Rapport projet d'algorithme 2

Stratégie générale

- D'abord on regarde si le serpent voit le fruit. Comme les différents arbres ont des cases en commun, il n'est pas nécessaire de fouiller tout les arbres de fond en comble. Pour tester toutes les cases de la vision du serpent le moins de fois possible il suffit de tester les cases de l'arbre *itn*, les trois cases à l'est du serpent, l'arbre *its* et les trois cases à l'ouest du serpent.
- Il n'est pas tout à fait vrai que l'on ne connaît pas la carte. On sait qu'il y a un mur en x=0 et en y=0 et on connaît toutes les coordonnées du serpent. Ce qu'il nous manque c'est la position des murs en *xmax* et en *ymax*. Pour déterminer ces valeurs on regarde les cases les plus à l'est et les plus au sud du serpent et de sa vision. Notre programme marche donc dans une map plus petite que la map véritable.
- Pour trouver la case où le serpent doit aller on distingue deux cas :

* On voit le fruit. On utilise alors le principe de recherche A*. On liste les cases possibles autour de la tête du serpent de tel sorte que la distance au fruit soit croissante (dans cette liste). On utilise la première case de cette liste pour créer un nouveau serpent (le nouveau serpent serait identique à l'ancien si celui-ci s'était déplacé vers la première case de la liste). On donne alors ce nouveau serpent à la même fonction. Ainsi soit on trouve le fruit et on renvoie le chemin emprunté (comme on ne peut pas enregistrer le chemin, en pratique on renvoie juste les coordonnées de la première case où le serpent doit aller). Soit le serpent n'a plus de chemin où aller, dans ce cas on remonte et on prend la deuxième case de la liste des cases possibles (ou la troisième si la deuxième a été déjà testée etc).
Malheureusement le chemin emprunté par le serpent peut alors sortir le fruit de la vision du serpent.

* On ne voit pas le fruit. On utilise le même principe. Ce qui change c'est que la fonction s'arrête au bout de (taille du serpent)*2 appelle et la liste des cases possibles est triée de manière aléatoire. De cette manière le serpent va errer sur la map jusqu'à voir le fruit sans pour autant se coincer. Si aucun chemin n'est trouvé (ce qui peut arrivé sachant que l'on utilise une map plus petite que l'originale), on réessaye et on s'arrête à (taille du serpent). (si aucun chemin n'est trouvé, on réessaye avec (taille du serpent)/2 et ainsi de suite jusqu'à ce que l'on ait un chemin ou que nb_iteration=0 ie qu'il n'y ait aucune cases possibles autour du serpent).

Modules centraux

-Recherche

Recherche renvoie les coordonnées du fruit si il le trouve et {-100,-100} sinon.

coord recherche(item_tree, item_tree, item_tree, item_tree,coord);

Ce module renvoie les coordonnées du fruit à partir des 4 arbres de vision du serpent et des coordonnées de la tête du serpent. L'écrire comme fonction est donc le plus simple et le plus efficace.

Déclaration

Variables fruit en coordonnée et invalide en coordonnées

Paramètres tree_north en item_tree, tree_east en item_tree, tree_south en item_tree, tree_west en item_tree et serpent en coordonnée.

Début

```
invalide ← {-100,-100};
fruit ← recherche_arbre(tree_north);
/* on regarde l'intégralité de l'arbre au nord */
SI !egalite(invalide,fruit) ALORS
/* si on trouve le fruit, on ajuste les coordonnées et on les renvoie */
        fruit.y ← fruit.y-1;
        fruit.x ← serpent.x+fruit.x;
        fruit.y ← serpent.y+fruit.y;
        retourner(fruit);
FIN SI
fruit ← recherche_troncon(tree_east,EAST);
/* on regarde les 3 cases à l'est du serpent*/
SI !egalite(invalide,fruit) ALORS
/* si on trouve le fruit, on ajuste les coordonnées et on les renvoie */
        fruit.y ← fruit.y-1;
        fruit.x ← serpent.x+fruit.x;
        fruit.y ← serpent.y+fruit.y;
        retourner(fruit);
FIN SI
fruit ← recherche_arbre(tree_north);
/* on regarde l'intégralité de l'arbre au sud */
SI !egalite(invalide,fruit) ALORS
/* si on trouve le fruit, on ajuste les coordonnées et on les renvoie */
        fruit.y ← fruit.y-1;
        fruit.x ← serpent.x+fruit.x;
        fruit.y ← serpent.y+fruit.y;
        retourner(fruit);
FIN SI
fruit ← recherche_arbre(tree_north);
/* on regarde les 3 cases à l'ouest du serpent*/
SI !egalite(invalide,fruit) ALORS
/* si on trouve le fruit, on ajuste les coordonnées et on les renvoie */
        fruit.y ← fruit.y-1;
        fruit.x ← serpent.x+fruit.x;
        fruit.y ← serpent.y+fruit.y;
        retourner(fruit);
FIN SI
retourner(invalide);
/* Si on atteint ce stade de la fonction, cela signifie que le fruit n'est pas dans la vision du serpent */
FIN
```

Cette fonction est optimisé: chaque case que le serpent voit est testé le moins de fois possible
Cependant l'utilité pour de si petits arbres peut être débattue je le concède.

-Decision

Ce module renvoie les coordonnées du point où le serpent doit aller.

coord decision(coord, snake_list,int,int);
(coord fruit, snake_list s, int xmax, int ymax)

Comme ce module doit renvoyer des coordonnées, en faire une fonction paraît le plus simple.
Comme cette fonction se base sur le principe de recherche A*, la récursivité s'impose.

Déclarations

variables point en coord, invalide en coord, poubelle en coord, poss en pointeur sur coord, valeur_retourne en coord, i en entier, inter en struct snake_list et nouvs en snake_list.

Paramètres fruit en coord, s en snake_list, xmax en entier et ymax en entier

Début

```
point ← {s → x, s → y};
/*coordonnées de la tête du serpent*/
invalide ← {-100,-100};
poubelle ← {0,0};
i ← 0;
inter ← {SNAKE_HEAD,0,0,NULL};
nouvs ← adresse(inter);
SI egalite(point,fruit) ALORS
        retourner(poubelle);
FIN SI
/* si la tête du serpent a atteint le fruit on renvoie une réponse positive */
cases_posibles_decision(point,fruit,s,poss,xmax,ymax);
/* on calcul la liste des cases possibles */
SI egalite(poss[0],invalide) ALORS
        retourner(invalide);
FIN SI
/* si aucunes case n'est valide ie si il n'y a pas de chemin on envoie une répone négative */
nouvs ← modif(s,poss[0]);
/* on calcul le nouveau serpent grâce à l'ancien et à poss[0] */
valeur_retourne ← decision(fruit,nouvs,xmax,ymax);
/* on regarde si il y a un chemin jusqu'au fruit en passant par poss[0] */
TANT QUE egalite(valeur_retourne,invalide) FAIRE
/* tant qu'il n'y a pas de chemin valide, on essaye avec une autre case de la liste des cases possibles
*/
        SI (i==3  || egalite(poss[i+1],invalide)) ALORS
                libérer(nouvs);
/* on libère l'espace du nouveau serpent car on en a pas besoin */
                retourner(invalide);
        SINON
                i ← i+1;
                nouvs ← modif(s,poss[i]);
                valeur_retourne ← decision(fruit,nouvs,xmax,ymax);
```

```
        FIN SI
FIN TANT QUE
libérer(nouvs);
/* on libère l'espace du nouveau serpent car on en a pas besoin */
SI (!egalite(valeur_retourne,invalide)) ALORS
        retourner(poss[i]);
SINON
        retourner(invalide);
FIN SI
FIN
```

Même si certains passages de cette fonction sont peut être un peu brouillon (comme lors de la définition de nouvs avec inter), l'objectif principal de la fonction est remplie de manière extrêmement satisfaisante pour ce que j'en fais.

-Errer

Ce module renvoie les coordonnées du point où le serpent doit aller.

```
coord errer(snake_list,int,int,int);
(coord fruit, snake_list s, int xmax, int ymax,int nb_iteration)
```

Comme ce module doit renvoyer des coordonnées, en faire une fonction paraît le plus simple. Cette fonction marche de la même manière que decision donc la récursivité s'imposait.

Déclarations

variables point en coord, invalide en coord, poubelle en coord, poss en pointeur sur coord, valeur_retourne en coord, i en entier, inter en struct snake_list et nouvs en snake_list.

Paramètres fruit en coord, s en snake_list, xmax en entier et ymax en entier

Début

```
point ← {s → x, s → y};
/*coordonnées de la tête du serpent*/
invalide ← {-100,-100};
poubelle ← {0,0};
i ← 0;
inter ← {SNAKE_HEAD,0,0,NULL};
nouvs ← adresse(inter);
SI nb_iteration==0 ALORS
        retourner(poubelle);
FIN SI
/* si on a fait assez d'appel de cette fonction on renvoie une réponse positive */
cases_posibles_errer(point,fruit,s,poss,xmax,ymax);
/* on calcul la liste des cases possibles */
SI egalite(poss[0],invalide) ALORS
        retourner(invalide);
FIN SI
/* si aucunes case n'est valide ie si il n'y a pas de chemin on envoie une répone négative */
nouvs ← modif(s,poss[0]);
/* on calcul le nouveau serpent grâce à l'ancien et à poss[0] */
valeur_retourne ← errer(nouvs,xmax,ymax,nb_iteration-1);
/* on regarde si il y a un chemin jusqu'au fruit en passant par poss[0] */
```

```
TANT QUE egalite(valeur_retourne,invalide) FAIRE
/* tant qu'il n'y a pas de chemin valide, on essaye avec une autre case de la liste des cases possibles */
        SI (i==3  || egalite(poss[i+1],invalide)) ALORS
                libérer(nouvs);
/* on libère l'espace du nouveau serpent car on en a pas besoin */
                retourner(invalide);
        SINON
                i ← i+1;
                nouvs ← modif(s,poss[i]);
                valeur_retourne ← decision(fruit,nouvs,xmax,ymax);
        FIN SI
FIN TANT QUE
libérer(nouvs);
/* on libère l'espace du nouveau serpent car on en a pas besoin */
SI (!egalite(valeur_retourne,invalide)) ALORS
        retourner(poss[i]);
SINON
        retourner(invalide);
FIN SI
FIN
```

Même si certains passages de cette fonction sont peut être un peu brouillon (comme lors de la définition de nouvs avec inter), l'objectif principal de la fonction est remplie de manière extrêmement satisfaisante pour ce que j'en fais.

Limites du programmes

Un problème théorique majeur est qu'en mangeant un fruit le serpent peut se bloquer. Une solution (que j'ai fait pour le projet1) est de mettre 2 objectifs pour la fonction decision, le premier étant le fruit et le deuxième un point à l'opposé de la map. Decision trouverait un chemin allant jusqu'à objectif2 en passant par objectif1.Ainsi decision s'assurerait que le serpent mange le fruit sans se bloquer.

Un problème pratique majeur est que lorsque l'on fait fonctionner errer plusieurs fois avec différents nb_iterations pour touver un chemin dans l'hypothèse qu'un chemin de taille nb_iteration ne soit pas trouver, on fait d'importants calculs qui ralentissent le programme. A partir d'une certaine taille de serpent, ces calculs deviennent trop important et le processus est arrêté.

PS:
Si aucun chemin n'est trouvé (par decision ou errer après les réductions de nb_iteration), le programme va tuer le serpent en faisant l'inverse de la dernière action.