Caillaud Tanguy

Rapport TP1 Systèmes Multi-Agents

Pour ce TP j'ai considéré que les blocs-agents n'avait en perception locale que la possibilité de voir le bloc de dessous. Les actions possibles sont quant à elle celles énoncées dans le document.

Partie 1 :

Pour cette partie j'ai donc commencé par l'implémentation du système. Pour cela j'ai créé deux classes, "Environnement" et "Agent". Dans la première, celle-ci a en argument une liste de pile d'Agent qui correspond aux 3 parties de la table sur laquelle les blocs peuvent être posés et une liste d'Agent qui est la liste des agents qui seront présents dans l'environnement. L'Agent quant à lui possède en argument son environnement un tag pour lui donner un nom et le tag de l'Agent sur lequel il doit se poser pour obtenir la bonne solution. La classe possède aussi les fonctions suivantes :

- ***action()*** : fonction principale dans laquelle l'Agent dans laquelle il peut appeler toutes ses actions locales en fonction de ce qu'il perçoit localement.
- ***perception()*** : fonction retournant l'identité du bloc du dessous par le biais de l'environnement.
- ***move()*** : fonction avec laquelle l'agent se déplace dans l'environnement.
- ***push()*** : fonction poussant l'agent de dessus dans l'environnement.
- ***moveOrPush()*** : fonction appelant les fonctions ***push()*** ou ***move()*** selon si l'Agent peut bouger dans l'environnement ou non.

Le déroulé du fonctionnement du système est alors le suivant, l'environnement est créé et on lui associe les 4 Agents. On lance alors ensuite les actions d'Agents tour par tour en commençant par le "A". Le A utilise donc sa fonction "action()' dans laquelle il vérifie d'abord avec perception() si il n'est pas déjà sur le bon bloc si c'est le cas il ne fait rien passe son tour, sinon il lance moveOrPush() et bouge dans l'environnement si le bloc est libre ou pousse dans le cas inverse. En poussant avec push() le bloc du dessus lance alors lui aussi moveOrPush() et cela jusqu'à ce qu'un bloc bouge dans l'environnement. Lorsque l'on bouge dans l'environnement au travers de sa fonction move(), l'agent est placé aléatoirement dans un des Stack dans lequel il n'était pas. A la fin d'une action, la fonction verify() est lancée, celle-ci demande alors à chaque agent si il est sur sa bonne case (il le vérifie avec perception) et si c'est le cas le système a atteint son objectif et s'arrête alors. Ce système basant sa méthode de déplacement sur l'aléatoire peut plus ou moins boucler et effectuer beaucoup d'étapes redondantes et inutiles. En moyenne j'observe entre 20 et 80 actions à effectuer par le système pour résoudre le problème.

Partie 2 :

Caillaud Tanguy

Pour cette partie les agents peuvent communiquer, cependant ils ne peuvent toujours percevoir que le bloc sous eux. Ils possèdent aussi tous les mêmes capacités, il n'y a pas d'agent central décidant pour d'autres, la communication servira juste à laisser des priorités si un agent à plus besoin de bouger qu'un autre. Enfin comme ils peuvent communiquer et que chacun sait sur qui il doit se positionner, en partageant ces informations les agents pourraient reconstruire logiquement cette solution en mémoire mais comme de base ils n'ont pas cette solution qui leur est communiquée j'ai préféré partir du principe qu'il n'avait pas la capacité de stocker ou de générer logiquement cette solution. Ainsi la seule information de plus découlant de la communication que les agents pourront stocker en mémoire est le tag d'un autre agent qui correspondra à l'étape à effectuer en priorité.

Dans cette deuxième partie ayant pour but de faire coopérer les Agents entre eux j'ai créé une nouvelle classe héritant de Agent, AgentCommuniquant. En effet, la stratégie que j'ai choisie est la communication inter agents permettant à ceux-là de s'échanger les informations qu'ils possédaient localement dans le but d'établir des priorités dans la réalisation des actions des agents. Pour les agents se communiquent notamment l'étape courante à effectuer pour parvenir à la solution et cela au travers de l'argument currentStep. Ces arguments commencent alors pour tous les robots à "Table" car la première étape à effectuer est toujours de bien placer le bloc étant à la base de la pile, donc le bloc touchant la table. Ensuite les AgentCommuniquant se servent de ces fonctions pour évoluer dans le système :

- communicateNewStep() : cette fonction lancée lorsque l'Agent perçoit qu'il vient d'arriver sur le bon emplacement et que celui-ci correspond aussi à l'étape en cours. Cela implique que les agents peuvent passer à la prochaine étape et donc l'agent le communique à tout le monde. Étant bien placé, la prochaine étape correspond alors à se positionner sur l'agent même envoyant le signal et donc l'agent envoie son tag aux autres.
- askIfTargetBlocked() : cette fonction est une fonction de communication dans laquelle l'agent demande aux autres agents s' ils ne sont pas sur l'agent de l'étape courante. Si c'est le cas l'étape n'est pas valable en l'état et alors l'agent ayant perçu qu'il était sur le bloc de l'étape à valider bouge.
- askAgent() : cette fonction de communication demande aux autres agents si leur bloc objectif est le bloc étape. Si un agent répond positivement il passe alors à l'action.

Avec ses nouvelles fonctions le système fonctionne alors de la manière suivante, cette fois-ci pour chaque action (avec la nouvelle fonction action()) l'agent commence d'abord par regarder si il n'est pas sur son bloc objectif et sur le bloc étape, si c'est le cas il communique alors immédiatement sur la nouvelle étape avec communicateNewStep().

Ensuite l'agent vérifie si le bloc-étape n'est pas bloqué, c'est-à-dire si un bloc indésirable n'est pas au-dessus de lui, d'abord l'agent regarde alors par lui-même s'il ne bloque pas le bloc-objectif avec perception(), si oui alors moveOrPush() est lancé pour débloquer la situation. Si lui ne bloque pas il communique ensuite avec les autres agents pour leur demander si eux bloque l'objectif (il vérifie de la même manière avec perception()) et ainsi lorsque c'est le cas d'un agent celui-ci est alors priorisé et devient l'agent en action en utilisant son action().

Si il n’y a pas d’agent bloquant l’agent en action regarde si son objectif est le bloc-étape en cours, si oui il moveOrPush() et après cela il vérifie si il n’est pas sur son bloc-objectif et qu’il vient donc de finir une étape, si c’est le cas il lance communicateNewStep(). Sinon si l’agent en action n’est pas l’agent ayant pour objectif l’étape en cours, il communique avec les autres agents et demande cette fois-ci si un agent a pour cible le bloc-étape en cours, si c’est le cas cet agent est priorisé et son action() est donc lancée.

Les AgentsCommuniquant possèdent aussi une nouvelle fonction move() dans laquelle ils vérifient après avoir bouger dans l’environnement s'ils ne se sont pas placés sur le bloc de l’étape courante à l’aide de perception(). Si c’est le cas et que le bloc étape n’est pas leur bloc objectif, il rebouge en veillant à ne pas retourner sur la pile initiale sur laquelle il était.

Avec ces principes de communication et de priorités d’action le nombre d’action varie désormais en moyenne entre 5 et 10 actions, on voit alors que le système est plus optimisé, cependant il pourrait facilement l’être encore plus, si par exemple les agents connaissaient directement toute la solution et non pas une simple étape ou bien si un agent central pouvait décider pour tout le monde avec une perception globale mais je trouvais le système tel qu’il est actuellement plus intéressant à implémenter car les agents possèdent un minimum de capacités et de stockage et pourtant le système est tout de même correct niveau optimisation.