UQAC
Université du Québec
à Chicoutimi

Blin Sébastien
Drouin Viallard Victor

Université du Quebec à Chicoutimi

Maîtrise professionnelle
Parcours Informatique

TP 1 8INF846

Réalisation d'un agent intelligent

Sous l'encadrement de :

Bouchard Kévin

1 Introduction

Le but de ce premier TP était de réaliser un agent intelligent. Cet agent prend la forme d'un aspirateur qui doit aspirer les pièces d'un manoir où de la poussière et des bijoux apparaissent de manière aléatoire. De plus, l'agent doit posséder un état mental de type BDI comme vu en cours. Dans ce rapport, nous verrons dans une première partie comment exécuter le programme, puis en second temps nous justifierons nos choix de modélisation.

2 Exécution du programme

2.1 Dépendances du projet

Ce devoir a été développé en *C++14* et a été testé sur des environnements *Linux* (mais fonctionne normalement sur Windows). Pour compiler le programme il vous faut :

- *G++* (version 6 recommandée)
- La librairie *SFML*
- *Make*

2.2 Compilation

Pour compiler le programme, il suffit d'exécuter la commande *make*. Un binaire nommé *agentTP1* sera alors présent.

2.3 Configuration et exécution

Le dossier des sources possède 2 types de fichiers de configuration. Le premier permet de décrire la carte dans laquelle l'agent va évoluer. Voici un exemple de configuration correspondant à la carte du sujet du TP1 :

```
11100
11111
11100
```

Deux exemplaires de fichiers de configuration de carte sont présents et nommés mphmap et mphmap2.

Le second type de fichier configure l'agent. On peut lui donner le nom de la stratégie souhaitée (pour le moment deux stratégies ont été implémentées, *SuckWithLevel* et *State*) ainsi que sa position initiale sur la carte (cette position sera aussi la position de sa base de recharge) :

```
##############################################################################
#                                VACUUM PARAMETERS                           #
##############################################################################
```

```
# Strategy for the agent
# {SuckWithLevel}
# {State}
strategy=SuckWithLevel
base=0;0
# 0 = nothing, 1 = Choosed Action + Status, 2 = 1 + States process, 3 = 2 + internal map
log=4
speed=1
```

Le paramètre log change le niveau d'affichage d'informations dans la console. Le paramètre speed modifie la vitesse d'éxécution, qui est aussi modifiable en cours d'éxécution avec les touches p et m. Deux exemples de tels fichiers sont présents et nommés mphvacuum et mphvacuum2.

Une fois ces deux fichiers créés, il suffit d'exécuter le programme :

```
./agentTP1 map vacuum
```

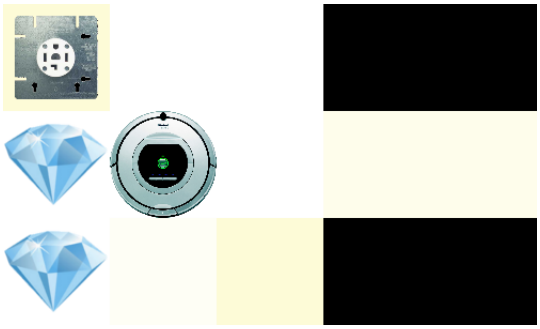

Figure 1: Le programme en cours d'exécution

3 Structure du programme

3.1 Structure générale

Comme demandé dans le sujet, notre programme s'exécute sur deux threads principaux. Le premier thread réalise toute la gestion de la carte en créant de la poussière et des bijoux. Le second thread correspond à l'agent en observant l'environnement visible, puis donne le résultat à une stratégie qui en déduit la prochaine action et l'agent l'exécute. Un troisième thread est présent et gère l'affichage des graphique des données.

Les fichiers intéressants pour la création et le lancement des threads sont *main.cpp*, *Loader.** et *Startable.**.

3.2 L'environnement

La partie environnement est implémentée dans les fichier *Map.**. La carte est représentée par un vecteur de vecteur de cases (*Case.h*) et est mise à jour dans la fonction

```
void Map::update(double delta);
```

qui consiste en une boucle où un évènement à réalisé est choisi (soit un ajout de bijoux avec une probabilité de 0.2, soit un ajout de poussière avec une probabilité de 0.8) sur une case tirée au hasard. Deux évènements sont espacés d'un temps aléatoire (entre 100 et 1000 millisecondes).

3.3 L'aspirateur

L'agent reprend le fonctionnement donné en pseudo-code dans le sujet du TP, c'est-à-dire :

```
While (amIAlive()){
    ObserveEnvironmentWithAllMySensors();
    UpdateMyState();
    ChooseAnAction();
    justDoIt();
}
```

Tout comme *Map*, on peut trouver la fonction :

```
void Vacuum::update(double delta);
```

puisque les deux classes héritent de *Startable*. Cette fois la méthode observera l'environnement en mettant à jour l'état observé via la méthode :

```
Sensors Vacuum::observe();
```

L'objet *Sensors* est décrit dans *Sensors.h* et contient le niveau de poussière et le nombre de bijoux de la case actuelle, ainsi que la possibilité de se déplacer dans les cases voisines. Cette captation est passée en paramètres à la stratégie pour l'empêcher de tricher en modifiant directement l'état de l'environnement. Elle s'occupera de mettre à jour l'état interne et dedonner la prochaine action à exécuter (donc *UpdateMyState()* et *ChooseAnAction()* qui seront expliquées plus bas). Enfin, l'agent exécute l'Action retournée par la stratégie via la fonction :

```
void Vacuum::executeCurrentAction(double delta);
```

3.4 Les stratégies implémentées

Deux stratégies ont été implémentées pour ce TP. Une première (*SuckWithLevelStrategy*) tente d'évaluer la meilleure action possible et ne possède pas d'algorithme de déplacement (et donc ne fonctionne pas bien sur une carte avec

des obstacles complexes, comme peut le montrer un test sur le fichier *map2*). Une seconde (*StateStrategy*) fonctionne à l'aide d'une machine à état et d'un algorithme pour planifier ses trajectoires de façon à fonctionner même sur les cartes avec obstacles complexes.

3.4.1 SuckWithLevelStrategy

L'intelligence de l'agent est implémentée dans une stratégie. Nous avons modélisé un agent basé sur les buts avec un état mental de type BDI dans les fichiers *SuckWithLevelStrategy.**. Les croyances de l'agent sont représentées par une map interne qui donne ce que sais l'agent à un moment T. Cette carte est mise à jour avant chaque décision via la fonction :

```
void SuckWithLevelStrategy::updateInternalMap(const Sensors& sensors);
```

et est représentée via un

```
std::deque<std::deque<StrCase>>
```

(plus facile à transformer lors de la découverte de la carte par l'agent). Une *StrCase* est une Case avec l'attribut :

```
std::time_t lastVisit;
```

Le désir de l'agent est de rendre la carte la plus propre possible en évitant d'aspirer les bijoux (mais en les ramassant), tout en évitant de tomber en panne. Pour modéliser ces choix, une fonction s'occupe de donner un score à chaque action possible. Le but de l'agent est de maximiser son score. Cette prise de décision est modélisée dans la fonction :

```
Action SuckWithLevelStrategy::findNextAction(const Sensors& sensors);
```

Ainsi, l'action retournée correspond à son intention. Et on peut voir 4 intentions différentes :

- Nettoyer la poussière de la case quand l'agent pense qu'il y a assez de poussière
- Ramasser les bijoux, dans ces 2 premiers cas le désir et l'intention sont identiques
- Se déplacer vers la case qu'on a visité il y a le plus longtemps (ou il y a des chances d'avoir assez de poussière maintenant)
- Se déplacer vers la case d'origine ou on peut se recharger. Dans ces deux derniers cas, I=la prochaine case visitée, D=Le chemin à parcourir pour se rendre à cette case. Bien sûr, sur la route, l'agent peut décider d'aspirer/ramasser les bijoux s'il le juge utile.

3.4.2 StateStrategy

Tout comme la précédente stratégie, *StateStrategy* possède une fonction pour mettre à jour sa carte interne représentée par une *std::deque¡std::deque¡StrCase¿¿* :

```
void updateInternalMap(const Sensors& sensors);
```

La principale différence vient du choix d'action qui est donné par une machine à état (visible dans *StateStrategyStates.**). Tout comme la première stratégie on peut voir 4 intentions différentes représentées par les 4 états implémentés. Lorsque la stratégie doit calculer l'action à réaliser, elle demande à son état courant. Si son état retourne une action valide (autre que None) alors c'est l'action à effectuer. Sinon (l'action retournée est None) alors c'est que la stratégie a changé d'état et il faut demander au nouvel état l'action à réaliser. La seconde différence vient de la planification de la trajectoire. En effet, la première stratégie n'a pas d'algorithme de déplacement, mais se contente de se rapprocher de sa case objectif. Si un obstacle est présent, le robot risque de rester dans le même coin. Ici, la planification de la trajectoire est implémentée à l'aide d'un algorithme $A*$ dans les méthodess :

```
std::vector<ActionType> StateStrategy::pathTo(const Pos& target);
std::vector<ActionType> StateStrategy::constructpath(...);
```

En outre, cette stratégie aspire les cases pas à pas (restant au maximum 0.2s par aspiration) pour vérifier de manière régulière si un bijou ne serait pas tombé sur la case actuelle depuis la dernière observation de l'environnement.

4 Conclusion

Au final, ce projet nous a permis d'implémenter plusieurs fonctionnements décrits en cours. Tout d'abord le concept d'agent ou nous avons dû réfléchir à quel type d'agent notre aspirateur correspondait. Ensuite, le principe d'état mental BDI. Enfin, comment implémenter la prise de décision à l'aide d'un scoring des différentes actions, ou d'une machine à états finie.