CHOIX DES PROGRAMMEURS

La réalisation de notre projet a nécessité une répartition du code source en plusieurs fichiers. Chacun (hormis celui du programme principal) dispose d'un header comprenant les structures et prototype de fonction de son homologue.

Avant de commencer, nous tenons à annoncer que l'ensemble du code ainsi que la documentation ont été réalisé à deux, sans répartition du travail.

I- Initialisation de la simulation

Avant de d'effectuer la simulation de la sélection naturelle, nous devons l'initialiser. C'est ainsi que le fichier *Init.c* contient toutes les fonctions nécessaire à l'initialisation des différentes structures ou tableaux nécessitant une allocation de mémoire. On y trouve aussi d'autres fonctions de base liées à des structures.

- Nous avons des fonctions permettant d'initialiser les structures correspondantes en allouant l'espace mémoire nécessaire à leur utilisation, tout en ayant une complexité de $\theta(1)$.
- Nous avons rencontré des difficultés à allouer de la mémoire pour notre tableau à deux dimensions, qui gère la présence ou non de nourriture dans une cellule. C'est ainsi que nous nous sommes renseignés sur internet et avons appris que pour cela, nous devions allouer *h* tableaux d'entiers int*. Et pour chacun, nous devions allouer *w* entiers int (*h* et *w* représentent respectivement la hauteur et largeur du monde). Cependant, mon binôme et moi même avions préféré utiliser un calloc, pour la fonction *InitCell*, qui permet l'initialisation direct de tous les champs à zéro. Cela permet un gain de complexité, passant de $\theta(h*w)$ à $\theta(h)$. Ce tableau sera libéré à la fin grâce à la fonction freeCells, de complexité $\theta(h)$ dans laquelle on libère chaque sous tableau, puis le tableau principale.
- Les fonctions *AddAnimal* et *SuppAnimal* sont deux fonctions de base pour les listes chaînées. La première ajoute un animal en tête de liste pour une complexité de $\theta(1)$ et la seconde supprime n'importe quel animal, pour une même complexité. Il faut cependant sauvegarder l'animal précédant l'élément à supprimer; ce qui a d'ailleurs été l'une des difficultés rencontrée était, comme nous l'expliquerons un peu plus tard, lorsque nous cherchons à supprimer un animal bien précis. Cela permet donc de garder une complexité de $\theta(1)$ et de ne pas avoir à reparcourir toute la liste chaînée à chaque fois.
- *CopyAnimal* servira plus tard à copier tous les champs d'un animal, car, les animaux sont dupliqués lors de la reproduction mais aussi lors du classement par famille. Cela nous permet d'avoir deux animaux « différents », au sens allocation mémoire, et non pas deux pointeurs sur le même animal.

La simulation nécessite une récupération des informations à partir du fichier entré en ligne de commande par l'utilisateur et l'écriture dans le fichier de sortie *sortie.phine*. Pour cela, *ReadPhine.c* permet l'extraction des parametre obligatoire à l'execution ainsi que l'ecriture dans notre fichier de sortie.

- La fonction *ReadPhine* extrait les informations relatives au monde. Elle aussi vérifie que les informations sont correctes, i.e du bon format. La fonction se termine en renvoyant le nombre de familles *nbFam*, i.e le nombre d'animaux initiaux, ce qui nous servira plus tard à créer un tableau de famille (donc un

tableau de pointeurs sur listes chaînées d'animaux) de taille *nbFam* (qu'on appelle *m* par la suite).

- La fonction *PrintPhine* écrit dans le fichier de sortie les résultats de la simulation. Elle parcourt la liste des animaux survivants afin de les afficher, cette opération nécessite un temps de $\theta(n)$, où *n* représente le nombre de survivants. Par la suite, elle parcourt le tableau de familles avec une complexité totale de $\theta(n * m)$.

II- La simulation

Le fichier *Simulation.c* inclut les fonctions permettant d'effectuer un seul tour de la simulation. Pour réaliser l'ensemble de la simulation, ces fonctions sont appelées dans une boucle de *t* itérations. (*t* représente le l'unité de temps entrer en ligne de commande par l'utilisateur).

- *RandomDirection* définit la direction selon laquelle l'animal se déplace. Afin de simuler l'aléatoire pondéré d'activation d'un gène, nous avons, en premier lieu, trouvé la stratégie suivante : nous créons un tableau *T* de taille *S* qui correspond à la somme des valeurs des gènes dans lequel chaque numéro de gène se répète autant de fois que sa valeur. Ainsi, nous choisissons un nombre aléatoire *i*, entre 0 et *S*, le gène ainsi activé est alors de numéro *T[i]*. Cependant, lors de la simulation nous avions constaté que notre temps d'exécution était extrêmement supérieur par rapport à nos camarades. C'est ainsi, qu'en vérifiant notre code, nous nous apercevions que l'intégralité des fonctions de ce fichier étaient optimisées, à nos yeux, à l'exception de cette partie. Donc, nous avons changé de stratégie. Tout d'abord, nous réalisons la somme *S* des valeurs des gènes, puis nous choisissons un nombre aléatoire *r* entre 0 et *S*-1. Ensuite, nous gérons cela comme si r se trouvait dans un tableau de somme cumulée décroissante de nos valeurs de gènes, afin de choisir lequel activer. La boucle utilisée explicite bien ce mécanisme. Cette nouvelle stratégie nous a permi de passer d'un $\theta(S)$ à un $\theta(1)$, puisque maintenant, dans le pire des cas, il y a 8 itérations.

- *AfterMoveAnimal* correspond à tout ce que subit chaque animal de notre liste à un temps t, à savoir son déplacement ainsi que son éventuelle reproduction ou mort. La première difficulté était de supprimer un animal mort avec une complexité de $\theta(1)$, car pour cela, nous avons dû garder le précédent à chaque fois, auquel cas il aurait fallu réitérer sur toute la liste pour chaque suppression, jusqu'à retrouver l'animal en question. Sans cette astuce et sans l'optimisation de RandomDirection, AfterMoveAnimal aurait été en $\theta(n^2 * S)$ au lieu du $\theta(n)$ actuel. La seconde difficulté était liée à un accès mémoire interdit. En effet, lorsque l'on dupliquait un animal, nous ne faisions que mettre un pointeur sur le même élément et l'ajouter en début de liste. Donc si un parent et son nouveau né venaient à mourir, on libérait deux fois le même espace mémoire, ce qui est interdit. C'est à ce moment là que nous est venue l'idée de créer la fonction *CopyAnimal* évoquée plus haut, afin d'allouer un nouvel espace mémoire pour chaque naissance et permettre une copie de chaque champ individuellement.

III- Traitement des données

Afin de pouvoir traiter nos résultats et finalement les analyser, nous avons créé le fichier *Traitement.c* qui, comme son nom l'indique, dispose des fonctions nécessaires à la préparation des résultats afin de les étudier. Premièrement, nous répartissons les animaux dans un tableau *TabFam* de listes chaînées d'animaux, créé dans la fonction *CreateListFam*. Cette dernière fait appel à une autre fonction, nommée *FulfillFam* avec une complexité de $\theta(n)$ (n le nombre d'animaux final). Cette dernière parcourt la liste des animaux obtenus à la fin de la simulation et remplit le tableau *TabFam* grâce au numéro de famille de l'animal stocké pendant la récupération des animaux dans *ReadPhine*.

Dans un second temps, nous classons les animaux par rapport à leur distance au centre la Beauce (si elle existe), calculé grâce à la fonction *FindCenterB*. Pour cela, il faut chercher l'animal de la liste dont la distance à la Beauce est la plus grande (nous l'appellerons par la suite *max*), l'insérer dans une nouvelle liste triée puis le supprimer de la liste initiale. Nous répétons cette opération jusqu'à ce que la liste initiale soit vide. La fonction *PrevMaxDist*, avec une complexité de ($n!$), parcourt la liste initiale afin de trouver *max* tout en renvoyant son précédent, nécessaire à la suppression de *max* (nous pouvons obtenir l'élément à supprimer en gardant son précédent, mais la réciproque est fausse). Cette stratégie permet un gain de complexité comme lors de la simulation. C'est ainsi qu'intervient le rôle de la fonction *ListSort*, qui supprime le maximum trouvé à l'aide la fonction précédente et l'insère dans une nouvelle liste qui constitue notre liste triée. Finalement, lorsque la liste initiale est vide, nous avons récupéré une nouvelle liste triée par ordre croissant de distance par rapport à la Beauce.

Et enfin, le fichier Traitement.c contient la fonction *freeAnimals* qui permet, à la fin du programme de libérer l'espace mémoire alloué pour chaque animal d'une liste passée en paramètre.

IV- Génération d'image

Finalement, dans le but d'avoir des résultats pas seulement écrits mais aussi graphiques, nous générons, à l'aide du fichier *Image.c*, des images dont le nombre est definit par l'utilisateur lors du lancement de la simulation. Dans les images, chaque couleur dispose d'une signification. C'est ainsi que nous avons mis le monde en noir, la beauce en blanc et la nourriture en gris. Cependant, les autres couleurs représentent chacune une famille différente.

Premièrement, pour chaque famille, nous associons une couleur choisie aléatoirement. Néanmoins, afin d'éviter au maximum l'obtention de couleurs très rapprochées et indistinguable à l'oeil nu, nous fixons à 255 un des niveaux de RGB choisit aléatoirement (maximum d'intensité lumineuse). Par la suite, nous choisissons (toujours aléatoirement) un autre niveaux que l'on fixe à 0. Et enfin, le niveau restant est un nombre aléatoire entre 0 et 255. Nous avons choisi cette méthode afin, d'une part, d'éviter une couleur de famille noir, blanche ou grise, correspondant respectivement au monde, la beauce et la nourriture, et d'autre part, d'obtenir une couleur sans aucune nuance de gris.

Après avoir fixé une couleur pour chaque famille, grâce à la fonction *GenerateColors* avec une complexité de $\theta(n)$ (avec n étant le nombre de familles), nous créons le nom de chaque fichier image, comme défini dans le sujet, avec *CreateName*.

Enfin, pour réaliser l'image, nous initialisons un tableau de pixels à deux dimensions analogue aux cellules du monde, utilisé avec *ColorPixels* qui stock, pour chaque cellule contenant un ou plusieurs animaux, la couleur de famille de celui qui dispose de la plus haute énergie. Pour cela, notre structure d'un pixel contient non seulement un tableau d'entiers correspondant aux couleurs, mais aussi un entier représentant une potentielle énergie (initialisée à 0) sur la case correspondante. Cela permet donc de parcourir la liste **une et une seule fois** pour chaque image. En conséquences, nous optimisons la complexité de cette fonction passant de $\theta(n^2)$ à $\theta(n)$.

En parcourant toutes les coordonnées *i,j* de notre monde avec *CreateImage*, le mécanisme de génération d'images, basé sur la couleur de chaque pixel, est ensuite le suivant :

- Nous testons l'énergie potentielle du pixel d'indice *i,j* (qui peut être positif ou nul). S'il est strictement positif, alors nous copions les champs de couleur dans le fichier image.
- Sinon, nous testons si de la nourriture est présente sur la cellule *i,j* (grâce à notre tableau de cellules). Si c'est le cas, la couleur est grise.
- Sinon, si la Beauce existe nous testons si les coordonnées *i,j* appartiennent à celle-ci. La grosse difficulté rencontrée ici provient du fait que la Beauce ne se trouve pas forcément au centre du monde (elle peut être entière, divisée en plusieurs parties si elle est placée vers une limite du monde, etc.). Nous n'avons alors pas eu d'autre choix d'appliquer ce que l'on a appris en architecture des ordinateurs. C'est à dire de lister tous les cas possibles où nous nous trouvons dans la Beauce, puis de simplifier tout cela au maximum (Et quand bien même il reste un grand pavé de conditions !). Si toutes ces conditions sont respectées, la couleur est blanche.
- Sinon, nous avons choisi le noir pour définir une case lambda de notre monde, qui ne contient ni animal, ni nourriture, et qui n'appartient pas à la Beauce.

Cette fonction, dans laquelle est appelée *ColorPixels* (en $\theta(n)$), a une complexité de $\theta(n + h * w)$, où h et w correspondent respectivement à la hauteur et largeur du monde en question.

<u>Pour conclure</u> : Notre *main.c* ne fait que suivre ces quatre étapes de manière linéaire, en faisant appel aux fonctions une par une. Nous sommes assez fiers de notre code niveau complexité. Cependant, comme nous l'avons cité dans le fichier *exp.pdf,* les animaux en tête de liste ont plus de chance de survivre si plusieurs se trouvent sur une même case contenant de la nourriture, ce qui gâche légèrement l'aléatoire. Nous voulions pouvoir arranger cela en mettant un système de combat qui ferait que sur une même cellule, l'animal ayant le plus d'énergie tuerait les autres (et son énergie baisserait en conséquence). Nous espérons que ce code saura combler vos exigences.