemin le plus court entre chaque point bleu du musée et la sortie la plus proche nous avons relié un point source fictif à tous nos pixels rouges, cela nous permet de choisir la sortie la plus proche pour chaque point au lieu d'une sortie fixe pour tous les points.

Nous avons donc pu créer un vecteur du même type pour l'image du musée qui associe à chaque indice de point les indices de ses voisins. Nous avons utilisé le même algorithme de Dijkstra et nous avons pu adapter notre fonction qui trace le chemin le plus court à notre image du musée en reprenant le plan du musée et en y ajoutant le chemin entre un point bleu tiré au hasard et sa sortie la plus proche.

Informations importante :

Dans tout notre programme nous avons considéré que les arêtes reliant la source au corps du graphe et le puits au corps du graphe sont des arêtes de longueur 1.