Résultat de la phase d'analyse

Mode a : Pour la lecture d'une grille, la fonction « read_grid » a été créée par principe d'abstraction. Pour la résolution de cette grille, la fonction « resoudre_passage » fait le travail par principe de réutilisation. Par l'avancement du projet, cette méthode a été modifiée pour retourner un booléen indiquant si la grille à une traversée ou non. Cela évite de faire une vérification supplémentaire.

Cette fonction appelle la fonction « contruire_passage » pour chaque case libre de la première ligne. Cette dernière va par la suite vérifier à son tour les cases adjacentes par appel de cette même méthode. C'est donc une fonction récursive. Elle a aussi été modifiée par la suite par l'ajout d'un paramètre booléen par référence qui indique s'il y a une traversée. La même variable est passée à chaque appel récursif et est mis à vrai si la ligne sur lequel est la cellule indiquée est la dernière. Cette méthode permet d'éviter, lors de la vérification de traversée, de parcourir les n cellules de la dernière ligne, mais ajoute aussi du travail à la fonction récursive qui risque d'être appelée un grand nombre de fois. Pour aller plus loin (pour les modes b et c) on peut quitter la fonction dès que cette valeur est passée à vrai car on n'a pas besoin de savoir s'il y a d'autres traversées, on veut juste savoir s'il y en a au moins une.

Finalement, la dernière fonction pour le mode a créé est « output_grid_pbm » qui permet de sortie en console la grille passage sous forme d'image de format pbm.

Mode b : Pour la génération de grilles, la fonction « generate_grid » est créée par principe de réutilisation. Pour la résolution d'un certain nombre de grilles générées automatiquement, c'est la fonction « resourde_nbt_grilles » qui est réalisée. Elle va tout simplement générer une grille et la résoudre avec la fonction « resoudre_passage » 'nbt' fois. Le résultat retourné est le nombre de grilles qui ont une traversée.

Mode c : 3.3.1 : La fonction « echantillonnage_constant » est la fonction utilisée pour l'échantillonnage avec un pas constant de 1/(NBP-1). NBP-1 est utilisé plutôt que NBP-2 car cela semble plus correct, pour un NBP de 3 par exemple, on voudra un pas de 0.5=1/2 = 1/(3-1). Cette méthode va échantillonner la courbe p'=f(p) en un nombre défini de points 'nbp' par un appel de la fonction « resoudre_nbt_grilles » pour p de 0 à 1 avec le pas constant défini plus haut.

3.3.2 : Cette fois-ci, on veut un pas adaptif. C'est la méthode « echantillonnage_adaptif » par principe d'abstraction qui s'en charge. Une structure est créée pour faire un tableau de paires de points (p, p'). Le tableau est initialisé avec les valeurs évidentes (0,0) et (1,1). Par la suite, on appelle la fonction « resoudre_dichotomique » qui est récursive et qui va faire le reste du travail. On l'appelle avec les valeurs min=min'=0 et max=max' = 0. Puis on prends le point milieu milieu=(min+max)/2 et on calcule milieu' pour ce point par la méthode « resoudre_nbt_grilles ». On stocke ce résultat (milieu, milieu') dans la liste de paires de points. On fait le calcul de l'erreur absolue par erreur=milieu' – (min'+max')/2 puis on vérifie les conditions pour continuer. Si elles sont valides, on appelle à nouveau cette fonction « resoudre_dichotomique » mais cette fois avec min=milieu et min'=milieu' si l'erreur est négative OU max=milieu et max'=milieu' si l'erreur est positive. Par la suite, la liste n'est pas dans l'ordre croissant. On va donc la trier avec la fonction « std::sort » de base, et comme fonction de comparaison, la fonction « comparer_points » qui fait une comparaison de sorte que la liste soit croissante. Pour finir, on affiche les paires de points dans la console pour en faire un graphique.

Pseudocode de l'algorithme de dichotomie - section 3.3.2

echantillonnage_adaptif	**resoudre_dichotomique**
Entrée : n : Taille de la matrice nbt : Nombre de grilles à générer	Entrée : min : Point minimum max : Point maximum min' : p' pour p = min max' : p' pour p = max

// Liste des paires de points (p,p')
points <- {(0,0), (1,1)}
 	
resoudre_dichotomique(0, 1, 0, 1, points, n, nbt)
trier la liste des points par ordre croissant de p
 	
Pour i de 1 à points.size()
Afficher : paire « p p' » à l'emplacement i
	p <- (min+max)/2
	p' <- resoudre_nbt_grilles(n, p, nbt) / nbt
	error <- p' – (min' + max')/2
	Ajouter la paire (p,p') a la fin de la liste points
	Si (max-min) < min_delta_p
	Sortir
	si error < -max_error
	resoudre_dichotomique(p, max,
	pPrime,maxPrime,points, n, nbt)
	sinon si error > max_error
	resoudre_dichotomique(min, p, minPrime,
	pPrime, points, n, nbt)

Réponses 3.3.1 et 3.3.2

	3.3.1 - Pas constant	**3.3.2 - Pas adaptif**
Temps	24.992	1.254
Echantillons	102	7
p ; f(p)=0.5	0.5923	0.5936

On peut voir que pour le point 3.3.2, on prend 20x moins de temps, 14x moins d'échantillons et on arrive très proche du même résultat.
On ne sait pas exactement quel est le plus proche de la vraie valeur car aucun des deux n'a calculé exactement le point précis, mais il est estimé par le graphe de la fonction p'=f(p).
On ne peut donc pas dire que la méthode avec pas adaptif est plus précise.
Par contre elle est bien plus rapide et utilise aussi beaucoup moins de mémoire.
Cependant, la zone recherchée est plus ou moins précise mais à d'autres zones, la précision diminue.
Ce n'est donc pas une solution parfaite pour tous les problèmes