1ère année informatique

Projet de programmation fonctionnelle

Rapport du projet Factory

Team :
Bahrami Tara
Maghraoui Sahar
Nis Cedric
Dermigny Basile

Encadrant :
M. LOMBARDY Sylvain

18 mai 2018

Table des matières

1 Introduction

Le but de ce projet est d'implémenter des chaînes de production et de consommation de produits manufacturés. Les chaînes de production dans ce jeu consistent à composer des ensembles d'usines produisant des denrées à partir d'autres, elle-même obtenu à partir d'autre. Tous ceci dans le but de maximiser les golds obtenus en fin de productions.

Nous allons tout au long du projet nous intéresser aux chaînes de production elle-mêmes, et leur optimisation. Chaque chaîne de production est composée d'usines qui ont un coût, un ensemble de ressources d'entrée consommées et un ensemble de sorties produites par unité de temps. Pour qu'une chaîne de production soit viable il faut que la première usine ne consomme rien et que la dernière produise du gold.

Chaque tour de jeu est défini de la manière suivante. Tout d'abord, le joueur commence par choisir de construire de nouvelles usines pour renforcer la capacité de production, en fonction de ses ressources en Gold. Ensuite, les usines peuvent consommer et produire en parallèle les ressources, si elles reçoivent les ressources nécessaires.

Ce projet est composé de trois niveaux afin de nous simplifier l'implémentassions de ce dernier. Les caractéristiques des usines différencient ces différents niveaux.

— 1^{er} niveau : chaque usine a au plus une ressource d'entrée et une ressource de sortie.
— 2^{e} niveau : les usines peuvent avoir plusieurs ressources d'entrée, deux usines distinctes ne peuvent avoir la même ressource d'entrée mais toujours qu'une.
— 3^{e} niveau : les usines peuvent avoir plusieurs ressources d'entrée et de sortie.

2 Niveau 1

2.1 L'implémentation

À ce niveau du jeu, chaque usine a au plus une ressource d'entrée et de sortie. Nous avons donc décidé d'implémenter les usines de la manière suivante :

```
(struct factory (consommation production cout))
```

La structure *factory* est composée de deux paires : une pour la consommation et une pour la production. Ainsi que d'un entier représentant le coût d'achat de la factory. Le premier élément de la paire définit le nom de la ressource et le deuxième élément le nombre de ressources consommées ou produites par unité de temps. Nous définissons donc une usine de la manière suivante :

```
(define wheatfact (factory (cons "wheat" 2) (cons "flour" 1) 10))
```

L'usine *wheat-fact* consomme deux unités de blé, produit une unité de farine et coûte 10 gold.

Ainsi, chaque chaîne de production est représentée par une liste d'usines. Nous avons donc créé la structure *bench* :

```
(struct bench (lst-fact))
```

Et enfin, la structure *chain* composée d'une liste de *bench*. Cette structure permet de mettre en évidence toutes les chaînes de productions créées.

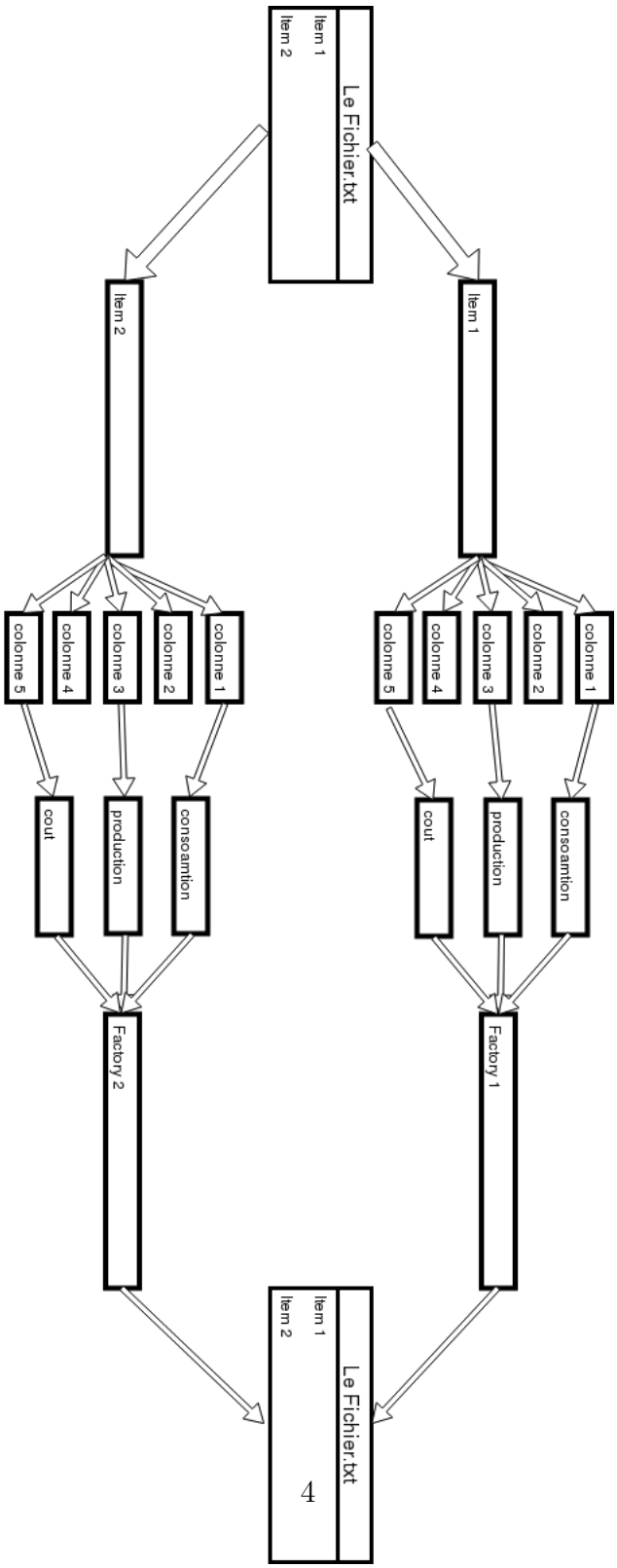
```
(struct chain (lst-bench))
```

Maintenant que nous avons défini l'implémentation des usines et des chaînes de production, nous pouvons passer au parseur.

2.2 Le parseur

Le parseur permet de transformer un fichier texte de donnée en une liste d'usine.

Cependant, les fichiers textes doivent avoir un format spécial afin de pouvoir convertir ce dernier. En effet, nous prenons en compte les colonnes 1, 3 et 5 des lignes ne commençants pas par un dièse (la colonne 1 contient la consommation de l'usine, la colonne 3 la production et la colonne 5 le coût). Nous avons donc une fonction *conversion* qui trie les lignes commentées et qui traduit les lignes utiles à l'aide de la fonction *traduction*. Les chaînes de production étant implémentées, nous pouvons passer aux fonctions qui nous permettent d'implémenter la boucle du jeu.


Le Fichier.txt
Item 1
Item 2
Item 1
Item 2
colonne 1
colonne 2
colonne 3
colonne 4
colonne 5
colonne 1
colonne 2
colonne 3
colonne 4
colonne 5
consoamtion
production
cout
consoamtion
production
cout
Factory 1
Factory 2
Le Fichier.txt
Item 1
Item 2

2.3 Tour de jeu

On peut jouer le tour de jeu selon deux règles différentes :
— La première règle qui consiste à acheter qu'une seule usine par tour.
— La seconde qui consiste à acheter autant d'usines que possible à chaque tour.

Règle 1

Pour pouvoir coder la fonction qui créée la boucle jeu, il suffit de connaître le gold du joueur, la chaîne de production actuelle et les achats réalisés pendant le tour. Elle retournera ensuite le gold actuel et la chaîne de production après le tour.

Le code sera de la forme suivante :

Algorithm 1 tour-de-jeu-r1

Require: *achat*
Require: *chaines*
Require: *gold*
  $gold \leftarrow gold - (factory - coutachat)$
  $add - end - rec(achat, chaines)$

Pour pouvoir coder le tour de jeu nous avons besoin d'une fonction auxiliaire *add-end-rec* qui retournera faux et la chaîne actuelle si une usine ne peut pas être ajoutée en fin de chaîne ou vrai et la chaîne de production contenant l'usine si cette dernière peut être ajoutée. Cet algorithme sera implémenté de la manière suivante :

Algorithm 2 add-end-rec

Require: gold
Require: achat
Require: chaines
  if chaines vide **then**
    retourner faux et chaines
  else if on peut ajouter la factory à la première liste se trouvant dans chaines **then**
    retourner vrai et la chaine de production modifiée
  else
    tour de jeu (factory (cdr chaines) gold)
  end if

Règle 2

Pour implémenter la 2^e règle nous avons créé la fonction *tour-jeu-r2* qui prend en argument le gold, la chaîne de production (*chaines*) et contrairement à la règle 1, elle prend une *market* qui est une liste d'usines. Nous faisons ce choix car on peut acheter autant d'usines que possible à chaque tour. Ainsi, la fonction *tour-jeu-r2* est récursive.

Elle suit le schéma suivant :

Si *add-end-rec* appliquée à gold, *(car market)* et *chaines* est vraie on renvoie *tour-jeu-r2* appliquée au gold actuel (on soustrait le gold initial au gold de la première market car elle est achetée), à chaines modifiée à l'aide de *add-end-rec* et au cdr de market. Sinon, on retourne *tour-jeu-r2* au gold initial, chaines intial et (cdr market) car la première usine de market n'a pas été achetée.

2.4 Production du Gold optimale en n tours de jeu

Le but de cette sous-partie est d'implémenter un algorithme permettant de déterminer la quantité de Gold optimale que l'on peut produire en n tours de jeu.

Pour cela nous avons cherché à implémenter plusieurs fonctions permettant de trouver le chemin le plus rentable. Nous avons procédé de la manière suivante :

- Tout d'abord nous extrayons dans les chaînes de production toutes les usines qui produisent du gold à l'aide de *extract-factg* qui prend en argument la liste de toutes les usines que le joueur a acheté et une liste initialement vide qui sera remplie au fur et à mesure des usines qui produisent du gold.
- Ensuite, on enlève toutes ces usines de la liste d'origine.
- Enfin, on crée une forêt dont la racine de chaque arbre est une usine qui produit du gold.

Ainsi par exemple, considérons la chaîne de production *Chaines* qui est de la forme suivante :

$$'((fact0fact1fact2)(fact3fact2)(fact5fact1fact4)(fact0fact1fact4))$$

En appliquant nos différentes fonctions à cette chaîne de production nous obtenons la forêt suivante.

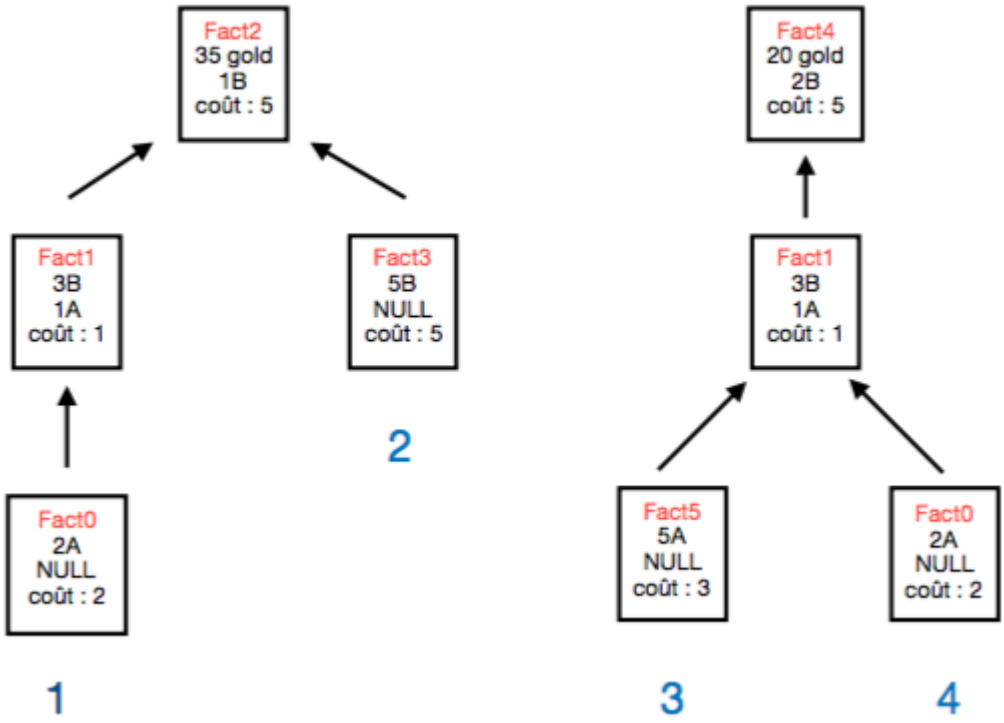


FIGURE 1 – Forêt créée à partir de la chaîne *Chaines*

Nous pouvons voir sur la FIGURE 1 qu'il y a quatre chemins possibles pour créer du gold. Ainsi, à l'aide de cette forêt nous pourrons facilement déterminer le chemin le plus rentable.

Pour trouver le meilleur chemin, nous avons implémenté la fonction *best-road-tree* qui dans chaque arbre de la forêt nous indique le chemin le plus rentable. Après avoir appliqué *best-road-tree* à chaque arbre, nous cherchons le meilleur chemin parmi tous les arbres.

Reprenons l'exemple *Chaines*. Nous pouvons voir ci-dessous que chaque chemin a un coût différent.

— Chemin 1 : production de 27 golds*
— Chemin 2 : production de 25 golds*
— Chemin 3 : production de 11 golds*

— Chemin 4 : production de 13 golds*

**après déduction des coûts de chaque usine*

En appliquant *best-road-tree* à l'arbre de gauche, cette dernière retourne le chemin 1 et à l'arbre droit elle retourne 4. Cependant, le chemin 1 étant plus optimal que le 4 on trouve finalement que le meilleur chemin est le 1.

Ainsi, nous pouvons voir sur cet exemple qu'il est préférable de créer des chaînes de production de la première forme pour obtenir la quantité de Gold optimal.

3 Niveau 2

Dans le niveau 2, les usines peuvent avoir plusieurs ressources d'entrée mais une ressource de sortie. Cependant, deux usines distinctes ne peuvent avoir la même ressource d'entrée.

3.1 Problématique introduite par l'extension du niveau 2

Avec cette nouvelle définition, on aura besoin d'un graphe. Ce dernier remplacera notre structure *chain* qui était composée d'une liste de liste d'usines. Nous faisons ce choix afin de faciliter la lecture des chaînes de production.

Sur ce graphe, les noeuds représentent les usines, les arcs sortants représentent la production et les entrants la consommation.

Considérons l'exemple suivant :
— Usine1 consomme "Pain" et "Lait" et produit "Pain perdu"
— Usine2 consomme "Orange" et Sucre" et produit "Jus"
— Usine3 consomme "Pain perdu" et "Jus" et produit "Dej"
Ainsi, la chaîne de production est construite de la manière suivante :

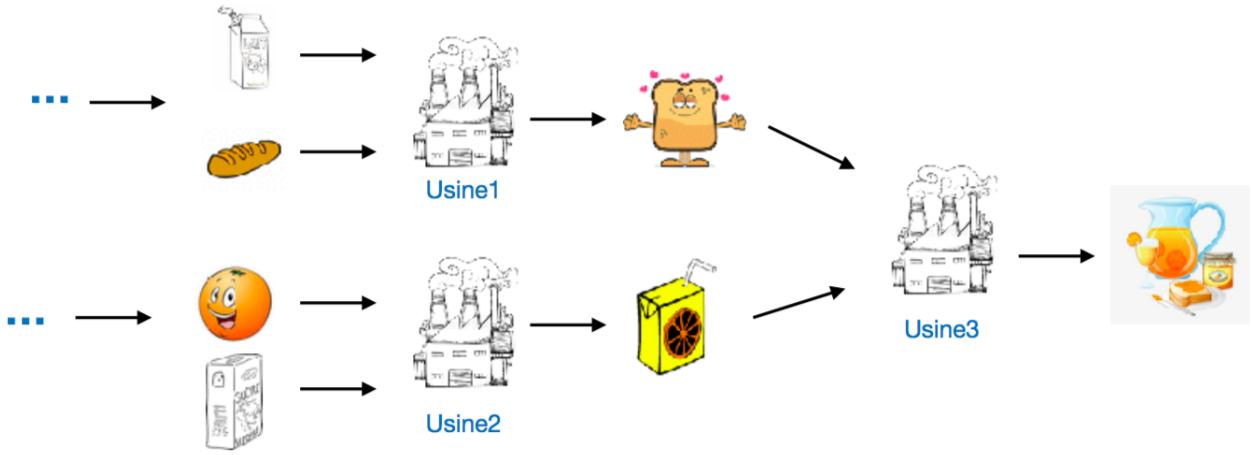


FIGURE 2 – Exemple : représentation de la chaîne de production

Nous voyons bien à travers cette exemple que l'utilisation d'un graphe facilite la construction et la lecture de la chaîne de production.

3.2 L'implémentation

Nos structures *factory, bench, chain* ne changent pas de forme vu qu'on a pas utilisé type racket et donc le langage n'est pas typé. Cependant, dans la structure *factory* qui est de la forme suivante,

```
(struct factory (consommation production cout))
```

le paramètre consommation n'est plus une paire mais une liste de paires au vu de la nouvelle définition de l'usine.

De plus nous avons créé quatre nouvelles structures pour pouvoir créer le graphe :

```
(struct graphe (node_racine lstnode lst-adj))
```

Avec *node_racine* est le noeud racine, *lstnode* une liste de noeud et *lst-adj* une liste de liste d'entiers qui représente la liste des adjacences.

```
(struct node (id factory))
```

Avec *id* un entier unique par noeud et *factory* une structure factory.

```
(struct arc (cout id-deb id-end))
```

Avec *cout id-deb id-deb* trois entiers : $id - deb \xrightarrow{\text{cout}} id - end$

```
(struct chemin (lst-id cout gain))
```

Avec *lst-id* la liste des id des noeud qui compose le chemin, *cout* le prix d'achat du chemin et *gain* la quantité de gold produite par le chemin.

Nous avons ici toutes les structures qui nous permettent d'implémenter un graphe. Nous allons donc maintenant étendre les algorithmes permettant de déterminer la quantité optimale de Gold produite en n tours.

3.3 Production du Gold optimale en n tours de jeu

Pour produire une quantité de Gold optimale nous allons procéder de la même manière qu'au niveau 1. Nous allons analyser tous les chemins possibles pour ensuite choisir le meilleur.

Ainsi par exemple, considérons le graphe de la figure 1.

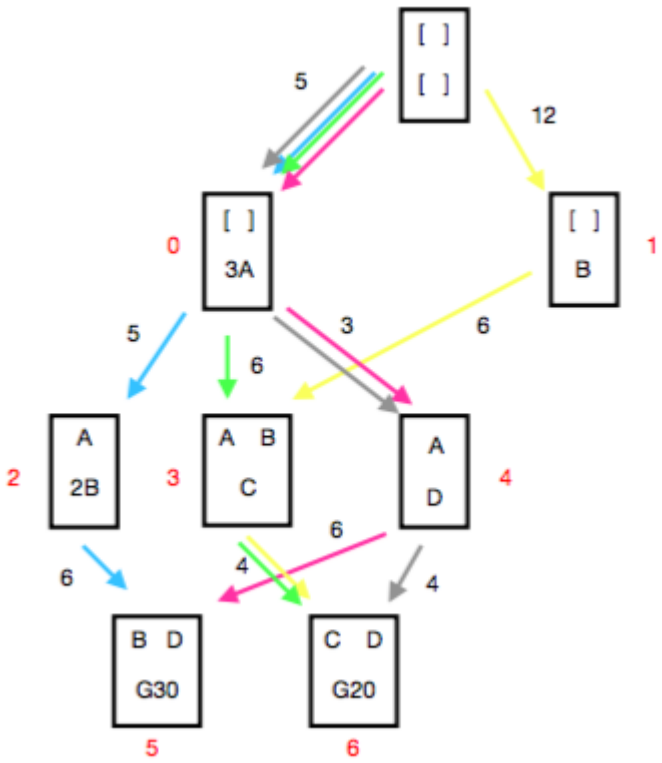


FIGURE 3 – Exemple : Graphe de niveau 2

Ce graphe admet pour liste d'adjacence la liste suivante :

```
{{2 3 4} {3} {5} {6} {5 6} {} {}}
```

Avec 0 1 2 3 4 5 6 l'*id* des noeuds. (Le coût des noeuds 5 et 6 sont respectivement 6 et 4.

Ainsi, la fonction *best-road-tree* détermine le meilleur chemin de l'arbre. Dans l'exemple ci-dessus :

- Pour acheter l'usine 5 on doit emprunter le chemin bleu et rose. On sera donc amené à acheter les usines 0, 2, 4, 5. Finalement cette chaîne produira 5 Golds.
- Pour acheter l'usine 6 on peut emprunter différents chemins :
  - Le chemin jaune et gris. Cette chaîne sera en déficit de 34 Golds.
  - Le chemin vert et gris. Cette chaîne sera en déficit de 2 Golds.

Nous pouvons donc en conclure que le meilleur chemin est le rose.

3.4 Les différentes stratégies

Dans cette sous-partie nous allons proposer des stratégies de rendement efficace. Malheureusement, nous ne les avons pas toutes implémentées, par faute de temps.

Random

La stratégie *Random* consiste à choisir une chaîne d'usines au hasard sous réserve qu'elle produise du Gold.

Ainsi par exemple, sur la figure 3 nous pouvons choisir soit le chemin bleu et rose qui nous permet l'achat de l'usine 5, soit les chemins vert et gris ou jaune et gris pour acheter l'usine 6.

Elle ne choisira pas par exemple le chemin bleu et gris car ces chemins ne permettent ni l'achat de l'usine 5 ni celui de l'usine 6.

Toujours la moins chère

La stratégie *Toujours la moins chère* consiste à choisir les usines moins chères. Si nous reprenons la figure 6, notre algorithme choisirai d'abord l'usine 0 ($5 < 12$) puis l'usine 4 ($3 < 5 < 6$) et enfin l'usine 5 qui crée du Gold. Le premier chemin choisi est donc le rose puis on sera contraint de choisir le chemin bleu pour pouvoir acheter l'usine 5.

Usine à plus forte production de Gold

La stratégie *Usine à plus forte production de Gold* consiste à choisir l'usine qui produit le plus de Gold et ensuite choisit les chemins nécessaires pour arriver à cette usine. Par exemple, si l'usine 6 produisait 40 Golds et non 20, l'algorithme choisirait à fortiori le chemin gris et le chemin jaune ou vert au hasard.

Meilleur rendement

La stratégie *Meilleur rendement* achète les usines qui permettent un meilleur rendement. Nous avons vu le fonctionnement de l'algorithme dans la partie 3.3.

3.5 L'extension du niveau trois

A ce stade du projet, les usines peuvent avoir plusieurs ressources d'entrée et de sortie. Nous n'avons pas eu le temps de l'implémenter. Néanmoins, si nous n'étions pas restreint par le temps nous aurions choisi comme pour le niveau 2 un graphe. Cependant,

il aurait été un peu plus complexe car plusieurs arcs peuvent sortir d'un seul et même noeud.

4 Difficultés rencontrées et conclusion

4.1 Difficultés rencontrées

Durant tout le projet nous avons rencontré plusieurs difficultés. Tout d'abord, le passage d'une programmation impérative en fonctionnelle. En effet, nous sommes habitués à utiliser des boucles *for while etc.* pour implémenter nos fonctions. Ce projet nous a donc poussé à utiliser des fonctions récursives. Ainsi, nous avons appris à nous adapter selon ce qui nous est demandé.

Ensuite, pour que le code soit compréhensible de tous, il faut faire un réel effort sur les commentaires de chaque fonction. Surtout que dans notre cas, il fallait être d'autant plus précis car nous avons décidé de ne pas typer le code.

4.2 Conclusion

Pour conclure, nous avons appris à nous adapter selon l'environnement de travail. Cependant, nous avons pas eu le temps de finir. Cela s'explique sûrement par la présence des deux projets assez longs et d'une répartition inégale du travail sur les deux projets.