ENSEIRB-MATMECA

Filière informatique : Première année

Rapport de Projet S6 :

Factory

BURE / CASTERES / KUHN / MARANNE
Encadrant : Mr Rollet

Mai 2018

Table des matières

1 Introduction

Le jeu Factory est un jeu qui permet de représenter des chaînes de production et de consommation de produits.

Le but principal du jeu est d'obtenir rapidement et de manière efficace une ressource précise : le "Gold". Cette ressource est essentielle dans le jeu car elle permet d'acheter des usines qui, une fois acquises, permettront de produire des ressources diverses (ces ressources obtenues peuvent éventuellement être des "Gold" également). Pour cela nous codons et testons différentes stratégies de jeu afin de comparer leur efficacité. Ces stratégies diffèrent de part la manière de construire la chaîne de production.

Au début du jeu, le joueur possède assez de "Gold" pour acheter une $1^{ière}$ usine (initialisation de ce nombre de "Gold" en fonction des coûts des usines disponibles).

Les usines que l'on peut acheter sont répertoriées dans une liste d'usines, fournie par un fichier texte d'entrée (le **parser**).

1.1 Travail en groupe

Nous avons travaillé pendant ces sept semaines en divisant les tâches entre nous : travail sur les algorithmes, création des tests, rédaction du rapport... Le travail a été également réalisé en dehors des heures de projets prévues pour résoudre les éventuels problèmes rencontrés. Chaque séance de travail commençait par un point sur nos avancements et nos objectifs.

1.2 Problématique

L'objectif de ce projet est d'implémenter le jeu Factory et de trouver une stratégie optimale nous permettant de maximiser les richesses produites.

1.3 Plan

Dans un premier temps nous présenterons le fichier d'initialisation du jeu (le parser).

Nous détaillerons ensuite la représentation des chaînes de production, les stratégies créées et la boucle principale de jeu.

Nous terminerons par la présentation des tests et les évolutions possibles ainsi que les compléments à notre code.

2 Découpage du projet

Pour coder ce projet, nous avons dû le diviser en plusieurs parties :

— un parser : il lit un texte d'entrée et le transforme , de manière à ce qu'il soit utilisable par les fonctions des autres fichiers.

— des fichiers contenant les différentes stratégies de jeu.

— des fichiers tests contenant des tests unitaires.

Nous avons découpé notre projet en deux dossiers principaux :

— Le dossier 'src', qui contient les sources du jeu, à savoir les stratégies, le main, et le parser.

— Le dossier 'tst' contient les fichiers de test pour les fonctions principales, et pour les fonctions du parser. Il contient aussi plusieurs fichiers d'entrées afin de tester le jeu avec des usines différentes.

3 Le parser

Pour initialiser le jeu, et plus particulièrement les usines existantes, il nous faut un fichier texte qui énonce toutes les usines différentes, ainsi que leurs caractéristiques (que doit-elle prendre pour produire quoi, et en quelles quantités ? Quel est le coût en "Gold" de cette usine ?).

Le fichier texte d'entrée doit absolument prendre la forme :

```
[Produit a consommer=quantite] --> [Produits cree=quantite] cost = cout
```

(Exemple de fichier d'entrée)

```
# Any line starting with a '#' is a comment line.
[] --> [Herb=1,Wheat=1] cost = 1
[] --> [Sugarcane=1,Almond=1] cost = 3
[Herb=1,Wheat=2] --> [Beer=1] cost = 2
[Wheat=2]       --> [Flour=1] cost     =     3
# Any facetious and seemingly unnecessary space should be handled correctly
# There shall never be a space between each pair of brackets.
[Flour=3]     --> [Bread=1] cost = 4
[Bread=1]  -->   [Gold=5]    cost = 2
[Sugarcane=1]      --> [Sugar=1] cost = 1
[Sugar=1,Almond=2] --> [Marzipan=1]            cost = 3
# Marzipan is yummy but expensive.
[Marzipan=1] --> [Gold=10] cost = 10
```

On remarquera que les espaces entre les différents éléments n'ont pas forcément à être identiques.

Donc, pour analyser ce fichier texte, nous avons créé le fichier **parser.rkt**. Il fonctionne en deux grandes phases :

— d'une part, il modifie le fichier d'en tête en éliminant tous les éléments superflus dont nous n'auront pas besoin (ex : suppression des commentaires, normalisation des espaces, etc).

— d'autre part, il traite les données obtenues et les mets en forme afin qu'elles soient utilisables par les autres fonctions du jeu.

Le parser possède deux fonctions principales qui sont appelées par le programme principal :

— une fonction 'create-list' qui retourne une liste de string contenant les noms de toutes les ressources disponibles dans la partie.

— une fonction 'factorisation' qui retourne une liste de listes de toutes les usines.

Exemple : le fichier à parser pris en exemple plus haut sera transformé en :

```
'("Gold" "Marzipan" "Sugar" "Bread" "Flour" "Beer" "Almond" "Sugarcane" "Wheat" "Herb")
'((1 1 (0 0 0 0 0 0 0 0 1 1))
  (2 3 (0 0 0 0 0 0 1 1 0 0))
  (3 2 (0 0 0 0 0 1 0 0 -2 -1))
  (4 3 (0 0 0 0 1 0 0 0 -2 0))
  (5 4 (0 0 0 1 -3 0 0 0 0 0))
  (6 2 (5 0 0 -1 0 0 0 0 0 0))
  (7 1 (0 0 1 0 0 0 0 -1 0 0))
  (8 3 (0 1 -1 0 0 0 -2 0 0 0))
  (9 10 (10 -1 0 0 0 0 0 0 0 0)))
```

4 Représentation des chaînes de production

Nous avons choisi de représenter les ressources par une liste.
Pour représenter les usines, nous avons dans un premier temps choisi d'utiliser des structures *factory* contenant leur indice, leur coût et une liste de ressources. Dans cette liste de ressources, les éléments négatifs représentaient les ressources nécessaires à l'usine et les éléments positifs représentaient ceux produits par l'usine.

Cependant, nous avons trouvé cette représentation difficile à utiliser dans nos fonctions. Nous avons donc choisi une nouvelle représentation sous forme d'une liste plus simple à manipuler.
Par exemple : '(3 1 (-2 1 0)) représente l'usine d'indice trois et de coût un. La liste (-2 1 0) signifie que cette usine a besoin de deux ressources d'indice un pour pouvoir produire une ressource d'indice deux.

La liste des ressources et celle des usines sont produites grâce au parser.

Les chaînes de productions dans le niveau 1 étaient représentées par une liste construite pendant le jeu : ajout d'une usine à la liste lors de son achat. Ces listes représentent en réalité des graphes : il existe une relation père/fils entre chaque usine.
Par exemple, ((1 1) (2 6) (3 8)) représente une chaîne de production. Le premier entier correspond à l'indice de l'usine et le deuxième à la quantité nécessaire de cette usine dans la chaîne de production. Le premier doublet correspond à l'usine que l'on veut obtenir au final. Mais pour cela, il faut au préalable avoir acheté l'usine présentée dans le doublet précédent et ainsi de suite jusqu'à la fin de la liste. C'est à ce moment là qu'est introduite la notion de graphes avec les liens entre usines père et usines filles.

Pour valider la création d'une chaîne d'usines, il faut vérifier la viabilité des chaînes de production. Pour cela, on utilise alors la fonction *viable*. *Viable* signifie que toutes les ressources peuvent être produites. Cette fonction vérifie donc que toutes les ressources consommées et crées par la chaîne de production s'équilibrent bien, que les quantités sont suffisantes.

On se pose alors plusieurs questions :

— y a-t-il plusieurs chaînes de production possibles ?

— si oui, comment choisir l'une ou l'autre ?

— quelle est la meilleure chaîne pour arriver à la création de "Gold", la plus optimale ?

5 Stratégies

Pour répondre aux mieux à ces questions, nous avons fait le choix de créer différentes stratégies. Ces stratégies correspondent à un ensemble de fonctions.

5.1 Stratégie 1

Dans un premier temps, nous avons élaboré une stratégie basique dont l'objectif était d'acheter pendant N tours de jeu la même usine. Pour coder cette stratégie nous avons créé les fonctions 'ajout1' et 'produce1'.

5.2 Stratégie 2

Notre deuxième stratégie est une stratégie plus évoluée.

Elle consiste en construire une chaîne viable permettant de créer de l'or. On parcourt alors la liste des usines fournie par le parser pour trouver l'ID de la première usine créant de l'or. A partir de là, il suffit de construire un chemin d'usines permettant d'arriver jusqu'à celle produisant ce "Gold". Pour cela, il faut à nouveau parcourir récursivement la liste d'usines (sorties par le parser).

Cette recherche correspond à un parcours de graphe : on va, par dépendances entre ressources créées et ressources utilisées, chercher le chemin vers l'or le moins coûteux.

Cette stratégie est déjà moins basique et automatique que la $1^{ière}$. Elle permet de diversifier les achats d'usines et éventuellement accéder de manière plus rentable à la création de "Gold".

En effet, par exemple, considérons la liste d'usines suivante (liste créée juste pour l'exemple. Pas forcément de lien entre les différents noms ou de cohérence entre la création des différentes ressources) :

```
1 '("Gold" "Wheat" "Herb")
2 '((1 1 (0 0 1)) (2 1 (0 1 -1)) (3 1 (6 -1 0)) (4 3 (3 0 0)))
```

Pour jouer avec ces usines là, le "Gold" attribué initialement au joueur sera de 3.

Pour la stratégie 1 : comme on ne peut acheter que la même usine à chaque tour, on va alors considérer l'usine 4 qui produit directement du "Gold".

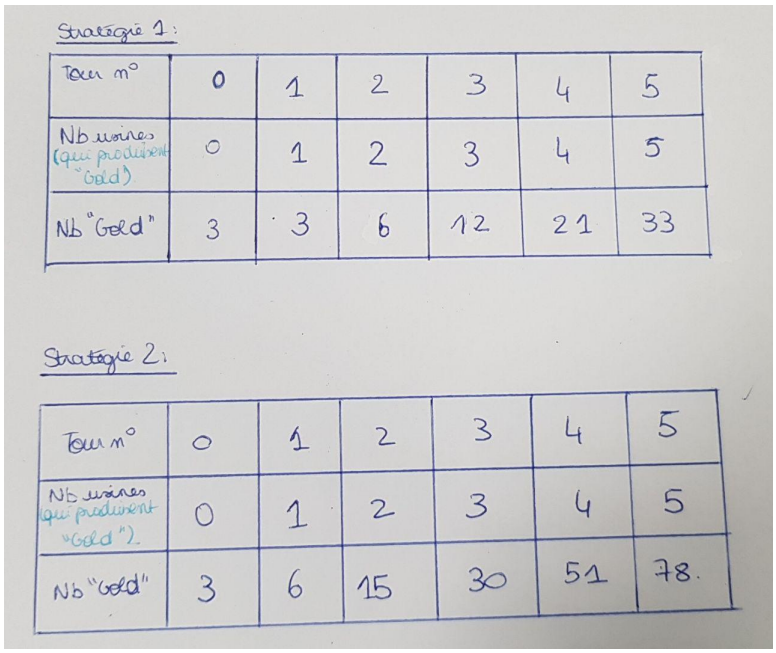
On remarque alors que pour acheter cette usine qui produit 3 "Gold", il en faut 3. Au final, au premier tour, on ressort avec 3 "Gold". (3 initiaux - 3 achat usine + 3 créés par usine achetée).

Pour la stratégie 2 : on cherche tout d'abord dans la liste des usines fournies par le parser, l'ID de la 1^{ere} usine qui produit du "Gold" (dont l'ID est 1 dans la liste des ressources). Cet ID correspond à celui de l'usine 3. Une fois cette usine trouvée, on produit la chaîne qui permet d'arriver à cette ressource. On obtient alors : '((3 1) (2 1) (1 1)).

Une fois cette chaîne trouvée, analysons-la. Elle est composée de 3 usines. Chaque usine est nécessaire uniquement une fois pour créer les ressources nécessaires à la création finale du "Gold". Le coût de chaque usine est de 1. Au final, le coût de cette chaîne est donc de 3 "Gold". On peut donc dès le début acheter une usine de chaque type pour obtenir dès le premier tour les "Gold" de l'usine finale. On se retrouve donc, en fin de tour, avec 6 "Gold" en notre possession. (3 initiaux - 3 achats usines + 6 créés par usine finale).

Comparaison des stratégies : Dès le 1^{ier} tour, on remarque que, dans ce cas là d'usines, la stratégie 2 est plus rentable que la stratégie 1.

Voici deux tableaux qui présentent l'évolution des "Gold" en fonction du tour de boucle et du nombre d'usines de "Gold" en notre possession.



Stratégie 1 :

Tour n°	0	1	2	3	4	5
Nb usines (qui produisent "Gold")	0	1	2	3	4	5
Nb "Gold"	3	3	6	12	21	33

Stratégie 2 :

Tour n°	0	1	2	3	4	5
Nb usines qui produisent "Gold"	0	1	2	3	4	5
Nb "Gold"	3	6	15	30	51	78.

Pour l'analyse de ces tableaux , il faut savoir que l'on part du principe que l'on peut acheter au max une usine / une chaîne d'usines créant du "Gold" à chaque tour.

On remarque donc que alors que le temps passe, la stratégie 2 permet d'obtenir plus rapidement du "Gold" que la stratégie 1.

La rentabilité de la 2ième stratégie est alors mise en valeur avec cet exemple. Cette mise en évidence va s'accetuer au fur et à mesure que les tours de boucle s'effectuent.

5.3 Stratégie 3

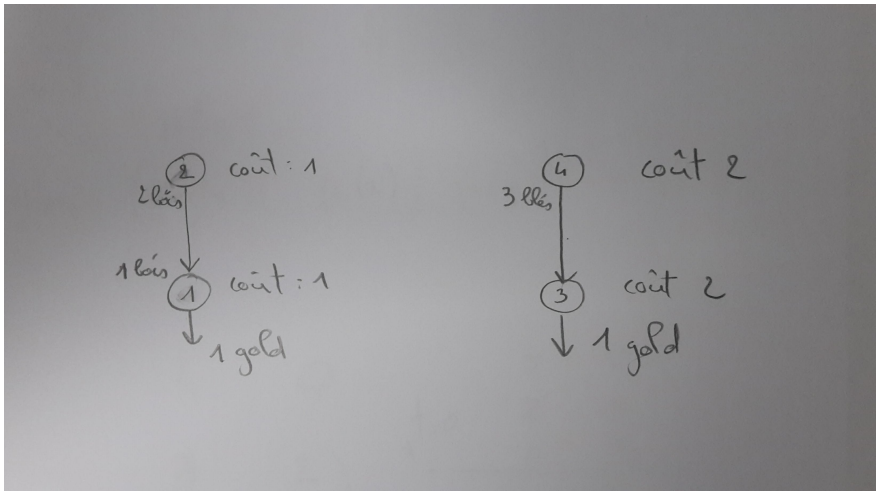
Au final, une limite se pose à la stratégie 2 : elle prend la $1^{ière}$ chaîne de production créant du "Gold", sans vérifier si une autre chaîne en produit et est éventuellement plus efficace.

Pour palier à ce manque et optimiser le jeu, nous avons alors choisi d'utiliser une stratégie considérant les automates finis.

Au préalable, à partir de la liste d'usine il est nécessaire de construire la liste des états des chaînes de production. Une Chaîne de production, comme définie plus haut est un ensemble d'usines à obtenir pour faire fonctionner une usine de Gold.

Prenons un exemple d'usines :
((1 1 (1 -1 0)) (2 1 (0 2 0)) (3 2 (1 0 -1)) (4 2 (0 0 3))))

On peut construire deux chaînes comme montré sur le dessin suivant

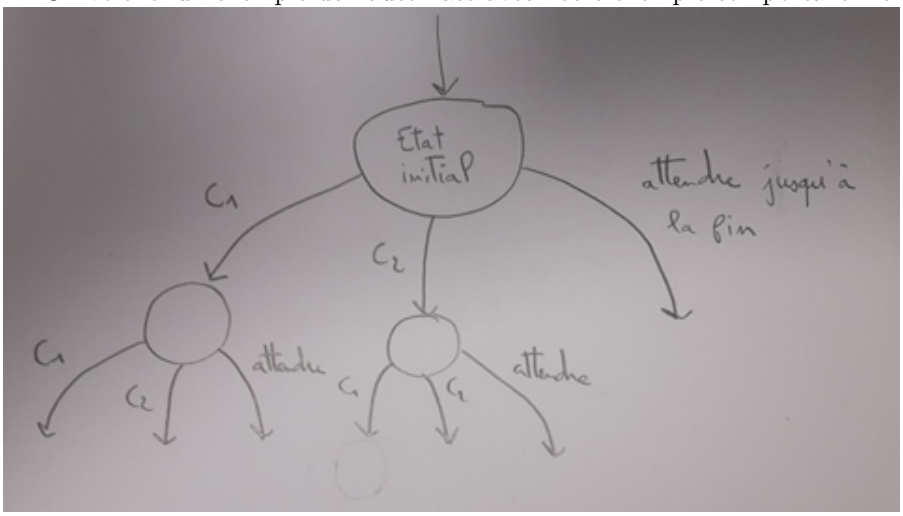


Pour chaque chaîne, vient alors la notion d'état. En effet si on achète une première fois la chaîne liée à la première usine, il faut acheter une fois l'usine 1 et une fois l'usine 2 pour un coût total de 2. On aura alors un surplus de 1 de bois et donc si on veut faire fonctionner cette chaîne de nouveau, il suffit d'acheter uniquement l'usine 1 pour un coup de 1. Ensuite il n'y a plus de surplus, on retombe sur le cas initial. Pour cette chaîne il y a donc 2 états possibles, coûtants 2 et 1.
On construit ainsi pour chaque chaîne la liste de ses états, associé à son gain. Pour notre exemple on obtient : ((1 (2 1)) (1 (4 2 2)))

Nous avons maintenant tous les outils pour faire fonctionner l'automate. Un état de l'automate est caractérisé par 4 paramètres : le stock de Gold, le revenu par tour, l'état de toutes les chaînes et le tour. Une transition représente le prochaine achat à faire. Il est aussi possible d'attendre jusqu'à la fin du jeu et de seulement profiter de ses revenus pour augmenter son capital Gold. L'objectif, pour déterminer la quantité de Gold optimale que l'on peut produire en n tours de jeu, est de recenser tous les chemins possibles menant à un état où le jeu est terminé et de sélectionner celui où le nombre de Gold est maximum. Alors, le "mot" pour arriver à cet état est l'enchaînement d'achats à réaliser afin

d'arriver à ce résultat.

On voit ici un exemple de l'automate avec notre exemple comportant 2 chaînes :



6 Boucle de jeu

La fonction principale qui réalise la boucle de jeu commence par parser le fichier d'entrée afin d'établir la liste d'usines disponibles et la liste des ressources existantes pour chaque usine.

Dans un second temps, en fonction de la stratégie considérée, la recherche des chaines de production se fait de manière plus ou moins différentes et de manière plus ou moins simple (cf les différentes stratégies utilisées précédemment).

Au final, suite à cette recherche de chaine, la boucle de jeu incrémente (si on a une usine qui en produit) et décrémente (si on achète une usine) la valeur de "Gold" que l'on a avant de la retourner.

7 Présentation des tests

Afin de tester au mieux les fonctions écrites, des tests unitaires ont été créés. L'utilité d'employer ce type de tests là est que l'on possède et connaît déjà les résultats attendus. On peut alors vérifier que les fonctions utilisées nous retournent bien ce que l'on attend et ainsi vérifier qu'elles sont bien codées.

Afin de couvrir un maximum de cas (qu'ils soient particuliers ou non), 3 types de fichiers d'entrée ont été créés et stockés dans des fichiers .txt. Ces fichiers textes sont écrits sous la forme décrite lors de l'explication du fonctionnement du parser, page 4 du rapport. Ces fichiers d'entrée seront alors chargés, au besoin, au début de chaque fichier test.

En plus de ces plateaux-test, d'autres pourront être créés spécialement pour effectuer certains

tests pour certaines fonctions. En effet, selon les tests à effectuer, il est parfois plus simple de coder à la main une liste d'usines simplifiée, plutôt que d'utiliser une grande liste d'usines sortie par le parser.

Ces définitions de plateau ou de structures en début de fichier test sont donc des compléments aux fichiers d'entrée déjà créés pour mieux tester les fonctions et tester un plus grand nombre de cas facilement et rapidement.

8 Problèmes rencontrés liés à la programmation fonctionnelle

La principale difficulté au départ de ce projet a été de s'adapter au langage `racket` et à la réalisation de programmes fonctionnels. En effet, c'est un langage assez complexe au départ puisque nous étions principalement habitués à de la programmation impérative en C. Le coté programmation fonctionnelle du Racket nous pousse donc à réfléchir de manière différente, notamment en terme de récursivité.

Un point qui a été compliqué est le fait que nous réalisions beaucoup de parallèles avec le C ou même d'autres langages de programmation dont nous avions plus l'habitude. L'utilisation de structures nous est alors apparu évident pour coder les usines, mais après réflexion, il nous est paru plus simple de procéder à l'aide de listes.

Le début du projet a donc consisté brièvement (une séance tout au plus) à faire des recherches sur le fonctionnement des structures dans un langage fonctionnel. D'autres recherches ont suivies, comme comment relier deux fichiers racket entre eux afin de pouvoir utiliser les fonctions d'un fichier à un autre, etc ... Parmi ces recherches, beaucoup ont été axées autour des fichiers et des listes pour réaliser le parser (comment convertir un fichier texte en liste pour pouvoir le décomposer et l'analyser etc).

Un autre soucis majeur qui a guidé notre choix pour l'utilisation de listes pour le codage des usines, et pour l'écriture des codes a été d'éviter les effets de bord. Une fonction est sujette à des effets de bord si elle modifie un état autre que sa valeur de retour. Ces effets peuvent être provoqués par la modification de variables globales par exemple.

Des fonctions utilisées dans le projet peuvent être optimisées. Par exemple, des fonctions auraient pu être utilisées pour diminuer la complexité.

9 Évolutions possibles

Les évolutions possibles pour notre code seraient d'introduire des usines de type niveau trois dans notre liste. Ces usines sont des usines prenant plusieurs ressources d'entrée et de sortie sans jamais créer de cycle d'usines. En effet, les usines de niveau 2, peuvent avoir plusieurs ressources d'entrée, mais deux usines distinctes ne peuvent avoir la même ressource d'entrée. De plus, il n'existe pas de cycle dans le graphe liant les différentes usines.

L'utilisation d'usines de niveau 3 nous fait alors encore une fois utiliser la technique des graphes pour représenter au mieux les chaînes de production des ressources.

Afin de trouver la stratégie optimale, nous avons pensé utiliser des automates finis, ou rajouter des poids sur les arrêtes des graphes afin de pouvoir déterminer le chemin le moins coûteux pour arriver à

certains états du jeu, et ainsi pouvoir optimiser les chaînes de production.

Afin d'être sur de la meilleure stratégie nous voulions mettre en place un système qui comparerai deux stratégies à chaque tour pour voir l'évolution des golds.

10 Conclusion

Tout au long du projet, nous avons essayé d'adopter un regard critique sur notre travail et de remettre en questions nos choix algorithmiques et la propreté de nos codes. Ce projet nous a permis de progresser en programmation fonctionnelle et de mettre en pratiques toutes nos connaissances en algorithmique, utilisation de graphes et d'automates finis.