

# Algorithmique et programmation (INF431)

Benjamin Werner      François Pottier

29 janvier 2012



# Introduction

Ce cours s'adresse aux élèves de deuxième année de l'Ecole Polytechnique, c'est-à-dire à des étudiants sachant déjà programmer et ayant acquis une certaine familiarité avec les notions d'algorithme et de complexité calculatoire. Ses objectifs sont typiques d'un cours avancé de second cycle universitaire :

- Compléter votre connaissance du paysage algorithmique et surtout la *structurer*. À l'issue de ce cours, vous aurez une vision relativement complète des techniques algorithmiques fondamentales. Nous avons choisi de regrouper les algorithmes d'après les principes qui les sous-tendent (algorithmes gloutons, diviser-pour-régner, programmation dynamique. . .). Nous espérons ainsi rendre cet apprentissage à la fois plus stimulant et plus profond.
- Renforcer votre maîtrise de la programmation et plus généralement de la conception des logiciels. À l'issue de ce cours, vous saurez non seulement implémenter et utiliser des algorithmes et des structures de données, mais aussi comment structurer un programme et comment tenter de s'assurer de sa correction. Nous souhaitons permettre à de futurs ingénieurs et décideurs, qu'ils soient informaticiens ou pas, d'approcher ce qu'est la réalisation d'un logiciel et de saisir ce qui est possible, ce qui ne l'est pas, et pourquoi.

Ces objectifs sont résumés en deux mots par l'intitulé : « Algorithmique & Programmation ».

## Pourquoi étudier l'algorithmique et la programmation ?

Il est inutile d'insister sur l'omniprésence des programmes informatiques dans le monde actuel. Rappelons simplement que :

1. Le logiciel n'est pas uniquement là où on l'attend. La recherche d'informations sur internet (Google, Bing, Yahoo. . .), le traitement d'images fixes ou animées (Photoshop, Gimp, Pixar. . .) ou les logiciels financiers sont des exemples où l'interaction avec l'ordinateur est manifeste. Mais on peut trouver de plus en plus d'exemples d'informatique *enfouie* (ou « *embedded* » en anglais). Il y a des programmes, parfois complexes, dans les freins ou les injecteurs des voitures, les montres, les machines à café ou à laver et, évidemment, les outils modernes de communication.
2. L'informatique pénètre chaque jour un peu plus les autres sciences ; on parle de bio-informatique, de physique sur ordinateur, d'expérimentation in silico. . . À chaque fois, ces nouvelles interactions amènent à repenser le monde un peu plus sous l'aspect algorithmique.

Si, à travers ces nouvelles applications, l'outil informatique est de plus en plus mêlé à des questions venant d'autres horizons, les principes algorithmiques fondamentaux n'en restent pas moins valables. Savoir les utiliser est souvent ce qui fait la différence.

## Pourquoi Java ?

De nouveaux langages de programmation sont conçus quotidiennement. En quelques dizaines d'années, il en a été inventé plusieurs milliers. C'est dire qu'il n'existe pas de langage parfait et qu'il faut faire un choix de compromis. La décision d'utiliser le langage Java (plus précisément Java 6) pour ce cours est un tel choix imparfait.

Les avantages de Java sont :

- Il est largement utilisé, et sa maîtrise est utile dans de nombreux champs de l'activité industrielle et scientifique,
- il dispose de nombreuses *bibliothèques*, c'est-à-dire d'éléments logiciels susceptibles d'être utilisés dans d'autres programmes,
- Java insiste sur le style de programmation dit *orienté-objet*, présentant de nombreux intérêts,
- enfin Java permet de ne pas se préoccuper de certains aspects techniques, comme la gestion de la mémoire.

Il y a également quelques désavantages. Par exemple, Java peut être considéré comme relativement verbeux, c'est-à-dire que le code écrit en Java peut être plus long que son équivalent dans d'autres langages. De manière générale, nous mentionnerons les aspects spécifiques à Java lorsque cela nous apparaîtra nécessaire mais présenterons les principes des algorithmes de manière aussi indépendante que possible du langage.

## Contenu du cours

L'objet de l'algorithmique est de comprendre si l'on peut résoudre tel ou tel problème par le calcul, et si oui, de quelle manière, et à quel prix en termes de temps et de mémoire. Cette discipline est essentiellement indépendante du choix d'une machine ou d'un langage de programmation particuliers.

La programmation consiste plus spécifiquement à exploiter un langage de programmation particulier pour organiser les programmes de façon simple, élégante, et robuste. Un programme complexe est toujours constitué de nombreux composants, si possible conçus indépendamment les uns des autres, et dont on tente de vérifier le bon fonctionnement indépendamment les uns des autres.

Algorithmique et programmation vont souvent de pair. Si certaines séances de ce cours sont à tonalité majoritairement « algorithmique » ou majoritairement « programmation », il ne sera pas rare que les deux aspects soient abordés au cours d'une même séance.

Voici un bref aperçu du contenu des dix-huit séances de cours :

1. **Introduction à l'algorithmique.** Le problème des mariages stables, emprunté à l'ouvrage de Kleinberg et Tardos, nous permettra d'illustrer la façon dont on conçoit un algorithme et les questions que l'on pose à son sujet. Voir le chapitre 1.
2. **Introduction à la programmation orientée objet.** Nous présenterons ou rappellerons quelques principes indépendants de Java, par exemple le découpage d'un programme en composants indépendants et la séparation entre interface et implémentation de chaque composant. Nous rappellerons comment les traits propres à Java (classes, interfaces, héritage, etc.) permettent de mettre en œuvre ces idées.
- 3, 5, 6. **Structures de données et récursivité.** Nous présenterons deux familles de structures de données fondamentales, à savoir les arbres et les graphes. Nous mettrons l'accent

sur l'utilisation de la récursivité, qui est essentielle pour parcourir et transformer aisément ces structures. Nous mettrons également l'accent sur la distinction entre structures modifiables et persistantes. Voir les chapitres 3, 4 et 6.

- 4, 7, 8, 9, 12. **Techniques algorithmiques fondamentales.** Un petit nombre d'idées et de techniques fondamentales président à la conception de nombreux algorithmes et font partie de la « boîte à outils » des algorithmiciens et des ingénieurs en informatique. Nous présenterons le « diviser pour régner » (« *divide and conquer* »), les algorithmes gloutons, la programmation dynamique, ainsi que l'exploration systématique d'un univers des possibles. Voir les chapitres 5, ??, ?? et 9.
- 10, 11. **Correction des programmes.** Écrire des programmes, c'est bien, mais s'ils sont corrects, c'est mieux, beaucoup mieux. Nous proposerons une introduction aux techniques de spécification, de test, et de preuve de programmes. Ainsi, nous définirons ce qu'est un « bug » et verrons comment on peut détecter et éliminer les bugs. Voir les chapitres 7 et 8.
- 13, 14, 15. **Traitement de la syntaxe.** Un programme est lui-même une donnée, que d'autres programmes peuvent construire, examiner, transformer. Nous verrons comment on passe d'une représentation textuelle des programmes à une représentation arborescente, que l'on peut ensuite exécuter. Ceci nous permettra d'une part de mieux comprendre comment la machine interprète les programmes que nous écrivons, d'autre part de continuer à étudier des algorithmes où les arbres et la récursivité jouent un rôle prépondérant. Voir les chapitres 10, 11 et 12.
- 16, 17, 18. **Concurrence.** Depuis plusieurs décennies déjà, mais à nouveau et surtout depuis l'avènement des processeurs multi-cœurs, un programme peut engendrer plusieurs processus dont l'exécution se fait de façon simultanée au sein d'une seule machine. Cette organisation conduit à une performance et à une flexibilité potentiellement accrues, mais introduit également des difficultés nouvelles. Nous étudierons comment ces processus (« *threads* ») fonctionnent et comment ils peuvent communiquer les uns avec les autres de façon cohérente. Voir les chapitres 13, 14 et 15.

Ce polycopié contient de nombreux exercices, souvent corrigés. Nous vous recommandons bien sûr d'étudier ces exercices afin de vous familiariser avec les notions que nous développons.

Pour en savoir plus, il existe de nombreux manuels d'algorithmique, dont « le Cormen » (Cormen *et al.*, 2002), « le Sedgewick » (Sedgewick et Wayne, 2011) ou *Algorithms* (Dasgupta *et al.*, 2008). Nous avons particulièrement utilisé *Algorithm Design* (Kleinberg et Tardos, 2006), dont la réflexion pédagogique nous semble très pertinente. D'autres ouvrages sont cités, au besoin, au fil des différents chapitres.

## À propos de ce document

Cette année, le contenu du cours et son organisation ont été profondément revus ; de même, ce polycopié est entièrement nouveau. Il est inévitable qu'il comporte de nombreuses imperfections.

Nous vous remercions de nous signaler les erreurs que vous relèverez, et de nous faire part de vos remarques et suggestions d'amélioration.

Janvier 2012

François Pottier et Benjamin Werner



# Chapitre 1

## Un problème type : les mariages stables

Nous commençons ce cours par le traitement algorithmique d'un problème particulier : celui des *mariages stables*, ou *stable matchings* en anglais. Le but est d'une part de reprendre contact avec les algorithmes après une rupture de quelques mois, mais aussi :

- d'illustrer la puissance de la notion d'algorithme, qui nous permettra ici d'établir un résultat mathématique combinatoire non trivial,
- d'illustrer comment l'analyse mathématique du problème permet de construire un algorithme,
- d'illustrer comment on prouve la correction de l'algorithme,
- de rappeler comment on établit la complexité d'un algorithme opérant sur une structure de données simple.

Pour toutes ces raisons, sans doute, ce problème et l'algorithme associé ont fait l'objet d'une série de conférences du grand informaticien et algorithmicien Donald Erwin Knuth lors d'une visite à l'université de Montréal. Le contenu de ces conférences a donné lieu à la publication d'un fascicule en français (Knuth, 1976) dont la lecture reste aujourd'hui étonnamment agréable et stimulante.

À la fin du chapitre, nous présentons un certain nombre de variantes du problème principal en montrant que leur résolution demande des techniques très différentes.

### 1.1 Le problème

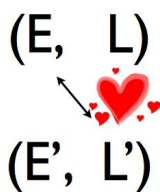
Le problème dit des mariages stables a été présenté et étudié à l'origine par deux économistes, David Gale et Lloyd Shapley, en 1962. Il s'agit d'étudier la question de l'admission d'étudiants par une université, ou de nouveaux employés par des entreprises.

Prenons l'exemple des élèves de l'École Polytechnique qui doivent choisir un stage de recherche à la fin de leur troisième année. Il y a deux ensembles distincts d'acteurs : les élèves d'une part, les laboratoires proposant des stages de l'autre. On suppose que chaque étudiant a ordonné les laboratoires par ordre de préférence, et chaque laboratoire a ordonné les étudiants par ordre de préférence.

S'il n'y a pas de règles claires pour régir le mécanisme d'affectation de laboratoires aux élèves, les choses peuvent facilement devenir chaotiques. Supposons que Séverine a accepté une offre de stage de l'entreprise *Winch mobile*. Mais un peu plus tard, au Liechtenstein, la banque *DarkWater* fait une offre à Séverine. Celle-ci, qui adore la montagne, se laisse convaincre par cette nouvelle offre et renonce au stage *Winch mobile*. Ces derniers proposent alors leur stage à d'autres étudiants dont Thomas qui, fanatique de réseaux, les rejoint en laissant tomber le stage qu'il envisageait au LIX. Le bureau des stages commence à s'arracher les cheveux.

La situation empire si les laboratoires commencent également à rompre des promesses de stages au fur et à mesure qu'ils rencontrent des étudiants qui leur plaisent plus.

L'affectation des stages restera instable tant qu'on pourra avoir la situation suivante :  $E$  a promis de faire son stage chez  $L$  et  $E'$  chez  $L'$ . Mais finalement  $E$  préfère  $L'$  à  $L$  et  $L'$  préfère  $E$  à  $E'$ .



Une instabilité manifeste

Pour éviter les dissymétries on va supposer que chaque laboratoire veut recruter un seul stagiaire, tout comme chaque étudiant recherche un seul stage. On considère de plus qu'il y a exactement autant de laboratoires que d'étudiants.

Ce problème peut donc aussi être présenté comme la formation de couples stables dans une population entièrement hétérosexuelle. Dans la suite du chapitre, on s'autorisera cette métaphore. *Nous soulignons qu'il ne s'agit en aucun cas de défendre telle ou telle vision de la vie en couple, de telle ou telle forme de sexualité, et certainement pas de différence de valeur entre les sexes. N'y voyez pas non plus une quelconque prise de position vis-à-vis des mariages de même sexe.*

On se donne donc deux ensembles finis et distincts  $H$  et  $F$ , de même cardinal. On appelle leurs éléments respectivement les hommes et les femmes.

**Définition 1.1.1 (Couplage)** Un ensemble de paires homme-femme  $A \subset H \times F$  est un couplage si chaque homme et chaque femme apparaissent dans au plus un élément de  $A$ .

Le couplage  $A$  est *parfait* si chaque homme et chaque femme apparaissent dans exactement une paire élément de  $A$ .  $\diamond$

**Définition 1.1.2 (Stabilité)** À chaque homme et chaque femme on associe un ordre total sur l'ensemble de sexe opposé. Cet ordre représente ses préférences. Lorsque  $h$  préfère  $f$  à  $f'$ , on note  $f >_h f'$ .

Un couplage parfait  $A$  est *stable* lorsque :

$$\forall ((h, f), (h', f')) \in A^2. \neg (f' >_h f \wedge h >_{f'} h'). \quad \diamond$$

## 1.2 L'algorithme

L'existence même d'un couplage stable n'est pas évidente à prouver. Il est remarquable que la manière la plus simple de le prouver est de commencer par donner l'algorithme, puis d'étudier ses propriétés.

**tant que** il existe un homme  $h$  qui est libre et n'a pas demandé toutes les femmes en mariage  
**soit**  $h$  un tel homme  
**soit**  $f$  la femme préférée de  $h$  parmi celles qu'il n'a pas encore demandées en mariage  
**si**  $f$  est libre  
**alors**  $h$  et  $f$  se fiancent  
**sinon**  
**soit**  $h'$  le fiancé de  $f$   
**si**  $f$  préfère  $h$  à  $h'$   
**alors**  $f$  quitte  $h'$  (qui redevient libre) et se fiance avec  $h$

FIGURE 1.1 – L'algorithme de Gale et Shapley

Comme nous le ferons souvent, nous donnons cet algorithme sous forme de *pseudo-code*. C'est-à-dire que nous le décrivons sous une forme plus ou moins proche du programme exécutable. Dans le cas présent, nous ne rentrons pas dans les détails d'implémentation.

Au cours de l'algorithme, des couples sont formés. Comme ces couples peuvent encore être défaits au cours de l'exécution, on parle de couples de fiancés, jusqu'à l'achèvement de l'algorithme. L'algorithme de Gale et Shapley est dissymétrique par rapport aux deux sexes : il donne l'initiative à l'une des deux parties. L'algorithme présenté dans la figure 1.1.

L'analyse de cet algorithme va nous permettre de prouver l'existence d'un couplage stable. Elle nous permet également d'illustrer les étapes essentielles qu'on retrouve dans l'analyse de la plupart des algorithmes : terminaison, propriétés finales, invariants de boucle, complexité, implémentation dans un vrai langage de programmation.

### 1.2.1 Description de l'algorithme

On remarque d'abord que le pseudo-code de la figure 1.1 n'est pas complètement précis. D'une part on ne précise pas les structures de données utilisées. Mais aussi, à chaque itération de la boucle, on laisse libre le choix de  $h$  parmi tous les hommes libres qui n'ont pas encore épuisé leur liste. Cela veut dire qu'il y a plusieurs exécutions possibles qui correspondent à cette description ; a priori, ces exécutions pourraient aboutir à des résultats différents. On peut donc voir cette description comme la donnée d'un *algorithme non-déterministe*.

Par ailleurs, dans un premier temps, on remarque qu'il n'est pas immédiatement garanti :

- que l'algorithme termine,
- que si l'algorithme termine, qu'il termine sur un couplage, et a fortiori sur un couplage parfait, c'est-à-dire que tout le monde est fiancé à la fin,
- s'il termine sur un couplage parfait, que ce couplage est stable.

Ces points sont donc l'objet des paragraphes suivants. On va voir que la description donnée de l'algorithme est suffisamment précise pour ces trois propriétés. Puis que toutes les implémentations respectant cette description aboutissent au même couplage.

### 1.2.2 Terminaison

Considérons l'ensemble des couples possibles  $(h, f)$ . Cet ensemble est de cardinal  $n^2$ . A chaque itération de la boucle, un homme demande une femme en mariage, et cette demande n'a pas été formulée précédemment. Notons  $\mathcal{P}(i)$  l'ensemble des demandes ayant été formulées après  $i$  itérations. Puisque, à chaque itération, la demande formulée est nouvelle, on voit que  $|\mathcal{P}(i)| = i$ . Comme  $|\mathcal{P}(i)|$  est borné par  $n^2$ , on en déduit :

**Théorème 1.2.1** L'algorithme termine après au plus  $n^2$  itérations.  $\diamond$

On a donc trouvé une *mesure* ou un *variant* de la boucle. C'est une technique habituelle, qui permet d'abord de garantir la terminaison de la boucle. Elle permet également de donner une borne supérieure sur le temps d'exécution : il suffira pour cela de donner une borne sur le temps de calcul nécessaire à un tour de boucle.

Remarquons également qu'un certain nombre d'autres grandeurs qui pourraient venir à l'esprit ne conviennent pas pour prouver la terminaison. Par exemple, le nombre d'hommes fiancés ne convient pas, parce qu'il peut rester constant au cours de certaines itérations. En général trouver un bon argument de terminaison est une tâche récurrente en informatique, et suivant les cas, elle peut être arbitrairement difficile.

### 1.2.3 Correction

Pour établir des propriétés du résultat d'un programme itératif, il est généralement utile de trouver des propriétés qui restent vraies au cours de l'exécution, c'est-à-dire qui sont préservées par chaque étape. C'est la notion d'*invariant de boucle*, qui sera au cœur du chapitre 8.

Il apparaît immédiatement à la vue du pseudocode que le seul cas où une femme quitte son fiancé, c'est pour rejoindre un fiancé meilleur (de son point de vue). Aussi a-t-on :

**Lemme 1.2.1** A partir du moment où une femme est demandée en mariage au cours de l'algorithme, elle reste fiancée. De plus, ses fiancés successifs s'améliorent, du point de vue de sa liste de préférences.  $\diamond$

De même, on vérifie facilement :

**Lemme 1.2.2** A tout moment, l'ensemble des couples fiancés est un couplage. En particulier, il y a autant d'hommes que de femmes fiancés.  $\diamond$

Ces premiers lemmes nous permettent de montrer :

**Lemme 1.2.3** Si un homme  $h$  est libre à un moment donné de l'exécution, alors il y a au moins une femme qu'il n'a pas encore demandée en mariage.  $\diamond$

**Démonstration** Si un homme a demandé toutes les femmes en mariage, on sait, d'après le lemme précédent, que toutes les femmes sont fiancées. Si toutes les femmes sont fiancées, alors tous les hommes le sont aussi ; donc  $h$  en particulier l'est aussi.  $\square$

On en déduit que si un homme est libre, alors il lui reste des propositions à faire. Donc l'algorithme ne s'arrête que lorsqu'il n'y a plus d'hommes libres, et dans ce cas, il n'y a plus non plus de femmes libres :

**Lemme 1.2.4** L'algorithme de Gale et Shapley termine en produisant un couplage parfait.  $\diamond$

Remarquons aussi que l'on peut donc reformuler l'algorithme en simplifiant la première ligne en "*tant qu'il existe un homme  $h$  libre*", puisqu'un homme libre aura toujours encore des propositions à faire.

Il nous reste à vérifier la stabilité.

**Théorème 1.2.2** L'algorithme de Gale et Shapley termine en produisant un couplage stable.  $\diamond$

**Démonstration** On vient de voir que l'algorithme produit un couplage parfait. Supposons maintenant que le résultat ne soit pas stable, c'est-à-dire que le résultat comporte  $(h, f)$  et  $(h', f')$  avec :

$$f' >_h f \wedge h >_{f'} h'.$$

On sait alors que  $h$  a fait une proposition à  $f$ . Or, puisque  $h$  préfère  $f'$  à  $f$ , il a, précédemment, au cours de l'exécution, fait une proposition à  $f'$ .

$f'$  a pu accepter ou non cette proposition. Dans les deux cas, d'après le lemme 1.2.1, elle doit, au terme de l'algorithme, être fiancée à un homme qui lui plaît au moins autant que  $h$ . Or, ce n'est pas le cas : contradiction.  $\square$

### 1.3 Propriétés supplémentaires

Nous n'avons pas encore traité de la question des ambiguïtés dans notre présentation de l'algorithme de Gale et Shapley. Pour les comprendre, il est utile de regarder un exemple admettant plusieurs couplages stables.

Prenons une population de deux hommes ( $h_1$  et  $h_2$ ) et deux femmes ( $f_1$  et  $f_2$ ). Dans certains cas, il n'y a qu'un couplage stable. Par exemple, si tous les hommes d'une part et toutes les femmes d'autre part sont d'accord et ont les mêmes listes de préférences :

$$\begin{aligned} h_1 \text{ et } h_2 & : f_1; f_2 \\ f_1 \text{ et } f_2 & : h_1; h_2 \end{aligned}$$

Dans cette situation, le seul couplage stable est celui qui marie les plus "populaires" entre eux :  $(h_1, f_1)$  et  $(h_2, f_2)$ .

Si les hommes d'une part, les femmes d'autre part, ne font pas le même choix, il y a un cas "favorable" qui permet d'attribuer à chacun son premier choix :

$$\begin{aligned} h_1 & : f_1; f_2 & h_2 & : f_2; f_1 \\ f_1 & : h_1; h_2 & f_2 & : h_2; h_1 \end{aligned}$$

Reste le cas où les préférences des hommes et des femmes sont opposées :

$$\begin{aligned} h_1 & : f_2; f_1 & h_2 & : f_1; f_2 \\ f_1 & : h_1; h_2 & f_2 & : h_2; h_1 \end{aligned}$$

On a, dans ce dernier cas, deux possibilités : donner leur premier choix soit aux deux femmes, soit aux deux hommes. Dans les deux cas on obtient un couplage stable, puisque les membres de l'un des deux sexes refuseront de changer de couple.

Si l'on essaie les différentes exécutions possibles de l'algorithme de Gale et Shapley sur cette configuration, on verra qu'elles aboutissent toujours à la configuration favorable aux hommes. Bien sûr, l'algorithme symétrique obtenu en inversant les rôles des hommes et des femmes favorisera les femmes.

On peut établir que l'algorithme de Gale et Shapley favorise toujours l'un des deux sexes.

**Définition 1.3.1** Soit  $f \in F, h \in H$ . On dit que  $h$  (resp.  $f$ ) est un(e) *partenaire valide* de  $f$  (resp. de  $h$ ) s'il existe un couplage stable auquel appartient  $(h, f)$ .

On désigne par  $best(h)$  la meilleure partenaire valide de  $h$  (du point de vue de  $h$ ). On désigne par  $worst(f)$  le pire partenaire valide de  $f$  (du point de vue de  $f$ ).

Enfin on appelle  $\mathcal{S}^*$  l'ensemble  $\{(h, best(h)) \mid h \in H\}$ .  $\diamond$

On a alors le résultat assez surprenant suivant :

**Théorème 1.3.1** Toute exécution de l'algorithme de Gale et Shapley termine en produisant l'ensemble  $\mathcal{S}^*$ .  $\diamond$

Ce résultat signifie plusieurs choses. D'une part, cela montre que  $S^*$  est un couplage parfait, et même un couplage stable. D'autre part, cela montre que les ambiguïtés dans la description de Gale et Shapley sont en fait sans conséquence sur le résultat final. En d'autres termes, même si l'algorithme est présenté de manière non-déterministe, il se comporte, du point de vue de l'utilisateur, de manière déterministe.

**Démonstration** On montre qu'au cours de l'exécution :

1. chaque homme  $h$ , s'il est fiancé, est fiancé à une femme supérieure ou égale à  $best(h)$  dans sa liste de préférences,
2. chaque homme  $h$  qui n'est pas fiancé, n'a fait des propositions qu'à des femmes strictement supérieures à  $best(h)$  dans sa liste de préférences.

On procède par récurrence sur le nombre d'itérations de la boucle principale effectuées.

C'est évidemment vrai au début, après 0 itérations, où aucun homme n'est encore fiancé (ce qui implique (1)) et aucun homme n'a fait encore aucune proposition (ce qui implique (2)).

Supposons que les propriétés soient vraies après  $n$  itérations. La  $n + 1$ -ième itération voit un homme  $h$  faire une proposition à une femme  $f$ . D'après (2), il n'a fait des propositions qu'à des femmes strictement supérieures à  $best(h)$ ; donc  $f \geq_h best(h)$ .

Si  $f$  se fiance avec  $h$ , on satisfait donc bien (1).

Si  $f$  est fiancée avec  $h'$  on sait de plus que  $f \geq_{h'} best(h')$ .

Si  $f$  quitte  $h'$  pour  $h$ , il faut de plus montrer que  $f >_{h'} best(h')$ . Il suffit pour cela de vérifier que  $f \neq best(h')$ . Supposons un couplage stable dans lequel on aurait  $(h', f)$ ; alors  $h$  serait associé à une femme  $f' \leq_h best(h) \leq_h f$ . Comme  $f \neq f'$ , on a aussi  $f' <_h f$ . Le couplage ne peut donc être stable.

Si  $f$  est fiancée avec  $h'$  et ne le quitte pas pour  $h$ , il faut montrer de plus que  $f >_h best(h)$ . Il suffit pour cela de vérifier  $f \neq best(h)$ . Supposons un couplage stable dans lequel on ait  $(h, f)$ . Alors  $h'$  serait associé à une femme  $f' \leq_{h'} best(h') \leq_{h'} f$ . Comme  $f \neq f'$ , on a aussi  $f' <_{h'} f$ . Le couplage ne peut donc être stable.  $\square$

On voit donc que l'algorithme de Gale et Shapley est complètement spécifié quant au résultat, et aussi qu'il favorise autant que possible un sexe sur l'autre.

Solution page 199. **Exercice 1.3.1** Montrer que l'algorithme de Gale et Shapley, dans la forme ci-dessus, est aussi défavorable que possible aux femmes. Plus précisément, montrer qu'il produit l'ensemble  $\{(worst(f), f) \mid f \in F\}$ .  $\diamond$

## 1.4 Le marché n'a pas toujours raison

Pour montrer que l'existence d'un couplage stable n'est pas si évidente, on peut se poser la question de ce qui se passe si l'on adopte une approche de "laisser-faire". Supposons que l'on part d'un couplage parfait arbitraire, et que l'on laisse les couples se reformer librement. C'est-à-dire que s'il existe deux couples  $(h, f)$  et  $(h', f')$ , il peuvent se recombinaison en  $(h, f')$  et  $(h', f)$ , pour peu que  $f' >_h f$  et  $h >_{f'} h'$ .

On peut penser qu'au terme d'un certain nombre "d'échanges", on finisse par aboutir à un couplage stable.

Toutefois, toutes les tentatives pour trouver une mesure qui décroisse lors d'un échange sont vouées à l'échec. Il est en effet possible de proposer une population où ces échanges peuvent se poursuivre à l'infini.

Le contre-exemple est décrit ci-dessous. On a  $H = \{a; b; x\}$  et  $F = \{A; B; X\}$ . On voit que  $X$  et  $x$  sont impopulaires ; mais si le hasard ne les fait jamais se croiser, les couples peuvent se défaire et se refaire à l'infini :

|                          |                          |
|--------------------------|--------------------------|
| $A : b; a; x$            | $a : A; B; X$            |
| $B : a; b; x$            | $b : B; A; X$            |
| $X : \text{peu importe}$ | $x : \text{peu importe}$ |

On note  $\mapsto$  la relation de transition correspondant à une reconfiguration et on souligne à chaque fois les deux individus qui trouvent un intérêt à cette transition. On a alors :

$$\begin{aligned} \{(b, X); (\underline{a}, B); (x, \underline{A})\} &\mapsto \{(b, X); (x, B); (a, \underline{A})\} \mapsto \\ \{(\underline{b}, A); (x, \underline{B}); (a, X)\} &\mapsto \{(b, \underline{B}); (x, A); (\underline{a}, X)\} \mapsto \\ \{(b, X); (a, B); (x, A)\} &\mapsto \dots \end{aligned}$$

## 1.5 Implémentation

Pour passer de la description de l'algorithme de Gale et Shapley à une implémentation exécutable en Java, il y a un peu de travail. Nous ne donnons pas l'implémentation ici, puisque c'est l'objet d'un devoir à la maison (DM). Faisons juste quelques remarques.

### Complexité

Nous avons borné le nombre d'itérations de la boucle principale par  $n^2$ . Avec des préférences bien choisies, il est possible de montrer qu'il faut parfois  $O(n^2)$  itérations pour terminer. En revanche, pour que la complexité de votre implémentation soit effectivement  $O(n^2)$ , il faut veiller à ce que le corps de la boucle puisse être exécuté en temps constant, c'est-à-dire en un temps qui ne dépende pas de  $n$ .

### Opérations nécessaires

La description de l'algorithme nous indique les opérations nécessaires à l'exécution du programme. En particulier il faut être capable d'effectuer en temps constant chacune des opérations suivantes :

- Identifier un homme libre,
- pour un homme donné, trouver la prochaine femme sur sa liste,
- pour une femme, vérifier si elle est fiancée ou non,
- vérifier si elle préfère son fiancé actuel ou son prétendant.

### Files d'attente

On peut vouloir, dans le programme, utiliser une structure de type "file d'attente". On rappelle qu'il s'agit d'une structure qui modélise ce qui se passe dans un magasin : on a un ensemble de clients en attente. On dispose de deux opérations ; d'une part, un nouveau client peut rejoindre la queue, d'autre part, un client peut être servi. Dans ce cas, ce client sera le "plus ancien" dans la queue.

En Java, on peut, par exemple, utiliser la classe `LinkedList<A>`. Celle-ci implémente l'interface <sup>1</sup> `Queue`, c'est-à-dire que si `A` est une classe, et `q` un objet de la classe `LinkedList<A>`, on dispose des méthodes suivantes :

```
boolean q.offer(A b)  ajoute l'objet a à la queue q.
                       Le résultat booléen indique si l'opération a réussi
                       (elle pourrait échouer par exemple par manque d'espace).

A q.poll()           renvoie le premier élément de la file, ou bien null
                       si la file est vide.
```

## 1.6 Problèmes représentatifs

À travers le problème des mariages stables, nous avons voulu illustrer la richesse de ce que peut être l'étude des algorithmes. Dans cette dernière partie, nous voulons motiver la suite de ce cours. Nous allons présenter un certain nombre de problèmes, qui tous semblent apparentés à celui des mariages stables. À chaque fois néanmoins, leur résolution fait appel à une technique algorithmique profondément différente ; nous apprendrons à les traiter au fur et à mesure des chapitres algorithmiques suivants.

### 1.6.1 Ordonnancement d'intervalles

Supposons une ressource, comme une salle informatique qui ne peut être utilisée que par un utilisateur (un cours) à la fois. Les requêtes des utilisateurs arrivent sous la forme d'un intervalle de temps [début; fin]. Deux requêtes qui se recoupent sont donc incompatibles et il faut en choisir une.

Le problème est, étant donné un ensemble fini de requêtes, de trouver un sous-ensemble de requêtes compatibles, qui satisfasse un nombre maximal de requêtes.

Nous verrons que ce problème peut être résolu facilement par un *algorithme glouton*. C'est-à-dire que l'on construira l'ensemble recherché progressivement ; on ajoutera à chaque étape la requête satisfaisant au mieux un critère simple. On verra que même en procédant de cette manière "myope", on arrivera à une solution optimale.

### 1.6.2 Ordonnancement d'intervalles valués

Le problème est un peu plus compliqué lorsque chaque intervalle est porteur d'une *valeur*. Il faut alors déterminer un ensemble de requêtes compatibles qui maximise la somme des valeurs des requêtes satisfaites. Remarquons qu'on retrouve le problème précédent dans le cas particulier où toutes les valeurs valent 1.

Ce problème est donc un peu plus difficile à résoudre et il n'y a pas d'algorithme glouton qui permette de trouver la solution optimale. On pourra la trouver en utilisant la technique dite de *programmation dynamique*.

### 1.6.3 Détection de graphes bipartis

Dans les problèmes précédents, il faut mettre en relation les éléments de deux ensembles donnés a priori (hommes et femmes, requêtes et salles...). Dans le problème des graphes

---

1. On verra précisément au chapitre suivant ce que sont les interfaces Java.

bipartis, on part au contraire d'une relation et on cherche à vérifier si elle détermine deux parties.

On a donc cette fois un graphe, c'est-à-dire un ensemble  $S$  de sommets, et un ensemble  $\mathcal{A} \subset S \times S$  d'arêtes. La question est de déterminer si l'on peut partager  $S$  en deux ensembles distincts  $S_1$  et  $S_2$  tel que chaque arête relie un élément de  $S_1$  à un de  $S_2$  :

$$\forall (x, y) \in \mathcal{A}, (x \in S_1 \wedge y \in S_2) \vee (x \in S_2 \wedge y \in S_1)$$

Nous verrons que ce problème peut être résolu en utilisant une forme adaptée de *parcours de graphe*, une famille d'algorithmes dédiés à ces structures de données.<sup>2</sup>

## 1.7 Exercices

**Exercice 1.7.1** Si l'on se passe de la contrainte d'hétérosexualité, le problème devient celui des *one-sex matchings*, qui à une autre époque était appelé aussi *roommate problem* ou problème des co-locataires. Cette fois on a un seul ensemble d'individus, de cardinal pair. Chaque individu classe tous les autres individus par ordre de préférence. La définition de couplage stable est celle attendue. Solution page 199.

Montrer qu'il n'existe pas toujours de couplage stable. On pourra s'inspirer du contre-exemple donné en 1.4. ◇

**Exercice 1.7.2** Supposons que l'on a un algorithme qui traite le problème précédent de la manière suivante : Solution page 199.

1. L'algorithme vérifie s'il existe une solution stable,
2. s'il en existe, l'algorithme en construit une.

Pourrait-on utiliser un tel algorithme pour remplacer celui de Gale et Shapley ? ◇

On peut consulter le wikipedia anglais "*roommate problem*" pour une description de l'algorithme. La complexité est  $O(n^2)$  comme G-L.

---

2. Il existe un autre problème lié, qui est, étant donné un graphe biparti, d'extraire un couplage de l'ensemble d'arêtes. Ce problème est traité comme une optimisation de flux. Nous n'étudierons pas en détail ces algorithmes dans ce cours.



# Solutions aux exercices

**Solution de l'exercice 1.3.1** Contrairement à la preuve précédente, il n'est pas nécessaire d'analyser le programme. Il suffit d'utiliser le lemme précédent.

Soit  $h$  le partenaire de  $f$  au terme de l'algorithme Gale et Shapley. On sait que  $f = best(h)$ . Soit maintenant un couplage parfait dans lequel  $f$  est couplée à  $h' <_f h$ . Dans ce couplage,  $h$  est couplé avec  $f'$ , et on a forcément  $f' \succ_h f$ . Un tel couplage ne peut donc être stable.  $\diamond$

**Solution de l'exercice 1.7.1** On a quatre individus :  $A, B, C$  et  $X$  avec les préférences suivantes :

$A$  :  $B, C, X$   
 $B$  :  $C, A, X$   
 $C$  :  $B, A, X$   
 $X$  : peu importe

Il est facile de vérifier que le partenaire de  $X$  trouve toujours quelqu'un avec qui quitter  $X$ .  $\diamond$

**Solution de l'exercice 1.7.2** Oui. Si on a un ensemble de filles et un de garçons, avec des préférences sur le sexe opposé, on se ramène à un problème unisexe. Pour cela il suffit de rajouter les (autres) filles à la fin de préférences de chaque fille, et les (autres) garçons à la fin des préférences de chaque garçon.

On peut vérifier que :

- Une solution stable pour cette nouvelle version (*single-gender*) du problème est aussi stable pour le problème originel (*two-gender*).
- Une solution stable du problème *two-gender* est solution du problème *one-gender* correspondant.

L'algorithme trouvera donc une solution, et cette solution est aussi solution du problème originel.  $\diamond$

**Solution de l'exercice 2.6.1** Dans tous les cas, la méthode `finalize()` doit être redéfinie pour Arthur. Il y a deux possibilités.

Soit on redéfinit `finalize()` pour tous les élèves, mais de telle manière à ce qu'il ne se passe quelque chose que pour Arthur. Par exemple :

```
class Eleve extends Personne {
    int promotion;
    String sectionSport ;
    boolean turbulent = false;
    String messageAdieu = "";
    Eleve(String n, Date d; int promo, String sec)
        {nom=n; naissance = d; promotion=promo; sectionSport = sec;}
    Eleve(String n, Date d; int promo, String sec, String message)
```