Implémentation de l'algorithme du fast-marching pour la résolution d'un labyrinthe

Jean Prost Lucas Potin Edouard Gouteux

21 novembre 2019

Table des matières

1 L'algorithme du fast marching **3**
1.1 Schéma numérique 3

2 Implementation **5**
2.1 Utilisation du programme 5
2.2 ClassMap.py 5
2.3 Implémentation du tri 6

3 Application à la recherche du plus court chemin dans un labyrinthe **6**
3.1 Première approche 6
3.2 2 fois plus de fast marching ! 7
3.3 Autres labyrinthes 7

1 L'algorithme du fast marching

La méthode du fast marching a été introduite par James Setian. Cette méthode possède notamment des applications en mécanique des fluides et en traitement d'image. Cette méthode permet de résoudre l'équation d'Eikonal, de la forme :

$$|\nabla T| = \mathcal{F} \tag{1}$$

Ici, $\mathcal{F}$ et T sont des fonctions $\mathbb{R}^n \to \mathbb{R}$, ou n peut prendre la valeur 1, 2, où 3. $\mathcal{F}$ est la métrique donné du problème, et T est la fonction à déterminer, avec comme condition initiale $T(x_0) = 0$. Ici nous nous concentrerons sur des fonctions $\mathcal{F}$ et T de $\mathbb{R}^2 \to \mathbb{R}$:

$$|\nabla T(x,y)| = \mathcal{F}(x,y) \tag{2}$$

L'algorihme du fast-marching nous permet d'obtenir la solution de viscosité de l'équation d'Eikonal. D'un point de vue géométrique, la fonction $T(x,y)$ obtenue correspond alors à la distance au point initial $x0$ sur la surface $x, y, \mathcal{F}(x,y)$, la distance étant alors donnée par $ds^2 = \mathcal{F}^2(x,y)(dx^2 + dy^2)$.
On suppose que la fonction F nous est donnée sur une grille $(i\Delta x, j\Delta y)_{i=\{1,..,N\},j=\{1..M\}}$.
On cherche alors à connaitre la valeur de T pour tous sommet de la grille.

1.1 Schéma numérique

L'algorithme du fast-marching repose sur approximation "ascendante" du gradient donnée par :

$$|\nabla T(x,y)|^2 \approx \left(\left(max\left(D_{ij}^{-x}T, -D_{ij}^{+x}T, 0 \right) \right)^2 + \left(max\left(D_{ij}^{-y}T, -D_{ij}^{+y}T, 0 \right) \right)^2 \right)^{\frac{1}{2}} \tag{3}$$

Où D_{ij} est l'opérateur approchant la dérivée en $T_{ij} \equiv T(i\Delta x, j\Delta y)$:

$$D_{ij}^{-x}T = \frac{(T_{ij} - T_{i-1,j})}{h} \tag{4}$$

L'algorithme consiste donc à résoudre :

$$\begin{cases} \left(max\left(D_{ij}^{-x}T, -D_{ij}^{+x}T, 0 \right) \right)^2 + \left(max\left(D_{ij}^{-y}T, -D_{ij}^{+y}T, 0 \right) \right)^2 & = & \mathcal{F}_{ij}^2 \\ T(x_0, y_0) & = & 0 \end{cases} \tag{5}$$

On suppose $\Delta x = \Delta y = h = 1$. On a alors :

$$max\left(D_{ij}^{-x}T, -D_{ij}^{+x}T, 0 \right) = max\left(T_{i,j} - min\left(T_{i-1,j}, T_{i+1,j} \right), 0 \right)$$

L'approximation 5 devient :

$$\left(max\left\{ T_{i,j} - min\left\{ T_{i-1,j}, T_{i+1,j} \right\}, 0 \right\} \right)^2 + \left(max\left\{ T_{i,j} - min\left\{ T_{i,j-1}, T_{i,j+1} \right\}, 0 \right\} \right)^2 = \mathcal{F}_{ij}^2 \tag{6}$$

En posant respectivement $t = T_{ij}$, $f = \mathcal{F}_{ij}$, $T_1 = min(T_{i-1,j}, T_{i+1,j})$ et $T_2 = min(T_{i-1,j}, T_{i+1,j})$, on obtient alors :

$$\{(max\{t - T_1\}, 0\})^2 + (max\{\{t - T_2\}, 0)^2 = f^2 (7)$$

Ici, seulement t est inconnu. On a alors 3 cas possibles :

— $t > maxT_1, T_2$. Dans ce cas on a

$$t = \frac{T_1 + T_2 + \sqrt{2f^2 - (T_1 - T_2^2}}{2}$$

— $T_2 > t > T_1$. Dans ce cas on a

$$t = T_1 + f$$

— $T_1 > t > T_2$. Dans ce cas on a

$$t = T_2 + f$$

On ajoute ensuite une initialisation , en fixant les T : Si une conidition au bord est associée à T , on l'affecte , sinon on l'affecte a ∞. On peut alors determiner une fonction pour mettre à jour un sommet , c'est à dire :

— Calculer la nouvelle valeur de T

— Modifier l'ordre de Q si besoin

La mise à jour d'un point de maillage suit alors l'algorithme suivant :
MaJ(u,Q)

(i, j)=u
$T_1 = (minT_{i-1,j}, T_{i+1,j})$
$T_2 = (minT_{i,j-1}, T_{i,j+1})$
if $|T_1 - T_2| < \mathcal{F}_{ij}$ then
—> t = $\frac{T_1+T_2+\sqrt{2f^2-(T_1-T_2^2}}{2}$
else
—> t= $minT_1, T_2 + \mathcal{F}_{ij}$
$T_{ij} = minT_{ij}, t$
if u $\in Q$ *then*
$--- > Tri(u, Q)$
else
$-- > u = Union(Q, u)$

La valeur d'un point qu'on a mis à jour est toujours plus grande que les valeurs des points utilisés pour calculer la mise à jour.
L'idée de l'algorithme Fast Marching va alors être celle utilisée dans l'algorithme de Dijkstra : On a un groupe S de sommets ayant un valeur finie pour T , et G le groupe de tout les sommets du maillage. Les valeurs de T pour les sommets $\in$ S étant fixées, cela veut soit dire l'on est dans le cas d'une condition aux bords fixée , ou que T vérifie l'approximation indiquée plus haut.On va placer tout

les sommets Adj(S), c'est à dire les adjacents a S dans une file G. On va alors chercher le T de valeur minimale, et transferer le sommet associé dans S. Une fois ceci fait , il nous faut mettre à jour la valeur des sommets adjacents au sommet qu'on vient de sélectionner.

On retrouve alors l'algorithme du Fast Marching suivant :

Fast Marching(F,G,S,Q)
$\forall v \in G$ privé de S faire $T_v = \infty$
tant que Q différent de 0
— u = Min(Q)
— S = Union(S,u)
— Pour chaque v $\in Adj(u)$
$- - - - - - - Maj(Q, v)$
$FinPour$
$FintantQue$

Il reste alors à determiner quelle méthode utiliser pour récuperer le min de Q. Un moyen efficace de trier nos valeurs dans notre file est alors le heapsort (tri par tas)

2 Implementation

2.1 Utilisation du programme

Pour ce projet , nous avons décidé d'utiliser python3. Pour utiliser le programme, il est nécéssaire de lancer le script "ClassMap.py", puis "testFM2.py". Il faut alors modifier 3 paramètres :

- ligne 14 : name = 'maze8'. Il faut ici indiquer le nom de l'image correspondant au labyrinthe
- ligne 43 : p0 = (X,Y). Ici , on remplace les coordonnées par le point qu'on considère point de départ
- ligne 54 : m2.calculGeodesic((X,Y)) : Ici , on remplace les coordonnées par le point qu'on considère point d'arrivée.

2.2 ClassMap.py

Ce script implémente les outils nécessaire à la réalisation du FastMarching :

- Tout d'abord , on implémente une classe sommet, contenant un attribut statut , indiquant si le sommet est front/far/visited, et un attribut value , représentant la valeur de T sur le sommet
- Ensuite , on aura une classe DistanceMap, proposant quand à elle les fonctions utilisées dans l'algorithme du Fast Marching.

Parlons maintenant des fonctions spécifiques au Fast Marching :

- init : initialise la méthode , en mettant les attributs des sommets en FAR ou en adjacent selon une liste de points de départ donnée.

— calculerDistance : Boucle sur iterate avec condition de sortie si le front est vide
— iterate : Realise une iteration de l'algorithme Fast Marching : algorithme au dessus
— update : Met a Jour la valeur de T pour le sommet , algorithme au dessus
— distanceMap : calcule la solution de l'équation d'Eikonal T
— calculGeodesic : réalise une descente de gradient pour calculer le plus court chemin du point p au point p0

Une utilisation du Fast Marching va alors être de la forme suivante :
— DistanceMap (appelle init)
— calculerDistance
— calculGeodesic

2.3 Implémentation du tri

Afin d'obbtenir un coût de calcul acceptable, il est nécessaire d'implementer un tri efficace, ici le tri par tas : nous avons pu ici utiliser l'implementation proposée par la librairie heapq.py.

3 Application à la recherche du plus court chemin dans un labyrinthe

3.1 Première approche

On cherche à trouver le plus court chemin pour aller d'un point a à un point b dans un labyrinthe. Le labyrinthe nous est donné comme une image en noir blanc. Tout d'abord on normalise l'image : aux pixels blanc (où l'on peut passer) on affecte la valeur 1, aux pixels noir (que l'on ne peut pas franchir), on affecte la valeur 0. On construit ensuite notre métrique W à partir de l'imager normalisé I, tel que pour tout pixel ij

$$W_{ij} = \frac{1}{\epsilon + I_{ij}} \tag{8}$$

Cette métrique pénalise donc les passages par les sommets infranchissables. La valeurϵ au dénominateur permet d'éviter les divisions par zéro. On applique ensuite l'algorithme du fast marchinq pour cette métrique, en choisissant comme point initiale l'entrée du labyrinthe. On obtient alors la fonction T représentant la distance à notre sommet initial par rapport à la métrique W (voir figure 1)

L'obtention du chemin nous est alors donné en appliquant la descente du gradient sur la fonction T obtenue, en partant de l'autre extrémité du labyrinthe (celle que l'on n'a pas utilisé comme sommet initial). Le chemin corresponds alors à la suite des points obtenus au cours des itérations successives de la descente (voir figure 2)

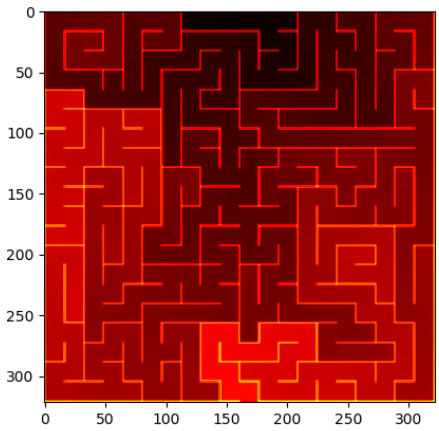


FIGURE 1 – distance au sommet initial (en haut au centre)

3.2 2 fois plus de fast marching !

Le chemin obtenu est correcte, néanmoins il a tendance à "coller" les bords. Cela n'est pas très beau visuellement, et, pour certaines applications cela peut être problématique. Par exemple, dans le cas de la planification du chemin d'un robot, on peut souhaiter que notre robot ne passe pas trop près des bords. Pour remédier à ce problème, une solution est d'inclure dans la métrique W une pénalité pour les passages près des bords. Pour cela il faut re en mesure de mesurer la distance d'un point du labyrinthe à son bord le plus proche. L'algorithme du fast-marching nous permet justement de réaliser cette opération. Pour cela on effectue l'algorithme en choisissant comme sommets initiaux l'ensemble des bords du labyrinthe. On appellera la solution obtenue S (speed), dans le sens où, plus la distance au bord sera importante, plus l'on pourra se déplacer au rapidement (figure 3)

Une fois la fonction S calculé, nous réapliquons l'algorithme du fast-marching, en ajoutant à notre métrique W un facteur prenant en compte la distance au bord :

$$W_{ij} = \frac{1}{\epsilon + I_{ij}} + \alpha * \frac{1}{\epsilon + S_i j} \tag{9}$$

Le paramètre α nous permet de choisir l'importance de la pénalisation des passages près des murs. Plus le paramètre α sera important, plus le chemin trouvé sera éloigné des murs. Si on prend $\alpha = 0$, alors on retrouve le chemin obtenu avec la première méthode (figure 4).

3.3 Autres labyrinthes

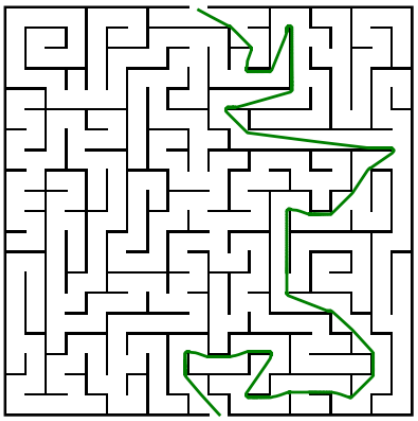


FIGURE 2 – Solution du labyrinthe obtenue avec la descente du gradient

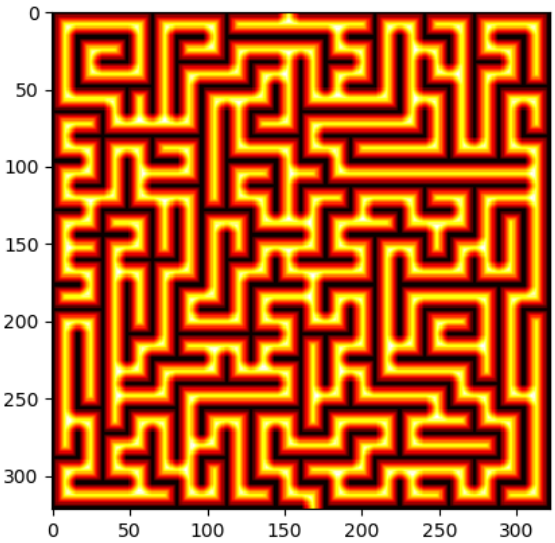
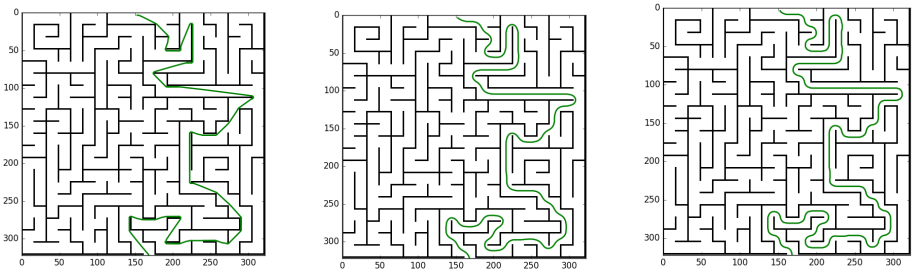
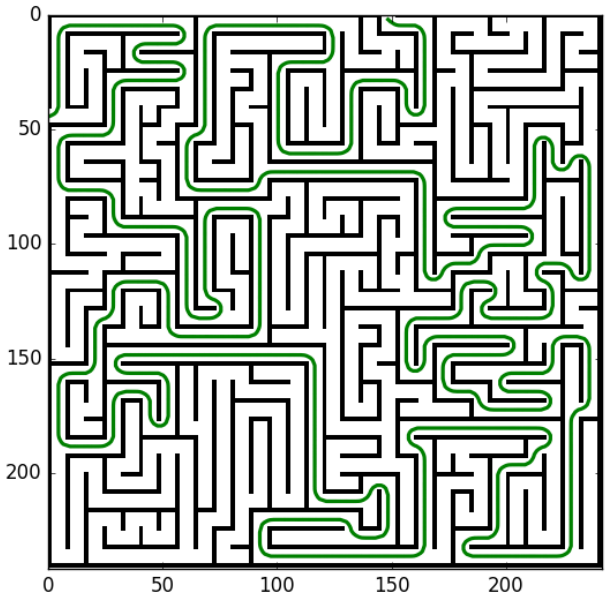


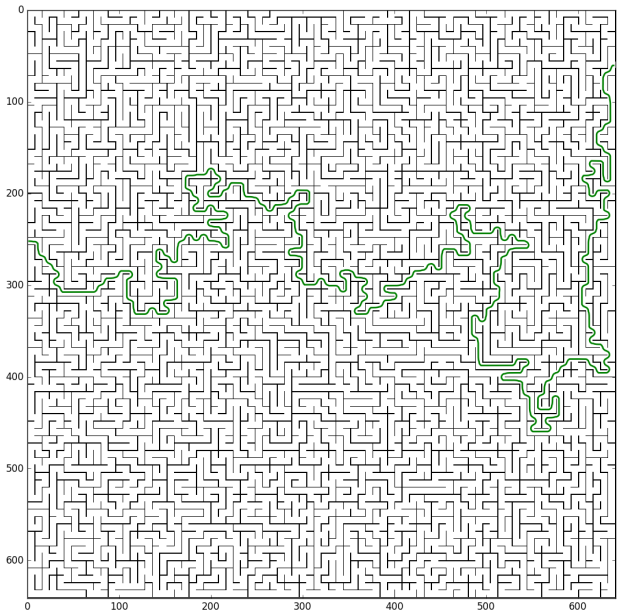
FIGURE 3 – Distance aux bords, en initialisant l'algorithme avec les sommets correspondants aux bords

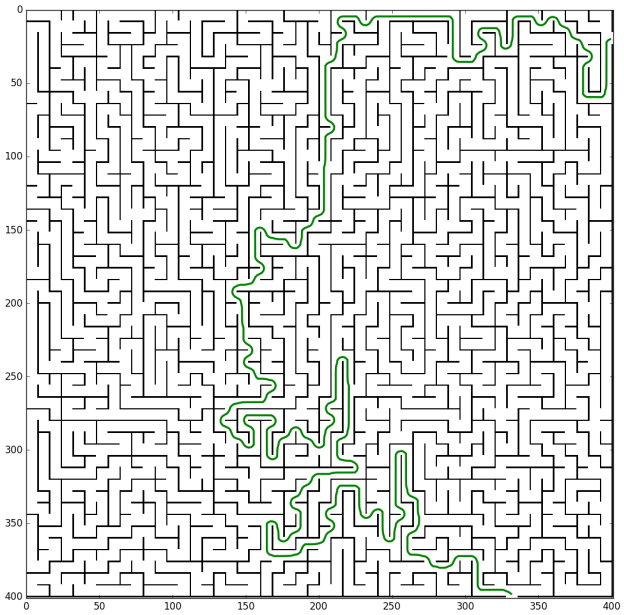


(a) $\alpha = 0$ (b) $\alpha = 0.5$ (c) $\alpha = 10$

FIGURE 4 – Influence du choix du paramètre alpha sur le chemin obtenu




0
100
200
300
400
500
600
0
100
200
300
400
500
600


0
50
100
150
200
250
300
350
400
0
50
100
150
200
250
300
350
400