Speleo

Existe-t-il un passage reliant les côtés Nord et Sud ?

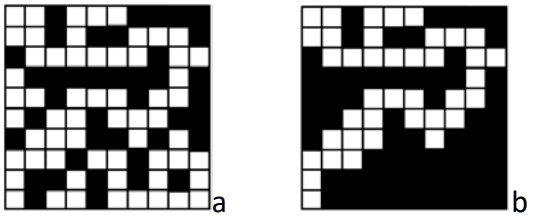

Figure 1: (a) vue en coupe du terrain montrant les passages libres en blanc ;
(b) résultat de l'analyse montrant les cellules de passage en partant depuis le côté Nord (première ligne de la grille)

1. Introduction

Ce projet explore la généralisation d'un problème de recherche d'un chemin à travers un terrain plus ou moins dense, d'où le titre du projet. Il exploite une abstraction du terrain (Fig 1a) sous forme d'une grille de cellules qui sont soit **libres** (blanches) soit **obstruées** (noires). Le problème est traité à trois niveaux de généralité résumés dans la Table 1:

Niveau élémentaire (section 3.1): le problème consiste à déterminer l'ensemble des cellules **libres** que l'on peut atteindre depuis une cellule **libre** de la *première ligne de la grille* comme visible en blanc sur la Fig 1b. Ces cellules sont appelées des cellules de **passage** avec un booléen à vrai (à faux sinon). Dans cette version élémentaire on fournit la grille du terrain au programme et celui-ci produit la grille des cellules de **passage** sous forme d'une image **pbm.**

Niveau intermédiaire (section 3.2): on donne seulement le nombre de cellules **n** d'un côté de la grille, la probabilité **p** *qu'une cellule soit libre* et un nombre **nbt** de terrains **n x n** que le programme doit générer automatiquement en utilisant **p**. Pour chaque terrain il détermine s'il existe une ***traversée*** entre les côtés Nord et Sud, c'est-à-dire s'il existe *au moins une cellule de passage sur la dernière ligne de la grille* (comme pour la Fig 1). Le résultat est la ***probabilité de traversée* p'** égale au rapport du nombre de cas ayant une traversée divisé par **nbt**.

Niveau général (section 3.3): on donne seulement **n** et **nbt** ; ici on cherche à visualiser comment **p'** évolue en fonction de **p** (toujours pour **nbt** cas à chaque fois). Le résultat est l'affichage de la suite des paires **p p'** (une par ligne) que nous pourrons visualiser avec le logiciel **gnuplot**.

Niveau	1ière donnée	Données suivantes	Résultat affiché **dans le terminal**
élémentaire	***a***	**n** format **pbm** : n lignes de n valeurs **0** (libre) ou **1** (obstruée)	Image binaire en format **pbm** avec 0 (passage) et 1 (inaccessible) à visualiser avec imageMagick.
intermédiaire	***b***	**n p nbt**	**p'**
général	***c***	**n nbt**	Suite des paires **p p'** (une par ligne) à visualiser avec gnuplot

Table 1: vue d'ensemble des trois niveaux de généralisation du problème avec les données attendues en entrées et les sorties à afficher dans le terminal selon le niveau de généralité.

2. Méthode du travail

2.1 Mise en oeuvre des grands principes

Le projet représente une quantité de travail bien supérieure aux exercices proposés dans les séries. La mise en oeuvre des principes ***d'abstraction*** et de ***ré-utilisation*** est un élément central dans la stratégie de résolution. En effet, la distinction des trois niveaux de généralité permet d'identifier des sous-problèmes (*abstraction*) dont la solution sous forme de fonctions devra être utilisée pour résoudre les niveaux de généralité supérieurs. Il est possible mais pas obligatoire qu'une fonction de bas niveau puisse même être *ré-utilisée* à plusieurs endroits dans le code.

2.2 Vérification précoce par les tests (*scaffolding*)

Il est important de réfléchir au *but* de chaque fonction pour déterminer sa « mission » : sur quelles données travaille-t-elle ? quel(s) résultat(s) fournit-elle ? Au-delà de cette réflexion, il faut ensuite *vérifier* que chaque fonction réalise bien son but avec un solide éventail de tests pour lesquels on connait les résultats attendus. Grâce à cette méthode un sous-problème est résolu une fois pour toute dès le niveau le plus élémentaire.

L'approche de vérification par des tests implique d'*écrire du code supplémentaire dédié à ces tests* AVANT de commencer à traiter les niveaux supérieurs du projet. Vous serez ainsi amené à écrire un certain nombre de petits programmes dédiés à ces tests. Cette méthode de travail est appelée *scaffolding* (échafaudage) et cherche à rendre votre code robuste à la grande variété des cas possibles (à défaut de disposer d'une méthode qui garantirait que votre code est toujours correct).

En effet nous disposons d'outils tels que l'éditeur et le compilateur pour détecter certaines erreurs mais ils ne permettent pas de toutes les trouver. Nous savons déjà qu'il est recommandé de *recompiler très fréquemment* afin réduire au minimum le temps de recherche des ***erreurs syntaxiques*** (fautes d'orthographe/grammaire du langage C++). Le raisonnement est qu'une erreur se trouve dans la portion de code écrite depuis la dernière compilation avec succès, ce qui permet de réduire à très peu de lignes de codes l'espace de recherche de cette erreur.

La méthode de l'échafaudage (scaffolding) est destinée à trouver les ***erreurs sémantiques*** que le compilateur ne trouve pas car le programme respecte la syntaxe du langage ; elles vont produire un comportement incorrect du programme. Le point essentiel dans cette méthode est qu'il faut tester *chaque fonction* ***individuellement*** pour *vérifier* qu'elle produit le résultat attendu AVANT de l'utiliser ailleurs. Là aussi cette approche vous rend plus efficace dans le développement de code car l'erreur sémantique se trouve normalement dans la dernière fonction en cours de test car celles qui sont appelées ont déjà été validées.

2.3 Bottom-up ou top-down ?

La méthode présentée dans la section précédente suggère de vérifier d'abord le niveau élémentaire avant les niveaux supérieurs ainsi que les fonctions de bas-niveaux avant celles qui les utilisent. C'est l'approche *bottom-up*. C'est ce que nous recommandons pour ce projet en ce qui concerne les trois niveaux de généralité. Pour ce qui est de la mise au point des fonctions, il peut être légitime de vouloir tester une fonction f() qui appelle une fonction g() avant que g() soit écrite en détail. C'est une approche *top-down*.

L'approche *top-down* est possible si le résultat de g() est facile à définir, par exemple si elle renvoie true ou false, car la seule chose utilisée par f() est ce résultat. On peut ainsi vérifier f() à l'aide d'une *forme minimale de la fonction* g() que l'on appelle un ***stub***. Cette forme minimale respecte le prototype prévu pour g() en terme de paramètres et de type de la valeur de retour ; par contre sa définition peut se restreindre à un corps vide si aucune valeur n'est retournée ou très peu de chose, par exemple une instruction return de *true* ou *false* si g() est supposée renvoyer un booléen.

2.4 Redirection des entrées-sorties pour automatiser les tests d'un programme

On utilisera **exclusivement** l'entrée standard (*clavier*) pour fournir les données et la sortie standard (*terminal*) pour afficher les résultats. Il ne faut pas écrire de code de lecture-écriture de fichier dans ce projet.

A la place, le mécanisme de *redirection* des entrées-sorties sera vu en TP (semaine6) et permettra de travailler avec des fichiers de test *sans avoir à changer une seule ligne de code*. En effet les données d'un test peuvent être éditées avec l'éditeur de programme et mémorisées dans un fichier de test. Ensuite il suffit de préciser le nom du fichier de test sur la ligne de commande qui lance l'exécution du programme et celui-ci ira chercher les données dans le fichier au lieu de les demander au clavier. C'est bien pratique pour le niveau élémentaire du projet pour lequel il faut donner les n^2 valeurs 0 et 1 sans se tromper…

Cette méthode de test est recommandée surtout pour relancer de manière très efficace un ensemble de tests et vérifier que votre programme est toujours correct. C'est aussi en redirigeant des fichiers de test sur l'entrée standard que nous testerons votre programme ; nous redirigerons le résultat dans un fichier que nous comparerons avec le résultat obtenu avec le programme de démo. Votre programme devra veiller à ne rien afficher de plus que les résultats demandés dans les formats précisés en section 3.

3. Spécifications détaillées

Cette section approfondit les éléments fournis en section 1 en cherchant à identifier des structures de données et des fonctions pouvant être ré-utilisées aux trois niveaux.

3.1 Niveau élémentaire de recherche d'un chemin du coté Nord jusqu'au coté Sud

3.1.1 Représentation du terrain et du passage **pour les entrées-sorties**:

Avant d'aborder la résolution de la recherche de chemin, il faut décider de la représentation du terrain. En section1 celui-ci est modélisé comme une grille de n x n cellules pouvant prendre seulement deux états possibles : **libre** ou **obstrué**. Un booléen suffit donc pour représenter chaque cellule. Dans ce projet nous voulons bénéficier des outils permettant de manipuler le format d'image **pbm** (*portable bitmap file format*). C'est pourquoi, en entrée l'état **libre** est fourni au programme avec la valeur **0** tandis que l'état **obstrué** est fourni au programme avec la valeur **1**. Ce choix nous permet de visualiser le terrain avec le logiciel imageMagick (complément série5).

Pour bénéficier du même outil de visualisation la représentation des cellules permettant le passage suit la même convention : l'état de **passage** affiche une valeur **0** en sortie et l'état d'une cellule **inaccessible** produit la valeur **1** en sortie.

Nous avons vu en cours la correspondance entre les valeurs 0 et 1 et les booléens associés *false* et *true*. En appliquant cette convention cela revient à dire que l'entrée du programme est la carte des cellules **obstruées** et que la sortie affiche les cellules **inaccessibles**.

3.1.2 Représentation **interne** du terrain et du passage :
Cela dit, la recherche du chemin du Nord au Sud s'exprime plus intuitivement en s'appuyant sur l'ensemble des cellules **libre** pour obtenir l'ensemble des cellules de **passage**. C'est pourquoi, l'algorithme que nous allons discuter plus loin s'appuie sur une grille des cellules qui sont dans l'état ***true*** pour une cellule **libre** ; cet algorithme produira une grille de cellules qui seront dans l'état ***true*** pour une cellule de **passage**.

Pour représenter la grille nous avons besoin de la notion de *tableau* (semaine 7-8) et plus particulièrement de la notion de **vector** car la taille **n** n'est pas connue au moment de l'écriture du programme. Nous montrons la déclaration de la grille **libre** pour faciliter la suite de la discussion sur la recherche de chemin. La grille **passage** est du même type.

```
vector<vector<bool>> libre(n,vector<bool>(n)); // init à false
```

Dans la suite de la donnée nous respectons la convention des tableaux : l'indice de gauche est celui indiquant la ligne et celui de droite indique la colonne: `libre[i][j]`.

3.1.3 Initialisation (Table 1)
Après avoir indiqué le niveau élémentaire avec le caractère '**a**' et le nombre de lignes avec l'entier strictement positif **n**, le programme devra lire les $\mathbf{n^2}$ valeurs 0 ou 1 pour initialiser la grille **libre**. Nous demandons de respecter le nombre maximum de caractères indiqué par le format **pbm** (section suivante). La grille **passage** doit être initialisée à ***false*** ; cela est fait automatiquement avec la déclaration fournie ci-dessus.

3.1.4 Le format **pbm** pour la sortie de la grille de passage
Le format **pbm**[1] est un format pratique pour ce projet car il est facile à afficher dans le terminal. De plus ce texte peut être redirigé vers un fichier avec l'extension **.pbm** qui est ensuite manipulable avec le logiciel ImageMagick (série semaine 5). La première ligne du format contient la chaîne de caractères « **P1** » pour indiquer qu'il s'agit d'un format **pbm**. La ligne suivante donne deux nombres entiers : le nombre de colonnes suivi par le nombre de lignes. Les lignes suivantes donnent les valeurs 0 (affiché en blanc) et 1 (affiché en noir). Voici l'exemple d'une image de 3 lignes et 3 colonnes avec une diagonale noire qui descend vers la droite :

```
P1
3 3
1 0 0
0 1 0
0 0 1
```

La seule limitation de ce format est de ne pas avoir plus de 70 caractères par ligne affichée ; si **n** est plus grand que 70, on peut quand même produire un affichage compatible avec le format **pbm** : il suffit d'ajouter un saut de ligne dès qu'on arrive à la limite de 70 caractères.

[1] source : http://fr.wikipedia.org/wiki/Portable_pixmap

3.1.5 ACTIONS 1 : tests de l'initialisation + stub du calcul + affichage du résultat

Il est essentiel de s'assurer de la bonne lecture des paramètres du niveau élémentaire pour construire la grille des cellules libres avant d'aller plus loin dans la mise au point du programme.

Concevez une première version du programme qui gère l'initialisation des grilles **libre** et **passage**. Ensuite une première fonction *stub* du **calcul** de la grille **passage** peut consister à copier la grille **libre** dans la grille **passage**. Enfin ce premier programme affiche le résultat attendu à partir des valeurs de **passage**. Dans ce premier exercice d'abstraction des actions à mettre au point, pensez à faire en sorte que les fonctions proposées soient ré-utilisables pour les niveaux supérieurs du problème.

Testez cette première version en tirant parti de la redirection des entrée-sortie pour couvrir une grande variété de cas.

3.1.6 Calcul de la grille de passage

Une fois l'étape précédente validée nous pouvons nous concentrer sur le cœur du problème : déterminer toutes les cellules de **passage** en commençant par une cellule qui se trouve du coté Nord du terrain, c'est-à-dire sur la première ligne de la grille **libre**. La figure suivante montre qu'il est important d'examiner toutes les cellules de la première ligne comme point de départ car certaines (Fig2 gauche) voire toutes (Fig2 droite) peuvent conduire à des impasses. Dans ce dernier cas il y a échec de la traversée car aucun chemin n'arrive jusqu'à la dernière ligne du terrain.

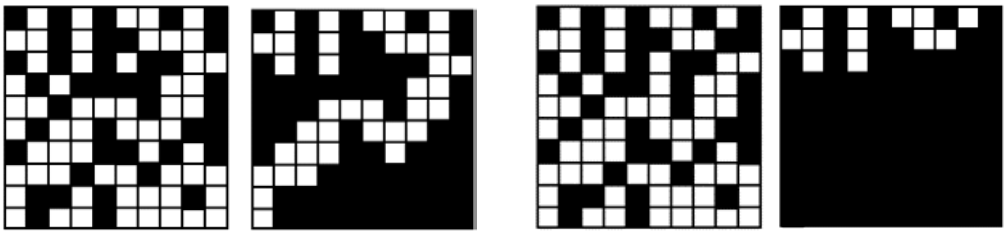

Fig2 : variante de terrain illustrant la nécessité d'explorer toutes les cellules de la ligne de départ (gauche) ; parfois il n'existe pas de chemin (droite)

3.1.6.1 Règle de mise à jour et définition du *voisinage* d'une cellule: A l'initialisation il n'y a aucune cellule de **passage**, c'est-à-dire que toutes les cellules de la grille **passage** sont dans l'état ***false***. Chaque cellule **libre** de la *première ligne* du terrain fait passer la cellule de **passage** correspondante à ***true***. Pour pouvoir activer à ***true*** une autre cellule de **passage** celle-ci doit être **libre** dans la grille du terrain et avoir un *contact sur au moins un coté complet* avec une cellule de **passage** qui est déjà dans l'état ***true***. Le *voisinage* d'une cellule est donc limité à un maximum de **4** cellules.

3.1.6.2 Ebauche de solution : à partir de chaque cellule **libre** trouvée sur la première ligne, on active la cellule de **passage** correspondante puis on effectue une recherche selon l'approche *en profondeur d'abord* : on fait passer chaque cellule *voisine* de **passage** dans l'état ***true*** si la cellule correspondante est **libre**. Cette mise à jour se propage de proche en proche aux cellules voisines libres comme illustré sur la Fig 3 (cas particulier où une seule voisine est libre). La mise à jour s'arrête lorsqu'il n'y a plus de nouvelle cellule **libre** accessible depuis une cellule de **passage**.

L'exemple de la figure 3 est simple :

- Traitement de toutes les cellules de la première ligne de la grille libre (étapes 1, 2, 6) : A l'étape 1 la cellule encadrée en rouge étant obstruée, rien n'est activé dans la grille de passage. A l'étape 2 la cellule étant libre la *recherche en profondeur* va être effectuée à partir de ce point de départ. Lorsqu'elle est terminée qu'on traite la dernière cellule de la première ligne (étape 6 / rien à faire car la cellule encadrée en rouge obstruée).

- Recherche en profondeur (étapes de 2 à 5) : à l'étape 2 la cellule du terrain encadrée en rouge étant libre, on active la cellule correspondante de la grille de passage à *true*. Ensuite la recherche se poursuit dans les 4 directions de la grille libre (au-dessus, au-dessous, à droite, à gauche) mais seulement si les conditions suivantes sont remplies sur chaque cellule voisine :
  - elle se trouve sur le terrain
  - elle est libre et n'a pas encore été visitée (**passage** à *false*)

Dans le cas de la Fig3 une seule des 4 cellules voisines est valide pour les étapes de 2 à 4 :

- étape 2 : Activation de la cellule correspondante de la grille **passage** à *true*. Ensuite la cellule au-dessus est en-dehors du terrain, les cellules droite et gauche ne sont pas libres. Il reste seulement la cellule du dessous avec laquelle on poursuit la recherche.

- Étape 3 : Activation de la cellule correspondante de la grille **passage** à *true*. Ensuite, la cellule au-dessus est libre mais a déjà été visitée (son état dans la grille **passage** est à *true*), les cellules à gauche et dessous ne sont pas libres. Il reste seulement la cellule à droite avec laquelle on poursuit la recherche.

- Étape 4 : Activation de la cellule correspondante de la grille **passage** à *true.* Ensuite, la cellule au-dessus n'est pas libre, celle de droite est en dehors du terrain, celle de gauche est libre mais déjà visitée. Il reste la cellule du dessous avec laquelle on poursuit la recherche.

- Étape 5 : Activation de la cellule correspondante de la grille **passage** à *true.* Ensuite, les cellules de droite et dessous sont en dehors du terrain, celle de gauche n'est pas libre et celle du dessus est libre mais déjà visitée. Aucune cellule n'est valide ; la recherche en profondeur se termine pour le point de départ de l'étape 2.
-

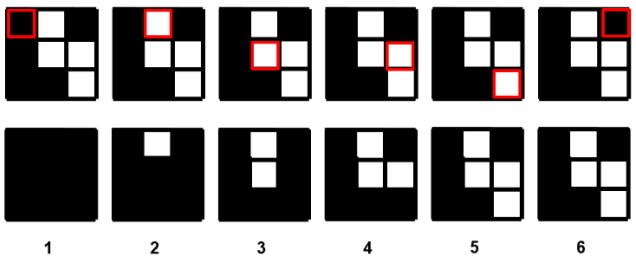

Fig 3 : recherche en *profondeur d'abord* dans la grille **libre** (ligne du haut) pour construire la grille **passage** (ligne du bas): pour chaque cellule **libre** de la première ligne on construit l'ensemble des cellules de **passage** en explorant les cellules voisines.

3.1.6.3 Stratégie récursive pour traiter une grille quelconque :

L'exemple de la Fig 3 est simplifié dans le sens où une seule cellule voisine est valide parmi les 4 candidates mais comment pouvons-nous envisager une solution algorithmique pour le cas général où plus d'une voisine est valide ? L'approche retenue effectue la recherche *en profondeur d'abord*, ce qui veut dire qu'on doit traiter la première cellule candidate et, si elle est valide, on doit traiter immédiatement ses 4 cellules voisines (à elle) comme illustré sur la figure 4 pour le traitement de la deuxième cellule de la première ligne.

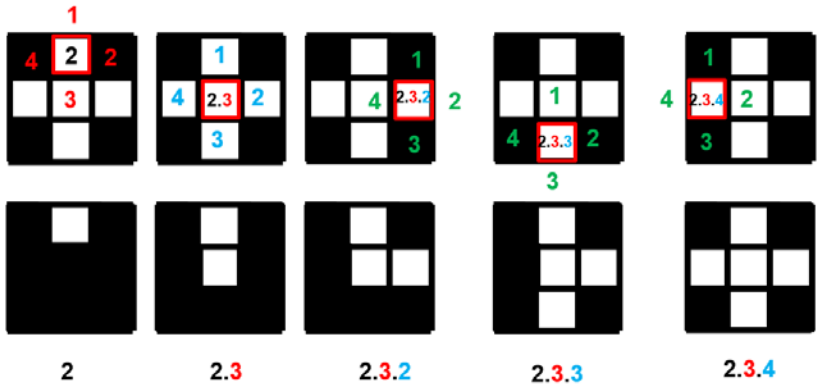

Fig 4 : détails de l'étape 2 avec une approche récursive permettant de traiter une grille quelconque ; les cellules voisines sont numérotées de 1 à 4 et sont traitées dans cet ordre : voisine 1 puis 2 puis 3 puis 4. La couleur rouge désigne le premier niveau (celui d'une cellule sur la première ligne). Les voisines 2.1 et 2.2 n'étant pas valide, leur traitement s'arrête tout de suite. La voisine 2.3 étant valide, on doit traiter ses voisines illustrées en bleu (2ième niveau de profondeur). La première valide est 2.3.2 ; ses voisines sont illustrées en vert (3ième niveau de profondeur) ; aucune n'est valide. On traite donc ensuite 2.3.3 ; aucune de ses voisines (en vert) n'est valide. Vient ensuite le tour de 2.3.4 ; aucune de ses voisines (en vert) n'est valide et cela termine la recherche en profondeur pour la cellule 2 de la première ligne.

Stratégie *récursive*: le problème de construction de la grille **passage** est idéal pour exprimer une solution récursive car le traitement à effectuer sur les cellules voisines est le même que celui à effectuer sur la cellule de départ. Nous connaissons également les critères d'arrêt qui ***empêchent*** d'effectuer un appel récursif sur une cellule voisine :

- si la cellule candidate n'est pas sur le terrain ; ex Fig4 : 2.1, 2.3.2.2, 2.3.3.3, 2.3.4.4
- si elle n'est pas libre ; ex Fig4 : 2.2, 2.4, 2.3.2.1, 2.3.2.3, …
- si elle est libre mais a déjà été visitée (**passage** est à *true*) ; ex Fig 4 : 2.3.1, 2.3.2.4,…

Nous vous demandons de mettre en œuvre cette stratégie récursive dans la fonction **`construire_passage`** qui est appelée pour chaque cellule de la première ligne comme montré dans le pseudocode suivant :

Entrées : indice de ligne, indice de colonne, grille **libre**, grille **passage**
L'entrée **passage** doit pouvoir être modifiée

Pour chaque cellule **j** de la première ligne de la grille **libre**
Si libre(1, j) est *true*
`construire_passage`(1, j, **libre**, **passage**)

La première action de **`construire_passage`** est de valider la cellule : **passage**(i,j) <- true
Ensuite on effectue les appels récursifs sur les cellules voisines si les conditions d'arrêt ne sont pas activées.

3.1.7 ACTIONS 2 : intégrer le test de la stratégie récursive

Vous pouvez maintenant remplacer la fonction stub de **calcul** par la boucle traitement de la première ligne et la compléter avec la fonction récursive **construire_passage**.
Procéder aux tests les plus simples possibles au début, par exemple quand **n** vaut 1, car vous risquez d'avoir des bugs dès cette configuration. Ensuite continuer avec des grilles simples pour lesquelles vous pouvez prédire avec papier-crayon la suite des cellules qui seront traitées. Enfin vérifier l'affichage pour les cas des Fig 1,2,3,4 et d'autres cas plus ambitieux en taille de grille.

Si vous ne l'avez pas fait pour l'étape ACTION1 il est ici plus qu'utile d'ajouter du code temporaire (encore du *scaffolding*) qui va afficher dans le terminal les informations sur la progression de l'algorithme, e.g. les indices i et j de la cellule en cours de traitement etc. Ce code devra ensuite être retiré pour le rendu.

A ce stade vous pourrez considérer que le niveau élémentaire est finalisé. Si votre décomposition du programme en fonction est pertinente les deux autres niveaux vont s'appuyer sur le travail déjà effectué et seront nettement plus rapide à mettre au point.

3.2 Niveau intermédiaire

Au niveau intermédiaire on s'intéresse à la propriété de ***traversée*** d'un terrain. On dit que la *traversée* est *true* pour un terrain donné s'il existe au moins une cellule à *true* sur la dernière ligne de la grille de **passage**. C'est le cas pour la Fig1b, Fig2 gauche et Fig 3 et 4 mais pas pour Fig2 droite.

A ce niveau on veut obtenir une information plus générale que ce simple booléen : on veut la ***probabilité de traversé*** pour la classe des terrains ayant une même densité **p** de cellules *libres*. Comme il serait très fastidieux de construire ces terrains nous-même, nous voulons automatiser cette tâche. C'est pourquoi, au lancement du programme, on fournit le caractère **'b'** qui indique le niveau intermédiaire, suivi par l'entier strictement positif **n**, une probabilité **p** comprise dans **[0. , 1.]** et l'entier strictement positif **nbt** indiquant le nombre de terrains qui seront automatiquement produits par le programme.

Le résultat de l'exécution est l'affichage dans le terminal de la ***probabilité de traversée*** **p'** égale au rapport du *nombre de cas ayant une* ***traversée*** divisé par **nbt**.

3.2.1 Construction d'un terrain à partir de la probabilité **p**

La probabilité **p** doit être appliquée à chaque cellule pour déterminer si elle est *libre* ou pas. La production de hasard est une fonctionnalité qu'il faut déléguer à C++11 car ce langage fournit des outils de bonne qualité. Voici le code C++ à adapter :

```
double p;
… lecture et vérification de la valeur de p …
bernoulli_distribution b(p); //booléen true avec une probabilité p
default_random_engine  e;

… ensuite boucle d'initialisation d'un terrain…
      libre[i][j]= b(e);
```

Attention : la déclaration du générateur de hasard **e** doit être faite en dehors de la boucle d'initialisation de la grille du terrain.

3.2.2 ACTION3 : test de génération et terrains et calcul de la probabilité de traversée

Ecrire d'abord du code supplémentaire (*scaffolding* une fois de plus) pour visualiser un terrain **n x n** produit avec une probabilité **p** de 0.5 ; il suffit de faire afficher le terrain ligne par ligne dans le terminal et d'inspecter visuellement. Le hasard obtenu correspond-il à vos attentes ?

Poursuivre les tests en variant **n** et **p**.

Pour une grande valeur de **n** on peut afficher en format **pbm** (section 3.1.4), rediriger la sortie puis visualiser avec ImageMagick (semaine 5 TP).

Après cette première phase de tests vous pouvez écrire la fonction qui détermine s'il y a traversée pour une grille de **passage** et intégrer la production de terrain pour obtenir la probabilité traversée recherchée. On cherchera à concevoir une fonction ré-utilisable pour le niveau général.

3.3 Niveau général

Au niveau général on se pose la question de savoir s'il existe une *valeur de seuil* de **p** caractérisant une soudaine augmentation de **p'**. En effet il est facile d'imaginer que pour **p** faible, **p'** vaut toujours 0.0 tandis que pour **p** proche de 1.0, **p'** vaut toujours 1.0 mais entre ces deux valeurs limites il n'existe pas de solution mathématique connue qui nous donnerait la valeur de **p'** en fonction de **p**. Le but de ce niveau général est de calculer **p'** pour un nombre suffisamment grand de valeurs de **p** (à chaque fois pour **nbt** terrains) pour disposer d'une approximation raisonnable de la fonction **p'= f(p).**

C'est pourquoi, au lancement du programme, on fournit le caractère **'c'** qui indique le niveau général, suivi par les entiers strictement positifs **n** et **nbt** indiquant le nombre de terrains qui seront automatiquement produits par le programme pour évaluer chaque échantillon de valeur de **p**. Le résultat de l'exécution est l'affichage dans le terminal des couples **p p'** (un couple par ligne du terminal) dans l'ordre croissant des valeurs de p.

La redirection de la sortie vers un fichier **out.dat** permet ensuite de demander le dessin de **p'=f(p)** avec l'outil de dessin **gnuplot**. Pour cela il faut d'abord lancer **gnuplot** sur la ligne de commande du terminal. Voici la commande à exécuter sur la ligne de commande de **gnuplot** :

```
plot "out.dat" using 1:2 with lines
```

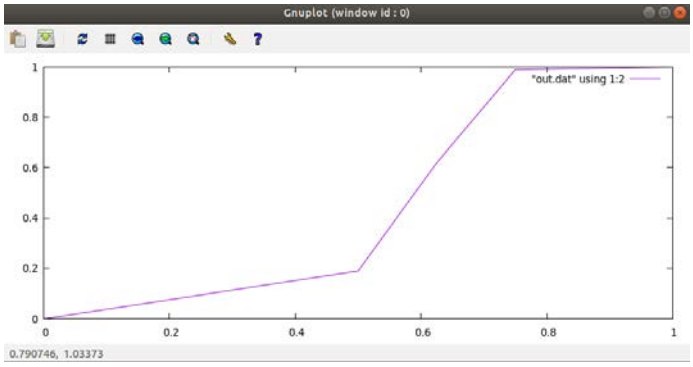

Fig 5: exemple de courbe possible avec gnuplot pour n=10 et nbt=100 (méthode 3.3.2)

3.3.1 ACTION4 : produire p'=f(p) pour un nombre d'échantillons NBP de p avec NBP > 2

Pour vous aider à mieux cerner la manière dont la fonction recherchée varie en fonction de **p**, le plus simple est d'écrire une boucle qui fait évoluer **p** entre **0.0** et **1.0** avec un pas constant égal à **1./(NBP-2)** car les valeurs 0 et 1 comptent pour deux échantillons. Ici aussi ce code est à considérer comme du scaffolding car il sert seulement à se familiariser avec **gnuplot** pour visualiser la forme de la fonction recherchée en fonction de **n, nbt**, **NBP**. Cette étape sert à évaluer le temps calcul nécessaire si on veut obtenir une bonne qualité de l'approximation.

Pour le rapport :

- Noter le temps calcul « **user** » fourni par la commande **time** dans le terminal pour l'exécution de votre programme de test **prog_test** pour **n**=100, **nbt**=1000 et **NBP**=102 :

```
time ./prog_test > out.dat
```

- Inclure l'image de p'=f(p) obtenue avec gnuplot. Quelle valeur de **seuil** obtient-on graphiquement avec cette valeur **NBP** ? (pour cela relever la valeur de **p** pour laquelle **p'** vaut 0.5 sur le dessin).

3.3.2 Echantillonnage adaptatif de la variable **p**

Le programme à rendre doit effectuer un échantillonnage adaptatif de la variable **p** pour éviter d'effectuer des calculs qui apportent peu d'information sur la forme de la fonction. Par exemple on sait que la fonction varie peu au voisinage des valeurs limites (0,0) et (1,1) ; on peut se permettre de prendre un grand pas d'échantillonnage **delta_p** dans ces régions. La question qui nous intéresse ici est : comment faire varier ce pas d'échantillonnage **delta_p** quand on sera proche de la région, a priori inconnue, du seuil ?

Nous vous demandons de fournir une approche dichotomique qui part de l'intervalle [0, 1] et *divise* l'intervalle [min, max] si les *deux* conditions suivantes sont vraies (Fig 6):

- L'intervalle selon **p** est suffisamment grand : **(max – min) > MIN_DELTA_P**
- La valeur absolue de l'écart **error** entre la valeur obtenue **p'** et la valeur donnée par la droite reliant **(min, min')** à **(max, max')** est encore trop grande : **¦error¦ > MAX_ERROR**

Avec **MAX_ERROR** valant 10^{-2} et **MIN_DELTA_P** valant 10^{-6}

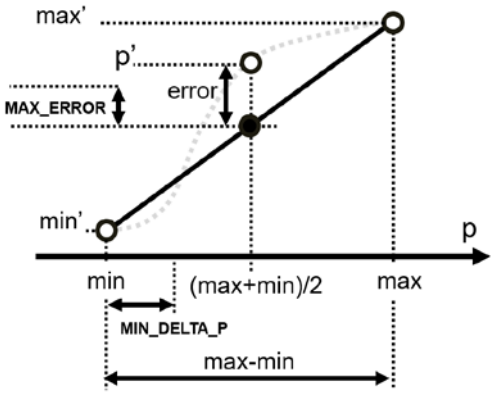

Fig 6 : illustration des critères selon **p** (MIN_DELTA_P) et selon **p'** (MAX_ERROR) ;

dans cette illustration on peut et doit poursuivre la dichotomie car les deux conditions sur **p** et sur **p'** sont vraies.

La dichotomie doit exploiter le signe de **error** car nous savons que le graphe recherché pour **p'=f(p)** est monotone croissant (courbe en trait pointillé gris dans la fig6), c'est-à-dire qu'*il ne coupe qu'une seule fois la droite qui va de (min, min') à (max, max')*.

Donc, si **error** est positif la recherche doit se poursuivre entre min et (max+min)/2,
et si **error** est négatif elle doit se faire entre (min+max)/2 et max.

Cette approche doit être mise en œuvre avec une mémorisation des paires (p p') car l'échantillonnage adaptatif ne permet pas toujours de produire les paires avec p dans l'ordre croissant. On combinera donc la dichotomie avec un le tri ***selon l'ordre croissant de p***.

L'affichage sera fait après la fin de l'algorithme de dichotomie.

3.3.3 ACTION5 : produire p'=f(p) avec l'approche adaptative

Prendre le temps décrire le pseudocode de l'échantillonnage adaptatif (3.3.2) avant de se lancer dans le codage et le test car il sera demandé pour le rapport.

Ne pas hésiter à tester cette fonction pour des valeurs différentes des constantes MAX_ERROR et MIN_DELTA_T pour vérifier que la forme de la fonction p'=f(p) est conforme à vos prédictions.

Pour le rapport :

- Comme pour 3.3.1, noter le temps calcul nécessaire pour **n**=100, **nbt**=1000 et les valeurs de MIN_DELTA_P et MAX_ERROR indiquées plus haut et comparer au temps calcul de 3.3.1.

- combien d'échantillons ont été utilisés selon p ?

- Inclure l'image de p'=f(p) obtenue avec gnuplot. Quelle valeur de seuil obtient-on avec l'approche adaptative ? La précision vous semble-t-elle meilleure ? Si oui pourquoi ?

---

4. Mise en œuvre en langage C++ (*sera compilé avec* `-std=c++11`)

Au sem1 nous demandons que le code soit écrit dans un seul fichier source dont le nom et l'extension doivent être :

`projet.cc`

4.1 Clarté et structuration du code avec des fonctions

La clarté de votre code sera prise en compte pour le rendu. Un examen manuel du listing de votre code source sera effectué par une personne chargée d'évaluer le respect des conventions de programmation utilisées dans ce cours. Par souci d'efficacité seuls les codes indiqués dans nos conventions seront écrits sur votre listing qui vous sera rendu.

4.2 Variables locales ou globales ?

Toutes les ***variables*** ou ***tableaux*** utilisés pour ce projet seront déclarés ***localement*** et transmis en paramètres aux fonctions **seulement** si c'est nécessaire. Aucune exception ne sera admise.

4.2.1 Qu'en est-il des ***constantes*** ?

Voici les règles que nous nous donnons, en conformité avec nos conventions de programmation :

- Si une constante n'est utilisée qu'à l'intérieur d'*une seule fonction*, alors la déclaration d'une variable avec **constexpr** peut être faite dans cette fonction ou globalement.

- Si la constante est utilisée dans *plus d'une fonction*, alors la déclaration d'une variable avec **constexpr** *doit* être faite de manière globale, en début de fichier comme décrit dans les conventions de programmation.
- L'alternative de la déclaration de symboles avec #define est autorisée pour des constantes utilisées dans plusieurs fonctions. Elles sont aussi déclarées en début de fichier comme décrit dans les conventions de programmation.

4.3 Précision sur les nombres à virgule

Tous les nombres à virgule nécessaires pour ce projet tels que **p** et **p'**, doivent utiliser le type **`double`**. Pour l'affichage demandé aux niveaux intermédiaire et général, on utilisera ces instructions C++ :

```
cout << setprecision(6)   // 6 chiffres à droite de la virgule
cout << fixed
```

4.4 Fichiers test :

La donnée identifie plusieurs phases de tests (sections ACTION) pour valider votre programme. Nous vous fournissons quelques fichiers de tests. A vous de compléter avec vos propres fichiers plus élaborés ou juste différents.

4.4.1 Méthode d'évaluation de l'exécution de votre projet

4.4.1.1 Niveau élémentaire

L'affichage sera redirigé vers un fichier de type pbm. De même pour l'affichage du programme de démo. Cependant il peut exister des différences de formatage entre ces 2 fichiers. Pour s'en affranchir nous utiliserons la commande **convert** de imageMagick pour obtenir le même format de fichier et ensuite nous les comparerons avec la commande **diff** de Linux.

4.4.1.2 Niveau intermédiaire

Le nombre nbt détermine la précision de la probabilité affichée, par exemple avec nbt valant 1000 on ne peut pas avoir plus de 3 chiffres significatifs. Le résultat peut cependant varier d'une implémentation à une autre du fait du hasard qui intervient dans la construction automatique des grilles. Nous *calibrerons les variations autorisées* en exécutant le programme de démo plusieurs fois (avec une initialisation différente du hasard d'une exécution à l'autre).

4.4.1.3 Niveau général

Nous écrirons un programme de test qui va prendre le fichier out.dat en entrée et déterminer la valeur p du seuil pour lequel p' vaut 0.5. Comme pour le niveau intermédiaire, nous exécuterons le programme de démo plusieurs fois pour calibrer les variations autorisées.

5. Rendus : Il faudra fournir :

1) **Votre code source imprimé** avec une mise en page portrait, police de caractères utilisée par défaut par **geany**, ou Courrier New de taille 10. Votre objectif est de ne pas avoir de passages à la ligne parasites (wrapping) ; il est autorisé d'avoir 87 caractères au maximum par ligne et 40 lignes par fonction (conventions de programmation). Vous devez vérifier votre mise en page avec la commande Print Preview de geany.
2) **Votre code source devra être téléchargé (upload)** à l'aide du lien qui sera mis à disposition sur **moodle** (Topic 12). Vous êtes responsables de vérifier que l'upload s'est bien passé en le téléchargeant (download) dans votre compte et en examinant que tout est bien présent.

Vous pouvez toujours faire un nouvel upload jusqu'à la date limite.

3) **Un rapport imprimé** d'au maximum **une feuille** de papier A4 recto-verso (**2 pages max**) écrit à l'ordinateur. Ce rapport doit être amené dans les casiers devant la salle INJ 141.

Vous devez avoir téléchargé votre code pour le **dimanche 8 décembre à 23h59** et livré votre code imprimé avec votre rapport (avec prénom et nom) avant le **lundi 9 décembre à midi**.

5.1 Rapport

Le Rapport ne contient PAS de page de titre, ni de table des matières
Le Rapport contient :

a) Résultat de la phase **d'analyse** (max 1 page, police de taille 11) :
Décrire l'organisation générale du programme en faisant ressortir la mise en oeuvre des principes *d'abstraction* et de *ré-utilisation*.

b) Pseudocode de votre **algorithme** de dichotomie pour **l'étape** décrite dans la section **3.3.2**. (pas de code source)

c) fournir les réponses aux questions des sections 3.3.1 et 3.3.2.

6. Barème indicatif (12pts):

Ce barème est indicatif et provisoire. Il sera éventuellement modifié par la suite.

(2pt) rapport (Fig7) : description de la mise en œuvre du principe d'abstraction et de réutilisation, analyse , pseudocode de la tâche 3.3.2 et réponses à 3.3.1 et 3.3.2.

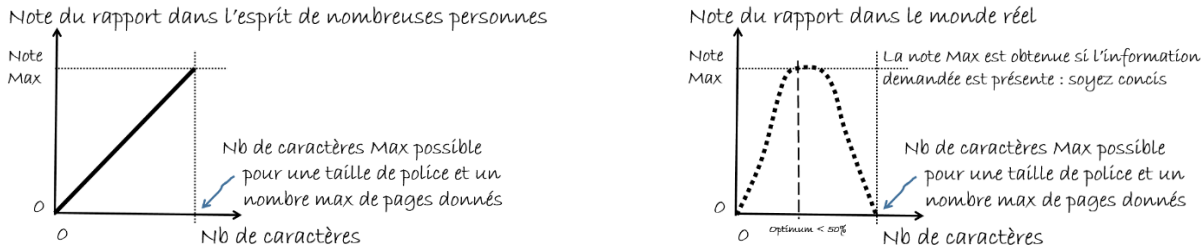

Figure 7 : un mauvais rapport dilue l'information utile jusqu'à atteindre le nombre maximum de caractères possibles du fait d'une croyance scolaire antérieure (gauche) ; un bon rapport est celui qui fournit les informations demandées avec concision (droite)

(4pt) Lisibilité, structuration du code et conventions de programmation du cours

(3pt) votre programme fonctionne correctement avec les fichiers fournis

(3pt) votre programme fonctionne correctement avec les fichiers non-fournis

Dernier rappel : le projet d'automne est INDIVIDUEL. Le détecteur de plagiat sera utilisé selon les recommandations du SAC.