Table des matières

Préface

Première partie

Leçons

Leçon 1

Exemples de méthodes et outils pour la correction des programmes.

Auteur·e·s: Emile Martinez
Références :

Commentaire 1.1 *Ici correction est pris majoritairement dans le sens «est correct» et non dans le sens de «corriger», à savoir rectifier car c'est souvent dans ce sens là qu'on parle de correction de programme.*

Commentaire 1.2 *On se concentre ici surtout sur les outils théoriques, en estimant que le terme «méthode» de l'intitulé de la leçon laisse penser que c'est sur ceux-là qu'il faut se concentrer*

La conjecture de Syracuse est un problème ouvert de Mathématiques :

La suite u définie par : $\begin{cases} u_0 = a \in \mathbb{N}^* \\ u_{n+1} = \begin{cases} \frac{u_n}{2} & \text{si } u_n \text{ est pair} \\ 3u_n + 1 & \text{sinon} \end{cases} \end{cases}$ finie-t-elle toujours par le cycle $1, 2, 4$?

(toujours $=$ pour tout $a \in \mathbb{N}^*$)

Cela revient à savoir si l'algorithme

Algorithme 1.1 : Syracuse(a)

```
u ← a
tant que u est une nouvelle valeur faire
    si u est pair alors
        u ← u/2
    sinon
        u ← 3u + 1
retourner u
```

renvoie toujours 1, 2 ou 4.

Commentaire 1.3 *C'est un exemple fleuve qui nous permettra d'illustrer toutes les notions autour de la correction et leur non trivialité*

1 Terminaison

Une première question est de savoir si `Syracuse` finit (ne boucle pas à l'infini) sur toute entrée.

Définition 1.1 *Prouver la terminaison d'un algorithme revient à prouver que sur toute entrée il termine*

Remarque 1.1 *On se limite parfois aux entrées valides (pour Syracuse, $a \in \mathbb{N}^*$ par exemple)*

Exemple 1.1

```
Tant que a > 0:
    a = a - 1
```

Termine sur tout entrée si on n'autorise pas a à valoir $+\infty$, sinon ne termine pas sur toute entreé.

Définition 1.2 *Un* **variant** *est une fonction des variables, à valeurs dans $\mathbb{N}$ qui décroit strcitement :*

- *à chaque passage dans la boucles pour les algorithmes itératifs*
- *à chaque appel récursif pour les algorithmes récursifs.*

Exemple 1.2

Algorithme 1.2 : pgcd(a,b)

```
tant que min(a,b) > 0 faire
    si a < b alors
        b ← b − a
    sinon
        a ← a − b
retourner max(a,b)
```

La fonction `pgcd(a,b)` qui calcule le pgcd de a et b pour $a, b \in \mathbb{N}$ admet comme variant $a+b$ (qui est en effet toujours positif, et décroit à chaque fois)

Commentaire 1.4 *Nous réutiliserons plusieurs fois cet exemple du pgcd*

Proposition 1.1 *Si une boucle a un variant de voucle, alors elle s'éxecute un nombre fini de fois. De même pour un algorithme récursif.*

Exemple 1.3 *pgcd(a,b) termine pout tout $(a,b) \in (\mathbb{N}^*)^2$*

Remarque 1.2 *Si un algorithme termine sur toute entrée, il existe toujours un variant, mais il peut être difficile à trouver (ou à prouver)*

Commentaire 1.5 *Il suffit de prendre comme variant le nombre d'étapes de calcul restantes en focntion de l'état de la mémoire*

Exemple 1.4

On définit $ack(n,m)$ *pour* $n,m \in \mathbb{N}$ *par*

$$\begin{cases} ack(0,m) = m+1 \\ ack(n,0) = ack(n-1,1) \\ ack(n,m) = ack(n-1, ack(n,m-1)) \end{cases}$$

Algorithme 1.3 : ack(n,m)

```
si n = 0 alors
    retourner m + 1
sinon
    si m = 0 alors
        retourner ack(n − 1, m)
    sinon
        retourner ack(n − 1, ack(n, m − 1))
```

Il n'est pas immédiat que ack *termine.*

Définition 1.3 *On dit qu'un ordre est un* **ordre bien fondé** *si toute suite décroissante est stationnaire*

Exemple 1.5 $\mathbb{N}$ *avec l'ordre naturel est bien fondé*

Proposition 1.2 *Les ordres produit et lexicographiques d'ordre bien fondés sont bien fondés*

Exemple 1.6 *Un ordre total sur un ensemble fini est bien fondé, donc l'ordre alphabétique est bien fondé (c'est un ordre lexicographique)*

Définition 1.4 *On étend alors la définition du variant aux fonctions à valeurs dans un bon ordre.*

Proposition 1.3 *La propriété 1.1 reste valide avec notre définition étendue du variant*

Exemple 1.7 *Pour ack,* (n,m) *est un variant dans* $\mathbb{N}^2$ *avec l'ordre lexicographique, donc ack termine*

2 Correction partielle

Une autre question pour `Syracuse` est de savoir si on peut tomber sur un autre cycle que 1,2,4 et donc renvoyer autre chose que 1,2 ou 4.

Définition 1.5 *On appelle* **spécification** *d'un algorithme deux propriétés* P_1 *sur les entrées (pré-condition) et* P_2 *sur les sorties (post-condition)*

Exemple 1.8 *Pour Syracuse,* $P_1 :\ll a \in \mathbb{N} \gg$ *et* $P_2 :\ll Syracuse(a) \in \{1,2,4\} \gg$

Définition 1.6 *On dit qu'un algorithme est* **partiellement correct** *si pour toute entrée vérifiant la pré-condition, si l'algorithme termine, la sortie vérifie la post-condition.*

Exemple 1.9 *L'algorithme de l'exemple 1.2 est partiellement correct si la pré-condition est «*$a \in \mathbb{N}, b \in \mathbb{N}$*» et la post-condition «*`pgcd(a,b)` *renvoie le PGCD de a et de b»*

I Correction partielle des algorithmes impératifs

Pour prouver la correction partielle des langages impératifs, on utilise un invariant de boucle.

Définition 1.7 *Un invariant de boucle est une propriété qui est vrai avant la boucle, et si elle est vraie quand on commence un tour de boucle, alors elle l'est quand on le finit.*

Proposition 1.4 *Si un invariant de boucle est valide, alors il est vrai après la boucle, et la condition d'arrêt de la boucle est fausse.*

Exemple 1.10 *Pour pgcd, «$PGCD(a,b) = PGCD(a_0,b_0)$» où a_0 et b_0 sont les valeurs initiales de `a` et `b`, est un invariant valide.*

A la fin de l'exécution, on a donc $\min(a,b) = 0$ *et* $PGCD(a,b) = PGCD(a_0,b_0) = PGCD(\min(a,b),\max(a,b)) = PGCD(0,\max(a,b)) = \max(a,b)$. *D'où l'assertion de l'exemple 1.9*

II Correction partielle des algorithmes récursifs

Commentaire 1.6 *On ne présente souvent pas la correction partielle des algorithmes récursifs, en se contentant de la correction totale directement. Néanmoins on présente quand cela, car cela illustre très bien la manière de raisonner quand on veut coder en récursif (et que tout est vrai)*

Principe 1.1 *Pour prouver la correction partielle d'un algorithme récursif, on vérifie que si*

- *la pré-condition est vérifiée*

, alors

- *on ne fait que des appels récursifs où les arguments vérifient la pré-condition*
- *Si la post-condition des appels récursifs est vérifiée, alors celle de notre appel est vérifiée*

Théorème 1.1 *Si le principe 1.1 est respecté, alors l'algorithme est partiellement correct.*

Exemple 1.11 *exp(a,n) pour $a, n \in \mathbb{N}$ renvoie a^n.*

Algorithme 1.4 : exp(a, n)

```
si n = 0 alors
    retourner 1
sinon
    x = exp(a, n)
    si n est pair alors
        retourner x * x
    sinon
        retourner x * x * a
```

Précondition : «`a` est un flottant et `n` un entier».
Postcondition : `exp(a,n)` $= a^n$.
Alors, la pré-condition est valide à chaque appel, et comme $a^n = a^{\lceil n \rceil} * a^{\lfloor n \rfloor}$, le principe 1.1 est respecté dans tous les cas donc l'algorithme est partiellement correct.

3 Correction

La conjecture de Syracuse dit donc que notre fonction Syracuse termine, et quand elle termine est correcte (i.e. partiellement correcte).

I Cas général

Définition 1.8 *Quand un programme termine sur toute entrée valide et est partiellement correct, on dit qu'il est correct, ou encore totalement correct.*

Exemple 1.12

Algorithme 1.5 : fusion(L_1, L_2)

```
res ← []
i, j ← 0
tant que i < |L1| et j < |L2| faire
    si L1[i] < L2[j] alors
        res.ajouter(L1[i])
        i ← i + 1
    sinon
        res.ajouter(L2[j])
        j ← j + 1
Ajouter le reste de L1 et de L2 à res
retourner res
```

Algorithme 1.6 : tri_fusion(L)

```
n ← |L|
si n ≤ 1 alors
    retourner L
L1, L2 ← partionner(L)
retourner fusion( tri_fusion(L1), tri_fusion(L2) )
```

Developpement ?? Correction totale de tri_fusion.

Néanmoins, ce n'est pas toujours facile. La conjecture de Syracuse est toujours un problème ouvert. Et c'est parfois même pire.

Théorème 1.2 *La correction partielle et la terminaison sont indécidables.*

Developpement ?? Preuve du théorème 1.2

II Cas des algorithmes récursifs

Dans le cas des algorithmes récurifs, on fait régulièrement la correction totale directement.

Proposition 1.5 *SI $(E, \preceq)$ est un ordre bien fondé, alors toutes parties non vides à un élément minimal (plus grand que personne)*

Théorème 1.3 *Soit $(A, \preceq)$ un ensemble muni d'un ordre bien fondé et $\mathcal{P}$ une propriété sur A, alors* $\forall x \in A, (\forall y \in A, y \preceq A \Rightarrow \mathcal{P}(y) \Rightarrow \mathcal{P}(x)) \Rightarrow \forall x, \mathcal{P}(x)$

Remarque 1.3 *Cela étend le principe de réccurence forte sur $\mathbb{N}$*

L'ordre bien fondé nous donne alors le variant et la propriété l'invariant. On montre alors la terminsaison et la correction partielle en même temps.

Exemple 1.13 *Dans l'exemple 1.2, la propriété $\mathcal{P}(n)$: «exp(a,n)= a^n» vérifie les bypothèses du théorème 1.3. Donc, $\forall n$, `exp(a,n)` $= a^n$, et ce pour tout $a \in \mathbb{N}$*

Commentaire 1.7 *Ici on suppose que exp(a,n)= a^n veut dire que exp termine et renvoie a^n*

4 Outils pratiques

- ⋆ Typer : Le fait d'utiliser un typage fort comme en OCamL permet d'éviter beaucoup d'erreurs bêtes
- ⋆ Programmer défensivement en utilisant la bibliothèque assert.h permet de vérifier qu'à un moment donné du code, les hypothèses (ou les invariants) sont satisfaits (et pas seulement une erreur aléatoires parmi 1000 lignes)
- ⋆ Faire des test tout au long de la programmation, en utilisant au maximum la modularité, pour détecter le plus tôt possible les erreurs.
- ⋆ Utiliser des logiciels comme GDB ou valgrind pour détecter les fuites mémoires, ou l'inspecter au cours du programme.
- ⋆ Commentez ! C'est primordial pour déclarer la spécification des fonctions et rendre le code compréhensible, donc déboguable.

Commentaire 1.8 *On pourrait développer cette partie également en parlant plus longuement des tests (cf lecon 3) qui est un autre type d'outils pour éprouver la correction d'un programme. Mais ca nous emmène un peu loin pour cette leçon (il faut bien faire des choix)*

Leçon 2

Paradigmes de programmation : impératif, fonctionnel, objet. Exemples et applications.

Auteur·e·s: Emile Martinez
Références :

Programmer, c'est mettre en relation un cahier des charges et des instructions compréhensibles par la machine.

Définition 2.1 *Un* **paradigme de programmation** *définit la façon d'approcher la programmation informatique.*

Suivant le contexte il en existe plusieurs que nous verrons ici.

1 La programmation impérative

C'est la plus classique.

Définition 2.2 *La* **programmation impérative** *consiste à donner une suite d'instructions, chacune ayant pour seul effet de modifier l'état du programme (la mémoire, la valeur des variables, l'endroit où on en est etc...).*

Remarque 2.1 *Ainsi, dans la programmation impérative, il n'existe pas de valeurs de retour. Si on en veut une, il faut écrire la valeur que l'on veut dans la mémoire.*

Remarque 2.2 *Informellement, programmer impérativement, c'est utiliser des variables, des affectations, des tableaux, des boucles for et while, etc...*

Exemple 2.1 *La majorité du code en python est impératif*

```
x = 1
y = x + 3
while (x != y):
    print(y)
    y -= 1
```

Ce programme écrit 1 dans la case mémoire de x, puis y accède pour écrire 4 dans celle de y, puis écrit la valeur de y dans l'espace mémoire dédié à l'affichage, etc...

Remarque 2.3 *Impératif est pris ici dans son sens courant (en informatique). Une autre définition d'impératif est qu'on dit exactement ce que la machine doit faire (ex : l'assembleur), ce qui s'opose alors au déclaratif (comme SQL). Mais cette notion est à degré (dans tous langage il y a une marge plus ou moins grande pour la machine) et n'est pas nécessairement celle à laquelle on pense quand on pense à de la programmation impérative (même si les deux sont très liées).*

2 Programmation fonctionnelle

Définition 2.3 *La* **programmation fonctionnelle** *consiste à composer le programme de fonctions (au sens mathématiques), et de récupérer la valeur de retour.*
Les changements d'état ne peuvent pas être représentés par des évaluations de fonctions, donc la programmation fonctionnelle ne les admet pas. On dit que les structures de données fonctionnelles sont immuables.

Exemple 2.2 *let max (x,y) = if x > y then x else y*
(fonction de type int∗int −> int)

Informellement, programmer en fonctionnel, c'est considérer les fonctions comme des objets comme les autres, et n'avoir que des structures de données immuables.

Remarque 2.4 *Un argument d'une fonction ou la valeur de retour d'une fonction peut être une fonction. C'est ce que l'on appelle la programmation d'ordre supérieure.*

Définition 2.4 *La* **curryfication** *est la transformation d'une fonction à plusieurs arguments en une fonction à un argument qui retourne une fonction sur le reste des arguments*

Exemple 2.3 *On peut transformer la fonction max de l'exemple 2.2 en la fonction*
let max x y = if x > y yhen x else y de type int −> int −> int

Exemple 2.4 *Si, sur l'exemple 2.3, on veut que max puisse comparer des éléments sur lesquels on ne connait pas l'ordre, on peut en faire une fonction d'ordre supérieur en lui fournissant une fonction de comparaison :*
let max compar x y = if compar x y then x else y de type ('a −> 'a −> bool) −> 'a −> 'a −> 'a
On a alors

- *max (fun x y −> x > y) qui calcule le max*
- *max (fun x y −> x > y) 3 qui est une fonction de type int −> int renvoyant le maxmimum de son argument et 3*
- *max (fun x y −> x < y) qui calcule le min*

Remarque 2.5 *La puissance du fonctionnel vient de la récursivité*

3 Programmation orientée objet

I Obtenir de la modularité

Définition 2.5 *Une* **classe** *est un ensemble de types de données appelés* **attributs** *et de* **fonction** *appelées méthodes.*
Un **objet** *est un représentant d'une classe. C'est un espace en mémoire contenant les valeurs des différentes attribut, les méthodes étant communes à tous les objets.*

Exemple 2.5

```
class Noeud:
    def __init__(self, x):
        self.valeur = x
    def afficher(self):
        print(self.valeur)
    def est_egal(self, autre):
        return self.valeur == autre.valeur
a = Noeud(5)
a.afficher()
```

Ici Noeud est une classe, valeur un attribut de la classe Noeud, afficher et est_egale des méthodes de la classe Noeud et a un objet (représentant) de la classe Noeud

Définition 2.6 *La* **programmation orienté objet** *consiste à utiliser des classes et des objets de ces classes quand on programme.*

Remarque 2.6 *Une utilisation massive des classes et de permettre de la modularité : on peut avoir une interface entre un type abstrait et son utilisation, rendant l'utilisation et la structure implémentant le type indépendant.*

Exemple 2.6 *En python, le package numpy propose les objets numpy.array que l'on crée via la commande a = numpy.array([...]). Un tel objet possède des attributs comme sa taille (a.size) mais aussi des méthodes tq a.sort()). Cette classe implémente des tableaux de taille fixe et de nombreuses méthodes dessus. On peut les utiliser en ne comprenant rien à comment elles fonctionnent, seulement ce qu'elles font, mais on peut aussi les réimplenter sans rien changer à l'utilisation de ces tableaux par des millions de personnes.*

II Pour résoudre un problème

Une autre utilité de la programmation orienté objet, et de représenter un problème avec ses différents objets que l'on fera intéragir entre eux.

Exemple 2.7 *Sur l'exemple 2.5, on peut rajouter la classe Arbre contenant des noeuds.*

```
class Arbre:
    def __init__(self, n, liste_arbre):
        self.noeud = n
        self.fils = liste_arbre
    def afficher(self):
```

```
        n.afficher()
        for x in self.fils:
        x.afficher()
```

4 Dans la vraie vie ?

I Le multiparadigme

Dans la vraie vie, la plupart des langages de programmation implémente plusieurs paradigmes. En effet, python, comme C ou Ocaml, permettent de faire des boucles while, de faire des tableaux, de faire des structures et des fonctions récurisves, et même les fonctions d'ordre supérieur dans une certaine mesure.

On appelle cela le multiparadigme. Néanmoins, certains langages sont plus adapatés à certains paradigmes, eux-mêmes plus adaptés à certaines contraintes.

Des langages comme C, C++, Fortran, python, Java sont des langages impératifs, quand Haskell, ML, OcamL sont fonctionnels. De plus, python, C++ sont orientés objets.

II Comparaison des paradigmes

- ⋆ Pour des structures récursives comme des arbres (ou des graphes peu denses), le paradigme fonctionnel est approprié
- ⋆ Le paradigme fonctionnel offre également élégance et lisibilité au code, avec moins d'instructions ≪superflues≫
- ⋆ Le caractère intrinséquement modulaire et sans effet de vord le rend aussi plus facile à tester et sécuriser : C'est en OcamL (en Coq) qu'est implémenté CompCert, un compilateur C vérifié.
- ⋆ La programmation impérative est beaucoup plus proche de la machine et rend donc la compilation plus simple, et le développement intelligent potentiellement plus efficace.
- ⋆ Il est aussi très performant pour des structures de données séquentilles et des accès ≪aléatoires≫ à des données. Par exemple représenter une matrice, en faire des multiplications, etc... paraît beaucoup plus simple en C qu'en OcamL.

  Exercice 2.1 *Implémenter un tas min en C et en OCamL*

- ⋆ L'orienté objet est quant à lui de plus haut niveau et repose souvent sur d'autres paradigmes plus bas niveau.
- ⋆ Il est souvent utilisé pour représenter des situtations complexes grâce à sa modularité

  Exercice 2.2 *Implémenter les classes représentant un personnage de jeu vidéo, ses objets, ses compétences, etc...*

III Et SQL ?

Il existe néanmoins bien d'autres paradigmes, comme par exemple le paradigme logique, où seul le résultat est présenté par le code, et non la manière de l'obtenir. C'est par exemple le cas du SQL pour les bases de données, où l'exécution n'est pas dicté par la requête, seul son sens l'est, laissant le SGBD se charger du déroulement.

Leçon 3

Tests de programme et inspection de code.

Auteur·e·s: Emile Martinez, Malory Marin
Références :

1 Introduction

I Qu'est-ce qu'un test

Tester est un anglicisme pour le mot français essayer (ou éprouver). Me soumettant à la folie anglomane ambiante je garderai ce mot, soucieux de ma cohérence avec le monde extérieur.

Définition 3.1 **Tester** *un programme consiste à essayer d'y trouver des erreurs*

Remarque 3.1 *On ne cherche pas ici à prouver directement que le programme est correct, mais à prouver qu'on n'arrive pas à se rendre compte qu'il est incorrect.*

II Données de tests

Définition 3.2 **Une donnée de test** *est un couple (valeur d'entrée, valeur de sortie), où à l'évidence, le deuxième élément représente la valeur de sortie quand la fonction est appelée sur le premier élément.*

Définition 3.3 **Un jeu de données de test** *(ou jeu de tests) est alors un ensemble de tels couples, permettant de vérifier la validité du programme sur certaines entrées.*

Remarque 3.2 *Certaines sorties (attendues) peuvent être des erreurs.*

Exemple 3.1 *Un jeu de tests pour une fonction calculant le pgcd peut être* $\{((1,2),1),((-3,6),3),((0,0),0),((2,2.45),\textit{Erreur de type})\}$

III Types de tests

Il existe deux types de tests :

— Les tests en boites noires : On ne connait pas le code de la fonction, on peut simplement l'appeler.

— Les tests en boites blanches : On connait le code et on génère un jeu de test en fonction.

2 Tests en boîtes noires

I Caractéristiques

Pour un test en boîtes noires, comme on ne connait pas le code, il faut tester beaucoup de données. Idéalement, toutes, mais cela se trouve souvent impossible.

Exemple 3.2 *On peut tester toutes les valeurs d'une fonction qui implémente une fonction booléenne mais pas celles de notre fonction calculant le pgcd de deux nombres*

Viennent alors deux problèmes : Générer suffisament de données d'entrées et effectuer le test suiffisament rapidement.

II Générer des données d'entrée

Dans de nombreux cas, ne pouvant pas essayer toutes les données d'entrées, on va devoir faire des choix.

Idée 3.1 *La première approche consisterait à générer des valeurs aléatoires dans un domaine, et espérer en prendre suffisamment pour que cela fonctionne.*

Principe 3.1 *Une approche plus maline, à partitionner le domaine, puis à appliquer l'approche naïve sur chaque domaine.*

Exemple 3.3 *Pour le calcul de pgcd(a,b), on peut parititionner le domaine d'entrées en comparant a, b, et 0, (avec donc 6 domaines : $a <= 0 <= b$, $0 <= a <= b$, $0 <= b <= a$, $a <= b <= 0$, $b <= a <= 0$, $b <= 0 <= a$)*

Remarque 3.3 *On se contente souvent de prendre un seul test par classe.*

Remarque 3.4 *Le choix du partitionnement est arbitraire et doit donc être fait suivant la manière d'approcher le problème*

Une fois cela fait, il est très commun que les erreurs puissent venir des cas limites.

Principe 3.2 *On essaye alors de se placer au limites des domaines, et de vérifier spécifiquement ces cas là.*

Exemple 3.4 *Pour l'exemple précédent, on testerai les cas d'égalité : $0 <= a = b$, $a = 0 = b$, $a = 0 <= b$ etc... (en effet par exemple, le cas 0, 0 est différent des autres, la valeur pouvant être $+\infty$ ou 0 suivant les définitions).*

III Utiliser les caleurs de sorties efficacement

Néanmoins, maintenant que l'on a les valeurs d'entrées, il faut pour avoir notre jeu de tests avoir également les valeurs de sorties.

Exemple 3.5 *si l'on veut générer la sortie du pgcd sur des entrées que l'on a pris au hasard, il faut connaitre déjà le pgcd. Quel intérêt d'avoir notre fonction alors si on a déjà une fonction qui le fait*

On a alors plusieurs méthodes :

1. Générer un jeu de tests à la main
2. Utiliser un programme moins performant mais que l'on sait correct.

   Exemple 3.6 *Si on calcule le pgcd par soustraction successives, on peut tester en calculant le pgcd en testant tous les nombres inférieurs à a et a garder le plus grand qui divise a et b.*

3. Ne pas calculer la réponse mais simplement vérifier que la réponse fourni est correcte.

   Exemple 3.7 *Si on a un programme qui nous donne la décomposition en facteurs premiers d'un nombre, il nous suffit de tester la primalité de chaque sortie et de vérifier que leur produit fait l'entrée.*

Exemple 3.8 *Si on a un algorithme performant effectuant le produit de matrice, on peut :*

1. *Créer à la main quelque petite matrice et faire leur produit pour vérifier que tout fonctionne*
2. *Comparer avec l'algorithme naif du calcul de produit de matrice*
3. *Vérifier de manière probabiliste que le résultat est bien le bon.*
   Developpement ?? *Vérification probabiliste du produit de matrice.*

3 Tests en boite blanche

I Graphe de flot de contrôle

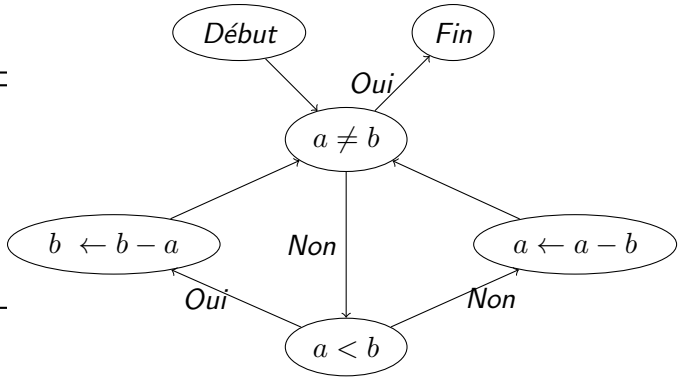
Pour un test en boîte blanches, on connaît le code, et on va vouloir générer des données d'entrées en fonction de ce code là.

Pour cela, on extrait du code le graphe de flot de contrôle.

Définition 3.4 *Le graphe de flot de contrôle est un graphe où chaque boîte contient des lignes de codes, les boites sont reliés si on peut exécuter l'une puis l'autre.*

Exemple 3.9 *On prend l'exemple du pgcd pour $a, b \in \mathbb{N}^*$*

```
tant que a ≠ b faire
    si a < b alors
        b ← b − a
    sinon
        a ← a − b
retourner a
```



II Utilisation du graphe

On essaye alors de générer un jeu de données qui parcourt une bonne partie du graphe.
Par exemple, un jeu couvrant :

— Tous les nœuds (On veut un jeu de tests tel que tous les tests pris ensemble, chaque nœud du graphe est parcouru au moins une fois).

— Tous les arcs

— Tous les chemins

Commentaire 3.1 *Si on veut aller plus loin, on peut aussi rajouter toutes les conditions décisions, toutes les p utilisations, etc... On peut alors rajouter un exercice proposant de montrer la hiérarchie entre ces tests.*

Exemple 3.10 *Sur l'exemple , $\{((1,1),1),((1,3),3)\}$ ne couvre pas tous les nœuds quand $\{((1,3),3),((3,1),3)\}$ couvre tous les nœuds et tous les arcs mais pas tous les chemins.*

Remarque 3.5 *Quand il y a une boucle, tous les chemins peut-être un critère infini. On peut alors se limiter aux chemins d'une certaine taille.*

Remarque 3.6 *Parfois les critères sont insatisfiables.*

Remarque 3.7 *Aucun de ces critères ne garantissent la validité d'un algorithme. Elles permettent simplement de vérifier que notre jeu de tests n'est pas trop lacunaire.*

Developpement ?? Intérets et insuffisances de ces critères

III Test exhaustif de condition

Idée 3.2 *Une autre approche consiste à avoir un jeu qui satisfait ou invalide toutes les conditions de toutes les manières possibles.*

Exemple 3.11 *On voit l'utilité sur l'exemple suivant :*

```
int max(int a, int b){
  if(a > b || a == 500){
        return a;
    else
        return b;
}
```

Pour détecter le problème (que max ne calcule pas le $\max$), il faut des tests où on mets à vrai le premier if à cause de $a == 500$, donc des tests où a vaut 500.

4 Pratiques pour éviter d'avoir à déboguer

Dans la pratique, de bonne pratique de code sont très efficace pour éviter de passer trop de temps à debugguer son code.

⋆ Compiler avec -Wall (activant tous les warnings, donnant beaucoup de bugs stupides)

⋆ Respecter la ponctuation et éventuellement utiliser un linter (de manière à rendre lisible le code par d'autres personnes)

⋆ Faire de la programmation défensive en utilisant assert par exemple

Leçon 4

Principe d'induction

Auteur·e·s: Emile Martinez
Références :

1 Principe

I Definition

Définition 4.1 *$(\mathcal{B}, (f_i))$ est une signature sur X si :*

- *$\mathcal{B} \subset X$ (appelé cas de base)*
- *$f_i : X^{\alpha(i)} \to X$ appelé constructeurs, d'arité $\alpha(f_i)$ avec f_i injectif et $\Im(f_i) \cap \Im(f_j) =$ et $\Im(f_i) \cap \mathcal{B} =$ pour tout $i \neq j$*

Commentaire 4.1 *L'intérêt de cette définition est d'être rigoureux. Néanmoins le programme demande de se contenter de présenter des choses proches de ce que l'on rencontrera, or là c'est plus ou moins la version formelle de ce qui se passe en OcamL. Donc meme si ce cadre théorique n'est pas explicitement au programme, sa présence là est justifiée par sa proximité avec OcamL*

Remarque 4.1 *On se contente souvent de dire que les constructeurs existent, sans donner leur définition. (de même pour les cas de bases, et pour X)*

Exemple 4.1 *On prend une constante Z et un constructeur d'arité 1 $Succ$*

Exemple 4.2 *On peut prendre les constructeurs $\oplus$, $\ominus$ $\otimes$ et $\otimes$ avec $\alpha(\oplus) = 2$, $\alpha(\ominus) = 1$ et $\alpha(\otimes) = 2$ et comme cas de bases $\mathbb{N}$.*

Définition 4.2 *Un ensemble inductif est définit par une signature $(\mathcal{B}, (f_i))$:*

1. *Le plus petit ensemble contenant $\mathcal{B}$ et stable par tous les f_i (définition par le haut)*
   ou de manière équivalente
2. *$\bigcup T_i$ où $T_0 = \mathcal{B}$ et $T_{n+1} = T_n \cup \bigcup_i f_i(T_n^{\alpha(f_i)})$ (définition par le bas)*

Exemple 4.3 *L'ensemble inductif définit par l'exemple 4.1 peut être une définition des entiers naturels*

Exemple 4.4 *Exemple : L'ensemble inductif $\mathcal{A}_{simp}$ définit par l'exemple 4.2 est l'ensemble des expressions arithmétiques simplifiées.*

Remarque 4.2 $\oplus(1,1) \neq \otimes(1,2) \neq \otimes(2,1)$. *On s'intéresse à l'expression et non au résultat.*

Développement ?? Validité de la construction d'un ensemble inductif (remarque 4.1 et définition 4.2)

II Induction structurelle

On prendra maintenant $(\mathcal{B}, (f_i)_{i\in I})$ une signature.

Proposition 4.1 *Soit E un ensemble inductif construit par $(\mathcal{B}, (f_i))$*

Alors la donné de fonction g_i (avec $\alpha(g_i) = \alpha(f_i)$ et $\Im(g_i) \subset Dom(g_j)$) et de $f(b)$ pour $b \in \mathcal{B}$ définit une unique fonction f sur E ayant la propriété :

$$\forall i \in I, \forall x_1, \ldots, x_{\alpha(i)} \in X,\ f\left(f_i(x_1, \ldots, x_{\alpha(i)})\right) = g_i\left(f(x_1), \ldots, f(x_{\alpha(i)})\right)$$

Commentaire 4.2 *Cette propriété est fondamentale, et justifie en partie (parce que la il ne faut pas un matching complet mais directement les constructeurs) la bonne définition des fonctions OcamL. D'où d'ailleurs les références à cette propriété dans la suite.*

Exemple 4.5 *Sur $\mathcal{A}_{Simp}$, on peut définir $eval : \mathcal{A}_{Simp} \to \mathbb{N}$ par*

- $eval(a) = a$ *pour* $a \in \mathbb{N}$
- $eval(\oplus(a,b)) = eval(a) + eval(b)$
- $eval(\otimes(a,b)) = eval(a) \times eval(b)$
- $eval(\ominus(a)) = -eval(a)$

Théorème 4.1 (Induction structurelle) *Soit E l'ensemble inductif définit par $(\mathcal{B}, (f_i)_{i\in I})$, et $\mathcal{P}$ une propriété définie pour tout $x \in E$.*

Alors $\begin{cases} (i) & \forall b \in \mathcal{B}, \mathcal{P}(b) \\ (ii) & \forall i \in I, \forall x_1, \ldots, x_{\alpha(i)}, (\forall j, \mathcal{P}(x_j)) \implies \mathcal{P}\left(f_i(x_1, \ldots, x_{\alpha(i)})\right) \end{cases}$

Remarque 4.3 *La récurrence est un cas particulier dans le cas de la définition des entiers par l'exemple 4.3*

Exemple 4.6 *On montre par induction structurelle que $eval(e)$ pour $e \in \mathcal{A}_{simp}$ est multiple du pgcd des constantes apparaissant dans e*

Définition 4.3 *Un ordre bien fondé est un ordre où toute partie non vide admet un élément minimal (plus grand que personne)*

Proposition 4.2 *L'ordre produit et l'ordre lexicographique d'ordres bien fondés sont bien fondés*

Exemple 4.7 $\mathbb{N}$ *avec l'ordre naturel est bien fondé, donc* $\mathbb{N}^k$ *avec l'ordre produit ou lexicographique aussi.*

Théorème 4.2 *Soit* $(A, \preceq)$ *un ensemble muni d'un ordre bien fondé et* $\mathcal{P}$ *une propriété sur* A*, alors* $\forall x \in A, (\forall y \in A, y \preceq A \Rightarrow \mathcal{P}(y) \Rightarrow \mathcal{P}(x)) \Rightarrow \forall x, \mathcal{P}(x)$

Remarque 4.4 *Cela étend le principe de réccurence forte.*

Définition 4.4 *Soit* E *l'ensemble inductif défini par* $(\mathcal{B}, (f_i)_{i \in I})$
On définit l'ordre structurel $\leq_s$ *sur* E *comme la clôture transitive réflexive de* $x_j \leq_s f_i(x_1, \ldots, x_{\alpha(i)})$

Proposition 4.3 $\leq_s$ *est une relation d'ordre bien fondé*

Corollaire 4.1 *On peut alors réécrire l'induction structurelle comme une induction sur l'ordre structurelle*

Commentaire 4.3 *Si on manque de place, on peut mettre les ordres bien fondés en prérequis (mais alors écrire la formule dans le corollaire)*

III En OCaml

Commentaire 4.4 *On essaye de faire au maximum le lien entre les définitions formelles et OcamL*

Syntaxe 4.1 *En OcamL on peut créer un type représentant un ensemble inductif avec cette syntaxe :*

```
type t = Casdebase1 | Casedebase2 | ...
       | Constructeur1 of type11 *type12 * ....
       | Constructeur2 of type21 * tpye22 * ...
```

Où :

- `Casdebase` *peut soit être un type déjà défini d'OcamL, soit une constante (nom commençant par une majuscule)*
- `Constructeur` *est une étiquette commencée par une majuscule*
- `typei` *est est un type OCamL (pouvant contenir* `t`*)*

Exemple 4.8 *Pour définir les entiers de l'exemple 4.3, on peut écrire*

```
type entier = Zero | Succ of entier
```

Syntaxe 4.2 *Pour gérer les types, on peut utiliser le filtrage comme pour les listes.*

Exemple 4.9 *Pour l'addition sur notre type entier, on peut écrire :*

```
let rec ajouter x y = match y with
  | Zero -> x
  | Succ(z) -> ajouter (Succ(x)) z
```

Exemple 4.10 *La validité de cette définition vient de la propriété 4.1*

2 Structures de données inductives

I Les listes chaînées

En OcamL on peut définir des listes d'entier simplement chaînés par

```
type liste = V | Cons of int * liste
```

Ainsi, une liste c'est soit une liste vide, soit un entier et le reste de la liste.

Remarque 4.5 *Ici `V` est le cas de base, et `Cons` le constructeur. $Cons$ est défini sur $\mathbb{N} \times \{\text{ensemble des listes}\}$ et non $\{\text{ensemble des listes}\}^2$. Cela est un raccourci d'OCamL, où en réalité on définit un constructeur pour chaque premier argument, et donc on construit non pas $Cons(x, l)$ mais $Cons_x(l)$.*

Remarque 4.6 *Cela correspond au type `int list` d'OCamL ;-)*

Exercice 4.1 *Définir inductivement la taille d'une liste chaînée*

II Les arbres binaires

Exercice 4.2 *Définir inductivement la taille d'une liste chaînée.*

Définition 4.5 (Arbre binaires) *Soit A un ensemble. On définit de manière inductive les arbres binaires sur A par :*

- *l'arbre vide E (cas de base)*
- *si $e \in A$ et g et d sont des arbres binaires, alors $Noeud(e, g, d)$ est un arbre binaire.*

Implémentation 4.1 *Ce qui en OCamL nous donne `type 'a arbre = E | Noeud of 'a * 'a arbre * 'a arbre`*

Exemple 4.11 *La hauteur d'un arbre binaire se calcule alors inductivement par*

```
let rec hauteur arb = match arb with
  | E -> 0
  | Noeud(e, g, d) -> 1 + max (hauteur g) (hauteur d)
```

Exercice 4.3 *Prouver par induction structurelle la terminaison de la fonction hauteur*

Exercice 4.4 *Donner la définition inductive de la taille d'un arbre binaire (son nombre d'éléments)*

III Les arbres généraux

Définition 4.6 *Un arbre général est un noeud (la racine) et une liste d'arbre (ses fils)*

On voudrait alors définir le type `arbre` par (pour les arbres d'entier)

```
type arbre = Noeud of int * arbre_liste
```

Il faut alors définir le type `arbre_liste`, par

```
type arbre_liste = Vide | Cons of arbre * arbre_liste
```

On remarque que chaque type a besoin de l'autre pour exister. On écrit alors

```
type arbre = Noeud of int * arbre_liste
and type arbre_liste = Vide | Cons arbre*arbre_liste
```

Remarque 4.7 *On passe souvent cela sous le tapis grâce au polymorphisme qui définit des manières de construire des types et non des types directement*

Commentaire 4.5 *Dans la défense de plan, on peut parler ici des différentes définition des arbres (par coinduction, avec une infinité de constructeur (pour chaque $k \in \mathbb{N}^*$ d'arité k, pour les arbres à k fils), par un constructeur $Ajout\_fils$ d'arité 2 où le premier argument est l'arbre sans son premier fils, et le dernier argument le premier fils (on en déduit les cas de bases))*

3 Ensembles inductifs

I Formules propositionnelles

Définition 4.7 (formule propositionelle) *Soit V un ensemble de variables et la signature*

- *$\mathcal{B} = \{\top, \bot\} \cup V$*
- *le constructeur $\neg$ d'arité 1*
- *les constructeurs $\vee$, $\wedge$ et $\rightarrow$ d'arité 2 (en forme infixe)*

Les formules propositionnelles forment l'ensemble inductif défini par cette signature

Exercice 4.5 *Défini inductivement le nombre de variables présent dans la formule*

Définition 4.8 *On appelle valuation (ou distribution de vérité) toute fonction $v : V \rightarrow \{0, 1\}$*

Définition 4.9 (Evaluation d'une formule) *Soit v une valuation. La fonction d'évaluation d'une formule $[.]_v : F \rightarrow \{0, 1\}$ se définit inductivement par*

- $[\top]_v = 1$
- $[\bot]_v = 0$
- $[x]_v = v(x)$ *pour* $x \in V$

- $[\neg F]_v = 1 - [F]_v$
- $[f_1 \wedge f_2]_v = \min([f_1]_v, [f_2]_v)$
- $[f_1 \vee f_2]_v = \max([f_1]_v, [f_2]_v)$
- $[f_1 \to f_2]_v = \begin{cases} 0 & \textit{si } [f_1]_v = 0 \textit{ et } [f_2]_v = 1 \\ 1 & \textit{sinon} \end{cases}$

Exercice 4.6 *Définir l'équivalence entre formules et montrer par induction structurelle que, à équivalence près, on peut ne garder que les constructeurs $\neg$ et $\vee$*

II Langages

Commentaire 4.6 *Ici comme exemple on aurait aussi pu prendre les langages réguliers, cela est plus pertinent comme construction, mais nécessite d'avoir déjà les définitions de bases sur les langages. Et de plus trouver des inductions parait non trivial*

Définition 4.10 *Soit Σ un ensemble (appelé alphabet) fini et non vides d'éléments (appelés lettres). On définit inductivement l'ensemble des mots sur Σ, Σ^* par la signature :*

— *ε (le mot vide) comme cas de base*

— *Si $a \in \Sigma$ et $w \in \Sigma^*$, $wa \in \Sigma^*$*

Remarque 4.8 *On aurait aussi pu prendre comme définition de Σ^*, $\bigcup_{n \geq 0} \Sigma^n$*

Définition 4.11 *La concaténation de deux mots $w_1, w_2 \in \Sigma^*$ se définit inductivement comme :*

— $concat(w_1, \varepsilon) = w_1$

— $concat(w_1, w_2.a) = concat(w_1, w_2).a$

Exercice 4.7 *Montrer par induction structurelle que $concat(w_1, concat(a, w_2)) = concat(w_1.a, w_2)$*

Leçon 5

Implémentations et applications des piles, files et files de priorité

Auteur·e·s: Emile Martinez
Références :

> **Commentaire 5.1** *Comme il y a beaucoup à dire, et que le titre ne mentionne pas d'aspects théoriques sur les structures, on suppose ici les structures déjà définie dans le cours. Ainsi, on rappelera uniquement, pour uniformisation les méthodes caractéristiques. On se concentre ainsi sur les implémentations et les applications.*

1 Les piles

Rappel : Une pile est une structure de données avec les méthodes `vide`, `est_vide`, `empiler`, `depiler`.

I Implémentation par listes

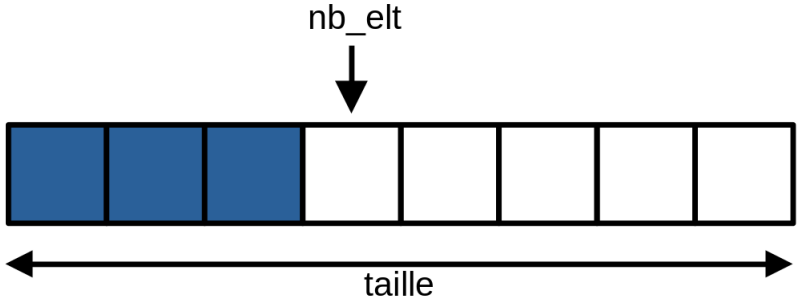
Cette manière est immédiate et donc naturelle.

Exercice 5.1 (Au tableau avec participation des élèves) *A quoi correspondent les opérations de bases de la pile sur une liste (à l'oral et éventuellement avec le dessin d'une liste simplement chaînés pour montrer les modifications).*

Les listes étant naturelles en OCaml, nous les implémenterons de cette manière en Ocaml. Néanmoins cette pile est immuable.

II Implémentation par tableaux

La deuxième manière est en utilisant un tableau.



Idée 5.1 *Pour cela, on utilise un indice `nb_elt` qui nous indiquera à quelle case du tableau correspond le sommet de la pile, les cases précédentes du tableau contenant les autres éléments empilés.*

Implémentation 5.1 *On doit soit alors utiliser un tableau de taille fixe (et donc connaître à l'avance le nombre max d'éléments dans la pile)*

Exercice 5.2 *Quels sont les inconvénients de cette implémentation ? (le fait qu'il faut soit utilisé un tableau dynamique, soit connaître à l'avance la taille maximale de la pile)*

Implémentation 5.2 *en C en TP avec les déclarations déjà faites, `vide` également, et le corps des autres fonctions à remplir*

> **Remarque 5.1** *Les problèmes sont exactement similaires à ceux de l'implémentation d'une liste par un tableau. Cela vient de la proximité entre une liste et une pile.*

Exercice 5.3 *Laquelle de ces deux implémentations est mutable ? Immuable ?*

III Complexité

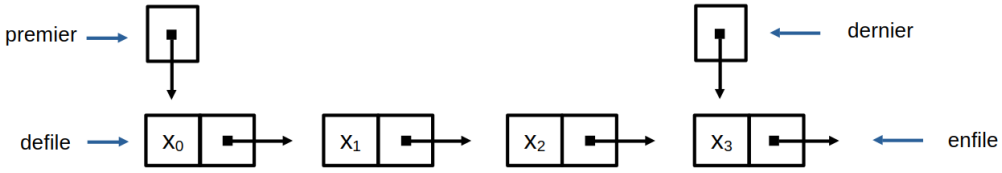
Toutes ces opérations sont en $O(1)$. De plus sa complexité spatiale n'est, si l'implémentation est bien effectuée, que O(hauteur maximale de la pile)

2 Les files

Rappel : Une file est une structure de données avec les méthodes `vide`, `est_vide`, `enfiler`, `defiler`.

I Par une liste chaînée

Idée 5.2 *On stocke les entrées dans une liste et on à l'aide de pointeurs le début et la fin de la file.*



Implémentation 5.3 *Ainsi, ajouter des éléments consiste seulement à enlever l'élément du début et à passer début sur l'élément suivant de la liste et enlever un élément consiste simplement à ajouter un élément à la fin de la liste et à y faire pointer le pointeur fin.*

Exercice 5.4 *Quelle est l'intérêt des pointeurs fin et début ?(fin pour trouver la fin de la liste en O(1), début pour repérer la liste, équivalent à garder le premier élément de la file)*

Exercice 5.5 *Comment encode-t-on la file vide ?*

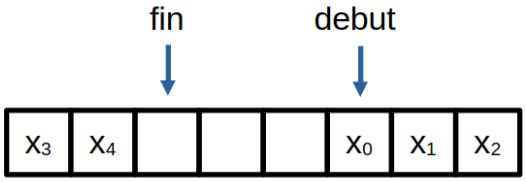
II Par un tableau

Idée 5.3 *Les éléments sont stockés consécutivement dans un tableau avec deux indices début et fin indiquant dans le tableau le début de la liste et la fin de la liste.*

Exercice 5.6 *Quelle est la complexité de chaque opération élémentaire ?*

Problème : La complexité spatiale est en O(nombre de `enfile`) alors que l'on souhaiterait avoir O(longueur maximale de la file)

Idée 5.4 *Pour cela, si l'on connaît à l'avance une borne sur cette longueur maximale, on peut utiliser des indices modulo cette borne (et ainsi utilisé un tableau ≪circulaire≫)*



Ces deux implémentations sont mutables.

III Par deux listes mais de manière immuable

Idée 5.5 *Il est possible de créer efficacement une telle structure en utilisant uniquement deux piles :*

— *La première contiendra les éléments prêts à sortir*

— *La deuxième contiendra les éléments que l'on vient de rajouter*

Remarque 5.2 *En choisissant efficacement quand on les fait passer de l'un à l'autre, on peut avoir une structure efficace.*

Proposition 5.1 *Il existe une implémentation de la structure de file pour laquelle la complxité temporelle de chaque opération est en O(1) amorti*

Développement Implémentation d'une file PAPS avec deux listes et (au choix) introduction à la complexité amortie / implémentation par une liste doublement chaînées

IV Pour aller plus loin

Une liste doublement chaînée peut, au pris d'une lourdeur relative dans l'implémentation, fournir immédiatement les opérations de la pile et de la file combinée en O(1)

3 File de priorité

Rappel : Une file de priorité est une structure de donnée ayant les méthodes `vide`, `est_vide`, `insérer`, `extraire_min`

I En théorie

Idée 5.6 *Nous implémenterons les files de priorité à l'aide de tas min.*

Définition 5.1 *Un tas min est un arbre binaire presque complet où chaque noeud à un attribut plus petit que ceux de ses fils*

L'insertion On ajoute l'élément à l'endroit au seul endroit qui préserve la structure de tas (petit dessin). Ensuite, on l'inverse successivement avec son père, si il est plus petit que lui.

Théorème 5.1 *cet algorithme préserve la structure de tas min*

Récupérer l'élément minimum On enlève la racine, on prend le dernier élément du tas, on le met à la position de la racine, et tant qu'un fils est plus petit que lui, on l'échange avec son fils le plus petit.

Théorème 5.2 *Cet algorithme préserve la structure de tas.*

Théorème 5.3 *Toutes ces opérations sont en $O(\log(n))$ où $\log(n)$ est le nombre d'éléments dans le tas.*

Commentaire 5.2 *Cette application n'est pas avec les autres car assez immédiate*

Application 5.1 *Tri par tas en $O(n\log(n))$*

II En pratique

Idée 5.7 *Pour implémenter de tels arbres, on peut les implémenter comme des structures récursives, en gardant le chemin jusqu'à la dernière feuille sous forme de listes de droite/gauche*

Exercice 5.7 *Expliciter l'algorithme permettant de mettre à jour le chemin*

Commentaire 5.3 *C'est un algorithme d'addition binaire, où quand on doit rajouter un 1 (droite), on rajoute un 0 (gauche). Donc c'est cet algorithme avec 111+1 = 0000 et 0000-1 = 111 (et le nombre de 0 qui importe)*

Exercice 5.8 *Implémenter cette structure en OcamL*

Idée 5.8 *Sinon, on peut l'implémenter comme un tableau où le fils de l'élément à l'indice i, sont stockés aux indices $2i+1$ et $2i+2$.*

Exercice 5.9 *Faire cette implémentation en C*

4 Applications aux graphes

Algorithme 5.1 : Parcours de graphe

```
a_faire ← vide()
a_faire.ajouter(s0)
tant que a_faire n'est pas vide faire
    u ← a_faire.extraire()
    si u n'a pas encore était visité alors
        pour v voisin de u faire
            a_faire.ajouter(v)
```

Si a_faire est une pile, on fait un parcours en profondeur et si a_faire est une file PAPS, on fait un parcours en largeur.

Algorithme 5.1 (Algorithme de Djikstra)
On cherche la distance dans un graphe pondéré d'un sommet s à chaque autre sommet.

On fait le parcours de graphe avec une file de priorité pour a_faire. On garde un tableau des distances, que l'on met à jour à chaque étape. La priorité dans la file est la distance que l'on a au moment de l'insertion, et on ajoute toujours les voisins non encore visité.

On obtient alors les plus courts chemins.

5 Applications systèmes

Ces structures sont utilisés dans de nombreuses applications à bas niveau.

I Pile d'appel

Quand on exécute un programme avec des appels successifs à des fonctions, de nouvelles variables sont crées, des emplacements pour les arguments, etc..., mais à la fin de la fonction, on doit restaurer l'environnement précédent.

Idée 5.9 *Pour cela, à chaque appel de fonctions, on empile l'état actuel sur une pile, et à chaque retour de fonction, on restaure l'état au sommet de la pile (que l'on dépile).*

> **Commentaire 5.4** *Suivant la place (notamment si on ne prend pas tous les exemples), on peut rajouter un schéma*

II Ordonnancement

Pour gérer le parrallélisme, un ordonnanceur doit décider, quel processus doit être executé.
Pour cela on peut par exemple :

— stocker les processsus dans une file de priorité pour que les processus les plus importants soient executés en premier.

  Exemple 5.1 *avec comme clé, la date butoire de fin d'éxecution, on exécute en premier les plus urgents*

— Mettre les processus dans une file, en prendre le premier élément, l'exécuté pendant un certain temps, puis le remettre en queue de file. C'est l'algorithme du tourniquet, qui garantie que tous les processus s'exécuterons un jour.

III Tampon

Dès que l'on a deux programmes dont la sortie de l'un est branché sur l'entrée de l'autre, et qui ne travaille pas à la même vitesse, on doit mettre un tampon entre les deux. On utilise alors une file PAPS pour stocker les données en attendant que le deuxième programme les récupère.

Exemple 5.2 *On reçoit des paquets du réseau. On doit les récupérer. On les mets dans un tampon, le temps que le programme viennent les traiter. On les restiture alors dans l'ordre d'arrivée grâce à la file.*

> **Commentaire 5.5** *Par manque de place, on peut éventuellement passer sur certains exemples, et eventuellement rajouter plein de schéma un peu partout dans la leçon. De plus, on peut évoquer à l'oral la vraie application dans un cours, surtout pour les piles et les files, qui seraient l'application pédagogique d'illustrer la différence type abstrait et structure sous-jacente, et donc ce que sont les méthodes sur les structures et la modularité de tout cela.*

Leçon 6

Implémentations et applications des ensembles et des dictionnaires

Auteur·e·s: Emile Martinez
Références :

1 Dictionnaire : type abstraits et motivation

Définition 6.1 (Dictionnaire) *Soit X un ensemble d'éléments appelés valeurs, et K un ensemble d'éléments appelés clés. Un dictionnaire D est une structure de données abstraites ayant les trois opérations :*

— *`Inserer`(D, k, x) : insère le couple (k, x) dans D, en écrasant un éventuel couple (k, x') préexistant.*

— *`Recherche`(D, k) : renvoie la valeur de x telle que (k, x) est dans D, si elle existe (peut renvoyer une valeur pas dans X sinon)*

— *`Supprime`(D, k) supprime un éventuel couple (k, x) présent dans D.*

Exemple 6.1 *annuaire téléphonique : les clefs sont les noms des personnes, les valeurs leurs numéros de téléphone*

Remarque 6.1 *Les dictionnaires sont aussi appelés tableaux associatifs*

Application 6.1 *Si on a une base de données de personnes, identifiés par un pseudonyme, on peut stocker leur informations dans un dictionnaire.*

Remarque 6.2 *C'est ce qui se passe dans une base de données*

Implémentation 6.1 (naive) *On pourrait stocker tous les éléments dans une liste, sans ordre particulier*

Exercice 6.1 *Implémenter alors les fonctions de bases. Quelles sont leur complexité ?*

2 Implémentation

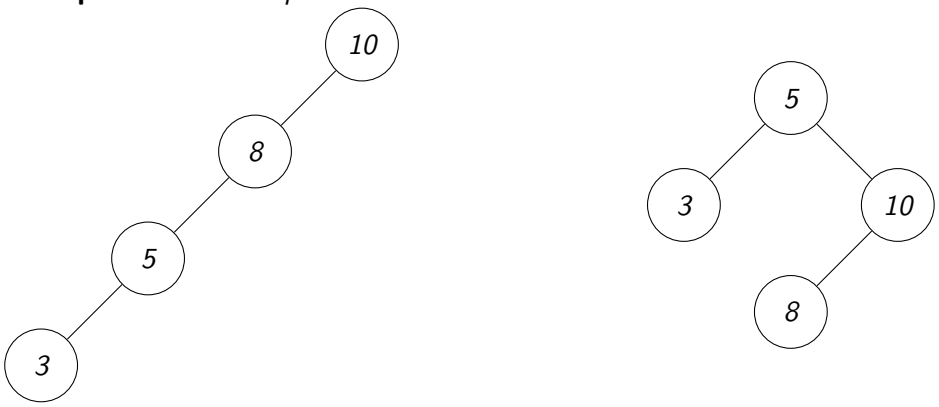
I Par des arbres

A Arbres binaires de recherche (ABR)

Définition 6.2 (ABR) *Un arbre binaire de recherche (ABR) est un arbre binaire dont les éléments sont munis d'un ordre total et où, pour chaque sous-arbre $N(g, x, d)$, l'élément x est supérieur à tous les éléments de g et inférieur à tous les éléments de d.*

Implémentation 6.2 *On peut implémenter un dictionnaire à l'aide d'un ABR à condition que l'ensemble des clefs soit muni d'un ordre total (On insère les couples (clé, valeur) et l'ordre est sur les valeurs)*

Exemple 6.2 *Deux implémentations du même ABR :*



Insertion Dans un ABR, on insère un élément x en descendant depuis la racine, en prenant le fils gauche ou le fils droit selon si x est plus petit ou plus grand que la racine courante, et en créant un nouveau nœud étiqueté par x lorsque l'on ne peut plus avancer.

Recherche Elle se fait de manière analogue

Suppression Lorsque le nœud est une feuille ou si le nœud n'a qu'un enfant, alors la transformation est simple. Par contre, si le nœud possède deux enfants, alors il faut retirer le minimum du sous-arbre droit (ou le maximum du sous-arbre gauche) pour le remplacer.

Exercice 6.2 *Implémentation des ABR en OCamL*

Proposition 6.1 *Soit $\mathcal{A}$ un ABR à n nœuds et de hauteur h. La recherche, l'insertion et la suppression se font dans le pire cas en $O(h)$ comparaisons. Or un ABR peut être déséquilibré : la hauteur est alors en $O(n)$*

B Arbres rouge-noir (ARN)

Définition 6.3 *Un arbre rouge noir (ARN) est un ABR où chaque nœud porte une couleur rouge ou noir, et qui vérifie les deux propriétés suivantes :*

- *la racine est noire*
- *les potentiels fils d'un nœud rouge sont noirs*
- *pour chaque nœud, tous les chemins menant de ce nœud à une feuille ont le même nombre de nœuds noirs.*

Proposition 6.2 *Les ARN sont équilibrés ($h = O(\log n)$)*

Corollaire 6.1 *Dans un ARN, on peut effectuer les opérations d'insertion, de recherche et de suppression en $O(h)$, ie $O(\log n)$.*

Developpement Preuve de la propriété 6.2 et présentation de l'insertion dans un ARN

II Tables de hachage

Idée 6.1 *Le but ici va être de stocker nos données dans un tableau (d'où le nom tableau associatif). Cela se fait donc en deux étapes :*

- *Associer à notre données un nombre (pas trop grand) (appelé hachage) par une fonction appelée fonction de hachage.*
- *Avoir un tableau avec une case par hachage possible, où l'on stocke la donnée*

A Fonction de hachage

Définition 6.4 *Une fonction de hachage est une fonction $h : K \to [\![0, m-1]\!]$ (avec $m << |K|$)*

Proposition 6.3 *Une fonction de hachage a la propriété du hachage parfait si pour $x \neq y \in K$, $\mathbb{P}(h(x) = h(y)) = \frac{1}{m}$*

Remarque 6.3 *Le hachage parfait veut dire que les valeurs de hachage sont comme pris au hasard dans $[\![0, m-1]\!]$. Néanmoins, comme on veut que $h(x)$ vaille toujours la même chose, on ne prend pas de fonctions aléatoires.*

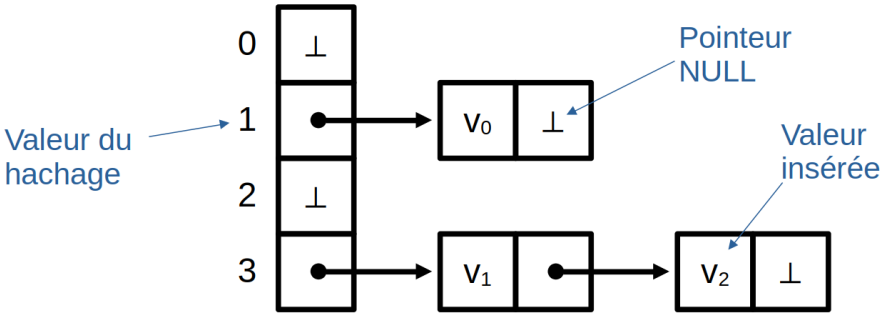
Exemple 6.3 *On choisit un flottant A. On interprète les bits de données de la structure comme un entier x (en les collant). On prend alors $h(d) = \lfloor A * x \rfloor \mod m$*

B Table de hachage

Il y a trois étapes dans le stockage dans un tableau :

1. Hacher la valeur et la mettre dans à sa case dans le tableau
2. Si la case était déjà occupée (collision), stocker la donnée dans une structure annexe
3. Si le tableau est trop plein, augmenter m (et donc recopier toutes les données).

Exemple 6.4 *Schéma où la structure annexe est une liste chaînée pour chaque case*



Commentaire 6.1 *Si on a la place on peut ecrire la remarque, sinon le dire à l'oral que si les abr nécessite un ordre total, la table de hachage nécessite de sérialiser ou de donner directement le hachage*

Commentaire 6.2 *Normalement là on pourrait parler de compexité, mais on considère que le tableau 3 suffit.*

3 Ensembles

Définition 6.5 *Un ensemble est un dictionnaire dans lequel il n'y a pas de clefs. La fonction recherche renvoie donc un booléen qui indique la présence ou non de l'élément.*

Remarque 6.4 *On déduit alors de l'implémentation des dictionnaires, l'implémentation des ensembles*

Commentaire 6.3 *Cette remarque justifie que l'on parle si brièvement des ensembles, et si tard dans la leçon. Beaucoup des choses sur les ensembles se déduisant de celles des dictionnaires*

Idée 6.2 *Comme ce sont des ensembles on pourrait également vouloir des opérations ensemblistes comme l'union, l'intersection, etc...*

Implémentation 6.3 *On peut faire ces opérations sur les structures précédentes en les parcourant (pour l'union, on insère tous les éléments de l'un dans l'autre par exemple)*

Remarque 6.5 *Ces opérations réhabilite l'idée de la liste triée*

Récapitulatif des complexités :

Structure	Insertion	Suppression	Recherche	union	intersection
liste	$O(n)$	$O(n)$	$O(n)$	$O(n \times m)$	$O(n \times m)$
liste triée	$O(n)$	$O(n)$	$\log n$	$O(n+m)$	$O(n+m)$
ARN	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n+m)\log(n+m)$	$O(\min(n,m)\log(\max(n,m)))$
Table de hachage	$O(n)$ $O(1)$ moy.	$O(n)$ $O(1)$ moy.	$O(n)$ $O(1)$ moy.	$O(n \times m)$ $O(n+m)$ moy.	$O(n \times m)$ $O(\min(n,m))$ moy.

Exercice 6.3 *Choisir des implémentations en C et les comparer. Si l'on traite les listes triées, ouverture sur les skip listes*

4 Application

I Dictionnaire d'adjacence

Soit $G = (S, A)$ un graphe. (où $S = [\![1, n]\!]$)

On peut représenter G par un tableau de dictionnaire D tq $D[u]$ est un dictionnaire contenant les voisins de u

intérêts : On a les avantages de la matrice d'adjacence (on detecte rapidement si il y a une arête) et des listes d'adjacence (stockage en —A— et obtention de tous les voisins linéairement).

Exemple 6.5 *Sur un graphe de personnes se connaissant, on sait si A connait B rapidement, mais on n'a pas à stocker tous les false des couples de personnes en se connaissant pas.*

II Mémoïsation

Supposons que l'on ait une fonction du type

```
fonction (f_args):
    corps de f
    return x
```

Il se peut que l'on ait beaucoup d'appels à `f` sur les mêmes arguments, comme des fonctions récurisves en programmation dynamique. En s'inspirant de cette dernière, on peut alors ne faire qu'une fois les appels sur chaque argument grâce à un dictionnaire :

```
fonction f(args):
    Si args est dans dictionnaire:
        renvoyer dictionnaire[args]
    Sinon:
        corps de f
    dictionnaire[args] = x
    renvoyer x
```

III Autres applications

Commentaire 6.4 *On peut dire à l'oral que en gros, utiliser un dictionnaire est un peu équivalent à $f : K \to [\![1, K]\!]$. (c'est plus que simplement le fait logique que, par un tableau classique, c'est équivalent)*

Les dictionnaires et les ensembles servent dès que l'on veut accéder rapidement à un élément dont la structure n'est pas un entier.

Exemple 6.6 *En algorithmique du texte, pour associer des valeurs à des chaînes de caractère (Huffman, Boyer-Moore)*

Exemple 6.7 *Quand on fait un parcours de graphe dont les sommets ne sont pas $[\![1, n]\!]$ on peut utiliser un ensemble pour savoir quels éléments on a déjà parcouru.*

Commentaire 6.5 *On peut soit insister à l'oral sur cet exemple, soit en faire une sous partie, du fait que c'est une application pour les ensembles*

Mais on peut les retrouver également dans beaucoup de domaines comme en bases de données, et dès que l'on manipule un nombre faible de valeur comparés à l'ensemble des valeurs possibles.

Développement : Amélioration du tri par comptage par l'usage de dictionnaires.

Leçon 7

Accessiblité et chemins dans un graphe. Applications

Auteur·e·s: Emile Martinez
Références :

1 Definition

Commentaire 7.1 *Les applications sont éparpillées tout le long de la leçon, et relativement peu développer, donc il peut-être rentable des les souligner pendant la défense de plan.*

Définition 7.1 (chemin) *Dans un graphe orienté G (resp. non orienté) on appelle chemin de longueur λ une suite de $(\lambda + 1)$ sommets $(s_0, s_1, \ldots, s_\lambda)$*

Remarque 7.1 *Par convention, on dit qu'il y a un chemin de longueur 0 de tout sommet vers lui-même.*

Remarque 7.2 *Dans un graphe non-orienté, les chemins sont aussi appelés chaînes.*

Application 7.1 *Dans un graphe de flot de contrôle, le critère tous les chemins consiste à trouver des tests qui font tous les chemins possibles du graphe*

Commentaire 7.2 *Une première application, dont on ne parle pas beaucoup plus parce que ca a pas grand chose a voir. Donc les chemins servent là, mais la théorie derrière n'est pas intéressante par la théorie des graphes. Illustre simplement la diversité de l'utilisation des graphes.*

Commentaire 7.3 *On peut mentionner que ca fait déjà une première application*

Définition 7.2 (chemin élémentaire) *Un chemin est dit élémentaire s'il ne contient pas plusieurs fois le même sommet.*

Commentaire 7.4 *Par manque de place, cette définition peut être enlevé*

Définition 7.3 (circuit/cycle) *Dans un graphe orienté (resp. non orienté), un chemin $(s_0, \ldots, s_\lambda)$ dont les λ arcs (resp. arêtes) sont distincts deux à deux et tels que $s_0 = s_\lambda$, est un circuit (resp. un cycle).*

Exemple 7.1 *On dit qu'un circuit est hamiltonien si il passe par tous les sommets une et une seule fois. Décider de l'existence d'un tel circuit dans un graphe est NP-complet (i.e. dur)*

Exercice 7.1 *On dit qu'un graphe non orienté est Eulérien s'il existe un cycle passant par chaque arête exactement une fois. Montrer qu'un graphe est eulérien si et seulement si tous ses sommets sont de degré pair.*

Commentaire 7.5 *Ces deux choses là peuvent être présenter comme des formes d'applications, du moins d'application des définitions (il semble important que la NP-complétude de quelque chose soit mentionné dans cette leçon)*

Définition 7.4 (accessibilité) *Étant donné un graphe G (orienté ou non) et deux sommets s et t, on dit que t est accessible depuis s s'il existe un chemin allant de s à t dans G.*

2 Accessibilité

I Tri topologique

Définition 7.5 (tri topologique) *Étant donné un graphe orienté $G = (S, A)$, on dit que $\sigma : S \to [\![1, |S|]\!]$ est un tri topologique si $(s,t) \in A \implies \sigma(s) < \sigma(t)$*

Corollaire 7.1 *Soit σ un tri topologique sur G. Pour $s, t \in S$, si il existe un chemin de s à t, alors $\sigma(s) \leq \sigma(t)$*

Proposition 7.1 *G est acyclique si et seulement si G admet un tri topologique.*

Démonstration pour le sens direct. On montre par l'absurde par récurrence qu'un graphe acyclique a un sommet source.
On montre par récurrence la proposition sur $|S|$ □

Algorithme 7.1 *Construction d'un tri topologique*

- *Créer une pile vide*

- *Faire un parcours en largeur postfixe de G en appliquant la fonction empile*
- *Renvoyer la pile*

Application 7.2 *Ordonnancement de tâches*

Définition 7.6 *Soit $T = \{t_1, \ldots, t_n\}$ un ensemble de tâches d'un ordinateur et $C = T^2$ un ensemble de contraintes ($(t_i, t_j) \in C$ veut dire que t_i doit être fait avant t_j). Un ordonnancement de ces tâches est un ordre d'éxécution de ces tâches.*

Proposition 7.2 *On peut trouver un ordonnancement en temps linéaire*

Démonstration. On prend le graphe (T, C) est on fait un tri topologique. □

II Connexité

Proposition 7.3 *La relation «il existe un chemin de s à t et de t à s» est une relation d'équivalence.*

Exercice 7.2 *Montrer que le graphe quotienté par cette relation d'équivalence est acyclique*

Commentaire 7.6 *Ici on introduit le DAG des classes des composantes (fortement) connexes. On peut néanmoins dire que suivant le niveau de la classe, il est très probable que le terme quotienté ne soit pas maitrisé. Donc la on le met par concision, au cas où il aurait déjà était vu en maths (hors programme en maths), mais en vrai on pourrait périphraser et faire des définitions. De plus la c'est un plan, donc on dit ce qu'on ferait dans l'exercice, pas tout l'énoncé rigoureux.*

Définition 7.7 *Dans un graphe non orienté (resp. orienté), les classes d'équivalence de cette relation sont appelés composantes connexes (resp. fortement connexes).*
Si il n'y a qu'une seul classe, le graphe est dit connexe (resp. fortement connexe)

Commentaire 7.7 *On prend ca comme déf et non pas la propriété suivantes en ayant défini connexe comme un chemin de tous le monde à tout le monde, pour avoir une définition plus explicites et naturelles que de passer par la maximalité. Néanmoins, cela nécessite l'abstraction de la classe d'équivalence (vue en maths). Si la classe est faible, on peut faire tout ca plus avec les mains.*

Exercice 7.3 *Dans un graphe connexe, deux chemins élémentaires maximaux ont un nœud en commun.*

Proposition 7.4 *Une composante connexe (resp. fortement connexe) est un sous-graphe connexe (resp. fortement connexe) maximal.*

Algorithme 7.2

Algorithme 7.1 : Calcul des composantes connexes (cas non orienté)

tant que un sommet n'est pas visité **faire**
Créer une nouvelle composante connexe
Parcourir le graphe des sommets non visités depuis un sommet v non visité en ajoutant a la composante tous les sommets que l'on croise

Pour les composantes fortement connexes, il faudra plus qu'un simple parcours.

Algorithme 7.3

Algorithme 7.2 : Algorithme de Kosaraju

Appliquer l'algorithme de construction d'un tri topologique (meme si le graphe a des cycles)
Prendre G^T le graphe transposé de G (i.e., (u,v) devient (v,u)) **pour** i allant de 1 à n **faire**
Si $\sigma(i)$ n'a pas déjà été visité
Créer une nouvelle composante fortement connexe
Visiter G^T depuis $\sigma(i)$ en ajoutant les sommets visités non encore attribués

Proposition 7.5 *Cet algorithme construit les composantes fortement connexes en temps linéaire*

Définition 7.8 *Le problème 2-SAT est le problème de savoir si étant donné n variables $x_1, \ldots, x_n$ et p clauses $C_1, \ldots, C_p$ de taille 2 sur $x_1, \ldots, x_n$, $\varphi = \bigwedge_{i=1}^{p} C_i$ est satisfiable*

Developpement : Résolution en temps linéaire de 2-SAT.

3 Plus court chemin

Définition 7.9 (graphe pondéré) *Un graphe pondéré est un graphe $G = (S, A)$ muni d'une fonction de poids $w : A \to Z$.*
Le poids d'un chemin est alors défini comme la somme des poids des arêtes qui le compose.

Remarque 7.3 *On peut étendre w à S^2 en posant $w(u,v) = +\infty$ lorsque $(u,v) \notin A$.*

Définition 7.10 *Dans un graphe G, un plus court chemin (pcc) de u à v $(u,v \in S)$ est un chemin de poids minimal de u à v*

Remarque 7.4 *Si les poids sont unitaires, on peut trouver le pcc entre deux sommets u et v en faisant un parcours en largeur depuis u.*

Commentaire 7.8 *On considère les parcours déjà vu (servant dans bien d'autres contextes que les chemins dans les graphes). Ici on fait simplement le lien, et on illustre la notion. On pourrait également le faire sur un exemple au tableau, et essayer de le faire devnier aux élèves (lien entre différentes parties du cours)*

I D'un sommet à tous les autres

Lorsque la fonction de poids est à valeur dans $\mathbb{N}$, on peut utiliser l'algorithme de Dijkstra.

Algorithme 7.3 : dijkstra(G, S)

```
distance ← tableau de taille |S| initialisé à +∞
distance[s] ← 0
F ← FilePriorite()
Inserer(F, s, 0)
tant que ¬estVide(F) faire
    u, du ← ExtraireMin(F)
    pour v ∈ N(u) faire
        d ← du + w(u, v)
        si d < distance[v] alors
            si distance[v] = +∞ alors
                Inserer(F, v, d)
            sinon
                DiminuerPriorite(F, v, d)
                distance[v] ← d
retourner distance
```

Algorithme 7.4

Proposition 7.6 *L'algorithme de Dijkstra réalise au plus $|S|$ appels à $Inserer$ et $ExtraireMin$, et $|A|$ appels à $DiminuerPriorite$.*
Donc avec un tas min, on obtient $O(|A| \times \log |S|)$

Remarque 7.5 *Si la structure d'entrée est une matrice d'adjacence, on peut faire l'algorithme en $O(n^2)$ sans structure particulière pour F.*

Remarque 7.6 *Lorsque la fonction de poids est à valeur dans $\mathbb{Z}$ et que le graphe ne contient aucun circuit absorbant, on peut utiliser l'algorithme de Bellman-Ford.*

Application 7.3 *Détection des cycles absorbants.*

Remarque 7.7 *Si l'on souhaite seulement le plus court chemin entre deux points, on peut utiliser l'algorithme A* (Dijkstra + heuristique du chemin restant).*

II De tous les sommets à tous les sommets

Lorsque le graphe ne contient aucun cycle absorbant, l'algorithme de Floyd Warshall calcule les plus courts chemins entre toute paire de sommets de $S = \{1, \dots, n\}$ par programmation dynamique.

- ⋆ **Sous-problèmes :** $d^{(k)(i,j)}$ la distance du pcc de i à j avec seulement $[\![1, k]\!]$ comme sommets intermédiaires
- ⋆ **Relation de récurrence**

$$d^{(0)}(i,j) = w(i,j)$$
$$d^{(k+1)}(i,j) = \min\left(d^{(k)}(i,j),\ \ d^{(k)}(i,k) + d^{(k)}(k,j)\right)$$

- ⋆ **Résolution :** On résout sur S^2 à k croissant

Proposition 7.7 *Floyd Warshall est en $O(n^3)$*

Remarque 7.8 *Si les poids sont positifs, appliquer dijkstra à chaque sommet nous donne $O(n \times |A| \times \log n)$*

Remarque 7.9 *Cette algorithme utilise une matrice d'adjacence (algorithme centralisé)*

Algorithme 7.5 *Si on a des listes d'adjacence (algorithme décentralisé/distribué), on peut utiliser l'algorithme de Bellman Ford*

Application 7.4 *Routage des paquets d'un réseau par le protocole IP avec l'algorithme de Bellman Ford.*

Developpement : Presentation et terminaison de l'algorithme de Bellamn Ford.

Commentaire 7.9 *On a fait une leçon sur les graphes, sans aucun dessin de graphes. En vrai, il pourrait y en avoir sur des exemples, et puis les schémas ne sont pas absolument necessaires pour illustrer la plupart des concepts, dont surtout la formalisation est imporante (la def d'un chemin est explicite).*