Théorie de la complexité des algorithmes
Problème des K-centres

FRANÇOIS HERNANDEZ - LÉO PONS
CentraleSupélec
February 3, 2017

Contents

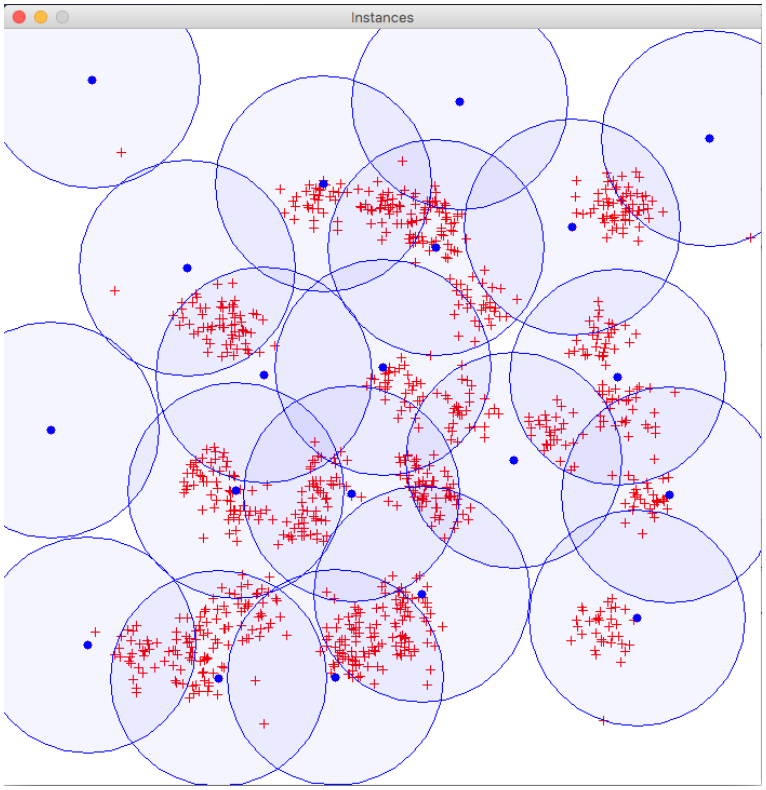
Introduction

Le problème des K-centres est un problème d'optimisation combinatoire. Le problème peut se décrire de façon informelle ainsi (Wikipedia) : étant donné n villes, il faut ouvrir une caserne de pompiers dans k villes, tel que la distance entre chaque ville et la plus proche caserne soit minimisée.

Il existe plusieurs définitions des problèmes K-centres. Ici, nous considérons la formulation suivante. Supposons un ensemble E de n points dans un espace vectoriel ou un graphe complet muni d'une fonction de distance satisfaisant l'inégalité triangulaire. Il s'agit de trouver k "centroids" afin de minimiser la distance maximale entre un point de E et le centre le plus proche.

Ce problème de décision est NP-complet.

Au cours de ce projet, nous avons mis en place un programme en Java constituant une représentation du problème, et proposant différents algorithmes de résolution sur différents ensembles d'instances.



I Implémentation

Nous avons choisi d'implémenter ce problème dans le langage Java. Nous avons mis en place des fonctions de génération et d'affichage d'instances (ensemble de points), ainsi que différents algorithmes de résolution à tester, et des méthodes de lecture et écriture des données sous format texte.

Le programme comporte ainsi les packages suivants :

- `Generation` : contient les différentes classes de génération d'instances, ainsi que celle associée à la lecture des fichiers .txt ;
- `Graphics` : contient les différentes classes associées à l'affichage des instances ;
- `Resolution` : contient les différents algorithmes de résolution ;
- `Main` : contient le main qui permet de lancer les algorithmes de test, ainsi que la définition des différents objets utilisés ;

I.1 Objets utilisés

Les objets utilisés dans ce programme sont définis de la façon suivante :

- `Point` : ensemble de coordonnées, ici `x` et `y` dans l'espace à deux dimensions ;
- `Instance` ensemble de n `Points` sous forme d'`ArrayList`, et un attribut k définissant le nombre de centres attendus, contient également une méthode permettant d'exporter l'instance au format .txt ;
- `Solution` : ensemble de k `Points` constituant les centres d'une `Instance`, ainsi qu'un attribut *rayon* définissant le rayon minimal pour lequel tous les points de l'instance sont couverts.

I.2 Génération d'instances

Nous avons choisi d'implémenter deux méthodes principales de génération d'instances :

- `Uniforme` : génère n `Points` de façon aléatoire dans l'espace défini (en conservant des marges afin de ne pas avoir de points trop proches des bords, pour des raisons d'affichage) ;
- `Cluster` : LEO

I.3 Affichage des données

Afin d'avoir une représentation graphique claire des différentes instances et des résultats des algorithmes de résolution, nous avons choisi de définir une interface graphique à

l'aide des outils `Swing`. Cela consiste en deux classes, `Fenetre` et `Panel`. `Fenetre` hérite de la classe `JFrame` et définit les caractéristiques de la fenêtre qui contiendra le `Panel`. `Panel` hérite de la classe `JPanel` et contient les méthodes associées au dessin des représentations. La méthode `paintComponent` dessine les différents éléments dans un `bufferGraphics`, attribut d'une image.

I.4 Algorithmes de résolution

LEO

I.5 Conversion en fichier texte

Un standard d'export des instances a été défini afin de pouvoir sauvegarder et échanger différentes instances de test. Celui-ci est le suivant.

```
nombre d'instances i

n1, k1
x1, y1, x2, y2, [...], xn1, yn1

n2, k2
x1, y1, x2, y2, [...], xn2, yn2

[...]

ni, ki
x1, y1, x2, y2, [...], xni, yni
```

La méthode `createFile` de la classe `Instance` permet de créer de tels fichiers à l'aide de la classe `FileWriter` de Java.

La classe `Importer` du package `Generation` permet de lire de tels fichiers à l'aide des classes `FileReader` et `BufferedReader` de Java.

II Théorie, NP-complétude et approximation

II.1 Implémentation Générale

Afin d'implémenter le problème des philosophes, nous créons un nouveau package 'philosophes' contenant différentes classes héritant des classes génériques de la plateforme, ainsi que de nouvelles destinées à représenter d'autres aspects spécifiques au problème.

Les classes héritant de la plateforme sont les suivantes :

- `Philosophe`, héritant de la classe `Agent`.
- `Table`, héritant de la classe `Environnement`.
- `philosophes.actions`, package de classes héritant de la classe `Action`. Chacune de ces classes définit une action exécutable par les philosophes.
- `Fourchettes`, héritant de la classe `Donnee` et contenant la représentation de l'état des N fourchettes de l'environnement.

Les classes supplémentaires sont les suivantes :

- `Etat`, type énuméré représentant l'état d'un `Philosophe`.
- `main`, permettant de faire tourner le système.

Comme précisé en introduction, le problème du dîner des philosophes peut être construit de différentes manières. Voici quelques points de comportement que nous avons décidé d'adopter :

- Le nombre de philosophes est défini par l'attribut `effectif` de la `Table`.
- Les philosophes commencent tous avec un compteur de faim `effectif` à 0, et leur état est `en_train_de_penser`.
- Quand un philosophe pense, un compteur de mesure `compteurPensee` est incrémenté à chaque tour au niveau de la `Table`.
- Le fait de penser donne faim aux philosophes, ce qui est quantifié par l'attribut `deltaFPenser` de la `Table`.
- Passé un certain seuil de faim défini par l'attribut `seuilFaim` de la `Table`, le philosophe ne peut plus penser et a faim, il essaie alors de ramasser ses fourchettes pour manger.
- Lorsqu'un philosophe a faim et qu'il ne mange pas, sa faim s'aggrave, ce qui est quantifié par l'attribut `deltaFFaim` de la `Table`.
- Passé un certain seuil de faim défini par l'attribut `seuilFamine` de la `Table`, le philosophe est en état critique de famine : un compteur de mesure `compteurFamine` est incrémenté à chaque tour au niveau de la `Table`.
- Lorsqu'un philosophe mange, sa faim diminue, ce qui est quantifié par l'attribut `deltaFManger` de la `Table`.
- Passé le seuil de satiété défini par défaut à 0, le philosophe s'arrête de manger et peut recommencer à penser.

II.2 Machine à états

Nous avons défini quelques actions de base pour nos philosophes. À chaque tour, selon l'état du philosophe et l'observation des données, une action est exécutée. Pour plus de clarté, nous avons représenté l'ensemble des actions implémentées sur une machine à états simplifiée. Les messages ne sont pas encore pris en compte.

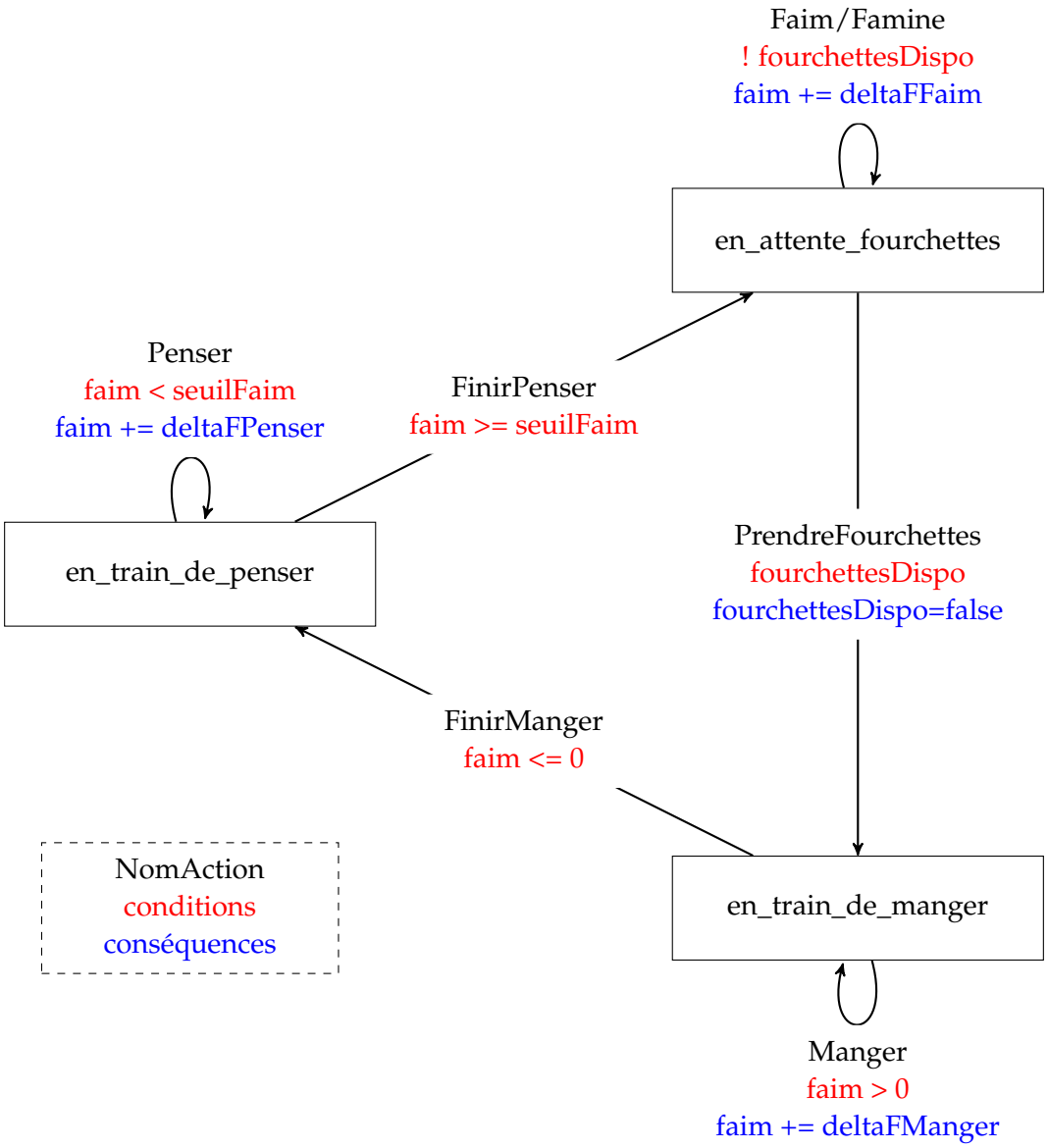


Figure 1: Représentation en machine à état du système

II.3 Rôle des paramètres

Bien évidemment, le comportement du système est largement affecté par le choix des constantes de l'environnement. Par exemple, un choix de `deltaFManger` démesurément grand (en valeur absolue) par rapport à `deltaFPenser` et `deltaFFaim` a pour conséquence de simplifier grandement le problème : chaque philosophe peut passer son temps à penser et ne manger qu'une fois de temps pour rattraper son retard. Les fourchettes sont alors presque tout le temps disponible.

L'intérêt bien sûr n'est pas de regarder ses philosophes manger mais bien de définir des constantes équilibrées pour étudier un système nuancé. Après quelques essais, nous avons décidé de travailler dans un premier temps `seuilFaim = 10, deltaPenser = 5, deltaManger = -2, deltaFaim = 1`. On remarquera que le choix de `deltaFFamine` ne joue que sur la variable de mesure `compteurFamine` et non sur le fonctionnement même du système.

Le choix de `deltaFFamine` est donc un problème de convention. Pour le contexte donné établi précédemment, on trouve par exemple une famine totale accumulée sur 20 000 tours de 2662, 521, et 23 pour des valeurs respectives de `deltaFFamine` de 50, 60, et 70. On choisira `deltaFFamine`=60 pour les prochaines simulations.

Le choix du nombre de philosophe est nettement plus intéressant. En effet, on remarque vite que le comportement du système est assez chaotique : les scores de famine et de pensée peuvent varier drastiquement d'une configuration à l'autre, ceci sans que l'on ai pu identifier de relation logique entre l'effectif et ces scores (cf. Figure 3).

Ce comportement est problématique. Dans le cas où l'on a 8 philosophes, le comportement du système est désastreux. Il faut donc mettre en place une solution pour régulariser ce genre de cas, et pourvoir travailler avec un système moins imprévisible.

II.4 Ajout des messages

Le problème de beaucoup de configurations malheureuses, c'est que certains philosophes s'accaparent les couverts pendant que d'autres n'ont jamais l'occasion de les récupérer. On peut essayer d'éviter cela en mettant en place un système de communication entre agents.

La communication entre agents passe par la classe `Message`. Un agent peut envoyer un message à un autre agent en créant une instance de `Message` spécifiant son identifiant, l'identifiant du destinataire, et le contenu du message. L'envoi est effectué via l'objet d'environnement avec la méthode `send` qui redistribue le message au destinataire.

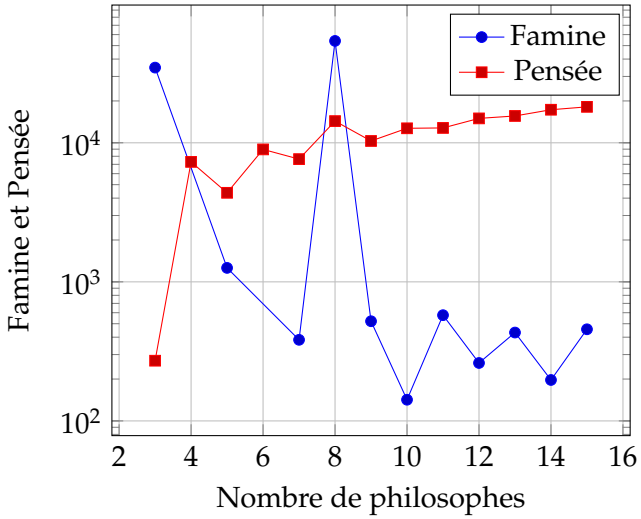


Figure 2: Famine et pensée finales en fonction du nombre de philosophes

Le message est alors stocké dans la boite aux lettres du destinataire dans l'attente d'être traité. Les messages sont ensuite régulièrement supprimés (par exemple à la fin de chaque tour) pour éviter d'accumuler de vieux messages. Pour le traitement, plusieurs solutions sont envisageables au niveau de l'implémentation d'un problème particulier.

Pour nos philosophes, nous avons décidé de mettre en place une solution simple : au début de son tour le philosophe consulte le dernier message qu'il a reçu, les autres sont ignorés. Le contenu du message intervient alors au niveau des conditions d'exécution des actions, au même titre que les autres données du système.

Un premier type de message a été mis en place pour diminuer les scores de famine : `Help`. Cette démarche a été très fructueuse, donc nous nous sommes contentés de ce type de message dans un premier temps. Le message est émis par un philosophe lorsqu'il passe le seuil de famine, et est envoyé à ses deux voisins pour leur demander de lâcher leurs fourchettes. Ainsi, au tour suivant, le philosophe peut manger et on évite un état de famine prolongé.

III Méthodes de résolution

IV Évaluation des performances

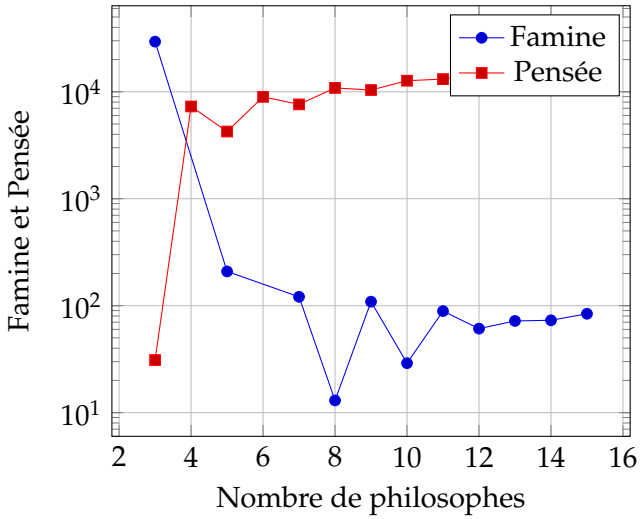


Figure 3: Famine et pensée finales en fonction du nombre de philosophes - avec messages