LES FONCTIONS :

Ces trois fonctions sont chargés de la sécurisation des entrées au clavier.

int lire(char *chaine, int longueur);

La fonction lire va recuperer au clavier au maximum longueur caracteres ou n caracteres jusqu'au retour a la ligne.

void viderBuffer();

La fonction viderBuffer va vider le buffer a la suite d'un appel à la fonction lire.

long lireLong();

Dans notre programme nous recuperrerons des chaines de caracteres ainsi que des entiers, pour cela nos appelons la fonction lireLong avec un changement de type (int). Cette fonction appel lire() et converti la chaine de caractere en un nombre du type long.

void afficher_plateau(struct Case tab[SIZE][SIZE]);

Cette fonction prend en argument le plateau cad un tableau de dimension 2 de Cases et l'affiche. Pour cela on utilise une double boucle for pour parcourir les lignes puis les colonnes en affichant a chaque fois l'element present dans la case. A a fin du parcours d'une ligne on retourne à la ligne. Pour plus de clareté, nous avons ajouter une ligne et une colonne qui correspondent aux numeros de lignes/colonnes.

- ➔ Complexité de l'algorithme : $O(SIZE^2)$ en raison des deux boucles for imbriquées.

void Combat (struct Monde* m,struct Fourmi* ant1, struct Fourmi* ant2);

Pour gerer le combats, nous avons créer une fonction Combat qui prend en argument le monde, et deux pointeurs vers des fourmis. Il est important de noter que ant1 est la fourmi qui se deplace et ant2 la fourmi qui reste immobile. Pour gerer l'issue des bataille, nous utilisons la commande srand()%(3)+1 qui nous permet de tirer aleatoirement un entier entre 1 et 4.

Dans cette fonction on traite deux cas :

1) Les agents qui se battent sont soit 2 soldats soit des non-soldats : On estime donc que chacun a autant de chances que l'autre de gagner (1chance/2) ;
2) Un seul des deux agents est un soldat : On estime alors que la soldat a 75% de chances de gagner (donc 3chances/4).

A l'issue du tirage aleatoire et en fonction du cas traité (1 ou 2) nous devons effacer le maillon correspondant a l'agent vaincu de la liste chainée de sa fourmiliere. Ici aussi deux cas s'offraient a nous :

1) L'agent n'est pas un fourmiliere, il suffit donc de supprimer juste son maillon.
2) L'agent est une fourmiliere, la tache est plus complexe. En effet il faut d'abord detacher cette fourmiliere de ces eventuelle voisine, puis il fait parcourir toute la liste chainnée correspondant aux agents qui luis sont liés en les supprimant un par un. Ici aussi Nous avions deux cas a traiter :
    1) L'agent ant2 n'est pas une Ouvriere (ant2->type !=3), on peut donc supprimer le maillon lui correspndant.
    2) L'agent ant2 est une Ouvriere (ant2->type ==3), il nous faut supprimer cette agent, et en créer un autre de meme type, à la meme position mais appartenant au camp ennemi. Pour cela on stocke sa position et on fait appel à la fonction CreerOuvriere sur une fourmi dont on vient d'allouer l'espace mémoire, on la place sur le plateau à la bonne position et on lie cet agent à la liste chainée de la fourmi ant1.

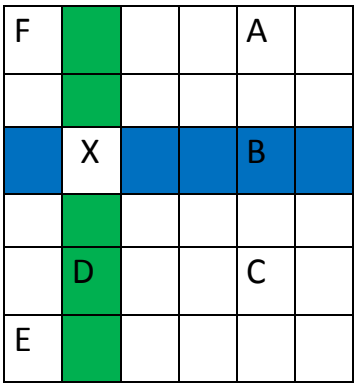
Pour la gestion des cas où il s'agit d'une fourmilière nous avons crée une deuxieme fonction :
void CombatFour(struct Monde* M, struct Fourmi* ant1, struct Fourmi* ant2) ;

➔ Sa complexité ne depend que de la longueur (L) de la liste chainnée parcourue pour la suppresion des agents, elle est donc au maximum en O(L). De plus comme la fonction combat appel cette fonction la complexité de Combat est egalement de l'ordre de O(L) (pire des cas) car toutes les autres operations effectuées par Combat sont en O(1)).

int ordre_fourmi (struct Monde* World, char *ordre, int lignedep, int colonnedep,int lignearr, int colonnearr, int*lignetmp, int*colonnetmp);

Cette fonction joue un rôle clé dans le programme, en effet elle est chargée de gerer tout les ordres donnés aux par l'utilisateur. On a trois ordres majeurs :

1) L'utilisateur veut faire bouger la fourmi se trouvant a la position (lignedep, colonnedep) vers la position (lignearr, colonnearr), donc *ordre= « bouger ». Avant toute chose, nous verrifions si nous pouvons faire bouger l'agent en question cad s'il ne sagit pas d'une ouvriere immobilisé (type==3 et immo==1), ou encore si lignedep==lignearr et colonnedep==colonnearr. Si l'agent peut se deplacer nous continuons. Pour cela nous avons repertorié tous les cas de deplacement possibles



Imaginons que nous voulons depacer la fourmi X sur ce plateau, nous travaillons alors en fonction de ligneX et colonneX,
1) Deplacement en A, on a ligneA<ligneX et colonneA>colonneX, on va donc incrementer ligneX et ligneY (lignedep++ et colonnedep++). On modifie donc X->pos[0] et X->pos[1]. X se deplace donc d'une case en
2) Depaclement en B, on a ligneB==ligneX et colonneB>colonneX, on ne va incrementer que colonneX.
3) Deplacement en C, on a ligneC>ligneX et colonneA>colonneX, on va donc decrementer ligneX et incementer ligneY (lignedep-- et colonnedep++). On modifie donc X->pos[0] et X->pos[1]. X se deplace donc d'une case en

4)Deplacement en D, on a ligneD>ligneX et colonneD==colonneX, on ne va incrementer que ligneX.
5)Deplacement en E, on a ligneE>ligneX et colonneE<colonneX, on va donc incrementer ligneX et decrementer colonneX (lignedep++ et colonnedep--). On modifie donc X->pos[0] et X->pos[1]. X se deplace donc d'une case en
6) Deplacement en F, on a ligneF<ligneX et colonneF<colonneX, on va donc incrementer ligneX et decrementer ligneX et colonneX (lignedep++ et colonnedep--). On modifie donc X->pos[0] et X->pos[1]. X se deplace donc d'une case en.

Il est important ici de noter que la fonction ne fait bouger l'agent X que d'un position, il va donc falloir la rappeler tant que fourmi->pos[0] !=fourmi->dest[0] ou que fourmi->pos[1] !=fourmi->dest[1] .

2) L'utilisateur veut immobiliser l'agent, ici il y'a deux cas auxquels nous devons preter attention :

1) L'agent en question n'est pas une ouviriere, on passe juste sans executer d'instructions particulieres.

2) L'agent en question est une ouvriere, on doit donc l'immobiliser pour le reste du jeux, donc : World->plateau[lignedep][colonnedep].fourmi->immo=1.

3) Enfin, l'utilisateur veut detruire l'agent en question, pour ce faire, nous supprimons le maillons correspondant à cette agent de la liste chainnée à laquelle il appartient.

-> Ici quel que soit la valeur des entrées, on effectue au maximum deux operations classiques (dans le cas ou on bouge la fourmi

void CreerFourmiliere (struct Fourmi* ant, char color, int ligne, int colone);void CreerReine (struct Fourmi* ant, char color, int ligne, int colone);

void CreerOuvriere (struct Fourmi* ant, char color, int ligne, int colone);

void CreerSoldat (struct Fourmi* ant, char color, int ligne, int colone);

Ces quatres focntinos sont similaires, ells ont toutes les quatres pour but dinitialiser les champs de la structures Fourmi pour creer des fourmilieres, Reines, Ouvrieres ou encore des Soldats. Elles differents cependant au niveaux d'une instruction.

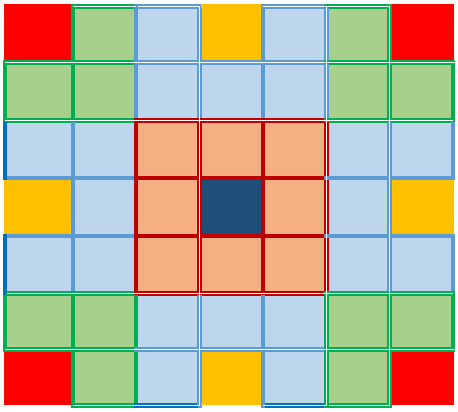
Evidement toutes les fourmis ne sont pas du meme type : les fourmillieres sont du type (1), les Reines du type (2), les Ouvrieres du type (3) et les Soldats du type (4). Les valeurs de champ ant->*type* differes donc selon la fonction appelée.

Pour le reste les quatres fonctions vont faire la meme chose. Le champ *ant->color* va prendre la valeur du parametre formel color, et les champs *ant->pos[0]* et *ant->pos[1]* vont respectivement prendre les valeurs ligne et colonne passées en paramètre. Tous les champs correspondants aux pointeurs *suiv, prec, next* et *prev* sont quant à eux initialisés à NULL.

int Libre (struct Monde* world, int ligne, int colonne, int* i, int* j);

Cette fonctions joue un role important lors de la production d'un nouvel agent par une fourmiliere, en effet apres appel, les parametres *i* et *j* prennent la valeurs de la ligne et de la colonne d'une case voisine de la fourmiliere. Si une case est trouvé elle renvoie 1, sinon elle renvoie 0. Les parametres ligne et colonne eux correspondent à la position de la fourmiliere qui tente de produire un agent.

Pour cette fonction nous avons traité 8 cas (beacoup de lignes de code ce qui n'est pas tres esthetique, mais ceci est la premiere idée qui nous soit venue, elle nous assure egalement que nous traitons tous les cas). Nous n'avons pas crée une fonction qui parcours plusieurs 'couches' de cases autour de la fourmiliere, en effet un fourmiliere doit créer ses agent et les placer sur une de ses cases voisines, donc nous ne travaillons que sur les cases qui lui sont mitoiennes.



Considérons le plateau ci-contre,

-Notre fonctions va d'abord traiter les cas « critiques » c'est dire les quatre coins en rouge.
En effet en fonction du coin considéré ((0,0), (0, SIZE-1), (SIZE-1, SIZE-1) ou (SIZE-1, 0)) donc on fonction des valeurs des paramètres ligne et colonne, nous allons parcourir les 3 cases voisines correspondantes (cases aux bordures vertes).

-Puis nous traitons le cas des bords. Ici aussi en fonction du bord considéré (droit : (X, 0), haut : (0,X), gauche : (X,SIZE-1), ou bas : (SIZE-1, X) avec X ϵ]SIZE-1, SIZE-1[), en fonction des valeurs des paramètres ligne et colonne, comme pour le cas des coins, nous savons donc parfaitement quelles sont les 5 cases à parcourir(cases aux bordures bleues).

-Enfin tous les autres cas se traitent de la meme manière, nous n'avons plus a nous soucier de le position de la fourmiliere, en effet il ne nous reste plus qu'a parcourir les 8 cases autour de la fourmiliere (exemple des cases aux bordures rouges).

Dans tous les cas, nous n'utilisons aucune boucle.

int produire_agent (struct Monde* plat, int ligne, int colonne, int code,char color, struct Fourmi*ant);

Voici la fonction en charge de la production d'un agent par une fourmilière, ses parametres formels correspondent à:

- Un pointeur sur le Monde,
- Deux entiers, ligne et colonne qui representent la position de la fourmilière sur le plateau
- Un entier, code qui correspond à la classe de la fourmi que l'on veut créer
- Un caractere color pour la couleur de l'agent
- Un pointeur vers la future fourmi que l'on va créer.

On commence d'abord par verifier que la fourmi qui se trouve à la position (ligne, colonne) est bien une fourmi (*plat->plateau[ligne][colonne].fourmi->type==1*) et aussi qu'il y'a une case libre autour d'elle (*Libre(plat, ligne, colonne, &i, &j)* !=0 avec i et j deux entiers déclarés auparavant).

Puis en fonction de la valeur de *code* nous créeons une nouvelle Reine, Ouvriere ou un nouveau Soldat grace aux fonctions Créer (pour une Reine par exemple : *CreerReine(ant, color, i, j)* où ant est une fourmi à laquelle nous avons declaré l'espace mémoire necessaire).

Il nous faut ensuite placer l'agent fraichement crée sur le plateau, à la position plat->plateau[i][j].

Enfin, il faut ajouter cet agent à la liste doublement chainée des agents de la fourmiliere, nous avons decidé d'inserer le maillon du nouvel agent au debut de la liste doublement cahinée (c'est-à-dire juste apres le maillon de la fourmilliere) pour plus de facilité.

L'orsque l'agent est finalement produit est installé nous prevenons le joueur avec un message à l'ecran.

Si nous n'arrivons pas à créer un agent parcequ'il n'ya pas de case libres autour de la fourmilière, nous prévenons le joueur en lui disant que la production est en attente et qu'il doit libérer une case pour l'agent soit produit.

void Save (struct Fourmi *ant1, struct Fourmi *ant2, int goldred, int goldblack, int roundPlayer, int round) ;

Cette fonction va nous permettre de sauvegarder la partie. On peut faire appel à cette fonction au debut de chaque tour, c'est à dire lorsque le joueur choisit de sauvegarder la partie au debut de son tour.

Elle prend comme argument :

-Les listes de fourmis rouges et noires ;

-Le nombre de pieces de chaque joueur(on prend seulement les valeurs, les variables ne seront pas modifiees).

-La variable round, soit quel tour de la partie ;

-Dans le main, roundPlayer=0 si le joueur 1 est en train de jouer, 1 si c'est le 2^e joueur qui joue. Cette variable nous permet de savoir quel joueur a sauvegarde, afin de retomber directement sur le tour de ce joueur lorsque l'on charge la partie.

Dans cette fonction save, on initialise 4 pointeurs par malloc en leur allouant l'espace d'une structure fourmi :

-On commence par créer un fichier f, puis on ecrit dessus la premiere ligne le nombre de pieces de chacun, la valeur de round et roundPlayer ;

-Ensuite, on affecte aux pointeurs p et pp la liste de fourmis rouges.

1ere etape : p et pp vont pointer sur le premier element de la liste, soit la fourmiliere. On dispose de deux boucles iteratives :

Tant que pp different de NULL (soit la principale) :

Au debut p=pp, soit la fourmiliere, ensuite la deuxieme boucle :

Tant que p different de NULL :

P pointe sur un element, non nul. On va alors transcrire sur le fichier les informations de l 'element sur le fichier, selon le type :

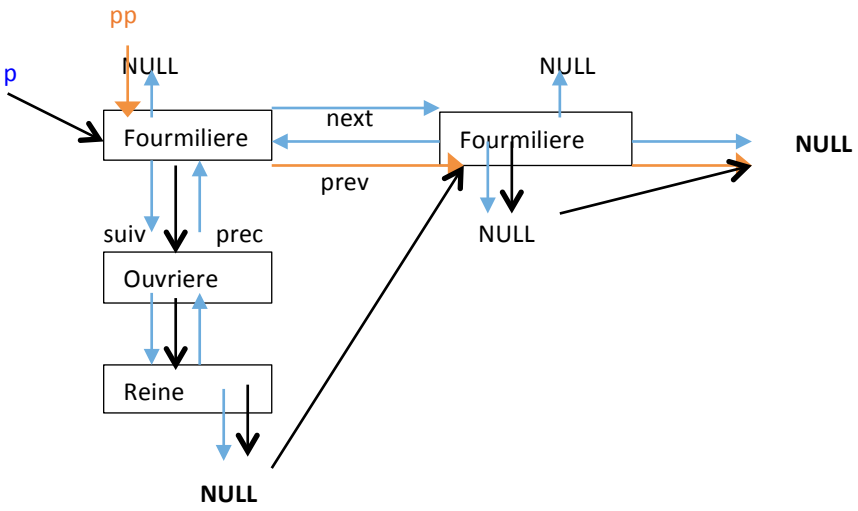
+ Si fourmiliere (type 1) , alors la couleur, le type, la position, le type et temps restant de la production, ainsi que tmps d'immobilisation (celui ci inutile, il servira seulement à ne pas laisser une case vide dans un tableau pour plus tard).

+Sinon, alors la couleur, type, position, case en destination, temps d'immobilisation.

Lorsque p=NULL, soit fin tant que,

Alors on affecte a pp= pp->next, soit la prochaine fourmiliere si existe.

Si elle existe, alors p=pp, la 2e fourmiliere, et on reitere ensuite notre procede (represente par le schema suivant).

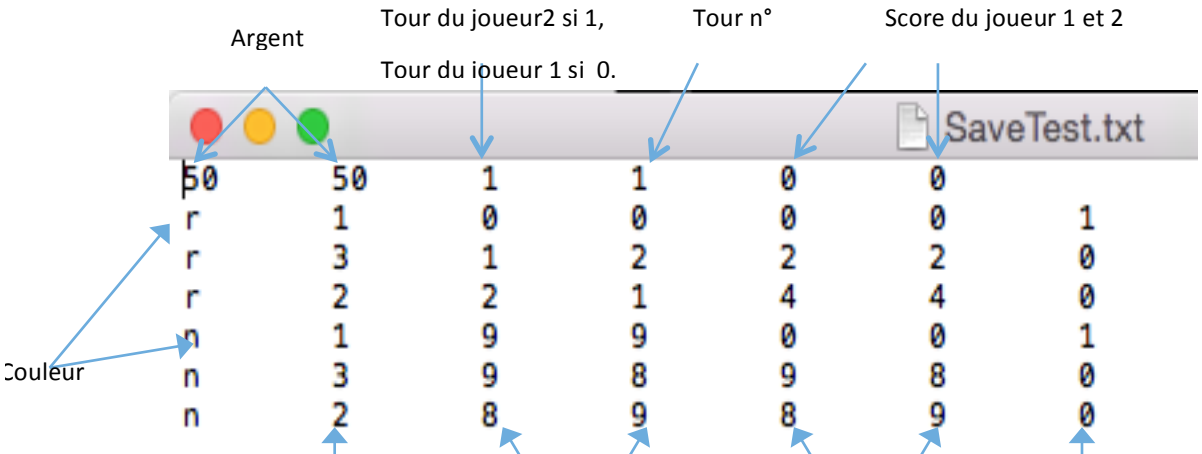


pp va parcourir chaque liste chainee (fleches en orange).

P va parcourir chaque element de la liste des fourmis rouges.

Meme procede pour la liste de fourmis noires, puis on termine en fermant le fichier et liberant l'espace.

Au tour n°1 et lorsque le joueur 1 joue, on obtient un fichier comme ceci:



Complexité= depend de la longueur de la liste du monde de fourmis=O(1) car il y a au plus 100 fourmis,

J'ai choisi de creer la fonction save comme ceci afin de parcourir tout la liste et de noter toutes les informations dans un fichier texte ensuite. Plurs tard, si l'on souhaite charger l'ancienne partie, on aura recours a ce fichier texte afin de recuperer toutes les informations.

void init_monde (struct Monde* world, struct Fourmi* Fn, struct Fourmi* Fr, struct Fourmi* Or, struct Fourmi* On, struct Fourmi* Rn, struct Fourmi* Rr);

Il s'agit ici d'initialiser le plateau en debut de partie. Cette fonction prend donc en argument :

- Un pointeur sur le Monde
- Six pointeurs sur des Fourmis qui correspondent aux 6 agents présents lors du lancement d'une nouvelle partie.

Notre fonction fait appel aux fonctions Créer pour créer tous les agents necessaires (Fourmiliere Rouge, Ouvriere et une Reine pour chacunes des deux couleurs) , une fois les six agents crées, elle les lies dans deux listes doublement chainées ayant pour premier element les deux fourmilieres, puis elle fait pointer les deux pointeurs Frouge et Fnoir de la structure Monde vers ces deux fourmiliere (On a donc pour les camp noir la liste chainée suivante : World->Fnoir=FourmilereNoire->OuvriereNoire->ReineNoire).

Int main() :

C'est dans le main que nous avons décidé de mettre toutes les instructions concernant l'interface de jeux. Ce n'est pas la meilleure methode, en effet au final nous nous retrouvons avec une fonction main assez longue, ce qui rend le code peu esthetique.

Le main comporte deux parties proncipales : une pour chaque joueur. Comme les deux parties sont assez similaires, nous n'allons en expliquer qu'une seule.

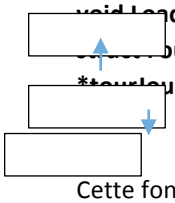
Au début de la fonction, nous avons commencé par déclarer toutes les variables dont nous allions avoir besoin lors de nos appels aux fonctions. Avant de faire jouer le premier joueur, nous affichons d'abord un menu avec 3 Options :

1) New Game (On demmare une nouvelle partie : appel à la fonction init_World
2) Load Previous Game (On charge une ancienne partie : appel à la fonction Load)
3) Show the 10 best scores (On fait appel à la fonction top_players pour afficher les 10 meilleurs scores stockés dans un fichier payers.txt).

L'utilisateur doit alors entrer un entier compris entre 1 et 3 en fonction de ce qu'il veut faire. Tant qu'il n'a pas entrer un valeur correcte (entre 1 et 3) le programme continue d'attendre une entrée.

Pour le niveau 1 nous n'avons utilisé que deux pointeurs rouges et noirs pour pointer vers les les fourmilières pointées par Frouges et Fnoire. Puis à parti du niveaux 2 nous avons ajouté 2 nouveaux pointeurs pour pointer vers les autres eventuelles fourmillieres.

void Load(struct Monde* World, struct Fourmi* Fn, struct Fourmi* Fr, struct Fourmi* Or, struct Fourmi* On, struct Fourmi* Rn, struct Fourmi* Rr, struct Fourmi* Sr, struct Fourmi* Sn, int *gold1, int *gold2, int *tourJoueur, int *tour) ;



Cette fonction est similaire à la précédente (init_monde).Au départ, l'environnement n'est pas initialise. Si l'on choisit une nouvelle partie, alors on fait appel à init_monde, ce qui nous permet de tout initialiser.

Sinon, alors on fait appel a load, celle ci prend pour arguments :

-Les memes pointeurs utilises précedemment dans init_monde

-Deux pointeurs de plus (ici appeles Sr et Sn pour soldats rouges et noirs si existants).

-Les pointeurs sur gold1 et gold2 pour modifier la valeur du nombre de pieces.

-Pointeurs pour modifier la valeur du de la variable tourJoueur(c'est a dire pour revenir sur le tour du joueur precise sur le fichier : si dans le fichier, tourJoueur=1 alors on passe directement au joueur 2, sinon on reste sur le 1), et la valeur de la variable tour.

On initialise 4 pointeurs p, pp, q, et qq à null.

On commence par créer le plateau à l'aide de deux boucles for(comme dans init_monde).

Ensuite on fait appel au fichier texte de sauvegarde que l'on va lire :

-On va commencer par lire la premiere ligne, contenant 4 entiers que l'on va stocker dans le tableau vars, et l'on va affecter les valeurs collectées à, respectivement, gold1, gold2, tourJoueur, tour. Les variables sont alors modifiees.

-Sachant que, dans tous les cas, on ne peut sauvegarder une partie si un joueur ne dispose plus aucune fourmi. Alors on peut etre sur qu'au pire des cas, chacun dispose une fourmiliere. (car si un joueur ne possede AUCUNE fourmiliere, celui ci a perdu la partie.)

De ce fait, sachant aussi qu'une fourmiliere est le premier element d'une liste et que la premiere du joueur 1 est rouge, alors on va commencer par lire la seconde ligne et ensuite créer une fourmiliere (d'argument le pointeur Fr initialise precedemment dans le main) en faisant appel à la fonction concernee, et avec les valeurs collectees (dans les tableau coul et characs). On place ensuite la fourmiliere dans le plateau de jeu.

On alloue à pp l'espace d'une structure fourmi, lui affecte Fr, puis on affecte pp à p.

Ensuite on initialise un entier n à 0 que l'on va utiliser plus tard.

Ici, on lance une boucle while, tant que fscanf different de null, soit tant qu'il reste une ligne à lire.

Au debut de chaque iteration, on va collecter les informations de la ligne prise en compte,

Si la couleur est rouge, soit un charactere 'r' perçu, alors on fait appel à une boucle switch, de parametre charac[0], soit le type :

Si type= 2 ou 3 ou 4, alors on cree un agent comme ce que l'on a fait pour la fourmiliere precedent, sauf que l'on va allouer à p->suiv l'espace d'une structure fourmi, que l'on va remplir par un la case d'un agent crée, pour ensuite faire avancer le pointeur p vers cette case. P pointe alors sur cet agent créé, et on termine par p->suiv=NULL.

Sauf que si type= 1, soit une fourmiliere, alors on va recréer une fourmiliere comme ce que l'on a fait pour la premiere. Or, sachant que pp pointe toujours sur cette premiere fourmiliere, alors on va affecter pp->next à p.

On lance ensuite une boucle itérative tant que p different null, alors on fait avancer p(next). Lorsque p pointera sur une case vide, alors on va lui affecter cet agent crée en lui ayant alloué au préalable l'espace d'une structure fourmi. P va alors pointer sur cette nouvelle fourmiliere, et les prochains agents que l'on va créer seront relies à cette nouvelle fourmiliere.

Lorsque l'on aura complété cette liste de fourmis rouges (lorsqu'il n'ya plus de caractere 'r'), alors le premier élément que l'on va trouver est une fourmiliere noire. Pour ce faire, entrer dans la boucle else, puis ensuite dans la boucle if comprise (si n=0) dans celle ci, et on va créer une fourmiliere noire comme procédé effectué pour la liste rouge, sauf que l'on va utiliser q et qq à la place de p et pp. On va ensuite modifier n pour ne plus revenir dans cette boucle if, et on fait appel à continue pour scanner la prochaine ligne et sauter les prochaines commandes.

Cette procedure impliquant la variable n va nous permettre de garder qq comme un pointeur sur la premiere fourmiliere noir.

Ensuite, on va effectuer le meme procede effectue pour la liste rouge de fourmis.

A la fin de la boucle while, on accorde a World->Frouge le pointeur pp, soit ici la liste de fourmis rouges prises en arguments va pointer sur la premiere fourmiliere rouge. De meme pour World->Fnoir et qq.

On finit par liberer l'espace utilise.

Complexite identique à la fonction save.