RAPPORT EXPLICATIF

UBFC
UNIVERSITÉ
BOURGOGNE FRANCHE-COMTÉ

du projet d'Info 4A - Labyrinthe en C et C++
Partie 3 - Course-poursuite de robots

Par : Evan PETIT - IE4 I912
Professeur de TP : M A. OUAZZANI

1. Prérequis :

Afin de mettre en place les algorithmes de déplacement de robots demandés, une fonction supplémentaire à implémenter sera très utile.

Il s'agit de **vector<int> Laby::voisins(int id, bool diag)**.
Cette fonction prend en paramètre deux éléments :
***int id* -** L'id d'une case de la grille (n'étant idéalement pas un mur) dont on veut connaître les cases voisines n'étant pas un mur.
***bool diag* -** Si *diag* vaut false, on ne vérifie que pour les 4 strictement adjacentes à *id* c'est à dire en omettant les diagonales. Si *diag* vaut true, on vérifie pour les 8 cases adjacentes, c'est à dire en incluant les cases voisines incluant les diagonales

La fonction renvoie un **vecteur d'entiers** contenant l'id des cases adjacentes à *id* n'étant pas un mur.

Elle fonctionne ainsi :
Si diag = false, à l'aide de if, on vérifie une à une si les 4 cases voisines à *id* existe (n'est pas hors du tableau) et vaut 0 (case vide).

- Si oui, on ajoute l'id dans le vecteur à retourner
- Si non, on ne l'ajoute pas.

Si diag = true, on procède de la même manière, mais on vérifie aussi pour les 4 cases diagonales adjacentes.

	(x-1,y)	
(x,y-1)	**id(x,y)**	(x,y+1)
	(x+1,y)	
diag = false		

(x-1,y-1)	(x-1,y)	(x-1,y+1)
(x,y-1)	**id(x,y)**	(x,y+1)
(x+1,y-1)	(x+1,y)	(x+1,y+1)
diag = true		

2. Algorithmes des robots :

Prédateur :

On implémente l'algorithme demandé dans le sujet : **DIRECT PREDATEUR**. Il s'agit de l'algo 1.
A chaque déplacement du robot proie :

- Grâce à la fonction **voisins**() vue précédemment, on récupère toutes les cases vides voisines à la proie (sans les diagonales) qu'on place dans le **vecteur<int> voisinsA**
- On boucle sur **voisinsA** et on récupère la case possédant la plus petite distance minimum au prédateur (fonction **distMin**). Si plusieurs cases remplissent cette condition, on les récupère toutes. L'id de ces cases sont placées dans le **vecteur<int> choix**
- On retourne aléatoirement l'un des id contenu dans choix, c'est à dire une case aléatoire parmis toutes celles réduisant le plus la distance entre les deux robots -> Cela évite, au prix de déplacements parfois non optimaux, d'avoir des cycles infinis.
- On récupère tous les voisins (diagonales incluses cette fois ci) du robot prédateur, et on place les ID de ces cases dans **vecteur<int> voisinsA**.
    - Si l'id du robot proie est contenu dans **voisinsA**, la fonction retourne false (il y a contact)
    - Sinon la fonction retourne true (il n'y a pas contact)

Proie :

On implémente deux algorithmes, dont celui demandé par le sujet. **RANDOM PROIE** (algo 1) et **PROIE FUYANTE** (algo 2). Pas besoin de vérifier s'il y a contact entre les deux robots puisque on s'en occupe dans l'algorithme du robot prédateur.

RANDOM PROIE :

Cette algorithme tient en deux lignes.

- Grâce à la fonction **voisins()**, on récupère les cases vides voisines à la proie sans les diagonales. Donc, de 1 à 4 cases, et on place leur ID dans le **vecteur<int> voisinsB**
- On retourne un ID contenu dans **voisinsB** de manière aléatoire.

Cela traduit un déplacement complètement aléatoire du robot proie, qui se fera souvent manger tout cru par le prédateur assez rapidement.

PROIE FUYANTE :

Cette algorithme est quasi identique à **DIRECT PREDATEUR.**
Sauf qu'au lieu de choisir aléatoirement entre toutes les cases réduisant la distance entre proie et prédateur, celui ci choisit aléatoirement entre toutes les cases augmentant la distance entre proie et prédateur.

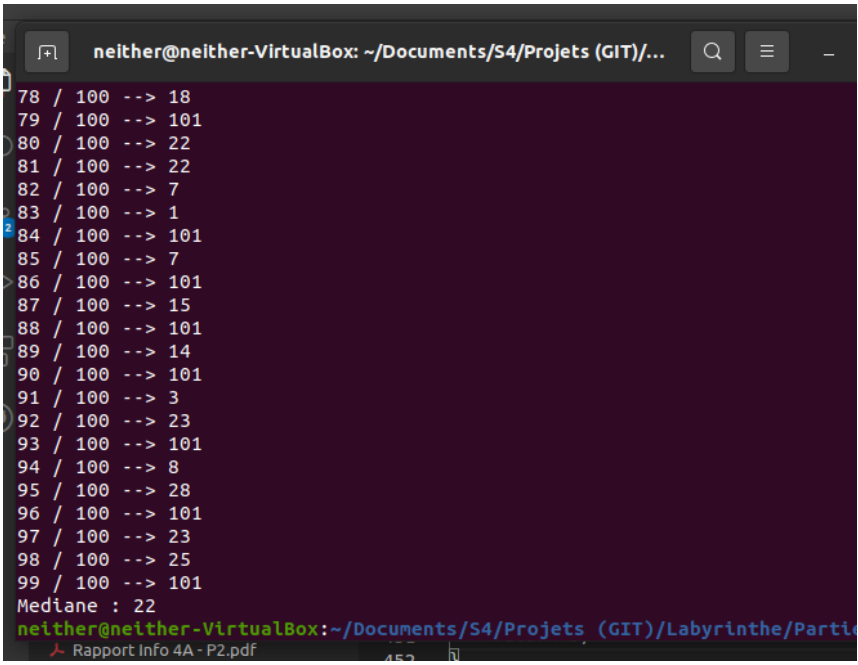
3. Tests

Les tests sont effectués sur un labyrinthe crée avec le descripteur 2 et à l'aide de la fonction testEval(), qui rappelons-le, récupère la médiane de la durée des courses poursuites sur 100 courses durant au maximum 100 tours.

DIRECT PREDATEUR VS RANDOM PROIE

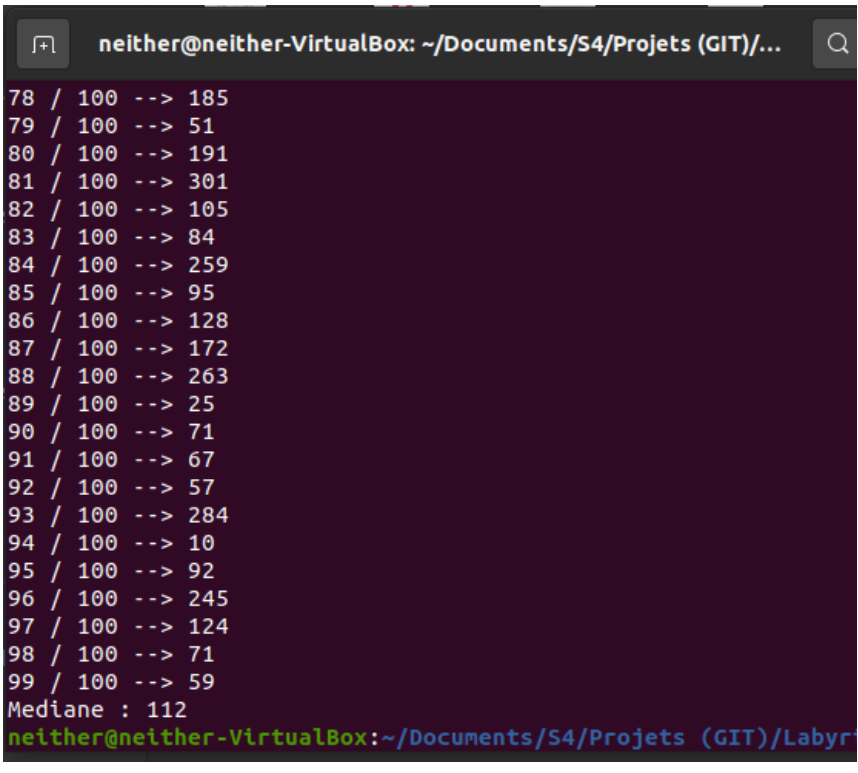
Comme on s'y attend : Si le prédateur chasse la proie avec un semblant de méthode tandis que la proie se déplace de manière aléatoire (donc en moyenne ne bouge pas beaucoup de son point de départ), la proie se retrouve vite rattrapée. On peut voir que sur 100 essais, ici la médiane est de 22.

Piètre performance, donc.



DIRECT PREDATEUR VS PROIE FUYANTE

L'algorithme proie ici semble déjà être plus intelligent. Pour récolter de meilleures données, faisons cette fois-ci des tests durant 300 tours. C'est un peu plus long mais l'attente en vaut la chandelle : La médiane atteint cette fois-ci 112 sur un essai. C'est significativement supérieur!



Cela s'explique au moins de 2 façons.

Le robot proie cherche toujours à s'éloigner du prédateur. La distance à parcourir pour ce dernier est donc plus grande.

Le robot proie et le robot prédateur entrent maintenant dans des *danses,* des cycles de durée indéterminée (quand ils doivent choisir entre 2 cases de façon équiprobable), cycles brisés uniquement par l'aléatoire. Cela arrange le robot proie car la durée de la course en est allongée.

Cependant, ce robot est très vite pris au piège dans des situations d'impasses. Quelques idées d'algorithmes l'améliorant seront proposées comme contribution personnelle!