Mercredi 9 Janvier 2008

**Programmation : Rapportons les $\lambda$-termes sur nos chameaux**

Vincent Delaitre
et
Anne-Laure Mouly

# 1 Plus_naïf_tu_meurs

Il était une fois deux élèves dans une école fort lointaine du nom de ENS Lyon. Ils décidèrent d'aider le peuple des chameaux à sauver de pauvres $\lambda$-termes en transe paranoïaque. Pour cela, avec l'aide des savants et généreux ($\lambda$x.x-men) TD, ils programmèrent un chameau au doux nom de Plus-naïf-tu-meurs. Laissez-moi vous conter son histoire.

Plus-naïf-tu-meurs déambule dans un monde étrange qui fut décrit dans le roman Types.mli. Dans ce monde on peut trouver :

- Notre chameau héros
- Les méchants chameaux concurrents qui menacent de sauver les $\lambda$-termes avant Plus-naïf-tu-meurs et d'ainsi ravir la main de la belle princesse chamelle
- Une carte magique, grâce à laquelle notre chameau pourra visualiser son avancée ainsi que celle des autres chameaux
- Une liste des bases où trouver des $\lambda$-termes à sauver
- Une table de hachage qui mutile les pauvres victimes $\lambda$-terme.

Plus-naïf-tu-meurs $\lambda$-termes naquit dans main.ml. Sa mère, nommée Serveur, lui a donné toutes les informations dont il a besoin pour se débrouiller dans ce monde hostile.

Nous l'avons en plus doté d'une intelligence prodige provenant de IA.ml. Nous avons dit "Lève-toi et marche vers l'Est dès que tu le pourras. Si vers l'Est avancer tu ne peux, vers le Nord tu iras et si vers le Nord avancer tu ne peux, vers le Sud tu iras" et le Plus-naïf-tu-meurs avança en sauvant les $\lambda$-termes qui passaient à sa portée.

En bonnes aides pour notre chameau nous avons aussi scruter le monde afin d'en extraire les informations nécessaires à notre chameau : positions de $\lambda$-termes à sauver, des méchants concurrents, des trous-noirs, manière d'agir des méchants chameaux... Cet opération fut de loin la plus lourde car Plus-naif-tu-meurs est très avide de connaissances. Mais sans il n'eut pu commencer son épopée.

Plus-naïf-tu-meurs avança droit vers sa destinée et fut malheureusement bloqué très rapidement en laissant nos victimes à leur triste sort. En expirant sa dernière bouffée d'air, il nous donna le souffle nécessaire à confier sa tache à un autre chameau qu'on espère plus valeureux.

# 2 Sauvons les $\lambda$-termes

Comme vous en avez marre de ce style pompeux et ennuyant, et puis que de toute façon ça devient sérieux, voilà la suite en français. Il s'agit cette fois de réussir à sauver les $\lambda$-termes en imaginant que nous sommes le seul chameau au monde (il s'appelle Will Smith - le chameau, pas le monde).

## 2.1 Différentes stratégies

### 2.1.1 Rechercher un chemin

Pour aller d'un endroit à un autre, nous avons implémenté les algorithmes suivant (IA_sauve_chemin) :

- un algorithme simple : Will Smith va droit vers l'objectif en coutournant les éventuels obstacles
- Dijkstra
- $A^*$

Pour les deux derniers, nous avons eu besoin de manipuler des tas. Nous avons choisi d'utiliser les tas de Fibonacci pour plus de rapidité (qu'il dit le wiki). Dijkstra donne le chemin optimal mais est beaucoup plus lent (cf. 2.2) que $A^*$. Malgré le fait que ce soit un algorithme approximatif, nous avons donc gardé $A^*$. Nous avons voulu améliorer le chemin en regardant s'il est intéressant de le modifier si on passe proche de choses intéressantes. Les résultats expérimentaux (cf. 2.2) ont montré que ce n'était pas utile, voire que ça gênait notre chameau.

Le chemin renvoyé est une liste de positions. Ce n'était pas très pratique dans beaucoup de cas, il a fallu jongler entre position et direction mais le gros avantage a été de pouvoir vérifier la validité du chemin. En particulier, en mode multi-joueurs le chemin peut être erroné si on a été poussé ou si on a changé de direction. On peut alors le détecter et créer un nouveau chemin vers l'objectif de l'ancien chemin.

### 2.1.2 Trouver des $\lambda$-termes

Le chameau doit choisir vers quelle base (ou pas base) s'approvisionner lorsqu'il n'a plus de $\lambda$-termes à sauver (IA_sauve_livre deuxième partie). Il peut alors :

- choisir la base la plus proche, si les bases sont vides il va vers le $\lambda$-terme posé par terre le plus proche et s'il y en a pas il traque un chameau portant des $\lambda$-termes pour lui en piquer un
- choisir la base la plus "intéressante", la notion d'intéressant étant quantifiée par le ratio moyen des $\lambda$-termes de la base, sa distance et le nombre de $\lambda$-termes qu'elle contient

On a sélectionné le deuxième algorithme qui équilibre mieux les bases. En particulier, si la seule autre base que celle où il est est très loin, il préfère écumer celle proche de lui d'abord. Ces algorithmes nécessitent de bien mettre à jour toutes les données, en particulier dès qu'un chameau prend ou dépose un $\lambda$-terme, pour gérer les cas ou les bases sont vides mais où il reste encore des $\lambda$-termes.

### 2.1.3 Sélectionner les $\lambda$-termes

Il s'agit ici de déterminer quels $\lambda$-termes prendre lorsqu'on arrive sur une base. Les algorithmes proposés (IA_sauve_ramasse) sont :

1. ramasser les $\lambda$-termes dans l'ordre où ils apparaissent
2. ramasser les $\lambda$-termes dans l'ordre décroissant de taille
3. ramasser les $\lambda$-termes dans l'ordre décroissant du ratio taille sur distance à la base
4. idem mais cette fois en ne prenant si possible que des $\lambda$-termes dont le ratio est supérieur à une certaine valeur. Si ce n'est pas possible alors regarder les $\lambda$-termes avec un plus faible ratio
5. idem mais cette fois on calcule le ratio avec la distance au dernier $\lambda$-terme choisi

Les deux premiers sont très peu efficaces, le troisième l'est plus mais on s'est rendu compte que c'était parfois une perte de temps que de prendre des $\lambda$-termes peu rentables. Ce sera encore plus utile en mode multi-joueurs. Par contre, calculer le ratio en fonction de la distance à la base n'est pas nécessairement judicieux vu qu'on ne va pas faire d'aller-retours mais plutôt ules livrer les uns à la suite des autres, d'où la naissance du cinquième algorithme. Toutefois, et contre toute attente, cette fonction donne de moins bon résultats expérimentaux. On ne l'utilisera donc pas (mince, tout ça pour rien !).

Dans tous les cas, on ne ramasse pas de $\lambda$-terme inatteignable. Pour cela, on a remplacé au début toutes les cases que l'on ne pouvait pas atteindre par des palmiers. Si l'objectif d'un $\lambda$-terme est un palmier, on ne le prends pas.

### 2.1.4 Ordre de livraison

Une fois des $\lambda$-termes récupérés, il faut choisir dans quel ordre les livrer. Là encore plusieurs algorithmes ont été implémentés :

- déposer le premier de la liste
- déposer le plus près de l'endroit où on se trouve
- déposer celui qui a le meilleur ratio taille/distance

C'est bien entendu le troisième algorithme qui a été choisi.

## 2.2 Résultats expérimentaux

Voilà les résultats expérimentaux obtenus en fonction des différents stratégies.

| IA_sauve_chemin.ml | | IA_sauve_ramasse.ml | IA_sauve_livre.ml | | |
|---|---|---|---|---|---|
| trouve_chemin | ameliore_chemin | ramasse_lterm | calcule_livraison | calcule_chargement | score |
| Dijkstra | stupide | stupide | stupide | plus_proche | 337 |
| Dijkstra | stupide | moins_stupide | stupide | plus_proche | 911 |
| Dijkstra | stupide | ratio | stupide | plus_proche | 1181 |
| Dijkstra | stupide | ratio_borne_inf | stupide | plus_proche | 2189 |
| Dijkstra | stupide | ratio_borne_inf | ratio | plus_proche | 2144 |
| Dijkstra | stupide | ratio_borne_inf | moins_stupide | plus_proche | 2278 |
| $A^*$ | stupide | ratio_borne_inf | moins_stupide | plus_proche | 2435 |
| Dijkstra | stupide | ratio_borne_inf | moins_stupide | meilleur_ratio | 2613 |
| $A^*$ | stupide | ratio_borne_inf | moins_stupide | meilleur_ratio | 2571 |
| $A^*$ | inteligent | ratio_borne_inf | moins_stupide | plus_proche | 2397 |

Quelques commentaires :

- Ce n'est pas flagrant içi mais $A^*$ est réellement plus performant que Dijkstra : sur une carte 1000*1000 traversée en diagonale, les ordres de grandeur sont les suivants :

| | Temps de calcul | Longueur du chemin |
|---|---|---|
| Dijkstra | 100 secondes | 2000 |
| A* | 0.3 seconde | 2300 |

  On choisit donc Dijkstra sur les petites cartes, et $A^*$ sur les grandes.

- Améliore_chemin_intelligent est mal conçue et prend de toute façon trop de temps de calcul pour être vraiment rentable.
- La réelle augmentation du score se fait en changeant la manière de ramasser les $\lambda$-termes. C'est un point crucial de l'efficacité du chameau.
- Il y a aussi possibilité de gagner du temps en calculant une tournée efficace pour déposer les $\lambda$-termes mais nous n'avons pas exploité assez cet aspect.

# 3 Des rivaux sans merci

Pour améliorer tout ça, et parce que sinon ce n'est pas drôle, nous avons aussi voulu faire évoluer notre chameau en mode multi-joueurs. La stratégie générale (IA) étant de modifier le chemin prévu en fonction de l'environnement chameauesque. En clair, si je suis dans un duel où je risque ma tête, je la sauve en misant gros et, si possible, en fuyant la situation. Sinon, je regarde s'il est intéressant d'attaquer et je propose une attaque. Si ce

n'est pas intéressant, j'évite de me mettre en danger avec l'action prévue, en particulier j'évite les zones où je peux me faire tuer. Ce n'est pas la peine de vérifier dans le cas où j'attaque car Attaque ne proposera jamais de bouger dans une zone dangereuse.

Une amélioration a été ajoutée à tout ça. En effet, des blocages entre chameaux intervenaient relativement souvent. On détecte donc dans main.ml d'éventuels blocages et lorsque l'on choisit quelle action faire, on débloque s'il y a blocage. En pratique, si on met deux chameaux lancés avec notre client, ils se débloquent puis se rebloquent quelques tours après mais ça devrait mieux se passer avec d'autres joueurs.

## 3.1 Défense

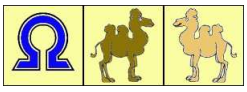
La première chose à faire est donc de protéger notre chameau d'un odieux meurtre (IA_defendre). La stratégie est relativement simple : si notre chameau est menacé de meurtre immédiat, il part dans la direction sure (ie sans danger) fuyant au mieux la situation. Pour cela, regarder uniquement les voisins de notre chameau suffit.
Dans le cas où une attaque n'est pas intéressante, je dois encore vérifier qu'aller où je veux n'est pas se jeter dans la gueule du chameau (plein de bave). Je regarde alors, mes voisins diagonaux pour regarder quelle direction serait la mieux.

En regardant ces différents cas, on ne protège pas notre chameau contre des coups arrivant au tour d'après, au mouvement des chameaux présents aux alentours. On utilise alors la fonction qui nous donne les alentours utilisée pour Attaque. On a ainsi directement accès aux cases dangereuses et on peut alors éviter d'y aller.

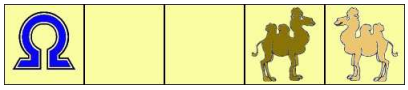
## 3.2 Attaque

Notre chameau ne cherche, en général, pas les problèmes. Disons qu'il ne va pas chercher un adversaire à l'autre bout de la carte pour lui faire des misères (sauf s'il n'y a plus de $\lambda$-termes sur la carte, auquel cas il peut devenir agressif). Cependant, il vaut mieux pour les autres ne pas se balader trop près quand même.
En effet la partie Attaque (IA_attaque) de notre chameau s'active lors qu'un adversaire est située à moins de 3 cases (à vol d'oiseau). Notre chameau essaye alors d'effectuer les actions suivantes dans l'ordre :

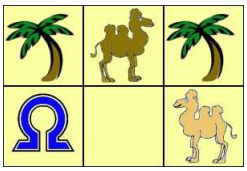
1. Si on peut pousser quelqu'un au contact dans un omega, on ne s'en prive pas (dans les illustrations ci-dessous, notre chameau est le gentil donc le clair, les méchants sont en sombre par définition du côté obscur de la force) :



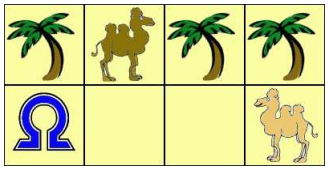
2. Si on peut pousser dans un omega en moins d'un certain nombre de coups (réglé à 4 actuellement), on tente notre chance :



3. Si on peut pousser quelqu'un qui n'est pas au contact mais qui va peut être y aller, on essaye :



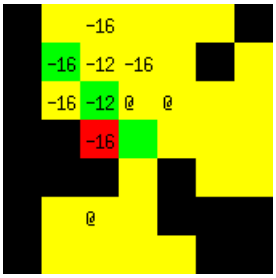
4. Si on pourra pousser quelqu'un au prochain tour, on y va :



5. Si Sauve préconise de ramasser des $\lambda$-termes, on regarde si ça vaut le coup : si oui, on les rammasse, si non, ceux qui ont programmé Sauve sont débiles.
6. Si un méchant au contact avec nous est sur une base : on le pousse (sauf s'il y a un palmier derrière lui : il ne bougerait pas), parce que les bases, c'est pour nous.
7. Si le gain est suffisant, on peut pousser un méchant pour récupérer ses $\lambda$-termes (sauf s'il y a un palmier derrière lui : on ne pourrait pas rammasser ce qu'il a perdu, ça serait balot). Pour calculer le gain potentiel, on procède de la manière suivante :
   - Soit $C_{totale}$ la capacité d'un chameau et $C_{libre}$ la place libre qu'il reste à notre chameau. Soit $T$ l'ensemble des $\lambda$-termes que porte le méchant. On note $|t|$ la taille d'un terme élément de $T$.
   - Soit $Somme_{vol}$ la somme des tailles des $\lambda$-termes sur le méchant et que l'on peut ramasser : $Somme_{vol} = \sum_{t \in T, |t| < C_{libre}} |t|$. Un terme dont la taille est inconnue compte pour [taille moyenne]*[proba qu'on puisse le prendre] dans le total, soit $[\frac{C_{totale}}{2}] * [\frac{C_{libre}}{C_{totale}}] = \frac{C_{libre}}{2}$

     Alors la taille moyenne du $\lambda$-terme que l'on va rammasser sera $vol = \frac{Somme_{vol}}{card(T)}$
   - On ajoute à $vol$ tous les $\lambda$-termes qui étaient déjà par terre et que l'on peut ramasser pour avoir $Somme_{totale}$. $Somme_{totale}$ ne peut donc pas dépasser $C_{libre}$.
   - Malheureusement, on doit passer un tour à ramasser les $\lambda$-termes et le méchant va nous pousser : on va perdre un $\lambda$-terme de valeur moyenne :
     $perte = \frac{C_{totale} - C_{libre} + Somme_{totale}}{\text{nombre de termes total (ramassés+anciens)}}$
   - Le gain moyen est alors : $gain = Somme_{totale} - perte$

Toutes les actions énumérées ci-dessus ne mettent pas notre chameau en danger, mais ne le sauve pas s'il est déjà en danger (d'où l'utilité du module Défense). Pour choisir efficacement qu'elle action on exécute, on calcule une représentation des altentours dans un tableau 7*7, centré sur notre chameau, dont je donne ci-dessous une représentation graphique :



La légende est la suivante :

- noir : case infranchissable
- jaune : sable
- rouge : case où il vaut mieux ne pas aller car on se mettrait alors en danger de mort.
- vert : case intéressante car permet de pousser un méchant dans un omega, soit parce qu'il est déjà en danger, soit parce qu'il s'y exposera peut-être au tour prochain (pour cela on calcule la probabilité de déplacement d'un méchant sur une case donnée avec les poids suivants : le méchant reste sur place ou recule : poids 1, le méchant avance : poids 3, le méchant prend une autre direction : poids 2)
- chiffre ou @ : gain potentiel ou base

On choisit alors l'action à effectuer en fonction de la couleur de la case sur laquelle notre gentil chameau se trouve et les 4 voisines environnantes.

### 3.3 Réponse à une question cruciale : combien miser ?

Au moins on sait que quand on n'attaque ni ne défend, on mise 1. Après, pas de certitudes... Au niveau du signe de la mise pas trop de problème, suivant les situations, on sait si l'on veut agir avant ou après le méchant. Le problème est vraiment de savoir combien miser. On a donc décidé de faire une mise qui évolue suivant que l'on perd ou l'on gagne un duel. Chaque méchant se voit attribuer une mise optimale qui correspond à un certain pourcentage de l'eau réservée au combat (=eau-(nombre maximum de tour que notre chameau joue avant d'être éjecté par le serveur)). Lorsque qu'on perd un duel : on multiplie cette mise optimale par un certain coefficient pour l'augmenter, idem lorsque l'on gagne pour la réduire (pour économiser l'eau, si précieuse dans un désert). On se limite quand même à une mise maximum pour pas mourrir de soif qui est calculé à partir de la fréquence des combats, de l'eau qu'il nous reste et du nombre de tours qu'il reste à jouer.
La question n'est cependant pas complètement résolue : combien miser pour pousser dans l'omega (ou ne pas être poussé, telle est la question) ? Certains diront beaucoup... certes mais combien ? Si on défend ça doit bien tourner autour de 80% de notre réserve d'eau, mais si on attaque, sachant que l'autre va miser 80%, est-on prêt à en faire autant ? Cela reste une question ouverte. Pour l'instant, nous faisons l'autruche en refusant de voir le problème et on mise 10 fois la mise maximale calculée ci-dessus...

## 4 Une myriade de mondes

Pour tester notre bébé (chameau), nous avons fait un générateur de carte. Pour l'utiliser il faut d'abord créer un fichier .par (voir ci-dessous) puis lancer :
./mapgen nom_du_fichier_par
par exemple, avec un fichier test.par, on tape ./mapgen test et il génère test.lterm et test.map.
Le fichier .par comprend les champs suivant :

- LARGEUR : la largeur de la carte
- HAUTEUR : la hauteur de la carte
- CAPACITE_MAX : la taille maximale des $\lambda$-termes
- NB_CHAMEAUX : le nombre maximum de joueurs
- NB_LTERMS : le nombre de $\lambda$-termes

– NB_BASES : le nombre de bases
– FREQ_OMEGA : la fréquence des omegas sur la carte
– FREQ_PALMIER : la fréquence des palmiers sur la carte
– FREQ_SABLE : la fréquence du sable sur la carte
– MAGIC_NUMBER : le nombre pour initialiser le générateur de nombres aléatoires

Avec tout ça on peut faire plein de choses, mais le générateur ne garantit pas que la carte n'a qu'une composante connexe : il possible qu'un chameau ne puisse pas atteindre une base ou sauver un $\lambda$-terme, dur la vie.

# 5 The queusteumizachieune

Comme vous l'avez compris, nous allons un petit peu expliquer comment choisir entre les différentes stratégies ou changer divers paramètres. Devant un manque de temps lié sans doute aux repas de fêtes nous n'avons pas fait d'interface utilisateur, ce qui manque cruellement. Nous avons toutefois tout rassemblé dans IA.ml.

Donc, aller dans IA.ml. Tout en haut dans set_strategie se trouvent plein de variables au noms étranges. Il suffit de les changer avant la compilation pour avoir un tout nouveau chameau. Ouéééé !

Et pendant qu'on y est, le fichier common.ml rassemble plein de fonctions en vrac qui sont utilisées par les autres modules. C'est une sorte de partage.

# 6 Conclusion

Un petit mot de la fin pour faire le point. D'abord le sujet avait certes ses côtés casse-pieds mais c'était marrant et surtout ça nous permettait de réfléchir sur des types de stratégies et sur des questions d'autonomie sur la prise de décision que l'on aurait pas vu autrement, donc c'est chouette !

Les difficultés rencontrées on surtout été liées au débuggage peu facilité par la présence du serveur et aussi à la multiplicité des modules et des fonctionnalités à gérer.

Merci à vous pour tout le travail de préparation derrière et puis aussi d'avoir lu tout se blabla.

Ah ! J'oubliais. BONNE ANNEE !