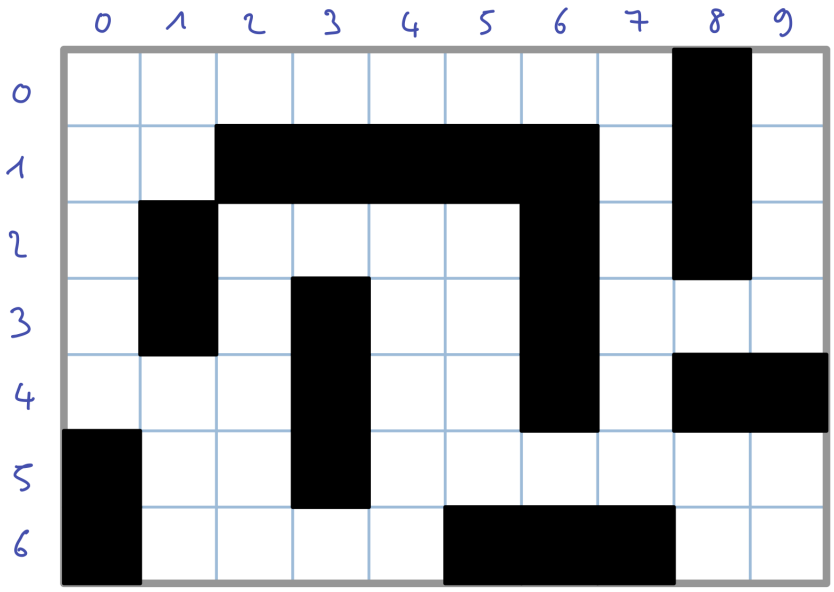
Langages C et C++ : projet

Version du 4/02/2021

Présentation

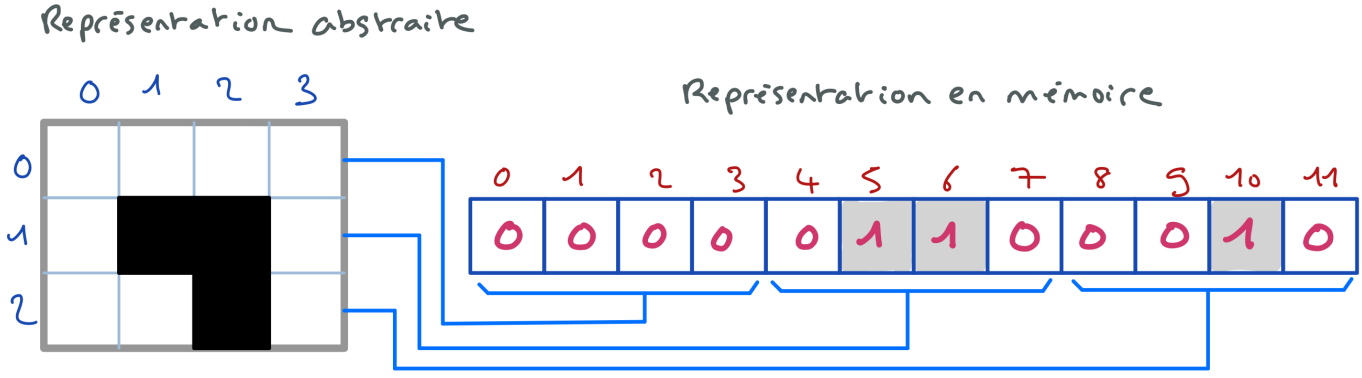
Ce projet comporte plusieurs parties facultatives de différents niveaux, pouvant rapporter chacune un certain nombre de points. Vous pourrez étaler les livraisons en fonction de votre rythme de progression.

Les différentes parties concernent toutes la notion de labyrinthe et de recherche de chemin dans un labyrinthe.



D'un point de vue abstrait, on appellera labyrinthe une grille à deux dimensions avec deux types de cases : les cases libres (blanches) et les murs (noirs).

Concrètement, un labyrinthe sera représenté en mémoire par un tableau à une dimension de **char** dans lequel les contenus des lignes de la grille seront placés bout à bout. Les cases libres seront représentées par des 0, les murs par des 1.



Partie 1 (2 points)

Travail à réaliser

Le but de cette partie est d'implanter une grille et les fonctions permettant de déterminer ou modifier les statuts des cases (blanches ou noires). On travaille pour l'instant avec un tableau dynamique accessible via une variable globale qui représente une grille dont la longueur et la largeur sont fixées par des variables globales. En changeant les valeurs de ces variables, par modification des deux lignes qui les initialisent dans le code source, on doit pouvoir travailler avec des grilles plus grandes que l'exemple donné de 3 lignes et 4 colonnes. Les limites des valeurs applicables sont celles de l'affichage de la console utilisée pour exécuter l'application. Les consoles peuvent afficher en général 80 colonnes, voire plus, et leur nombre de lignes visibles dépend de différents paramètres d'affichages du système utilisé, tel que la police de caractères de la console, mais peut largement atteindre 50 dans la plupart des cas. Ces valeurs ne sont pas critiques, mais quand vous ferez des programmes de démonstration, merci de rester dans les limites de 78 colonnes et 45 lignes.

```
int NB_COLONNES = 4; //longueur
int NB_LIGNES   = 3; //largeur

char* Grille=NULL;
```

La ligne…

```
Grille = (char*)calloc(NB_LIGNES*NB_COLONNES,sizeof(char));
```

…devra être ajoutée au début de la fonction `main` pour initialiser ce tableau, et la ligne…

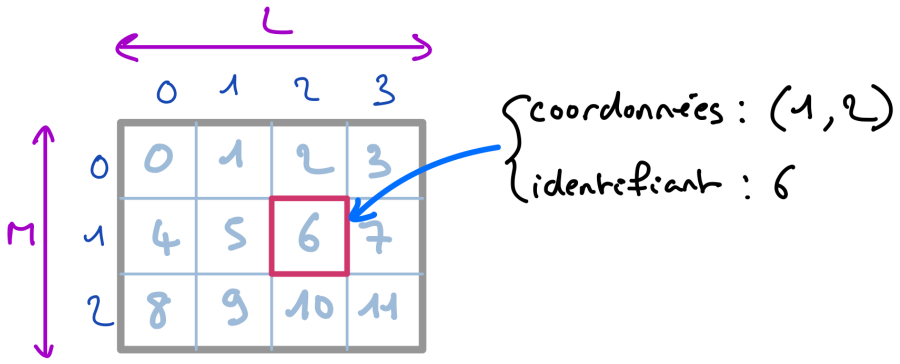
```
free(Grille) ;
```

…devra être ajoutée à la fin de la fonction `main` pour libérer la mémoire occupée par ce tableau. (Dans les faits, l'arrêt de l'exécution d'un programme libère toute la mémoire réservée dans le tas, mais c'est plus propre de le faire explicitement.)

Chaque case de la grille peut être identifiée de deux manières :

- soit par ses coordonnées (ligne, colonne) dans la grille,
- soit par son indice dans le tableau `Grille`. Cet indice sera appelé **identifiant** de la case.

Par exemple, la case marquée d'une croix dans la figure ci-dessous à pour coordonnées (1,2) et pour identifiant 6.



Vous devez implémenter les fonctions suivantes :

```
//retourne l'identifiant d'une case dont on donne les coordonnées
int getID(int ligne, int colonne);

//retourne la première coordonnée (ligne) d'une case dont on donne l'identifiant
int getLigne(int id);

//retourne la deuxième coordonnée (colonne) d'une case dont on donne l'identifiant
int getCol(int id);

//place la valeur x dans le case de coordonnée (i,j)
void modifie(int ligne, int colonne, char x);


//retourne la valeur de la case de coordonnées (i,j)
char lit(int ligne, int colonne);


//affiche la grille

char AFF_VIDE = ' '; //Caractère représentant les cases vides pour l'affichage
char AFF_MUR  = 'X';  //Caractère représentant les murs pour l'affichage
char AFF_BORD = '+'; //Caractère représentant les bords pour l'affichage

void affiche();
```

Merci de placer toutes les constantes au début du fichier source.

Voici un exemple de fonction **main** qui modifie et affiche la grille, et l'affichage attendu dans lequel les tirets représentent les cases blanches et les X des cases noires.

```
int main()
{
    modifie(1,1,1);
    modifie(1,2,1);
    affiche();

    return 0;
}
```

```
 ----
 -XX-
 ----
```

Les bordures ne sont pas représentées dans le tableau (qui contient seulement les valeurs des cases de la grille). Elles doivent être affichées autour de la grille par la fonction `affiche`.

Par exemple si on change la valeur de la constante AFF_BORD pour lui donner la valeur '0' et AFF_VIDE pour lui donner la valeur ' ', on doit obtenir l'affichage suivant.

```
000000
0    0
0 XX 0
0    0
000000
```

Livrable

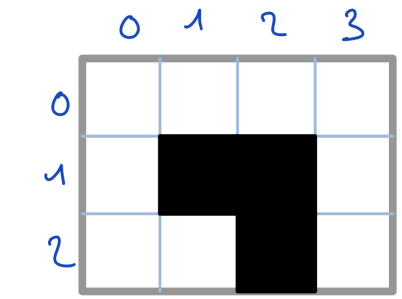
Vous devrez livrer uniquement le code source dans un unique fichier, en respectant scrupuleusement les consignes spécifiées à la fin de ce document.

Partie 2 (5 points)

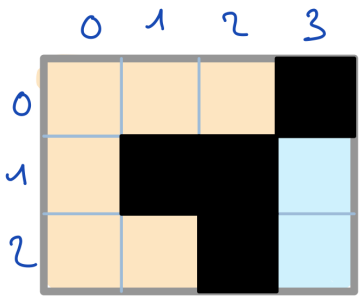
Connexité des cases blanches

Le but de cette partie est de réaliser une fonction permettant de déterminer si toutes les cellules blanches d'un labyrinthe sont connectées entre elles.

On considère que deux cellules sont connectées si elles sont contigües horizontalement ou verticalement (pas en diagonale). Voici un exemple de labyrinthe dans lequel toutes les cases blanches sont connectées et de labyrinthe à poches dans lequel les cases blanches ne sont pas toutes connectées.



Toutes les cellules blanches sont connectées



Il y a 2 poches non connectées

D'où une question très intéressante : comment déterminer si toutes les cellules blanches sont connectées ?

Nous allons présenter un algorithme permettant de faire cela. Il utilise le tableau `Grille` et une pile d'entiers. Cette pile sera réalisée avec un tableau global d'entiers appelé `Pile`, dont la taille sera la même que celle du tableau `Grille`, et d'une variable globale nommée `Sommet`. Plus tard, nous remplacerons cette représentation rudimentaire par une structure de donnée plus élaborée.

```
int* Pile = NULL;

// Ajouter :
// Pile = (char*)calloc(NB_LIGNES*NB_COLONNES,sizeof(char));
// Au début de la fonction main.
// et :
// free(Pile);
// A la fin de main.

int Sommet=0;
```

Deux fonctions permettent d'utiliser cette pile :

```
//empile un entier
void push(int x);

//dépile un entier et le retourne
int pop();
```

Le principe de l'algorithme est le suivant :

1. On parcourt le tableau **`Grille`** pour récupérer deux informations : le nombre de cases blanche et l'identifiant **`id`** d'une de ces cases, peu importe laquelle. S'il n'y a aucune case blanche, on affiche un message d'erreur. On place la valeur 2 dans **`Grille[id]`**, ce qui signifie que la case ayant cet identifiant a été visitée. On appellera cette opération : **marquer** la case d'identifiant id. Quand une case a un identifiant **`i`** et que **`Grille[i]`** vaut 2, on dira que cette case est **marquée**. On empile **`id`**.
2. Ensuite, on répète la séquence suivante jusqu'à ce que la pile soit vide :
    a. On dépile un identifiant **`id`**.
    b. On recherche les identifiants de toutes les cases blanches voisines de celle identifiée par **`id`** et qui ne sont pas marquées. (Deux cases sont voisines si et seulement si elle se touchent verticalement ou horizontalement.) On marque chacune de ces cases et on empile son identifiant.
3. On parcourt le tableau **`Grille`** pour la nettoyer en remplaçant les 2 par des 0 et on en profite pour compter les cases ayant été marquées.
4. Toutes les cases blanches sont connectées si et seulement si le nombre de cases marquées est égal au nombre total de cases blanches.

Vous devez implémenter cet algorithme sous la forme de la fonction **`connexe`** qui détermine si toutes les cases blanches sont connexes.

```
//détermine si toutes les cases blanches sont connectées.
int connexe();
```

Vous devez faire des tests pour vérifier que cette fonction a le comportement attendu.

Génération automatique de labyrinthe

Vous devez réaliser une fonction **`void genLaby(int k)`** qui produit un labyrinthe « intéressant » avec les critères suivants :

- Les cases situées dans les coins opposés de coordonnées (0, 0) et (NB_LIGNES-1, NB_COLONNES-1) doivent être blanches.
- Les cases blanches doivent être connectées entre elles.
- Les cases noires doivent être positionnées aléatoirement lors de la construction.
- Le nombre de cases blanches doit s'approcher autant que possible de la valeur du paramètre k.

La fonction **`genLaby`** ne doit pas entrer dans une boucle infinie, elle doit toujours se terminer.

Faites des tests avec des petits de de plus grands labyrinthes. Voici un exemple de résultat obtenu avec NB_COLONNES=10, NB_LIGNES=6 et k=30.

```
-XX-XX-XXX

-X------X-

---X-XX---

-XXXXX--XX

---XXX-XXX

XX-XXX----
```

Livrables

Vous devez livrer

- un unique fichier source respectant scrupuleusement les consignes spécifiées à la fin de ce document,
- un document pdf d'au plus 3 pages expliquant votre solution pour produire le labyrinthe « intéressant » et donnant quelques exemples de labyrinthes obtenus.

Partie 3 (5 points)

Il est temps de s'amuser un peu. Dans cette partie, on vous **donne** une classe Labyrinthe toute faite, sous la forme d'un fichier binaire pouvant être intégré à votre projet. Mais vous n'avez pas accès au code source de cette classe. Voici les méthodes disponibles. Celles qui ont le même nom que dans les parties précédentes font la même chose.

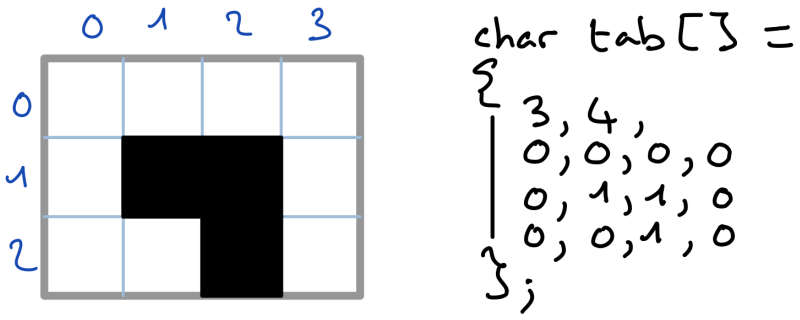
```
class Labyrinthe
{
public:
    Labyrinthe(int nLignes, int nColonnes);
    Labyrinthe(char data[]) ;
    ~Labyrinthe();
    int getLongueur();
    int getLargeur();
    int getID(int ligne, int colonne);
    int getLigne(int id);
    int getCol(int id);
    void modifie(int ligne, int colonne, char x);
    void affiche();
    char lit(int ligne, int colonne);
    bool connexe();
    void genLaby(int nb);
    int distMin(int id1, int id2);
};
```

Le constructeur à deux paramètres crée une grille remplie de cases blanches dont le nombre de lignes (largeur) et de colonnes (longueur) sont passés en paramètres.

Le constructeur à un paramètre crée une grille à partir d'un tableau de caractères appelé **descripteur**, avec la convention suivante :

- Les deux premières valeurs du tableau sont le nombre de lignes et le nombre de colonnes de la grille.
- Les valeurs suivantes sont les contenus des cellules de la grille (0 ou 1) avec la convention expliquée dans la partie 1.

Voici un exemple de grille et de son descripteur.



Ce constructeur vous permettra de créer des labyrinthes avec des descripteurs fournis par vos enseignants à des fins de tests par exemple.

Une méthode **`distMin`** est également apparue. Cette méthode retourne le plus petit nombre de déplacements nécessaires pour aller de la case blanche d'identifiant **`id1`** jusqu'à celle d'identifiant **`id2`** en ne passant que par des cases blanches. Chaque étape de déplacement consiste à passer d'une case à une case voisine horizontalement ou verticalement (pas en diagonale).

Travail à réaliser

Le but de cette partie est de simuler une poursuite entre deux robots à l'intérieur d'un labyrinthe. Les robots sont identifiés par des lettres A et B. Leurs positions initiales peuvent être soit déterminées aléatoirement, soit être imposées aux deux coins opposés de la grille.

Les mouvements se font tour par tour. À chaque tour, chacun des robots fait un mouvement en se basant sur la position actuelle de l'autre.

Le robot A, qui est le poursuivant, choisit toujours une case de manière à minimiser la distance (déterminée avec la méthode **`distMin`**) qui le sépare de la position du robot B.

Le robot B doit essayer d'échapper au robot A. Son algorithme de déplacement n'est pas imposé. C'est à vous d'imaginer plusieurs solutions et de les tester. Vous pouvez utiliser la distance calculée par la méthode **`distMin`**, mais aussi la distance à vol d'oiseau, et vous pouvez faire intervenir une part de hasard (par exemple, certains mouvements sont aléatoires).

Vous devrez implémenter une animation très rudimentaire en mode texte, qui consiste, après chaque mise à jour, à afficher quelques dizaines de retour à la ligne suivi de l'affichage du labyrinthe et des robots (ou à utiliser tout autre moyen pour effacer la console qui fonctionne sous Linux). Les mises à jour peuvent se faire toutes les secondes. La fonction **`time`** est votre amie.

La performance du robot B sera mesurée sur un labyrinthe produit avec un descripteur de labyrinthe qui vous sera fourni. Pour cette mesure, les deux robots partiront de deux coins opposés de la grille. La poursuite sera simulée 1000 fois, sans affichage ni temporisation. Chaque poursuite s'arrête après 1000 mouvements ou lorsque les deux robots arrivent en contact. Le score du robot B est la durée moyenne de la poursuite.

Livrables

Vous devez livrer

- un unique fichier source respectant scrupuleusement les consignes spécifiées à la fin de ce document,
- un document pdf d'au plus 4 pages expliquant présentant les algorithmes que vous avez essayé pour le robot B et donnant les résultats obtenus avec le labyrinthe de référence.

Partie 4

Vous devez faire un choix en fonction de votre niveau et de votre avancement, entre deux variantes appelées 4A et 4B. Vous êtes libre de ne traiter aucune des deux ou de traiter l'une des deux.

Partie 4A (2 points)

Dans cette partie, vous devez concevoir et implémenter la classe `Labyrinthe` de la partie 3 **sans la méthode distMin**. Vous pouvez bien sûr reprendre en l'adaptant, pour les méthodes où cela est applicable, le code développé dans les parties 1 et 2. Toutes les méthodes implémentées doivent être testées.

Partie 4B (6 points)

Cette partie s'adresse aux étudiantes et étudiants les plus avancés.

Vous devez concevoir et implémenter la classe **`Labyrinthe`** de la partie 3 **avec la méthode distMin**. Vous pouvez bien sûr reprendre en l'adaptant, pour les méthodes où cela est applicable, le code développé dans les parties 1 et 2. Toutes les méthodes implémentées doivent être testées.

C'est à vous de trouver un algorithme pour la méthode **`distMin`**. Vous pouvez vous inspirer de l'algorithme utilisé pour tester la connexité des cases blanches. Vous pouvez aussi vous renseigner sur l'algorithme de Dijkstra pour la recherche de plus court chemin. (Les deux conseils ne sont pas exclusifs.)

Livrables

Si vous traitez la partie 4A, livrez juste un unique fichier source respectant scrupuleusement les consignes spécifiées à la fin de ce document.

Si vous traitez la partie 4B, vous devez, en plus du fichier source, livrer un document pdf d'au plus 4 pages expliquant votre algorithme pour la méthode **`distMin`** et décrivant les tests que vous avez faits et les résultats.

Contribution personnelle (2 points)

Vous pouvez ajouter au projet une contribution personnelle, telle que par exemple l'affichage des chemins optimaux, ou une poursuite avec plusieurs robots poursuivants et / ou plusieurs robots poursuivis, ou une version avec un robot contrôlé par l'utilisateur au tour par tour, ou un moyen de produire des labyrinthes encore plus intéressants, etc., voire, si vous êtres

très motivé, l'utilisation de la bibliothèque graphique SDL (Mais dans ce cas vous devrez livrer aussi une version texte tournant dans une console.)

Dates de livraison

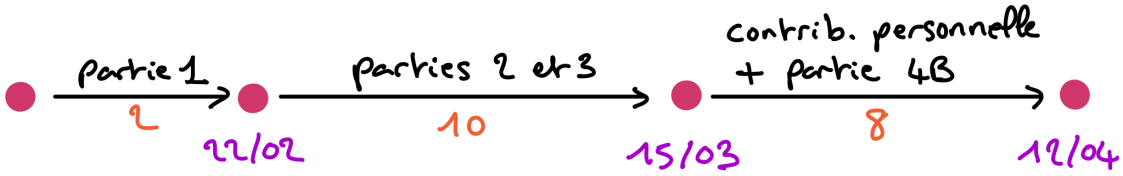
Les dates de livraison sont étalées selon trois parcours possibles qui vous permettrons de travailler à votre rythme, au mieux de vos capacités. Ces dates sont données à titre indicatif. En cas de modification, vous serez informés via TEAMS.

Rappel des objectifs des parties

- Partie 1 : représentation du labyrinthe (2 points).
- Partie 2 : génération de labyrinthes intéressants (5 points).
- Partie 3 : poursuite de robots dans le labyrinthe (5 points).
- Partie 4A : encapsulation dans une classe C++ (2 points).
- Partie 4B : encapsulation et recherche de chemins optimaux (6 points).
- Contribution personnelle (2 points).

Parcours MAX (note ≤ 20)

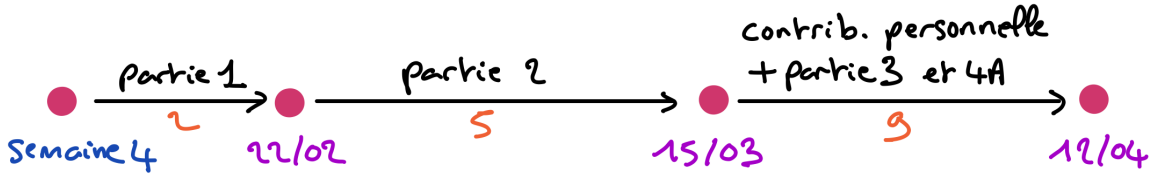
Si vous choisissez ce parcours, le plus difficile, vous devrez livrer les parties 1, 2, 3 et 4B, en cas de retard, vous pourrez adopter un parcours moins difficile mais qui pourra tout de même vous permettre d'avoir une bonne note.



Les livraisons doivent être faites avant minuit aux dates indiquées.

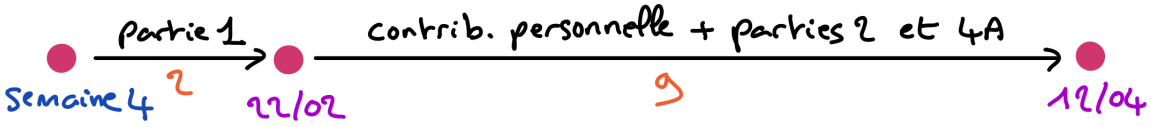
Parcours Standard (note ≤ 16)

Ce parcours est un peu plus facile que le précédent et permet d'avoir une note très honorable.



Parcours Mini (note ≤ 11)

Ce parcours peut permettre aux élèves les moins rapides d'optimiser leur progression et leur note sans compromettre, s'ils arrivent au niveau de compétence requis au moment de l'examen final, leur possibilité de valider l'unité d'enseignement.



Consignes concernant le code source

Le fichier source doit indenté et chaque accolade fermante doit être exactement en face (horizontalement ou verticalement) de l'accolade ouvrante associée.

Toutes les constantes doivent être placées au début du fichier source.

Les commentaires ne doivent pas paraphraser le code mais donner toute information utile à sa compréhension. Chaque fonction doit être précédée d'un commentaire détaillant son rôle et celui de chacun de ses paramètres. Chaque déclaration de variable doit être suivie (sur la même ligne) ou précédée d'un commentaire précisant le rôle de la variable concernée.

Le code doit être compilable sans erreur ni message d'avertissement (warning) avec gcc ou g++ sous Linux en utilisant les options -g -Wall. La ligne de commande permettant la compilation doit être mise en commentaire au début du fichier source.

Les variables doivent avoir des nom explicites et pertinent évoquant leurs rôles. Les noms doivent avoir une longueur minimum de trois lettres. Quelques variables nommées avec une seule lettres peuvent toutefois être utilisées si cela améliore la lisibilité du code.

La clarté du code sera prise en compte lors de l'évaluation.

Seul ou en binôme

Vous pouvez traiter ce projet seul ou en binôme, mais si vous travaillez à 2, vous devrez en plus implémenter des tests unitaires automatiques de toutes les fonctions et méthodes à réaliser. Pour chacune des fonctions et méthodes (sauf **`affiche`**), vous devez développer une méthode de test associée qui retourne **`true`** si la méthode testée passe le test et dans le cas contraire retourne **`false`** et affiche un message expliquant le problème détecté.

Par exemple, la méthode ou la fonction **`test_modifie()`** effectue un test automatisé de la méthode **`modifie`** en réalisant des modifications et en vérifiant qu'elles ont bien eu l'effet attendu dans le tableau représentant la grille du labyrinthe.

Une fonction **`test`** doit être ajoutée au parties 1 et 2, et une méthode de classe **`test`** doit être ajoutée aux parties suivantes. Cette fonction ou méthode exécute toutes les fonctions ou méthodes de test. Le nom de chaque fonction ou méthode testée est affiché, suivi de la mention OK ou ERREUR selon le résultat. En cas d'erreur, la fonction ou méthode de test concernée aura affiché un message expliquant le problème.

Si vous travaillez seul, vous devez bien évidemment tester les méthodes à développer, mais vous n'êtes pas tenus d'automatiser ces tests. Vous pouvez les valider par examen visuel d'affichages écran.

Concours de la meilleure démo

Pour participer à ce concours, vous devrez réaliser un programme de démonstration déroulant automatiquement une séquence d'exécutions des fonctionnalités implémentées (production de labyrinthe, poursuite, visualisation de chemins optimaux ou autre contribution personnelle). Cette démonstration devra durer 2 minutes, à raison d'un mouvement par seconde, sur des grilles de 78 par 45. La capture vidéo devra être faite sans musique ni effet sonore, lors d'une exécution sur votre machine personnelle (quitte à recompiler avec un autre compilateur que gcc) pour éviter des problèmes de latence ou de qualité d'image. Les propositions seront compilées (en accéléré) en une seule vidéo sur la base de laquelle toute la promo sera invitée à voter pour choisir la meilleure démo. Si elles sont de qualité suffisante, une vidéo spéciale présentant les meilleures contributions (sur la base des votes) sera réalisée et proposée à la diffusion dans les actualités de l'UFR et / ou l'université (Je n'ai pas encore pris les contacts nécessaires pour garantir cette diffusion).