Algorithmique des arbres
Générations aléatoire de labyrinthe

VILAYVANH Johnson TP 8

Licence Mathématiques et Informatique

PARTIE UTILISATEUR

DESCRIPTION DU PROGRAMME

- Le programme permet à l'utilisateur de créer et d'afficher un labyrinthe créé de façon aléatoire.
- Il peut également personnaliser son labyrinthe, sa taille, sa difficulté (son nombre de chemin), ou bien l'affichage de celui-ci. De plus, la création de celui-ci peut se faire instantanément, ou alors il peut observer la création de celui-ci s'il le souhaite, en instaurant un délai dans sa création.

LANCER LE PROGRAMME

Compiler le programme

- Un **makefile** permet de faciliter la compilation du projet. Ainsi, un simple « make », ou « make install » dans le dossier où se situent les fichiers permet de compiler celui-ci dans sonintégralité. Un exécutable « Labyrinthe » va ainsi se créer (ou un dossier « Projet » si on « make install »).

Lancement du programme

- Lorsqu'on lance le **programme sans paramètre**, un labyrinthe de taille 6x8 va se créer de façon aléatoire et s'afficher instantanément.

Exemple : **./Labyrinthe**

- Cependant, comme indiquer dans la description du programme, l'utilisateur peut personnaliser le labyrinthe qu'il va créer, en fournissant des arguments à l'exécution du programme :

  (1) *--taille=NxM* (hauteur x largeur), afin de personnaliser la taille de celui-ci ;

  (2) *--graine=X*, où X est un entier qui permet de fixer la graine d'aléa : ainsi, à chaque fois qu'on fournit cette même graine à l'exécution du programme, le même labyrinthe se créera ;

  (3) *--attente=X*, avec X un temps en millisecondes, correspondant à l'intervalle entre chaque suppression d'un mur ;

  Si X = 0 ou si l'option n'est pas fournie, le labyrinthe se crée instantanément (même avec une énorme taille, par exemple un labyrinthe de 500x500, avec --unique et –acces, options suivantes, se crée en seulement 2-3 secondes)

  (4) *--unique*, qui imposera au labyrinthe de n'avoir qu'un seul et unique chemin ;

  (5) *--acces*, qui impose au labyrinthe de s'arrêter seulement lorsque toutes les cases de celui-ci sont accessibles. Sans cette option, il peut arriver que certaines cases du labyrinthe se retrouvent enfermées et inaccessibles depuis l'entrée ;

  (6) *--mode=texte*, afin d'afficher le labyrinthe en mode texte. A noter que la largeur du labyrinthe sera contrôlée par le programme, afin qu'il rentre dans la fenêtre du terminal sur lequel a été exécuté le programme, afin de ne pas déborder sur les bords, ce qui rendrait son affichage incorrect

Exemple : **./Labyrinthe --taille=300x300 --graine=10 --unique --acces**

PARTIE DÉVELOPPEUR

LE DÉCOUPAGE DU PROGRAMME

- Le programme est découpé en différents modules, et le makefile permet de tous les compiler afin de créer l'exécutable du programme :
  - **Main.c**, qui comme son nom l'indique, et le fichier main qui exécute le code pour le fonctionnement du programme ;
  - **Case**, **Coordonnees**, **Laby** et **Structures**, qui contiennent les structures principales du projet, plus précisément celles qui nous ont été indiquées de créer ;

    Quelques fonctions sont également contenues dans ces modules, notamment celles pour initialiser les cases et le labyrinthe. ;
  - **Fusion**, également au cœur du projet, avec les fonctions qui reposent sur le principe d'**Union Find**
  - **Graphique** et **Texte**, modules contenant les fonctions permettant d'afficher le labyrinthe respectivement en mode graphique, ou en mode texte ;
  - **Arguments** permet de récupérer les paramètres fournis lors de l'exécution du programme ;
  - **Murs**, qui comme son nom l'indique, va s'occuper des murs du labyrinthe, et les enlever de la structure labyrinthe ;
  - **Cellule_Valide** va essentiellement aider au bon fonctionnement de l'option « --unique »
  - Et enfin, **Liste_Chainee** : implantation personnelle d'une structure d'une liste chaînée, qui permettra de réduire de façon considérable la création des labyrinthes (comme indiqué dans la partie utilisateur, permet notamment de créer des labyrinthe de taille 500x500 en seulement 2-3 secondes !)

DESCRIPTION DES MODULES

- Le **main** aura naturellement une partie qui lui sera dédiée (la partie 3) Dans cette partie on se concentra ainsi uniquement sur les modules du projet.

Coordonnees.h & Structures.h

- Ces headers contiennent très peu de code, étant donné qu'il ne contient pour le premier, que la structure de coordonnes_t, et le second les chaînes UTF8 joint au sujet, pour l'affichage avancé du mode texte.

Case & Laby

- Comme indiqué lors de l'introduction du découpage du projet, ces modules contiennent principalement les structures **case_t** et **labyrinthe_t**. Cependant, plusieurs changements ont été apportés à ces structures.
- Dans un premier temps, on peut noter que j'ai d'abord utilisé les structures en un seul tableau fournis à la fin de l'énoncé.
- Pour la struture **laby_t** :
- On supprime le champ « *taille* » de labyrinthe, et on utilise des variables extérieures « hauteur_laby » et « largeur_laby », qui étaient un peu plus explicites d'un point de vue personnelle
- Le champ « *int cases_non_accessibles* » est crée, qui permet, lorsqu'on voudra utiliser l'option « --acces » (mais pas « --unique » en même temps, bien que ça ne paraisse pas très naturelle, ni vraiment utile d'utiliser cette option sans l'autre).
- Tant que cette variable n'aura pas atteint 0, on continuera de supprimer des murs. Celle-ci sera décrémenter chaque fois qu'on s'aperçoit qu'une case a la même classe que celle de la case du départ, cases qu'on étudiera seulement à partir du moment où un chemin acceptant du départ jusqu'à l'arrivée est crée, avant ça, très peu d'intérêt de le faire, les classes changeant sans arrêt.

  Pour savoir quelles cases n'ont pas encore la même classe que la case du départ, on utilisera une **table de hachage**, dont le fonctionnement sera décrit plus amplement dans une partie suivante.

- Pour la struture **case_t** :
  - ➔ On crée « *int mur_a_detruire* », indiquant le nombre de mur qu'on peut encore détruire :
    - 0 → aucun mur ne peut être détruit ;
    - 1 → on peut détruire le murEst et le murSud ;
    - 2 → on peut détruire que le murEst ;
    - 3 → on peut détruire que le murSud ;
  - ➔ Ainsi, une case sur le bord sud, pour laquelle on ne peut détruire que le murEst, aura mur_a_detruire = 2.

➔ Va également servir pour l'option « --unique » : lorsqu'on remarque qu'un mur ne peut pas être détruit, car la case appartient à la même classe que ses cases adjacentes est et sud, on change la valeur de son champ. Couplé avec les listes chaînée, cela nous permettra de choisir aléatoire seulement parmi les murs qui peuvent encore être détruit

- On note également qu'on stocke dans Laby la fonction qui vérifie qu'on a fini de générer notre labyrinthe, ***labyrinthe_fini***, qui return une condition différente selon les options fournis au lancement du programme (unique et/ou acces)

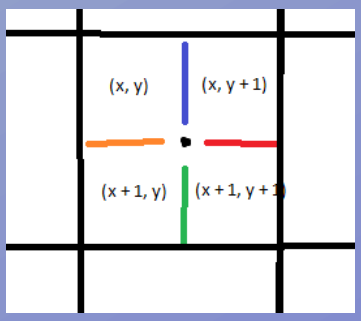
Fusion

- On stocke dans ce module les fonctions ***Trouve*** et ***Fusion*** qui repose sur le principe d'**Union-Find**, qu'on a bien évidemment adapté avec la structure de donnée des cases. Leur principe reste cependant le même.

Graphique et Texte

- Le module Graphique reste très basique : on dessine un mur aux coordonnées correspondantes pour toutes les cases qui possède encore au moins 1.
- Le module Texte possède l'affichage en texte basique, avec des « * » et « - », mais également l'affichage avancée avec les chaînes UTF8 qui ont été fournies (en mode texte, on utilise bien évidemment l'affichage avancée)
  - ➔ L'affichage basique, qu'on affiche avec la fonction **affichage_texte**, utilise la librairie « ncurses », et son principe reste également très basique, puisqu'on dessine un mur pour chaque case qui en possède encore, aux coordonnées indiquées.
  - ➔ L'affichage avancé va quant à lui, tracer ses cases d'une façon un peu plus complexe, mais pas tant qu'on aurait pu le croire à première vue :
    - On affiche dans un premier temps la 1ère ligne, le grand mur en haut
    - Puis, en même temps et dans cet ordre pour chaque ligne :
      - Le mur à l'extrémité gauche
      - Le corps d'une ligne
      - Le mur à l'extrémité droite
    - Et enfin, la dernière ligne, le grand mur en bas
  - ➔ A chaque fois qu'on va afficher un caractère, il faudra faire attention à l'existence des murs qu'ils l'entourent, et ainsi afficher le caractère qu'il faut pour la situation
  - ➔ Pour tout l'intérieur du labyrinthe, on se sert plus précisément de la fonction ***print_intersection*** : comme son nom l'indique, à chaque intersection entre 4 cases, on observe l'existence du mur Nord, Est, Sud et Ouest. On affiche ensuite le caractère correspondant

- ➔ **Ligne Bleu** : murEst de (x, y)
- ➔ **Ligne Rouge** : murSud de (x, y + 1)
- ➔ **Ligne Verte** : murEst de (x + 1, y)
- ➔ **Ligne Orange** : murSud de (x, y)



Arguments

- ➢ C'est le module qui s'occupe de récupérer les options que va fournir l'utilisateur lors l'exécution du programme.
- ➢ A l'aide d'un tableau de tableaux de caractères, contenant chacun les chaînes qui décrivent l'énoncé d'une option (par exemple « --unique », « --graine= », on va comparer chacun de ces éléments aux arguments fournis par l'utilisateur.
- ➢ Si celui-ci fournit bien une option, on change son homologue dans un tableau d'int qui correspond aux variables de chaque option :
  - ➔ 0 ou 1 pour activer l'option ou non ;
  - ➔ Ou alors la valeur de la variable de base si aucune valeur n'est fournie, par exemple la taille du labyrinthe qui vaut 6x8)
- ➢ Le programme va ainsi s'adapter selon les options fournis par l'utilisateur

Murs

- ➢ On va ici s'intéresser à la suppression des murs d'une case du labyrinthe, régit par la fonction ***enleve_mur***
  - ➔ Selon la valeur de son champ « mur_a_detruire » comme expliqué précédemment, on va **supprimer** le mur qu'on a sélectionné, et **fusionner** les cases qui ne sont plus séparés par un mur :
    - Lorsque mur_a_detruire vaut 1, on lui ajoute de façon aléatoire, soit 1 ou 2, pour ainsi supprimer soit son murEst, ou son murSud.
    - Et si mur_a_detruire vaut 2 ou 3, on détruit le mur correspondant
  - ➔ A la fin de la fonction, on met à jour son champ « mur_a_detruire » grâce à la fonction ***maj_mur_a_detruire*** :
    - s'il ne lui restant plus qu'1 mur à détruire, donc si sa valeur était 2 ou 3, sa valeur vaut maintenant 0
    - s'il lui restait 2 murs à détruire, on change la valeur en 2 ou 3, selon le mur qu'on vient de détruire
  - ➔ A noter comme dit précédemment que les murs sélectionnés par cette fonction sont piochés parmi une liste de mur qu'on était certain de pouvoir détruire, et non pas parmi tous les murs de toutes les cases du labyrinthe, détruit ou non. On expliquera comment on procède dans la partie des listes chaînées

Cellule_Valide

- Les fonctions de ce module vont principalement servir au bon fonctionnement de l'option « --unique » :
  - ➔ La fonction ***meme_classe*** va tout d'abord récupérer les cases du labyrinthe indiquées en paramètre afin de contrôler l'appartenance à la même classe ou no
  - ➔ Par la suite, la fonction ***verif_voisins_classe*** contrôlera l'appartenance à la même classe ou non d'une case et la case adjacente au mur qu'on veut détruire
  - ➔ Ainsi, avant même de détruire un mur, on pourra voir au préalable s'ils ne sont pas déjà dans la même classe
- Le champ « mur_a_detruire », introduit précédemment, refait également son apparition : si on remarque qu'un mur ne peut pas être détruit car 2 cases, séparés par un mur, sont de la même classe, alors on change le champ « mur_a_detruire » de la case dont le mur appartient, afin de se souvenir qu'on ne pourra plus supprimer ce mur

Mais comment peut-on savoir quelles sont les cases qu'on peut encore détruire ? Même si on sait qu'on ne peut plus détruire un mur, on peut tout de même tomber sur lui lorsqu'on tire aléatoire un mur à détruire : on saura peut-être tout de suite qu'on ne peut plus le détruire, mais on va quand même ratisser un certain nombre de case avant d'enfin tomber sur 1 case qui possède encore 1 mur à détruire.

C'est précisément cette question que je me suis posé lorsque je suis tombé face à la problématique d'optimisation du programme : comment faire en sorte de ne sélectionner que les cases qu'on peut encore détruire…

Avec une liste chaînée ! Et même mieux, une table de hachage.

Liste_Chainee

- Si on stocke dans une **table de hachage** toutes **les coordonnées des cases qui ont des murs qu'on peut encore détruire**, donc qui ont mur_a_detruire != 0, et qu'on pioche seulement parmi toutes ces cases, on accélère de façon exponentielle notre génération de labyrinthe !
- Naturellement, on a toutes les fonctions liées aux listes chaînées :
  - ➔ ***free_listecasedetruisable*** et ***free_liste_hachage*** pour libérer l'espace qu'on a alloué pour créer les listes chaînées ;
  - ➔ ***alloue_case_liste***, pour allouer l'espace d'une case d'une liste chaînée ;
  - ➔ ***init_liste_mursdetruisables*** et ***init_hachage_mursdetruisables***, afin d'initialiser la table de hachage ;
  - ➔ ***calcul_indice_hachage*** et ***init_tailles_listes***, pour savoir à quelle liste dans la table de hachage on va s'intéresser

    init_tailles_listes initialise un tableau de int dans lequel chaque indice correspond aà la taille chaque liste de la table de hachage, mis à jour chaque fois qu'on réajuste la taille d'une liste;
  - ➔ ***extrait_coordonnees_case,*** pour extraire les coordonnées d'une case qu'on a tiré de façon aléatoire dans un intervalle [0, nombre de cases restantes avec un mur] ;

- ➔ ***supprime_premiere_case***, ***supprime_case*** et ***supprime_case_quelconque*** pour supprimer les coordonnées d'une case qui ne possède plus de mur à détruire

- On a également ***maj_cases_accessibles***, étant donné qu'on va également se servir d'une table de hachage lorsqu'on activé l'option « acces » sans « unique », mais une table de hachage au fonctionnement différent de celle décrite plus haut :
  - ➔ Dans cette fonction, on étudie chaque case du labyrinthe qui n'ont pas encore la même classe que celle de la case du départ
  - ➔ Si une case a la même classe que la case du départ, on la supprime de la liste
  - ➔ On lance pour la 1ère fois cette fonction dès lors qu'on réussit à créer un chemin acceptant dans le labyrinthe : à partir de ce moment là, on s'arrête seulement dès que toutes les cases sont dans la même classe, donc il faudra observer et mettre à jour cette liste jusqu'à qu'elle soit vide

- Avec une **table de hachage**, on n'a pas à tirer aléatoire une case parmi TOUTES les cases du labyrinthe : on peut simplement rirer aléatoirement parmi LES cases encore valides.
- S'il ne nous restait plus qu'1 case à détruire avec l'option « --acces », avec un labyrinthe de 400x400, on devrait trouver 1/160 000 cases sans cette méthode… Alors qu'avec une table de hachage, on sait **instantanément** quelle case et mur il nous restait à détruire

C'est avec toutes ces modules qu'on va créer notre programme main, et ainsi créer aléatoirement nos labyrinthes.

MAIN

Initialisation des variable

- On n'oubliera pas ici de créer les ***tableaux options*** et ***texte_options***, qui coupler avec la fonction teste_arguments, vont nous aider à récupérer les options fournis par l'utilisateur à l'exécution du programme.
- On créer notre **labyrinthe**, nos **tables de hachage**, et on ouvre une fenêtre si on veut afficher graphiquement le labyrinthe.
- A noter que le **mode texte** ne permet pas d'afficher des labyrinthes énorme, la taille étant limitée à la taille du terminal : on limite donc la largeur du labyrinthe (mais pas sa hauteur, on peut scroll pour voir le labyrinthe, mais si une ligne du labyrinthe est plus grande que la largeur du terminal, elle passe à la ligne suivante, et résultat, l'affichage du labyrinthe n'a plus aucun sens)

Boucle principal

- On ne quittera pas la boucle tant que la fonction ***labyrinthe_fini*** ne return pas 1, fonction qui comme dit précédemment, évalue une condition différente selon les options fournis (unique et/ou acces)
- On tire aléatoirement une case dans l'intervalle [0, nb_case_detruisable], nb_case_detruisable qui correspond au nombre de case qui ont encore au moins 1 mur détruisable
- On extrait les coordonnées de la case tiré aléatoirement
  - ➔ Si on a l'option « unique », on vérifie tout de même avant de supprimer le mur de la case sélectionnée que la case n'a pas la même classe qu'un de ses cases adjacentes au sud ou à l'est. Et si c'est le cas, on met à jour son champ de mur_a_detruire, car on peut considérer le mur comme intouchable

    Lorsqu'on effectue un tel changement, on fait attention si ce champ devient 0, au quel cas on supprime les coordonnées de la case correspondante dans la table de hachage
  - ➔ Il est possible qu'en ayant les 2 options « acces » et « unique » activées simultanément qu'il ne reste plus de mur détruisable : dans ce cas là, on sort de la boucle principal
- On continue en détruisant un des murs encore détruisable de la case qu'on a tiré aléatoirement
  - ➔ Si on a crée un chemin acceptant et qu'on a « acces » mais pas « unique », comme dit précédemment, on va à partir de ce moment là observer continuellement la table de hachage des cases restantes qui n'ont pas encore la même classe que la case de départ, jusqu'à que celle-ci soit vide
- Si la case qu'on vient de détruire n'a plus de mur détruisable, on doit l'enlever de la table de hachage des cases restantes encore détruisables
- Si on a une attente > 0, on affiche le labyrinthe et on attend durant le délai fourni avant de tirer à nouveau un mur à détruire

Une fois le labyrinthe terminé, plus qu'à l'afficher ! Sans oublier pas de libérer tout l'espace alloué…

CONCLUSION

- Projet très intéressant à créer : sans nous donner la méthode explicitement, on se rend compte que les listes chaînées, énorme notion apprise au 1er semestre, vient nous servir et nous optimise énormément notre création de labyrinthe, en accélérant le processus de façon exponentiel.
- Ironiquement, la partie sur les arbres, l'**Union-Find,** m'a été très facile à réaliser : le plus gros problème sur lequel je me suis heurté, c'est l'optimisation de celui-ci.
- Cependant, c'est sans compter l'option « **--victor** » sur laquelle je n'ai pas la moindre piste que je n'ai donc pas réussi à créer : étant donné que c'est la seule option que je n'ai pas pu résoudre, le problème majeur du projet résidait sans doute dans la résolution de celui-ci...