D A E M O N L A B

INTELLIGENCE ARTIFICIELLE

- DaemonLab -

Bienvenue.

Vous allez apprendre à réaliser à la main des objets d'une science qu'on appelle relativement abusivement « intelligence artificielle ».

Nous allons aborder : la machine a état, l'algorithmie génétique et les réseaux de neurones.

01 – Détails administratifs

Certains exercices vous demandent d'utiliser le C **et** le C++. Dans ces cas là, les règles sont les suivantes :

Chaque fonction est demandée en double. Vos interfaces C doivent également être utilisable en C++. Les versions C++ sont donc présente pour une question de confort. Usez de la constante de précompilation **__cplusplus** et de la directive **extern "C".** *Votre fichier en-tête doit être compatible avec les deux langages.* Bien entendu, votre implémentation dans un langage **doit** appeler votre implémentation dans l'autre langage, **il ne doit pas y avoir de doublon de code !** Idéalement, implémentez les deux fonctions dans le même fichier.

Vous compilerez avec g++ ou clang++.

Votre rendu doit respecter **strictement** l'ensemble des règles suivantes :

- Il ne doit contenir **aucun** fichier objet. (*.o)
- Il ne doit contenir **aucun** ichier tampon. (*~, #*#)
- Il ne doit pas contenir votre production finale (programme ou bibliothèque)

Le respect de la table des normes est obligatoire.

Vous devez écrire des tests unitaires et exploiter lcov/gcov pour disposer à chaque étape d'une couverture de code avoisinant les 100 %. Un patron de projet avec couverture de code et programmes de tests basés sur assert vous est fourni en annexe.

02 – Machine à état

Fichier à rendre : state/include/*.h* state/src/*.c*

Une machine a état est un système dont l'action dépend de son **état**, et dont **toutes** les possibilités d'états sont couvertes, et **d'éléments extérieurs** également **exhaustivement** représentés.

Une machine à état, par exemple, pourrait consister en un robot pouvant aller dans quatre direction en fonction de la valeur d'un entier ne pouvant valoir qu'entre 0 et 3. Une information extérieure serait l'impact avec un mur. En cas de non impact, l'état serait conservé. L'impact avec un mur provoquerait l'augmentation de cet entier et un dépassement de la valeur maximale un retour à 0.

Si cela vous semble peu clair, nous allons passer à l'implémentation.

Vous allez donner vie à une **petite boule d'ASCII**. Votre programme affichera un carré de caractères de 30 cases sur 30 cases, chaque case étant modélisé par une **paire** de caractères se suivant. La petite boule d'ASCII sera **()** tandis que les murs occupant les bordures de l'espace seront **XX**. Les espaces autres seront constitués… d'espaces. Des murs peuvent être aléatoirement placés également.

Sur le terrain de jeu, il y aura également à manger, une unique feuille de salade : >< et à boire : \/.

La petite boule d'ASCII dispose d'un état **direction** indiquant si elle va 0 : en haut, 1 : à droite, 2 : en bas, 3 : à gauche. Elle dispose également d'une **faim** et d'une **soif**. Si elle a **faim**, elle n'a pas **soif**, si elle a **soif**, elle n'a pas **faim**.

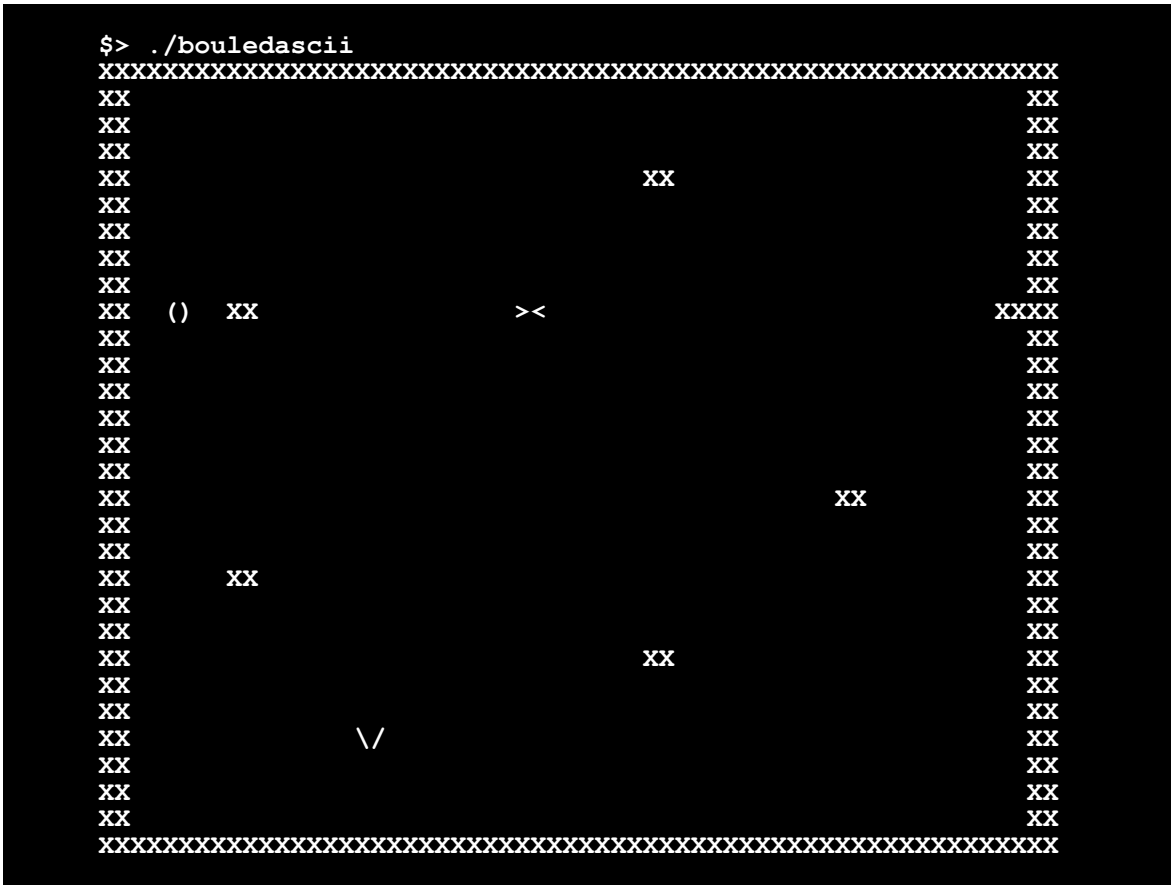
Elle sait si il y a oui ou non de la nourriture en vue (sur la même ligne ou colonne qu'elle, sans obstacle entre elle à la nourriture)

Elle sait si il y a oui ou non à boire en vue (sur la même ligne ou colonne qu'elle, sans obstacle entre elle et sa boisson)

Elle sait si elle va taper ou non dans un mur.

La petite boucle d'ASCII au départ par dans une direction aléatoire et elle a faim. Sa position de départ est aléatoire mais légale (en dehors d'un mur).

Sur la page suivante, vous trouverez un exemple de représentation.



Il y a plusieurs manières de réaliser une machine a état. Vous pouvez simplement définir une structure ou une classe et utiliser des conditions, faisant de votre programme une machine à état… ou bien programmer une machine à état dans votre programme en rendant explicite les différentes combinaisons.

Pour cet exercice, vous réaliserez la deuxième méthode : votre programme contiendra une table de transition d'état.

https://fr.wikipedia.org/wiki/Table_de_transition_d%27%C3%A9tat

Combiné avec des pointeurs sur fonction, le résultat devrait être intéressant.

03 – Algorithmie génétique
« N'attrape pas froid »

Fichier à rendre : dont_get_cold.h *.cpp

Toutes les fonctions et structures de cet exercice doivent être accessible en C. Vous êtes libre d'employer C++ tant que cela se passe en arrière-boutique. **Vous aurez besoin de connaître un peu la LibLapin.**

Un algorithme génétique est déterminé par un fonctionnement particulier. Sa construction est la suivante : l'algorithme manipule des individus (des structures) déterminés par des gênes (des attributs).

L'algorithme dispose d'une condition à la reproduction des individus (« **l'heuristique** »).

Il fonctionne ainsi : au départ sont générés des individus dont les gênes sont **aléatoires**. Par la suite, l'opération suivante est répétée, on dit qu'il s'agit la d'une **génération** :

Tous les individus sont testés pour vérifier si ils remplissent la condition et à quel point ils la remplissent. Ceux qui échouent complètement sont éliminés ou simplement interdit de reproduction. La condition ici est la distance avec un point chaud. Le double de retour est cette satisfaction.

Les autres peuvent se reproduire : c'est à dire qu'un nouvel individu est crée à partir de leurs attributs, sélectionnés de manière encore **aléatoire**.

Les individus qui se sont reproduis peuvent-être ou non retiré de la population au bout d'un certain nombre de génération ou d'une seule.

En plus des enfants, vous pouvez créer de nouveaux individus générés aléatoirement où introduire des **mutations**, changeant aléatoirement certains gênes de certains individus.

En partant de ce principe, vous allez réaliser un algorithme génétique qui entraîne les individus de sa population à se regrouper autour de certains points dit « chaud ». Réalisez les fonctions suivantes. **t_people** est un individu. Les **spots** des points chauds.

```
// space est allouée mais non initialisé.
t_people    *dgc_new_people(t_people                *space);
t_people    *dgc_new_people(t_people                *space,
                            const t_people          *a,
                            const t_people          *b);
double      dgc_test_people(const t_people          *people,
                            const t_bunny_position  *spots,
                            size_t                  nbr_spots);
void        dgc_display(t_bunny_pixelarray          *px,
                        const t_people              *people,
                        size_t                      nbr_people,
                        const t_bunny_position      *spots,
                        size_t                      nbr_spots);
```

Votre programme effectuera une génération par pression sur la touche espace et affichera le résultat à l'écran, point blanc pour les individus, rouge pour les points chauds. Vous devriez voir des points se rapprocher au fur et à mesure des points chauds.

04 – Algorithmie génétique
« Rejoins le sommet »

Fichier à rendre : dgenetic.h *.cpp

Chargez l'image reçu avec l'exercice. Cette image contient des dégradés. Plus une couleur est claire, plus sa « hauteur » est grande.

Programmez un algorithme génétique permettant de trouver, le plus possible, les points les plus élevés de l'image. Attention : vous n'avez pas le droit de parcourir explicitement l'image ! Vous ne pouvez vous baser que sur l'état des individus !

Pour l'exercice courant, vous pouvez limiter la portée de la reproduction : seulement les individus relativement proche peuvent générer de nouveaux individus.

```
t_people    *peak_new_people(t_people                *space);
t_people    *peak_new_people(t_people                *space,
                             const t_people          *a,
                             const t_people          *b);
double      peak_test_people(const t_people          *people,
                             const t_map             *map);
void        peak_display(t_bunny_pixelarray          *px,
                         const t_people              *people,
                         size_t                      nbr_people,
                         const t_map                 *map);
```

Tout comme pour l'exercice précédent, réalisez un programme qui avance d'une génération par pression sur la touche espace. Affichez chaque étape. N'oubliez pas d'afficher l'image. Le rouge représente des pics bas, les bleus des intermédiaires, les verts de hauts. Le plus haut pic étant le bloc blanc.

05 – Algorithmie génétique « Suis-le »

Fichier à rendre : dgenetic.h *.cpp

Reprenez votre code pour « N'attrape pas froid ».

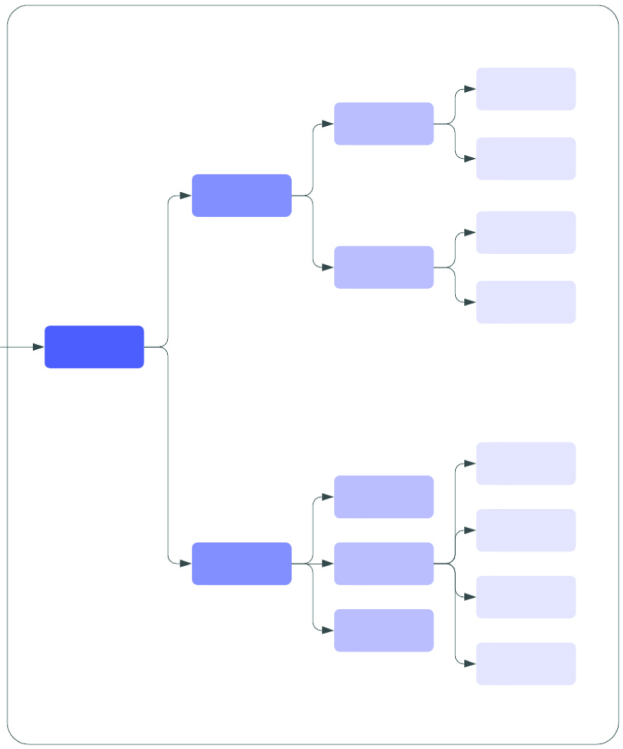
Changez les préfixes **dgc** par **flw** et au lieu de générer des points chauds fixes, vous allez en générer un seul. Ce point cependant sera fort différent des points de l'exercice 3, car il effectuera une rotation régulière autour du centre de l'écran. Utilisez **cos** et **sin** afin de faire tourner celui-ci régulièrement sur un cercle d'un rayon 33 % de la fenêtre.

Observez la manière dont les individus réagissent au déplacement de ce point.

06 – Réseaux de neurones

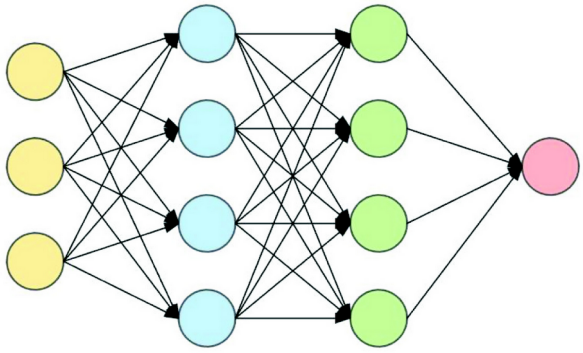
Imaginons que vous souhaitiez créer à l'aide d'un algorithme génétique un pilote de voiture. Vous pouvez entraîner votre voiture sur un circuit donné, ses gènes déterminant la vitesse et l'orientation à suivre ce circuit. Au bout de plusieurs générations, vous auriez une voiture parfaitement à l'aise sur ce circuit… mais là, vous voudriez évidemment tester cet individu « parfait » sur un autre circuit ! Sauf que celui-ci n'a été entraîné que pour être efficace sur un seul et donc votre stratégie de considérer la direction sur le circuit comme les gènes de la voiture aurait été un échec. *Finalement le résultat de votre travail ne serait pas un super pilote… mais le meilleur parcours possible sur le circuit ! Intéressant, mais insuffisant !*

La solution ne serait donc pas d'entraîner un individu à conduire sur un circuit donné mais d'entraîner un individu à… conduire tout court. Petit problème : vous devriez donc disposer d'un programme ! Les gênes étant les instructions de celui-ci. Savez-vous comment on représente les programmes graphiquement ? Sous la forme d'arbre de décision.



Serait il possible de construire en mémoire un arbre de décision ? Chaque nœud de cet arbre serait un gène et sa décision varierait d'un individu à l'autre. La réponse est **oui**.

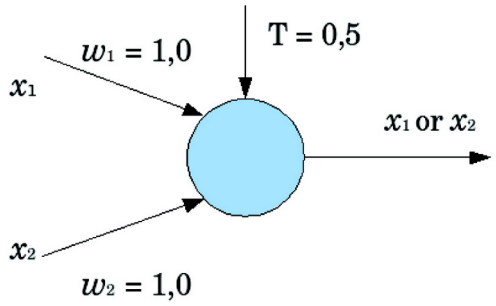
Cet arbre de décision prendra une forme particulière : celle d'un système avec ses entrés, ses sorties, et ses états intermédiaires. Chaque état étant potentiellement amené à considérer chaque entrée, chaque sortie étant potentiellement amené à considérer chaque état… Ci-dessous, un **réseau de neurones formels** :



A gauche, les entrées, à droite, les sorties. Dans cet exemple, trois entrées et une sortie ainsi que deux strates intermédiaires. Vous pouvez remarquer que chaque strate considère chaque strate précédente : tous les neurones d'une strate sont reliés à tous les neurones des strates précédentes.

07 – Création d'un neurone formel

Fichier à rendre : dneuron.h *.cpp



Un neurone formel dispose de N entrées, associées à N coefficients un à un. **Les valeurs de ces coefficients sont des gènes.** La valeur interne du neurone est la somme de tous les coefficients multipliés par les valeurs de toutes les entrées. La valeur de sortie du neurone dépend d'une valeur de basculement **qui est également un gène.** Cette sortie est booléenne. Il est donc possible de représenter un neurone ainsi :

```
struct s_neuron;

typedef struct          s_neuron_input
{
      struct s_neuron   *neuron;
      double            coefficient;                    // Gène
}                       t_neuron_input;

typedef struct          s_neuron
{
      double            toggle_value;                   // Gène
      t_neuron_input    *inputs;
      size_t            nbr_input;
      double            inner_value;
      bool              output_value;
      bool              ready;
}                       t_neuron;

t_neuron                *new_neuron(t_neuron            *space,
                                    size_t              nbr_input);
bool                    compute_neuron(t_neuron         *neuron);
```

A vous d'écrire les deux fonctions indiquées ici. Il doit être possible de préciser 0 entrées, afin de définir plus tard les entrées du réseau de neurones.

08 – Création d'un réseau de neurones

Fichier à rendre : dneuron.h *.cpp

Vous allez maintenant réaliser un réseau de neurones en entier. Pour commencer, vous allez réaliser une unique strate. Réalisez la fonction suivante :

```
typedef struct          s_neuron_array
{
      t_neuron          *neurons;
      size_t            nbr_neuron;
}                       t_neuron_array;

t_neuron_array          *new_neuron_array(size_t      nbr_neuron,
                                          size_t      nbr_input);
void                    neuron_link(t_neuron_array    *right_array,
                                    t_neuron_array    *left_array);
```

La fonction **new_neuron_array** réalisera d'un coup une strate du réseau de neurone. **nbr_neuron** indiquant le nombre de neurones de la strate et **nbr_inputs** le nombre de neurone de la strate précédente. La fonction **neuron_link** réalisera les connexions entre **right_array** et **left_array**. Ce sont les neurones de **right_array** qui doivent pointer sur **left_array**.

Réalisez ensuite la fonction suivante :

```
typedef struct          s_neuron_network
{
      t_neuron_array    *neuron_array;
      size_t            nbr_neuron_array;
}                       t_neuron_network;

t_neuron_network        *new_neuron_network(size_t    *array_width,
                                            size_t    nbr_array);
```

Cette fonction construit un réseau de neurones entier constitué de **nbr_array** strates (incluant la strate d'entrée et la strate de sortie). **nbr_array** ne peut pas être inférieur à 2. Elle renvoi un tableau de **t_neuron_array** de **nbr_array** cases. Les liens entre strates doivent être établis dans cette fonction. Ci-dessous, un exemple d'utilisation permettant de réaliser le réseau de neurone servant d'illustration dans la partie 2.

```
size_t width[4] = {2, 4, 4, 1};
t_neuron_network *network = new_neuron_network(width, 4);
... // Entrainement
network->neuron_array[0].neurons[0].output_value = true;
network->neuron_array[0].neurons[0].ready = true;
network->neuron_array[0].neurons[1].output_value = false;
network->neuron_array[0].neurons[1].ready = true;

compute_neuron(network->neuron_array[3].neurons[0]);
```

09 – Et maintenant ?

Vous avez réalisé un réseau de neurone mais celui-ci est incapable de réaliser la moindre tache ! Ses coefficients ne sont pas configurés pour qu'il puisse faire quelque chose : selon le principe de l'algorithmie génétique, ses coefficients sont aléatoires…

Vous aurez donc compris que cette énorme structure que vous avez créée… n'est qu'un unique individu dans votre algorithme génétique. L'heuristique de votre algorithmie génétique sera de constater une correspondance entre une entrée envoyée à votre réseau de neurone et un résultat qu'il fournira. Votre génération de nouvel individu… mélangera aléatoirement les coefficients entre deux réseaux de neurones.

Est-ce tout ? Non, bien sur. Vous connaissez le mécanisme d'entraînement (l'algorithme génétique) et le mécanisme de décision (le réseau de neurones) mais comment s'en servir ? Vous pouvez vous en servir en commençant par définir votre besoin en entré et en sortie, en préparant un set d'entraînement (les tests de votre heuristique)

Commencez simplement, réaliser les systèmes suivants :

- Une porte **OR** : deux entrées, une sortie. La sortie est vraie si au moins une entrée est vraie.
- Une porte **ET** : deux entrées, une sortie. La sortie est vraie si les deux entrées sont vraies.
- Une porte **XOR** : deux entrées, une sortie. La sortie est vraie si les deux entrées sont différentes.

Le nombre de strates intermédiaires est laissé à votre discrétion, ainsi que leur largeur.

Une fois ces portes réussies, cassez-vous les dents sur la porte NON : la sortie est vraie si l'entrée est fausse, et vous devriez remarquer un problème…

Comment à partir d'une entrée valant 0, d'additions et de multiplications de valeurs strictement positives, trouver une valeur positive en sortie ? Cela semble bien impossible.

Il existe plusieurs solutions, voici deux propositions : Vous pouvez ajouter des entrées, par exemple, disposer des **inverses** des entrées, ou alors changer de modèle : passer de **bool** à **int**, pour disposer trois états : -1, 0 et 1. Vos coefficients devront pouvoir valoir des valeurs négatives également de ce fait, de même que la valeur de bascule de vos neurones.

10 – Sérialisation

Vous avez réussi à entraîner votre réseau de neurone pour qu'il parvienne à avoir le comportement qui vous intéresse ! Bien, mais comment l'exploiter ? L'objectif n'était pas de programmer un entraînement mais de préparer un réseau pour s'en servir ailleurs...

Vous devez pouvoir l'enregistrer pour pouvoir le recharger ensuite sans avoir à l'entraîner à nouveau, car l'entraînement peut prendre des heures et des heures dans le cas d'opérations complexes ! Le passage d'une information en mémoire à une information en texte s'appelle la sérialisation. Vous allez donc sérialiser, en transformant votre réseau de neurone en hiérarchie compatible avec un format de configuration.

```
t_bunny_configuration *save_neuron_network(t_neuron_network *net);
t_neuron_network *load_neuron_network(t_bunny_configuration *cnf);
```

Vous pouvez utiliser **bunny_load_configuration** et **bunny_save_configuration**.

11 – OXO

Vous allez maintenant entraîner un réseau de neurones pour qu'il apprenne à jouer au morpion et qu'il devienne un redoutable adversaire. En entrée, vous aurez par exemple l'état du plateau de jeu, et en sortie la décision à prendre. Votre heuristique devra évidemment être sévère avec les individus qui enfreignent les règles du jeu.

La sévérité avec ceux qui enfreignent les règles du jeu devrait suffire à ce que votre intelligence artificielle sache jouer.

N'hésitez pas à équiper votre programme d'une interface graphique pour pouvoir vous mesurer à votre intelligence artificielle une fois que celle-ci sera entraînée. Votre programme devrait prendre en paramètre au moins deux paramètres : l'IA du joueur 1, l'IA du joueur 2, avec comme possibilité de spécifier « HUMAN » pour jouer à la main à la place.

Il existe de nombreuses façons d'entraîner votre réseau : récompenser les gagnants, bien sur, mais également fournir un set de test contenant les coups à jouer, par exemple… Si vous faites jouer des réseaux entre eux des parties entière, vous risque seulement de les entraîner à jouer le premier coup, *prenez garde, c'est beaucoup plus complexe que ça en a l'air.*

Si vous êtes parvenu jusqu'ici en ayant réussi tous les exercices,
vous êtes parés pour le projet IAP4 !