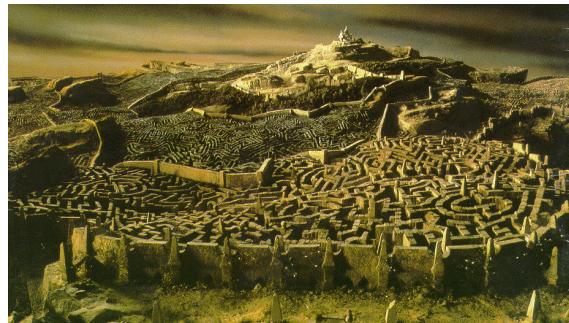


# Projet labyrinthe



*Figure 1 : Labyrinthe*

## 1 Présentation

Le projet se décompose en deux phases, correspondant chacune approximativement à une semaine de travail. La première phase est commune à tous les groupes et met en place des outils permettant de manipuler des images et d'interagir avec l'utilisateur. Cette phase est ponctuée de petits exercices assez libres où vous devrez démontrer votre maîtrise des points abordés. La seconde phase, plus spécifique à chaque groupe, consiste à réinvestir les acquis de la première phase afin de créer un prototype de votre propre « jeu vidéo », dans une thématique 'labyrinthe'.

# Firefox ne peut pas ouvrir cette page

Pour protéger votre sécurité, perso.isima.fr ne permettra pas à Firefox d'afficher un autre site. Pour voir cette page, vous devez l'ouvrir dans une nouvelle fenêtre

[En savoir plus...](#)

[Ouvrir le site dans une nouvelle fenêtre](#)

Signaler les erreurs similaires pour aider Mozilla à identifier et bloquer les sites malveillants

Figure 2 : Regarder depuis le début, puis sauter à 4'45"

## 1.1 Synopsis pédagogiques

### 1.1.1 Gestion de projet

Votre groupe doit mener à bout le travail, vous devez donc créer une synergie de groupe qui vous permettra d'obtenir des livrables de qualité maximale, dans une ambiance de travail productive. Le projet se déroule sur un temps plus long que celui d'un TP, vous avez donc un peu de temps pour vous tromper mais pour parvenir à un rendu de qualité, une bonne gestion de votre temps s'impose.

Vous **devez** également apprendre à utiliser [Git...](#) et l'utiliser de façon

efficace.



Figure 3 : Logo git

Vous devrez de plus maintenir au fil de votre projet une page web sur le site de l'ISIMA. Cette page devra proposer un lien sur votre dépôt *git*, des informations sur l'état du projet, et servira de support lors des deux présentations. Il est important que votre page soit tenue à jour, au minimum de façon journalière.

#### Attention

- vous devez pouvoir présenter des fiches/diagrammes de répartition des tâches (prévisionnelles, effectives) à jour à tout moment,
- chacun doit créer **plusieurs commits par jour** et au moins un **push chaque jour**,
- les branches (*git*) doivent être utilisées : à chaque nouvelle fonctionnalité correspond au moins une branche.

### 1.1.2 Programmation

Chacun d'entre-vous doit à la fin de ce projet avoir atteint un niveau minimal dans la qualité de son code, et en particulier, vous devrez :

1. Maîtriser la syntaxe du c,
2. Savoir utiliser à bon escient des bibliothèques standards,
3. Savoir rechercher rapidement de l'information dans le man / sur le net,
4. Comprendre l'utilisation des makefiles,
5. Savoir documenter votre code : code le plus possible **auto-documenté** (les noms de variable, la présentation du code sont si

lumineux que d'autres explications sont superflues) qu'on pourra compléter si nécessaire par des fichiers d'explications en [org mode](#) ou [markdown](#) et potentiellement [doxygen](#),

6. Savoir réaliser une page web présentant votre travail.

### 1.1.3 Algorithmique

Au cours du TP vous allez réinvestir vos acquis en algorithmique ainsi qu'en structures de données. Le projet utilisera au minimum des listes, tableaux, arbres, graphes et vous demandera de les manipuler et parcourir de façon adaptée.

Dans la partie obligatoire, vous découvrirez deux nouvelles structures de données :

- les tas, qui sont très utiles pour gérer des files de priorité, et que l'on utilisera en particulier dans des algorithmes de parcours de graphe afin de déterminer efficacement le prochain nœud à explorer,
- les partitions, qui nous serviront dans la suite à implémenter des algorithmes de graphe et en particulier à construire notre labyrinthe.

Dans la partie libre, plusieurs voies vous seront suggérées, certaines approfondissant la thématique graphe, d'autres vous proposant de découvrir la théorie des jeux (algorithmes min/max et ses raffinements).

### 1.1.4 Outils pour les interfaces

Le projet doit vous permettre de découvrir ou d'approfondir vos connaissances dans le domaine de la programmation graphique 2D. Vous devrez notamment :

1. (Re?-)Découvrir la SDL2,
2. Créer une image par juxtaposition et superposition d'autres images,
3. Gérer la transparence dans la superposition de deux images,
4. Créer une animation simple,
5. Gérer une boucle d'évènements,
6. Gérer les entrées utilisateur.

## **1.2 Évaluation**

Le projet est évalué avec les mentions suivantes : TB+F (Très Bien avec Félicitations du jury), TB (Très Bien), B (Bien), AB (Assez Bien), DFSP (Doit Faire Ses Preuves), I (Insuffisant).

Le projet est validé si la mention obtenue est au moins *Assez Bien*.

L'évaluation prendra notamment en compte les points suivants :

- Sérieux, persévérance, implication, quantité de travail, régularité,
- Git : 'commits' et 'push' réguliers,
- Web : régularité de la maintenance et pertinence des informations,
- Collaboration dans le groupe,
- Qualité du travail (codes, documentations, rendus, utilisation des outils),
- Bonus pour les étudiants ayant réalisé des formations.

Le résultat de l'évaluation ne sera pas nécessairement identique pour tous les membres d'un groupe.

Certains projets ou éléments de projets seront conservés afin d'être utilisés à fin de démonstration ou comme base de futurs travaux.

## **1.3 Présentations du travail**

L'évaluation se fera sur la durée des deux semaines, avec en particulier deux temps fort où vous réaliserez une soutenance de votre travail, chacun des deux vendredis.

La durée prévue de la première soutenance sera de 10 minutes (6 minutes de présentation, 4 pour les questions), celle de la seconde de 15 minutes (10 minutes de présentation, 5 pour les questions).

Dans l'éventualité où la première soutenance n'aurait pas été concluante, une troisième présentation pourra être imposée le lundi de la deuxième semaine.

## **1.4 Groupes**

Le travail est à réaliser en groupes libres de **trois** étudiants, les groupes ne doivent pas nécessairement respecter les groupes classes habituels (un groupe peut mixer un G31 avec deux G11...).

Si des étudiants ne parviennent pas à compléter leur groupe, les enseignants pourront se charger de le faire.

## **1.5 Présence obligatoire**

La présence est obligatoire de 9h à 12h et de 14h à 17h du lundi au vendredi pendant les deux semaines de projet.

Les deux vendredis (18/06 et 25/06), il est possible que votre groupe ait une soutenance en dehors des plages horaires indiquées ci-dessus.

Il n'est pas interdit d'avancer le projet en dehors des créneaux indiqués ;-).

## **1.6 Formations**

Au fil du projet, des formations pourront être réalisées par des étudiants sur des points particuliers. Ces formations se dérouleront vraisemblablement sur MS-Teams. La liste des sujets actuellement proposée est :

### **HTML**

Présentation de ce langage à balises, de façon à ce que chacun sache créer une page, la déposer chez un hébergeur (en particulier sur le site de l'ISIMA).

### **css**

Les feuilles de style, ne plus en avoir peur !

### **makefiles, rappels sur les phases de la compilation**

Chacun est sensé connaître les étapes de la compilation, et donc par exemple comprendre de quoi les 'header guards' protègent... Rappel sur les makefiles, et proposition de quelques astuces afin de les rendre un petit peu génériques.

### **git les rudiments (vraiment light)**

Le but est ici de proposer une présentation très brève de git qui aura lieu en tout début du projet et donc de ne donner qu'un minimum de commandes, afin que chacun puisse commencer à travailler, et surtout que personne ne prenne peur de cet outil. Chaque étudiant devra prendre l'habitude de réaliser de multiples commits journaliers (chacun contenant un faible nombre de modifications), et un minimum de un ou deux push chaque jour.

### **git pour aller plus loin**

Cette présentation devra s'avancer un peu plus loin dans les terres de git, en présentant par exemple comment il est possible d'annuler des opérations passées.

### **doxygen**

L'auto-documentation du code c'est bien (un code si bien organisé, avec des noms de variables explicites, etc... que son fonctionnement en est parfaitement lumineux), mais cela ne suffit pas toujours.

Doxxygen est un outil facile d'accès qui permet d'aider à la construction de la documentation technique d'un projet.

### **gdb – ddd – valgrind**

Suite à une erreur de segmentation, quel outil va permettre de tracer son origine ? gdb–ddd ont été utilisés avec plus ou moins de douleur pendant les TPs de SDD, il est temps de partager quelques bonnes idées ! Et les portes du Valhalla, quand est-il pertinent de les ouvrir ?

### **graphviz**

Graphviz, couplé à un moteur de rendu est un outil permettant d'avoir accès dans à peu près n'importe quel environnement à la conception de graphes, arbres, diagrammes.

## **1.7 Phases du projet**

### **1.7.1 Deux phases**

Durant la première semaine du projet, les apprentissages sont guidés. Une fois tous les exercices réalisés, il vous sera possible d'orienter ce projet dans une direction qui vous est plus personnelle, tout en restant dans la thématique proposée.

## 1.7.2 Personnalisation des exercices

Les exercices demandés acceptent des variations, acceptables tant qu'elles vous permettent de démontrer votre savoir-faire sur le point étudié. Pensez cependant à faire valider vos propositions de variations par un enseignant afin :

- de ne pas vous lancer dans un travail trop chronophage par rapport à la version originale,
- de ne pas avoir manqué un point particulier que les enseignants désirent vous voir démontrer.

## 1.7.3 Les jalons prévus

- la SDL (ouvrir une fenêtre, dessiner, poser une image, ...):
  - application finale : jeu de la vie, avec l'extension : [création d'un labyrinthe en modifiant les règles de transition](#),
- la SDL (interaction avec l'utilisateur):
  - application : space numbers,
- File de priorité:
  - tas binaire (ou binomial, ou fibonacci),
  - tri par tas,
- Type de donnée partition, composantes connexes :
  - application : création d'un labyrinthe arborescent,
  - extension : création d'un labyrinthe,
- Plus court chemin :
  - si on connaît le labyrinthe : algorithme de Dijkstra, A\*,
  - si on ne le connaît pas et que le coût à minimiser est le nombre de déplacements,
- Extensions proposées :
  - apparition des monstres (comportement des monstres, comportement du joueur),
  - positionnement optimal de ressources,

- jeu à deux joueurs où un adversaire peut ajouter un mur à chaque « tour » de jeu,
- ...

## 2 SDL2

### 2.1 Préliminaire

Vous avez déjà eu un aperçu de la SDL2, ne vous privez pas d'y retourner : [flood-it](#).

La construction des outils se fera en réalisant de petits exercices qui mettront en œuvre les outils que vous développerez. Ces mini applications vous serviront de tests fonctionnels, afin de vérifier l'utilisabilité pratique des codes créés.

Les explications qui suivent ne se substituent absolument pas à la consultation de la documentation du [wiki](#), mais pourra vous aider à mettre le pied à l'étrier.



*Figure 4 : Logo Simple DirectMedia Layer*

Votre première tâche consiste à découvrir la SDL2. Vous devez démontrer au fur et à mesure des exercices que vous savez :

- Afficher une image qui servira de fond,
- Incruster une image (sprite), en gérant la transparence,
- Déplacer le sprite selon les appuis de touche de l'utilisateur,
- Déplacer le sprite selon une trajectoire fournie par une fonction,
- Tracer des dessins.

## Points importants pour la suite

- Chargement propre des composants utiles de la SDL2,
- Libération de la mémoire,
- Travail modulaire afin de favoriser la ré-utilisabilité.

## 2.2 Installation de la SDL

- Installer la bibliothèque SDL2 : `libsdl2-dev`, à laquelle il faudra vraisemblablement ajouter des modules complémentaires (installer tout `libsdl2-*` est, je pense, la bonne solution).
- Recopier le code suivant :

```
#include <SDL2/SDL.h>
#include <stdio.h>

/*****************/
/* Vérification de l'installation de la SDL */
/*****************/

int main(int argc, char **argv) {
    (void)argc;
    (void)argv;
    SDL_version nb;
    SDL_VERSION(&nb);

    printf("Version de la SDL : %d.%d.%d\n", nb.major, nb.minor, nb.patch);
    return 0;
}
```

Pour compiler/linker ce programme, il est nécessaire d'inclure les bibliothèques nécessaires :

`gcc verif SDL.c -o verif(SDL)` (ou  
`gcc verif(SDL.c -o verif(SDL -lSDL2)` [auxquels vous ajouterez les drapeaux habituels `-Wall -Wextra` et vraisemblablement `~-Og -g~`].

L'exécution de `sdl2-config` avec les options voulues permet de ne pas indiquer 'à la main' les options, mais de faire appel à un script qui choisit les options, en particulier en les adaptant selon le système d'exploitation.

## 2.3 Première fenêtre

La création d'une première fenêtre va se faire en plusieurs étapes :

1. Initialisation des modes utilisables par la suite (vidéo, son, vibrations, ...) `SDL_Init` [il sera possible d'activer par la suite d'autres modules si nécessaire par d'autres fonctions d'initialisation],
  1. Création de la fenêtre : la fenêtre est le conteneur dans lequel on placera des éléments par la suite. Une bonne représentation mentale pour faire la distinction avec d'autres éléments qui suivront, est de penser la fenêtre comme la partie qui interagit avec l'interface graphique du système. La fonction de création d'une fenêtre est `SDL_CreateWindow`. Ses arguments permettent de choisir
    - son titre
    - sa position à la création
    - ses dimensions
    - plein écran / maximisée / réduite
    - visible / cachée
    - ...
  2. Libérer le pointeur sur la fenêtre,
2. Fermer l'utilisation de la SDL `SDL_Quit`

```

#include <SDL2/SDL.h>
#include <stdio.h>

/*****************/
/* exemple de création de fenêtres */
/*****************/

int main(int argc, char **argv) {
    (void)argc;
    (void)argv;

    SDL_Window *window_1 = NULL,           // Future fenêtre de ga
                  *window_2 = NULL;          // Future fenêtre de dr

    /* Initialisation de la SDL + gestion de l'échec possible */
    if (SDL_Init(SDL_INIT_VIDEO) != 0) {
        SDL_Log("Error : SDL initialisation - %s\n", SDL_GetError());
        exit(EXIT_FAILURE);
    }

    /* Création de la fenêtre de gauche */
    window_1 = SDL_CreateWindow(
        "Fenêtre à gauche",           // codage en utf8, donc
        0, 0,                         // coin haut gauche en
        400, 300,                      // largeur = 400, hauteur
        SDL_WINDOW_RESIZABLE);         // redimensionnable

    if (window_1 == NULL) {
        SDL_Log("Error : SDL window 1 creation - %s\n", SDL_GetError());
        SDL_Quit();
        exit(EXIT_FAILURE);
    }

    /* Création de la fenêtre de droite */
    window_2 = SDL_CreateWindow(
        "Fenêtre à droite",          // codage en utf8, donc
        400, 0,                       // à droite de la fenêtre
        500, 300,                      // largeur = 500, hauteur
        0);

    if (window_2 == NULL) {
        /* L'init de la SDL : OK
         * fenêtre 1 :OK
         * fenêtre 2 : échec */
        SDL_Log("Error : SDL window 2 creation - %s\n", SDL_GetError());
        SDL_DestroyWindow(window_1);
        SDL_Quit();
        exit(EXIT_FAILURE);
    }

    /* Normalement, on devrait ici remplir les fenêtres... */
    SDL_Delay(2000);                // Pause exprimée en ms
}

```

```

/* et on referme tout ce qu'on a ouvert en ordre inverse de la cr e
SDL_DestroyWindow(window_2);
SDL_DestroyWindow(window_1);

SDL_Quit();
return 0;
}

```

## 2.4 Travail  r aliser : un X fen tr 



*Figure 5 : X-Men (affiche film dark phoenix)*

Vous devez mettre en  vidence par un code votre aptitude  ouvrir et fermer des fen tres. Un r sultat possible de votre travail pourrait  tre : [./all\\_executables.tar.gz](#) (t l charger, ex cuter 'X\_fenetre'). N'h sitez pas  demander  un enseignant si ce que vous vous proposez de r aliser correspond  la difficult  recherch e.

Remarques :

- la fonction

```
void SDL_SetWindowPosition(SDL_Window * window, int x, int y)
```

- permet de positionner une fenêtre,  
• la fonction  

```
void SDL_GetWindowPosition(SDL_Window * window, int *x, int *y)
```

permet de récupérer la position d'une fenêtre,
- la fonction  

```
void SDL_GetWindowSize(SDL_Window * window, int *w, int *h)
```

permet de récupérer les dimensions d'une fenêtre,
- la fonction  

```
int SDL_GetCurrentDisplayMode(int displayIndex, SDL_DisplayMode * mode)
```

permet de récupérer les dimensions de l'écran.

## 2.5 Mettre un 'rendu' dans une fenêtre

Un rendu/renderer est une zone dans laquelle il sera possible de dessiner ou de poser des images. Un rendu se 'dépose' dans une fenêtre, on peut l'imaginer comme une toile que l'on placerait sur un chevalet (le chevalet étant la fenêtre), cette toile étant munie de tout l'équipement qui permet de la peindre. La fenêtre crée un lien avec l'interface graphique du système, le rendu un lien entre la fenêtre et ce qu'on y affiche. Par la suite on apprendra à peindre la toile, ou à y coller des images.

D'un point de vue plus technique, le rendu va nous permettre de définir quelles fonctions vont être utilisées pour réaliser la visualisation (utilisation ou non des accélérations de la carte graphique et autres options connues des utilisateurs de jeux vidéo telles que l'activation de la synchronisation verticale, etc...). Le rendu est alors pensé comme un *moteur d'affichage*.

Pour créer rendu on utilise la fonction `SDL_CreateRenderer`, et pour le détruire `SDL_DestroyRenderer`. Une fois un rendu créé (ou par la suite modifié), il est nécessaire de réaliser son affichage, fonction `SDL_RenderPresent`. Dans l'éventualité où on désire effacer un rendu, on utilise la fonction `SDL_RenderClear`, qui va 'repeindre' la surface par une couleur de notre choix (noir par défaut).

## 2.6 Dessiner sur un rendu

Pour dessiner sur le renderer, on peut par exemple :

- tracer un point `SDL_RenderDrawPoint`
- tracer une ligne `SDL_RenderDrawLine`
- tracer un rectangle `SDL_RenderDrawRectangle`
- tracer un rectangle plein `SDL_RenderFillRect`
- changer de couleur `SDL_RenderDrawColor`

Il existe également des variantes de ces fonctions permettant de tracer en une seule commande plusieurs points, plusieurs lignes, etc...

```

#include <SDL2/SDL.h>
#include <math.h>
#include <stdio.h>
#include <string.h>

//*****************************************************************************
/*                                         Programme d'exemple de création de r
//****************************************************************************

void end_sdl(char ok,
             char const* msg,
             SDL_Window* window,
             SDL_Renderer* renderer) {
    char msg_formated[255];
    int l;

    if (!ok) {
        strncpy(msg_formated, msg, 250);
        l = strlen(msg_formated);
        strcpy(msg_formated + l, " : %s\n");

        SDL_Log(msg_formated, SDL_GetError());
    }

    if (renderer != NULL) SDL_DestroyRenderer(renderer);
    if (window != NULL)   SDL_DestroyWindow(window);

    SDL_Quit();
}

if (!ok) {
    exit(EXIT_FAILURE);
}
}

void draw(SDL_Renderer* renderer) {
    SDL_Rect rectangle;

    SDL_SetRenderDrawColor(renderer,
                          50, 0, 0,
                          255);
    rectangle.x = 0;
    rectangle.y = 0;
    rectangle.w = 400;
    rectangle.h = 400;

    SDL_RenderFillRect(renderer, &rectangle);

    SDL_SetRenderDrawColor(renderer, 0, 0, 0, 255);
    SDL_RenderDrawLine(renderer,
                      0, 0,
                      400, 400);
}

```

```

/* tracer un cercle n'est en fait pas trivial, voilà le résultat si on fait ça
for (float angle = 0; angle < 2 * M_PI; angle += M_PI / 4000) {
    SDL_SetRenderDrawColor(renderer,
        (cos(angle * 2) + 1) * 255 / 2,
        (cos(angle * 5) + 1) * 255 / 2,
        (cos(angle) + 1) * 255 / 2,
        255);
    SDL_RenderDrawPoint(renderer,
        200 + 100 * cos(angle),
        200 + 150 * sin(angle));
}
}

int main(int argc, char** argv) {
    (void)argc;
    (void)argv;

    SDL_Window* window = NULL;
    SDL_Renderer* renderer = NULL;

    SDL_DisplayMode screen;

    *****
    /* Initialisation de la SDL + gestion de l'erreur */
    if (SDL_Init(SDL_INIT_VIDEO) != 0) end_sdl(0, "ERROR SDL INIT", window);

    SDL_GetCurrentDisplayMode(0, &screen);
    printf("Résolution écran\n\tw : %d\n\tth : %d\n", screen.w,
        screen.h);

    /* Création de la fenêtre */
    window = SDL_CreateWindow("Premier dessin",
        SDL_WINDOWPOS_CENTERED,
        SDL_WINDOWPOS_CENTERED, screen.w * 0.66,
        screen.h * 0.66,
        SDL_WINDOW_OPENGL);
    if (window == NULL) end_sdl(0, "ERROR WINDOW CREATION", window);

    /* Création du renderer */
    renderer = SDL_CreateRenderer(
        window, -1, SDL_RENDERER_ACCELERATED | SDL_RENDERER_PRESENTVSYNC);
    if (renderer == NULL) end_sdl(0, "ERROR RENDERER CREATION", window);

    *****
    /* On dessine dans le renderer */
    draw(renderer);
    SDL_RenderPresent(renderer);
    SDL_Delay(1000);
}

```

```

*****
/* on referme proprement la SDL */
end_sdl(1, "Normal ending", window, renderer);
return EXIT_SUCCESS;
}

```

### Remarques

- Une fonction de terminaison `end_sdl` assez brutale est apparue (il y a des `exit` alors que ce n'est pas le `main`!), à vous de voir comment gérer au mieux les erreurs avec la SDL... en attendant la ZZ2 pour avoir un mécanisme propre de gestion des exceptions en c++.
- Pour apprendre à tracer des cercles, et en général à faire du dessin sur ordinateur, ma référence reste l'ouvrage de Michael Abrash [Zen de la programmation graphique / Graphics programing handbook](#) dont je vous recommande la lecture.

## 2.7 Travail à réaliser : pavé de serpents

Le but est ici de démontrer vos talents pour réaliser une animation uniquement basée sur des tracés de figures simples. Une réalisation possible serait [./all\\_executables.tar.gz](#) (télécharger, exécuter 'snake').

### Remarques

- après avoir dessiné quelque chose, il faut penser à l'afficher en utilisant `void SDL_RenderPresent(SDL_Renderer * renderer)` suivi d'un `void SDL_Delay(Uint32 ms)` d'au moins (environ) 10ms.
- Si on désire effacer la fenêtre avec `int SDL_RenderClear(SDL_Renderer * renderer)`, il faut préalablement avoir choisi avec `int SDL_SetRenderDrawColor(SDL_Renderer * renderer, Uint8 r, Uint8 g, l` la couleur dans laquelle le fond devra être peint.

## 2.8 Travail à réaliser : jeu de la vie



## LIFE: THE GAME

Great Graphics, No Lag, But the Gameplay Was Awful

VERY DEMOTIVATIONAL .com

Pour mettre en évidence votre maîtrise du dessin, vous allez coder un [Jeu de la vie](#) (ou par [Etienne Ghys](#)). Vous travaillerez avec une grille de taille fixe (par exemple du 30 par 50). La grille aura la forme d'un tore (anneau), c'est à dire que la case à droite d'une case du bord droit sera la case à gauche de la même ligne (et inversement), et la case en haut d'une case du bord haut sera la case en bas de la même colonne (et inversement).

La configuration de départ sera 'en dur' dans le code (lorsqu'on aura abordé la gestion de la souris, on pourra ajouter une fonction de saisie de la configuration initiale). Une réalisation possible serait [./all\\_executables.tar.gz](#) (télécharger, exécuter 'vie'), qui est en boucle infinie (on ne sait pas encore arrêter un programme suite à un événement...).

Il est intéressant de coder les règles du jeu de la vie en utilisant deux tableaux de 9 booléens qui permettent de stocker les règles particulières de ce jeu de la vie ([règles alternatives](#)) :

- tableau 'survie' : si la cellule en cours d'examen est vivante et

- `survie[nb_voisins_vivants] == Faux`, alors la cellule meurt,
- tableau 'naissance' : si la cellule en cours d'examen est morte et `naissance[nb_voisins_vivants] == Vrai`, alors la cellule naît.

## 2.9 Manipuler des textures

Une *texture* est une structure qui peut contenir une image, ainsi que les informations *largeur* et *hauteur* sur l'image.

### 2.9.1 Chargement d'une image

La création d'une texture commence fréquemment par le chargement d'une image. Pour cela, la procédure la plus classique consiste à repasser temporairement en SDL1 :

- on crée une surface (type de données SDL1)
- on charge l'image dans la `SDL_Surface`,
- on transforme la surface en texture (type de données correspondant en SDL2)

#### Remarque

- la SDL ne fonctionne par défaut qu'avec le format bmp... Afin de pouvoir utiliser d'autres formats, il est nécessaire d'ajouter une bibliothèque `#include <SDL2/SDL_image.h>`. Il sera alors nécessaire de compiler avec le drapeau `-lSDL2_image`.

```
#include <SDL2/SDL_image.h>
```

On peut écrire une fonction qui charge une image dans une texture :

```

#include <SDL2/SDL_image.h>           // Nécessaire pour la fo
                                         // Penser au flag -lsdl2
//...

SDL_Texture* load_texture_from_image(char * file_image_name, SDL_W
    SDL_Surface *my_image = NULL;          // Variable de passage
    SDL_Texture* my_texture = NULL;         // La texture

    my_image = IMG_Load(file_image_name);   // Chargement de l'image
                                         // image=SDL_LoadBMP(fil
                                         // uniquement possible s
if (my_image == NULL) end_sdl(0, "Chargement de l'image impossib

    my_texture = SDL_CreateTextureFromSurface(renderer, my_image); /
    SDL_FreeSurface(my_image);             /
if (my_texture == NULL) end_sdl(0, "Echec de la transformation d

    return my_texture;
}

//...
IMG_Quit()                           // Si on charge une libr

```

Remarque :

- le paramètre `SDL_Window *window` ne sert qu'en cas de problème à fermer la fenêtre,

On peut préférer à cette méthode 'traditionnelle' :

```

#include <SDL2/SDL_image.h>
// ...
SDL_Texture my_texture;
my_texture = IMG_LoadTexture(renderer, "./img/Maze.png");
if (my_texture == NULL) end_sdl(0, "Echec du chargement de l'image
//...
IMG_Quit()

```

Dans tous les cas, il ne faut alors pas oublier la contrepartie à la création d'une texture : sa libération...

```

SDL_DestroyTexture(my_texture);

```

## 2.9.2 Affichage d'une texture sur la totalité de la fenêtre

Une fois une texture créée, on peut désirer l'afficher, c'est la fonction `SDL_RenderCopy` qui va copier la texture, ou une partie de la texture à l'endroit indiqué dans le renderer.

```
void play_with_texture_1(SDL_Texture *my_texture, SDL_Window *window
                           SDL_Renderer *renderer) {
    SDL_Rect
        source = {0},                                // Rectangle définis
        window_dimensions = {0},                      // Rectangle définis
        destination = {0};                           // Rectangle définis

    SDL_GetWindowSize(
        window, &window_dimensions.w,
        &window_dimensions.h);                      // Récupération des
    SDL_QueryTexture(my_texture, NULL, NULL,
                     &source.w, &source.h);                // Récupération des

    destination = window_dimensions;               // On fixe les dimen

    /* On veut afficher la texture de façon à ce que l'image occupe la
     * totalité de la fenêtre */

    SDL_RenderCopy(renderer, my_texture,
                  &source,
                  &destination);                    // Création de l'élé
    SDL_RenderPresent(renderer);                  // Affichage
    SDL_Delay(2000);                            // Pause en ms

    SDL_RenderClear(renderer);                  // Effacer la fenêtr
}
```

Remarque :

- Il ne faut pas oublier d'afficher avec `SDL_RenderPresent` et de mettre une pause `SDL_Delay` si on espère voir quelque chose.

## 2.9.3 Affichage d'une partie d'une texture à un endroit choisi

```

void play_with_texture_2(SDL_Texture* my_texture,
                        SDL_Window* window,
                        SDL_Renderer* renderer) {
    SDL_Rect
        source = {0},                                // Rectangle définissant la source
        window_dimensions = {0},                      // Rectangle définissant les dimensions de la fenêtre
        destination = {0};                           // Rectangle définissant la destination

    SDL_GetWindowSize(
        window, &window_dimensions.w,
        &window_dimensions.h);                      // Récupération des dimensions de la fenêtre
    SDL_QueryTexture(my_texture, NULL, NULL,
                     &source.w, &source.h); // Récupération des dimensions de la texture

    float zoom = 1.5;                            // Facteur de zoom à appliquer
    destination.w = source.w * zoom;            // La destination est agrandie
    destination.h = source.h * zoom;            // La destination est agrandie
    destination.x =
        (window_dimensions.w - destination.w) /2; // La destination est centrée
    destination.y =
        (window_dimensions.h - destination.h) / 2; // La destination est centrée

    SDL_RenderCopy(renderer, my_texture,      // Préparation de l'affichage
                  &source,
                  &destination);
    SDL_RenderPresent(renderer);
    SDL_Delay(1000);

    SDL_RenderClear(renderer);                  // Effacer la fenêtre
}

```

### Remarque

- Ici, la source a pris la totalité de la texture, en modifiant les quatre paramètres du rectangle `source`, il aurait été possible de n'en prendre qu'une partie (obligatoirement rectangulaire).

### 2.9.4 Créer une première animation

L'idée ici est de déposer successivement la texture à différents endroits de l'écran, en effaçant l'écran entre chaque affichage.

```

void play_with_texture_3(SDL_Texture* my_texture,
                         SDL_Window* window,
                         SDL_Renderer* renderer) {
    SDL_Rect
        source = {0},                                // Rectangle défini
        window_dimensions = {0},                     // Rectangle défini
        destination = {0};                          // Rectangle défini

    SDL_GetWindowSize(
        window, &window_dimensions.w,
        &window_dimensions.h);                      // Récupération de
    SDL_QueryTexture(my_texture, NULL, NULL,
                     &source.w,
                     &source.h);                        // Récupération de

    /* On décide de déplacer dans la fenêtre cette image */
    float zoom = 0.25;                            // Facteur de zoom

    int nb_it = 200;                             // Nombre d'images
    destination.w = source.w * zoom;             // On applique le
    destination.h = source.h * zoom;             // On applique le
    destination.x =
        (window_dimensions.w - destination.w) / 2; // On centre en largeur
    float h = window_dimensions.h - destination.h; // hauteur du déplacement

    for (int i = 0; i < nb_it; ++i) {
        destination.y =
            h * (1 - exp(-5.0 * i / nb_it)) / 2 *
            (1 + cos(10.0 * i / nb_it * 2 *
                      M_PI));                      // hauteur en fonction de i

        SDL_RenderClear(renderer);                  // Effacer l'image

        SDL_SetTextureAlphaMod(my_texture, (1.0 - 1.0 * i / nb_it) * 255); // Alpha
        SDL_RenderCopy(renderer, my_texture, &source, &destination); // Copie
        SDL_RenderPresent(renderer);                // Affichage de la fenêtre
        SDL_Delay(30);                           // Pause en ms
    }
    SDL_RenderClear(renderer);                  // Effacer la fenêtre
}

```

Remarque :

- il y a un *fade-out* dans cette animation : l'objet disparaît petit à petit à la fin de l'animation. Pour obtenir ce résultat, `SDL_SetTextureAlphaMod` est utilisée. Cette fonction permet d'appliquer un coefficient de transparence à une texture (0 pour

invisible, 255 pour opaque). S'il y avait un fond, il serait plus ou moins visible en fonction de la transparence choisie.

### 2.9.5 Et si on veut animer un 'sprite' ?

Les planches des *sprites* utilisées dans les animations sont construites en positionnant un objet dans ces différentes positions dans une même image (on peut appeler ces positions des vignettes). Il est important que :

- les positions soient placées intelligemment dans l'image : les vignettes sont si possible de la même taille, espacées régulièrement. L'objectif est de permettre de parcourir séquentiellement aisément les différentes vignettes qui constituent l'animation en déplaçant le rectangle `source` sur l'image de façon régulière,
- le fond de l'image soit transparent : sur les formats type *png* un point est codé par 4 valeurs (de 0 à 255) (*Rouge, Vert, Bleu, Transparence*). Cette dernière composante est à 0 pour une parfaite transparence, et à 255 pour une parfaite opacité. Il est aisément d'incorporer des *sprites* dont le fond est transparent à n'importe quel décor. Dans l'éventualité où le fond du *sprite* serait uni, il y a plusieurs possibilités :
  - reprendre l'image avec un logiciel de dessin (*gimp* par exemple, présent sur la plupart des systèmes Linux), l'outil permet même de réaliser un [détourage](#) avec un dégradé de transparence sur le bord du sujet afin qu'il se fonde au mieux dans n'importe quel décor,
  - utiliser des fonctions de la *SDL* qui permettent de choisir une ou plusieurs couleurs comme étant en réalité transparentes (c'est souvent ce que l'on fait avec des images au format *bmp* qui ne gèrent pas dans leur format la transparence),
  - utiliser un masque de transparence : on crée une image supplémentaire, en intensité de gris qui va servir, de façon totalement similaire à la 4<sup>ème</sup> composante du *png*, à indiquer la transparence. Cette méthode, peut également donner des résultats intéressants pour créer des effets (par exemple une lampe torche qui se déplace en utilisant un disque avec un dégradé radial au bord que l'on déplace sur une image : sur la partie peinte à 100% du disque, on voit l'image, sur la partie

peinte à 0%; on ne voit rien, et entre les deux une zone en clair obscur).

Vous pourrez par exemple utiliser [vignettes libres kenney](#), ou [celles-ci](#), ou rechercher par vous-même une planche sur votre moteur de recherche préféré "[free tileset dungeon](#)" (se limiter aux planches libres de droit !!).

```

void play_with_texture_4(SDL_Texture* my_texture,
                        SDL_Window* window,
                        SDL_Renderer* renderer) {
    SDL_Rect
        source = {0},                      // Rectangle définissant la
        window_dimensions = {0},           // Rectangle définissant la
        destination = {0},                // Rectangle définissant la
        state = {0};                      // Rectangle de la vignette

    SDL_GetWindowSize(window,             // Récupération des dimensions
                    &window_dimensions.w,
                    &window_dimensions.h);
    SDL_QueryTexture(my_texture,         // Récupération des dimensions
                    NULL, NULL,
                    &source.w, &source.h);

    /* Mais pourquoi prendre la totalité de l'image, on peut n'en
       prendre qu'une partie */

    int nb_images = 8;                  // Il y a 8 vignettes dans l'image
    float zoom = 2;                   // zoom, car ces images sont plus petites
    int offset_x = source.w / nb_images, // La largeur d'une vignette
        offset_y = source.h / 4;        // La hauteur d'une vignette

    state.x = 0;                      // La première vignette est à l'origine
    state.y = 3 * offset_y;           // On s'intéresse à la troisième vignette
    state.w = offset_x;               // Largeur de la vignette
    state.h = offset_y;               // Hauteur de la vignette

    destination.w = offset_x * zoom;   // Largeur du sprite à afficher
    destination.h = offset_y * zoom;   // Hauteur du sprite à afficher

    destination.y =                  // La course se fait en diagonale
        (window_dimensions.h - destination.h) / 2;

    int speed = 9;
    for (int x = 0; x < window_dimensions.w - destination.w; x += speed) {
        destination.x = x;              // Position en x pour la vignette
        state.x += offset_x;            // On passe à la vignette suivante
        state.x %= source.w;           // La vignette qui suit celle de début de ligne
        state.y -= speed;              // La course se fait en diagonale

        SDL_RenderClear(renderer);      // Effacer l'image précédente
        SDL_RenderCopy(renderer, my_texture, // Préparation de l'affichage
                      &state,
                      &destination);
        SDL_RenderPresent(renderer);    // Affichage
        SDL_Delay(80);                 // Pause en ms
    }
    SDL_RenderClear(renderer);          // Effacer la fenêtre
}

```

## 2.9.6 Et le fond ?

Fréquemment, on désire incruster un *sprite* sur un décor, et non sur un fond uni. Pour afficher, cela ne pose pas de problème particulier : à la façon d'un peintre, on positionne successivement les couches, et ce qui est peint après cache ce qui a été peint précédemment (superposition des couches).

Pour ce qui est de l'effacement, on peut effacer toute l'image, puis recommencer la totalité de la peinture (peut-être après avoir sauvegardé le fond, en particulier s'il est procédural). Cette solution est utilisée en particulier sur les jeux où il y a un scrolling de l'écran, et dans le cas général lorsque le 'fond' varie sur une grande partie.

Lorsque entre chaque image peu de choses ont été modifiées, on peut préférer sauvegarder un rectangle de fond qui englobe l'endroit où sera positionné l'élément mobile, superposer au fond l'élément mobile, afficher, et avant de passer à l'image suivante recopier la sauvegarde du fond pour effacer l'élément mobile. L'avantage de cette méthode est (potentiellement) sa vitesse car on ne modifie qu'une partie de la texture à afficher et pas sa totalité. Elle est moins utilisée maintenant suite à l'évolution des performances des circuits graphiques, et on préfère souvent simplifier le code, cette optimisation n'étant plus toujours nécessaire...

```

void play_with_texture_5(SDL_Texture *bg_texture,
                         SDL_Texture *my_texture,
                         SDL_Window *window,
                         SDL_Renderer *renderer) {
    SDL_Rect
        source = {0},                                // Rectangle définissant la source
        window_dimensions = {0},                      // Rectangle définissant la destination
        destination = {0};                           // Rectangle définissant la destination

    SDL_GetWindowSize(window,                      // Récupération des dimensions de la fenêtre
                     &window_dimensions.w,
                     &window_dimensions.h);
    SDL_QueryTexture(my_texture, NULL, NULL,      // Récupération des dimensions de la texture
                    &source.w, &source.h);

    int nb_images = 40;                            // Il y a 8 vignettes
    int nb_images_animation = 1 * nb_images;       //
    float zoom = 2;                               // zoom, car ces images sont plus petites
    int offset_x = source.w / 4,                  // La largeur d'une vignette
        offset_y = source.h / 5;                  // La hauteur d'une vignette
    SDL_Rect state[18];                           // Tableau qui stocke les rectangles

    /* construction des différents rectangles autour de chacune des vignettes */
    int i = 0;
    for (int y = 0; y < source.h ; y += offset_y) {
        for (int x = 0; x < source.w; x += offset_x) {
            state[i].x = x;
            state[i].y = y;
            state[i].w = offset_x;
            state[i].h = offset_y;
            ++i;
        }
    }
    state[14] = state[13];                        // state[14 à 16] la même chose que [13 à 15]
    state[15] = state[16];
    for(; i< nb_images ; ++i){                  // reprise du début de la boucle
        state[i] = state[39-i];
    }

    destination.w = offset_x * zoom;             // Largeur du sprite à afficher
    destination.h = offset_y * zoom;             // Hauteur du sprite à afficher
    destination.x = window_dimensions.w * 0.75; // Position en x pour l'affichage
    destination.y = window_dimensions.h * 0.7;   // Position en y pour l'affichage

    i = 0;
    for (int cpt = 0; cpt < nb_images_animation ; ++cpt) {
        play_with_texture_1_1(bg_texture,           // identique à play_with_texture_5
                             my_texture, &state[i], &destination);
        i = (i + 1) % nb_images;                  // Passage à l'image suivante
    }
}

```

```

        SDL_RenderPresent(renderer);           // Affichage
        SDL_Delay(100);                      // Pause en ms
    }
    SDL_RenderClear(renderer);             // Effacer la fenêtre
}

```

## 2.9.7 Résultat

Si on remet les différents morceaux de code précédents bout-à-bout, et peut-être quelque modifications mineures, on obtient [[.all\_executables.tar.gz]] (télécharger, exécuter 'textures').

## 2.9.8 Travail à réaliser : une animation

Il est temps de mettre en pratique ce que l'on vient de voir sur les textures, en créant une petite animation simple. Il est inutile qu'elle soit paramétrable, il ne faut pas y passer trop longtemps, faites uniquement de petites variations par rapport à ce qui a été proposé ci-avant, gardez vos idées géniales pour la suite :-).

# 2.10 Les évènements

Jusqu'ici, nous n'avons pas appris à interagir avec le programme (même le fermer...). Cette partie va présenter la logique de la boucle d'évènement ainsi que la gestion de quelques évènements particuliers.

## 2.10.1 La boucle d'évènements

C'est une boucle infinie, à l'intérieur de laquelle on va à chaque tour de boucle :

- gérer les affichages,
- faire les calculs qui sont à faire,
- examiner certaines entrées (mais avec des commandes non bloquantes, contrairement à `scanf` ou assimilés), c'est le point principal de la gestion d'évènements,
- faire les pauses nécessaires.

La boucle des évènements suit le squelette suivant:

```

SDL_bool program_on = SDL_TRUE;           // Booléen pour dire q
while (program_on){                      // Voilà la boucle des
    SDL_Event event;                     // c'est le type IMPOR

    while(program_on && SDL_PollEvent(&event)){ // Tant que la file de
        // pas vide et qu'on n
        // Défiler l'élémen
        switch(event.type){              // En fonction de la v
            case SDL_QUIT :             // Un évènement simple
                program_on = SDL_FALSE; // Il est temps d'arrê
                break;

            default:                  // L'évènement défilé
                break;
        }
    }

}

```

La page [SDL\\_Event](#) liste les valeurs possibles que peut prendre dans le code précédent la variable `event.type`. `event` est une structure qui contient une union. Ainsi, on peut accéder au champ `type` de cette structure, et selon la valeur du type, on peut accéder à ce qui est pertinent pour ce type, qui se trouve dans l'union.

La boucle d'évènement peut être un peu étoffée :

```

SDL_bool
program_on = SDL_TRUE, // Booléen pour dire si le programme est en cours
paused = SDL_FALSE; // Booléen pour dire si le programme est en pause
while (program_on) { // La boucle des événements
    SDL_Event event; // Evènement à traiter

    while (program_on && SDL_PollEvent(&event)) { // Tant que la file d'événements n'est pas terminée
        switch (event.type) { // En fonction de l'événement
            case SDL_QUIT: // Un évènement si l'utilisateur clique sur la croix
                program_on = SDL_FALSE; // Il est temps d'arrêter le programme
                break;
            case SDL_KEYDOWN: // Le type de événement est une touche appuyée
                switch (event.key.keysym.sym) { // comme la valeur de la touche
                    case SDLK_p: // 'p'
                    case SDLK_SPACE: // 'SPC'
                        paused = !paused; // basculement pause
                        break;
                    case SDLK_ESCAPE: // 'ESCAPE'
                    case SDLK_q: // 'q'
                        program_on = 0; // 'escape' ou 'q'
                        break;
                    default: // Une touche appuyée
                        break;
                }
                break;
            case SDL_MOUSEBUTTONDOWN: // Click souris
                if (SDL_GetMouseState(NULL, NULL) & // Si c'est un clic
                    SDL_BUTTON(SDL_BUTTON_LEFT) ) { // Fonction à exécuter
                    change_state(state, 1, window); // Fonction à exécuter
                } else if (SDL_GetMouseState(NULL, NULL) & // Si c'est un clic
                    SDL_BUTTON(SDL_BUTTON_RIGHT) ) { // Fonction à exécuter
                    change_state(state, 2, window); // Fonction à exécuter
                }
                break;
            default: // Les événements non gérés
                break;
        }
    }
    draw(state, &color, renderer, window); // On redessine
    if (!paused) { // Si on n'est pas en pause
        next_state(state, survive, born); // la vie continue
    }
    SDL_Delay(50); // Petite pause
}

```

## 2.10.2 Travail à réaliser : interactivité dans le jeu de la vie

Ajouter à votre jeu de la vie (télécharger, exécuter 'vie2') [click gche/dte, quitter q, pause 'SPC'] de l'interactivité. En particulier, il faudra que vous gériez au moins un évènement de chacun des types suivants :

- fermeture de la fenêtre,
- appui d'une touche,
- click souris,
- déplacement souris.

## 2.11 Écrire à l'écran

La bibliothèque `SDL_ttf` propose de multiples outils pour écrire et manipuler du texte.

Encore une fois, il faut charger la bibliothèque :

```
#include <SDL2/SDL_ttf.h>
```

Et initialiser ses fonctionnalités :

```
if (TTF_Init() < 0) end_sdl(0, "Couldn't initialize SDL TTF", window
```

Une fois le travail terminé, il ne faudra pas oublier de refermer cette bibliothèque :

```
TTF_Quit();
```

Ainsi, on peut obtenir un nouveau 'Hello World!' (télécharger, exécuter 'hello'), en n'oubliant pas d'ajouter le drapeau `-lSDL2_ttf` lors de la compilation.

```

#include <SDL2/SDL_ttf.h>

...

if (TTF_Init() < 0) end_sdl(0, "Couldn't initialize SDL TTF", window);

TTF_Font* font = NULL;
font = TTF_OpenFont("./fonts/Pacifico.ttf", 65);
if (font == NULL) end_sdl(0, "Can't load font", window, renderer);

TTF_SetFontStyle(font, TTF_STYLE_ITALIC | TTF_STYLE_BOLD);

SDL_Color color = {20, 0, 40, 255};
SDL_Surface* text_surface = NULL;
text_surface = TTF_RenderText_Blended(font, "Hello World !", color);
if (text_surface == NULL) end_sdl(0, "Can't create text surface", window);

SDL_Texture* text_texture = NULL;
text_texture = SDL_CreateTextureFromSurface(renderer, text_surface);
if (text_texture == NULL) end_sdl(0, "Can't create texture from surface");
SDL_FreeSurface(text_surface);

SDL_Rect pos = {0, 0, 0, 0};
SDL_QueryTexture(text_texture, NULL, NULL, &pos.w, &pos.h);
SDL_RenderCopy(renderer, text_texture, NULL, &pos);
SDL_DestroyTexture(text_texture);

SDL_RenderPresent(renderer);

...

TTF_Qui();
...

```

La fonction `TTF_RenderText_Blended` peut être remplacée par deux autres fonctions selon la vitesse demandée pour afficher et la qualité de la fusion entre le texte et l'image attendue : [documentation](#)

Il existe des fonctions adaptées à l'encodage UTF8 des caractères (au lieu du Latin1) disponibles dans cette bibliothèque.

## 2.12 Travail à réaliser : un premier chef d'œuvre

Le but ici est de mettre en œuvre tout ce qui a été vu sur la SDL, en créant un mini jeu. Vous pouvez créer une variation autour de `space`

[number](#) (télécharger, exécuter 'spacenumber') ou vous pouvez proposer un jeu totalement différent (à faire valider par l'enseignant), en gardant en tête que ce n'est qu'un exercice, et qu'il ne faudrait pas lui accorder plus d'une journée.

## 3 Structure de tas binaire

### 3.1 Présentation

Un tas binaire est un arbre binaire dont tous les niveaux sont complets, à l'exclusion peut-être du dernier qui est complet à gauche. À chaque nœud de l'arbre est associé une valuation  $A$ , et la propriété caractéristique de cet arbre est  $\forall nœud \neq \text{racine}, A(\text{mère}(nœud)) \leq A(nœud)$  (les ancêtres d'un nœud ont tous une valuation inférieure à celle de ce nœud).

Cette structure de donnée est particulièrement agréable pour implémenter une file de priorité, dont on aura besoin par la suite.

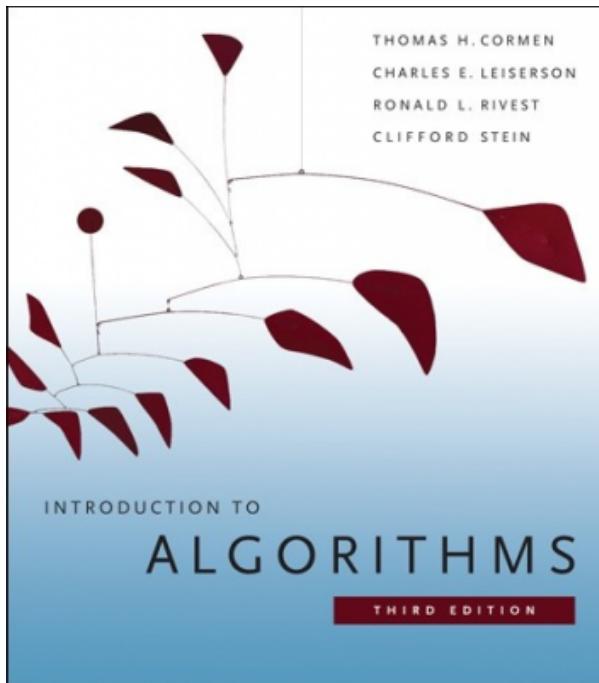


Figure 7 : Votre référence sur ce sujet

## 3.2 Travail à réaliser

1. implémenter un tas binaire dans le but de gérer une file de priorité où la racine porte la valeur minimale. Par la suite, cette structure sera utilisée afin de rechercher le prochain sommet à explorer dans un graphe dans des algorithmes de recherche d'un plus court chemin (algorithmes de Dijkstra et A\*),

2. mettre en œuvre un tri par tas, et faire une rapide comparaison de vitesse avec l'algorithme de tri proposé par défaut en c

```
void qsort(void *base, size_t nitems, size_t size, int (*compar)(const vc
```

Attention : vous devez être en mesure de répondre à des questions sur la complexité des opérations sur les tas.

## 4 Structure de partition

### 4.1 Type de données

### 4.1.1 Définition

L'idée derrière la notion de partition est la notion de classification : les éléments d'un ensemble sont regroupés par paquets.

Plus formellement, une partition d'un ensemble  $E$  est un ensemble  $F$  vérifiant :

- $\forall x \in F, x \subset E$  : les éléments de la partition sont des ensembles, qui contiennent des éléments de  $E$ . On appelle *classes* les éléments de  $F$ .
- $\forall (x, y) \in F^2 x \cap y \neq \emptyset \iff x = y$  : tout élément de  $E$  appartient à au plus une classe.
- $\bigcup_{x \in F} x = E$  : tout élément de  $E$  est dans une classe.

### 4.1.2 Primitives sur le type partition

Les primitives à implémenter sur ce type de donnée sont :

#### créer

crée à partir d'un ensemble  $E$  la partition où chaque élément est seul dans sa classe,

#### récupérer\_classe

prend en entrée un élément  $x \in E$  et renvoie un identifiant unique de la classe à laquelle appartient  $x$ ,

#### fusion

prend deux éléments  $x, y$  de  $E$  en entrée et fusionne les classes de  $x$  et de  $y$  dans la partition,

#### lister\_classe

prend en entrée une étiquette de classe, et renvoie les éléments de cette classe,

#### lister\_partition

renvoie la liste des classes.

### 4.1.3 Implémentation par marqueurs

On s'intéresse à réaliser des partitions de  $E = [0..N - 1]$  (ce cas peut en réalité être considéré comme générique). La représentation par marqueurs consiste à créer un vecteur d'entiers de  $N$  cases. Une partition  $P$  de  $E$  sera alors représentée par le vecteur  $C$  vérifiant :  $\forall (i, j) \in E^2, C[i] = C[j] \iff classe(i) = classe(j)$ .

## exemple

la partition  $\{ \{0, 1\}, \{2, 3, 10\}, \{4, 5, 6, 7, 8, 9\} \}$  pourrait être indifféremment représentée par les vecteurs  $C_1$  ou  $C_2$  (et bien d'autres encore).

Tableau 1 : Tableau  $C_1$

indice	0	1	2	3	4	5	6	7	8
valeur	42	42	156	156	389	389	389	389	389

Tableau 2 : Tableau C<sub>2</sub>

indice	0	1	2	3	4	5	6	7	8	9	10
valeur	0	0	3	3	5	5	5	5	5	5	3

A partir du tableau  $C_2$  en particulier, il est assez ais  de construire une impl mentation des primitives demand es. Pour le choix de l' tiquette de la classe, on choisit en g n ral   la cr ation la valeur de l'indice, et lors d'une fusion, l'une des deux  tiquettes d j  pr sent es. Ainsi le  $C_2$  peut, par exemple,  tre obtenu par la suite d'op rations suivantes :

- créer(11)

indice	0	1	2	3	4	5	6	7	8	9	10
valeur	0	1	2	3	4	5	6	7	8	9	10

- fusionner(0,1)

indice	0	1	2	3	4	5	6	7	8	9	10
--------	---	---	---	---	---	---	---	---	---	---	----

indice	0	1	2	3	4	5	6	7	8	9	10
valeur	0	0	2	3	4	5	6	7	8	9	10

- fusionner(3,2)

indice	0	1	2	3	4	5	6	7	8	9	10
valeur	0	0	3	3	4	5	6	7	8	9	10

- fusionner(5,6)

indice	0	1	2	3	4	5	6	7	8	9	10
valeur	0	0	3	3	4	5	5	7	8	9	10

- fusionner(6,7)

indice	0	1	2	3	4	5	6	7	8	9	10
valeur	0	0	3	3	4	5	5	5	8	9	10

- fusionner(6,8)

indice	0	1	2	3	4	5	6	7	8	9	10
valeur	0	0	3	3	4	5	5	5	5	9	10

- fusionner(3,10)

indice	0	1	2	3	4	5	6	7	8	9	10
valeur	0	0	3	3	4	5	5	5	5	9	3

- fusionner(5,4)

indice	0	1	2	3	4	5	6	7	8	9	10
valeur	0	0	3	3	5	5	5	5	5	9	3

- fusionner(7,9)

indice	0	1	2	3	4	5	6	7	8	9	10
valeur	0	0	3	3	5	5	5	5	5	5	3

On peut noter qu'avec cette implémentation, déterminer la classe d'un élément se fait en  $\Theta(1)$ , fusionner en  $\Theta(N)$  et lister une classe en  $\Theta(N)$ . Lister\_partition est en  $\Theta(N)$  (en pensant à utiliser un tableau de listes).

#### 4.1.4 Implémentation arborescente

L'idée est cette fois de représenter la partition par une forêt, dont chaque arbre (non nécessairement binaire) est une des classes de la partition. Les nœuds d'un arbre sont les éléments de la classe et la racine de l'arbre est choisie comme étiquette de la classe. On note alors que pour connaître l'étiquette de la classe d'un élément, il est nécessaire de remonter à la racine de l'arbre. Cela a deux conséquences :

- l'implémentation de l'arbre devra permettre de remonter d'un nœud vers la racine,
- la hauteur d'un arbre devra être minimale.

De plus, on désire améliorer le temps nécessaire pour réaliser une fusion par rapport à la version précédente. Il est donc important de ne pas réaliser d'opérations inutiles lors d'une fusion.

On choisit donc d'implémenter la forêt par un tableau de  $N$  cases. Le contenu de la case d'indice  $k$  du tableau est l'indice de la mère de  $k$ , en prenant la convention que la racine d'un arbre est sa propre mère.

Afin de limiter la hauteur de l'arbre on choisit de fusionner les classes des éléments  $x$  et  $y$  comme suit :

- On note  $T_x$  (respectivement  $T_y$ ) l'arbre qui représente la classe à laquelle appartient  $x$  (respectivement  $y$ ),
- Si la hauteur de l'arbre  $T_x$  est strictement supérieure à celle de  $T_y$ ,

alors  $T_y$  devient un nouveau fils de la racine de  $T_x$  (et inversement dans l'autre cas).

On note qu'avec cette méthode, si les deux arbres sont de hauteurs différentes, la hauteur de l'arbre résultant de la fusion est celle du plus grand des deux avant la fusion. La hauteur de l'arbre résultant augmente uniquement lorsque les arbres  $T_x$  et  $T_y$  sont de même hauteur (et la hauteur de l'arbre obtenu est la hauteur de ces arbres augmentée de 1). La partition  $\{ \{0, 1\}, \{2, 3, 10\}, \{4, 5, 6, 7, 8, 9\} \}$  peut être obtenue comme suit (il y a d'autres possibilités):

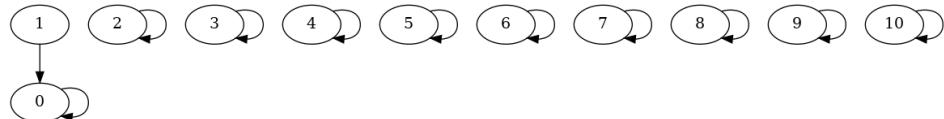
- créer(11) [il n'y a que des racines]

indice	0	1	2	3	4	5	6	7	8	9	10
valeur	0	1	2	3	4	5	6	7	8	9	10



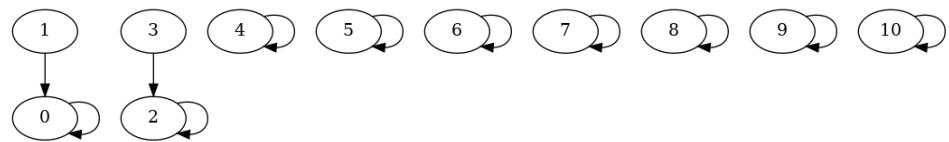
- fusionner(0,1)

indice	0	1	2	3	4	5	6	7	8	9	10
valeur	0	0	2	3	4	5	6	7	8	9	10



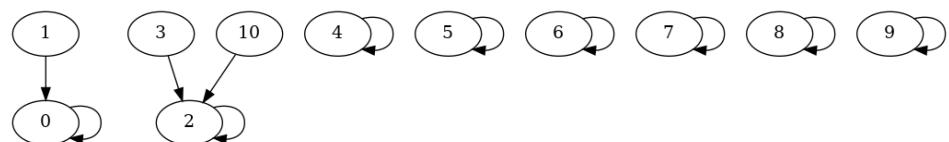
- fusionner(2,3)

indice	0	1	2	3	4	5	6	7	8	9	10
valeur	0	0	2	2	4	5	6	7	8	9	10



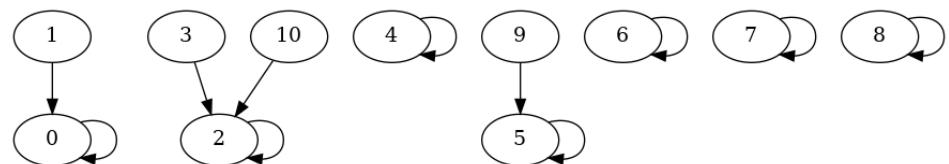
- fusionner(10,3)

indice	0	1	2	3	4	5	6	7	8	9	10
valeur	0	0	2	2	4	5	6	7	8	9	2



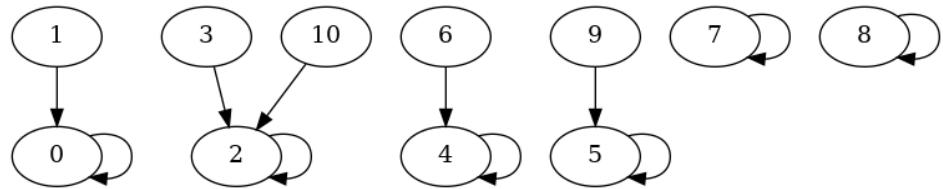
- fusionner(5,9)

indice	0	1	2	3	4	5	6	7	8	9	10
valeur	0	0	2	2	4	5	6	7	8	5	2



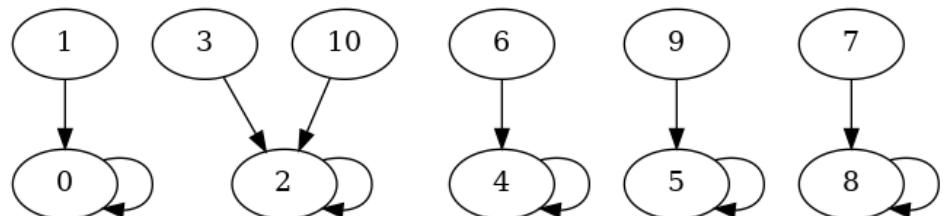
- fusionner(4,6)

indice	0	1	2	3	4	5	6	7	8	9	10
valeur	0	0	2	2	4	5	4	7	8	5	2



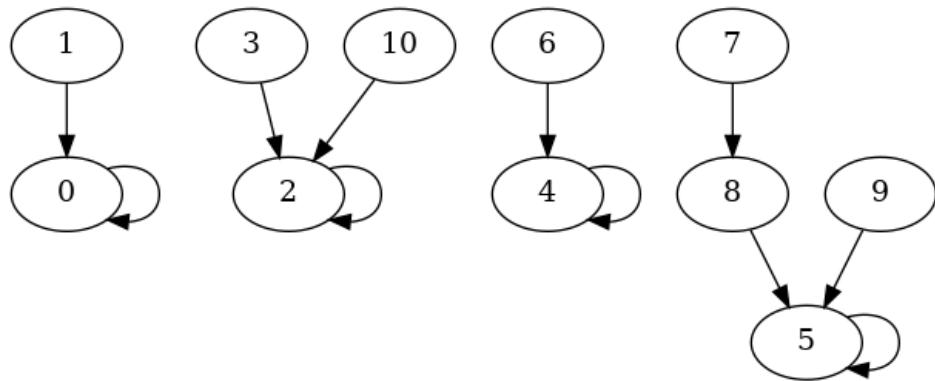
- fusionner(8,7)

indice	0	1	2	3	4	5	6	7	8	9	10
TO valeur	0	0	2	2	4	5	4	8	8	5	2



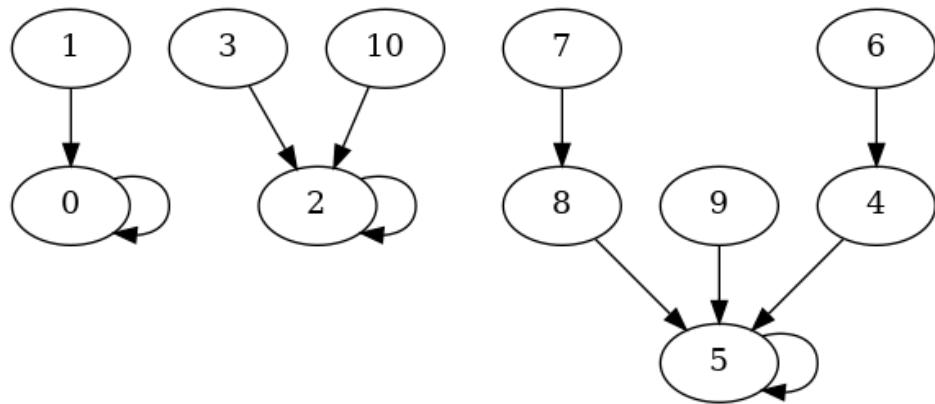
- fusionner(7,9)

indice	0	1	2	3	4	5	6	7	8	9	10
valeur	0	0	2	2	4	5	4	8	5	5	2



- fusionner(6,8)

indice	0	1	2	3	4	5	6	7	8	9	10
valeur	0	0	2	2	5	5	4	8	5	5	2



### Remarques

- Il est intéressant de maintenir un tableau supplémentaire d'entiers de taille  $N$  contenant à l'indice  $k$  la hauteur de l'arbre si  $k$  est une racine (et rien d'intéressant si  $k$  n'est pas une racine)
- Il est assez facile de montrer qu'avec le choix opéré pour la fusion, la hauteur des arbres est logarithmique par rapport au nombre de nœuds qu'il contient (et donc en  $\Theta(\log N)$  dans le pire des cas),

- La recherche de l'étiquette de la classe d'un élément est également logarithmique par rapport au nombre de nœuds de la classe, donc dans le pire des cas en  $\Theta(\log N)$ ,
- Lister les éléments d'une classe est en  $\Theta(N \times \log N)$  par un algorithme naïf (mais peut être assez facilement amélioré en  $\Theta(N)$ ).
- Lister la totalité des classes est en  $\Theta(N \times \log N)$  en utilisant un tableau de listes, et en pensant bien à insérer en tête ou en utilisant la même méthode que dans le point précédent ramené à  $\Theta(N)$ . On peut noter que pour obtenir la complexité linéaire pour la primitive précédente, il peut être profitable de réaliser la liste de toutes les classes préalablement.

#### 4.1.5 Travail à réaliser

Vous devez implémenter le type partition présenté dans sa version arborescente. Il faudra réaliser une représentation des partitions à l'aide de [Graphviz](#).

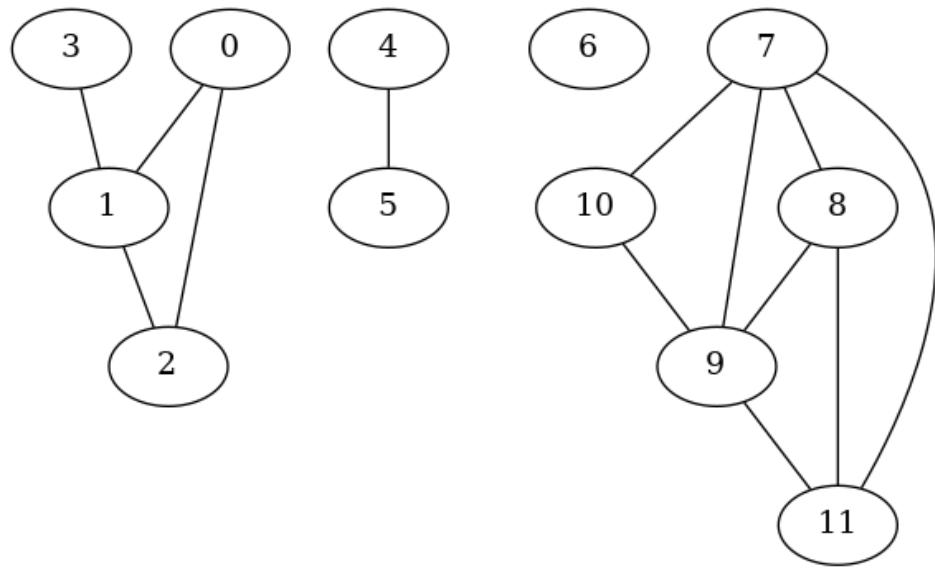
### 4.2 Composantes connexes

#### 4.2.1 Définitions

Un graphe non orienté est connexe s'il existe toujours un chemin reliant deux nœuds quelconques du graphe.

Les composantes connexes d'un graphe est l'ensemble contenant les sous-graphes connexes maximaux au sens de l'inclusion.

#### Exemple



Ce graphe admet 4 composantes connexes :

---

nœuds 0, 1, 2, 3	nœuds 4, 5	nœud 6	nœuds 7, 8, 9, 10, 11

---

#### 4.2.2 Algorithme

Le type de données partition permet de mettre en œuvre l'algorithme suivant afin de déterminer les composantes connexes d'un graphe :

- Créer une partition dont les éléments sont les nœuds,
- Pour chaque arête du graphe, fusionner dans la partition des classes contenant les extrémités de l'arête (ne rien faire s'ils sont déjà dans

la même classe),

- Si on désire récupérer les sous graphes (et pas uniquement les nœuds des sous-graphes) : pour chaque classe de la partition, prendre la restriction du graphe aux nœuds concernés.

### 4.2.3 Travail à réaliser : calcul des composantes connexes

Au préalable, lire [5.1](#) qui précise les deux représentations de graphe choisies dans ce paragraphe.

#### 1. Graphe représenté par une matrice d'adjacence

- Écrire une fonction qui génère la matrice d'adjacence d'un graphe non orienté aléatoire de  $N$  nœuds,
- Le représenter avec graphviz,
- Calculer les nœuds de chacune des composantes connexes de ce graphe,
- Représenter avec graphviz les sous-graphes correspondant à ses différentes composantes connexes.

#### 2. Graphe représenté par la liste de ses arêtes

- Reprendre le travail précédent, mais en implémentant cette fois le graphe par un couple (nombre de nœuds, liste des arêtes).

## 4.3 Arbre couvrant de poids minimal

### 4.3.1 Présentation, algorithme de Kruskal

Un arbre couvrant d'un graphe non orienté est un **arbre** (graphe acyclique connexe) qui vérifie que :

- ses nœuds sont les mêmes que ceux du graphe,
- la liste de ses arêtes est extraite de la liste des arêtes du graphe.

Sur un graphe non orienté valué, un arbre couvrant est de poids minimal si la somme des valuations de ses arêtes est minimale.

Le type partition permet de fabriquer aisément un arbre couvrant ou un arbre couvrant de poids minimal d'un graphe : algorithme de Kruskal.

1. Ordonner par ordre croissant de valuation les arêtes (cette étape peut être omise si on ne cherche pas un arbre couvrant de poids minimal),
2. Créer une partition dont les éléments sont les nœuds du graphe,
3. Créer une liste d'arêtes  $A$ , vide,
4. Pour chaque arête (ordonnée Cf. point 1), si les deux extrémités de l'arête ne sont pas dans la même composante connexe, alors, fusionner les classes auxquelles elles appartiennent et ajouter cette arête à  $A$ .

Après ces opérations,  $A$  contient les arêtes d'un arbre couvrant du graphe, et si le point 1 a été réalisé, celui-ci est de poids minimal.

#### **Remarque**

dans le cas où le graphe de départ n'est pas connexe, l'algorithme fonctionne et renvoie alors une forêt couvrante de poids minimal.

#### **4.3.2 Travail à réaliser : Kruskal**

Sur un graphe représenté par la liste de ses arêtes valuées :

- générer un graphe aléatoire,
- le représenter avec graphviz,
- en calculer une forêt couvrante de poids minimal,
- représenter avec graphviz cette forêt.

### **4.4 Crédation d'un labyrinthe arborescent**

Cette partie va relier vos travaux sur les composantes connexes, et en particulier l'algorithme de Kruskal, avec le thème du projet : les labyrinthes.

#### **4.4.1 Travail à réaliser : un labyrinthe arborescent**

On se place dans la situation d'un labyrinthe posé sur une grille de  $n$  lignes et  $p$  colonnes.

1. On considère le graphe où les nœuds sont les cases du labyrinthe, et

- où  $(i, j)$  est une arête si leurs cases correspondantes sont voisines (il n'est pas nécessaire de le construire effectivement),
2. Appliquer l'algorithme de Kruskal en modifiant l'étape 1 : les arêtes sont mélangées selon un ordre aléatoire par l'algorithme de [Fisher-Yate](#),
  3. Représenter graphiquement le labyrinthe, par des tracés de droites avec la SDL,
  4. Représenter graphiquement le labyrinthe, cette fois-ci en fabriquant une image à partir des vignettes (*tiles*) d'une planche (*tileset*).

## 4.5 Extension à un labyrinthe plus quelconque

Cette variation de l'algorithme précédent prend en argument un réel  $p \in [0; 1]$  supplémentaire. Dans l'algorithme de Kruskal, lorsque les deux extrémités d'une arête se trouvent déjà dans la même composante connexe, on n'utilise pas l'arête dans l'arbre, et donc on ne crée pas de passage entre les cases correspondant à ces sommets. Ici, l'algorithme choisira aléatoirement dans  $\alpha \in [0; 1]$  et si  $\alpha < p$ , alors un passage sera tout de même créé entre les cases correspondant aux sommets considérés. Le résultat ne sera donc plus un arbre, et la densité de murs dans le labyrinthe sera réglée par la valeur de  $p$  :  $p = 0$ , on retrouve un arbre,  $p = 1$ , il n'y a aucun mur.

En ce basant sur des variations comme celle-ci, on peut modifier la forme du labyrinthe, par exemple en élaguant les feuilles de l'arbre (alors toute les cases ne sont plus accessibles), en choisissant un ordre particulier au lieu d'un mélange complet, ...

### 4.5.1 Travail à réaliser : labyrinthe non arborescent

Modifier le code du labyrinthe arborescent pour introduire  $p$ .

## 5 Parcours du graphe

### 5.1 Représentation du graphe correspondant au labyrinthe

Le graphe qui correspond au labyrinthe possède pour nœuds les cases

du labyrinthe, et deux nœuds sont reliés dans le graphe si on peut transiter d'une case à l'autre sans passer par une case intermédiaire. **On considèrera que seuls les déplacements Nord, Sud, Est, Ouest sont possibles.**

On note que si on représente ce graphe par une matrice d'adjacence (chaque ligne de la matrice correspond à un nœud, chaque colonne de la matrice correspond à un nœud, et en case  $(i, j)$ ,  $i > j$  il y a un 1 s'il existe une arête du nœud  $i$  au nœud  $j$ , et un 0 sinon), alors la matrice possède  $(nb\_lignes\_labyrinthe \times nb\_colonnes\_labyrinthe)^2$  cases (ce qui est déjà beaucoup, même si on travaille avec un petit labyrinthe de 1000 par 1000). Sur cette matrice, si on considère labyrinthe classique (pas de téléporteurs ou ponts,...), le nombre d'arêtes ne dépasse pas le double du nombre de cases du labyrinthe, ce qui donne un ratio maximal du nombre de cases à 1 de  $\frac{nb\_lignes\_labyrinthe \times nb\_colonnes\_labyrinthe \times 2}{(nb\_lignes\_labyrinthe \times nb\_colonnes\_labyrinthe)^2} = \frac{2}{nb\_lignes\_labyrinthe \times nb\_colonnes\_labyrinthe}$  qui tend très vite vers 0. Cette représentation du graphe n'est donc pas envisageable dans cette situation.

On choisit donc de représenter le graphe par un couple (nombre de nœuds, liste des arêtes). Les nœuds seront représentés par des entiers de  $[0..N - 1]$ . Les arêtes sont des couples d'entiers, avec la première composante de ce couple inférieure à la seconde (cette contrainte sert à imposer la non orientation du graphe). Il faudra disposer de fonctions permettant de passer d'un nœud à 'son' entier, comme de l'entier à 'son' nœud.

Il sera possible dans la partie libre de réaliser des variations en choisissant des valuations différentes sur les arcs : cette valuation pourra représenter par exemple :

- le temps mis pour aller d'une case à une autre (modélisation de la difficulté à parcourir le terrain),
- le danger à passer par une case (modélisation des risques), ...

## 5.2 Algorithme de Dijkstra

Implémenter la recherche du plus court chemin sur un graphe à

valuations positives par l'algorithme de Dijkstra. Vous ferez attention à bien utiliser une file de priorité pour stocker les nœuds restant à explorer.

### 5.2.1 Travail à réaliser : promenade dans le labyrinthe

Le labyrinthe est connu, en particulier la position de départ et la position d'arrivée, vous devez réaliser une animation d'un personnage sur le labyrinthe créé précédemment partant d'une origine vers une destination. Ces deux points seront clairement marqués sur la carte du labyrinthe. Une fois la destination atteinte, celle-ci devient la nouvelle origine, et une nouvelle destination est choisie aléatoirement... et ainsi de suite.

## 5.3 Algorithme A\*

### 5.3.1 Présentation

L'algorithme A\* est une variation de l'algorithme de Dijkstra qui va ordonner un peu différemment les nœuds dans la file de priorité des nœuds à explorer. Au lieu que la priorité d'un nœud soit "la distance du plus court chemin connu à l'origine", on va utiliser "la distance **estimée** du plus court chemin entre l'origine et la destination". Cette distance estimée se décompose en "distance du plus court chemin connu à l'origine" + "estimation **minimisante** de la distance restant à parcourir" (le fait que cette deuxième estimation soit minimisante est ce qui permet de garantir que lorsque le nœud exploré est la destination, on **garantit** que le plus court chemin a été trouvé).

Dans le cas que nous traitons, il est facile d'obtenir une estimation minimisante de la distance restant à parcourir : c'est la distance à parcourir dans le cas où il n'y a pas de mur !

On peut voir A\* comme un algorithme de Dijkstra où l'exploration au lieu de se dérouler en cercles concentriques centrés sur l'origine, se déroule selon des ovales qui seraient ces cercles déformés et attirés par la destination. La petite longueur de ces ovales sera particulièrement petite si l'estimation est proche de la distance restant réellement à

parcourir, et dans le cas où cette estimation est nulle (la pire des estimations correctes), alors on retrouve l'algorithme de Dijkstra.

### 5.3.2 Travail à réaliser : A\*

- Reprendre le programme précédent en remplaçant l'algorithme de Dijkstra par l'algorithme A\*,
- Comparer la vitesse d'exécution sur un labyrinthe assez grand, en choisissant successivement pour l'estimation de la distance restant à parcourir :
  - la distance Euclidienne,
  - la [distance de Tchebychev](#),
  - la [distance de Manhattan](#).

## 5.4 Exploration d'un graphe en profondeur d'abord

### 5.4.1 Travail à réaliser : Exploration du labyrinthe

- Écrire un algorithme qui réalise une exploration en profondeur d'un graphe,
- Mettre en œuvre cet algorithme en affichant en simultané : le personnage qui se déplace dans le labyrinthe et sur une deuxième image, sa connaissance actuelle du labyrinthe, le programme se termine lorsque le labyrinthe est parfaitement connu.

## 6 Extensions proposées

Les lignes qui suivent vous proposent des pistes d'extension. Vous n'êtes pas tenu de vous limiter à ces propositions, et leur description est suffisamment succincte pour laisser libre court à votre créativité.

### 6.0.1 Extension : vie artificielle

L'idée est ici créer un environnement de vie artificielle. Des proies et des prédateurs sont posés dans le labyrinthe. Les proies ont besoin de ressources qu'elles doivent rejoindre, ce qui permettra leur reproduction (la proie trouve un diamant, et hop, maintenant il y a deux proies !), les prédateurs ont eux besoin des proies pour se reproduire

(elles mangent une proie, et hop, maintenant il y a deux prédateurs !).

Votre travail consiste à donner une « intelligence » à chacune de ces deux espèces. Dans cette extension, il sera intéressant de représenter les courbes de population et de tester différents paramètres afin d'en explorer l'influence. Dans cette analyse des résultats, il sera important d'avoir une démarche méthodique (même si tous les tests ne sont pas possibles à cause du temps nécessaire pour leur réalisation).

### 6.0.2 Extension : jeu vidéo

Le but est de définir un jeu vidéo d'arcade prenant place dans le labyrinthe. On pourra par exemple donner un comportement à des monstres, et créer un jeu qui pourra aussi bien être dans la lignée des 'pac-man' que des 'bomber-man', ou ...

Une autre version possible est un jeu où l'utilisateur est au centre de l'écran, il ne voit que les cases situées jusqu'à une certaine distance, et lorsque le joueur se déplace, c'est le labyrinthe qui se déplace autour du joueur, le joueur restant toujours au centre de la fenêtre, tourné vers le nord (au lieu d'avoir une vue extérieure, on est en vue personnage).

### 6.0.3 Extension : Thésée

Le labyrinthe est au départ avec peu de murs, il y a deux adversaires : le Minotaure et Thésée (tous les deux gérés par vos programmes, mais un humain peut les remplacer). À chaque tour de jeu, Thésée peut se déplacer de  $p$  cases à la recherche de la sortie et le Minotaure peut ajouter un mur (sachant qu'il ne peut pas en ajouter trop près de la destination). Le joueur gagne s'il atteint sa destination, le minotaure s'il n'y a plus de moyen de l'atteindre.

### 6.0.4 Extension : le Wi-Fi dans le labyrinthe

Le but est ici d'installer le Wi-Fi pour tous les habitants du labyrinthe. Les ondes Wi-Fi ont une portée de  $p$  cases (qui suivent le tracé du labyrinthe, les ondes ne traversent pas les murs). En fonction de  $p$  et du labyrinthe, où et combien faut-il placer de bornes afin qu'aucune case ne

soit en zone blanche.

### 6.0.5 Extension : exploration avant colonisation

Une destination est fixée dans le labyrinthe, sa position est inconnue d'un personnage également placé dans le labyrinthe.

- Le personnage doit explorer le labyrinthe en réalisant un minimum de déplacements,
- La phase d'exploration s'arrête dès que le chemin le plus court est déterminé.

## 7 Valgrind

[Valgrind](#) (avec son outil par défaut memcheck) analyse l'exécution d'un programme pour, entre autres, rapporter :

- la quantité de mémoire allouée dynamiquement,
- les fuites mémoires (avec numéro de ligne de l'allocation liée si compilé avec ` -g`),
- les erreurs d'accès en lecture ou écriture à la mémoire,
- les erreurs d'utilisation en lecture d'une variable non initialisée.
- Valgrind ne remplace pas un débogueur, en particulier dans une situation d'échec d'exécution, comme lors d'une erreur de segmentation (segfault).

Pour des explications plus précises et complètes, voir [le manuel de Valgrind](#).

### 7.1 Rapport sur l'utilisation mémoire

Une sortie possible de Valgrind :

```
==3719130== Memcheck, a memory error detector
==3719130== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Se
==3719130== Using Valgrind-3.16.1 and LibVEX; rerun with -h for
==3719130== Command: ./a.out
==3719130==
==3719130==
```

```
==3719130== HEAP SUMMARY:
==3719130==     in use at exit: 0 bytes in 0 blocks
==3719130==   total heap usage: 3 allocs, 3 frees, 70 bytes allo
==3719130==
==3719130== All heap blocks were freed -- no leaks are possible
==3719130==
==3719130== For lists of detected and suppressed errors, rerun w
==3719130== ERROR SUMMARY: 0 errors from 0 contexts (suppressed:
```

La section `HEAD SUMMARY` indique le nombre d'allocations, de libérations ainsi que le total d'octets alloués.

Valgrind indique ensuite que `All heap blocks were freed -- no leaks are possible`, il n'y a pas eu de fuite mémoire lors de cette exécution du programme.

Lorsqu'un programme ne libère pas toute sa mémoire, le rapport peut ressembler à ceci :

```
==3720916== HEAP SUMMARY:
==3720916==     in use at exit: 10 bytes in 1 blocks
==3720916==   total heap usage: 3 allocs, 2 frees, 70 bytes allo
==3720916==
==3720916== LEAK SUMMARY:
==3720916==   definitely lost: 10 bytes in 1 blocks
==3720916==   indirectly lost: 0 bytes in 0 blocks
==3720916==   possibly lost: 0 bytes in 0 blocks
==3720916==   still reachable: 0 bytes in 0 blocks
==3720916==   suppressed: 0 bytes in 0 blocks
==3720916== Rerun with --leak-check=full to see details of leak
```

Valgrind indique ici qu'il n'y a eu que deux libérations, alors que 3 allocations avaient été réalisées.

La section suivante du rapport Valgring, `LEAK SUMMARY`, précise comment l'accès à la mémoire a été perdu.

## 7.2 Rapport sur les variables non initialisées

Pour le code source suivant :

```

1: #include <stdio.h>
2:
3: void foo(int condition, int value) {
4:     if(condition) printf("value: %d\n", value);
5:     else           puts("no display");
6: }
7:
8: int main() {
9:     int i, j = 0;
10:
11:     foo(i, j);
12: }
```

Sans l'option `-g` à la compilation, le résultat obtenu est alors :

```

==3729129== Conditional jump or move depends on uninitialised va
==3729129==   at 0x109144: foo (in /tmp/tmp.1622971724.300q/a.o)
==3729129==   by 0x109174: main (in /tmp/tmp.1622971724.300q/a.
```

Avec l'option `-g` :

```

==3729359== Conditional jump or move depends on uninitialised va
==3729359==   at 0x109144: foo (main.c:4)
==3729359==   by 0x109174: main (main.c:13)
```

Dans la fonction `foo`, à la ligne 4, on utilise la variable `condition` en lecture sans que celle-ci ait été préalablement affectée. Cette variable est un paramètre de la fonction, il faut donc aller au lieu de l'appel. Valgrind nous indique que la ligne 13 de la fonction `main` appelle `foo` avec les arguments `i` et `j`. La variable `i` correspond à la `condition` dans la fonction `foo` : il s'agit de la variable non initialisée (ce que l'on vérifie facilement à la ligne 11).

## 7.3 Lecture/écriture invalide

Considérons le code source suivant :

```

1: #include <stdlib.h>
2:
3: int main() {
4:     int n = 5;
5:     int *tab = malloc(n*sizeof *tab);
6:     for(int i = 0; i <= n; ++i)
7:         tab[i] = i;
8:
9:     free(tab);
10: }

```

Une analyse par valgrind, sans l'option `-g` à la compilation, fournit le rapport :

```

==3845342== Invalid write of size 4
==3845342==   at 0x109189: main (in /tmp/tmp.1622971724.300q/a.
==3845342==   Address 0x4a3a054 is 0 bytes after a block of size
==3845342==   at 0x483877F: malloc (vg_replace_malloc.c:307)
==3845342==   by 0x109164: main (in /tmp/tmp.1622971724.300q/a.

```

Avec l'option `-g` à la compilation on obtient :

```

==3846517== Invalid write of size 4
==3846517==   at 0x109189: main (main.c:7)
==3846517==   Address 0x4a3a054 is 0 bytes after a block of size
==3846517==   at 0x483877F: malloc (vg_replace_malloc.c:307)
==3846517==   by 0x109164: main (main.c:5)

```

Cela permet de détecter une erreur dite « [off-by-one](#) ». En effet, il y a une écriture invalide de taille 4 (`Invalid write of size 4`) à la *ligne 7* du programme, en accédant à un espace mémoire alloué à la *ligne 5* du programme. Lorsque l'on connaît la machine sur laquelle on exécute le programme, on peut savoir qu'un `int` y possède une taille de 4 octets ([le standard oblige un minimum de 2 octets](#)). Il s'agit donc d'une affectation d'un entier hors des limites allouées pour le tableau. On corrige la boucle qui va jusqu'à `n` inclus pour s'arrêter avant : le test devient `i < n`.

## 7.4 Valgrind et la SDL

Lorsque le programme analysé par Valgrind utilise des bibliothèques, il peut arriver que le rapport comporte des éléments indésirables. Ce sont des erreurs dont l'origine n'est pas notre programme, mais qui proviennent de la bibliothèque. Par exemple, utiliser la SDL, selon sa version, et les différents éléments du système peuvent ainsi faire apparaître des « fuites mémoire » qui ne nous intéressent pas (la SDL n'étant pas même nécessairement à l'origine de ces fuites, leur origine est peut-être une bibliothèque utilisée par la SDL, X11??).

Pour remédier à ce problème, Valgrind permet de supprimer certains résultats dans ses rapports. Un ensemble de règles de suppression peut être écrit dans un fichier que l'on nomme fichier de suppressions.

Ce fichier de suppressions peut être généré à partir d'un log d'une session d'analyse avec l'option `--gen-suppressions=all`. Le principe est simple : on détecte des erreurs sur un programme qui n'a pas d'erreur, et on va indiquer à Valgrind que par la suite, ces erreurs ne doivent plus être notifiées dans les rapports.

### Attention

tout problème détecté durant cette session sera ensuite ignoré, il faut donc s'assurer de le faire avec un programme dont on est certain de la validité !

Exécution de Valgrind avec génération des informations de suppression :

```
valgrind --leak-check=full --show-reachable=yes --error-limit=no --g
```

Où :

- `<log>` doit être remplacé par un chemin vers un fichier à écrire,
- `<executable>` doit être remplacé par le chemin vers l'exécutable à exécuter.

Un script accessible [ici](#) (qu'il ne pas oublier de rendre exécutable après l'avoir téléchargé), permet de traduire ce log en un fichier de suppressions :

```
cat <log>| ./gen_valgrind_suppressions > sdl.sup
```

Ensuite, à chaque appel de Valgrind, on ajoute (en plus des options souhaitées) `--suppressions=.`/`sdl.sup` :

```
valgrind --suppressions=.
```

Ces explications sont extraites d'un [wiki](#).

## 7.5 Bonnes pratiques

Pour conclure sur l'utilisation basique de Valgrind (ou plus précisément, a propos de l'outil par défaut de Valgrind : Memcheck), quelques bonnes pratiques.

### 7.5.1 Options de compilation

Pendant la phase de développement, l'option `-g` (pour GCC et Clang au moins) permet d'intégrer dans l'exécutable de nombreuses informations utiles aux outils comme `gdb` et Valgrind.

### 7.5.2 Allocations

Un seul appel à `malloc` (ou apparentés) par ligne permet d'identifier facilement l'allocation qui pose problème lorsqu'une fuite de mémoire ou une erreur de lecture/écriture est détectée.

### 7.5.3 Définitions

Une seule définition de variable par ligne permet d'identifier facilement quelle variable pose problème lorsqu'une variable non initialisée est utilisée en lecture.

## 8 Astuces

### 8.1 Comparer les temps d'exécution

On dispose de plusieurs façons pour mesurer les temps d'exécution d'un programme.

### 8.1.1 Méthodes rudimentaires

- La commande Linux `time` permet de mesurer le temps d'exécution d'une commande en ligne de commande.

```
time ./a.out
time ls -al
```

- à l'intérieur du code C, insérer des affichages du temps passé. On peut s'inspirer du code suivant (src : [koor.fr](#))

```
#include <stdio.h>
#include <time.h>

int main( int argc, char * argv[] ) {

    clock_t begin = clock();

    // Do something
    // sleep( 2 );      // Wait 2 seconds, but no ticks are consumed
    int i;
    for( i=0; i<10000000000; i++ ) {

    }

    clock_t end = clock();
    unsigned long millis = (end - begin) * 1000 / CLOCKS_PER_SEC;
    printf( "Finished in %ld ms\n", millis );

    return 0;
}
```

### 8.1.2 Méthodes évoluées : les profileurs

Les profileurs permettent de mesurer le temps passé dans chaque fonction du programme. Ils ont en général pour vocation de permettre de détecter, sans faire d'étude théorique, les portions du programme qui nécessiteraient des optimisations (mémoire ou temps). Ainsi, une fonction déjà optimisée peut mériter des optimisations plus avancées,

voire une remise en cause de son principe, parce qu'elle est fréquemment appelée, alors qu'une autre fonction, peu appelée pourra être laissée un peu « bâclée ».

Il est à noter que la qualité des informations apportées par les profileurs est fortement liée à la qualité et au « réalisme » des instances sur lesquelles le programme est exécuté. Le but d'une étude de profilage est souvent de se placer soit dans le pire des cas (de façon à se rapprocher expérimentalement d'un temps garanti, très important en particulier dans le 'temps réel'), soit dans les cas qui seront les plus fréquents en production (ou un subtil mélange de ces deux situations).

Il existe deux grandes sortes de profileurs :

- ceux qui mesurent le temps mis en secondes sur une certaine machine, avec toutes ses caractéristiques particulières (son processeur, sa mémoire, sa carte graphique, ..., les tâches qui sont lancées en même temps, etc),
- ceux qui s'exécutent sur une machine virtuelle, et où les unités de mesure sont plutôt, le nombre d'opérations effectuées, chaque opération possédant un coût. Le défaut majeur de ce type de profileur est la lenteur (de assez lent à ... très très lent), son avantage est, par construction, sa quasi indépendance au matériel (quasi, car si les coûts réels des opérations sont relatifs au matériel).

Sous Linux, la première famille est représentée par `gprof` et la seconde par `callgrind` qui est un module de `valgrind` (dont vous n'avez pour l'instant utilisé que le module par défaut `memcheck`), que l'on utilise conjointement avec le visualiseur `kcacheGrind`.

`gprof` ([tutoriel gprof](#)) est d'accès beaucoup plus aisé que `kcacheGrind` ([documentation cachegrind](#) et [documentation kcachegrind](#), je ne connais pas de tutoriel particulièrement pertinent, vous pouvez en choisir un 'au hasard', il vous permettra à tout le moins de commencer à travailler si c'est l'outil que vous choisissez).

## 8.2 clang-format

`clang-format` est un formateur automatique de code, il va présenter automatiquement le code, en choisissant là où il y a des espaces, combien, la position des accolades... Cela va permettre d'avoir une unité dans la façon de présenter le code, et donc aider à sa lecture et sa maintenance. Avant son utilisation, il est nécessaire de créer un fichier de configuration qui va définir le *style* de la présentation.

```
clang-format -style=LLVM -dump-config > ~/.clang-format
```

Le style par défaut est `LLVM`, mais il existe d'autres styles directement accessibles :

- Google
- Chromium
- Mozilla
- WebKit
- Microsoft
- GNU

En plus de ces styles, il est possible de créer son propre style ex nihilo, ou en modifiant un style déjà existant.