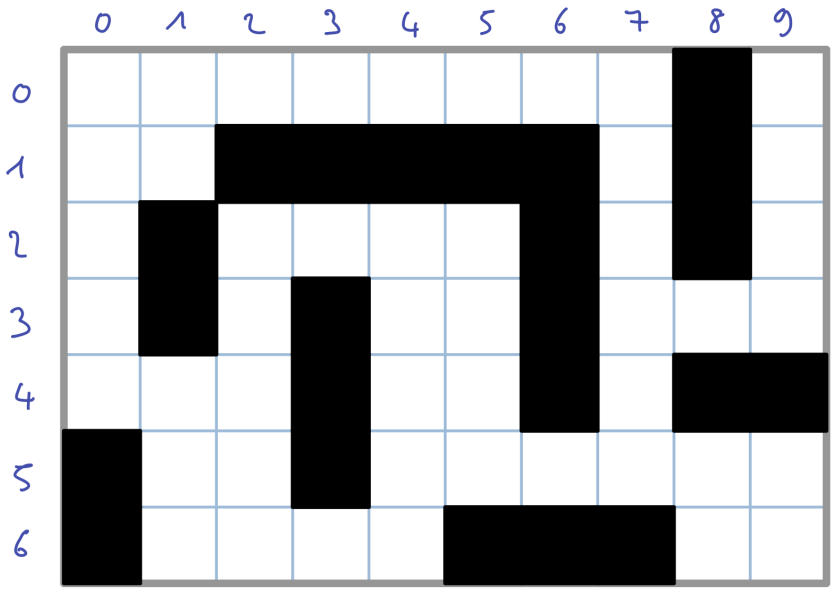
Langages C et C++ : projet

Version du 4/03/2021

Présentation

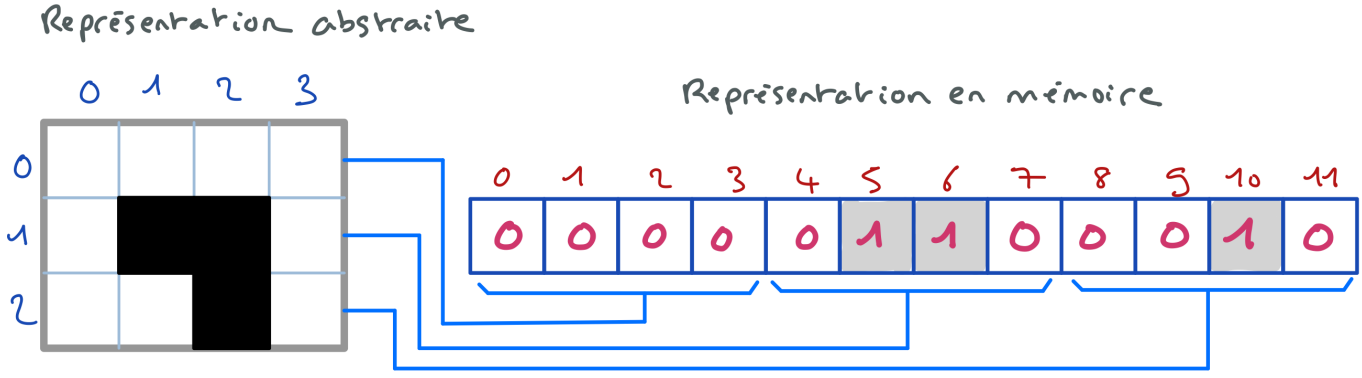
Ce projet comporte plusieurs parties facultatives de différents niveaux, pouvant rapporter chacune un certain nombre de points. Vous pourrez étaler les livraisons en fonction de votre rythme de progression.

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…

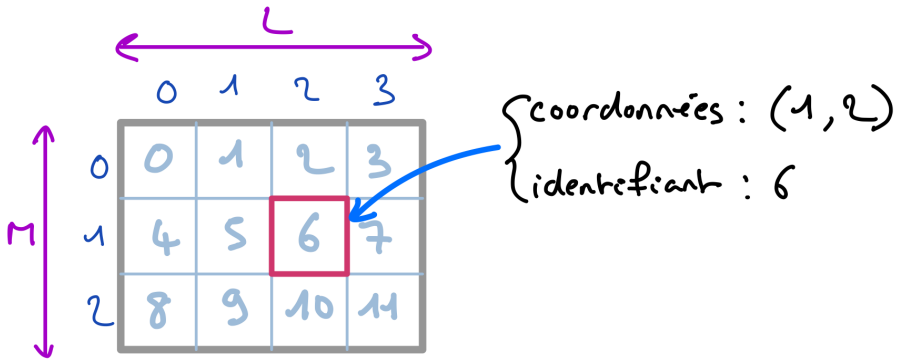
```
free(Grille);
```

…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()
{
    Grille = (char*)calloc(NB_LIGNES*NB_COLONNES,sizeof(char));
    modifie(1,1,1);
    modifie(1,2,1);
    affiche();
    free(Grille);
    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

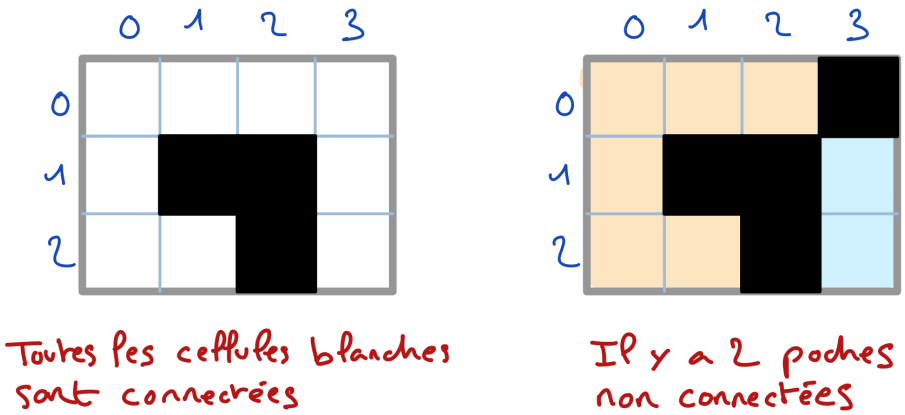
Vous devrez livrer uniquement le code source dans un unique fichier, en respectant scrupuleusement les consignes spécifiées à la fin de ce document.

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.



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 = (int*)calloc(NB_LIGNES*NB_COLONNES,sizeof(int));
// 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, avec le code automatisant les tests des fonction **genLaby** et **connexe** si vous travaillez en binôme.
- 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:
    /// Constructeurs et destructeurs
    Labyrinthe(int nLignes, int nColonnes); //Crée un labyrinthe vide.
    Labyrinthe(char data[]) ; //Crée un labyrinthe à partir d'un descripteur.
    ~Labyrinthe(); //Détruit le labyrinthe courant.

    /// Méthodes d'accès
    int getID(int ligne, int colonne); //Retourne l'identifiant d'une cellule.
    int getLigne(int id); //Retourne la ligne de la cellule d'identifiant id.
    int getCol(int id); //Retourne la colonne de la cellule d'identifiant id.
    void modifie(int ligne, int colonne, char x); //Modifie la valeur d'une cellule.
    char lit(int ligne, int colonne); //Retourne la valeur de la cellule de coordonnées (i,j).

    /// Méthode d'accès supplémentaires
    int getNbLignes(); //Retourne le nombre de lignes de la grille.
    int getNbColonnes(); //Retourne le nombre de lignes de la grille.
    int lit(int id); //Retourne la valeur de la cellule id.
    void modifie(int id, char x); //Modifie la valeur de la cellule id.

    /// Méthodes d'affichage
    // Initialise les caractères utilisés pour l'affichage.
    // Si motif contient n caractères alors
    // motif[i] est affiché pour représenter une case de valeur i entre 0 et n-2.
    // Toute autre valeur provoque l'affichage du caractère motif[n-1].
    void setAff(const char* motifs);
    void affiche(); // Affiche le labyrinthe.
    void afficheDescr(); //Affiche le descripteur du labyrinthe courant.

    /// Méthode de haut niveau
    bool connexe(); //Vérifie si toutes les cellules de valeur 0 sont connectées.
    void genLaby(int nb); //Produit un labyrinthe aléatoire connexe avec nb cases blanches.
    int distMin(int id1, int id2); //Retourne la distance minimum entre les cases id1 et id2.

    /// Méthodes de démonstration
    // Matérialise un chemin de longueur minimale entre les cases d'identifiants
    // id1 et id2 en plaçant des valeurs 2 dans les cases de ce chemin.
    void chemin(int id1, int id2);
    // Lance une démonstration de productions de labyrinthes et de recherche de
    // chemins de longueur minimale.
    static void demo();
};
```

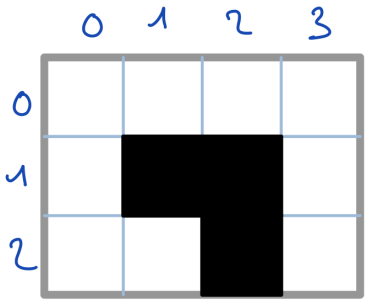
Le constructeur à deux paramètres crée une grille remplie de cases blanches dont le nombre de lignes (largeur) et de colonnes (longueur) sont passés en paramètres.

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.




```
char tab[] =
{ 3, 4,
  0, 0, 0, 0
  0, 1, 1, 0
  0, 0, 1, 0
};
```

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).

La méthode **`afficheDescr`** affiche le descripteur du labyrinthe courant. La méthode de classe **`demo`** lance une démonstration. La méthode **`chemin`** recherche un chemin optimal entre deux cases et place des valeur 2 dans la grille dans toutes les cases de ce chemin.

Il y a un changement dans la manière de spécifier les caractères utilisés par la méthode **`affiche`**. Ces caractères sont rassemblés dans une chaîne de caractères spécifiée grâce à la méthode **`setAff`**. Cette méthode accepte en paramètre une chaîne au format C appelée **`motifs`**. Si n est la longueur de cette chaîne, alors les caractères **`motifs[0]`** à **`motifs[n-2]`** seront utilisés pour représenter les valeurs 0 à n-2 de la grille, c'est-à-dire les cases vide (0), les mur (1), et tout autres objets pouvant être représentés par les valeurs 2 à n-2 (comme par exemple des robots) si applicable. Le caractère **`motifs[n-1]`** est utilisé pour l'affichage des bords. Par exemple, la méthode **`demo`** affiche des labyrinthes dont mes bords sont représentés par des points **`'.'`**, les cases vides par des espaces **`' '`**, les murs par des **`'-'`** et les cases ayant la valeur 2 (qui matérialisent dans la démonstration les chemins optimaux traversant le labyrinthe) par la lettre **`'A'`**. Ceci est obtenu par un appel de **`setAff(" -A.")`**.

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 est le poursuivant. Vous devez au moins implémenter un algorithme nommé **direct prédateur** : Soit E l'ensemble des cases vides voisines de la position actuelle du robot A. la nouvelle position est choisie au hasard parmi les cases de E qui minimisent la distance (calculé par la méthode **`distMin`**) avec le robot B. (Chacune des cases de E doit avoir la même probabilité d'être choisie.)

Le robot B doit essayer d'échapper au robot A. Son algorithme de déplacement n'est pas imposé mais vous devez à minima proposer au moins une version appelée **random proie** dans laquelle le robot se déplace vers une case choisie au hasard parmi ses cases voisines vides. 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).

Les fonctions d'affichage et de déroulement de la poursuite vous sont fournies, de même qu'une fonction mesurant le nombre médian d'étapes lors des poursuites. Soyez attentif aux annonces faites à ce sujet via l'environnement collaboratif TEAMS et aux mises à jour du padlet des ressources pédagogiques (identifié par un kiwi).

Barème

Un **code source propre**, respectant les consignes, implémentant correctement les algorithmes **direct prédateur** et **random proie**, accompagné (sur le document pdf), d'une explication de votre implémentation et des rôles des méthodes et /ou attributs que vous avez ajoutés, représente une note de **4 points**.

Les élèves travaillant en binôme devront automatiser les tests des méthodes de déplacement des robots A et B avec les algorithmes **direct prédateur** et **random proie**. Les vérifications se font sur un seul mouvement dans des situations de test prédéfinies, par exemple dans un des labyrinthes dont les descripteurs sont fournis. Les points à vérifier sont que les mouvements sont cohérents avec les spécifications des algorithmes et que la détection de collision (les robots sont dans des cases voisines) fonctionne correctement. Il faut prévoir plusieurs situations de test pour lesquelles on positionne les robots avant appel des méthodes à tester.

Pour obtenir les **5 points** de la partie 3, vous devrez proposer et implémenter un **algorithme de proie** plus performant que le random proie et qui doit avoir au minimum un score médian de 100 sur une grille sans mur avec départ des robots à des positions aléatoires. Cet algorithme pourra être proposé après la date limite de livraison de la partie 3. Des tests unitaires automatisés ne sont pas demandé, même pour les binômes.

La proposition et l'implémentation d'algorithmes originaux et performants pour le robot proie et/ou le robot prédateur pourront être gratifiés comme des contributions personnelles.

Livrables

Vous devez livrer

- un unique fichier source C++ respectant scrupuleusement les consignes spécifiées à la fin de ce document,
- un document pdf d'au plus 4 pages expliquant présentant les choix d'implémentation que vous avez faits pour les méthodes de mouvement des robots, ainsi que les mesures de performances que vous avez réalisées et leurs résultats.

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 les méthodes `distMin`, `chemin`, `afficheDescr`, `demo`**. 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 ayant eu au moins 4 points à la partie 3.

Vous devez concevoir et implémenter la classe **`Labyrinthe`** de la partie 3 sans les méthodes **`chemin`, `afficheDescr`, `demo`,** mais **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.

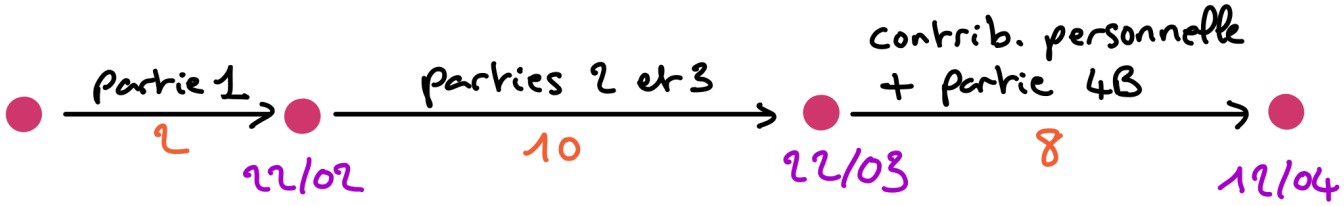
Rappel des objectifs des parties

- 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)

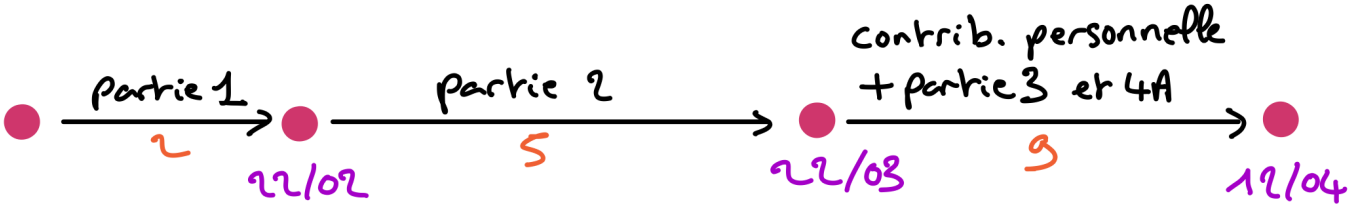
Si vous choisissez ce parcours, le plus difficile, vous devrez livrer les parties 1, 2, 3 et 4B, en cas de retard, vous pourrez adopter un parcours moins difficile mais qui pourra tout de même vous permettre d'avoir une bonne note.



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.

Le nom de l'auteur ou les noms des auteurs et la ligne de commande permettant la compilation doivent être mis 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 lettre 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).