Documentation pour le développeur
ANIMAUPHINE

Ce projet a pour objectif de simuler en langage C la survie de différentes espèces au cours de plusieurs milliers/millions de générations, selon leur génétique et les conditions de leur environnement. Tout au long de notre code, nous utilisons les conventions suivantes :

- les noms de variables et de fonctions sont en anglais
- les noms de variables et de fonctions sont de la forme nom_variable_ou_fonction
- les commentaires sont en français
- le code est indenté de façon conventionnelle

Afin d'implémenter cette simulation de sélection naturelle, nous avons divisé le projet en plusieurs parties : tout d'abord, nous lisons le fichier source et nous créons le monde. Dans un second temps, nous continuons à lire le fichier d'extension *.phine* et nous créons les animaux. Ensuite, le programme simule la vie, les déplacements, la reproduction et la mort des animaux. Enfin on génère un fichier de sortie d'extension *.phine* contenant les données initiales sur le monde, la Beauce, la nourriture et le seuil de reproduction, ainsi que tous les animaux ayant survécus. Notre code est réparti dans les fichiers *main.c, animal.c, list.c et world.c*, accompagnés de leur fichier d'en-tête *.h*.

I. Lecture des données du monde (Estelle Rohan)

Tout d'abord, le programme doit lire les informations sur le monde dans le fichier d'entrée.
Pour cela, nous voulions au départ lire ligne par ligne le fichier en lisant en premier la taille du monde, puis de la Beauce, etc. Cependant cette méthode suppose que les informations se trouvent exactement les unes à la suite des autres, or on doit pouvoir mettre des commentaires, même s'ils prennent une ligne entière. Il faut donc faire une boucle while.

Finalement, on lit donc le fichier ligne par ligne (avec la fonction `gets()`) dans une boucle while qui, tant que la liste des animaux n'a pas commencé :
1) appelle la fonction ***`remove_first_spaces`*** qui enlève les éventuels espaces et tabulations en début de ligne
2) appelle la fonction ***`remove_comments`*** qui parcourt la ligne caractère à caractère et si elle rencontre un '#' le remplace par '\0' pour gérer les commentaires
3) puis passe le premier caractère de la ligne dans un switch : si c'est un '#' la ligne est ignorée, si c'est un 'M' on lit les données du monde, etc… et si c'est un '(' on sort de la boucle car la liste des animaux commence.

Pour chaque ligne lue, si les données ne correspondent pas à ce qui est attendu on renvoie un message d'erreur et on arrête le programme.
Cette méthode permet un format plus souple du fichier d'entrée car les informations peuvent être écrites dans n'importe quel ordre (sauf celles des animaux). Cependant il faut s'assurer qu'on a bien récupéré toutes les informations nécessaires. Pour cela on a créé, avant la boucle while, un tableau à 4 cases initialisées à 0, une case correspondant à une information nécessaire (le monde, la beauce, l'énergie, et le seuil de reproduction) et lorsqu'une information est bien récupérée, on modifie la case correspondante à 1. Ainsi à la fin de la boucle on vérifie bien que toutes les cases sont à 1, sinon il manque une information et on affiche un message d'erreur et arrête le programme.

Puis on peut créer le monde, un tableau à deux dimensions avec de l'allocation dynamique en fonction des dimensions lues dans le fichier d'entrée. On y accède ainsi par son identificateur qui est de type int**, c'est-à-dire un pointeur de pointeurs sur des int. On peut alors facilement le passer en argument d'une fonction, contrairement à un tableau classique dont il aurait fallu préciser à chaque fois les dimensions alors qu'on ne les connaît pas avant la compilation. Les cases du tableau sont initialisées à 0.
La Beauce n'a pas besoin d'être modélisée car il suffit juste de connaître ses dimensions et coordonnées pour la simulation.

II. Gestion des données des animaux (Estelle Rohan)

A. structures "list" et "cell"

Pour stocker les animaux on crée et utilise des structures de listes doublements chaînées de cellules contenant chacune un animal. Le fait qu'elles soient doublement chaînées permet de supprimer une cellule de la liste sans devoir garder un pointeur sur la cellule précédente.

Nous voulions initialement créer une liste dont les cellules seraient directement des animaux, mais nous nous sommes rendues compte que chaque animal était dans deux listes différentes. Il n'y aurait donc pas pu avoir un seul pointeur next dans le type animal. Deux choix s'offraient alors à nous : donner deux pointeurs next à un même animal ou créer une structure de cellule. Nous avons choisi de créer cette structure de cellule pour stocker nos animaux plus facilement.

Pour pouvoir gérer plus facilement les listes tout au long du programme on a créé plusieurs fonctions : ***init_list*** , ***add_start*** , ***delete_cell*** et ***delete_list***. Les fonctions ***delete_cell*** et ***delete_list*** permettent de libérer les cellules et la liste mais pas les animaux contenus dedans, afin de pouvoir les garder dans une autre liste (la liste de leur famille par exemple). Cela s'est en effet révélé nécessaire car l'animal se trouve à la fois dans la liste globale et la liste de sa famille, donc si on veut le supprimer et qu'on le libère dans l'une des deux listes, on ne peut alors plus le retrouver dans l'autre liste pour supprimer la cellule correspondante. Les fonctions add_start et delete_cell ne font que jouer sur les pointeurs start, prev et next afin d'obtenir le résultat voulu. La fonction delete_list est une fonction récursive qui delete_cell la première cellule de la liste puis refait appel à elle-même avec la nouvelle liste, et ainsi de suite tant que la liste n'est pas vide.
On note qu'il existe aussi une fonction ***display_list*** qui nous a permis de vérifier le bon déroulement de la simulation tout au long du développement de ce projet.

Il y a deux types de listes : la liste globale contenant tous les animaux et les listes de familles. Pour accéder facilement aux listes familles, on créera un tableau ***family_array*** qui stockera les pointeurs vers ces listes familles. Chaque famille aura un numéro de famille attribué à la lecture du fichier d'entrée et ce numéro correspond à l'indice dans lequel la liste sera stockée dans *family_array*.

On crée alors la liste globale avant de commencer à lire les animaux pour pouvoir les ajouter au fur et à mesure. Cependant on ne peut pas encore créer le tableau *family_array* car on ne connaît pas encore le nombre d'animaux, donc la taille du tableau.

B. lecture des données des animaux

Avant tout, on a créé une structure ***animal*** qui contient les champs suivants :

```
typedef struct animal {
  int *chromosome;      /*pointeur sur un tableau de 8 int*/
  int energy_level;
  int direction;
  int x;          /*les coordonnées de l'animal dans le monde*/
  int y;
  int family;   /*le numéro de famille dont on a parlé précédemment*/
};
```

On peut maintenant lire et récupérer les données des animaux.
On rappelle que la première ligne contenant un animal a déjà été récupérée précédemment, il faut donc faire une boucle do...while, dans laquelle :
1) On commence par gérer les éventuels espaces en début de ligne et commentaires avec *remove_first_space* et *remove_comments*.
2) On alloue l'espace nécessaire et on crée l'animal et son chromosome (avec une fonction ***init_animal***).
3) On lit la ligne du fichier, renvoie un message d'erreur si besoin, sinon on remplit les champs de l'animal avec les informations lues. Le fait d'avoir créé l'animal avant permet de passer directement ses champs en argument de `sscanf`, et on évite donc de créer des variables intermédiaires inutilement.
4) Enfin on ajoute l'animal à la liste globale.
La boucle s'arrête lorsqu'on arrive à la fin du fichier ou lorsqu'on rencontre un saut de ligne.

A chaque passage dans la boucle, on incrémente également un compteur cpt_family qui permet ensuite d'allouer dynamiquement de l'espace pour le tableau family_array comme prévu. On parcourt ensuite la liste globale en créant, pour chaque animal, sa liste famille associée, dont on stocke le pointeur dans le tableau family_array à l'indice associé. Bien sûr, il faut aussi ajouter l'animal au début de sa liste famille.

On a alors enfin fini l'initialisation de la simulation et la lecture du fichier, on peut donc fermer ce dernier.

III. Simulation de la vie des animaux

La simulation consiste à effectuer de très nombreuses itérations (une itération est un "pas" de temps) du processus suivant :

A. Création de nourriture (Estelle Rohan)

A chaque itération, on place dans le monde une nouvelle unité de nourriture sur une case choisie aléatoirement, ainsi qu'une unité sur une case de la Beauce si celle-ci existe grâce à la fonction ***new_food***. Afin de savoir si la Beauce existe, on teste si elle est de dimension 0 x 0. Comme on a créé le monde comme un tableau d'entiers à deux dimensions ne contenant que des zéros par défaut (une case avec un zéro ne contient pas de nourriture), la présence d'une unité de nourriture supplémentaire est symbolisée par l'ajout de 1 dans la case aléatoirement choisie. On procède de même pour la Beauce si elle existe.

B. Déplacement des animaux (Julie Pibouteau)

Une fois que la nourriture a été rajoutée dans le monde, on va parcourir la liste de tous les animaux, et pour chaque animal on effectue le traitement suivant :

L'animal a déjà une orientation initiale (une direction symbolisée par un nombre entier entre 0 et 7 car l'animal est toujours entouré de 8 cases). Pour donner à l'animal une nouvelle orientation, on va choisir aléatoirement un des huit gènes composant son chromosome, en considérant que la probabilité qu'un gène soit choisi est proportionnelle à sa valeur. Pour implémenter cet "aléatoire pondéré", on utilise la fonction ***orientation***. Au départ, nous avions pensé à créer un tableau contenant le numéro de chaque gène (de 0 à 7) autant de fois que la valeur du gène, et choisir aléatoirement une case de ce tableau pour déterminer la direction de l'animal. Toutefois, on ne connaît pas la valeur maximale que peuvent prendre les gènes et le tableau pourrait devenir très coûteux en mémoire. L'idée finale est donc de créer un tableau d'effectifs cumulés `cumulative_headcount`, c'est-à-dire que la case i de ce tableau contient la somme des valeurs des gènes 0 à i. Puis on génère aléatoirement un entier x entre 0 et l'effectif total (somme des valeurs de tous les gènes), et on compare sa valeur à celle des effectifs cumulés. Le gène sélectionné aléatoirement avec pondération sera le gène i, choisi tel que cumulative_headcount[i-1] < x <= cumulative_headcount[i].
Puis on somme la direction initiale de l'animal avec le numéro du gène sélectionné, et on affecte à la direction de l'animal la valeur de cette somme modulo 8.

Il faut à présent déplacer l'animal d'une case selon sa direction. Pour cela, il nous a semblé que la solution la plus adaptée était un switch dont les cas sont la valeur de la direction de l'animal. Nous avons donc créé la fonction ***move,*** qui contient ce switch. De plus, le monde est un tore donc pour chaque cas, il fallait penser à vérifier que la/les coordonnée(s) modifiées de l'animal était toujours dans le monde via un modulo de la hauteur ou de la largeur du monde.

C. Gestion de l'énergie de l'animal, de sa mort, et de la nourriture du monde (Julie Pibouteau)

On remarque que 1 unité d'énergie ne vaut pas 1 énergie de nourriture. On diminue l'énergie de l'animal d'une unité après ce déplacement. Si la case sur laquelle se trouve l'animal contient une unité de nourriture, on diminue la quantité de 1 sur cette case du monde, et on augmente l'énergie de l'animal de la valeur d'une unité de nourriture.

On remarque également que les conditions de vie des animaux ne sont pas équitables car, à chaque itération, on parcourt la liste des animaux en partant de l'animal en tête de liste. Celui-ci a plus de chances de trouver de la nourriture (et donc de survivre) que son congénère en fin de liste.

Si le niveau d'énergie de l'animal est inférieur ou égal à zéro, il meurt. On le retire donc de la liste de tous les animaux et de la liste de sa famille grâce à la fonction ***delete_cell.*** On peut aussi libérer l'espace alloué pour l'animal car il est supprimé des deux listes.

D. Reproduction de l'animal (Julie Pibouteau)

Au contraire, si le niveau d'énergie de l'animal dépasse le seuil de reproduction précisé dans le fichier d'entrée, il se duplique. On utilise pour cela la fonction ***reproduction*** qui met à jour le niveau d'énergie du parent et crée un nouvel animal avec les mêmes données que son parent. On

remarque que le niveau d'énergie doit être un entier. Si le niveau d'énergie du parent est un nombre impair (par exemple 25), on donne à son enfant la partie entière supérieure de la moitié de ce nombre (donc 13), et la partie inférieure de la moitié de ce nombre au parent (donc 12). On choisit aléatoirement et uniformément un des gènes du nouvel animal et on applique la mutation aléatoirement déterminée uniquement si elle ne fait pas descendre la valeur du gène à moins de 1.

On décide que si le niveau d'énergie de l'animal vaut plusieurs fois le seuil de reproduction, l'animal ne se duplique pas plusieurs fois au cours du même déplacement.

IV. Génération du fichier de sortie (Julie Pibouteau et Estelle Rohan)

Après avoir fait évoluer le monde et les animaux durant le nombre d'itérations souhaité par l'utilisateur, on veut créer un fichier de sortie reprenant les données du monde, de la Beauce, le seuil de reproduction ainsi que la valeur énergétique d'une unité de nourriture. Puis, si la Beauce existe, on souhaite trier les animaux ayant survécu selon la distance entre leur position et le centre de la Beauce.

Si la hauteur et la largeur de la Beauce ne sont pas toutes les deux impaires, il n'y pas d'unique case centrale (2 cases si une seule des deux dimensions est paire, 4 cases si les deux dimensions sont paires). Afin d'être le plus précis possible, on calcule les coordonnées du centre de la Beauce avec des nombres réels au lieu de calculer avec des entiers. Même si les coordonnées ne correspondent pas nécessairement à une case, elles désignent le point central de la Beauce.

Pour trier les animaux par ordre de distance à la Beauce, nous avions d'abord l'intention de trier la liste de tous les animaux puis de l'écrire dans le fichier. Cependant, remanier ainsi la liste était fastidieux et plus long à l'exécution, donc on a préféré une autre méthode plus simple. Tant que la liste de tous les animaux n'est pas vide, on la parcourt afin d'en trouver la cellule contenant l'animal le plus proche du centre de la Beauce. Durant ce parcours, on compare les distances de l'animal courant et l'animal le plus proche du centre pour en déterminer le minimum; les distances sont calculées par la fonction ***distance_animal_Beauce***. On écrit l'animal le plus proche du centre de la Beauce dans le fichier de sortie avec la fonction ***write_animal***, puis on supprime la cellule contenant cet animal de la liste de tous les animaux.

Enfin, on écrit dans le fichier de sortie tous les animaux regroupés par famille. On utilise la fonction ***write_list*** tout en parcourant le tableau des familles d'animaux.

V. Analyse de la complexité de notre algorithme

On note n le nombre d'itérations à effectuer et A_0 le nombre d'animaux au départ (itération 0). On considère que les dimensions du monde et de la Beauce sont négligeables par rapport au nombre d'animaux et au nombre d'itérations de temps effectuées.

- world.c
  Les fonctions ***init_world***, ***new_food***, ***display_world*** et ***free_world*** ont chacune une complexité de Θ(1) (comme on néglige les dimensions du monde et de la Beauce).

- animal.c
  init_animal, display_animal, write_animal, free_animal, orientation, move, distance_animal_Beauce ont chacune une complexité de Θ(1).

- list.c
  Soit nb_animal le nombre d'animaux contenus dans la liste.
  init_list a une complexité de Θ(1).
  add_start a une complexité de Θ(1).
  display_list a une complexité de Θ(nb_animal).
  write_list a une complexité de Θ(nb_animal).
  delete_cell a une complexité de Θ(1).
  delete_list a une complexité de Θ(nb_animal).
  free_list a une complexité de Θ(nb_animal).
  reproduction a une complexité de Θ(1).

- main.c

Les fonctions ***remove_comments*** et ***remove_first_spaces*** ont une complexité de Θ(nombre de caractères par ligne). On les appelle à chaque fois qu'on lit une ligne. Cependant on peut supposer que le nombre de caractères par ligne est toujours constant et raisonnablement faible par rapport au nombre d'animaux. On peut donc considérer que ces fonctions ont une complexité de Θ(1).
La lecture du fichier se fait donc en Θ(longueur du fichier d'entrée), soit environ Θ(A_0) car on doit lire les infos sur le monde, la Beauce, le seuil de reproduction et la valeur de la nourriture (temps constant) puis tous les animaux.

La création du monde et des animaux implique l'utilisation des fonctions ***init_world*** et ***init_list***. Puis on utilise A_0 fois les fonctions ***init_animal*** et ***add_start*** pour créer la liste globale des animaux, ce qui a une complexité de Θ(A_0). Ensuite, on utilise A_0 fois les fonctions ***init_list*** et ***add start***, pour créer les listes par famille, ce qui a une complexité de Θ(A_0).

Pour chaque itération (il y a n itérations), on appelle ***new_food*** puis pour chaque animal de la liste globale, on appelle ***orientation, move,*** puis ***delete_cell*** et ***free_animal,*** ou ***reproduction.*** Toutes ces fonctions en Θ(1) sont appelées pour chaque animal, ainsi pour chaque itération i, on a une complexité de Θ(A_i), sachant que le nombre d'animaux peut augmenter, diminuer ou stagner entre deux itérations selon la survie, la mort ou la reproduction des animaux.
Au total, la complexité pour les n itérations est égale à

$$\Theta(\sum_{i=1}^{n}(A_i)). \qquad (1)$$

Enfin, si la Beauce existe (cas le plus courant), on trie les animaux selon leur distance à la Beauce. La complexité de cette algorithme est

$$\Theta(\sum_{k=1}^{A_n} k) = \Theta((A_n)^2). \qquad (2)$$

Et on écrit les listes familles dans le fichier de sortie en Θ(A_n).

Dans le pire des cas, chaque animal se duplique à chaque itération et ne meurt pas, donc au bout des n itérations, le nombre d'animaux à l'itération n est A_0x 2^n = A_n.
Ainsi (1) devient Θ($\sum_{i=1}^{n}(A_0 \, x \, 2^i)$) = Θ(2 x A_0 x(2^n-1)) = Θ(A_0 x 2^{n+1}) = Θ(A_n x 2)
Et (2) reste Θ((A_n) ²).

La complexité de notre projet dans le pire des cas est donc :

Θ((A_n) ²) = Θ(($A_0 x \, 2^n$) ²).

VI. Des améliorations possibles…

Nous n'avons pas eu le temps de générer des fichiers images. Par conséquent, ajouter le paramètre [c] à l'exécution de notre projet est considéré comme une erreur dans notre code.
Nous aurions pu améliorer le projet en lisant les arguments K-M-G en ligne de commande pour le nombre d'itérations à effectuer.
Nous aurions également pu rajouter un paramètre sur le nombre de cases d'un déplacement pour rendre le code plus avancé.
Nous aurions aussi pu créer une structure World et une structure Beauce pour faciliter l'appel de fonctions en passant uniquement les pointeurs vers les structures, au lieu de passer toutes les dimensions et coordonnées à chaque fois.