

Projet apprentissage de règles

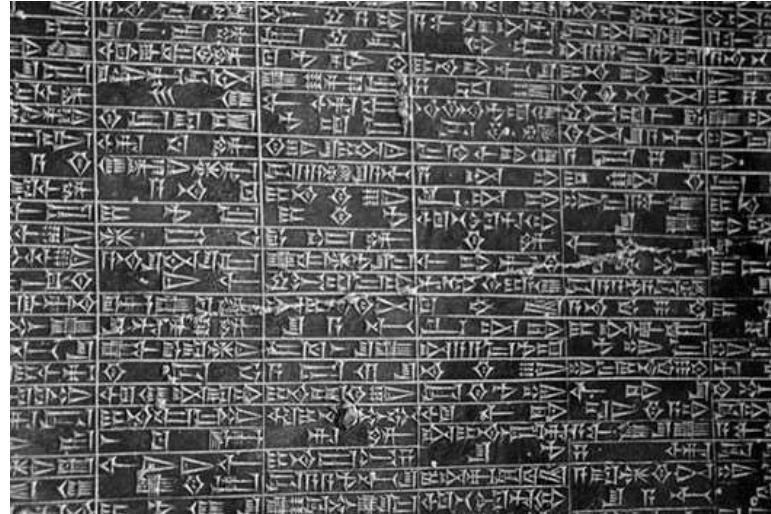


Figure 1 : Code d'Hammurabi 1750 Av JC, Musée du Louvre

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 en individuel et se termine par la création d'un petit jeu.

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 » respectant la thématique du projet.

1.1. Synopsis pédagogiques

1.1.1. Gestion de projet

Votre groupe doit mener le travail à bout, vous devez donc créer une synergie de groupe qui vous permettra d'obtenir des livrables de qualité maximale, dans une ambiance de travail agréable et productive. Le projet se déroule sur un temps plus long que le temps 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 2 : 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.

Le terme 'liste' est souvent utilisé dans ce document pour simplement signifier 'énumération' et n'implique pas de façon de l'implémenter.

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



Figure 3 : Evaluation

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),

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 : n'utilisez que des images, sons, planches de sprites (assets), etc... libres de droits.

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 des soutenances est d'environ 10 minutes (6 minutes de présentation, 4 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 avant la date du début du projet, les groupes seront alors imposés par l'équipe enseignante (en général cela ne présage pas d'un excellent projet lorsque vous arrivez le lundi matin sans avoir trouvé vos coéquipiers...).

1.5. Présence obligatoire

La présence est obligatoire sur les créneaux indiqués dans l'emploi du temps, et les retards ne sont pas considérés comme très 'professionnels' et seront sanctionnés.

Les deux vendredis, il est possible que votre groupe ait une soutenance en dehors des plages horaires indiquées ci-dessus ou que des retards imposent un dépassement des horaires prévus.

Il est recommandé d'avancer le projet en dehors des créneaux indiqués ;).

1.6. Phases du projet

1.6.1. Deux phases

Durant la première semaine du projet, les apprentissages sont guidés. Ils ont pour vocation à vous familiariser avec la création de graphiques, d'animations en 2D. Les premiers exercices sont individuels, afin que chacun dispose d'un socle commun, puis dans la deuxième moitié de la semaine, un premier travail de groupe est réalisé, un petit jeu vous permettant de mettre en œuvre vos acquis.

La seconde phase permettra à votre groupe de créer une application plus conséquente de votre choix, respectant les contraintes ainsi que la thématique imposées.

1.6.2. Personnalisation des exercices

Les exercices demandés acceptent des variations, acceptables tant qu'elles vous permettent de démontrer votre savoir-faire sur le point étudié. Le but n'est pas de plagier les exemples proposés, mais de démontrer votre savoir-faire ! 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. Récapitulatif des travaux à réaliser

La liste des programmes à réaliser, dont un contenu plus précis sera détaillé dans la suite du document est :

X fenêtré : travail individuel

Savoir ouvrir, fermer, déplacer, redimensionner des fenêtres graphiques.

Pavé de serpents : travail individuel

Savoir dessiner des formes géométriques, les animer.

Animer des sprites : individuel

Créer une animation possédant un fond (si possible en mouvement), et un sprite qui se déplace sur ce fond.

Voyageur de commerce : travail de groupe

Créer un jeu basé sur le voyageur de commerce ou équivalent, mettant en oeuvre la partie graphique (formes géométriques, interarctions à la souris et au clavier), une petite partie graphe, et des techniques d'optimisation.

Chef d'œuvre : travail de groupe

Créer un jeu d'arcade simple démontrant votre maîtrise de toute la partie graphique étudiée précédemment. Ce travail termine la première semaine.

Jeu avec apprentissage de règles : travail de groupe

Créer un jeu graphique, avec une partie du jeu où des règles sont apprises automatiquement.

1.8. Machines

1.8.1. Machines virtuelles

Afin de pouvoir travailler de la façon la plus agréable possible, il est possible d'utiliser une machine virtuelle personnelle fournie par l'ISIMA sur laquelle vous êtes administrateurs. Être administrateur vous permettra d'installer les logiciels de votre choix, et de configurer à votre guise le système d'exploitation.

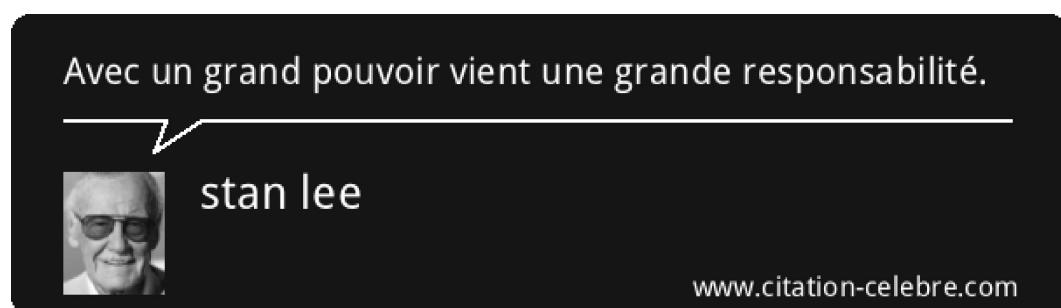


Figure 4 : Spiderman ou sudo ?

Pour rappel, il est nécessaire de démarrer la machine virtuelle avant de pouvoir l'utiliser sur guacamole, ou en s'y connectant via ssh `ssh -Y login@nom_machine`, où à l'aide de `remmina`. Il peut être nécessaire de reconfigurer le password (vous pouvez mettre celui de l'ENT ici).

1. Pour remmina

- choisir rdp (*remote desktop protocol*)
- nouveau profil de connexion
 - Name : votre choix
 - Group : isima
 - Protocol : doit déjà être rempli avec RDP
 - Server : vm-etu-login où *login* est à remplacer par votre login
 - Password : vous le mettez ou non...
 - Domain : local.isima.fr
 - ... si besoin, remplissez les option de votre choix ...
 - Resolution : use client resolution
 - Save !!!
- sur l'écran principal, dans les préférences (`CTRL p`), dans l'onglet RDP (tout en bas), cocher 'use client keyboard mapping' pour conserver votre configuration du clavier. Parfois, cela ne suffit pas (???), il faut alors rechercher dans la liste qui précède cette case à cocher un clavier approprié.

2. Précautions

Cette machine virtuelle n'est pas 'garantie', elle peut être effacée, remplacée, modifiée à tout moment. Il est par conséquent important que vos documents soient présents dans un endroit fiable, par exemple le répertoire 'shared'.

Une astuce pour disposer aisément de vos fichiers de config si vous utilisez plusieurs machines, est de les placer dans le répertoire 'shared', et de placer un lien symbolique vers ces fichiers. Par exemple, pour récupérer une configuration *emacs* placée dans 'shared' :

```
ln -s "~/shared/.emacs.d" "~/.emacs.d"
```

Évidemment, les risques de perdre beaucoup de votre travail sont quasi nuls, même si votre ordinateur personnel est capturé par des aliens, car il vous est obligatoire de faire des commits et des pushes très régulièrement.

Dans l'éventualité où le répertoire 'shared' ne serait pas monté, la commande `kinit` permet parfois de résoudre le problème (essayer au moins trois fois).

1.8.2. Machines personnelles

Vous pouvez utiliser votre ordinateur personnel, que ce soit avec une distribution Linux ou WSL pour Windows... mais dans tous les cas il vous est demandé de faire de la programmation sous Linux, et non sous Windows.

2. SDL2

2.1. Préliminaire

Vous avez déjà eu un aperçu de la SDL2 pendant les cours de C, ne vous privez pas d'y retourner : flood-it.

Le perfectionnement sur cette bibliothèque se fera en réalisant de petits exercices qui mettront en œuvre des outils que vous développerez. Il est intéressant de construire un code propre qui sera le plus facile possible à maintenir et réutiliser. Des mini applications vous serviront de tests fonctionnels, afin de vérifier l'utilisabilité pratique de vos codes.

Les explications qui suivent ne se substituent absolument pas à la consultation de la documentation du wiki, mais pourront vous aider à mettre le pied à l'étrier.



Figure 5 : 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, gérer la souris,
- 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, comme la gestion su son, des déformations d'image, réseau, etc... (installer tout `libSDL2-*` est, je pense, la bonne solution).
- Afin de vérifier la bonne installation, recopier le code suivant qui charge la bibliothèque et affiche le numéro de version

```
#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) $(SDL2-config --cflags --libs)` (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, ...) via `SDL_Init` [il sera possible d'activer par la suite d'autres modules si nécessaire par d'autres fonctions d'initialisation plus spécifiques],

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
- ...
 - 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 gauche
        *window_2 = NULL;                                // Future fenêtre de droite

    /* 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());                            // l'initialisation de la SDL a échoué
        exit(EXIT_FAILURE);
    }

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

    if (window_1 == NULL) {
        SDL_Log("Error : SDL window 1 creation - %s\n",
            SDL_GetError());                          // échec de la création de la fenêtre
        SDL_Quit();                                 // On referme la SDL
        exit(EXIT_FAILURE);
    }

    /* Création de la fenêtre de droite */
    window_2 = SDL_CreateWindow(
        "Fenêtre à droite",                           // codage en utf8, donc accents possibles
        400, 0,                                       // à droite de la fenêtre de gauche
        500, 300,                                     // largeur = 500, hauteur = 300
        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());                          // échec de la création de la deuxième fenêtre
        SDL_DestroyWindow(window_1);                // la première fenêtre (qui elle a été créée) doit être détruite
        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 création */
    SDL_DestroyWindow(window_2);                  // la fenêtre 2
    SDL_DestroyWindow(window_1);                  // la fenêtre 1

    SDL_Quit();                                   // la SDL

    return 0;
}

```

2.4. Les évènements

La logique de la gestion des évènements avec la SDL est celle de la boucle évènementielle. C'est une boucle 'infinie', où à chaque tour de boucle, on examine le contenu du buffer clavier, l'état de la souris, on fait évoluer les objets graphiques de quelques pixels, on produit ou on arrête un son... Chaque tour de boucle doit être assez court afin de donner l'illusion que les réactions du programme sont 'instantanées' et que tout se passe en même temps. Pendant une itération il est en général opportun de :

- gérer les affichages,
- examiner certaines entrées (mais avec des commandes non bloquantes, contrairement à `scanf` ou assimilés), c'est le point principal de la gestion d'évènements,
- calculer les nouveaux états,
- faire les pauses nécessaires afin d'obtenir une bonne fluidité (il vaut mieux être globalement un peu lent plutôt que saccadé).

La boucle des évènements suit le squelette suivant:

```
SDL_bool program_on = SDL_TRUE;           // Booléen pour dire que le programme doit continuer
SDL_Event event;                         // c'est le type IMPORTANT !!

while (program_on){                      // Voilà la boucle des évènements

    if (SDL_PollEvent(&event)){          // si la file d'évènements n'est pas vide : défilez
        // de file dans 'event'
        switch(event.type){            // En fonction de la valeur du type de cet évènement
            case SDL_QUIT :           // Un évènement simple, on a cliqué sur la x de la fenêtre
                program_on = SDL_FALSE; // Il est temps d'arrêter le programme
                break;
            default:                  // L'évènement défilé ne nous intéresse pas
                break;
        }
        // Affichages et calculs souvent ici
    }
}
```

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. Cette façon de ranger peut vous inspirer lorsque vous avez à conserver des éléments qui ont des rôles similaires mais qui peuvent prendre des formes différentes.

La boucle d'évènement peut être un peu étoffée :

```

SDL_bool
program_on = SDL_TRUE,                                // Booléen pour dire que le programme doit continuer
paused = SDL_FALSE,                                 // Booléen pour dire que le programme est en pause
event_utile = SDL_FALSE;                            // Booléen pour savoir si on a trouvé un event utile
SDL_Event event;                                    // Evènement à traiter

while (program_on) {                                // La boucle des évènements
    event_utile = SDL_FALSE;
    while(!event_utile && SDL_PollEvent(&event)) { // Tant que on n'a pas trouvé d'évènement utile
        // et la file des évènements stockés n'est pas vide
        // terminé le programme Défiler l'élément en tête de file dans 'event'
        switch (event.type) {                         // En fonction de la valeur du type de cet évènement
            case SDL_QUIT:                           // Un évènement simple, on a cliqué sur la x de la fenêtre
                program_on = SDL_FALSE;               // Il est temps d'arrêter le programme
                event_utile = SDL_TRUE;
                break;
            case SDL_KEYDOWN:                        // Le type de event est : une touche appuyée
                // comme la valeur du type est SDL_KeyDown, dans la partie 'union' de
                // l'event, plusieurs champs deviennent pertinents
                switch (event.key.keysym.sym) {       // la touche appuyée est ...
                    case SDLK_p:                      // 'p'
                    case SDLK_SPACE:                 // ou 'SPC'
                        paused = !paused;
                        event_utile = SDL_TRUE;
                        break;
                    case SDLK_ESCAPE:                 // 'ESCAPE'
                    case SDLK_q:                   // ou 'q'
                        program_on = 0;             // 'escape' ou 'q', d'autres façons de quitter le programme
                        event_utile = SDL_TRUE;
                        break;
                    default:                       // Une touche appuyée qu'on ne traite pas
                        break;
                }
                break;
            case SDL_MOUSEBUTTONDOWN:                // Click souris
                if (SDL_GetMouseState(NULL, NULL) &
                    SDL_BUTTON(SDL_BUTTON_LEFT) ) {   // Si c'est un click gauche
                    change_state(state, 1, window); // Fonction à exécuter lors d'un click gauche
                } else if (SDL_GetMouseState(NULL, NULL) &
                    SDL_BUTTON(SDL_BUTTON_RIGHT) ) { // Si c'est un click droit
                    change_state(state, 2, window); // Fonction à exécuter lors d'un click droit
                }
                event_utile = SDL_TRUE;
                break;
            default:                           // Les évènements qu'on n'a pas envisagé
                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
}

```

Parfois, la réactivité du clavier avec cette méthode n'est pas satisfaisante, le code suivant peut alors contourner ce problème afin d'obtenir des saisies claviers réactives :

```
const Uint8* keystates = SDL_GetKeyboardState(NULL); // Récupère l'état des touches du clavier
if (keystates[SDL_SCANCODE_UP]) { // Lorsque la touche flèche haut est enfoncée..
    // ... action à effectuer.
}
```

2.5. Travail à réaliser : un X fenêtré



Figure 6 : X-Men (affiche film dark phoenix)

Vous devez mettre en évidence par un code votre aptitude à ouvrir et fermer des fenêtres. Un résultat possible de votre travail pourrait être : ./all_executables.tar.gz (télécharger, exécuter 'X_fenetre'). N'hésitez pas à demander à un enseignant si ce que vous proposez de réaliser correspond à la difficulté recherchée.

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.

2.6. Mettre un 'rendu' dans une fenêtre

Un rendu/rendering 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 un 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).

2.7. 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 rendu + dessin
*/
//*****************************************************************************

void end sdl(char ok,                                     // fin normale : ok = 0 ; anomalie : ok > 0
            char const* msg,                                // message à afficher
            SDL_Window* window,                            // fenêtre à fermer
            SDL_Renderer* renderer) {                      // renderér à fermer

    char msg_formated[255];
    int l;

    if (!ok) {                                         // Affichage de ce qui ne va pas
        strncpy(msg_formated, msg, 250);
        l = strlen(msg_formated);
        strcpy(msg_formated + l, " : %s\n");

        SDL_Log(msg_formated, SDL_GetError());
    }

    if (renderer != NULL) {                           // Destruction si nécessaire
        SDL_DestroyRenderer(renderer);                // Attention : on suppose que les
        renderer = NULL;                             // Attention : on suppose que les
    }
    if (window != NULL) {                            // Destruction si nécessaire
        SDL_DestroyWindow(window);                  // Attention : on suppose que les
        window= NULL;
    }

    SDL_Quit();

    if (!ok) {                                       // On quitte si cela ne va pas
        exit(EXIT_FAILURE);
    }
}

void draw(SDL_Renderer* renderer) {                     // Je pense que vous allez faire mieux
    SDL_Rect rectangle;

    SDL_SetRenderDrawColor(renderer,
                          50, 0, 0, 255);                    // mode Red, Green, Blue (tous dans 0..255)
    // 0 = transparent ; 255 = opaque
    rectangle.x = 0;                                // x haut gauche du rectangle
    rectangle.y = 0;                                // y haut gauche du rectangle
    rectangle.w = 400;                              // sa largeur (w = width)
    rectangle.h = 400;                              // sa hauteur (h = height)

    SDL_RenderFillRect(renderer, &rectangle);

    SDL_SetRenderDrawColor(renderer, 0, 0, 0, 255);
    SDL_RenderDrawLine(renderer,
                       0, 0, 400, 400);                   // x,y du point de la première extrémité
    // x,y seconde extrémité

    /* tracer un cercle n'est en fait pas trivial, voilà le résultat sans algo intelligent ... */
    for (float angle = 0; angle < 2 * M_PI; angle += M_PI / 4000) {
        SDL_SetRenderDrawColor(renderer,
                               (cos(angle * 2) + 1) * 255 / 2,           // quantité de Rouge
                               (cos(angle * 5) + 1) * 255 / 2,           // de vert
                               (cos(angle) + 1) * 255 / 2,              // de bleu
                               255);                                 // opacité = opaque

        SDL_RenderDrawPoint(renderer,
                            200 + 100 * cos(angle),               // coordonnée en x

```

```

        200 + 150 * sin(angle));           // en y
    }

}

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'échec possible
    if (SDL_Init(SDL_INIT_VIDEO) != 0) end_sdl(0, "ERROR SDL INIT", window, renderer);

    SDL_GetCurrentDisplayMode(0, &screen);
    printf("Résolution écran\n\tw : %d\n\t h : %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, renderer);

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

    /*************************************************************************/
    /* On dessine dans le renderer
    /*************************************************************************/
    /* Cette partie pourrait avantageusement être remplacée par la boucle évènementielle
    draw(renderer);                                // appel de la fonction qui crée l'image
    SDL_RenderPresent(renderer);                   // affichage
    SDL_Delay(1000);                             // Pause exprimée en ms

    /* 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 programming handbook dont je vous recommande la lecture.

2.8. 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, Uint8 b, Uint8 a)` la couleur dans laquelle le fond devra être peint.

2.9. Manipuler des textures

Une *texture* est une structure qui peut contenir une image, ainsi que les informations *largeur* et *hauteur* de 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 fonction IMG_Load
                                         // Penser au flag -lSDL2_image à la compilation
//...

SDL_Texture* load_texture_from_image(char * file_image_name, SDL_Window *window, SDL_Renderer *renderer)
{
    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 dans la surface
                                         // image=SDL_LoadBMP(file_image_name); fonction standard de
                                         // uniquement possible si l'image est au format bmp */
    if (my_image == NULL) end_sdl(0, "Chargement de l'image impossible", window, renderer);

    my_texture = SDL_CreateTextureFromSurface(renderer, my_image); // Chargement de l'image de la surface
    SDL_FreeSurface(my_image);                                     // la SDL_Surface ne sert que comme
                                                               // modèle pour la texture
    if (my_texture == NULL) end_sdl(0, "Echec de la transformation de la surface en texture", window, renderer);

    return my_texture;
}

//...
IMG_Quit()                                // Si on charge une librairie SDL, il faut penser à la décharger
```

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 dans la texture", window, renderer);
//...
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 définissant la zone de la texture à récupérer
        window_dimensions = {0},                      // Rectangle définissant la fenêtre, on n'utilisera que :
        destination = {0};                           // Rectangle définissant où la zone_source doit être déposée

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

    destination = window_dimensions;               // On fixe les dimensions de l'affichage à celles de la fenêtre

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

    SDL_RenderCopy(renderer, my_texture,
                  &source,                                // Création de l'élément à afficher
                  &destination);                          // Affichage
    SDL_RenderPresent(renderer);                  // Pause en ms
    SDL_Delay(2000);                            // Effacer la fenêtre
}
```

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 zone de la texture à récupérer
    window_dimensions = {0},            // Rectangle définissant la fenêtre, on n'utilisera que la
    destination = {0};                 // Rectangle définissant où la zone_source doit être déposée

    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 l'image

    float zoom = 1.5;                  // Facteur de zoom à appliquer
    destination.w = source.w * zoom;   // La destination est un zoom de la source
    destination.h = source.h * zoom;   // La destination est un zoom de la source
    destination.x =
        (window_dimensions.w - destination.w) / 2; // La destination est au milieu de la largeur de la
    destination.y =
        (window_dimensions.h - destination.h) / 2; // La destination est au milieu de la hauteur de la

    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 définissant la zone de la texture à récupérer
    window_dimensions = {0},                      // Rectangle définissant la fenêtre, on n'utilisera que son taille
    destination = {0};                           // Rectangle définissant où la zone_source doit être déplacée

    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 l'image

    /* On décide de déplacer dans la fenêtre cette image */
    float zoom = 0.25;                            // Facteur de zoom entre l'image source et l'image

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

    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 du numéro d'image

        SDL_RenderClear(renderer);                // Effacer l'image précédente

        SDL_SetTextureAlphaMod(my_texture, (1.0 - 1.0 * i / nb_it) * 255); // L'opacité va passer de 255 à 0
        SDL_RenderCopy(renderer, my_texture, &source, &destination); // Préparation de l'affichage
        SDL_RenderPresent(renderer);           // Affichage de la nouvelle image
        SDL_Delay(30);                        // Pause en ms
    }
    SDL_RenderClear(renderer);                // Effacer la fenêtre une fois le travail terminé
}

```

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^{ème} 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 zone totale de la planche
        window_dimensions = {0},                      // Rectangle définissant la fenêtre, on n'utilisera que la
        destination = {0},                            // Rectangle définissant où la zone_source doit être déposée
        state = {0};                                 // Rectangle de la vignette en cours dans la planche

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

    /* Mais pourquoi prendre la totalité de l'image, on peut n'en afficher qu'un morceau, et changer de ligne */

    int nb_images = 8;                           // Il y a 8 vignette dans la ligne de l'image qui nous intéresse
    float zoom = 2;                             // zoom, car ces images sont un peu petites
    int offset_x = source.w / nb_images,         // La largeur d'une vignette de l'image, marche car la planche
    offset_y = source.h / 4;                      // La hauteur d'une vignette de l'image, marche car la planche

    state.x = 0;                                // La première vignette est en début de ligne
    state.y = 3 * offset_y;                      // On s'intéresse à la 4ème ligne, le bonhomme qui court
    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 l'écran
    destination.h = offset_y * zoom;             // Hauteur du sprite à l'écran

    destination.y =                               // La course se fait en milieu d'écran (en vertical)
    (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 l'affichage du sprite
        state.x += offset_x;                   // On passe à la vignette suivante dans l'image
        state.x %= source.w;                  // La vignette qui suit celle de fin de ligne est
                                                // celle de début de ligne

        SDL_RenderClear(renderer);           // Effacer l'image précédente avant de dessiner la nouvelle
        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 avant de rendre la main
}

```

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 réellement 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 zone de la texture à récupérer
    window_dimensions = {0},                      // Rectangle définissant la fenêtre, on n'utilisera que les dimensions
    destination = {0};                           // Rectangle définissant où la zone_source doit être dessinée

    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 l'image
                     &source.w, &source.h);

    int nb_images = 40;                          // Il y a 8 vignette dans la ligne qui nous intéresse
    int nb_images_animation = 1 * nb_images;       // zoom, car ces images sont un peu petites
    float zoom = 2;                            // La largeur d'une vignette de l'image
    int offset_x = source.w / 4;                // La hauteur d'une vignette de l'image
    offset_y = source.h / 5;                    // Tableau qui stocke les vignettes dans le bon ordre
    SDL_Rect state[40];                         // Tableau qui stocke les vignettes dans le bon ordre

    /* construction des différents rectangles autour de chacune des vignettes de la planche */
    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[15] = state[14];                      // ivaut 20 en sortie de boucle
    state[14] = state[13];                      // on fabrique des images 14 et 15 en reprenant la 13
                                                // donc state[13 à 15] ont la même image, le monstre n'a pas de bras
    for(; i< nb_images ; ++i){                  // reprise du début de l'animation en sens inverse
        state[i] = state[39-i];
        // 20 == 19 ; 21 == 18 ; ... 39 == 0
    }

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

    i = 0;
    for (int cpt = 0; cpt < nb_images_animation ; ++cpt) {
        play_with_texture_1_1(bg_texture,          // identique à play_with_texture_1, où on a enlevé l'affichage
                             window, renderer);
        SDL_RenderCopy(renderer,                   // Préparation de l'affichage
                      my_texture, &state[i], &destination);
        i = (i + 1) % nb_images;                 // Passage à l'image suivante, le modulo car l'animation est de 15 images
        SDL_RenderPresent(renderer);             // Affichage
        SDL_Delay(100);                        // Pause en ms
    }
    SDL_RenderClear(renderer);                  // Effacer la fenêtre avant de rendre la main
}

```

2.9.7. Gestion de la parallaxe

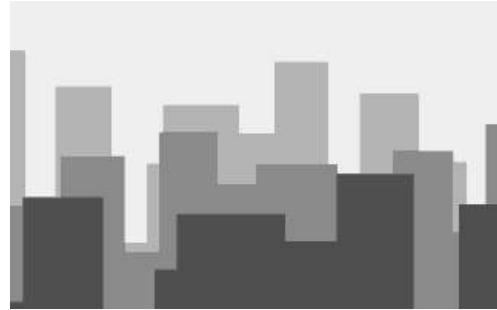


Figure 7 : parallaxe : Wikipedia

Le défilement parallaxe consiste à donner une impression de profondeur en utilisant plusieurs fonds superposés les uns sur les autres (algorithme du peintre : la nouvelle couche de peinture en recouvrant l'ancienne cache ce qui était dessiné) que l'on anime à des vitesses différentes (ce qui est plus loin bouge plus lentement).

Si on imagine que l'on regarde le paysage depuis une voiture :

- le soleil et la lune, à l'infini, ne bougent pas, on les dessine en premier
- les montagnes à l'horizon ne bougent quasiment pas : on va les dessiner ensuite,
- les collines un peu plus proches se déplacent à une vitesse moyenne,
- les arbres au bord de la route se déplacent à la vitesse de la voiture, comme ils sont les plus proches on les dessine en dernier.

Pour que tout se passe bien, il ne faut pas oublier de gérer la transparence : les arbres ne doivent se dessiner que là où il y a de la matière. Dans l'image de l'arbre, ce qui n'est pas arbre doit être de couleur totalement transparente (rappel : les images au format `png` possèdent un canal `a` pour gérer la transparence).

On peut encore améliorer l'effet de profondeur en jouant sur les couleurs et la netteté :

- plus un décor est lointain, plus on en diminue la netteté (le flou créé par les l'air)
- plus un décor est lointain, plus on en diminue le contraste.

Afin de réaliser ces effets, on peut utiliser le logiciel `gimp`, ou tout autre logiciel de traitement d'images prenant en charge ces éléments (quasiment tous).

2.9.8. 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').

2.9.9. 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-dessus, gardez vos idées géniales pour la suite :-).

2.10. 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, renderer);
```

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> // Charger la bibliothèque

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

TTF_Font* font = NULL; // la variable 'police de caractères'
font = TTF_OpenFont("./fonts/Pacifico.ttf", 65); // La police à charger, la taille
if (font == NULL) end_sdl(0, "Can't load font", window, renderer);

TTF_SetFontStyle(font, TTF_STYLE_ITALIC | TTF_STYLE_BOLD); // en italique, gras

SDL_Color color = {20, 0, 40, 255}; // la couleur du texte
SDL_Surface* text_surface = NULL; // la surface (uniquement transparente)
text_surface = TTF_RenderText_Blended(font, "Hello World !", color); // création du texte dans la surface
if (text_surface == NULL) end_sdl(0, "Can't create text surface", window, renderer);

SDL_Texture* text_texture = NULL; // la texture qui contient le texte
text_texture = SDL_CreateTextureFromSurface(renderer, text_surface); // transfert de la surface à la texture
if (text_texture == NULL) end_sdl(0, "Can't create texture from surface", window, renderer);
SDL_FreeSurface(text_surface); // la texture ne sert plus à rien

SDL_Rect pos = {0, 0, 0, 0}; // rectangle où le texte va être affiché
SDL_QueryTexture(text_texture, NULL, NULL, &pos.w, &pos.h); // récupération de la taille (width et height)
SDL_RenderCopy(renderer, text_texture, NULL, &pos); // Ecriture du texte dans le rendu
SDL_DestroyTexture(text_texture); // On n'a plus besoin de la texture

SDL_RenderPresent(renderer); // Affichage

...
TTF_Quit(); // Quitter la bibliothèque
```

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.

2.11. 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 'spacenumbers') 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. Optimisation discrète

3.1. Généralités



Figure 8 : "I'm late, I'm late, for a very important date!" Alice in wonderland (A. Caroll)

Cette partie a pour objectif de vous faire découvrir certaines techniques d'optimisation monocritère. On dispose d'une fonction ψ définie sur un espace E , à valeurs dans \mathbb{R}^+ . Le but est de déterminer un élément de E où la fonction ψ est maximale (ou minimale). Le nom de cette fonction varie en fonction du contexte et du domaine, on retrouve fréquemment pour la désigner les termes *fonction d'évaluation* (très général), *fonction objectif*, *fonction d'erreur*, *énergie*, *fitness*... Afin de bien comprendre en quoi ce problème est général voici quelques exemples:

- Une entreprise achète des bananes, des fraises et des livres, la fonction ψ associe à un nombre de bananes, fraises et livres achetés un lundi le bénéfice moyen engendré par leur vente au cours de la semaine (ici $E = \mathbb{N}^3$). Maximiser ψ permet de trouver les quantités à commander afin de maximiser les bénéfices.
- Un voyageur de commerce doit parcourir un ensemble de villes et revenir à son point de départ. la fonction ψ associe à une permutation des villes la longueur du trajet correspondant au parcours des villes dans l'ordre indiqué.

- Des formes doivent être découpées dans du tissu afin de fabriquer des vêtements. La fonction ψ associe à un placement des formes sur le tissu le rapport $\frac{\text{surface de tissu utile}}{\text{surface de tissu utilisée}}$.
- Un graphe étant donné, la fonction ψ associe à un chemin dans le graphe partant d'un certain nœud A à un certain nœud B la longueur de ce chemin, on peut alors chercher là où ψ est minimale et donc trouver un plus court chemin.
- Un graphe étant donné, les valuations des arcs représentent la probabilité de survie lors du passage d'un nœud à un autre. ψ associe à un chemin le produit des valuations des arrêtes qui le constituent. Trouver un chemin de A à B où ψ est maximal revient à rechercher un chemin de A à B de risque minimal.
- Une entreprise recherche à maximiser ses bénéfices, sa production, minimiser ses coûts, ...
- Un animal souhaite maximiser la quantité de nourriture qu'il mange, la taille de son territoire, la taille de sa descendance...
- Un pâtissier cherche à concevoir un nouveau gâteau, quelle doit en être la recette afin de maximiser le plaisir qu'il procurera ?
- Recherche des paramètres d'un réseau de neurones...

La suite de cette présentation se limitera au cas où E est un espace discret, ce qui implique en particulier l'absence de présentation de méthodes type gradient ou essaim particulière.

3.1.1. Créer un cache pour limiter les évaluations

La fonction d'évaluation coûte en général cher en temps de calcul (et pour la seconde partie du projet : très cher), il est donc intéressant de ne pas l'appeler plus que nécessaire. Pour cela, on peut utiliser un système de cache : on encapsule la situation dans une structure contenant deux champs supplémentaires :

- la dernière valeur connue de l'évaluation,
- un booléen indiquant si la valeur précédente est valide.

```
typedef struct{
    bool_t updated;
    configuration_t configuration;
    float evaluation;
} configurationEvaluated_t;
```

- Lorsque cette structure est initialisée :
 - le booléen est initialisé à faux pour signifier que la valeur du champ `evaluation` n'est pas fiable.
- Lorsque la fonction d'évaluation est appelée sur une variable de ce type :
 - si le booléen est à faux :
 - calculer effectivement la valeur de l'évaluation,
 - l'affecter au champ `evaluation`,
 - passer le booléen à vrai,
 - puis dans tous les cas renvoyer la valeur du champ `evaluation`.
- Lorsque la configuration est modifiée :
 - passer la valeur du booléen à faux pour signifier que la valeur du champ `evaluation` n'est plus valable.

3.2. Problème du voyageur de commerce



Figure 9 : « Je ne suis pas perdu, j'explore. » (Anonyme)

3.2.1. Problème

Un voyageur de commerce désire se rendre chez un ensemble de clients et voudrait connaître le chemin le plus court lui permettant de tous les visiter et revenir à son point de départ.

3.2.2. Formalisation

Un graphe non orienté connexe (pour tout couple de sommets du graphe, il existe un chemin de l'un à l'autre) étant donné, trouver un cycle de longueur minimale passant par tous les sommets.

3.2.3. Travail à réaliser

Faire une application qui crée un graphe connexe aléatoire, le dessine à l'écran. Un joueur humain doit deviner une solution au problème (le score peut être le temps mis, le nombre d'essais, la qualité de la solution trouvée, une fusion de plusieurs critères...)... ou toute variation sur le thème approuvée par vos enseignants.

Il y a deux difficultés algorithmiques dans cette application :

- créer un graphe connexe aléatoire,
- déterminer quelle est la longueur du plus court cycle passant par tous les sommets.

Votre travail consiste sur cette partie à créer l'application, et pour le second point à expérimenter un maximum de techniques. En semaine deux, vous devrez réappliquer ces techniques, mais cette fois sur un problème non classique (pas de documentation sur le net, moins d'aide des enseignants), c'est le moment de travailler avec filet !

1. Génération d'un graphe connexe aléatoire

Un algorithme possible qui ne génère pas une distribution uniforme sur l'ensemble des graphes connexes, mais qui est suffisant (voire plus intéressant) par rapport à notre objectif car il va permettre de régler aisément la densité du graphe. Il n'est pas évident que ce code soit particulièrement efficace, mais comme il n'est pas exécuté fréquemment, il n'est pas nécessaire de chercher à tout prix la vitesse d'exécution.

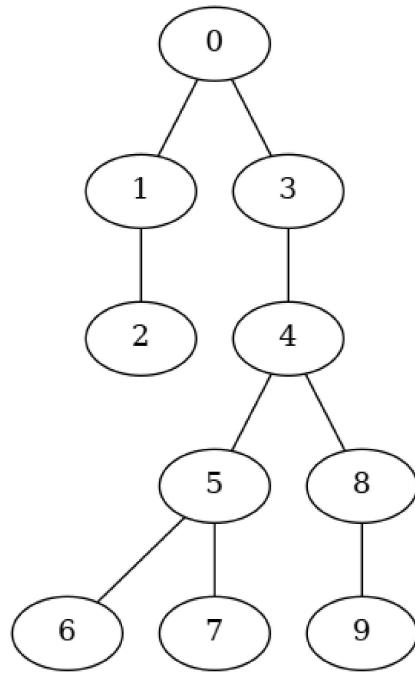
- Choisir un entier N , les sommets du graphes sont alors $\{0, 1, \dots, N - 1\}$,
- Phase 1 : création d'un arbre binaire couvrant du futur graphe (c'est cette phase qui permet de garantir que le graphe résultat sera connexe) La fonction python qui suit génère un arbre binaire sous forme d'une matrice d'adjacence, avec l'idée suivante :
 - prendre la liste des sommets, le premier sommet sera la racine,
 - couper ce qui reste en deux (gauche/droite), la partie gauche permettra de fabriquer le sous-arbre de gauche, la partie droite le sous-arbre de droite.

```
import random
def genere(matrice, bas, haut):
    if bas < haut:
        k = random.randint(bas+1, haut)
        matrice[bas][bas+1] = 1
        matrice[bas+1][bas] = 1
        if k+1 <= haut:
            matrice[bas][k+1] = 1
            matrice[k+1][bas] = 1
            genere(matrice, bas+1, k)
            genere(matrice, k+1, haut)

random.seed(42)
matrice = [[0]*N for _ in range(N)]
genere(matrice, 0, N-1)
return matrice
```

0	1	0	1	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	0	0	0
0	0	0	1	0	1	0	0	1	0	0
0	0	0	0	1	0	1	1	0	0	0
0	0	0	0	0	0	1	0	0	0	0
0	0	0	0	0	1	0	0	0	0	0
0	0	0	0	1	0	0	0	0	0	1
0	0	0	0	0	0	0	0	0	1	0

Ce que l'on interprète graphiquement par :



- Ajouter des arêtes au graphe de façon aléatoire. En fonction de la valeur de p choisie dans l'algorithme suivant, on obtiendra un graphe plus ou moins dense:
 - $p = 0 \rightarrow$ l'arbre obtenu précédemment sans altération,
 - $p = 1 \rightarrow$ graphe complet,
 - sur le graphique qui suit : $p = 0.1$

```

import random

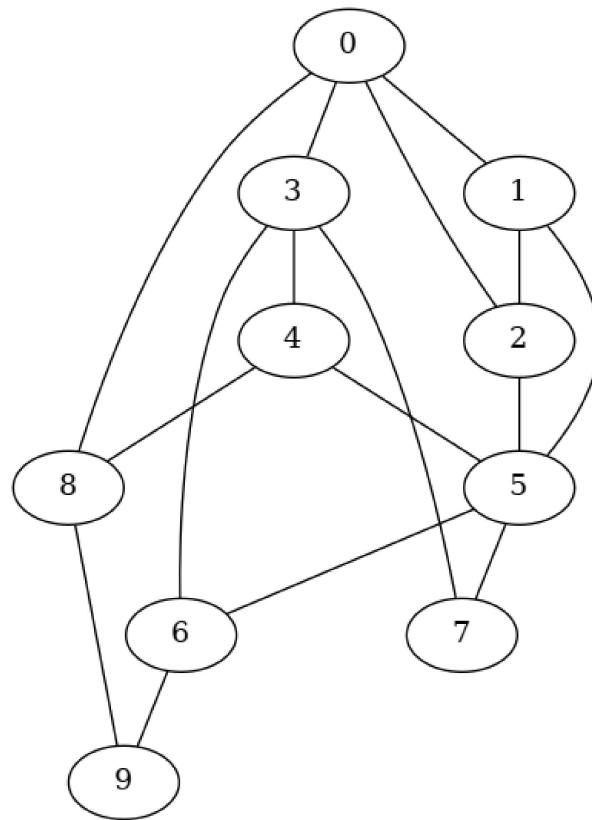
def genereGraphe(matrice, p):
    N = len(matrice)
    for i in range(N):
        for j in range(i+1, N):
            if random.random()<p:
                matrice[i][j] = 1
                matrice[j][i] = 1

random.seed(42)
genereGraphe(matrice, p)
return matrice
  
```

0	1	1	1	0	0	0	0	0	1	0
1	0	1	0	0	1	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
1	0	0	0	1	0	1	1	0	0	0
0	0	0	1	0	1	0	0	0	1	0
0	0	0	0	1	0	1	1	0	0	0
0	0	0	0	0	1	0	0	0	0	1

0	0	0	0	0	1	0	0	0	0
0	0	0	0	1	0	0	0	0	1
0	0	0	0	0	0	0	0	1	0

qui a pour représentation graphique :



2. Positionnement du graphe dans le plan

Pour obtenir un graphe positionné dans le plan:

- choisir les coordonnées de tous les nœuds dans l'espace image,
- appliquer l'algorithme précédent pour choisir quelles sont les arêtes qui existent,
- calculer les distances euclidiennes qui serviront de valuations aux arcs.

3.2.4. Recherche d'un chemin particulier

Le but est ici de trouver la longueur d'un cycle de longueur minimale passant par tous les sommets (ce n'est pas nécessairement un cycle hamiltonien, le chemin peut passer plusieurs fois par le même sommet).

3.2.5. Se ramener à un cas classique

Si on désire se ramener à la recherche d'un cycle hamiltonien de longueur minimale, on travaille sur le graphe G' construit comme suit :

- les sommets de G' sont les mêmes que ceux de G ,

- pour chaque couple de sommets (u, v) , créer dans G' une arrête reliant u et v dont la valuation est la longueur du plus court chemin dans G de u à v .

Pour le deuxième point, il existe un algorithme classique pour calculer la distance entre tout couple de sommets du graphe : Floyd-Warshall, de complexité $O(n^3)$ qui s'applique à tout graphe sans cycle de coût strictement négatif :

- nommer les sommets $0..n - 1$,
- $d_{i,j,k}$ est la longueur du plus court chemin du noeud i au noeud j parmi les chemins où les noeuds intermédiaires sont dans $[0..k - 1]$ (tout l'algorithme tient dans cette définition, la relire plusieurs fois, le reste en est la conséquence !!).

Avec cette définition :

- Initialisation : $d_{i,j,0}$ est la distance de i à j si on ne s'autorise aucun noeud intermédiaire et donc s'il existe une arrête de i à j (ceci est calculable avec la matrice d'adjacence) ; on note que lorsqu'il n'y a pas d'arrête entre i et j , cette distance est $+\infty$,
- But à atteindre : $d_{i,j,n}$ est la distance de i à j si on s'autorise le noeuds intermédiaires dans $[0..n - 1]$, c'est à dire tous les noeuds... et c'est donc le plus court chemin de i à j dans le graphe,
- pour $0 < k < n$, il existe une formule simple permettant de calculer $d_{i,j,k}$ en fonction des $d_{u,v,k-1}$. Le chemin le plus court de i à j ayant ses noeuds intermédiaires dans $[0..k - 1]$ passe... ou ne passe pas par le noeud $k - 1$.
 - S'il ne passe pas par le noeud $k - 1$, c'est alors le même que le plus court chemin ne s'autorisant comme noeud intermédiaires que des noeuds dans $[0..k - 2]$ et sa longueur est $d_{i,j,k-1}$,
 - s'il passe par le noeud $k - 1$ il ne peut y passer qu'une fois (c'est ici qu'on utilise l'absence de cycle de coût strictement négatif : s'il passait deux fois par un même noeud, en supprimant dans le chemin cette boucle on aurait un chemin qui serait au moins aussi court, ce qui n'est donc pas possible), et donc le chemin s'écrit $i \rightarrow k - 1 \rightarrow j$ et les chemins de i à $k - 1$ ainsi que de $k - 1$ à j ont leurs noeuds intermédiaires uniquement dans $[0..k - 2]$. La longueur de ce chemin est $d_{i,k-1,k-1} + d_{k-1,j,k-1}$,
 - La distance $d_{i,j,k}$ est donc le minimum entre $d_{i,j,k-1}$ et $d_{i,k-1,k-1} + d_{k-1,j,k-1}$

Il est à noter qu'avec cette construction du graphe, le cycle de longueur minimale passant par tous les sommets est nécessairement hamiltonien, car l'inégalité triangulaire y est vérifiée $d(\text{Noeud}_A, \text{Noeud}_B) \leq d(\text{Noeud}_A, \text{Noeud}_C) + d(\text{Noeud}_C, \text{Noeud}_B)$.

3.3. Méthodes d'optimisation

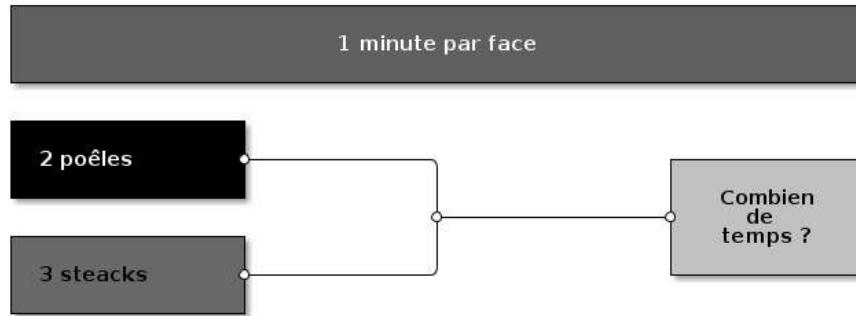


Figure 10 : un problème simple d'optimisation

Ce paragraphe a pour objectif de vous présenter quelques méthodes classiques qui peuvent permettre d'optimiser la fonction ψ . Beaucoup de ces méthodes sont ce qu'on appelle des 'métaheuristiques' : ce sont des techniques d'optimisation générales (elles s'appliquent à n'importe quel problème), conçues dans le but de s'attaquer à des problèmes difficiles (les meilleures techniques de résolution exacte ne sont pas accessibles car trop longues), et qui nécessitent d'être adaptées à chaque problème particulier pour pouvoir être utilisées. Ce document ne fera cependant pas de différence dans le vocabulaire entre métaheuristique et algorithme.

De nombreuses méthodes utilisent une notion de voisinage: soit \mathbb{F} une famille de fonctions, y est un voisin de x s'il existe une transformation f de \mathbb{F} telle que $f(x) = y$.

Par exemple :

- dans un graphe, un nœud a pour voisins tous les nœuds vers qui il existe une transition,
- dans un jeu vidéo fonctionnant avec un déplacement par case, les voisins d'une case sont les cases N, S, E, O si on ne peut se déplacer que dans 4 directions, ou N, NE, E, SE, S, SO, O, NO si on peut également se déplacer selon les diagonales,
- dans le jeu du taquin, une situation est données par l'ensemble des pièces du taquin, et ses voisins sont les situations accessibles après un déplacement,
- dans le cas du voyageur de commerce, les situations sont les cycles passant par tous les sommets. On peut s'autoriser à permute deux sommets d'un cycle (d'autres opérations sont possibles, mais celle-ci quoi que simple est déjà intéressante), les voisins sont les cycles accessibles après une permutation.

Certaines méthodes d'optimisation fonctionnent en raffinant petit à petit des solutions, en se déplaçant de voisins en voisins (recherche locale), d'autres ont une approche plus constructive et cherchent à bâtir des solutions qui sont de mieux en mieux , mais sans pour autant naviguer dans l'espace des solutions (méthode exhaustive, colonies de fourmis). Certaines méthodes essaient de tirer partie des deux approches en les hybridant, soit à chaque itération, soit en les utilisant dans des phases distinctes.

3.3.1. Méthode exhaustive

La première idée pour résoudre un problème d'optimisation est d'énumérer toutes les solutions possibles et de conserver celle qui est la meilleure.

Dans le cas du voyageur de commerce, on peut calculer tous les cycles possibles, et conserver un de ceux qui est de longueur minimale. Remplissez la table suivante, dont la première colonne est le nombre de sommets d'un graphe complet, la seconde le nombre de cycles à considérer et la troisième le temps nécessaire pour traiter la situation sous l'hypothèse où l'on serait capable de traiter 10^9 cycles par seconde.

nombre de sommets	nombre de cycles possibles	temps de traitement
5		
10		
15		
20		
30		
40		
50		
60		
70		
80		
90		
100		
1000		

3.3.2. Recherche locale

La recherche locale fixe une situation initiale (souvent générée aléatoirement), puis à chaque itération de l'algorithme un voisin de la situation actuelle est choisi qui va devenir la nouvelle situation actuelle.

Condition mathématique pour choisir la liste des transformations possibles :

Depuis n'importe quel état, il existe une suite finie de transformations qui permet d'accéder à une solution.

En général, comme on ne sait pas où se trouve la solution et que l'on va générer aléatoirement la situation initiale, cette condition devient :

Depuis n'importe quel état, il existe une suite finie de transformations qui permet d'accéder à n'importe quel état.

Pour choisir ce nouveau voisin, on peut:

- prendre le meilleur parmi tous les voisins (méthode gloutonne exhaustive),
- parcourir les voisins dans un ordre au hasard, et dès qu'il y en a un qui est meilleur que la situation actuelle, l'accepter (méthode gloutonne stochastique)

- avec une faible probabilité choisir un voisin de façon totalement aléatoire, le reste du temps utiliser une des méthodes précédentes,
- prendre une des méthodes précédentes mais en s'interdisant certains voisins (méthode tabou),
- prendre un voisin, et même s'il est moins bon, l'accepter de temps en temps, en diminuant cette probabilité au fil des itérations (recuit simulé).

Les deux premières méthodes sont des méthodes de *spécialisation* qui possèdent l'avantage de la vitesse, mais le problème est qu'elles ne convergent que vers un minimum local. Elles ne convergent donc vers une solution que :

- si on a de la chance (le point de départ était dans la bonne zone),
- ou si la fonction est convexe (auquel cas, minimum local = minimum global).

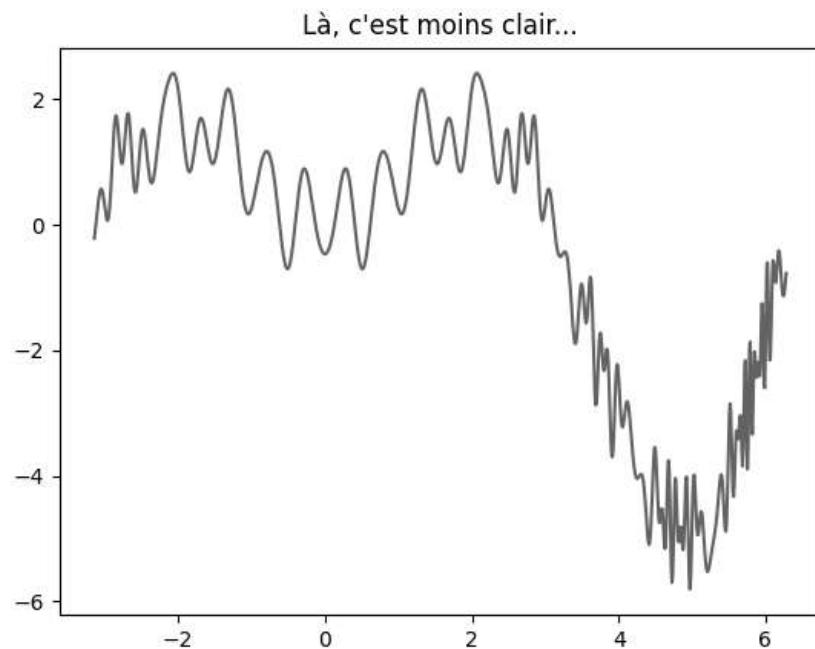
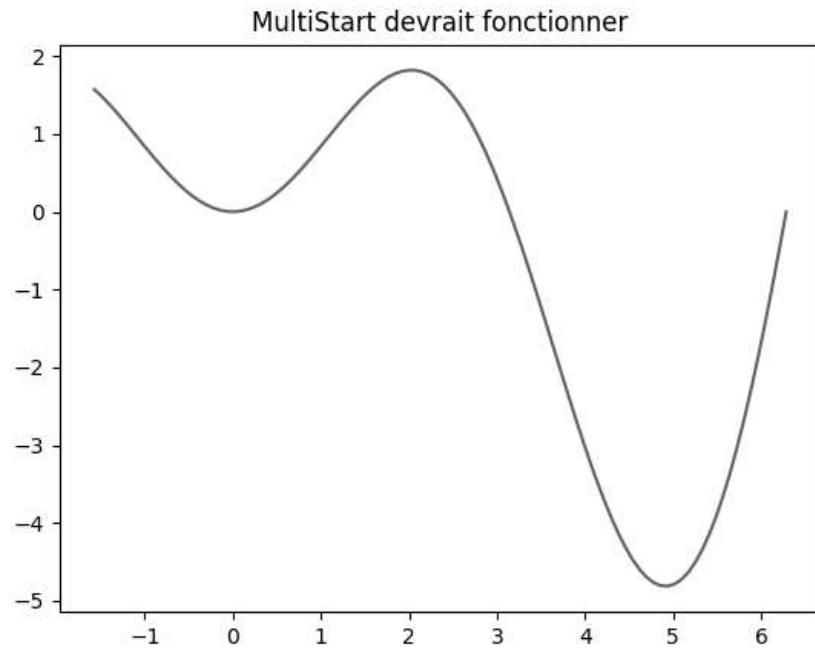
Les méthodes de recherche locales ont une qualité qui dépend en particulier de deux paramètres:

- les voisinages : partant d'une situation, quelles sont les situations atteignables,
- la fonction à minimiser : on aimeraient que l'augmentation ou la diminution de la valeur de la fonction d'évaluation nous indique au mieux le voisin à choisir. Si on prend une fonction qui vaut 0 sur la solution que l'on cherche et 1 partout ailleurs, une recherche locale ne sera pas très efficace...

Ces méthodes gloutonnes sont souvent utilisées, mais hybrides à d'autres techniques. Il est possible d'augmenter les chances d'obtenir un résultat satisfaisant en :

- augmentant la taille des voisinages (on s'autorise plus d'actions pour passer d'un état à un autre état). Cette idée est très efficace, mais la méthode devient de plus en plus lente en devenant de plus en plus exhaustive, on l'utilise donc en général lorsqu'un optimum local est atteint dans le but de le quitter pour une zone meilleure,
- les relançant plusieurs fois en conservant le meilleur des résultats (méthode *multistart*): on essaie d'augmenter les chances que parmi les minimums locaux explorés, il y en ait un qui soit de valeur proche de celle du minimum global.
- relançant la recherche à plusieurs reprises en prenant comme point de départ à chaque nouvelle recherche le résultat de la recherche précédente après l'avoir altéré (*recherche itérée*)

Sur les graphiques suivants, on observe une première situation où la vallée qui nous intéresse est large, et où relancer plusieurs fois l'algorithme va rapidement porter ses fruits, et sur le graphique suivant, chaque vallée est très resserrée, et donc pour obtenir le minimum global, il faudrait que la situation de départ soit dès le départ dans la bonne vallée (ce qui nécessite tout de même pas mal de chance !).



Ce phénomène n'est pas anecdotique, et se produit en particulier très fréquemment lorsqu'on travaille sur un espace discret. Il n'est en général pas évident de savoir si un espace est lisse ou s'il est très accidenté. La forme de l'espace dépend des voisinages que l'on choisit : si on trouve la bonne famille de transformations, l'espace est alors relativement lisse et des algorithmes gloutons

(plus ou moins améliorés) peuvent donner d'excellents résultats. Si en revanche on ne parvient pas à trouver de bonne famille de transformations, les méthodes gloutonnes, elles ne donnent que des résultats très pauvres.

1. Un exemple : une phrase auto-référente



Figure 11 : Un livre à l'origine de beaucoup de vocations en IA, avec des problèmes d'autoréférence

Le problème ici est de déterminer les valeurs des variables entières dans $[1..12]$ de a_0, \dots, a_9 , de façon à rendre la phrase vraie.

Cette phrase contient :

- a_0 chiffres 0,
- a_1 chiffres 1,
- a_2 chiffres 2,
- a_3 chiffres 3,
- ...
- a_9 chiffres 9.

Si la famille de transformations est 'choisir une des lignes et modifier une la valeur du a_i de la ligne', et la fonction objectif (que l'on minimise ici) est le nombre de a_i incorrects, alors la fonction objectif semble extrêmement chaotique... Est-ce vraiment le cas ? Pour le savoir on code une résolution avec une méthode gloutonne exhaustive, et on examine le résultat.

2. Expérimentation

- Une situation est représentée par une liste de 10 éléments, en position i se trouve une valeur de a_i ,
- la fonction `printL` affiche la phrase au format de l'exemple ci-dessus,

```
def printL(L):
    print("Cette phrase contient :")
    for i, v in enumerate(L):
        print("\t-", v, "chiffres", i)
```

- les fonctions `energy1` et `energy2` sont des fonctions d'évaluation, elles valent toutes les deux 0 lorsqu'une solution est trouvée (et sont strictement positives partout ailleurs), `energy1` semble plus précise que `energy2` (mais cela ne signifie pas qu'elle est meilleure),

```
def energy1(L):
    nbErrors = 0
    tabCount = [1] * 10
    for v in L:
        for c in str(v):
            tabCount[int(c)] += 1

    for v, c in zip(L, tabCount):
        nbErrors += abs(v - c)
    return nbErrors

def energy2(L):
    nbErrors = 0
    tabCount = [1] * 10
    for v in L:
        for c in str(v):
            tabCount[int(c)] += 1

    for v, c in zip(L, tabCount):
        nbErrors += v != c
    return nbErrors
```

- les fonction `generateState` servent à définir les voisinages, en indiquant vers quel état il est possible de transiter à l'itération suivante:
 - `generateState1` : choisit un des a_i et lui assigne une valeur aléatoire,
 - `generateState2` : choisit un des a_i ,
 - avec une probabilité de 1% lui assigne une valeur aléatoire
 - avec une probabilité de 99% lui assigne une valeur v qui est la somme d'une valeur aléatoire dans $[-2..2]$ et de la valeur qui est celle que l'on aimeraient mettre pour corriger la phrase au mieux (on compte combien il y a actuellement de i sans compter ceux qui sont dans a_i).
 - `generateState3` : choisit aléatoirement un des a_i et essaie toutes les valeurs possibles, renvoie celui qui minimise l'erreur,
 - `generateState4` : essaie pour chacun des a_i depuis la valeur -1 à la valeur +1 et conserve la meilleure ; à moins que l'on soit sur les bords du domaine, le nombre de possibilités testées ici est $3^{10} = 59049$
 - `generateState5` : essaie toutes les possibilités sur une variable ; attention, cette fonction devra être encapsulée pour définir quelle variable doit être optimisée.

```

def generateState1(L):
    M = list(L)
    M[random.randint(0, 9)] = random.randint(1, 12)
    return M

def generateState2(L, bord = 2):
    M = list(L)
    pos = random.randint(0, 9)
    if random.random() < 0.01:
        a = random.randint(1, 12)
    else:
        a = 1
    for v in L:
        a += str(v).count(str(pos))
        a -= str(pos).count(str(pos))
    M[pos] = min(12, max(0, a + random.randint(-bord, bord)))
    return M

def generateState3(L, energy):
    M = list(L)
    bestM = list(M)
    bestE=energy(M)
    pos = random.randint(0, 9)
    for i in range(1,13):
        M[pos] = i
        E= energy(M)
        if E<bestE:
            bestE = E
            bestM = list(M)
    return M

def generateState4(L, energy, bord = 1):
    bestE = energy(L)
    bestL = list(L)
    def positionI(L,i):
        nonlocal bestE
        nonlocal bestL
        for v in range(max(1,L[i]-bord), 1+min(12,L[i]+bord)):
            L[i] = v
            E = energy(L)
            if E < bestE:
                bestE = E
                bestL = list(L)
        if i<9:
            positionI(L,i+1)

    positionI(L,0)
    return bestL

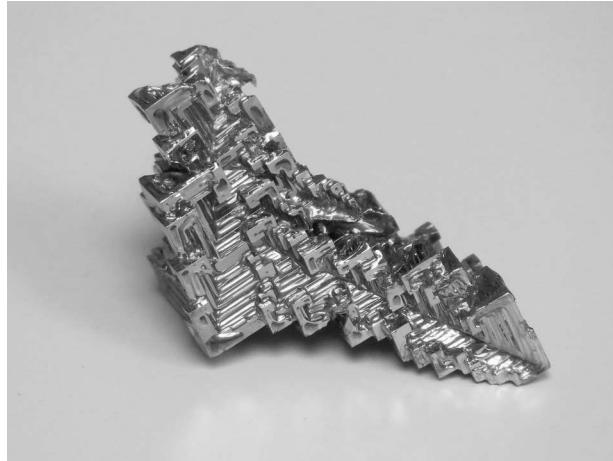
def generateState5(state, pos, energy):
    state[pos] = 1
    bestValue = 1
    state[pos] = bestValue
    bestEnergy = energy(state)
    for value in range(2, 12 + 1):
        state[pos] = value
        E = energy(state)
        if E < bestEnergy:
            bestValue = value
            bestEnergy = E
    state[pos] = bestValue

```

Il ne reste alors plus qu'à tester et comparer les différentes méthodes sur ce problème, car malheureusement on ne pourra pas en conclure sur ce qui se passe sur un autre problème, car comme maintenant vous devez vous en douter, le choix des transformations (déterministe/

stochastique, taille du voisinage,...) ont énormément d'influence sur le résultat et sont dépendants de la structure du problème.

3. Recuit simulé : une recherche locale où on peut remonter



Le recuit simulé est une méthode qui permet de s'attaquer au problème des minimums locaux trop fréquents. Dans cet algorithme la fonction à optimiser (ici minimiser) s'appelle l'énergie du système et les situations sur lesquelles on calcule cette énergie des *configurations*. Ce vocabulaire est originaire de la métallurgie et de la thermodynamique qui ont été les inspiratrices de cette méthode. Son principe est :

- partir d'un état quelconque,
- essayer une transformation sur cet état,
 - si l'état atteint est d'énergie plus faible que l'état actuel, alors il devient le nouvel état courant,
 - si l'état atteint est d'énergie supérieure à l'état actuel, il ne devient le nouvel état qu'avec une certaine probabilité.

Le passage de configuration en configuration se poursuit indéfiniment, en diminuant au fil du temps la probabilité d'accepter des configurations moins avantageuses.

La façon la plus classique de déterminer le passage (ou non) d'un état actuel à un nouvel état est la *règle de Metropolis* :

- si l'énergie du nouvel état est strictement inférieure à l'énergie de l'état actuel, on accepte la transition (lorsqu'on trouve mieux que ce que l'on a, sans se poser de question, on accepte),
- sinon, ce n'est qu'avec la probabilité $p = e^{\frac{-\Delta E}{T}}$ que l'on accepte la transition vers ce nouvel état ($\Delta E = E_{\text{nouvel état}} - E_{\text{ancien état}}$). C'est cette possibilité de 'remonter' qui va permettre de quitter un minimum local.

Le paramètre T de la formule s'appelle la *température* (strictement positif). A haute température, la probabilité d'accepter un état moins bon que celui où l'on était est haute, à basse température elle est infime. Ce paramètre de température va diminuer au fil de l'exécution de l'algorithme, rendant la possibilité d'accepter des transitions défavorables de plus en plus faibles (pour T proche de 0, on est revenu à un algorithme glouton).

Ce qui rend cet algorithme très intéressant est que sous certaines hypothèses, on garantit qu'avec une probabilité de 1, il converge vers un minimum **global** de la fonction. Les conditions de validité de cette affirmation sont que :

- chaque état doit être accessible en un nombre fini d'étapes et ce avec une probabilité non nulle à tout instant de l'algorithme,
- le nombre d'états doit être au plus dénombrable.

Le seul souci dans ce théorème est que pour garantir cette propriété, la température doit décroître très lentement. Un théorème de Hajek prouve que si on note h^+ la différence entre l'énergie maximale possible et l'énergie minimale possible que peut atteindre le système, pour toute valeur $h \geq h^+$, si on choisit comme température à l'itération n $T_n = \frac{h}{\log n}$ alors la décroissance est suffisamment lente pour que la propriété soit validée. Malheureusement... cette décroissance est **terriblement** lente, et n'est en général pas utilisable en pratique. On va donc la faire décroître plus rapidement, ce qui va faire perdre le théorème, et pratiquement, risque d'entraîner une convergence vers un minimum qui n'est pas global (mais que l'on espère d'énergie proche de celle du minimum, sous condition de ne pas faire descendre la température trop vite). Il existe plusieurs *plans de recuit* (c'est le nom que l'on donne à la méthode choisie pour réaliser la baisse de la température). On peut :

- diminuer la température à chaque itération,
- conserver la même température sur un *pallier*, et lorsqu'une certaine condition est vérifiée (par exemple : « il y a eu au moins 10 changements d'état ou on a essayé 1000 configurations ») diminuer la température pour passer au pallier suivant.

Pour diminuer la température, deux méthodes sont fréquemment utilisées :

- décroissance linéaire : $T_n = \frac{A}{B+n}$ où $A > 0$ et $B \geq 0$ sont des constantes (à trouver par expérimentation),
- décroissance géométrique $T_{n+1} = \alpha \times T_n$ avec $0 < \alpha < 1$ et α très proche de 1 (0.999 par exemple, mais à régler par expérimentation, et évidemment plus petit si on utilise des palliers que si on n'en utilise pas).

1. Astuce

Parfois, l'algorithme s'égare dans des zones peu pertinentes (la température est descendue trop vite, et il va être difficile de quitter la zone dans laquelle l'état actuel se trouve). Pour remédier à cela on peut :

- diminuer moins vite la température (solution simple, mais qui parfois est très coûteuse en temps de calcul),
- relancer plusieurs fois l'algorithme, en stockant les meilleurs résultats, et en les choisissant comme graine de départ que l'on peut utiliser lorsque l'algorithme se 'perd'.

2. Phrases autoréférentes et recuit

Le code suivant permet d'obtenir des solutions au problèmes présenté précédemment des phrases auto référentes:

- en entrée :
 - l'état actuel,
 - la température,
 - la fonction d'énergie.
- en sortie :
 - le nouvel état
 - un booléen valant True si et seulement si la transition a été acceptée

```

def generateStateSimulatedAnnealing(state, T, energy):
    E = energy(state)
    newState = generateState(state) # can use any generateState
    E(newState)
    if E(newState) < E:
        return newState, True, E(newState)
    else:
        deltaE = E(newState) - E
        p = exp(-deltaE / T)
        if random.random() < p:
            return newState, True, E(newState)
        else:
            return state, False

```

Cette fonction va être appelée, avec la température qui va décroître au fil des itérations. Ici, le programme de recuit met en place des paliers : la température est constante sur un palier, on change de palier si on y est resté assez longtemps, ou si on a déjà accepté beaucoup de transitions.

```

def simulatedAnnealing(initialTemperature = 30, stageLength = 1000, nbAcceptations = 100, energ
T = initialTemperature # This value of 30 comes from the estimation of the difference betwe
state = [random.randint(0, 13) for _ in range(10)]

nbAccepts = 0
i = 0
while True: #infinite loop, maybe it would be fine to leave it one day and obtain results :
    state, accepted = generateStateSimulatedAnnealing(state, T, energy1)
    nbAccepts += accepted # adds 0 or 1 depending of the boolean value
    if nbAccepts == nbAcceptations or i % stageLength == 0:
        print(state)
        print("\tItération :", i, "\tTempérature :", T, "\tEnergie :", energy(state))
        nbAccepts = 0
        T = T * 0.9994
    i += 1

```

4. Méthode tabou



Figure 12 : La méthode tabou : interdire les états que l'on vient vient juste d'explorer

La méthode tabou est un algorithme de recherche locale, qui comme le recuit simulé vise à autoriser à transiter de temps en temps vers des états de moindre intérêt. Il utilise une file d'états de taille fixe K qui contient à tout instant la liste des K états dernièrement visités. A chaque itération de l'algorithme :

- si tous les états accessibles sont dans la file des tabous, il en choisit un au hasard (normalement cette situation est extrêmement rare, voire ne se produit jamais, et si elle se produit on peut préférer relancer l'algorithme complètement),
- sinon, le nouvel état est :
 - le meilleur de tous les états qui ne sont pas dans la file des tabous (choix 1),
 - le premier état que l'on trouve qui est meilleur que l'état actuel (et qui n'est pas tabou), ou le meilleur état qui n'est pas tabou.

Par conséquent, si le meilleur état accessible n'est pas bon... il est tout de même le meilleur et il est donc accédé.

Une des difficultés de cette méthode est de configurer la taille de la liste des tabous:

- si elle est trop courte, il n'est pas possible de remonter pendant longtemps (en particulier plus on élève la dimension, plus il y a de sentiers et de façons de descendre dans une vallée locale), et donc quitter un minimum local large est impossible,
- si elle est trop longue, il n'y a pas de possibilité d'essayer de petites variations qui permettraient de découvrir un petit chemin qui mène à un bon minimum.

On se rend compte que la longueur de la liste des tabous devrait dépendre de :

- la forme de la fonction d'évaluation au voisinage du point courant : plus elle est plate, plus la liste est longue
- du moment de l'algorithme : lorsqu'on cherche une bonne zone, on privilégie une liste de tabous longue ; lorsqu'on cherche à améliorer au mieux une solution (et donc qu'on a déjà choisi la zone) on préférerait utiliser une liste de tabous plus courte.

On peut améliorer les résultats de la méthode tabou en s'inspirant de l'idée de tabou, en introduisant des tabous moins rigides qui défavorisent des configurations qui ne semblent pas prometteuses:

- cette zone a déjà été très explorée
- ce type de solutions a peu de chances d'aboutir à de bons résultats (pour des raisons logiques, ou parce que statistiquement depuis le début de l'exécution ce type de solution n'a mené à aucune solution de qualité)
- ...

Ce type de tabous, plus difficiles à mettre en œuvre, permettent d'obtenir des comportements faisant intervenir une guidance à moyen ou à long terme. On peut en particulier :

- favoriser les endroits / motifs qui ont été les plus fructueux jusqu'ici : on réalise aux bons endroits une *intensification* des recherches,
- favoriser la *diversification* : si certaines zones n'ont jamais été approchées, il peut être intéressant de se téléporter dans l'une d'entre elles plutôt que de rester dans des zones très connues.

On peut alors remarquer qu'en étendant cette notion de tabou, le recuit simulé devient alors un cas particulier de la recherche tabou, les états tabous devenant en fait 'tabous avec une certaine probabilité'.

Le code suivant illustre la méthode la plus simple de recherche tabou, qui n'utilise que la version originale : la liste des derniers états sont totalement tabous.

- La première version converge vers un minimum local rarement pertinent (à peu près indépendamment de la longueur de `tabus`) et n'est pas très efficace,
- la seconde converge bien, mais n'a pas vraiment besoin de la méthode tabou, il suffit de la relancer quelques fois pour avoir une solution, avec une liste de tabous ne contenant que la position actuelle ($K=1$).

Sur ce problème, cette méthode ne semble guère pertinente, mais cela ne la remet pas en cause pour autant sur d'autres problèmes.

```

def generateState4Tabu(state, tabus, energy, bord=1):
    bestStates = []
    bestE = 1000 # infinity
    tabus[0][tabus[1]] = list(state)
    tabus[1] = (tabus[1] + 1) % len(tabus[0])

    def positionI(state, i):
        nonlocal bestE
        nonlocal bestStates
        left = max(1, state[i] - bord)
        right = min(12, state[i] + bord)
        L = []
        for v in range(left, right + 1):
            state[i] = v
            if state not in tabus[0]:
                E = energy(state)
                if E == bestE:
                    bestStates.append(list(state))
                elif E < bestE:
                    bestE = E
                    bestStates = [list(state)]
            if i < 9:
                positionI(state, i + 1)

    positionI(state, 0)
    if len(bestStates) == 0:
        print("Pas d'état accessible à cause des tabous")
        bestState = [random.randint(1, 12) for _ in range(10)]
    else:
        bestState = random.choice(bestStates)
    return bestState

def generateState7Tabu(state, tabus, energy, bord=1):
    bestStates = []
    bestE = 1000 # infinity
    tabus[0][tabus[1]] = list(state)
    tabus[1] = (tabus[1] + 1) % len(tabus[0])

    def bestInPos(state, i):
        nonlocal bestE
        nonlocal bestStates
        for v in range(1, 13):
            state[i] = v
            if state not in tabus[0]:
                E = energy(state)
                if E == bestE:
                    bestStates.append(list(state))
                elif E < bestE:
                    bestE = E
                    bestStates = [list(state)]

    def bestPos(state):
        tabCount = [1] * 10
        for v in state:
            for c in str(v):
                tabCount[int(c)] += 1

        nbErrors = -1
        pos = []
        for (i, (v, c)) in enumerate(zip(state, tabCount)):
            if abs(v - c) > nbErrors:
                nbErrors = abs(v - c)
                pos = [i]
            elif abs(v - c) == nbErrors:
                pos.append(i)

    bestPos(state)

```

```

        return random.choice(pos)

bestInPos(state, bestPos(state))
if len(bestStates) == 0:
    print("Pas d'état accessible à cause des tabous")
    bestState = [random.randint(1, 12) for _ in range(10)]
else:
    bestState = random.choice(bestStates)
return bestState


def tabu(K=10, energy=energy1):
    state = [random.randint(1, 12) for _ in range(10)]
    tabuQueue = [[None] * K, 0] # queue of size K : the first element is the list of tabus, the second is the cpt
    cpt = 0
    mini = 1000 # infinity
    found = set() # will contain the different solutions (E = 0)
    while True: # cpt < 10:
        state = generateState7Tabu(state, tabuQueue, energy1, 1) # this function or the other
        E = energy1(state)
        if E < mini:
            mini = E
            print(state, end="\t<< ")
            print("Itération :", cpt, "\tEnergie :", E)
        elif E == 0:
            found.add(str(state))
            print("Trouvé :", state)
            print(found)
            tabuQueue = [[None] * K, 0]
            state = [random.randint(1, 12) for _ in range(10)]
            print(state)
        elif cpt % 10000 == 0: # just to have something on screen from time to time
            print(state, end="\t--- ")
            print(tabuQueue[0])
            print("Itération :", cpt, "\tEnergie :", E)
        cpt += 1

```

3.3.3. Méthodes constructives

Les méthodes constructives visent à construire des solutions directement, sans avoir l'idée de passer de solution en solutions. Elles peuvent ou non tenir compte des expériences précédentes pour construire leur solution.

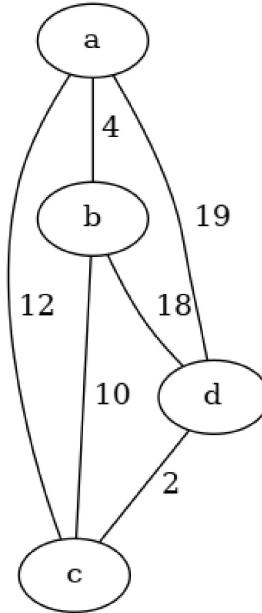
1. Méthodes gloutonnes

Ces méthodes sont souvent utilisées pour initialiser un autre algorithme en fournissant un point de départ qui n'est pas trop mauvais. Comme ce sont des méthodes qui font le 'meilleur' choix par rapport à un critère donné sans jamais remettre en cause les choix précédents, elles sont généralement très peu coûteuses ; si de plus l'heuristique choisie pour décider ce qui est le mieux est assez pertinente, on peut obtenir des solutions qui, au moins sur les instances simples, sont assez proches de l'optimum.

Dans le cas du voyageur de commerce, une heuristique gloutonne assez simple consiste à choisir comme prochaine ville du circuit la ville la plus proche qui ne fait pas encore partie de la tournée, et quand toutes les villes ont été incorporées, revenir au point de départ.

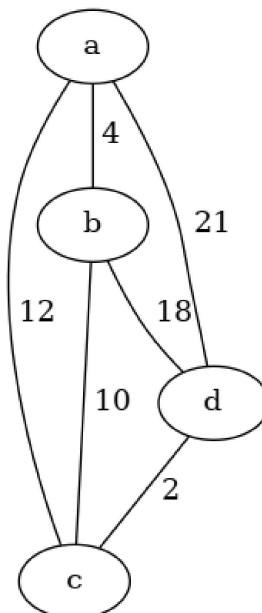
Tout se passe très bien dans ce premier cas avec un coût total de 35 qui est effectivement l'optimum :

- départ de a
- le plus proche est b
- puis c
- puis d
- et finalement a



Mais sur celui qui suit, assez similaire, le coût total est 37 alors que le meilleur coût est 36 (acdba) :

- départ de a
- le plus proche est b
- puis c
- puis d
- et finalement a



2. Colonies de fourmis

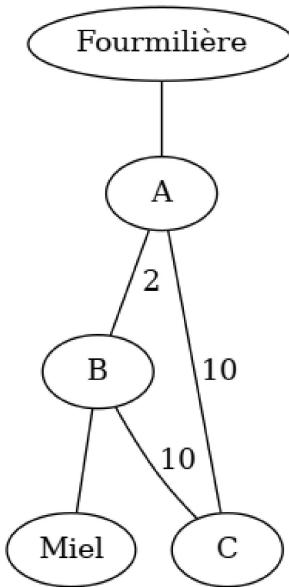


Figure 13 : Est-ce vraiment le meilleur chemin ?

1. Idée originelle

L'observation de fourmis a permis de remarquer que lorsqu'elles trouvent une source de nourriture, elles partagent l'information avec le reste de la fourmilière ou les fourmis proches. Afin de la rapporter à la fourmilière, le trajet suivi par les fourmis s'améliore. Il y a émergence d'une *intelligence collective* qui a optimisé le circuit. Pour obtenir ce résultat, les fourmis vont laisser des traces, appelées phéromones (sortes d'hormones) sur les chemins qu'elles empruntent. Examinons comment un mécanisme *local* très simple va permettre de faire émerger une solution *globale* de qualité.

On s'intéresse à la situation représentée par le graphe suivant où les valuations des arrêtes représentent les temps de parcours.



Une fourmi part de la fourmilière, à chaque intersection elle choisit au hasard l'arc qu'elle parcourt, en déposant à chaque déplacement un peu de phéromones. Un fois le miel découvert elle revient à la fourmilière en déposant le long du chemin des phéromones. Une autre fourmi arrive, quand elle doit choisir un arc, elle choisit chaque arc avec un hasard légèrement biaisé par la présence de phéromones qui l'attirent. une fois arrivée au miel, elle repart en déposant des phéromones sur son chemin du retour.

On se place dans la situation où :

- les chemins ACB et AB sont au départ équiprobables,
- à chaque date de temps, 1 fourmi emprunte BA et 1 fourmi emprunte BCA.

Après 10 minutes, 9 fourmis sont arrivées en A et ont déposé des phéromones sur le trajet AB, aucune n'est encore arrivée par BCA, le chemin le plus court est plus renforcé que l'autre, et le phénomène va s'auto-entretenir, et renforcer de plus en plus le chemin le plus court.

Si malgré cela (le hasard n'a pas bien fait les choses...), le plein régime arrive, il y a alors autant de fourmis qui arrivent par chacun des deux chemins par unité de temps, mais comme les phéromones sont volatiles, et le temps plus long pour parcourir BCA, il y a donc plus de phéromones restantes sur le trajet BA que sur le trajet BCA, le phénomène s'auto-amplifiant, le chemin AB deviendra alors de plus en plus prépondérant.

2. Adaptation à l'informatique

La météuristiche des colonies de fourmis est une méthode constructive qui va relancer plusieurs fois la construction, et essayer d'extraire des informations statistiques des constructions précédemment réalisées. Le point de départ est une heuristique gloutonne qui va être améliorée afin de tenir compte des réussites et des échecs des constructions précédentes.

Chaque *fourmi* va construire une solution. La colonie complète part à la recherche d'une solution :

- Après avoir trouvé sa solution, chaque fourmi indique sur tout le trajet qu'elle a suivi sa satisfaction par des phéromones qu'elle dépose sur chacun des choix qu'elle a effectué, ces phéromones sont d'autant plus puissantes qu'elle est satisfaite de sa solution.
- La nuit passe, une partie des phéromones s'évapore.
- Le lendemain chaque fourmi repart, mais lorsqu'elle effectue un choix, elle va prendre en compte son a priori (la méthode gloutonne), et la quantité de phéromones sur chacun des choix. Une fois qu'elle aura construit sa solution, elle reviendra comme son aînée déposer une quantité de phéromones proportionnelle à sa satisfaction sur chacun des choix qu'elle a effectué.

Au fil des jours, les choix qui produisent les meilleurs résultats vont amasser des phéromones, et il va y avoir émergence des meilleures constructions...

3. Application de l'idée aux phrases auto-référentes

- `apriori` a pour entrée un état et une position et renvoie la liste des évaluations obtenues en parcourant toutes les valeurs possibles en position `pos`, les autres variables étant inchangées.

```
def apriori(state,pos):
    qualities =[1000]*12
    save = state[pos]
    for v in range(1,13):
        state[pos] = v
        qualities[v-1] = energy1(state)
    state[pos] = save
    return qualities
```

- `f` calcule $x, n \rightarrow \frac{1}{x^n}$. Plus n est grand, plus les petites différences sont amplifiées. Pour n très grand, seule la plus grande valeur est prise en compte, pour n proche de 0, les valeurs n'ont plus aucune importance.

```
def f(x, power):
    return 1/(x**power)
```

- `construction` part d'un état peu favorable fixé, choisit un ordre aléatoire (si on décommente la ligne) pour parcourir les variables et va mettre à jour en prenant en compte deux considérations qui vont définir l'intérêt de chaque choix :

- la qualité estimée par la fonction `apriori`,
- la quantité de phéromone sur chacune des alternatives.

Une fois l'intérêt calculé, le choix de l'action est obtenu par un tirage aléatoire privilégiant les alternatives de plus grand intérêt. Comme on veut essayer de restreindre les descentes dans des minimums locaux peu profonds, on essaie de garder une probabilité proche de `pmin` pour les alternatives qui ont très peu de phéromones, et au maximum de `pmax` pour celles qui sont trop représentées (attention, avec le calcul proposé, on ne garanti ni `pmin`, ni `pmax`, mais on n'en est pas loin).

```

def construction(pheromones, alpha, power, pmin, pmax):
    order = list(range(10))
    #random.shuffle(order)
    state = [3]*10 # arbitrary state, just a state that is not a good starting point, so it is
    for pos in order:
        qualities = apriori(state, pos)
        interest = [f(qualities[i-1]+1, power) + alpha *pheromones[pos][i-1] for i in range(1,
        total = sum(interest)
        interest = [max(min(i/total, pmax), pmin) for i in interest]
        total = sum(interest)
        cumulative = []
        p = random.random()
        cumulative = 0
        i = 0
        while p > cumulative:
            cumulative += interest[i] /total
            i+=1
            state[pos] = i
    return state

```

- `antColony` va gérer les vagues successives de fourmis.

- `0 < beta < 1` règle la vitesse de disparition des phéromones, plus ce paramètre grandit, plus les phéromones sont volatiles.
- `alpha`, `power`, `pmin` et `pmax` sont des paramètres pour les appels des fonctions précédentes

Au départ il n'y a pas de phéromones, puis par vagues de 100 fourmis, elles explorent le territoire (construction de leur `state`). Une fois toutes les fourmis passées, mise à jour des phéromones pour la vague suivante.

```

def antColony(alpha = 100, beta = 0.1, power = 2, pmin =0.03, pmax=0.5):
    pheromones = [[0 for _ in range(1,13)] for __ in range(10) ]
    total = 0
    cpt =0
    while True:
        newPheromones = [[0 for _ in range(1,13)] for __ in range(10) ]
        bestE = 10000
        for ant in range(100):
            state = construction(pheromones, alpha, power, pmin, pmax)
            E = energy1(state)
            if E==0:
                print(cpt, "*****", state)
            else:
                if E <bestE:
                    bestE = E
                for i in range(10):
                    newPheromones[i][state[i]-1] += f(E, power)

        pheromones = [[beta * newPheromones[i][j-1]/200+ (1-beta)* pheromones[i][j-1] for j in
        total += bestE
        cpt +=1
        if cpt%10==0:print(int(total/cpt*100)/100,"t", bestE, " ", state)

```

4. Application au voyageur de commerce

L'idée est exactement la même, mais cette fois les formules qui vont assez bien qui permettent de bien gérer les phéromones sont très étudiées et la page wikipedia peut être un bon point de départ documentaire. C'est l'exemple traité dans quasiment la totalité des cours

d'introduction à cette métaheuristique et vous n'aurez normalement aucune difficulté à trouver des sources qui vous conviennent.

3.3.4. Algorithmes génétiques

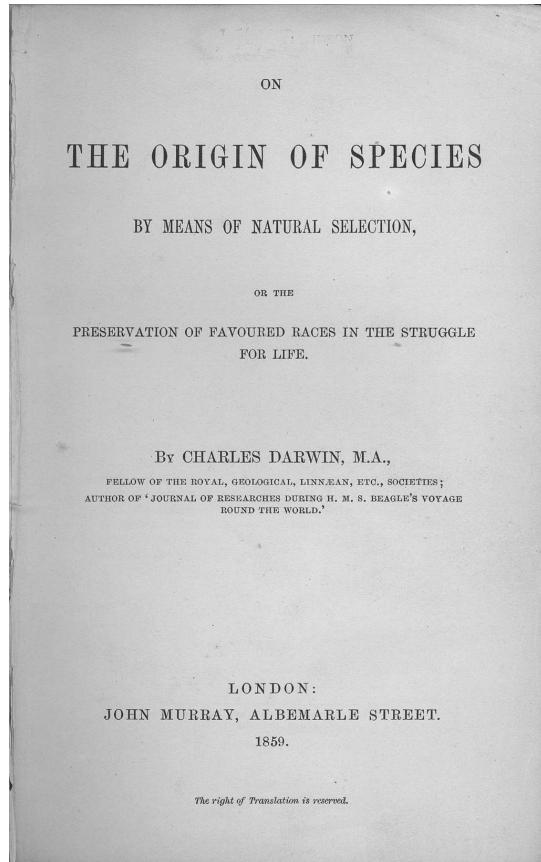


Figure 14 : L'origine des especes (C. Darwin 1859)

En s'inspirant de la théorie Darwinienne, les informaticiens ont adapté la théorie de l'évolution à la recherche d'optimums. L'évolution se fait par sélection naturelle, ce sont les individus les plus adaptés à leur milieu qui survivent le mieux. Les mutations se font au hasard et celles qui apportent un avantage vont augmenter les chances de survie. En augmentant la survie et la qualité de vie, les individus augmentent leurs chances de procréation et donc les caractères adaptés au milieu seront favorisés de générations en générations.

1. Présentation générale

Les algorithmes génétiques sont des métaheuristiques manipulant des groupes de configurations que l'on appelle population. Le principe général est le suivant:

1. Créer la population initiale.
2. Sélection des individus de la population qui deviendront des *parents*.
3. Ces parents ont des enfants qui constitueront la génération suivante.
4. Faire vivre et évoluer ces enfants.
5. Repartir au point 2 avec cette nouvelle génération.

2. Créer la population initiale

On crée un ensemble de situations, soit en les tirant au hasard, soit en essayant de les répartir dans l'espace de recherche afin d'essayer d'avoir des chances que certains d'entre eux soient ne soient pas trop éloignés d'un optimum local de qualité.

3. Sélection des individus qui deviendront des parents

Il existe plusieurs méthodes, celles qui reviennent le plus souvent sont :

- Conservation des individus possédant les meilleures évaluations : va souvent permettre une convergence rapide, au risque d'une diversité amoindrie et donc un risque de ne converger que vers un optimum de faible qualité.
- Probabilité qu'un individu soit choisi comme parent proportionnelle à sa qualité. On réalise autant de tirages que l'on désire de parents, et à chaque tirage, la probabilité qu'un individu soit choisi est le rapport entre sa qualité et la somme des qualités des participants. On peut ou non enlever un individu une fois qu'il a été élu (si on ne l'enlève pas, il pourra alors apparaître plusieurs fois dans la liste des parents)
- Compétition par tournoi : On choisit la taille des équipes N . On choisit au hasard N individus, et on conserve celui possédant la meilleure évaluation parmi les N . Encore une fois, un individu qui a été élu peut ou non être retiré de la liste des participants. On répète le processus jusqu'à avoir choisi le nombre désiré de parents. Cette méthode donne souvent des résultats assez satisfaisants, le paramètre N permettant de régler le niveau d'élitisme que l'on désire (dans le cas extrême où N est la taille de la population, on ne choisit que les meilleurs, avec $N = 1$, c'est un tirage aléatoire équiprobable ; avec N assez grand on choisit comme parents les champions des pays, avec N petit on choisit comme parents les champions de village).
- La sélection par distanciation : on définit une distance sur l'ensemble des configurations, et on va choisir en priorité des individus qui sont les plus éloignés possibles de façon à maximiser la diversité de la population. Cette approche est intéressante mais elle peut rapidement devenir assez coûteuse en temps de calcul. Un algorithme approché permettant d'envisager cette méthode est le suivant :
 - Distribuer de façon aléatoire les individus dans deux ensembles A et B (chaque individu a une chance sur deux d'être affecté aux ensembles A et B).
 - Pour chaque couple $x \in A, y \in B$, calculer la distance de x à y .
 - Décider d'une distance de coupe, et pour chaque distance calculée inférieure à cette distance de coupe, supprimer l'individu de l'ensemble A

Ce processus peut être itéré si on le désire, chaque passe ayant un coût de l'ordre de $O(\text{nb_individus}^2)$. Cette forme de sélection utilisée de façon hybride ou pure devient particulièrement pertinente lorsque le temps d'évaluation de la fonction objectif devient important, et dans son coût devient proportionnellement moins élevé.

- Ces méthodes sont mélangeables et il n'est pas rare de conserver les meilleurs puis de compléter la liste des parents par des tournois.

4. Construire la génération suivante

Cette partie est celle qui est la plus variable. Cependant deux considérations doivent rester à l'esprit :

- la spécialisation : dans les zones intéressante, il est utile de raffiner les solutions afin de les améliorer. C'est en général l'opération la plus fréquente. Les opérations les plus fréquemment rencontrées sont :
 - le crossover à deux parents : s'applique aux configurations qui sont des vecteurs. Un point de rupture est tiré au hasard, et un nouvel individu est créé en prenant la partie

gauche de la configuration du premier parent, et la partie droite du second. Cette transformation est la plus célèbre et a donné des résultats intéressants. Elle est parfois plus compliquée à mettre en œuvre, car les nouveaux individus ne sont pas toujours valides : dans le cas du voyageur de commerce par exemple, avec des configurations qui sont l'ordre de parcours des villes, cette opération va générer des configurations non valides (2 fois certaines villes, absence d'autres villes) et il sera nécessaire de rectifier les configurations obtenues pour qu'elles soient valides. Dans le cas du voyageur de commerce, on peut choisir de conserver une partie de la liste de parcours du premier parent, et de la compléter par les villes qui n'y sont pas, dans l'ordre dans lequel elles apparaissent dans le parent n°2.

- le crossover à 3 parents : cette opération est identique à la précédente, mais cette fois il y aura 2 points de rupture, et à gauche on récupère la partie gauche de la configuration du premier parent, au milieu la partie centrale du deuxième, et à droite la partie droite du troisième. Cette opération est préférable à la précédente sur certains problèmes.
- la diversification : la spécialisation a tendance à converger vers des extrêmes locaux. La diversification va permettre d'explorer des parties plus importantes de l'espace des configurations.
 - les mutations : une partie plus ou moins importante d'un parent est altérée aléatoirement pour fabriquer le nouvel individu. Plus l'altération est importante, plus le taux de mutation (pourcentage de la nouvelle génération obtenue par mutation) doit en général être faible. Il est à noter que la majeure partie des mutations est défavorable, et que les individus générés par mutation posséderont rarement de bonnes évaluations. Parfois on ajoute à ce mécanisme un temps de survie des mutants de générations en générations afin de laisser la chance aux mutations de porter leurs fruits. Dans le cas du voyageur de commerce, on peut choisir $k \geq 2$ villes et les échanger aléatoirement dans le circuit. Plus k est grand, plus la force de la mutation est importante.
 - la mutation peut être portée à l'extrême en choisissant simplement d'intégrer des individus générés comme à l'initialisation. La mise en œuvre est alors très simple, mais plus l'algorithme avance, plus ses nouveaux individus sont défavorisés par rapport à ceux ayant subi plusieurs générations d'évolution et ils ont tendance à être éliminés de la population très rapidement.

5. Évolution des enfants

Cette phase n'est présente que dans certaines implémentations, elle correspond à une hybridation de cette météuristiche avec des recherches locales, on parle alors d'*algorithme mémétique*. On vise à améliorer les solutions par des algorithmes rapides. Il n'est pas nécessaire de pousser cette optimisation au maximum et on se contente parfois de quelques itérations d'une recherche locale, sans la mener à son terme.

Attention, cette amélioration peut être contre-productive, par exemple dans le cas où une mutation d'un attribut vient d'être appliquée pour générer un enfant : une recherche locale a des chances assez importantes de supprimer la mutation, car c'est souvent la suite d'opération la plus simple pour 'corriger' l'individu et améliorer son évaluation. On remplace donc souvent la mutation partielle par l'incorporation de nouveaux individus, qui sont alors passés dans l'algorithme de recherche locale avant d'être incorporés à la population, et après chaque croisement, on applique une recherche locale partielle afin de 'donner une chance' à ce nouvel individu.

4. Règles et jeu

4.1. Proies / prédateurs

Un écosystème de type proie prédateurs est dans sa version la plus simple :

- un monde : par exemple une grille à maille rectangulaire, avec ou sans quelques obstacles, avec sa topologie (rectangulaire ou torique le plus souvent)
- des proies et des prédateurs sont placés sur le monde,
- le système fonctionne à temps discret : à chaque date de temps, les créatures réalisent une action
 - les prédateurs peuvent 'manger' les proies (dès qu'il y a contact, dès qu'au moins deux prédateurs entourent une proie, ...)
 - les proies essaient de survivre en se sauvant au mieux de leurs capacités.

Dans une version à peine étendue, on retrouve le jeu PACMAN, mais on peut rentrer dans ce cadre aussi bien un jeu de raquettes (la balle est la proie, le prédateur la raquette), un shooter, ... que des simulations environnementales ou économiques.

4.2. Système à base de règles

Les systèmes à base de règles sont des outils permettant de résoudre des problèmes d'une façon qui nous semble proche du raisonnement scientifique. Au départ, ils étaient conçus en extrayant de la connaissance des experts (on parlait souvent de système expert). L'informaticien discutait avec un spécialiste du domaine (souvent il y avait même un cogniticien comme intermédiaire), et cherchait à mettre en algorithme tout le raisonnement que suivait le spécialiste afin de recopier son raisonnement.

Dans un système à base de règles, la connaissance est écrite sous forme de *règles*, et un *moteur d'inférence* est chargé de découvrir un chaînage des règles qui permet d'obtenir une réponse à une question posée (« quelle maladie à cette personne ? », « combien de béton est nécessaire pour que le pont tienne ? », « où dois-je investir ? »).

Ces systèmes ont été utilisés dans de nombreux domaines (médical, construction, banque, ...). Ils ont parfois produit des résultats intéressants mais sont sous cette forme passés de mode :

- difficulté de conception : extraire des règles de la tête de quelqu'un... cela n'est pas simple du tout, cela nécessite une formation particulière, beaucoup d'entraînement pour apprendre à poser les bonnes questions, trouver un interlocuteur qui (en l'aidant) parvient à dire ce qu'il fait et non ce qu'il croit faire, gommer toutes les parties 'intuitives' de son raisonnement...
- risque important de ne jamais le voir aboutir : beaucoup de tentatives ont été infructueuses ou non satisfaisantes, l'étape précédente ayant échoué,
- maintenance et extension difficiles : plus le nombre de règles augmente, plus des subtilités apparaissent, des contradictions...
- coût : c'est une conséquence des points précédents.

Les techniques numériques d'apprentissage automatique utilisées maintenant permettent d'extraire de la connaissance, non plus à partir des dire d'un humain, mais à partir de situations observées (d'où l'importance des bases de données). On est passé de liens de causalité (raisonnement) à lien statistique (ce sont les corrélations qui permettent d'inférer : « le sol de mon jardin est mouillé » donc (?????) « personne ne mange de glace sur la plage »). La puissance des

machines permet de détecter des liens statistiques présents dans une base de données (il existe malgré la taille des bases de données un risque de coïncidence statistique, que l'on appelle le sur-apprentissage dans le domaine de l'apprentissage artificiel).

Les techniques les plus productives en termes de résultats possèdent actuellement le problème d'être des boîtes noires, c'est à dire que ce sont des systèmes possédant des millions voire des centaines de milliards de paramètres, et que comprendre a posteriori pourquoi une réponse a été obtenue n'est pas évident. Les réseaux de neurones en particulier, les forêts aléatoires, etc... possèdent ce problème, ils donnent des réponses, mais on ne sait pas extraire quel 'raisonnement' ils ont suivi et ne peuvent donc pas 'justifier' leurs résultats afin qu'un humain le comprenne. Les systèmes à base de règles sont classés dans la catégorie des boîtes transparentes : il est plus facile a posteriori de déterminer quelles règles ont été utilisées, ce qui peut permettre d'extraire le 'raisonnement' qui a été suivi afin d'obtenir un résultat.

4.3. Système à base de règles et projet

Il est possible de choisir des formes de règles qui autorisent plus ou moins de libertés (inspirées de la logique des prédictats, logique des propositions,...). Pour le projet on va appauvrir au maximum la forme des règles afin de parvenir à faire de l'apprentissage automatique. Le système fonctionnera comme suit :

- mesures d'une liste fixée de paramètres sur le monde,
- recherche dans la base de règles des situations les plus proches connues,
- choix d'une de ces règles,
- application de l'action correspondant à la conclusion de la règle choisie.

4.3.1. Observations sur le monde

Une mesure de l'état du monde renvoie un vecteur d'observations de taille fixe. Chacun des champs peut prendre une valeur entière dans un domaine [0..Max_i]. Par exemple, sur un PACMAN, le vecteur de perception d'un fantôme pourrait inclure différentes informations telles que :

1. ce qui se trouve sur la case juste au Nord,
2. dans quel cadran par rapport au fantôme se trouve la proie,
3. à quelle distance la proie trouve-t-elle,
4. dans quel cadran se trouve ...

On peut également ajouter une valeur joker -1 à certains attributs dont l'interprétation serait 'inconnue'.

Exemple

Le vecteur d'observation contient 4 éléments, décrivant respectivement le contenu des cases devant, gauche, derrière, droite. Chacune de ces cases peut prendre comme valeurs :

- 0 : mur,
- 1 : vide,
- 2 : trésor.

Ainsi [0, 1, 0, 2] signifie qu'il y a un mur devant, à gauche une case vide, un mur derrière, à droite un trésor.

L'utilisation d'un type enuméré peut rendre la relecture plus agréable. On peut alors définir le nombre nécessaires de types énumérés pour représenter les différents types d'attributs.

```

typedef enum {
    JOKER=-1, MUR, VIDE, TRESOR
} case_t;

typedef enum{
    N,O,S,E
} direction4_t;

typedef enum{
    N, NO, O, SO, S, SE, E, NE
} direction8_t;

```

Une observation sera alors par exemple :

```

typedef struct {
    case_t devant, gauche, derrière, droite; // le contenu des 4 cases voisines
    direction4_t direction_predateur;          // dans quelle direction se trouve le prédateur le plus
    direction8_t direction_proie;              // dans quelle direction se trouve la proie la plus pro
} RuleTyped;

```

4.3.2. les actions

Les actions sont dans une liste prédéfinie, et sont encodées par un entier. Par exemple, l'action 1 signifie 'aller à droite, la '2' signifie creuser un trou, ...

Exemple

Les actions possibles sont :

- 1 : tourner à gauche puis avancer,
- 2 : tourner à droite puis avancer,
- 3 : se retourner puis avancer.

L'utilisation d'un type enuméré peut rendre la relecture plus agréable.

```

typedef enum {
    AV, GAUCHE, DROITE, RETOUR
} direction_relative4_t ;

```

4.3.3. Les règles

les règles sont constituées de trois parties :

- priorité de la règle : à quel point cette règle est à privilégier par rapport à d'autres règles. On peut choisir que les priorités varient de 0 à 5 par exemple,
- prémissse de la règle : la condition d'application de la règle, c'est un vecteur totalement similaire à celui des observations, mais où pour chaque position on a rajouté une nouvelle valeur possible : -1. Pour qu'une règle puisse s'appliquer, il faut que la valeur de l'observation soit la même que celle obtenue dans l'observation ou que la valeur correspondante dans la règle soit -1 (le -1 à un rôle de joker).

Si on a utilisé le -1 dans la partie observation, il joue également le rôle de joker, et 'matche' avec n'importe quelle valeur d'attribut dans une règle.

- conclusion de la règle : quelle action doit être réalisée dans cette situation.
- Une façon efficace pour encoder les règles est de les placer dans un vecteur mettant bout à bout les observations, l'action et la priorité.

Il est également possible de placer les attributs dans une structure. Cela rend certaines parties du code un peu plus lisibles, mais il vous faut alors bien réfléchir à l'alignement des données en mémoire afin que lorsqu'il y a des copies partielles de règles, ce soient bien des copies par blocs qui soient utilisées (et pas une copie par attribut). Ce point ne sera pas poursuivi par la suite, mais vous pouvez l'aborder avec les enseignants si c'est une approche qui vous convient mieux. Si vous ne manipulez que des variables de même type, cette approche ne devrait pas augmenter notablement la difficulté.

- La base de règles est la suivante (`prémissse --> conclusion (priorité)`):

```

- [-1,  2, -1, -1] --> 1 (5)
- [-1,  1, -1, -1] --> 1 (4)
- [ 2,  0, -1, -1] --> 0 (5)
- [ 1,  0, -1, -1] --> 0 (4)
- [ 0,  0,  2, -1] --> 3 (5)
- [ 0,  0,  1, -1] --> 3 (5)
- [ 0,  0,  0,  2] --> 2 (5)
- [ 0,  0, -0,  1] --> 2 (5)
- [-1, -1, -1, -1] --> 0 (0)
- [-1, -1, -1, -1] --> 1 (0)
- [-1, -1, -1, -1] --> 2 (0)
- [-1, -1, -1, -1] --> 3 (0)

```

Et on peut par exemple encoder la première règle par $[-1, 2, -1, -1, 1, 5]$ ou $[5, 1, -1, 2, -1, -1]$.

Si on a choisi la version `struct` peut alors ressembler à :

```

typedef struct {
    case_t devant, gauche, derrière, droite; // le contenu des 4 cases voisines
    direction4_t direction_predateur;        // dans quelle direction se trouve le prédateur le plus proche
    direction8_t direction_proie;            // dans quelle direction se trouve la proie la plus proche
    int priorite;
    direction_relative4_t action;           // quelle action doit être prise
} RuleTyped;

```

4.3.4. Filtrage des règles

Le filtrage consiste à passer toutes les règles en revue et à ne conserver que celles qui s'appliquent à la situation actuelle. Il s'opère après avoir reçu l'état du monde.

Exemple

Si la situation actuelle est $[0,1,0,2]$ alors le résultat du filtrage est :

```

- [-1,  1, -1, -1] --> 1 (4)
- [-1, -1, -1, -1] --> 0 (0)
- [-1, -1, -1, -1] --> 1 (0)
- [-1, -1, -1, -1] --> 2 (0)
- [-1, -1, -1, -1] --> 3 (0)

```

4.3.5. Choix de l'action à réaliser

Un fonction de différenciation est définie, par exemple la fonction $\mathbb{A} : (x, n) \mapsto (x + 1)^n$, que l'on applique à toutes les priorités des règles. On fixe la valeur de n . La probabilité de choisir la règle i est alors

$$\frac{\mathbb{A}_n(\text{priorité}_i)}{\sum_k \mathbb{A}_n(\text{priorité}_k)}$$

Le rôle de la fonction \mathbb{A} est de régler l'importance de la priorité. Plus n est grand, plus la distribution de probabilité favorise les règles de haute priorité, et au contraire plus n se rapproche de 0, plus la distribution de probabilité se rapproche d'une distribution uniforme.

Exemple

- Avec $n = 2$, la première règle à une probabilité de $\frac{5^2}{5^2+1^2+1^2+1^2+1^2} = \frac{25}{29}$ et les autres actions une probabilité de réalisation valant $\frac{1}{29}$. Dans cette situation, en moyenne 25 fois sur 29, ce sera la première action qui sera choisie.
- Avec $n = 0$, la première règle à une probabilité de $\frac{5^0}{5^0+1^0+1^0+1^0+1^0} = \frac{1}{5}$ et les autres actions une probabilité de réalisation valant $\frac{1}{5}$. Dans cette situation, en moyenne 1 fois sur 5, ce sera la première action qui sera choisie.

4.3.6. Réalisation de l'action

Une fois l'action choisie, il y a une tentative d'exécution de l'action. Il est possible que l'action soit impossible.

4.4. Apprentissage des règles

Pour découvrir les règles, on va utiliser des techniques d'optimisation. Pour cela il faut tout d'abord décider ce qui doit être optimisé :

- on veut maximiser les bénéfices,
- on veut maximiser la quantité de terrain bleu,
- on veut maximiser le temps de survie,
- on veut maximiser le nombre de proies attrapées,
- on veut maximiser la quantité de ressources collectées,
- on veut minimiser les coûts,
- on veut minimiser les efforts,
- ...

Le but est alors de trouver un ensemble de règles qui permet d'optimiser au mieux le critère choisi.

ATTENTION

Une configuration (un point de l'espace E vu dans la partie 3) est l'**ensemble des règles**, la taille de l'espace de recherche est de taille rapidement assez importante. Par exemple, avec

- 8 attributs,
- chacun pouvant prendre 3 valeurs,
- 4 actions possibles,
- 5 niveaux de priorité,
- 10 règles,

la taille de l'espace de recherche E est $(3^8 \times 4 \times 5)^{10} = 1513562413202902254756692072513364967434240000000000 > 10^{50}$

De plus, on ne connaît pas d'expression analytique de la fonction d'évaluation, la seule façon dont nous disposons pour évaluer une situation est de réaliser une simulation (faire jouer le jeu). Cette évaluation sera longue à obtenir, et si on la désire fiable, il sera certainement nécessaire de relancer plusieurs fois la simulation... ce qui va la rendre encore plus longue.

5. Parallélisation

Ce projet est une occasion très pertinente de découvrir les *threads* : les méthodes évolutionnaires (un individu == 1 thread), les colonies de fourmis (une fourmi == 1 thread) par exemple s'adaptent particulièrement bien au *multi-threading*.

Les fils d'exécution permettent de faire tourner en parallèle (non simulé sur les ordinateurs actuels qui disposent des capacités pour exécuter un certain nombre de tâche au même instant) différentes fonctions. Pour que cette fonctionnalité des processeurs soit assez facilement exploitable, pour tout ce que vous désirez paralléliser, vous devez penser dès la conception à :

- ne pas avoir d'effet de bord,
- ne pas avoir de synchronisation nécessaire (une fois lancée, une thread n'interagit pas avec les autres threads jusqu'à ce qu'elle ait rendu sa réponse...).

5.1. Présentation



Figure 15 : Des milliers d'exécutions en parallèle...

Dans la majorité des programmes que vous avez développé jusqu'ici, l'exécution était linéaire. C'est à dire que chaque instruction était exécutée après la précédente dans l'ordre dans lequel elles étaient écrites dans le code source.

Il est possible et de plus en plus courant de développer les programmes pour qu'ils utilisent plusieurs fils d'exécutions. Le système d'exploitation sera alors en charge de choisir quel fil exécuter. Une machine de bureautique moderne possède plusieurs processeurs (une dizaine) et chacun peut exécuter un fil d'instruction. Un serveur de calcul comme ceux du campus possède une centaine de processeurs.

Il est donc possible (voire recommandé si on veut profiter de la puissance des machines) d'exécuter plusieurs fils d'instructions simultanément. En général, il est très préférable que les fils d'exécution soient indépendants. Il y a deux types d'indépendance :

- indépendance vis à vis des ressources : imaginez une variable globale qui est accédée par plusieurs programmes, il va être difficile de faire une conception sans erreur !,
- indépendance des fils d'exécution entre eux : il n'y a pas besoin d'attendre qu'une thread ait finie son travail pour qu'une autre puisse continuer le sien (il existe des outils de synchronisation permettant de mettre en place ce type de comportement mais nous n'approfondirons pas cet aspect ici).

5.2. Mise en garde

On ne fait ici qu'une introduction au mécanisme de parallélisation. Ainsi nous passons sous silence une majorité d'informations. **Si vous travaillez sur les machines du campus, ne dépassez pas la dizaine de fils d'exécution. Il faut laisser de la place aux autres.**

5.3. Utilisation théorique

A partir de maintenant nous appellerons *thread* un fil d'exécution.

Un programme peut créer de nombreux threads. Il est cependant responsable de s'assurer qu'ils soient tous arrêtés. Pour ce faire, on attendra la fin de chaque thread créé.

L'exemple suivant crée deux threads effectuant chacun un traitement. Un autre traitement est fait parallèlement sur le thread principal. Ainsi trois traitements sont fais en simultané. Le thread principal (avec le traitement 0) est responsable de s'assurer que les deux autres se terminent et attend donc la fin des threads 1 et 2.

En pseudo code

```

Programme principal:
    Creation d'un premier fil.
        - traitement effectué dans le thread 1
    Creation d'un second fil.
        - traitement effectué dans le thread 2

    Traitement effectué dans le thread 0

    Attente de la fin des deux fils.

    Traitement utilisant les résultats de 0, 1 et 2.

```

La durée d'exécution de ce programme est la somme du temps d'exécution du traitement le plus long parmi 0, 1 et 2 et du traitement final qui utilise les résultats précédents. En réalité, il y a aussi du temps perdu pour la gestion des threads mais il devrait être faible par rapport aux traitements (il peut devenir non anecdotique lorsqu'on crée énormément de threads, chacun ne faisant que peu de travail).

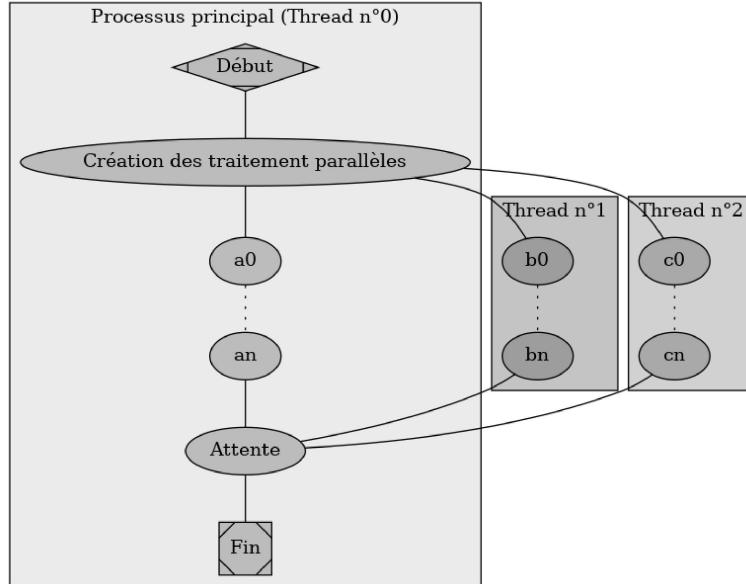


Figure 16 : Un thread principal et deux threads fils

5.4. Implémentation en C

Les outils dont nous avons besoin en **C** sont disponibles dans le header `threads.h`, vous trouverez la documentation ici <https://en.cppreference.com/w/c/thread>. La création d'un fil d'exécution doit systématiquement impliquer l'attente de la fin de son traitement. Ainsi on pourra créer une multitude de fils à la suite. Puis on attendra que tous se finissent.

1. On commence par créer la variable qui stockera l'information sur le thread, elle est de type `thrd_t`.
2. Ensuite on crée le thread avec `thrd_create`. Cette fonction prend en arguments :
 - l'adresse de la thread que l'on crée,
 - l'adresse d'une fonction à exécuter, dont la signature est `void* --> int` :
 - la valeur de retour est prévue pour que si nécessaire le traitement renvoie un code d'erreur
 - l'adresse (non typée) permet le passage d'un paramètre de type quelconque, mais il ne faut pas oublier dans le code de la fonction de 're-typer' le paramètre avant de l'utiliser.
 - l'adresse du paramètre lors de l'exécution de la fonction précédente.
3. Enfin on attendra la fin de l'exécution du thread avec `thrd_join`. C'est grâce à cette fonction que l'on récupère la valeur de retour des fonctions exécutées dans les threads.

```

int treatment(void * parameters) {
    int *p = (int*) parameters; // p is now the address of an int, and not the address of a "??"
    printf("Compute with value: %d.\n", *p);
    return 0;
}

int treatment2(void* parameters){
    float p = *((float*) parameters); // the value starting at the address 'parameters' is interpreted
    p = p * p - 1;
    // ...
    printf("I have computed with this value: %f.\n", p);
}

int main() {
    thrd_t thread_handle_a;
    int a = 0;
    thrd_create(&thread_handle_a, treatment, &a);

    thrd_t thread_handle_b;
    float b = 2;
    thrd_create(&thread_handle_b, treatment2, &b);

    int error_code_of_thread_a = 0;
    int error_code_of_thread_b = 0;
    thrd_join(thread_handle_a, &error_code_of_thread_a);
    thrd_join(thread_handle_b, &error_code_of_thread_b);
    return 0;
}

```

En exécutant plusieurs fois votre programme vous apercevrez quelques fois que les lignes ne sont pas affichées dans le même ordre. La commande shell `watch` permet d'exécuter en boucle (jusqu'à un `CTRL-c`) une commande ou un fichier. Par exemple la commande suivante va exécuter toutes le 0.1 secondes le programme « `./executable` ».

```
watch -n 0.1 ./executable
```

6. Compléments algorithmiques

6.1. Tirer une v.a. selon une loi tabulée

Lorsque la distribution de probabilité d'une variable aléatoire est connue sous la forme d'un tableau, on peut générer des réalisations de cette variable par l'algorithme qui suit.

```

Entrée :
- T un tableau de n cases, dont la case i contient P(X=i)
Sortie :
- un entier i qui est une réalisation de X
Processus :
- Définir alpha : un réel tiré en aléatoire uniforme dans [0 ; 1[
- Définir s = n-1                                // par défaut, le programme renvoie la dernière réalisation
- Définir cumul = 0                               // on construit les probabilités cumulées

- Pour i = 0 à n-2 :
  - cumul += T[i]
  - Si alpha < cumul :
    - s = i

```

- break
- renvoyer s

6.2. Mélanger les éléments d'un tableau

L'algorithme classique afin de mélanger les éléments d'un tableau est l'algorithme de Fisher-Yates. Il est simple à implémenter et efficace. Sa complexité est $N \times T$ où N est la taille du tableau et T la complexité de la génération d'un nombre aléatoire de la taille d'une case du tableau.

```

Entrée :
- T un tableau de taille N
Sortie :
- T a été modifié, il contient une permutation de ses éléments, tirée en aléatoire uniforme
Process :
- Pour i = n-1 à 1 par pas de -1
- Définir j : un entier tiré en aléatoire uniforme dans [0 .. i]
- échanger le contenu des cases i et j de T
  
```

6.3. Bibliographie

Pour ceux qui ont besoin de précision sur une structure de données, ou un algorithme de parcours, recherche, ... le premier endroit où chercher :

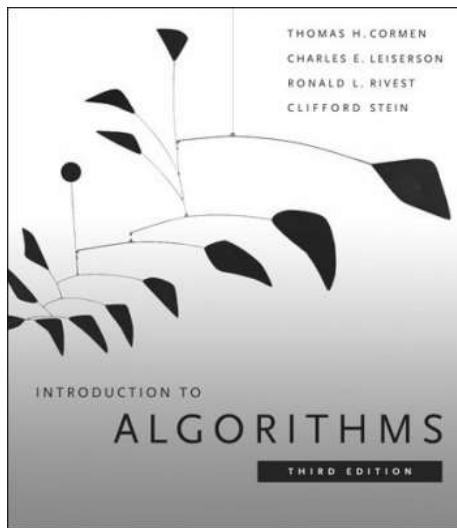


Figure 17 : Avant d'aller chercher ailleurs !

7. Généralités sur le code

- Ce n'est pas après 3 jours que l'on écrit son premier makefile, c'est le plus vite possible :
 - developpez.net
 - et pour ceux qui ont beaucoup de difficultés avec les makefiles, voici une base possible :

```

SRC=q_learning.c parameters.c general.c
SRC+=world.c etc1.c etc2.c
#SRC=$(wildcard *.c) # en commentaire, je ne suis pas un grand amateur
EXE=q_learning.out

CC=gcc
CFLAGS:=-Wall -Wextra -MMD -Og -g $(sdl2-config --cflags)
#CFLAGS:=-Wall -Wextra -MMD -O2 $(sdl2-config --cflags) # pour la version release
LDFLAGS:=-lSDL2_image -lSDL2_ttf -lSDL2_gfx -lm -lSDL2

OBJ=$(addprefix build/, $(SRC:.c=.o))
DEP=$(addprefix build/, $(SRC:.c=.d))

all: $(OBJ)
    $(CC) -o $(EXE) $^ $(LDFLAGS)

build/%.o: %.c
    @mkdir -p build
    $(CC) $(CFLAGS) -o $@ -c $<

clean:
    rm -rf build core *.o

-include $(DEP)

```

7.1. 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.

8. 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'un 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.

8.1. 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 Seward et al.
==3719130== Using Valgrind-3.16.1 and LibVEX; rerun with -h for copyright info
==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 allocated
==3719130==
==3719130== All heap blocks were freed -- no leaks are possible
==3719130==
==3719130== For lists of detected and suppressed errors, rerun with: -s
==3719130== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

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 allocated
==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 leaked memory

```

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

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

8.2. 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 value(s)
==3729129==   at 0x109144: foo (in /tmp/tmp.1622971724.300q/a.out)
==3729129==   by 0x109174: main (in /tmp/tmp.1622971724.300q/a.out)

```

Avec l'option `-g` :

```

==3729359== Conditional jump or move depends on uninitialised value(s)
==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).

8.3. 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.out)
==3845342==    Address 0x4a3a054 is 0 bytes after a block of size 20 alloc'd
==3845342==    at 0x483877F: malloc (vg_replace_malloc.c:307)
==3845342==    by 0x109164: main (in /tmp/tmp.1622971724.300q/a.out)
```

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 20 alloc'd
==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`.

8.4. 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 peut 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 --gen-suppressions=all --log-file <log>
```

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=./sdl.sup <executable>
```

Ces explications sont extraites d'un wiki.

8.5. 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.

8.5.1. 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.

8.5.2. 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.

8.5.3. 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.

9. Comparer les temps d'exécution

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

9.0.1. 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<1000000000; i++ ) {

    }

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

    return 0;
}
```

9.0.2. Méthodes évoluées : les profileurs

Les profileurs permettent de mesurer le temps passé dans chaque fonction du programme. Ils ont en général pour vocation de 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 le 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).