Rapport de projet

Barjokart - Équipe 4

Florian Savouré | Axel Sergent | Maxime Lemoult | Antoine Delahaye

I. Outils et bibliothèques utilisés

Afin de travailler sur le projet dans de bonnes conditions, nous avons décidé d'utiliser les mêmes outils et de procéder de la même façon, permettant d'avoir une certaine unicité sur tout le long du projet.

Ainsi, le code du projet a été versionné avec Git et différentes branches ont été créées pour développer les différentes fonctionnalités. Concernant le code, nous avons décidé d'utiliser un IDE, CLion, pour profiter de ses différentes fonctionnalités lorsque l'on programme. De plus, lorsque l'on crée un projet avec CLion, ce dernier utilise CMake dans le but de générer automatiquement le Makefile en fonction des paramètres entrés dans le CMakeList, notamment les différentes bibliothèques utilisées.

Concernant les bibliothèques qui ont été utilisées, elles sont aux nombres de deux, il s'agit de CImg pour extraire les données des images et toml++ pour gérer les fichiers toml en C++.

II. Structure du projet

La structure du projet est relativement simple, elle se compose de 10 fichiers et de 2 dossiers. Ces derniers sont ***input*** et ***output***, l'un sert à stocker les images des circuits ainsi que les paramètres associés et l'autre stocke les fichiers images avec le tracé généré et les fichiers binaires.

Pour les fichiers du projet, nous avons deux bibliothèques, ***CImg.h*** pour traiter les images et ***toml.hpp*** pour récupérer les données des fichiers de configuration. Concernant le code source, nous avons ***node.h*** et ***node.cpp,*** la classe nœud qui correspond à un point de l'image, ***map.h*** et ***map.cpp*** pour le vecteur de nœud qui correspond à une image, ***astar.h*** et ***astar.cpp***, l'implémentation de l'algorithme A*, ***main.cpp*** qui sert à exécuter l'algorithme, générer le tracé et le fichier binaire. Enfin CMakeLists.txt qui va générer le Makefile de compilation en fonction des paramètres qu'on aura spécifiés dans ce fichier.

III. Fonctionnement

A. Nœud

Afin de pouvoir utiliser l'algorithme A* nous avons besoin de plusieurs informations à chaque pixel visité, c'est pour cela que chaque pixel est représenté par un nœud. Un nœud va donc contenir plusieurs informations, telle que sa position sur l'image avec les coordonnées X et Y, le nœud parent par lequel on a dû passer afin d'accéder à ce dernier, l'évaluation heuristique pour estimer le meilleur chemin, le coût, la somme, la liste de ses voisins pour précalculer ces derniers pour gagner du temps lors de l'algorithme ainsi que des booléens pour savoir si le nœud en question est un mur, si c'est un nœud de destination et s'il est ouvert ou fermé.

```
int x, y;
```

```
Node *parent;
double f, g, h; // f la somme, g le coût et h l'heuristique
bool isWall, isDestination, isOpen, isClosed;
std::vector<std::pair<Node *, double>> *neighbors;
```

B. Map

Pour représenter une image et calculer le chemin le plus court, il va être nécessaire de stocker tous les nœuds dans un tableau ordonné. On va donc itérer sur chaque pixel de l'image afin d'instancier un nœud correspondant aux caractéristiques de ce pixel.

```
// Exemple d'une boucle qui va parcourir l'image en largeur et hauteur
// Pour ajouter à chaque pixel un nœud dans le vecteur
for (int i = 0; i < img->width(); i++) {
      for (int j = 0; j < img->height(); j++) {
            Node *node = new Node(i, j);
            map.push_back(node);
      }
}
```

Chaque nœud doit connaître ses voisins, on va de ce fait parcourir une nouvelle fois l'image afin d'instancier ces derniers.

```
// Gauche
if (i > 0) {
      node->neighbors->push_back(std::make_pair(getNode(i - 1, j), 1));
      // Haut gauche
      if (j > 0) {
            node->neighbors->push_back(
                        std::make_pair(getNode(i - 1, j - 1), 1));
      }
}
```

Là aussi, étant donné qu'il ne va pas être possible de se déplacer sur les murs, on va devoir savoir si un nœud correspond à un mur, pour cela on va vérifier si la couleur est noir, si elle l'est, on va ainsi mettre à vrai le booléen indiquant que le nœud est un mur. De même, pour savoir si le nœud est un point de destination, si la couleur est rouge, on définit à vrai le booléen correspondant et on ajoute le nœud dans un vecteur dédié.

```
// Si c'est un mur
if (img->atXY(i, j, 0) == 0 && img->atXY(i, j, 1) == 0 &&
img->atXY(i, j, 2) == 0) {
      node->isWall = true;
}
```

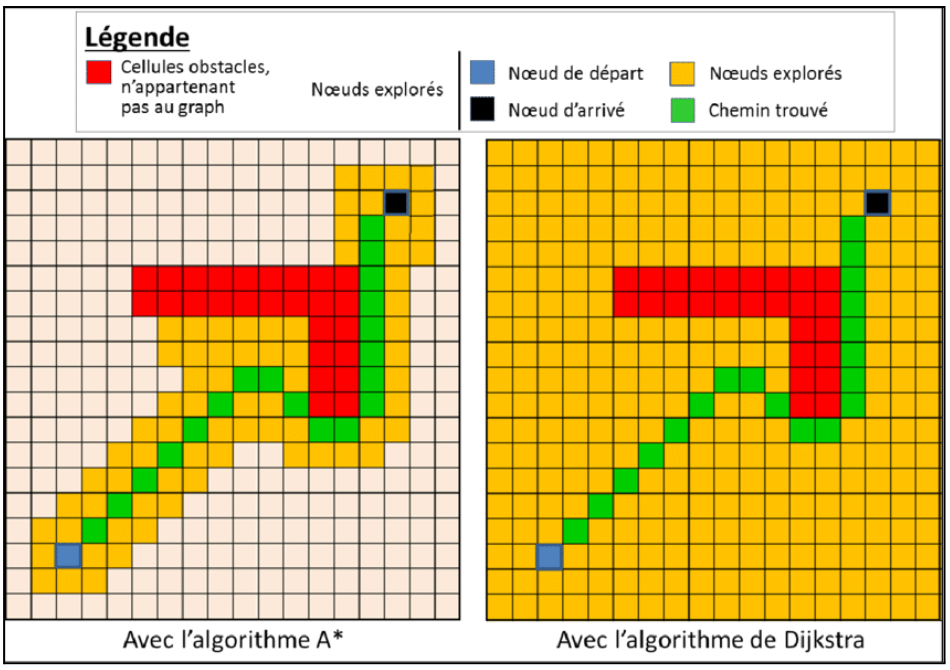
```
// Si c'est un pixel de destination
if (img->atXY(i, j, 0) == (int) color_dest[0] && img->atXY(i, j, 1) ==
(int) color_dest[1] && img->atXY(i, j, 2) == (int) color_dest[2]) {
      node->isDestination = true;
      this->destinationsNodes.push_back(node);
}
```

Enfin, on veut connaître au préalable le nœud de destination dans le vecteur, on va donc calculer ce dernier en prenant tous les nœuds de destination (là où les pixels sont rouges) présent dans le vecteur et faire la moyenne.

```
// On fait la moyenne des coordonnées des destinations
// Pour avoir à la fin un point central
double x = 0;
double y = 0;
for (auto &i: this->destinationsNodes) {
      x += i->x;
      y += i->y;
}
x /= (double) this->destinationsNodes.size();
y /= (double) this->destinationsNodes.size();
this->destination = getNode((int) x, (int) y);
```

C. Algorithme A*

A* est une modification de l'algorithme de Dijkstra, en effet A* est optimisée pour trouver le chemin vers une destination précise en passant par les points qui lui semble le plus proche de l'arrivée tout en étant plus rapide que Dijkstra, car ce dernier visite tous les points, même ceux qui ne sont pas prometteurs.



Le pseudo-code suivant décrit le fonctionnement de l'algorithme A* :

```
Fonction CheminLePlusCourt(vecteur: Vecteur, départ: Nœud,
arrivée: Nœud)
      listeFermée: Vecteur de nœud
      listeOuverte: Vecteur de nœud trié par heuristique
      ajouter départ à listeOuverte
      tant que listeOuverte n'est pas vide
            n = retirer le premier élément de listeOuverte
            si n.x == arrivée.x et n.y == arrivée.y
                  p = n.parent
                  chemin: Vecteur de nœud
                  ajouter p à la fin de chemin
                  tant que parent n'est pas vide
                        ajouter p à chemin
                        p = p.parent
                  retourner chemin
            pour chaque voisin v de n dans vecteur
                  si v n'existe pas dans listeFermée ou v n'existe pas
dans listeOuverte avec un coût inférieur)
                        v.cout = n.cout + 1
                        v.heuristique = v.cout + distance([v.x, v.y],
[objectif.x, objectif.y])
                        ajouter v à listeOuverte
            ajouter n à listeFermée
      retourner erreur
```

D. Algorithme de lissage

L'algorithme de lissage va permettre de rendre les tracés plus droits, cela va permettre de faciliter les changements de vitesse. Afin de lisser le tracé, on va construire un nouveau vecteur. Pour construire ce dernier, on va parcourir l'ancien vecteur en vérifiant pour chaque point si on peut tracer une ligne, si c'est le cas on continue jusqu'à que ça ne soit plus possible et on ajoute ces points dans le vecteur.

```
auto vec_nodes = bresenham(currentNode, nextNode);
while (isVectorValid(vec_nodes)) {
      if (next_i + 1 < vecPath->size()) {
            nextNode = (*vecPath)[next_i + 1];
            vec_nodes = bresenham(currentNode, nextNode);
            next_i++;
      } else {
            for (Node *n: vec_nodes) { newPath->push_back(n); }
            return newPath;
} }
```

E. Accélération

L'accélération avec Bresenham

Dans notre implémentation de la vitesse, nous essayons de trouver la vitesse la plus élevée possible entre deux points tout en étant inférieur à la vitesse maximum. Pour cela on va créer un nouveau vecteur de nœuds et on va itérer sur le vecteur de nœuds passé en paramètre. Ainsi, on va prendre le premier et le dernier nœud du vecteur et on va regarder s'il est possible d'y arriver directement avec l'accélération maximum. Si ce n'est pas le cas, on prend le nœud précédent et ainsi de suite jusqu'à y arriver et alors on va repartir de ce nœud et refaire les étapes précédentes.

```
for (int i = 1; i < vecPath->size(); i++) {
        for (int j = (int) vecPath->size() - 1; j >= i; j--) {
            Node *potentialNode = (*vecPath)[j];
                  int speed = std::abs(currentNode->x -
potentialNode->x) + std::abs(currentNode->y - potentialNode->y);
            if (speed <= acc_max) {
                        ...
                  }
            }
      }
}
```

L'accélération sans Bresenham

Pour la deuxième proposition d'accélération, nous utilisons les vecteurs vitesses d'escargot pour faire l'accélération. Grâce à la fonction cutting() qui nous permet d'obtenir les différentes trajectoires en retournant un vecteur de Pair. Dans la fonction accélération2(), nous calculons la longueur de la trajectoire et ainsi notre algo sait quand il faudra accélérer avec l'accélération max et décélérer pour pouvoir prendre en compte le changement de trajectoire. Il prend ensuite en compte la dernière vitesse pour faire la vitesse de la prochaine trajectoire pour éviter un tête-à-queue.

Répartition du travail

Nous avons fait quelques réunions pour répartir le travail et chacun a travaillé sur ce qu'il avait envie et à sa vitesse. Nous avons un channel discord dédié au projet afin de communiquer facilement sur l'avancement ainsi que les problèmes rencontrés.

Score barjokart

Prairie : **9855**, C'est tout droit : **9987**, π : **9953**, dilemne dilemne : **9980**, Ever given : **9970**, Very Fast Food (is unhealthy) : **9804**, Mon premier circuit : **9996**, Balade amnésique : **9364** ,Inter5tellar : **9890**, adaptallure : **9938**, Athens, France : **9525**