Collecte de trésor multi-agents

Comlan Amouwotor Lin Tianyuan

February 2024

1 Introduction

Pour résoudre le probleme de collecte de trésor posé, nous proposons un programme Python se basant sur le classes de bases données et doté d'une interface graphique qui facilite l'experimentation de notre approche de solution sur un grand nombre d'instances du probleme. Il est possible d'accélérer ou de ralentir l'execution, redemarrer, generer de nouvelles instances et mettre en pause l'exection a tout instant pour examiner les comportements des differents éléments de plus près.

Le code du fichier Environment.py n'a pas été modifié et se trouve dans le repertoire principal du projet.

Ce rapport presente brièvement les algorithmes utilisés et indique les parties du code (bien documenté) qui les implémentent.

2 Le code

2.1 Prérequis

Le code requiert l'installation de la version `2.5.2` du module `pygame` qui a servit à développer l'interface graphique.

Ceci peut etre fait automatiquement en exécutant le script `install-requirements.sh` depuis le repertoire principal du projet.

```
bash install-requirements.sh
```

2.2 Execution

La manière la plus simple d'exécuter le programme est par la commande suivante, à lancer depuis le repertoire principal du projet:

```
python main.py
```

L'exécution de cette commande lance le programme en le laissant générer automatiquement un fichier d'environnement (dans le repertoire `./environments/`) puis lancer l'interface graphique du programme qui affichera l'état initial de l'environnement.

Pour spécifier un fichier d'environnement personalisé, il suffit de lancer programme avec la commande:

```
python main.py env=path/to/env###.txt
```

Ensuite, il faudra appuyer sur le bouton jaune `"go"` visible en bas à gauche de la fenêtre du programme.

2.3 L'interface graphique

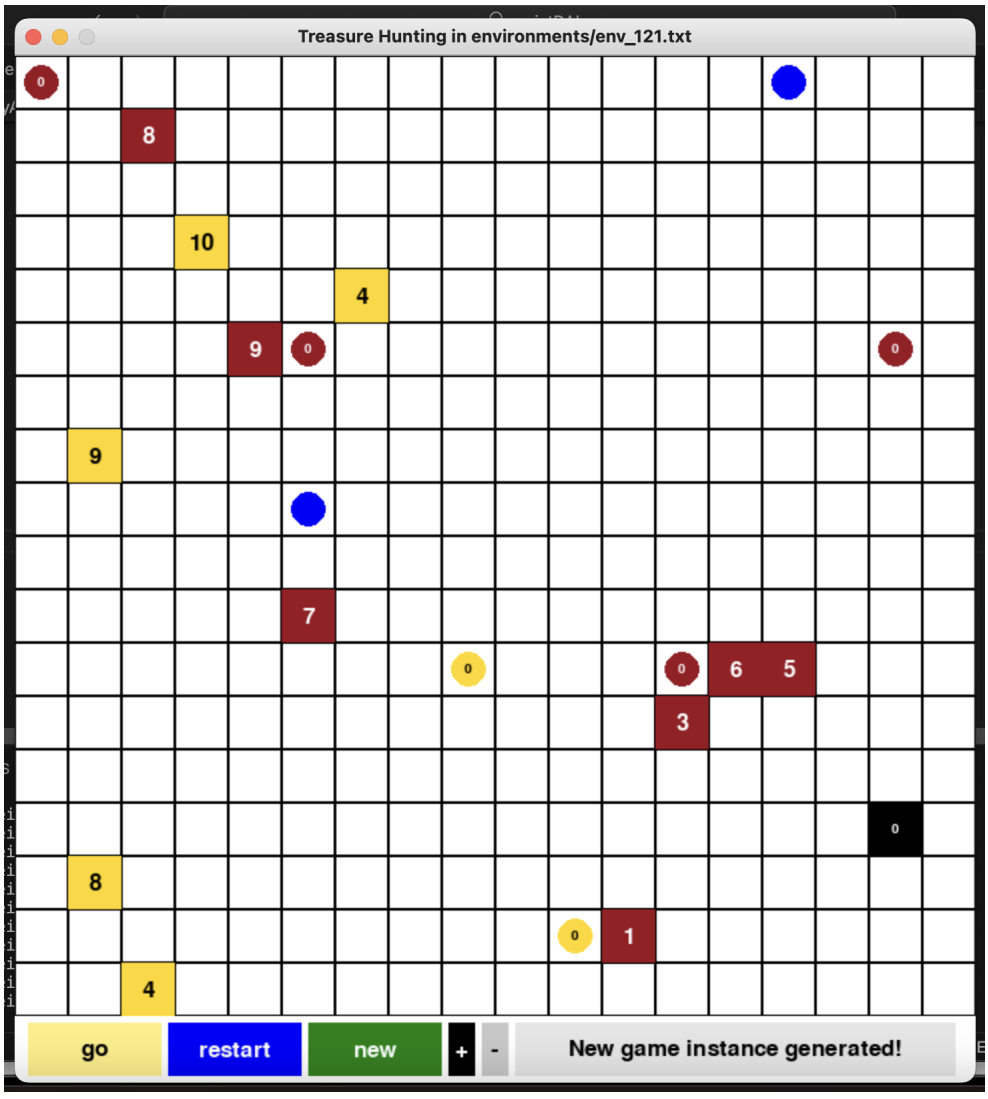

Figure 1: Exemple d'une instance du problème

Dans cette section nous expliquons les choix graphiques que nous avons faits:

- sur la grille, les disques sont des agents tandis que les carrés jaunes et rouges sont des coffres. Le carré noir est la zone de dépôt.
- les disques jaunes et rouges collectent respectivement de l'or et des pierres tandis que les disques bleus sont des agents ouvreurs
- une fois qu'un coffre a été ouvert sa couleur devient plus claire (jaune clair pour les coffres d'or et rose pour les coffres de pierres précieuses).
- les nombres sur les coffres (carrés) représentent leur valeur tandis que les nombres sur les agents ramasseurs représentent la quantité de trésor dans leur sac à dos

En dessous de la grille nous avons une série de boutons:

- le bouton "go/pause" permet de mettre le jeu en marche ou en pause
- le bouton "restart" redémarre l'instance actuelle du jeu
- le bouton "new" génère une nouvelle instance du problème
- les bouton "+" et "-" permettent d'accelerer et ralentir le jeu
- une zone en bas à l'extrême droite affiche des notifications

3 Protocoles et algorithmes

Chaque instance du problème se résoud en suivant plusieurs étapes à savoir:

- l'allocation des tâches: elle se fait par le plusieurs encheres de vicheres, dont chacune se conclut par l'allocation d'un coffre à ouvrir ou à collecter au meilleur enchérisseur
- la planification: le squelette du plan local de chaque agent se construit en meme temps que se fait l'allocation des coffres; son raffinement se fait à chaque temps d'exécution en fonction des echecs (echec d'une tentative de mouvement, echec d'une collecte de tresor)
- la coordination par des regles sociales: des protocoles de mouvemements aléatoires sont établis pour garantir la résolution des situations d'embouteillages ou d'echecs de collecte de trésors que peuvent rencontrer les agents.

Plus de détails dans les paragraphes suivants.

3.1 L'allocation des tâches

L'allocation des tâches se fait en 3 séries d'enchères de Vichère séparées, une série d'enchères de Vichères pour les ouvreurs de coffres, une autre séries pour les ramasseurs d'or et enfin une autre pour les ramasseurs de pierres. Chacune de ces enchères est organisée par l'objet unique de la classe `PlanManager` dans la méthode `tasks_allocation()` dont le corps est le suivant:

```
def tasks_allocation(self):
    # Let the manager organize the auctions
    # For chest openers
    ChestOpenersAuction(self.env).go()
    # For the gold chest collectors
    RamasseursAuction(self.env, agent_type=0).go()
    # For the stones chests collectors
    RamasseursAuction(self.env, agent_type=1).go()
```

3.1.1 Allocation des coffres aux agents ouvreurs

Le rôle des agents ouvreurs étant nécessaire à l'exécution des tâches des agents ramasseurs, l'allocation des coffres se fait dans le but d'ouvrir tous les coffres aussi vite que possible. L'allocation se fait comme suit (méthode `allocate()` de la classe `Auction`):

- A chaque tour d'enchère, chaque agent renvoie à l'instance `PlanManager`, son coffre préféré (le coffre ayant la meilleure evaluation locale pour cet agent) ainsi que son evaluation pour ce coffre. Cette evaluation tient compte de la distance entre l'agent lui-meme et le coffre, de la liste des coffres qui lui ont été alloués depuis le début de la série d'enchères.
- L'instance `PlanManager` sélectionne l'agent ayant la meilleure evaluation (comparé aux évaluations des autres agents participants) pour son coffre préféré et lui alloue ce coffre.
- L'agent ouvreur gagnant de cette enchère reçoit un mail de la part de l'allocateur (l'objet PlanManager) lui spécifiant le coffre qui lui a été alloué. L'agent ou met à jour ses champs `evaluating_dist_so_far` (le cumul des distances qu'il lui faudrait parcourir pour ouvrir les coffres qui lui ont déjà été alloués) et `evaluating_from` (la position sur la grille du dernier coffre qui lui est alloué) afin que sa prochaine évaluation pendant la prochaine enchère tienne compte des tâches qui lui ont été allouées.
- Le coffre alloué est retiré des prochaines enchères. La prochaine enchère suit le meme protocole sur les coffres restants.
- La dernière enchère alloue le dernier coffre à celui qui lui a donné la meilleure evaluation.

Étant donné la nature imprévisible de l'environnement d'execution et pour rendre le programme plus dynamique, l'ordre de parcours des coffres à èvaluer pendant chaque vote est rendu aléatoire. Cette composante aléatoire ne sert qu'à permettre à chaque agent de choisir aléatoirement entre deux options équivalentes (deux coffres à la meme distance et de meme valeur). L'evaluation des coffres disponibles par chaque agent pendant les enchères se fait avec la méthode `evaluate()` de la classe `MyOwnAgentChest`.
Une propriété intéressante de cette methode d'allocation est qu'elle assure une répartition optimale des taches (en ne tenant compte que des facteurs prévisibles avant l'execution).

3.1.2 Allocation des coffres aux agents collecteurs

L'allocation des coffres aux agents ramasseurs se fait d'une manière similaire au cas des agents ouvreurs, par la methode allocate() de la classe `Auction`. La seule difference se trouve dans le protocole d'evaluation des coffres disponibles par chaque agent.
Ce nouveau protocole est rendu légèrement plus compliqué par le fait qu'on doive tenir compte de la capacité des backpack et de la necessité de vider les sacs à dos lorsqu'ils sont pleins. La methode utilisée pour cette evaluation est la methode `evaluate()` de la classe `MyAgent`. Les nouvelles particularités de la methode d'evaluation qui permet aux agents ramasseurs (collecteurs) de faire leur vote sont:

- l'evaluation des coffres tient compte non seulement de la distance (calculee par la methode `MyAgent.distance(from, to)`) mais aussi de la capacité du backpack restant si l'agent executait les taches qui lui ont deja été affectées.

- l'objectif de la solution etant avant tout de collecter le maximum possible de tresor, l'evaluation des coffres dont la charge depasse celle du backpack de l'agent est faite de façon à permettre à un autre agent éventuel dont la capacité serait plus grande de s'en charger. L'algorithme garantit que ces coffres-là (dont la charge dépasse toutes les capacités de backpack) sont alloués seulement aux agents qui peuvent en prendre le maximum (la distance ne compte plus dans ces cas-là).

Cette methode d'allocation des tâches optimise le temps d'execution ainsi que la charge totale collectée. Des experimentations faites sur des milliers d'instances de ce probleme permettent de conclure qu'elle fonctionne comme souhaité.

3.2 La planification locale

A l'issue de la phase d'allocation des coffres aux agents, chaque agent dispose d'un plan local abstrait qui est la liste ordonnée des positions des coffres qui lui ont été alloués. A l'execution, ce plan abstrait est raffiné afin de resoudre les situations d'échecs inattendues qui peuvent survenir. La methode qui se charge

du calcul de la prochaine action à executer par l'agent est la methode `act()` du fichier `MyAgent.py`.
Par cette methode, si le plan local de l'agent n'est pas accompli, alors il:

- évalue la destination du prochain `move()` à exécuter pour rapprocher du prochain coffre ou de la zone de dépôt
- l'agent execute cette action et verifie si l'action a réussi en examinant le code de retour. En cas de succès, il n'y a rien à faire pour l'instant. Sinon il execute un aleatoire qui pourrait eventuellement le tirer de la situation de conflit et attend le prochain instant pour poursuivre son plan local
- si l'agent est arrivé à l'emplacement de son prochain coffre il charge son son sac. S'il s'agit de la zone de dépôt, alors il décharge son sac.

Si le plan local de l'agent est accompli et qu'il a dechargé tous les trésors qu'il a collectés, alors la methode `act()` fait deplacer l'agent sur les bords de la grille afin d'eviter de perturber les mouvements des agents qui ont encore des tâches à accomplir. Il s'agit là d'une des deux règles sociales qui ont améliorent le temps d'execution de la tâche collective.

3.3 La coordination par des règles sociales

Nous avons déterminé après beaucoup d'expérimentations que l'une des methodes qui s'adapte le mieux à une mise à l'échelle du problème est d'établir des règles sociales adaptées qui ne necessitent pas de communication entres les agents concernés (ce nombre pouvant être très grand).
La premiere règle sociale est l'execution d'un mouvement aléatoire autorisé lorsque le mouvement souhaité echoue. Cette règle est simple et efficace pour régler les situations d'echec de mouvement et de collecte de trésor. Elle permet également de résoudre les situations d'embouteillage autour de la zone de dépôt. Dans les instances du problème ayant un grand nombre d'agents, ces embouteillages sont difficiles à résoudre par une négociation directe entre agents.
La deuxieme règle sociale est le mouvement des agents qui ont terminé l'exécution de leur plan local. Le but ici est de limiter les échecs de mouvements pour les agents encore occupés. Ces agents libres exécutent une routine de mouvement sur les bords de la grille est changent de direction dès qu'ils rencontrent un obstacle.
L'execution de notre code sur un grand nombre d'instances du probleme permet de conclure que ces règles sociales sont efficaces.

4 Conclusion

Nos protocoles et algorithmes ont été expérimentés sur des centaines d'instances (faciles à générer grâce au bouton "new") de l'interface graphique.