Mamourou
Hugo
Matthieu

SAÉ 2.02

Exploration algorithmique d'un problème

Table des matières

Présentation

Ce projet consiste en la réflexion, la conception et le développement d'un algorithme de recherche de chemin (« Pathfinding ») dans une carte (« Map ») possédant différentes propriétés.

L'idée est d'avoir plusieurs zones (« Biomes ») sur la map, qui possèdent chacun des propriétés qui ralentissent ou accélèrent le personnage qui y navigue.

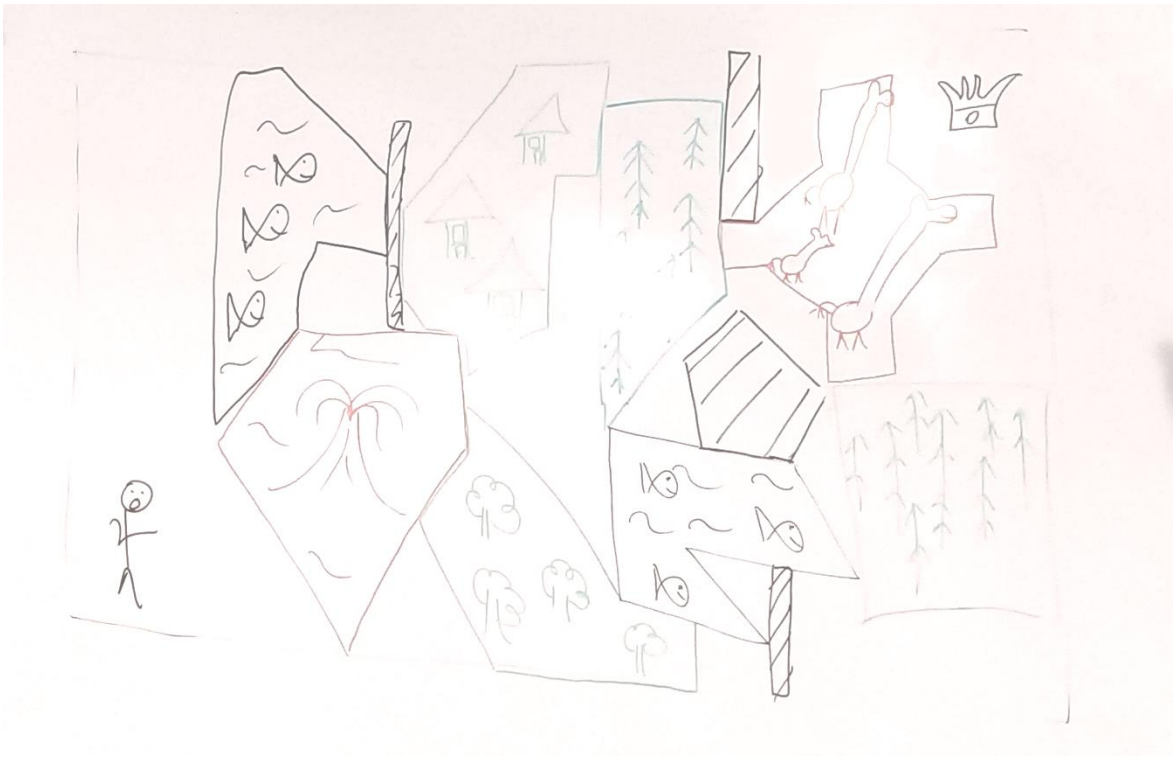


Photo de la map ayant servie d'explication du projet. Art par Maxime BERGER.

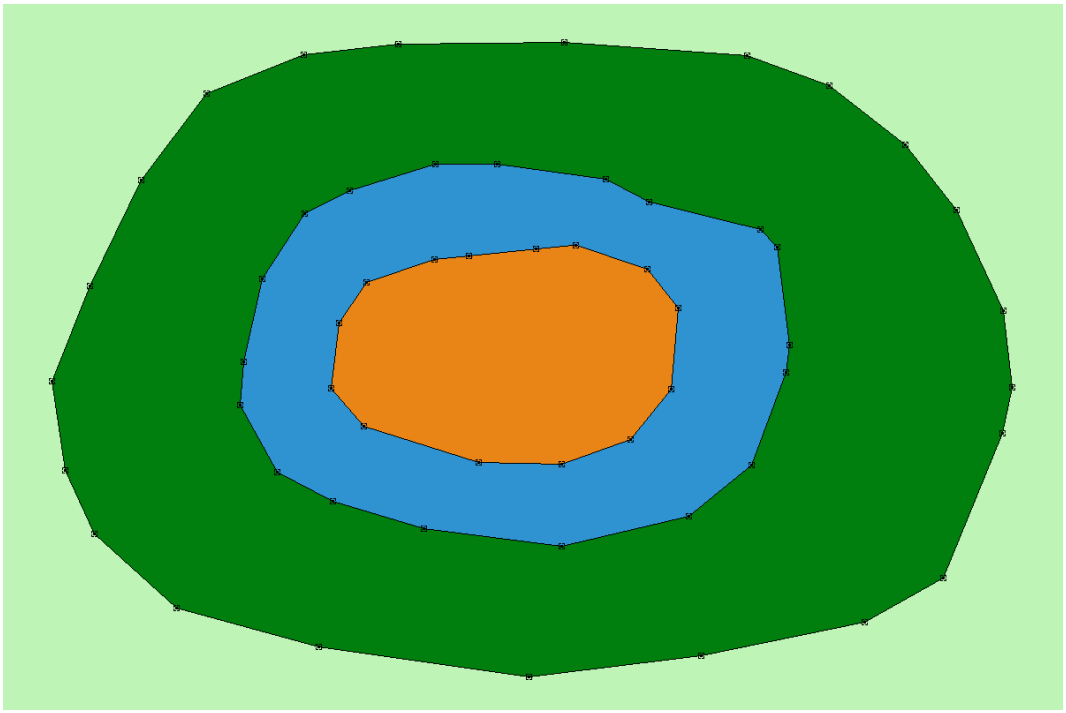
Les propriétés des biomes peuvent être diverses et fondées (par exemple, la zone de savane ralentit le personnage car ses girafes le ralentissent, alors que le village lui rend ses forces et le fait donc se déplacer plus rapidement).

La finalité du projet est donc de trouver le chemin le plus rapide entre deux points de la map, en respectant les contraintes des biomes.

Notre approche

Nous avons fait le choix d'utiliser un maillage pour calculer nos chemins.

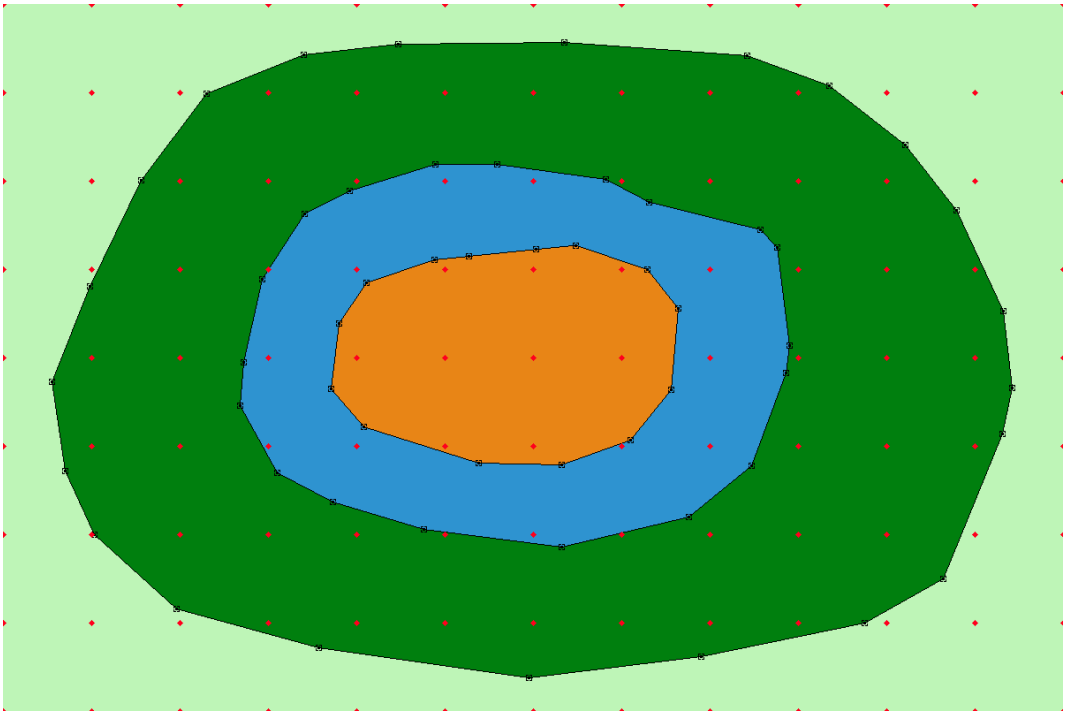
Pour illustrer, prenons la map suivante :



Map composée d'une plaine entourant une forêt entournant une rivière entourant un volcan.

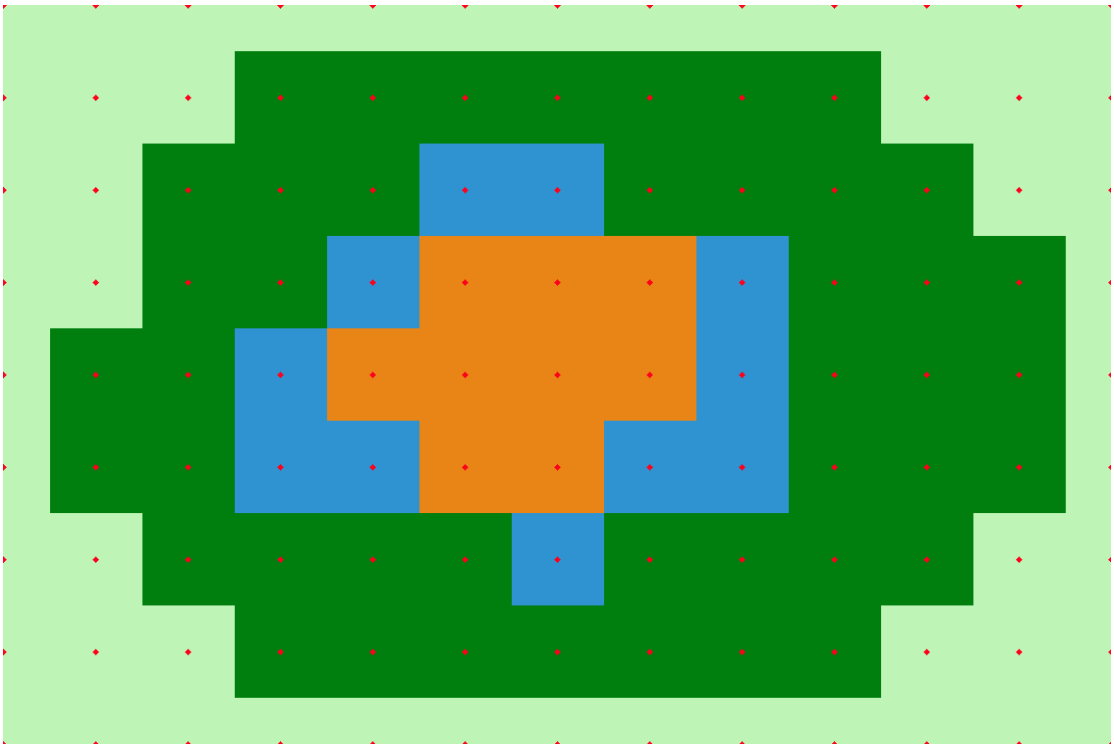
Cette map fait 120 × 80 unités. L'échelle utilisée est de 10 pixels par unité de coordonnées.

Si l'on créé un maillage d'une valeur d'un nœud pour 10 unités (= 0,1 mailles par unité), on obtient alors les points suivants :



Même map, avec des points rouges toutes les 10 unités

Ce maillage nous permet de grandement simplifier la map, et chacun de ses points représentent une position à laquelle le personnage peut aller. Nous pouvons donc redessiner notre map sous une forme de *tile mapping* :



Même map, mais chaque point est le centre d'une tuile (« Tile ») du biome dans lequel le point se trouvait

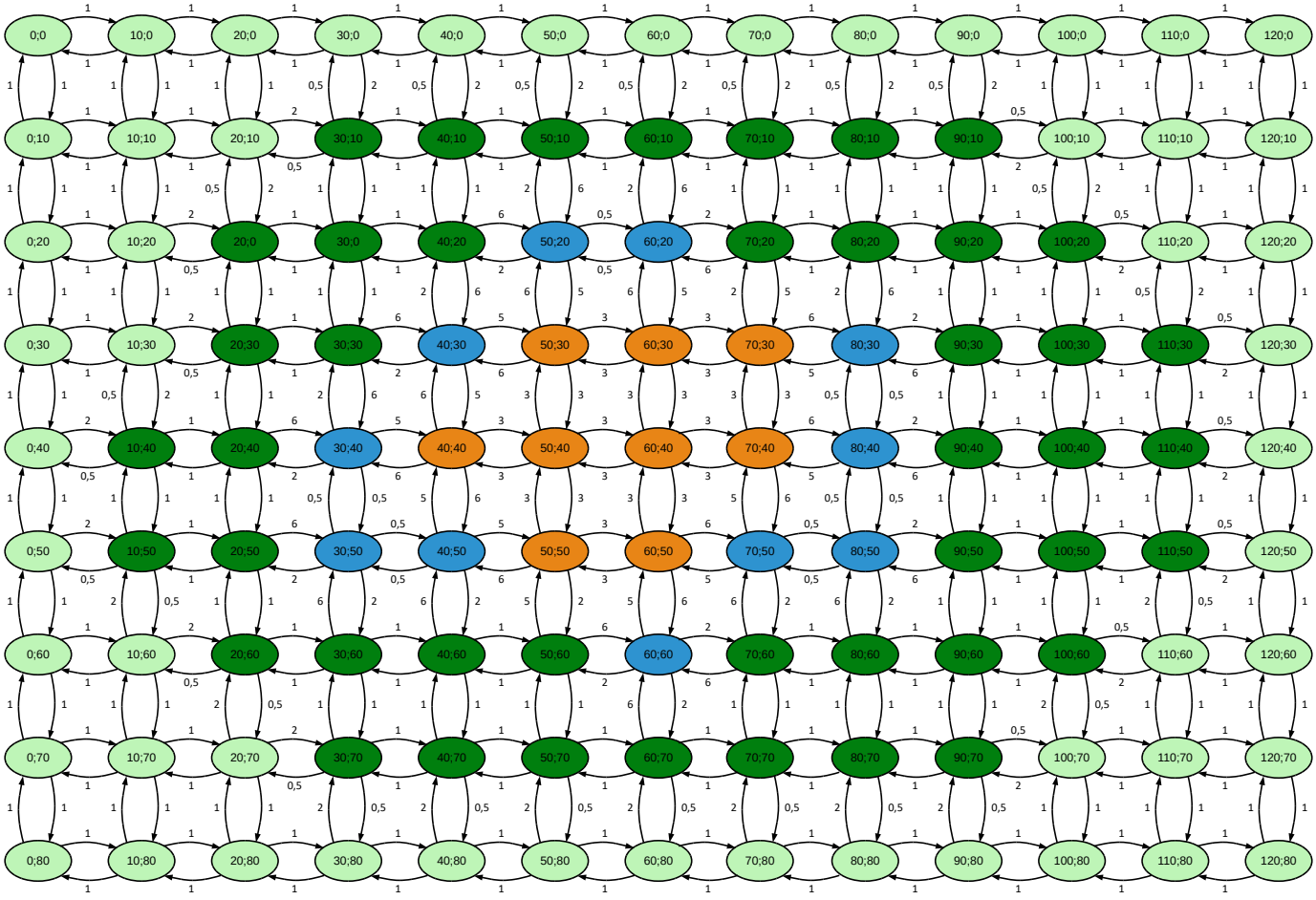
Par cette représentation, nous pouvons à présent créer un graphe orienté qui schématise notre map. Prenons par exemple les poids suivants :

- Entrée dans un biome plaine : 0,5
- Navigation dans un biome plaine : 1
- Entrée dans un biome forêt : 2
- Navigation dans un biome forêt : 1
- Entrée dans un biome volcan : 5
- Navigation dans un biome volcan : 3
- Entrée dans un biome rivière : 6
- Navigation dans un biome rivière : 0,5

Ces poids représentent le temps (unité arbitraire) qu'il faut pour se déplacer d'une unité de coordonnées selon la situation, avec des pénalités lors des changements de biomes.

Un biome ayant un poids négatif agira comme un obstacle, c'est-à-dire qu'il sera impossible d'y entrer et qu'il faudra donc le contourner.

Nous pouvons alors construire le graphe suivant :



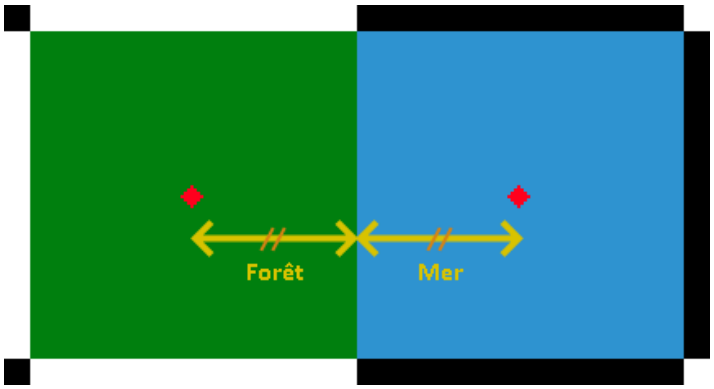
Graphe orienté représentant la même map, mais sous forme de nœuds et d'arcs

Ce graphe représente notre situation, où les arcs ont pour poids les temps liés à chaque situation.

Il n'est cependant pas terminé : il faut encore ajuster ces poids par rapport à la distance que leurs arcs fait parcourir :

- Dans le cas de la navigation intra-biome, il suffit de multiplier le poids de l'arc par la distance entre les deux nœuds qu'il relie. Ainsi, son poids sera égal au réel temps qu'il faut pour naviguer d'un nœud à l'autre.

- Dans le cas de la navigation inter-biome, cela est plus compliqué.
  Reprenons notre représentation en tuile :



  Quand un déplacement inter-biome est effectué, la moitié de la distance est faire dans le premier biome, et l'autre moitié dans le second. À ça, il faut ajouter la pénalité de changement de biome correspondant au biome de destination.

  Donc, soit :
  - d la distance entre les deux nœuds
  - $P_{B1_{nav}}$ le poids de navigation du biome source
  - $P_{B2_{nav}}$ le poids de navigation du biome de destination
  - $P_{B2_{ent}}$ le poids d'entrée du biome de destination

  Alors, le poids de l'arc allant du biome source au biome de destination sera de :

$$\frac{P_{B1_{nav}} * d}{2} + \frac{P_{B2_{nav}} * d}{2} + P_{B2_{ent}}$$

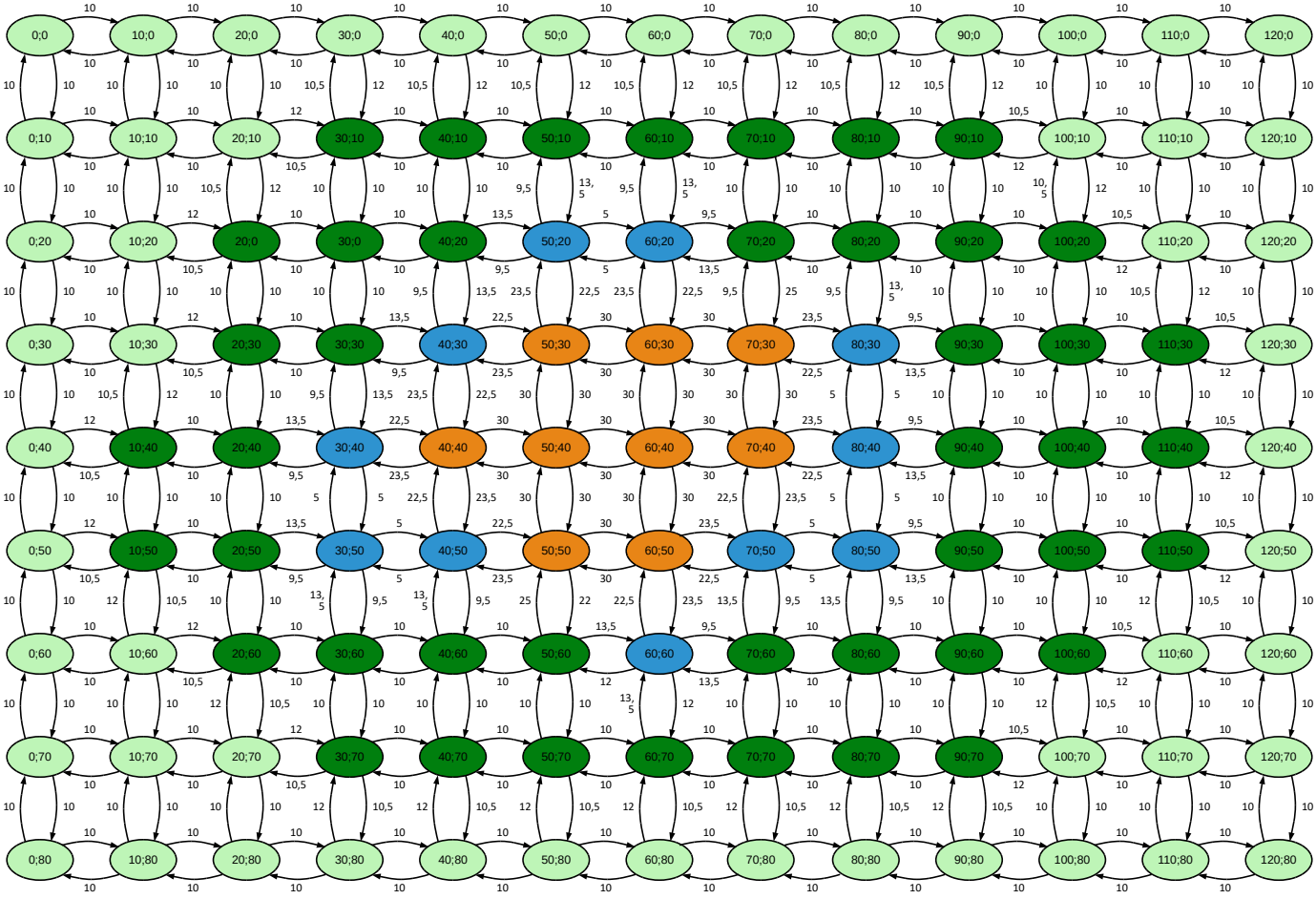
  Factorisable en :

$$\frac{\left(P_{B1_{nav}} + P_{B2_{nav}}\right) * d}{2} + P_{B2_{ent}}$$

  Le poids de l'arc étant déjà définit au poids entrant du biome de destination, il suffit de lui ajouter le premier opérande de l'addition, c'est-à-dire la somme des poids de navigation de chaque biome multiplié par la distance séparant les deux points, le tout divisé par deux.

  Bien sûr, ce n'est qu'une approximation : la vraie map n'a pas nécessairement une égale distance passée dans chaque biome entre les deux nœuds. Mais le rôle du maillage ici est de simplifier la map pour pouvoir calculer le plus court chemin. Là où le calcul avec précision de la proportion de temps passé dans chaque biome est possible, cela agrandirait le temps de calcul et irait à l'encontre du principe de « mailler pour simplifier ».
  Si l'on veut une précision plus grande, on peut toujours augmenter le nombre de nœuds du maillage.

Alors, on obtient le graphe suivant :



Même graphe orienté avec des poids décrivant correctement les temps

Ce graphe représente notre map. Bien que simplifié par le maillage, il reste plutôt fiable et nous servira de base pour calculer le plus court chemin entre deux points.

Pour compléter le travail, il n'y a plus qu'à appliquer un algorithme de recherche de plus court chemin tel que Dijkstra ou A*.

L'intérêt de cette approche est que la précision est ajustable : si l'on prend un maillage plus serré (= avec plus de nœuds), on aura un chemin plus rapide, mais celui-ci mettra plus de temps à être calculé.

Répartition des tâches

Nous avons séparé le projet en cinq grandes tâches :

1) La création et l'implémentation de la map
2) La transformation de la map en un maillage de n points par unité de coordonnées
3) L'écriture d'algorithmes trouvant le chemin le plus rapide entre deux points du maillage
4) L'organisation des diverses fonctions en un programme principal qui les appelle pour trouver le chemin le plus court entre deux points dans un graphe avec un maillage de précision choisie
5) L'écriture du présent rapport

Nous avons réparti ces tâches à travers notre trinôme de la manière suivante :

1) Matthieu
2) Hugo
3) Mamourou (Dijkstra) & Matthieu (A*)
4) Matthieu & Mamourou
5) Matthieu & Hugo

Notez que cette répartition n'est pas très précise : elle représente les rôles principaux de chaque membre, mais chacun a tout de même un peu contribué au travail des autres.

Création et implémentation de la map

Une map est composée des éléments principaux suivants :

- Une liste de biomes disponibles
- Une liste de polygones qui représentent les zones, chacun associé à un biome existant
- Un biome par défaut pour les zones non attribuées

Puisque les maps peuvent vite devenir grandes, leur création « à la main » en écrivant manuellement les coordonnées des points est très difficile.

Il a donc été décidé de créer un petit logiciel permettant de fabriquer des maps, puis de les sauvegarder dans un fichier qui pourra ensuite être lu par le programme de pathfinding.

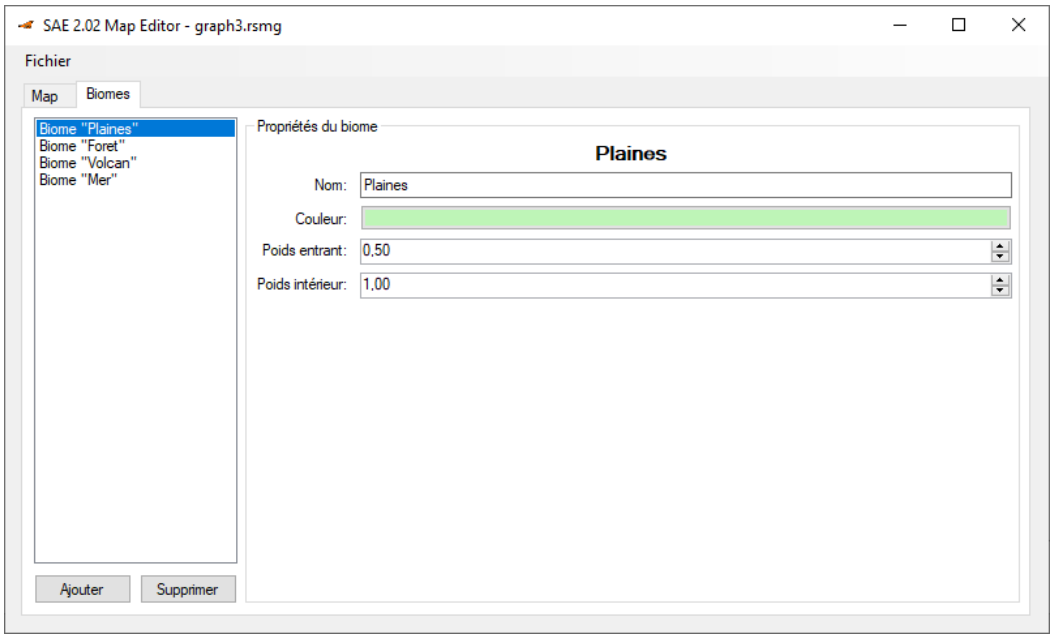
Logiciel d'édition de maps

Ce logiciel a été développé en C# avec Windows Forms, une infrastructure d'interface utilisateur qui permets de créer des applications de bureau pour Windows grâce au framework .NET.

Création de maps

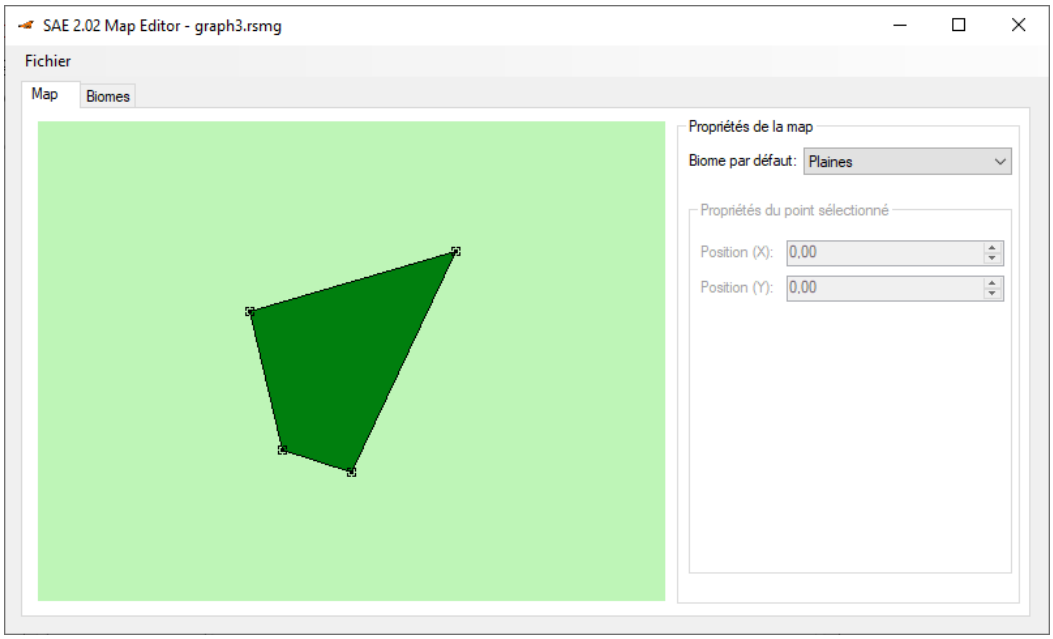
L'interface se présente en les trois parties suivantes :

- L'onglet d'édition de biomes



Une liste de biome est affichée sur la gauche, à laquelle il est possible d'ajouter ou de retirer des biomes.
Quand un biome y est sélectionné, il est possible d'éditer ses propriétés sur la droite.

- L’onglet d’édition de la map



La map actuelle est visible sur la gauche.

Il est possible d’ajouter le premier point d’un nouveau polygone à l’aide d’un clic droit sur celle-ci. Les points suivants s’ajoutent par des clics gauches, puis la validation du biome se fait par un clic droit qui reliera le dernier point au premier. Le biome correspondant au biome du nouveau polygone sera demandé à l’utilisateur.

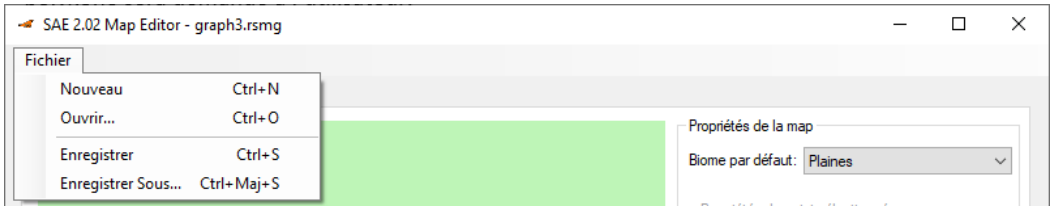
Un clic gauche sur un polygone le sélectionne, un double clic permets de changer son biome et l’appui sur la touche « Suppr » supprime le polygone sélectionné.

Un clic gauche sur un point existant le sélectionne et active la boite d’édition de ses coordonnées sur la droite.

Quand un clic est effectué pour placer un point, si celui-ci est fait proche d’un autre point, alors le nouveau point sera placé aux mêmes coordonnées que celui le plus proche. De même si une bordure d’un polygone se trouve non loin du clic.

Le biome par défaut pour les zones non attribuées peut être défini par le sélectionneur en haut à droite.

- Le menu supérieur



Celui-ci permets de :

- Réinitialiser le logiciel (« Nouveau »)
- Ouvrir une map existante (« Ouvrir… »)
- Enregistrer la map actuelle (« Enregistrer »)
- Enregistrer la map actuelle dans un nouveau fichier (« Enregistrer Sous… »)

Exportation des maps dans un fichier pour une utilisation ultérieure

Les maps créées peuvent être ensuite exportées dans des fichiers dont la structure est détaillée ici.

Notes : « Offset » signifie ici le nombre d'octet entre un élément et le début du fichier.
Le préfixe « 0x » indique un nombre hexadécimal.

En-tête

Offset	Type	Description
0x00	Byte[4]	Octets de vérification. Toujours égaux à 4D 41 50 42.
0x04	Int32	Numéro du biome par défaut. -1 si aucun.
0x08	UInt64	Offset de la liste de polygones.
0x10	UInt64	Offset de la liste de biomes.
0x18	Fin de l'en-tête.	

Liste de polygones

Offset	Type	Description
0x00	UInt32	Nombre de polygones.
0x04	UInt64[**N**]	Offset de chaque polygone.
0x04 + (0x08 × **N**)	Fin de la liste de polygones.	

Polygone

Offset	Type	Description
0x00	Int32	Numéro du biome associé au polygone. -1 si aucun.
0x04	UInt32	Nombre de points.
0x08	UInt64[**N**]	Offset de chaque point.
0x08+ (0x08 × **N**)	Fin du polygone.	

Point

Offset	Type	Description
0x00	Float	Coordonnée X du point (axe horizontal).
0x04	Float	Coordonnée Y du point (axe vertical).
0x08	Fin du point.	

Liste de biomes

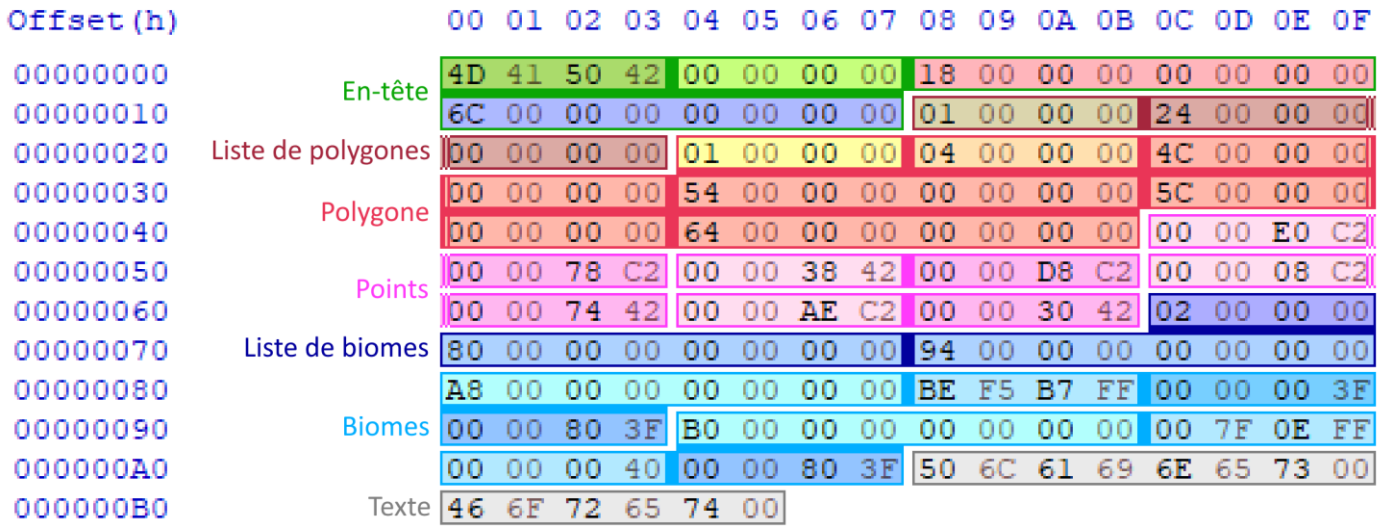
Offset	Type	Description
0x00	UInt32	Nombre de biomes.
0x04	UInt64[**N**]	Offset de chaque biome.
0x04 + (0x08 × **N**)	Fin de la liste de biomes.	

Biome

Offset	Type	Description
0x00	UInt64	Offset du texte contenant le nom du biome.
0x08	Byte[4]	Couleur du biome. Stocké sous la forme RR GG BB AA.
0x0C	Float	Valeur du poids entrant du biome.
0x10	Float	Valeur du poids intérieur du biome.
0x14	Fin du biome.	

Vue d'un fichier

Pour illustrer, voici les octets bruts du fichier contenant la map donnée dans la capture d'écran de l'interface du logiciel d'édition de maps :



Chaque bloc de couleur représente une structure, et chaque variante du surlignage d'un groupe d'octets est un attribut.

Ces fichiers peuvent être réimportés par la suite pour visualiser et modifier la map à nouveau.

De plus, ils peuvent être lu par n'importe quel programme connaissant la structure du fichier, l'intérêt étant de faire lire ce fichier à notre programme de pathfinding.

Programme d'exemple

Le programme de pathfinding étant entièrement programmé en C++, il a fallu faire une implémentation d'un objet lisant les fichiers exportés du logiciel d'édition de maps dans ledit langage.

Afin de tester cette implémentation avant l'écriture du programme final, un programme d'exemple écrivant l'intégralité des propriétés d'un fichier a été écrit. Pour la map donnée dans la capture d'écran de l'interface du logiciel, le programme affiche le contenu suivant :

```
Map:
- Biome par défaut: 0
- Polygones:
  - Nombre de polygones: 1
  - Polygone 0:
    - Biome: 1
    - Points:
      - Nombre de points: 4
      - Point 0:
        - X: -112.000000
        - Y: -62.000000
      - Point 1:
        - X: 46.000000
        - Y: -108.000000
      - Point 2:
        - X: -34.000000
        - Y: 61.000000
      - Point 3:
        - X: -87.000000
        - Y: 44.000000
- Biomes:
  - Nombre de biomes: 2
  - Biome 0:
    - Nom: Plaines
    - Couleur: #BEF5B7FF
    - Poids entrant: 0.500000
    - Poids intérieur: 1.000000
  - Biome 1:
    - Nom: Foret
    - Couleur: #007F0EFF
    - Poids entrant: 2.000000
    - Poids intérieur: 1.000000
```

Cela nous a permis de tester la bonne implémentation de l'objet lecteur de fichiers de map.

Génération du maillage

Explication du principe

Nous avons créé une fonction maillage qui prends en paramètre un pointeur vers la map ainsi qu'un flottant « `nbMailleUnite` » représentant le nombre de mailles par unité.

Celle-ci fut particulièrement laborieuse à élaborer, nous détaillerons donc nos étapes dans son écriture ici.

La première partie de la fonction permet de calculer les coordonnées minimales et maximales de la carte afin de déterminer sa hauteur et la largeur. On réalise ensuite des boucles imbriquées créer chaque nœud de la carte aux indices X et Y correspondant au variables « `nbMailleX` » et « `nbMailleY` » calculées grâce à la hauteur, la largeur et le nombre d'unité par maille de la carte.

A ce stade du programme, on a un tableau qui contient tous les nœuds de notre graphe. Il faut désormais pouvoir définir les arcs les reliant. Comme dit précédemment, deux nœuds sont reliés par deux arcs orientés.

Dans un premier temps, on va donc compter le nombre total d'arc pour pouvoir ensuite créer un tableau qui stockera tous nos arcs.

Une fois ce tableau créé, il faut créer chaque arc du graphe en fonction de l'emplacement de chacun des nœuds de celui-ci. En effet, comme chaque duo de nœuds juxtaposés est relié par deux arcs, on sait que :

- Un nœud dans un coin aura 2 arcs sortant
- Un nœud sur côté aura 3 arcs sortant
- Les autres nœuds auront tous 4 arcs sortant

Notre démarche a donc été de créer 4 booléens permettant de savoir si un nœud se trouve dans un coin, sur un côté ou autre. Ensuite, en fonction de ces booléens, des arcs sont créés avec un nœud de départ, un nœud d'arrivé et un poids.

Le poids d'un arc est calculé par la fonction « `CalculPoids` » qui prend en paramètre les deux nœuds reliés par l'arc en question. Dans un premier temps, on calcule la distance entre les deux points (Pythagore), puis en fonction de leur biome on calcule le poids comme écrit dans la section de description de notre approche.

Optimisation du code

Après l'écriture de la première version du code, nous nous sommes rendu compte que certaines parties du code pouvaient être optimisées.

Voici un bref aperçu de l'une de ces parties (création des arcs) :

```
bool backX = (x == 0);
bool backY = (y == 0);
bool forwX = (x == (nbMailleX-1));
bool forwY = (y == (nbMailleY-1));

if((backX && backY) || (backY && forwX) || (forwX && forwY) || (forwY && backX)) {
		// Coin
		if(backX)
		{
			arcs[arcActuel].n1 = &noeuds[indiceMaillage];
			arcs[arcActuel].n2 = &noeuds[(y * nbMailleX) + (x+1)];
			arcs[arcActuel].poids = CalculPoids(arcs[arcActuel].n1, arcs[arcActuel].n2);
			arcActuel++;
		}
		else
		{
			arcs[arcActuel].n1 = &noeuds[indiceMaillage];
			arcs[arcActuel].n2 = &noeuds[(y * nbMailleX) + (x-1)];
			arcs[arcActuel].poids = CalculPoids(arcs[arcActuel].n1, arcs[arcActuel].n2);
			arcActuel++;
		}

		if(backY)
		{
			arcs[arcActuel].n1 = &noeuds[indiceMaillage];
			arcs[arcActuel].n2 = &noeuds[((y+1) * nbMailleX) + x];
			arcs[arcActuel].poids = CalculPoids(arcs[arcActuel].n1, arcs[arcActuel].n2);
			arcActuel++;
		}
		else
		{
			arcs[arcActuel].n1 = &noeuds[indiceMaillage];
			arcs[arcActuel].n2 = &noeuds[((y-1) * nbMailleX) + x];
			arcs[arcActuel].poids = CalculPoids(arcs[arcActuel].n1, arcs[arcActuel].n2);
			arcActuel++;
		}
}
else if(backX || backY || forwX || forwY) {
		// Bord
		if(backX || forwX)
		{
			arcs[arcActuel].n1 = &noeuds[indiceMaillage];
			arcs[arcActuel].n2 = &noeuds[((y+1) * nbMailleX) + x];
			arcs[arcActuel].poids = CalculPoids(arcs[arcActuel].n1, arcs[arcActuel].n2);
			arcActuel++;

			arcs[arcActuel].n1 = &noeuds[indiceMaillage];
			arcs[arcActuel].n2 = &noeuds[((y-1) * nbMailleX) + x];
			arcs[arcActuel].poids = CalculPoids(arcs[arcActuel].n1, arcs[arcActuel].n2);
			arcActuel++;

			if(backX)
			{
				arcs[arcActuel].n1 = &noeuds[indiceMaillage];
				arcs[arcActuel].n2 = &noeuds[(y * nbMailleX) + (x+1)];
				arcs[arcActuel].poids = CalculPoids(arcs[arcActuel].n1, arcs[arcActuel].n2);
				arcActuel++;
			}
			else
			{
				arcs[arcActuel].n1 = &noeuds[indiceMaillage];
				arcs[arcActuel].n2 = &noeuds[(y * nbMailleX) + (x-1)];
				arcs[arcActuel].poids = CalculPoids(arcs[arcActuel].n1, arcs[arcActuel].n2);
				arcActuel++;
			}
		}
		else if (backY || forwY)
		{
			arcs[arcActuel].n1 = &noeuds[indiceMaillage];
			arcs[arcActuel].n2 = &noeuds[(y * nbMailleX) + (x+1)];
			arcs[arcActuel].poids = CalculPoids(arcs[arcActuel].n1, arcs[arcActuel].n2);
			arcActuel++;

			arcs[arcActuel].n1 = &noeuds[indiceMaillage];
			arcs[arcActuel].n2 = &noeuds[(y * nbMailleX) + (x-1)];
			arcs[arcActuel].poids = CalculPoids(arcs[arcActuel].n1, arcs[arcActuel].n2);
			arcActuel++;

			if(backY)
			{
				arcs[arcActuel].n1 = &noeuds[indiceMaillage];
				arcs[arcActuel].n2 = &noeuds[((y+1) * nbMailleX) + x];
				arcs[arcActuel].poids = CalculPoids(arcs[arcActuel].n1, arcs[arcActuel].n2);
				arcActuel++;
			}
			else
			{
				arcs[arcActuel].n1 = &noeuds[indiceMaillage];
				arcs[arcActuel].n2 = &noeuds[((y-1) * nbMailleX) + x];
				arcs[arcActuel].poids = CalculPoids(arcs[arcActuel].n1, arcs[arcActuel].n2);
				arcActuel++;
			}
		}
}
else {
		// Interne
		arcs[arcActuel].n1 = &noeuds[indiceMaillage];
		arcs[arcActuel].n2 = &noeuds[(y * nbMailleX) + (x+1)];
		arcs[arcActuel].poids = CalculPoids(arcs[arcActuel].n1, arcs[arcActuel].n2);
		arcActuel++;

		arcs[arcActuel].n1 = &noeuds[indiceMaillage];
		arcs[arcActuel].n2 = &noeuds[(y * nbMailleX) + (x-1)];
		arcs[arcActuel].poids = CalculPoids(arcs[arcActuel].n1, arcs[arcActuel].n2);
		arcActuel++;

		arcs[arcActuel].n1 = &noeuds[indiceMaillage];
		arcs[arcActuel].n2 = &noeuds[((y+1) * nbMailleX) + x];
		arcs[arcActuel].poids = CalculPoids(arcs[arcActuel].n1, arcs[arcActuel].n2);
		arcActuel++;

		arcs[arcActuel].n1 = &noeuds[indiceMaillage];
		arcs[arcActuel].n2 = &noeuds[((y-1) * nbMailleX) + x];
		arcs[arcActuel].poids = CalculPoids(arcs[arcActuel].n1, arcs[arcActuel].n2);
		arcActuel++;
}
```

Sans lire les détails, vous pouvez donc constater que… c'est illisible, peu performant et répétitif.

Pour pallier ce problème, nous avons commencé par éliminer la répétitivité en réduisant nos « `if` » en opérations booléennes seules qui ensuite déterminent le comportement de la boucle.

Nous obtenons donc le code suivant :

```
bool backX = (x == 0);
bool backY = (y == 0);
bool forwX = (x == (nbMailleX-1));
bool forwY = (y == (nbMailleY-1));

bool xP1 = false;
bool xM1 = false;
bool yP1 = false;
bool yM1 = false;

if((backX && backY) || (backY && forwX) || (forwX && forwY) || (forwY && backX)) {
        // Coin
        if(backX)
                xP1 = true;
        else
                xM1 = true;

        if(backY)
                yP1 = true;
        else
                yM1 = true;
}
else if(backX || backY || forwX || forwY) {
        // Bord
        if(backX || forwX)
        {
                yP1 = true;
                yM1 = true;

                if(backX)
                        xP1 = true;
                else
                        xM1 = true;
        }
        else if (backY || forwY)
        {
                xP1 = true;
                xM1 = true;

                if(backY)
                        yP1 = true;
                else
                        yM1 = true;
        }
}
else {
        // Interne
        xP1 = true;
        xM1 = true;
        yP1 = true;
        yM1 = true;
}

if (xP1) {
        arcs[arcActuel].n1 = &noeuds[indiceMaillage];
        arcs[arcActuel].n2 = &noeuds[(y * nbMailleX) + (x + 1)];
        arcs[arcActuel].poids = CalculPoids(arcs[arcActuel].n1, arcs[arcActuel].n2);
        arcActuel++;
}
if (xM1) {
        arcs[arcActuel].n1 = &noeuds[indiceMaillage];
        arcs[arcActuel].n2 = &noeuds[(y * nbMailleX) + (x - 1)];
        arcs[arcActuel].poids = CalculPoids(arcs[arcActuel].n1, arcs[arcActuel].n2);
        arcActuel++;
}
if (yP1) {
        arcs[arcActuel].n1 = &noeuds[indiceMaillage];
        arcs[arcActuel].n2 = &noeuds[((y + 1) * nbMailleX) + x];
        arcs[arcActuel].poids = CalculPoids(arcs[arcActuel].n1, arcs[arcActuel].n2);
        arcActuel++;
}
if (yM1) {
        arcs[arcActuel].n1 = &noeuds[indiceMaillage];
        arcs[arcActuel].n2 = &noeuds[((y - 1) * nbMailleX) + x];
        arcs[arcActuel].poids = CalculPoids(arcs[arcActuel].n1, arcs[arcActuel].n2);
        arcActuel++;
}
```

Cependant, celui-ci est encore améliorable.

Nous avons alors fait un tableau pour repérer d'éventuels liens entre nos quatre booléens et nos arcs à créer. On obtient donc le tableau suivant (0 = pas de changements ; 1 = un arc doit être créé) :

backX	backY	forwX	forwY	X + 1	X – 1	Y + 1	Y - 1
0	0	0	0	1	1	1	1
1	0	0	0	1	0	1	1
0	1	0	0	1	1	1	0
0	0	1	0	0	1	1	1
0	0	0	1	1	1	0	1
1	1	0	0	1	0	1	0
0	1	1	0	0	1	1	0
0	0	1	1	0	1	0	1
1	0	0	1	1	0	0	1

Grâce à ce tableau on peut remarquer un lien entre les différentes colonnes du tableaux (mis en évidence deux à deux par les couleurs). Ce lien est un lien de négativité : chaque 0 devient un 1 et inversement (opération « not »). On peut donc réduire notre code à 4 booléens et donc 4 conditions :

- X-1 si on n'est pas sur le X minimal
- X+1 si on n'est pas sur le X maximal
- Y+1 si on n'est pas sur le Y maximal
- Y-1 si on n'est pas sur le Y minimal

On obtient donc le code suivant :

```
bool xPlus1 = (x < (nbMailleX-1));
bool xMoins1 = (x > 0);
bool yPlus1 = (y < (nbMailleY-1));
bool yMoins1 = (y > 0);

if(xPlus1) {
        arcs[arcActuel].n1 = &noeuds[indiceMaillage];
        arcs[arcActuel].n2 = &noeuds[(y * nbMailleX) + (x+1)];
        arcs[arcActuel].poids = CalculPoids(arcs[arcActuel].n1, arcs[arcActuel].n2);
        arcActuel++;
}
if(xMoins1) {
        arcs[arcActuel].n1 = &noeuds[indiceMaillage];
        arcs[arcActuel].n2 = &noeuds[(y * nbMailleX) + (x-1)];
        arcs[arcActuel].poids = CalculPoids(arcs[arcActuel].n1, arcs[arcActuel].n2);
        arcActuel++;
}
if(yPlus1) {
        arcs[arcActuel].n1 = &noeuds[indiceMaillage];
        arcs[arcActuel].n2 = &noeuds[((y+1) * nbMailleX) + x];
        arcs[arcActuel].poids = CalculPoids(arcs[arcActuel].n1, arcs[arcActuel].n2);
        arcActuel++;
}
if(yMoins1) {
        arcs[arcActuel].n1 = &noeuds[indiceMaillage];
        arcs[arcActuel].n2 = &noeuds[((y-1) * nbMailleX) + x];
        arcs[arcActuel].poids = CalculPoids(arcs[arcActuel].n1, arcs[arcActuel].n2);
        arcActuel++;
}
```

Nettement plus court et plus lisible, on est passé de 116 lignes à 29 lignes, soit une réduction de 75%.

Le code a légèrement changé depuis, notamment pour ajouter les biomes obstacles.

Calcul du plus court chemin

Algorithme de Dijkstra

Explication du principe

Cet algorithme consiste en le parcours « brut » de chaque nœud en notant leurs prédécesseurs le plus rapide afin de trouver le chemin le plus court.

Bien que cet algorithme fonctionne à la perfection, il est de complexité $O((a+n)\log(n))$ (pour n nœuds et a arcs), donc bien plus élevé que d'autres algorithmes plus poussés.

Le principe de Dijkstra est de créer un tableau de distances portant une case pour chaque nœud du graphe, chacune initialisée à $+\infty$, à l'exception de la case du nœud de départ qui sera à 0.

Chaque nœud a une marque (un booléen à `true` si le nœud est marqué, `false` sinon) initialisée à false par défaut.

Ensuite, on répète la suite d'instructions suivante jusqu'à ce que tous les nœuds soient marqués :

- Prendre le nœud non marqué ayant la valeur la plus petite
- Vérifier ses voisins
- Si un voisin a une distance plus grande que celle du nœud actuel + le poids du chemin qui les relie, la mettre à jour avec la valeur mentionnée et stocker le nœud actuel comme son prédécesseur
- Marquer le nœud

Il ne reste plus qu'à remonter les prédécesseurs en partant du nœud d'arrivée et voilà, on a notre chemin.

Optimisation du code

À l'origine, l'algorithme de Dijkstra utilise des nombres pour catégoriser ses nœuds.

Or, nous utilisons une structure « Noeud » plutôt qu'un simple numéro de nœud. Cela s'explique par le fait que les nœuds doivent retenir leurs coordonnées ainsi que leur biome.

Alors, nous avons décidé d'utiliser l'index des nœuds dans la liste de nœuds du maillage comme numéro, et notre fonction renvoie à la fin un tableau de pointeurs vers chacun des nœuds concernés par le chemin.

De cette manière, nous optimisons notre espace en n'allouant que l'équivalent en octets d'une adresse par nœud, plutôt qu'une copie du nœud entier. L'utilisation d'index pour le fonctionnement interne de Dijkstra permets également un accès très rapide à travers les tableaux, en $O(1)$.

Algorithme A*

Explication du principe

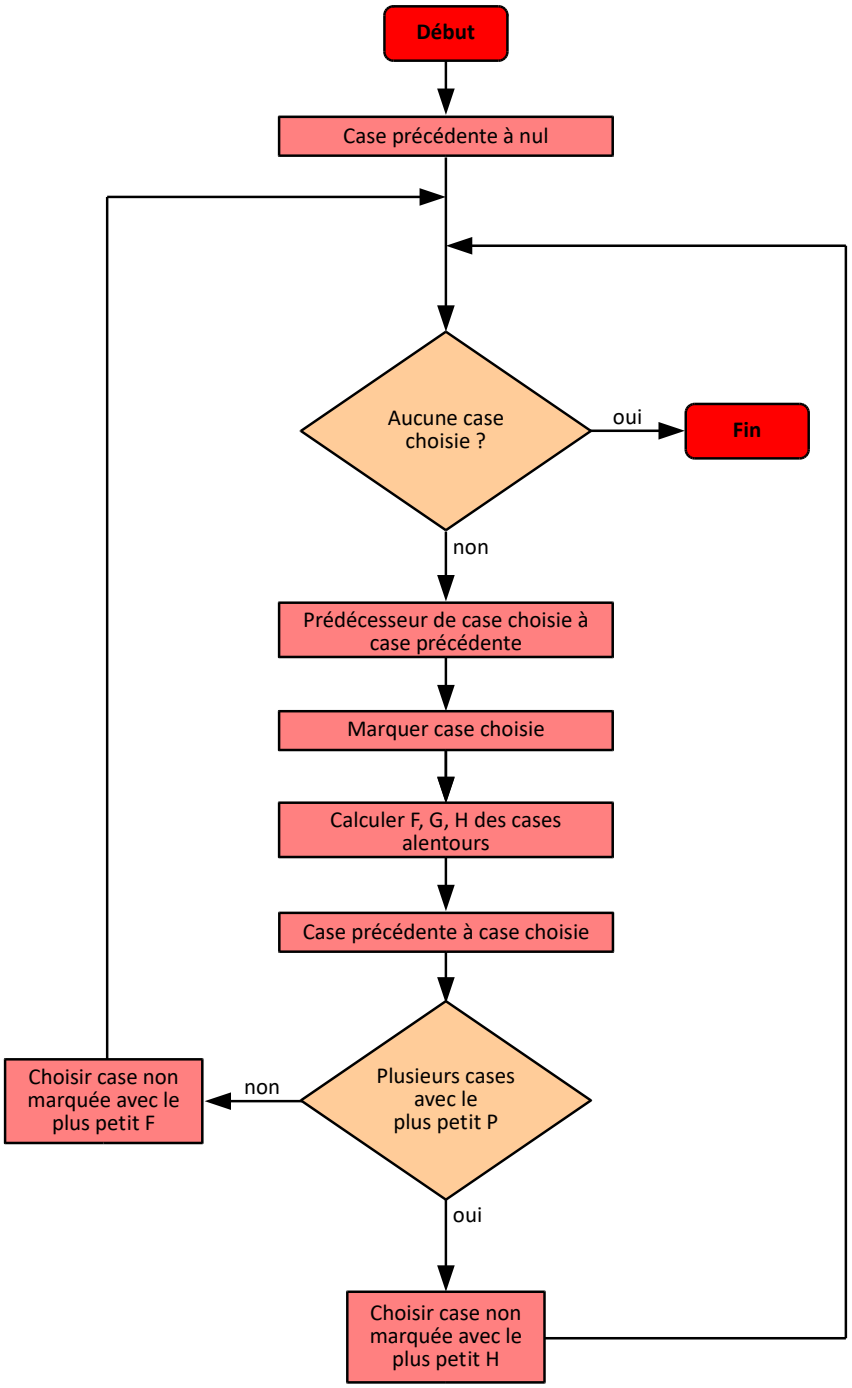
Cet algorithme consiste en la recherche du plus court chemin dans un graphe entre un nœud initial et un nœud final, tous deux donnés.

Cet algorithme est plus performant que Dijkstra avec une complexité en temps de $O(|E|) = O(b^d)$ et en espace de $O(|V|) = O(b^d)$.

Pour chaque nœud on calcule trois variables :

- G la distance entre le nœud sélectionné et le point de départ
- H une distance heuristique entre le nœud sélectionné et le point d'arrivée
- F = G + H

Puis, il ne nous reste qu'à suivre l'algorithme suivant :



Calcul d'une distance heuristique entre deux nœuds

L'un des plus grands problèmes dans l'implémentation de cet algorithme fut le calcul de la valeur H de nos nœuds.

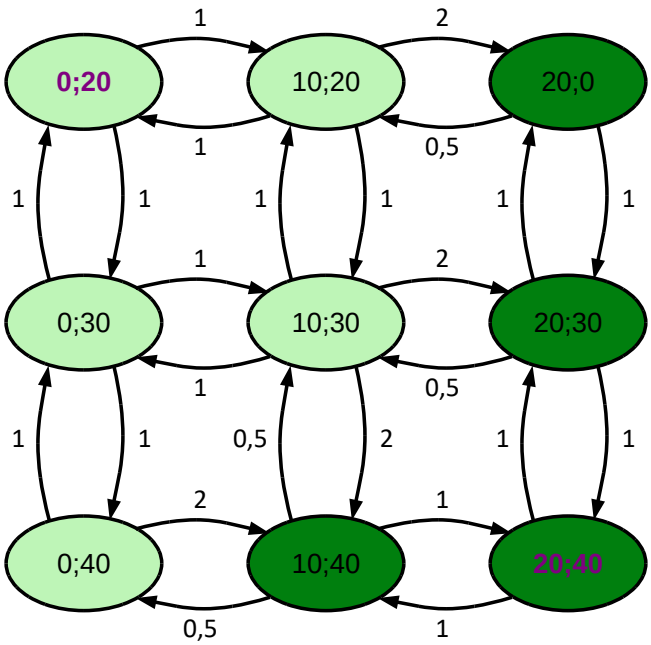
En effet, A* est conçu pour fonctionner dans des repères sous forme de tuiles, mais avec des distances dépendantes des coordonnées de chaque tuile plutôt que des poids arbitrairement définis.

Ainsi, le calcul de H, c'est-à-dire de la distance heuristique entre un nœud et le nœud d'arrivé, est une tâche plutôt complexe.

Une distance heuristique est une distance approximative qui peut être calculée rapidement, mais qui n'est pas nécessairement optimale ou exacte.

L'intérêt étant d'estimer pour notre algorithme la distance entre un nœud et le nœud d'arrivée rapidement et sans trop de précision, afin de pouvoir lui indiquer une direction à privilégier. On ne peut pas trouver l'exacte distance la plus courte puisque l'intérêt de notre algorithme est de trouver un chemin qui s'en rapproche au maximum.

Nous pouvons imager le problème sous la forme suivante : prenons cet extrait de map et disons que nous cherchons la distance entre le point **0;20** et le point **20;40**.



Nous avons plusieurs moyens de calculer la distance entre ces deux points.

Distance euclidienne

La réponse la plus évidente serait de calculer la distance euclidienne, c'est-à-dire la longueur du segment reliant les deux points « à vol d'oiseau ». Cette distance se calcule avec la formule suivante :

Pour :

d la distance entre le point A et le point B

x_P et y_P les coordonnées respectives x et y du point P

Alors :

$$d = \sqrt{(x_B - x_A)^2 + (y_B - y_A)^2}$$

Dans notre exemple, on aurait donc $d = \sqrt{(20-0)^2 + (40-20)^2} \approx 28{,}284$

Cette valeur est cependant très peu utilisable, puisqu'elle ne prend pas en compte les propriétés des biomes de notre graphe et la nature orientée qu'elles lui procure.

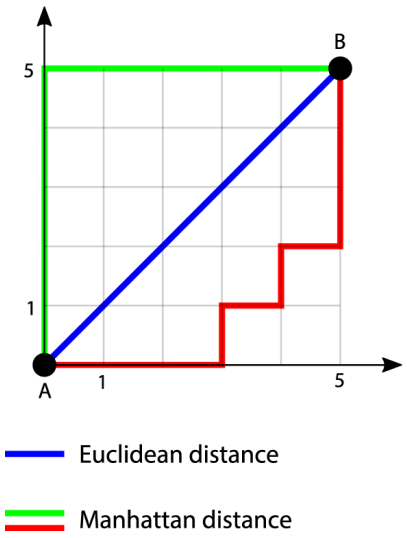
Une distance heuristique n'est certes qu'une estimation, mais cette imprécision va totalement à l'encontre de l'esprit du projet qui est de considérer les contraintes de la map.

Distance de Manhattan

Principe

Penchons-nous donc vers un deuxième type de distance : celle de Manhattan.

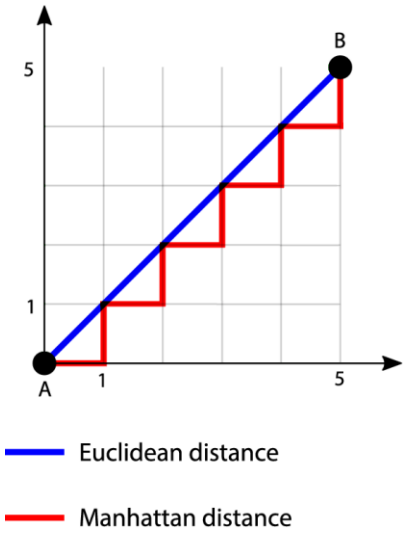
Cette distance représente l'image de la distance parcourue par un véhicule dans une ville où les rues forment un quadrillage :



Par sa nature, la distance de Manhattan n'est pas un seul chemin, mais une multitude de chemins qui relient deux points mais ont la même valeur.

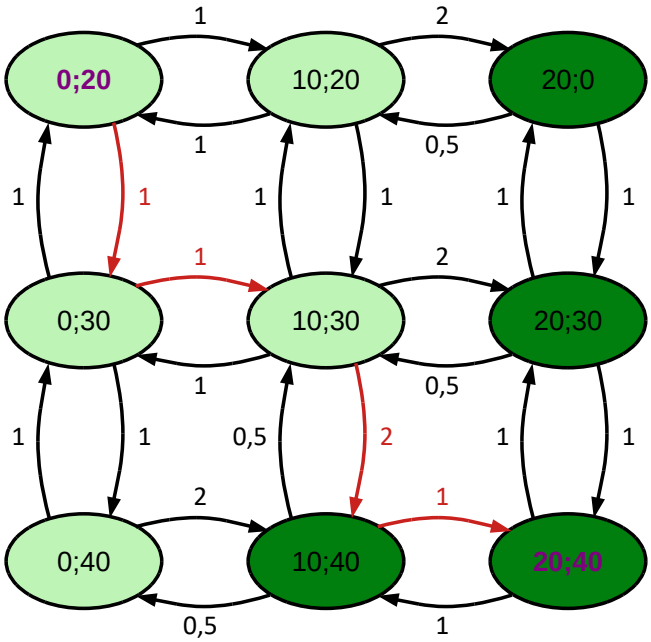
Notre version

Nous avons donc choisi d'utiliser la distance de Manhattan avec le modèle suivant :



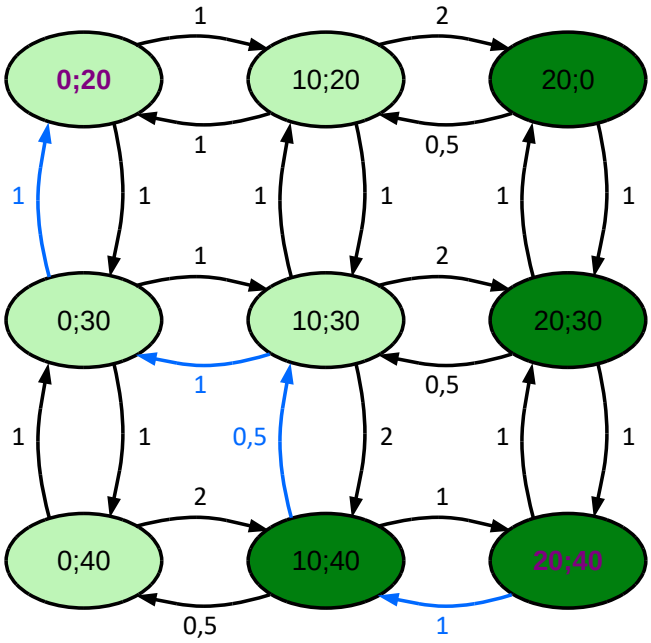
Ce modèle consiste en l'utilisation des points les plus proches de la ligne tracée par la distance euclidienne.

Ainsi, pour l'extrait de map utilisé précédemment, nous calculons notre distance heuristique de la manière suivante :



Les flèches rouges représentent notre chemin. Ainsi, notre distance serait de $1 + 1 + 2 + 1 = 5$.

L’intérêt étant, les propriétés de nos biomes sont prises en compte et la nature orientée du graph également :



Dans l’autre sens, on aurait donc une distance de $1 + 0{,}5 + 1 + 1 = 3{,}5$.

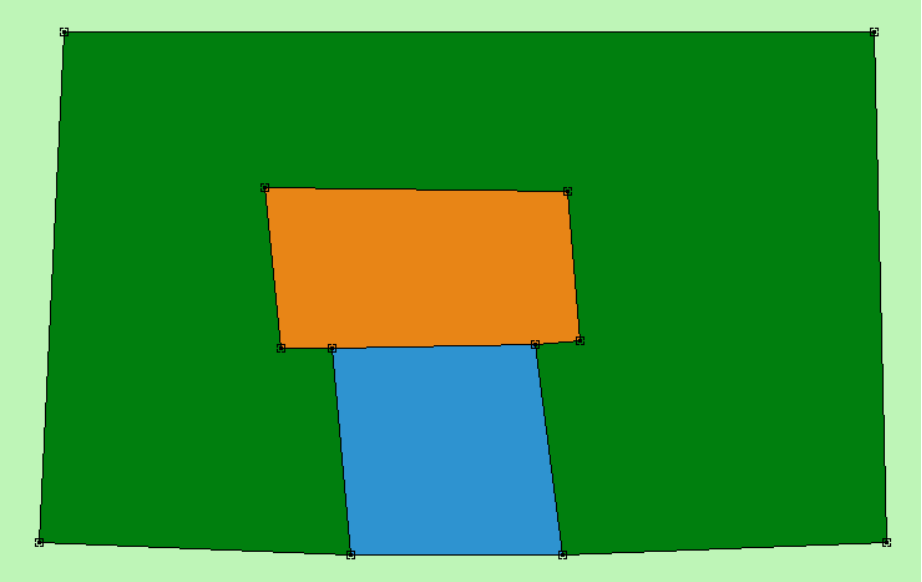
Notre problème est donc réglé, et, bien que cette distance ne soit pas optimale, elle représente au mieux le but à atteindre dans un temps raisonnable de calcul.

Analyses de performances

L'un des objectifs de cette étude est d'analyser les performances de nos calculs afin d'établir un lien entre l'affinage du maillage et le temps de calcul.

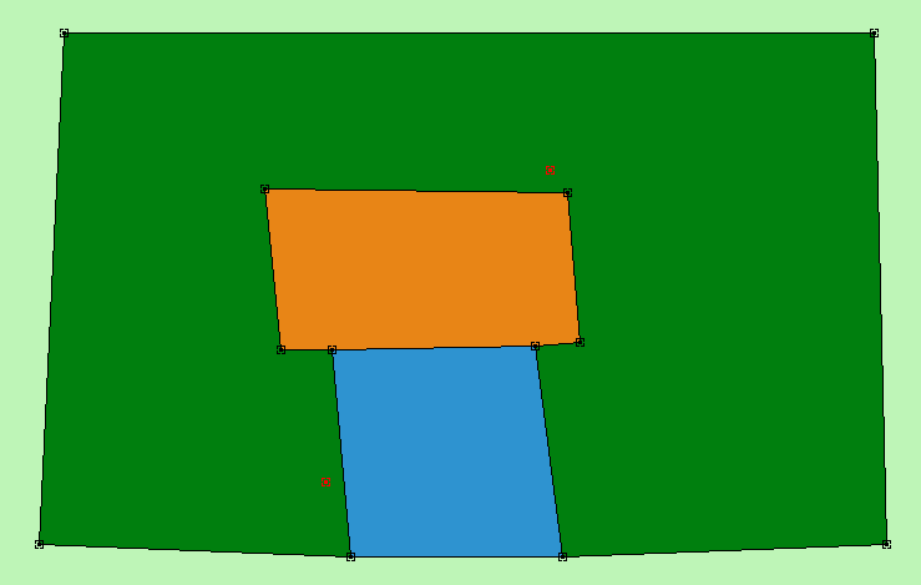
Contexte

Nous avons donc confectionné la map suivante :



Celle-ci a pour extrémités horizontales « -32 » et « 36 », et pour extrémités verticales « -27 » et « 15 ».

Annotés en rouge les deux points dont nous voulons calculer le chemin :



Point de départ (au milieu gauche bas) : X = -9 ; Y = 9

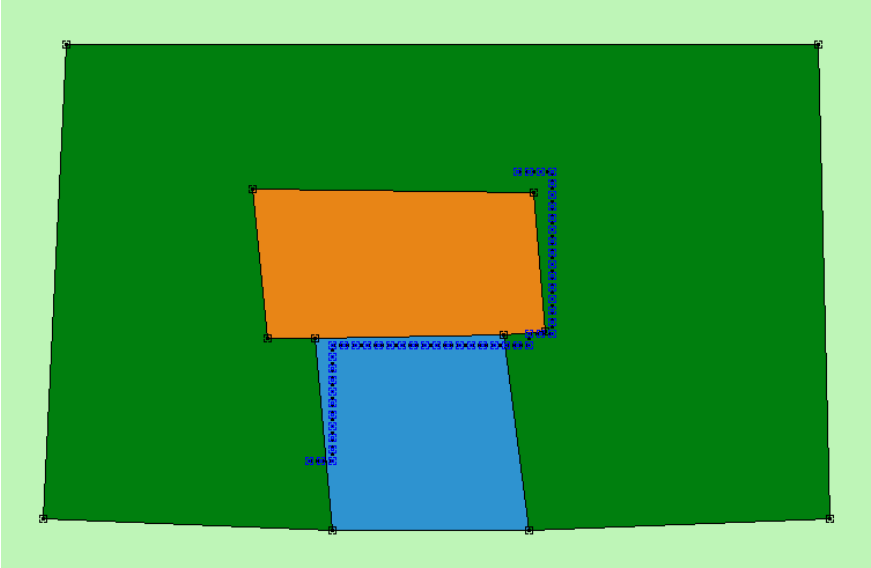
Point d'arrivée (au milieu droit haut) : X = 9 ; Y = -16

Les biomes et leurs poids sont les mêmes que dans l'explication de l'approche.

Chemins calculés

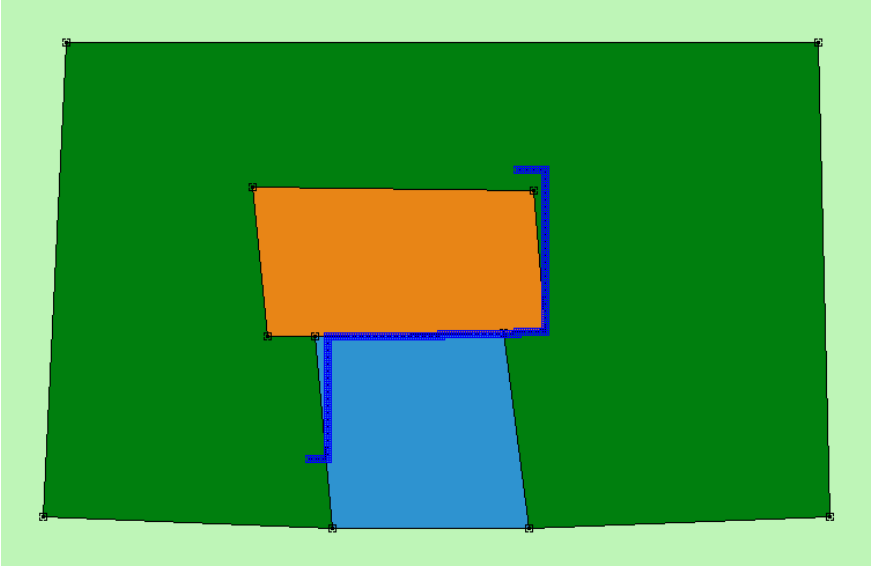
Dijkstra

L'algorithme trouve le résultat suivant avec 1 maille par unité :



On peut voir qu'il a choisi de passer par la rivière, mais a contourné le volcan.

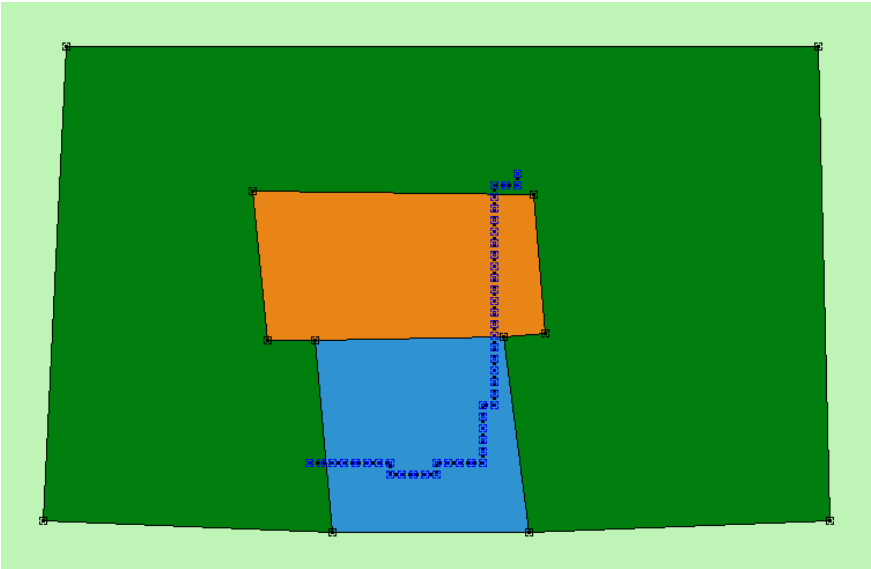
Avec 5 mailles par unité, on obtient le résultat suivant :



On peut remarquer que le chemin serre plus le bord de la rivière.

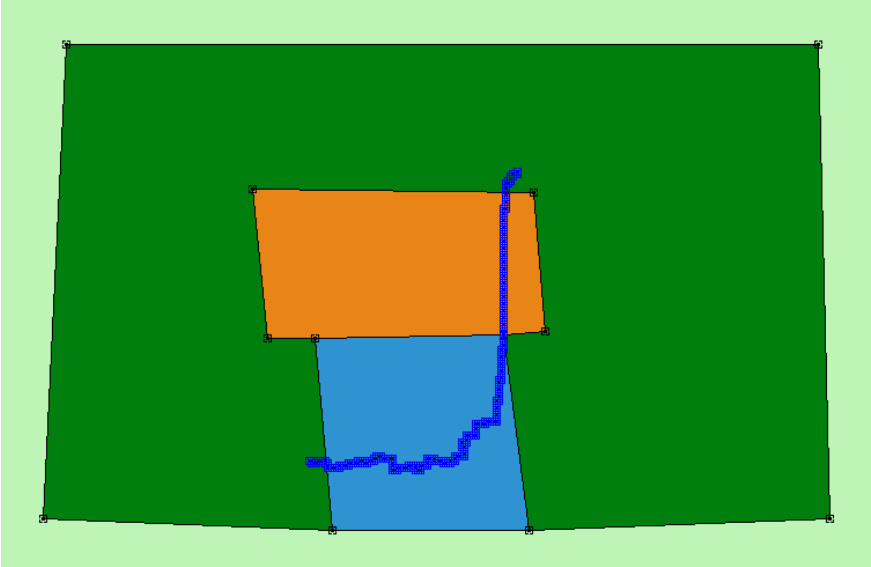
A*

L’algorithme trouve le résultat suivant avec 1 maille par unité :



Cette fois-ci, on passe à travers la rivière de manière un peu bancale, et on passe également à l’intérieur du volcan.

Avec 5 mailles par unité, on obtient le résultat suivant :



Le chemin choisi est presque le même, avec plus d’arrondis.

Mesures

Statistiques

Les statistiques mesurés sont les suivants.

Pour la génération du maillage :

Nombre de mailles par unité	**1**	**2**	**3**	**4**	**5**	**6**
Temps	<1ms	3ms	5ms	9ms	14ms	20ms
Nombre de mailles	2856	11424	25704	45696	71400	102816

Pour le calcul du chemin :

Nombre de mailles par unité / **Temps par algorithme**	**1**	**2**	**3**	**4**	**5**	**6**
Dijkstra	0,1s	1,579s	8,139s	25,558s	66,692s	140,250s
A*	0,013s	0,190s	1,017s	3,193s	8,202s	16,518s

On peut constater que le temps de génération du maillage est très faible et ne représente que très peu du temps de calcul.

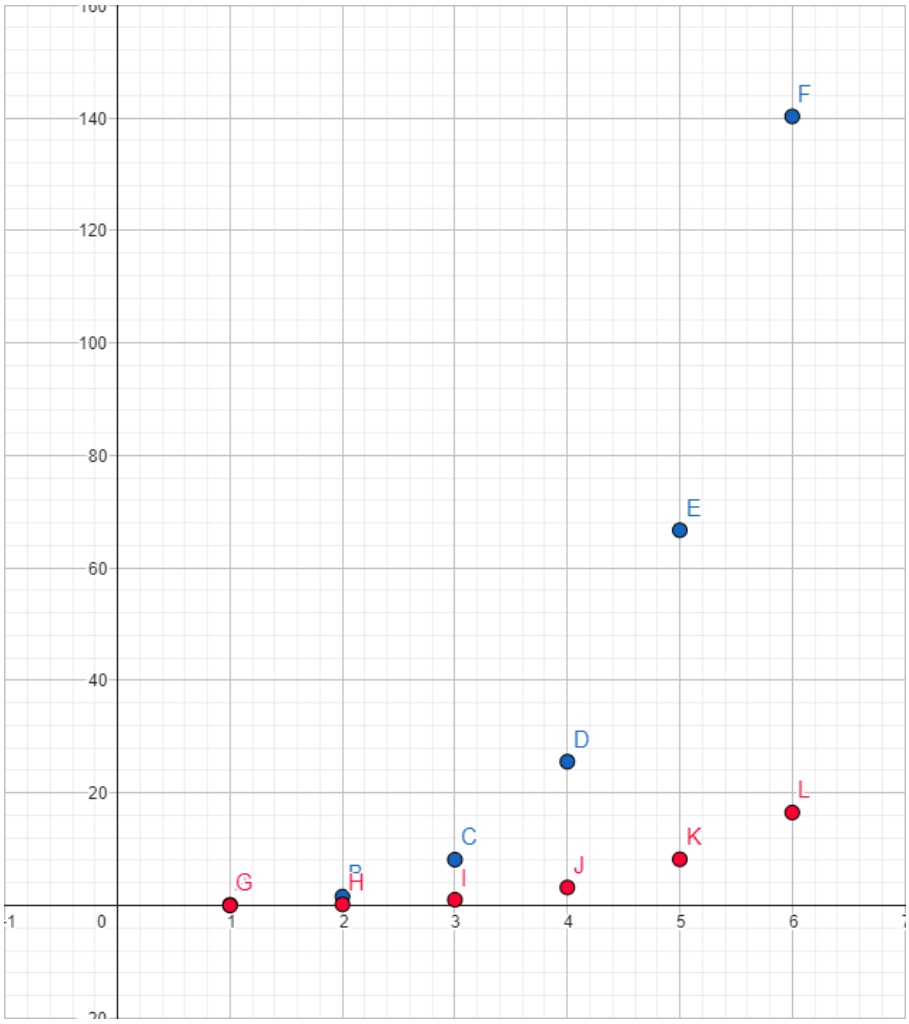
On peut également voir que A* est nettement plus rapide :

- 7,7 fois plus rapide pour une maille par unité
- 8,5 fois plus rapide pour six mailles par unité

Or, A* ne calcule pas une solution optimale contrairement à Dijkstra qui calcule la meilleure solution possible.

Analyses

Afin de pouvoir établir une fonction d'estimation des temps de calculs, nous avons représenté les temps (en secondes) en fonction du nombre de mailles par unité sur GeoGebra :

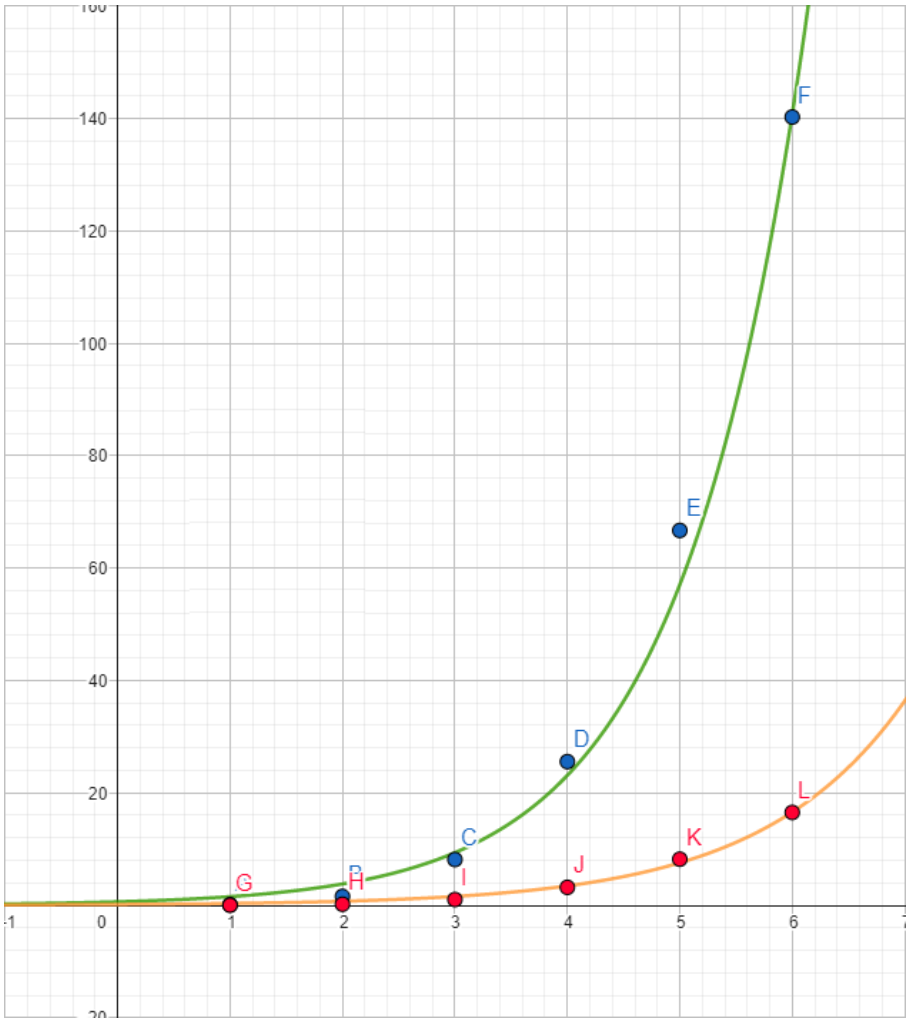


En bleu les points pour Dijkstra (A à F -> 1 à 6 mailles par unités
En rouge les points pour A (G à L -> 1 à 6 mailles par unités*

Nous pouvons constater que le temps de calcul a une tendance exponentielle, quel que soit l'algorithme.

Geogebra nous permet de faire une régression exponentielle pour prédire (avec un certain degré d'incertitude) le temps de calcul pour un nombre de mailles par unité donné.

On obtient donc le graphique suivant :



En vert l'exponentielle estimée de Dijkstra : $0{,}6233384962655e^{0{,}903284266726x}$
En orange l'exponentielle estimée de A :* $0{,}1493297475483e^{0{,}784998953008x}$

Nous pouvons constater que bien que les temps de calculs des deux algorithmes soient exponentiels, leurs évolutions sont d'une vitesse très différente.

Nous pouvons ainsi estimer les temps de calcul suivants :

Nombre de mailles par unité / **Temps par algorithme**	**7**	**8**	**9**	**10**	**20**	**30**
Dijkstra	≈ 6 mins	≈ 14 mins	≈ 35 mins	≈ 1h27	≈ 1,4 ans	≈ 11 605 ans
A*	36s	80s	≈ 3 mins	≈ 6 mins	≈ 11 jours	≈ 80 ans

Déductions

Pour 30 mailles par unité, on est donc 145 fois plus rapide grâce à A*.

L'intérêt n'étant bien entendu pas d'aller aussi loin, mais de démontrer le fait suivant : Un algorithme optimal mettra très souvent un temps bien trop conséquent à effectuer ses calculs. Il faut donc opter pour des algorithmes qui s'approchent au mieux de la solution optimale pour obtenir un résultat satisfaisant dans un temps raisonnable.

Dans notre cas, s'il était mis en pratique dans un jeu vidéo par exemple, on ne dépasserait pas les deux ou trois mailles par unité.

Mais pour notre toute petite map, un chemin sur 3 mailles par unité mettrait plus de 8 secondes à se calculer avec Dijkstra, contre 1 seconde avec A*. Cet algorithme est donc préférable.

Conclusion

Voilà qui conclut ce rapport.

Nous avons pu, au cours de ce projet, réfléchir sur un sujet mettant en jeu de l'algorithmie, des mathématiques et de l'optimisation.

Hugo, Mamourou, Matthieu