Table des matières

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

Exemple 1.3:

```
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 1.4: 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 1.5:

Algorithme 1.2 : pgcd(a,b)

tant que $\underline{\min(a,b) > 0}$ **faire**
  si $\underline{a < b}$ **alors**
    $\underline{b \leftarrow b - a}$
  sinon
    $a \leftarrow a - b$
retourner $\underline{\max(a,b)}$

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*

Propriété 1.6: Si une boucle a un variant de voucle, alors elle s'éxecute un nombre fini de fois. De même pour un algorithme récursif.

Exemple 1.7: pgcd(a,b) termine pout tout $(a, b) \in (\mathbb{N}^*)^2$

Remarque 1.8: 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 1.9:

On définit $ack(n,m)$ pour $n,m \in \mathbb{N}$ par
$$\begin{cases} ack(0,m) = m+1 \\ ack(n,0) = ack(n-1,1) \\ ack(n,m) = ack(n-1, ack(n,m-1)) \end{cases}$$

Il n'est pas immédiat que ack termine.

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))
```

Définition 1.10: On dit qu'un ordre est un **ordre bien fondé** si toute suite décroissante est stationnaire

Exemple 1.11: $\mathbb{N}$ avec l'ordre naturel est bien fondé

Propriété 1.12: Les ordres produit et lexicographiques d'ordre bien fondés sont bien fondés

Exemple 1.13: Un ordre total sur un ensemble fini est bien fondé, donc l'ordre alphabétique est bien fondé (c'est un ordre lexicographique)

Définition 1.14: On étend alors la définition du variant aux fonctions à valeurs dans un bon ordre.

Propriété 1.15: La propriété 1.6 reste valide avec notre définition étendue du variant

Exemple 1.16: 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 1.17: On appelle **spécification** d'un algorithme deux propriétés P_1 sur les entrées (pré-condition) et P_2 sur les sorties (post-condition)

Exemple 1.18: Pour Syracuse, $P_1 :\ll a \in \mathbb{N} \gg$ et $P_2 :\ll \text{Syracuse}(a) \in \{1,2,4\} \gg$

Définition 1.19: 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 1.20: L'algorithme de l'exemple 1.5 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 1.21: 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.

Propriété 1.22: 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 1.23: 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 1.20

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 1.24: 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 1.25: Si le principe 1.24 est respecté, alors l'algorithme est partiellement correct.

Exemple 1.26: 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 1.24 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 1.27: Quand un programme termine sur toute entrée valide et est partiellement correct, on dit qu'il est correct, ou encore totalement correct.

Exemple 1.28:

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 1.29: La correction partielle et la terminaison sont indécidables.

Developpement ?? Preuve du théorème 1.29

II Cas des algorithmes récursifs

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

Propriété 1.30: SI $(E, \preceq)$ est un ordre bien fondé, alors toutes parties non vides à un élément minimal (plus grand que personne)

Théorème 1.31: 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 1.32: 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 1.33: Dans l'exemple 1.29, la propriété $\mathcal{P}(n)$: «exp(a,n)= a^n» vérifie les bypothèses du théorème 1.31. Donc, $\forall n, \texttt{exp(a,n)} = a^n$, 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 2.3: 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 2.4: Informellement, programmer impérativement, c'est utiliser des variables, des affectations, des tableaux, des boucles for et while, etc...

Exemple 2.5: 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 2.6: 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 2.7: 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 2.8: 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 2.9: 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 2.10: 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 2.11: On peut transformer la fonction max de l'exemple 2.8 en la fonction
let max x y = if x > y yhen x else y de type int -> int -> int

Exemple 2.12: Si, sur l'exemple 2.11, 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 2.13: La puissance du fonctionnel vient de la récursivité

3 Programmation orientée objet

I Obtenir de la modularité

Définition 2.14: 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 2.15:

```
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 2.16: La **programmation orienté objet** consiste à utiliser des classes et des objets de ces classes quand on programme.

Remarque 2.17: 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 2.18: 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 2.19: Sur l'exemple 2.15, 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 2.20**: 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 2.21**: 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 3.2: 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 3.3: **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 3.4: **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 3.5: Certaines sorties (attendues) peuvent être des erreurs.

Exemple 3.6: Un jeu de tests pour une fonction calculant le pgcd peut être
$\{((1,2),1),((-3,6),3),((0,0),0),((2,2.45),\text{Erreur de type})\}$

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 3.7: 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 3.8: La première approche consisterait à générer des valeurs aléatoires dans un domaine, et espérer en prendre suffisamment pour que cela fonctionne.

Principe 3.9: Une approche plus maline, à partitionner le domaine, puis à appliquer l'approche naïve sur chaque domaine.

Exemple 3.10: 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 3.11: On se contente souvent de prendre un seul test par classe.

Remarque 3.12: 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 3.13: On essaye alors de se placer au limites des domaines, et de vérifier spécifiquement ces cas là.

Exemple 3.14: 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 3.15: 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 3.16: 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 3.17: 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 3.18: 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

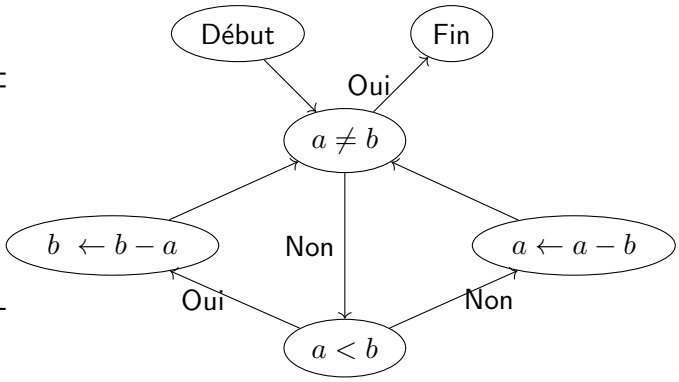
Pour un test en boîte blanches, on connaît le code, et on va vouloir générer des données d'entrées en fonction de ce code là.

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

Définition 3.19: 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 3.20: 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 3.21: 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 3.22: 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 3.23: Parfois les critères sont insatisfiables.

Remarque 3.24: 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 3.25: Une autre approche consiste à avoir un jeu qui satisfait ou invalide toutes les conditions de toutes les manières possibles.

Exemple 3.26: 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 4.2: 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 4.3: On prend une constante Z et un constructeur d'arité 1 $Succ$

Exemple 4.4: 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 4.5: 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 4.6: L'ensemble inductif définit par l'exemple 4.3 peut être une définition des entiers naturels

Exemple 4.7: Exemple : L'ensemble inductif $\mathcal{A}_{simp}$ définit par l'exemple 4.4 est l'ensemble des expressions arithmétiques simplifiées.

Remarque 4.8: $\oplus(1,1) \neq \otimes(1,2) \neq \otimes(2,1)$. On s'intéresse à l'expression et non au résultat.

Développement : Validité de la construction d'un ensemble inductif (remarque 4.2 et définition 4.5)

II Induction structurelle

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

Propriété 4.9: 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 4.10: 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 4.11 (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 4.12: La récurrence est un cas particulier dans le cas de la définition des entiers par l'exemple 4.6

Exemple 4.13: On montre par induction structurelle que $eval(e)$ pour $e \in \mathcal{A}_{simp}$ est multiple du pgcd des constantes apparaissant dans e

Définition 4.14: Un ordre bien fondé est un ordre où toute partie non vide admet un élément minimal (plus grand que personne)

Propriété 4.15: L'ordre produit et l'ordre lexicographique d'ordres bien fondés sont bien fondés

Exemple 4.16: $\mathbb{N}$ avec l'ordre naturel est bien fondé, donc $\mathbb{N}^k$ avec l'ordre produit ou lexicographique aussi.

Théorème 4.17: 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 4.18: Cela étend le principe de récurrence forte.

Définition 4.19: 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 $x_j \leq_s f_i(x_1, \ldots, x_{\alpha(i)})$

Propriété 4.20: $\leq_s$ est une relation d'ordre bien fondé

Corollaire 4.21: 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 4.22: 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 4.23: Pour définir les entiers de l'exemple 4.6, on peut écrire

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

Syntaxe 4.24: Pour gérer les types, on peut utiliser le filtrage comme pour les listes.

Exemple 4.25: 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 4.26: La validité de cette définition vient de la propriété 4.9

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 4.27: 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 4.28: Cela correspond au type `int list` d'OCamL ;-)

Exercice 4.29: Définir inductivement la taille d'une liste chaînée

II Les arbres binaires

Exercice 4.30: Définir inductivement la taille d'une liste chaînée.

Définition 4.31 (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 4.32: Ce qui en OCamL nous donne `type 'a arbre = E | Noeud of 'a * 'a arbre * 'a arbre`

Exemple 4.33: 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 4.34: Prouver par induction structurelle la terminaison de la fonction hauteur

Exercice 4.35: Donner la définition inductive de la taille d'un arbre binaire (son nombre d'éléments)

III Les arbres généraux

Définition 4.36: 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 4.37: 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))*

Développement : Équivalence des arbres binaires et des arbres généraux.

3 Ensembles inductifs

I Formules propositionnelles

Définition 4.38 (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 4.39: Défini inductivement le nombre de variables présent dans la formule

Définition 4.40: On appelle valuation (ou distribution de vérité) toute fonction $v : V \to \{0, 1\}$

Définition 4.41 (Evaluation d'une formule): Soit v une valuation. La fonction d'évaluation d'une formule $[.]_v : F \to \{0, 1\}$ se définit inductivement par

- $[\top]_v = 1$
- $[\bot]_v = 0$
- $[x]_v = v(x)$ pour $x \in V$
- $[\neg F]_v = 1 - [F]_v$
- $[f_1 \wedge f_2]_v = \min([f_1]_v, [f_2]_v)$
- $[f_1 \vee f_2]_v = \max([f_1]_v, [f_2]_v)$
- $[f_1 \to f_2]_v = \begin{cases} 0 & \text{si } [f_1]_v = 0 \text{ et } [f_2]_v = 1 \\ 1 & \text{sinon} \end{cases}$

Exercice 4.42: 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 4.43: 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 4.44: On aurait aussi pu prendre comme définition de Σ^*, $\bigcup_{n \geq 0} \Sigma^n$

Définition 4.45: Soit X un ensemble et $\delta : X \times \Sigma \to X$.
On définit alors $\delta^* : X \times \Sigma^* \to X$ inductivement par :

— $\forall x \in X, \delta^*(x, \varepsilon) = x$

— $\forall x \in X, \forall a \in \Sigma, \forall w \in \Sigma^*, \delta^*(x, w.a) = \delta(\delta^*(x, w), a)$

Exercice 4.46: Montrer par induction structurelle que $\delta^*(x, a.w) = \delta^*(\delta(x, a), w)$

Commentaire 4.7 *On pourrait ici mettre plein d'exercices plus intéressants (vu qu'on est sur les automates) mais ca nécessiterait probablement d'introduire trop de choses (déjà qu'ici on utilise $a.w$ sans avoir défini la concaténation). Meme si on pourrait déjà commencé par δ^* est un morphisme de Σ^* avec la concaténation dans $X \to X$ pour la composition (en gros, $\delta^* \circ \delta^* = \delta^*$).*

Commentaire 4.8 *Si cette fonction est trop compliqué, on peut remplacer par l'exercice suivant*

Définition 4.47: *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 4.48: *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

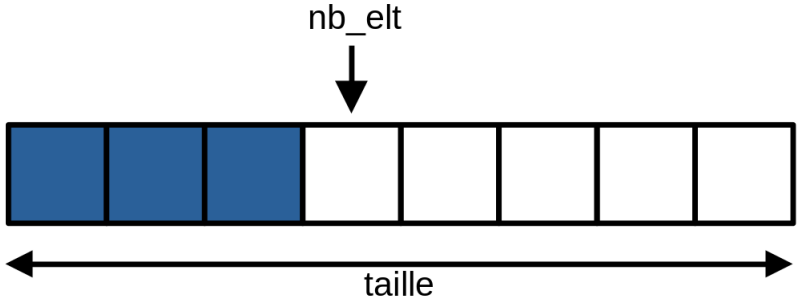
Cette manière est immédiate et donc naturelle.

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 5.2: 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 5.3: 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 5.4: 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 5.5: en C en TP avec les déclarations déjà faites, `vide` également, et le corps des autres fonctions à remplir

Remarque 5.6: 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 5.7: Laquelle de ces deux implémentations est mutable ? Immuable ?

III Complexité

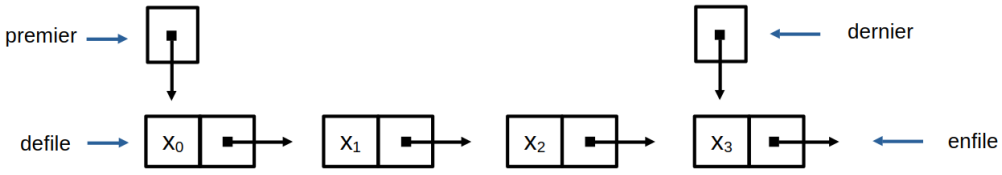
Toutes ces opérations sont en $O(1)$. De plus sa complexité spatiale n'est, si l'implémentation est bien effectuée, que O(hauteur maximale de la pile)

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



Implémentation 5.9: 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 5.10: 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 5.11: Comment encode-t-on la file vide ?

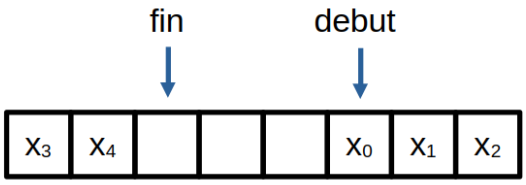
II Par un tableau

Idée 5.12: 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 5.13: 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 5.14: 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 5.15: 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 5.16: En choisissant efficacement quand on les fait passer de l'un à l'autre, on peut avoir une structure efficace.

Propriété 5.17: 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 5.18: Nous implémenterons les files de priorité à l'aide de tas min.

Définition 5.19: 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 5.20: 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 5.21: Cet algorithme préserve la structure de tas.

Théorème 5.22: 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 5.23: Tri par tas en $O(n\log(n))$

II En pratique

Idée 5.24: 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 5.25: 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 5.26: Implémenter cette structure en OcamL

Idée 5.27: 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 5.28: 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 5.29 (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 5.30: 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 5.31: 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 5.32: 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 6.2: annuaire téléphonique : les clefs sont les noms des personnes, les valeurs leurs numéros de téléphone

Remarque 6.3: Les dictionnaires sont aussi appelés tableaux associatifs

Application 6.4: Si on a une base de données de personnes, identifiés par un pseudonyme, on peut stocker leur informations dans un dictionnaire.

Remarque 6.5: C'est ce qui se passe dans une base de données

Implémentation 6.6 (naive): On pourrait stocker tous les éléments dans une liste, sans ordre particulier

Exercice 6.7: Implémenter alors les fonctions de bases. Quelles sont leur complexité ?

2 Implémentation

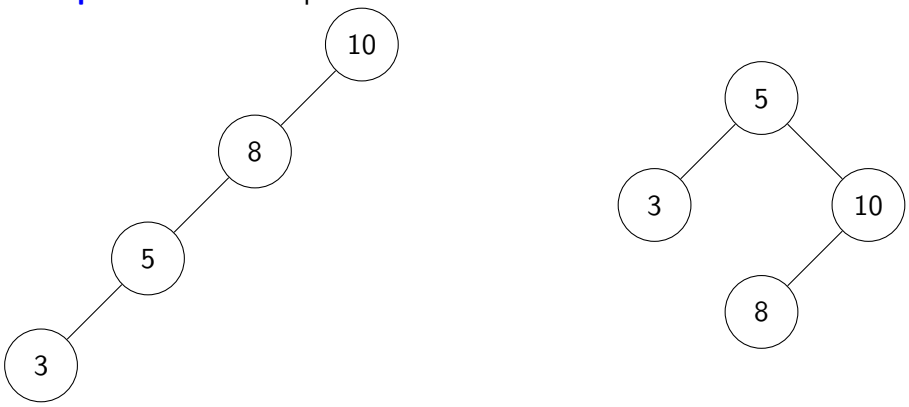
I Par des arbres

A Arbres binaires de recherche (ABR)

Définition 6.8 (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 6.9: 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 6.10: 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 6.11: Implémentation des ABR en OCamL

Propriété 6.12: 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 6.13: 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.

Propriété 6.14: Les ARN sont équilibrés ($h = O(\log n)$)

Corollaire 6.15: 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 6.14 et présentation de l'insertion dans un ARN

II Tables de hachage

Idée 6.16: 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 6.17: Une fonction de hachage est une fonction $h : K \to [\![0, m-1]\!]$ (avec $m << |K|$)

Propriété 6.18: Une fonction de hachage a la propriété du hachage parfait si pour $x \neq y \in K$, $\mathbb{P}(h(x) = h(y)) = \frac{1}{m}$

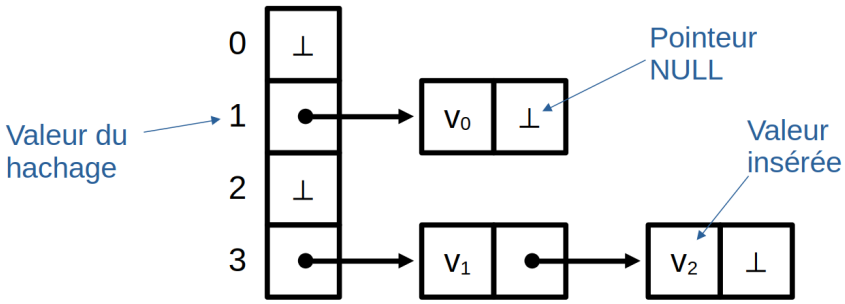
Remarque 6.19: Le hachage parfait veut dire que les valeurs de hachage sont comme pris au hasard dans $[\![0, m-1]\!]$. Néanmoins, comme on veut que $h(x)$ vaille toujours la même chose, on ne prend pas de fonctions aléatoires.

Exemple 6.20: 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 6.21: 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 6.22: 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 6.23: 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 6.24: Comme ce sont des ensembles on pourrait également vouloir des opérations ensemblistes comme l'union, l'intersection, etc...

Implémentation 6.25: 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 6.26: 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 6.27: 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 6.28: 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 6.29: En algorithmique du texte, pour associer des valeurs à des chaînes de caractère (Huffman, Boyer-Moore)

Exemple 6.30: 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

Accessibilité et chemins dans un graphe. Applications

Auteur·e·s: Emile Martinez
Références :

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.*

1 Definition

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

Remarque 7.3: Dans un graphe non-orienté, les chemins sont aussi appelés chaînes.

Application 7.4: 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 7.5 (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 7.6 (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 7.7: 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 7.8: 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 7.9 (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 7.10 (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 7.11: Soit σ un tri topologique sur G. Pour $s, t \in S$, si il existe un chemin de s à t, alors $\sigma(s) \leq \sigma(t)$

Propriété 7.12: 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 7.13: 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 7.14: Ordonnancement de tâches

Définition 7.15: 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.

Propriété 7.16: On peut trouver un ordonnancement en temps linéaire

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

II Connexité

Propriété 7.17: La relation «il existe un chemin de s à t et de t à s» est une relation d'équivalence.

Exercice 7.18: 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 7.19: 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 7.20: Dans un graphe connexe, deux chemins élémentaires maximaux ont un nœud en commun.

Propriété 7.21: Une composante connexe (resp. fortement connexe) est un sous-graphe connexe (resp. fortement connexe) maximal.

Algorithme 7.22:

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 7.23:

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

Propriété 7.24: Cet algorithme construit les composantes fortement connexes en temps linéaire

Définition 7.25: 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 7.26 (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 7.27: On peut étendre w à S^2 en posant $w(u, v) = +\infty$ lorsque $(u, v) \notin A$.

Définition 7.28: 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 7.29: 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 7.30:

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
```

Propriété 7.31: 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 7.32: 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 7.33: 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 7.34: Détection des cycles absorbants.

Remarque 7.35: 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 $S = \{1, \ldots, n\}$ 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

Propriété 7.36: Floyd Warshall est en $O(n^3)$

Remarque 7.37: Si les poids sont positifs, appliquer dijkstra à chaque sommet nous donne $O(n \times |A| \times \log n)$

Remarque 7.38: Cette algorithme utilise une matrice d'adjacence (algorithme centralisé)

Algorithme 7.39: Si on a des listes d'adjacence (algorithme décentralisé/distribué), on peut utiliser l'algorithme de Bellman Ford

Application 7.40: 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).*

Leçon 8

Algorithme de tri. Exemple, Complexité et Applications

Auteur·e·s: Emile Martinez
Références :

1 Introduction

Définition 8.1: Un algorithme de tri prend en entrée un liste d'éléments E muni d'un ordre total $\leq$ (mais dont on peut se passer de l'antisymétrie) et renvoie o=une liste S telle que :

1. elle contient exactement les éléments de E
2. les éléments de S apparaissent dans l'ordre croissant

Application 8.2: Pourquoi trier ? Cela permet de simplifier des opérations (min, max, recherche, etc...)

Exercice 8.3: Chercher à la main un mot dans une liste triée et non triée.

Commentaire 8.1 *On peut faire cet exercice en seconde. Avec éventuellement le fait de comparer deux listes de mots. Tout ca pour voir l'intérêt du tri. Ensuite on peut leur demander comment faire pour trier, permettant de faire émerger des algorithmes sans le formalismes nécessaires. Et suivant le format des mots, on obtiendra probablement des choses différentes (une liste écrit sur un papier -¿ tri par sélection, des cartes avec les mots -¿ tri par insertion, tri a bulle, voir même du tri par paquet).*

Ainsi on peut faire de l'informatique sans l'écueil de l'apprentissage de la syntaxe python qui bloque beaucoup d'élèves.

Définition 8.4 (Propriétés des tris):

- en place : Utilise $O(1)$ espace en plus de l'espace des données d'entrées
- stable : Si $E[i] \leq E[j]$ et si $E[j] \leq E[i]$, avec $i < j$, dans S, $E[i]$ apparaîtra avant $E[j]$
- en ligne : on peut trier les données même si elles arrivent au fur et à mesure
- parallélisable : on peut diviser le travail sur plusieurs fils.

2 Tri quadratiques

I Tri par sélection

Algorithme 8.1 : $Tri\_par\_selection(E)$

$S \leftarrow []$
pour i allant de 1 à $|E|$ **faire**
  Extraire $mini$, le minimum de E
  Ajouter $mini$ à la fin de S
retourner S

Exemple 8.5:

$E = [4, 3, 6, 1]$

i	E	m	S
$\times$	[4, 3, 6, 1]	$\times$	[]
0	[4, 3, 6]	1	[1]
1	[4, 6]	3	[1, 3]
2	[6]	4	[1, 3, 4]
3	[]	6	[1, 3, 4, 6]

$S = [1, 3, 4, 6]$

Commentaire 8.2 *Cet exemple peut être remplacer par simplement : ≪Exemple : mettre un exemple d'execution de l'algorithme≫ si on manque de place*

Propriété 8.6:

- ⋆ Terminaison : l'algorithme n'utilise que des boucles bornées
- ⋆ Correction : Invariant de boucle ≪S est trié et contient les i plus petits de E≫
- ⋆ Cout : environ $|E|^2$

Propriété 8.7: Le tri par sélection est stable (si on extrait le premier min) mais ni en ligne, ni en place, ni facilement parallélisable

Exercice 8.8: Réécrire le tri pour qu'il soit en place. Qu'advient-il de la stabilité ?

II Tri par insertion

Algorithme 8.2 : $Tri\_par\_insertion$

pour i allant de 0 à $|E| - 1$ **faire**
  $v \leftarrow E[i]$
  $j \leftarrow i - 1$
  tant que $j > 0$ et $E[j] > E[j + 1]$ **faire**
    Échanger $E[j]$ et $E[i]$
    $j \leftarrow j - 1$
retourner E

Propriété 8.9: L'algorithme 8.2 est stable, en place et en ligne.

Remarque 8.10: Il est difficilement parallélisable

Exercice 8.11: Quel est la complexité de cet algorithme

III Recherche Dichotomique

Algorithme 8.3 : $Recherche\_dichotomique(E, x)$

```
si |E| = 0 alors
    retourner Faux
sinon
    a ← 0, b ← |E| − 1, m ← ⌊(a + b)/2⌋
    tant que E[m] ≠ x et (b − a) > 0 faire
        si E[m] < x alors
            a ← m + 1
        sinon
            b ← m − 1
        m ← ⌊(a + b)/2⌋
    si E[m] = x alors
        retourner Vrai
    sinon
        retourner Faux
```

Terminaison : variant de boucle : $\ll b - a \gg$
Correction : invariant de boucle : $\ll$Si x est dans E alors son indice est entre a et $b\gg$
Cout : nombre de bit de $|E|$

Implémentation 8.12: Implémentation en Python et comparaison avec la recherche linéaire

Exercice 8.13: Écrire une version récursive de l'algorithme

3 Tri efficace

I Tri fusion

Définition 8.14: L'algorithme de tri fusion est un algorithme de tri qui repose sur deux opérations :

- $partitionner(L)$: coupe L en deux listes de taille équivalente L_1, L_2
- $fusion(L_1, L_2)$: avec L_1 et L_2 deux listes triées, renvoie $L3$ la liste triée contenant tous les éléments de L_1 et L_2.

Algorithme 8.4 : 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 8.5 : 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 du tri fusion

Propriété 8.15: Ce tri est stable mais pas en place. Parallélisable mais pas en ligne.

Complexité : $C(n) = 2 \times C(\frac{n}{2})$ donc $C(n) = O(n \times \log n)$

II Tri rapide

Idée 8.16: Choisir un élément appelé pivot et mettre à gauche tous les éléments plus petit, à droite tous les plus grands, puis à trier cette partie à droite et à gauche.

Algorithme 8.6 : $tri\_rapide(L, debut, fin)$

```
si debut ≥ fin − 1 alors
    retourner L
pivot ← L[0]
i ← debut
j ← fin
tant que i < j faire
    si L[i + 1] ≤ pivot alors
        échanger L[i + 1] et L[i]
        i ← i + 1
    sinon
        échanger L[i + 1] et L[j]
        j ← j − 1
    tri_rapide(L, debut, i − 1)
    tri_rapide(L, i + 1, fin)
```

Commentaire 8.3 *Par manque de place, on peut remplacer l'écriture de cet algorithme par un dessin expliquant l'idée, et mettre son écriture en exo*

Propriété 8.17 (Complexité):
- pire des cas : $O(n^2)$
- meilleur cas : $O(n \log n)$
- cas moyen avec pivot aléatoire : $O(n \log n)$
- pire cas avec pivot astucieux : $O(n \log n)$

Propriété 8.18: Ce tri est en place, non stable, non en ligne mais parallélisable

Commentaire 8.4 *On peut éventuellement mentionner que dans beaucoup d'implémentation, pour le pivot on prend quelques valeurs (exemple 5) et on prend la médiane de ces valeurs là (ainsi on est quasiment jamais dans le pire cas et sans devoir calculé la médiane). Mais avec l'avantage de rester en place*

Commentaire 8.5 *Pour le caractère en place, il faut discuter de la pile d'appel. A priori, l'espace supplémentaire utilisé est en $O(\log n)$ (voir $O((\log n)^2$ si on considère le stockage des indices mais là on part un peu trop loin)) et non $O(1)$, mais on s'en contente souvent.*

Exercice 8.19: Écrire une version stable de ce tri. Préserve-t-on le caractère en place.

III Tri par tas

Algorithme 8.7 : $tri\_tas(L)$

Insérer les éléments de L dans un tas initialement vide
Extraire successivement l'élément minimum du tas.

Exercice 8.20: Quelles propriétés vérifie ce tri ?

IV Minoration de la complexité du tri

Propriété 8.21: On ne peut pas trier n éléments avec une complexité inférieure à $n-1$

Propriété 8.22: Un algorithme de tri par comparaison aura une complexité pire cas au mieux $O(n \log(n))$

Démonstration. S'intéresser à l'arbre des chemins que prend l'algorithme en fonction du résultat des comparaisons. □

Remarque 8.23: Si on ne trie pas par comparaison, on peut avoir des complexités plus faibles. Par exemple, si E ne contient que des entiers entre 0 et 9, on peut simplement compter le nombre de 0 et de 9 puis reconstruire là dessus le tableau trié.

Développement : Amélioration du tri par comptage avec des dictionnaires

4 Application

I Algorithmes gloutons

Définition 8.24: Un algorithme glouton est un algorithme qui résout un problème d'une entrée E de la forme :

— on trie les éléments de E

— on construit une solution en les parcourant, sans revenir en arrière

Exemple 8.25: On a n évènements sportifs ayant lieu respectivement entre les dates d_i et f_i ($i \in \mathbb{N}$) ayant chacun besoin d'un gymnase. Comment allouer des gymnases à des évènements pour utiliser le moins possible de gymnase ?

Algorithme glouton :

1. Trié les évènements par d_i croissant (impliquant souvent de trier)
2. Mettre chaque évènement dans le premier gymnase vide (peut en être un nouveau)

Propriété 8.26: Le glouton de l'exemple 8.25 renvoie une solution optiamle (minimisant le nombre de gymnases)

Exercice 8.27: Écrire un algorithme glouton pour le problème de rendu de monnaie

Définition 8.28: Un arbre couvrant minimal d'un graphe pondéré est un sous graphe connexe acyclique de poids minimal

Algorithme 8.8 : $kruskal$

Entrées : L liste d'arêtes
Sorties : Arbre couvrant de poids minimal
Trier L
$res \leftarrow []$ **pour** $a \in L$ **faire**
  si $\{a\} \cup res$ n'ajoute pas de cycles **alors**
    $res \leftarrow res \cup \{a\}$
retourner res

Exercice 8.29: Proposer des algorithmes gloutons pour la coloration de graphe. Sont-ils optimaux ?

II Implémentation des ensembles

Commentaire 8.6 *Cette partie fait un peu écho à l'activité mentionner dans le commentaire 8.1*

Propriété 8.30: On peut implémenter les ensembles par des listes, par des listes triées, par des listes que l'on trie pour l'union et l'intersection. On obtient alors les complexités suivantes :

Structure	Insertion	Suppression	Recherche	union / intersection
liste	$O(n)$	$O(n)$	$O(n)$	$O(n \times m)$
liste triée	$O(n)$	$O(n)$	$\log n$	$O(n+m)$
liste avec tri	$O(1)$	$O(n)$	$O(n)$	$O((n+m)\log(\min(n,m)))$

Exercice 8.31: Proposer une implémentation mélangeant les listes triées et les listes avec tri ayant de meilleur complexité

Commentaire 8.7 *On s'attend ici à ce que on aient d'une part les éléments déjà triés, de l'autre les éléments pas encore, et on ne trie que quand on fait l'union ou l'intersection (ou éventuellement la recherche) en triant les éléments pas encore triés, puis en fusionnant (cf tri fusion) les deux listes.*

Leçon 9

Algorithmique du texte. Exemples et applications.

Auteur·e·s: Emile Martinez
Références :

Un auteur envoie son manuscrit à un éditeur. Il commence par compresser le texte. A la réception, l'éditeur compare le manuscrit à d'autres ouvrages pour vérifier qu'il n'y a pas plagiat. Il peut ensuite chercher un motif dans le texte.

Commentaire 9.1 *Cela justifie l'ordre dans lequel on a mis nos parties. De plus, suivant le niveau du cours, on pourrait inaugurer chaque partie en rappelant quelle partie de ce contexte cela concerne. Mais là le niveau (MPI) est un peu élevé pour s'apesentir sur ce genre de choses*

Notation : On prend les conventions de slicing de python

Commentaire 9.2 *Il n'y a pas de partie application car souvent les applications sont directes depuis les algorithmes. Ainsi on les dissémine illustrant directement le concept*

1 Encodage et compression

I Encodage par lettre

Définition 9.1 (Algorithme de compresssion): Un algorithme de compression sans perte est la donnée d'une fonction $f : \Sigma^* \to \Sigma^*$ injective (il existe un unique décodage).

Algorithme 9.2 (Codage préfixe):

1. Prétraitement : On construit un arbre binaire $\mathcal{A}$ qu'on appelle arbre de Huffman dont les feuilles sont les lettres $c \in \Sigma$.
2. Compression : On code chaque lettre c par une suite de 0 et 1 correspondant au chemin (0 : gauche, 1 : droite) de la racine à la feuille contenant c dans $\mathcal{A}$.
3. Décompression : On décode une suite de 0 et 1 en parcourant le chemin correspondant dans $\mathcal{A}$.

Algorithme 9.3 (Codage de Huffman): On note f_a la fréquence du caractère a dans le texte et on note $f_{\mathcal{A}} = \sum_{a \in \mathcal{A}} f_a$

Créer une forêt $\mathcal{F}$ d'arbres binaires réduits à un noeud (a, f_a)
tant que $|\mathcal{F}| > 1$ **faire**
  Extraire de $\mathcal{F}$ les arbres $\mathcal{A}_1$ et $\mathcal{A}_2$ de plus petites fréquences // $f_{\mathcal{A}_1} <= f_{\mathcal{A}_2}$ à la racine
  Insérer un arbre de racine $f_{\mathcal{A}_1} + f_{\mathcal{A}_2}$ ayant pour fils $\mathcal{A}_1$ et $\mathcal{A}_2$ dans $\mathcal{F}$
retourner $\mathcal{A}_H$ tel que $\mathcal{F} = \{\mathcal{A}_H\}$

Commentaire 9.3 *Mentionner à l'oral qu'on ferait le lien avec le fait que c'est un algorithme glouton sur les arbres unifiant ceux de fréquences des lettres agrégés minimal.*

Théorème 9.4: L'arbre construit par le codage de Huffman 9.3 minimise la quantité $S = \sum_c \in \Sigma f_c d_c$ où d_c est la profondeur du caractère c dans l'arbre.

Idée 9.5: Cela revient à dire que l'algorithme de Huffman est optimal parmi les codages préfixes

Exercice 9.6: Quelle structure pour $\mathcal{F}$ dans l'algorithme 9.3 ? Quelle complexité alors ?

II Encodage par séquences

Idée 9.7: On va associer un code à une séquence de lettres (ou motifs)

Remarque 9.8: Pour cela, on peut utiliser le codage de Huffman, en prenant comme alphabet les mots apparaissant dans le livre. Mais on peut faire quelque chose de spécifique aux séquences, mais moins spécifiques à un livre (car ici le caractère " " comme délimitateur est arbitraire).

Commentaire 9.4 *Si on manque de places, on peut faire cette remarque uniquement à l'oral*

Idée 9.9: L'agorithme LZW détermine un codage dans un dictionnaire d au fur et à mesure de la lecture du texte (algorithme online). On va coder certains motifs (groupement de lettres consécutives) présents dans le texte.

Algorithme 9.10 (Compression par LZW):

Initialement, les lettres de Σ sont codées par un entier (par ex. avec le code ASCII).
$m \leftarrow''$
tant que il reste du texte s à coder **faire**
  Retirer le plus long préfixe w de s qui soit dans d
  Ajouter le codage de w à la fin de m
  $w' \leftarrow w$ concaténé avec la prochaine lettre de s
  Ajouter un nouveau codage pour w' dans d
retourner m

Remarque 9.11: Pour la décompression, il n'est pas nécessaire de transmettre le dictionnaire car on peut le reconstruire à la volée.

Application 9.12: Le format de fichier .zip compresse un dossier en utilisant (entre autre) les algorithmes de Huffman et LZW.

Développement : Présentation de la compression et de la décompression de l'algorithme LZW

2 Comparaison

I Plus longue sous-suite commune (PLSSC)

Définition 9.13 (Problème PLSSC): Soit $x, y \in \Sigma^*$. Une plus longue sous-suite commune à x et y, noté $PLSSC(x,y)$ est $argmin\left\{k \,/\, \exists i,j : x[i:i+k] = y[j:j+k]\right\}$

Application 9.14: Ce problème correspond au fait de trouver des morceaux d'ADN communs entre deux séquençages.

Exemple 9.15: Pour $x =' patate'$ et $y =' frites'$ on a $'te'$

Algorithme 9.16 (Brute force): On essaie toutes les séquences possibles $\rightarrow$ exponentiel

Algorithme 9.17 (programmation dynamique): On considère les sous-problèmes $c_{i,j} = |PLSSC(x[:i], y[:j])|$.

On a alors $c_{i,j} = \begin{cases} 0 & \text{si } i = 0 \text{ ou } j = 0 \\ c_{i-1,j-1} & \text{si } x[i-1] = y[j-1] \\ \max(c_{i-1,j}, c_{i,j-1}) & \text{sinon} \end{cases}$

Propriété 9.18: L'algorithme 9.17 calcule $PLSSC(x,y)$ en $O(|x| \times |y|)$

Remarque 9.19: Pour obtenir la sous suite, on sauvegarde d'où l'on vient pour pouvoir reconstruire la solution

II Distance entre deux chaînes

Définition 9.20 (distance): Une distance sur un ensemble E est une application $d : E^2 \to R^+$ tq :

$$\begin{array}{ll} d(x,y) = d(y,x) & \text{(symétrie)} \\ d(x,y) = 0 \equiv x = y & \text{(séparation)} \\ d(x,z) <= d(x,y) + d(y,z) & \text{(inégalité triangulaire)} \end{array}$$

Définition 9.21 (Distance de Hamming): La distance de Hamming entre deux chaines de caractères de même taille est le nombre de caractères distincts.

Application 9.22: Dans un protocole réseau, on peut ajouter des bits de contrôle, ayant une information redondante avec les bits du message. Certains messages sont donc invalides (si les bits de contrôle ne correspondent pas). On prend alors le message valide ayant la plus petite distance de Hamming (et ainsi, on peut espérer retrouver le message corrigé).

Exemple 9.23: Pour ≪truc≫ et ≪troc≫ c'est 1.

Définition 9.24: La distance d'édition (ou de levenshtein) entre deux chaînes de est le nombre minimal de transformation pour passer de l'une à l'autre parmi ($a \in \Sigma$, $i \in \mathbb{N}$) :

- $ins_{a,i}$: insertion de a à la position i
- $sub_{a,i}$: substitution de la i-ème lettre par a
- sup_i : suppression de la i-ème lettre.

Application 9.25: On peut utiliser cette algorithme pour détecter des mutations ponctuelles dans des brins d'ADN, et ainsi essayer de les faire correspondre.

Application 9.26: Dans les logiciels de traitement de texte, les correcteurs orthographiques cherchent dans un dictionnaire le mot le plus proche de celui mal orthographié pour proposer une correction.

Propriété 9.27: La distance d'édition $lev : \Sigma^* \times \Sigma^* \to \mathbb{N}$ est une distance. De plus,

$$lev(u.a, v.b) = \min \begin{cases} lev(u,v) + (a \neq b) & \text{Rien, ou substitution si } a \neq b \\ lev(u, v.b) + 1 & \text{Suppression de } a \\ lev(u.a, v) + 1 & \text{Insertion de } b \end{cases}$$

Corollaire 9.28: Il existe un algorithme de programmation dynamique calculant $lev(a,b)$ en $O(|a| \times |b|)$

3 Recherche de motifs

I Recherche de motifs dans un texte

Définition 9.29 (Problème de recherche de motif): Pour $m \in \Sigma^*$, $t \in \Sigma^*$, m est il un sous mot de t ?.

Remarque 9.30: On peut éventuellement rajouter les questions «où ?» et «combien de fois»

Application 9.31: CTRL + F

Algorithme 9.32 (Solution naïve): On essaie toutes les positions possibles pour m dans t. L'algorithme est donc en $O(|m| * |t|)$

Propriété 9.33:

1. Le problème est équivalent à savoir si $t \in \Sigma^* m \Sigma^*$ qui est rationnel.
2. Déterminer si un mot est dans un langage rationnel est linéaire en la longueur du mot
3. On peut résoudre le problème en $O(|t|) + f(m)$ (où f peut être très grand)

Développement : Construction de l'automate des motifs

Idée 9.34: On essaye de décaler de plus de 1 quand on se trompe

Commentaire 9.5 *C'est ce qu'on fait avec l'automate des motifs. Mais on peut essayer de le faire directement sans passer par le formalisme des automates*

Algorithme 9.35: Boyer-Moore Amélioration de l'algorithme naif en décalant de plus de 1 à chaque fois qu'on se trompe

```
i ← 0 tant que i < |t| − |m| faire
    pour j allant de |m| − 1 à 0 faire
        si t[i + j] ≠ m[j] alors
            i ← i + decalage[t[i + j]]
            break
```

où $decalage[a]$: l'indice en partant de la fin de la dernière occurrence de a dans m ($|m|$ si a non présent)

Propriété 9.36: Complexité :
Pire des cas : $O(|t| \times |m|)$
meilleur des cas : $O\left(\frac{|t|}{|m|}\right)$ Précalcul : $O(|m|)$ (mais dans le meilleur cas en $O(|\Sigma|)$)

Remarque 9.37: On peut mélanger les deux idées pour obtenir quelque chose qui sera en général sous linéaire, mais au pire linéaire (en $|t|$ mais on augmente le cout du précalcul).

Idée 9.38: Dans l'algorithme de Rabin Karp, on compare directement t[i : i + —m—] avec m grâce à des fonctions de hachage (donc rapidement). Si les hachages sont égaux, on vérifie caractère à caractère, et sinon on incrémente i.

Remarque 9.39: Le hachage h défini par$h(t_0, ..., t_{|m|-1}) = \sum_{j=1}^{|m|-1} B^{|m|-1-j} \times t_j$ peut se calculer en $O(1)$ à partir du calcul précédent : $h(t_i+1, ...t_i+|m|) = B \times (h(t_i, ...t_i+|m|-1) - B^{|m|-1}t_i) + t_i + 1$

II Analyse lexicale

Application 9.40: Lors de la compilation d'un programme C, le compilateur reconnait des léxèmes définis par des langages réguliers.

Exemple 9.41: `if (abs < 5) abs += 52;` reconnu en IF LPAR VAR INF INT RPAR VAR PLUSEGAL INT

Algorithme 9.42:

- On ordonne les règles $e_i \to t_i$ et on construit $\mathcal{A}_i$ un automate reconnaissant $\mathcal{L}(ei)$
- On exécute en parallère les $\mathcal{A}_i$ sur le texte.
- Quand tous les automates sont bloqués on prend :
  - le préfixe du texte le plus long qui finit dans un état acceptant d'un automate
  - en prenant l'automate de priorité maximale si plusieurs reconnaissent le même préfixe
- On recommence sur le texte restant.

Commentaire 9.6 *Par manque de place on peut ici résumer plus succintement l'algorithme en disant simplement qu'on reconnait chaque lexème avec un automate, en parrallèle pour trouver le bon.*

Application 9.43: grep suivi d'une regexp en norme POSIX et de noms de fichiers affiche toutes les lignes des fichiers qui contiennent la regexp.

Application 9.44: En SQL, on peut comparer des attributs de type CHAR avec des expression régulières grâce au mot clef LIKE.

Exemple 9.45: SELECT prénom FROM clients WHERE nom LIKE "%on%" ;

Leçon 10

Arbres : représentations et applications

Auteur·e·s: Emile Martinez
Références :

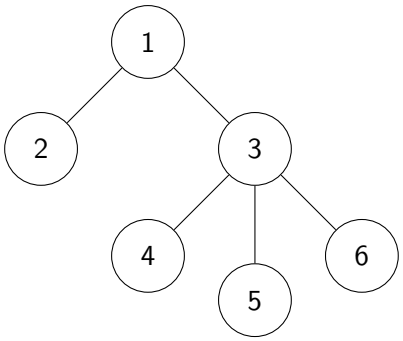
1 Généralités sur les arbres

Un arbre est une structure de données récursives stochant hierarchiquement les données. Dans un arbres, les données sont appelées des noeuds.

Définition 10.1 (arbre): Un arbre est un couple (u, l) où u est un noeud (élément) et l une liste (potentiellement vide) d'arbre

- u est appelé la racine de l'arbre (u, l)
- si l est vide, u est appelé une feuille
- les arbres de l, sont appelés sous arbres de (u, l)
- Les racines des arbres de l sont appelées enfant de u (et ont u pour parent)
- la hauteur $h(u) = h((u, l)) = 1 + \max_{A \in l} h(A)$ (et donc 1 si l est vide)

Exemple 10.2: $(1, [(2, []), (3, [(4, [])(5, []), (6, [])])])$ $\longrightarrow$



Remarque 10.3: Il n'existe ici pas d'abre vide.

Exemple 10.4: L'organisation des fichiers en répertoire peut être représenté sous forme d'arbre.

Remarque 10.5: On peut à chaque noeud associer une étiquette. On parle alors d'arbre étiqueté.

Théorème 10.6: Identifions les arbres à des graphes non orientés où un sommet (la racine) est choisi, en considérant les liens de parentés comme des arêtes. Les arbres sont alors les graphes connexes acycliques

Démonstration. Exercice □

Corollaire 10.7: On représente les arbres comme des graphes (cf exemple 10.2)

Définition 10.8: Un arbre binaire est définit inductivement par :

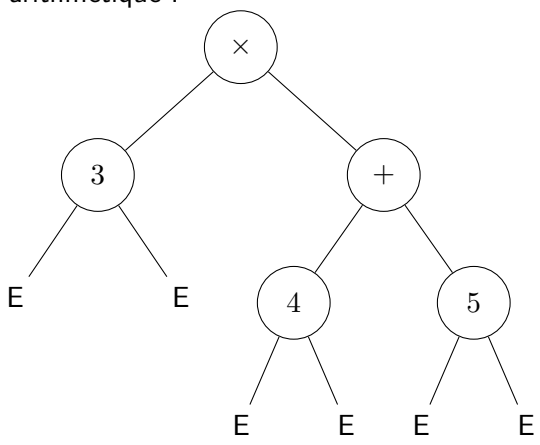
— E est un arbre vide

— Si A_1 et A_2 sont des arbres binaires et u une donnée, $B(u, A_1, A_2)$ est un arbre binaire, A_1 étant le sous-arbre gauche, et A_2 le droit.

Idée 10.9: Les arbres binaires sont des arbres dont les listes sont de taille 2, mais où l'on rajoute des arbres vides.

Exemple 10.10: Représentation d'une expression arithmétique :

$3 \times (4 + 5)$
$\to N(\times, N(3, E, E), A)$
avec $A = N(+, N(4, E, E), N(5, E, E))$ $\to$



2 Représentation informatique

I Représentation comme structure inductive

En représentant un arbre en utilisant sa structure inductive, on obtient :

- **Pour les arbres généraux**

```
type 'a arbre = N of 'a arb_liste and
type 'a bin arb_liste = V | Cons of (a arb *('a arb_liste));;
```

- **Pour les arbres binaires**

```
type 'a bin = E | B of 'a * 'a bin * 'a bin;;
```

Développement 1 : Correspondance entre les arbres binaires et les arbres généraux de taille n et utilisation en C

II Représentation comme un graphe

Les arbres étant des graphes connexes acycliques (cf. théorème 10.6), on peut les représentés comme tels :

— par des listes (ou dictionnaires) d'adjacence

— Par une matrice d'adjacence

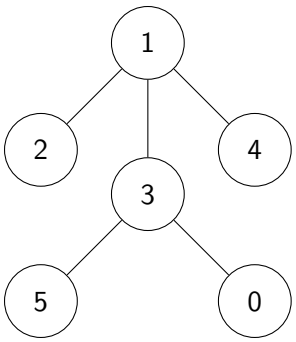
— Comme la liste de toutes les arêtes

Remarque 10.11: N'exploitant pas la structure d'arbres, ces structures sont peu utilisées.

III Représentation par un tableau

SI les identifiants des noeuds sont des entiers de 0 à $n-1$, on peut stocker l'arbre dans un tableau de taille n, la case i contenant l'identifiant du père du noeud i, -1 si il est racine.

Exemple 10.12:



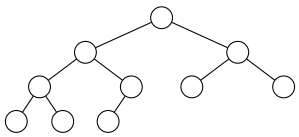
3	-1	1	1	1	3
0	1	2	3	4	5

Remarque 10.13: On ne stocke ici que la structure. Pour stocker des données, on produit un tableau de couples.

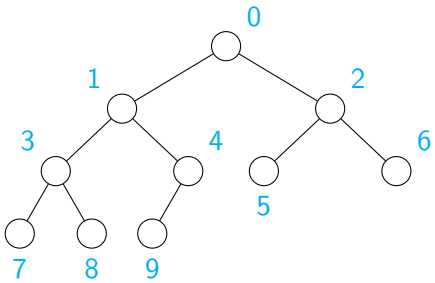
IV Représentation d'un arbre binaire presque complet

Définition 10.14 (Arbre binaire presque complet):

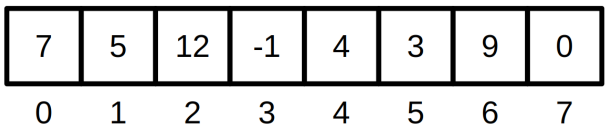
Un arbre binaire presque complet est un arbre binaire dont tous les étages sont remplies sauf éventuellement le dernier qui est alors rempli à gauche

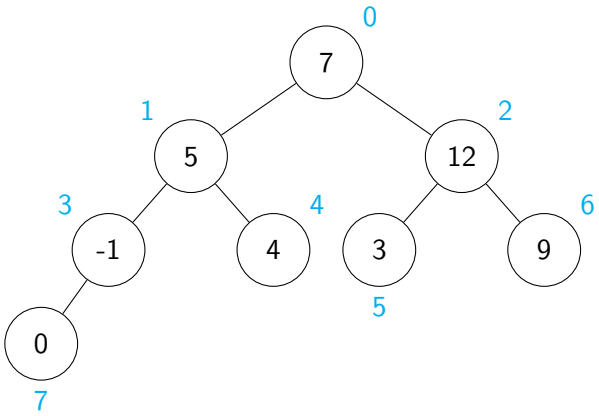


Principe 10.15: On peut alors représenter un tas par un tableau. On numérote alors les sommets ci contre, donnant l'indice dans le tableau. Les fils du noeuds d'indice i se retrouve aux cases $2\times i+1$, $2\times i+2$, et son père $\left\lfloor \frac{i-1}{2} \right\rfloor$



Exemple 10.16:





3 Application

Commentaire 10.1 *A chaque fois on va écrire en début quelle représentation on utilise pour notre application. Néanmoins, en classe, on pourrait le laisser proposer par les élèves (une fois l'application présentée). On justifie de ce fait notre organisation, ou les applications arrivent toutes ensembles à la fin. (mais on écrit quand même la représentation au début pour que ce soit plus facile de naviguer à la partie que l'on souhaite).*

I Dictionnaires

Représentation : Structure inductive

Définition 10.17: Un arbre binaire de recherche $N(x, G, D)$ est un arbre binaire de recherche (ABR) si G et D sont des ABR et si, selon un attribut a, $\max_{u \in G} u.a \leq x.a \leq \min_{v \in D} v.a$ (avec E qui est un ABR et $\max_{u \in E} u.a = -\infty$ et $\min_{u \in E} u.a = +\infty$)

Propriété 10.18: Chercher et insérer dans un ABR est en $O(h)$

Propriété 10.19: En imposant des contraintes supplémentaires (exemple arbres rouge-noir), on peut forcer $h = O(\log n)$

Définition 10.20: Un dictionnaire (ou tableau associatif) est un tableau où les indices (clés) ne se limite pas à $[\![0, n]\!]$

Idée 10.21: On peut alors implémenter un dictionnaire par un ABR, les clés étant les étiquettes des nœuds, à condition d'avoir un ordre total sur les clés

Théorème 10.22: Une implémentation efficace des dictionnaires par ABR permet des opérations de base en $O(\log n)$

Commentaire 10.2 *Si on ne veut pas faire le développement 2, et qu'on préfère celui sur les arbres k-dimensionnel, on peut aussi parler aussi du pb des k-plus proches voisins et parler des ABR comme des arbres k-dim pour résoudre le pb*

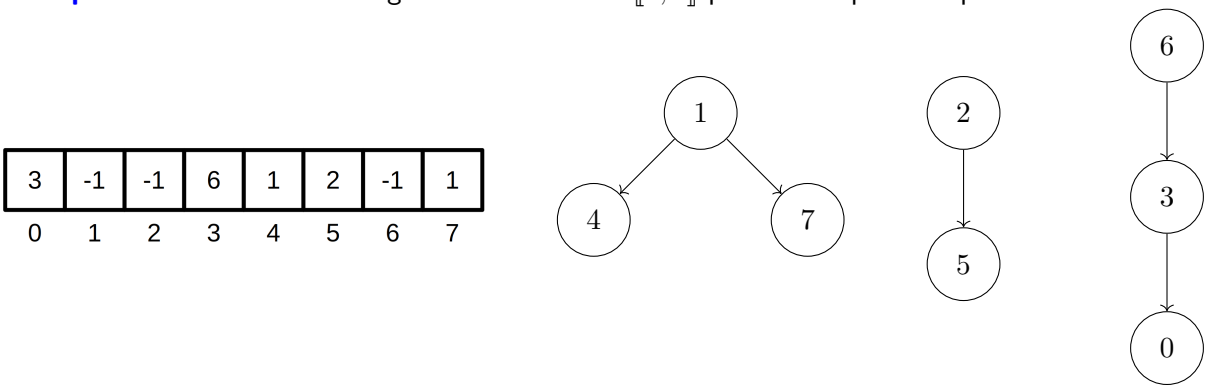
II Classes d'équivalence

Représentation : Tableaux

Définition 10.23: Une structure union & trouver est une structure implémentant les classes d'équivalence, permettant de trouver les représentants d'une classe et d'unir deux classes.

Idée 10.24: On implémente cette structure sous formes d'un ensemble d'arbres (forêt), où chaque arbre représente une classe et où la racine est le représentant de la classe

Exemple 10.25: La relation d'égalité modulo 3 sur $[\![0, 7]\!]$ peut être représenté par :



Propriété 10.26: Si on unifie en faisant pointer la racine de l'arbre le moins profond vers celle de l'arbre le plus profond, on obtient des opérations unir et trouver en $O(\log(n))$

III Arbres couvrant de poids minimal

Représentation : liste des arêtes

Définition 10.27: Dans un graphe pondéré connexe, un arbre couvrant de poids minimal, est un sous-ensemble maximal d'arêtes sans cycles de somme de poids minimal.

Algorithme 10.1 : $Kruskal(L)$

Entrées : L liste des arêtes
Sorties : Liste des arêtes d'un ACM
$L \leftarrow L$ trié
$res \leftarrow []$
pour a parcourant L **faire**
  si a ne rajoute pas de cycle dans res **alors**
    Ajouter a à res
retourner res

Propriété 10.28: En implémentant la détection de cycle par une structure union & trouver, Kruskal renvoie un arbre couvrant de poids minimal en $O(|L| \times \log |L|)$

IV Files de priorité

Représentation : celles des arbres semi-complet

Définition 10.29: Une file de priorité est une séquence de données dont on peut extraire la donnée d'attribut minimum, et rajouter des données.

Idée 10.30: On peut implémenter les files de priorité par des tas min

Définition 10.31: Un tas min est un arbre semi-complet où chaque noeud a un attribut plus petit que ses fils

Développement 2 : Correction de l'insertion dans un tas min et discussion sur l'implémentation

Principe 10.32:

- ⋆ Pour ajouter un élément, on le met au bout du tablea et on l'échange avec son père tant qu'il est plus petit
- ⋆ , Pour extraire le min, on met le dernier élément du tableau au début et on l'inverse avec le plus petit de ses fils tant qu'il est plus grand.

Propriété 10.33: Cette implémentation permet des opérations en $O(\log n)$

Application 10.34: Trier par tas en $O(n \log n)$

Leçon 11

Exemples d'algorithmes d'approximation et d'algorithmes probabilistes

Auteur·e·s: Emile Martinez
Références :

Beaucoup de problèmes sont durs à résoudre. On cherchera alors à sacrifier un peu de la fiabilité du résultat pour gagner en complexité. Il y a deux manières de faire un tel compromis :

— utiliser des algorithmes probabilistes (pouvant prendre beaucoup de temps, ou renvoyer faux)

— n'utiliser des algorithmes ne fournissant que des solutions approchées.

1 Algorithmes probabilistes

Définition 11.1: Un algorithme est déterministe si, pour une entrée x donnée, il s'exécute toujours de la même manière. En particulier, la sortie ne dépend que de l'entrée.

Un algorithme probabiliste est un algorithme dont l'exécution dépend de son entrée x et de valeurs obtenues via un générateur de nombre (pseudo)-aléatoire.

Exemple 11.2: Recherche dans un tableau T de booléens d'un indice i tel que $T[i] == True$

Algorithme 11.1
: $deterministe(T)$

```
pour i de 1 à |T| faire
    si T[i] alors
        retourner True
retourner False
```

Algorithme 11.2
: $probabiliste(T, k)$

```
pour j de 1 à k faire
    Tirer uniformément i dans ⟦1, n⟧
    si T[i] alors
        retourner True
retourner False
```

Remarque 11.3: Soit $p < 1$ la proportion de booléen à Faux dans T. Alors l'algo probabiliste se trompe avec une probabilité $= p^k$. Pour $p = \frac{1}{4}$ et $k = 5$, cela fait 10^{-3}

I Algorithme de type Monte-Carlo

Définition 11.4: Un algorithme probabiliste A pour un pb P est de type Monte-Carlo si pour toute instance i de P,

— $A(i)$ est une solution, erronée avec une certaine probabilité

— Le temps d'exécution de A sur i est indépendant des choix aléatoires.

Exemple 11.5: L'algorithme probabiliste de l'exemple 11.2 est de type Monte-Carlo

Développement 1 : Vérification probabiliste du produit matriciel

Propriété 11.6 (Amplification): Soient $0 < \varepsilon_2 < \varepsilon_1 < 1$.
S'il existe un algorithme Monte Carlo pour un problème Π ayant une probabilité d'erreur ε_1, alors on peut construire un algorithme Monte Carlo pour le problème Π ayant une probabilité d'erreur ε_2, en appliquant plusieurs fois l'algorithme initial.

Algorithme 11.3 : $mediane(L)$

$n \leftarrow |L|$
$L_0 \leftarrow k = n^{\frac{3}{4}}$ éléments de L aléatoirement
Trier L_0
$x_1 \leftarrow L_0[\frac{k}{2} - \sqrt{n}]$
$x_2 \leftarrow L_0[\frac{k}{2} + \sqrt{n}]$
$L_1 \leftarrow$ liste des éléments $< x_1$
$L_2 \leftarrow$ liste des éléments entre x_1 et x_2
$L_3 \leftarrow$ liste des éléments $> x_2$
si $|L_1| > \frac{n}{2}$ ou $|L_3| > \frac{n}{2}$ **alors**
  retourner n'importe quoi; `// $L_2$ n'a pas capturé la médiane`
si $|L_2| > n^{\frac{3}{4}}$ **alors**
  retourner n'importe quoi; `// On a trop d'éléments dans` L_2 `pour les trier`
Trier L_2
retourner $L_2[\frac{n}{2} - |L_1|]$

Idée 11.7: On fait la médiane sur un échantillon de L. Puis on prend tous les éléments autour de cette médiane et on les trie pour trouver la vraie médiane (si elle y est, sinon tant pis).

Commentaire 11.1 *Ici le temps dépend du hasard. On considère quand même que c'est un Monte Carlo car c'est borné et prend le temps moyen est un Ω du temps dans le pire des cas. On pourrait aussi a la place de retourner n'importe quoi, trier $n^{3/4}$ éléments et remplir L_2 avec des $= \infty$ mais alors on complexifie pour rien.*

II Algorithmes de type Las Vegas

Définition 11.8: Un algorithme probabiliste A est de Las Vegas si :
— Si A termine, alors la solution renvoyée est correcte.
— Le temps d'exécution de A est une variable aléatoire.

Remarque 11.9: Quand on parle de complexité en moyenne pour un algorithme de Las Vegas, on ne considère pas (ou pas uniquement) la moyenne des complexité sur toutes les entrées possibles prisent uniformément, mais la plus grande espérance du temps d'exécution sur une entrée

Algorithme 11.10:

Algorithme 11.4 : $tri\_rapide\_randomise(T)$

si $|T| = 1$ **alors**
  retourner
Tirer q uniformément dans $[\![1, |T|]\!]$; // $T[q]$ sera le pivot
$i \leftarrow partition(T, q)$
$tri\_rapide\_randomise(T[:i])$
$tri\_rapide\_randomise(T[i+1:])$

où $partition(T, q)$ mets dans T, tous les éléments $< T[q]$ puis tous les éléments supérieurs, et renvoie l'indice du milieu.

Remarque 11.11: Le temps d'exécution de l'algorithme 11.10 dépend du choix du pivot : $O(n^2)$ dans le pire cas, $O(n \log n)$ dans le meilleur et en moyenne.

Remarque 11.12: Il est possible de transformer un algorithme de type Las Vegas en Monte Carlo en l'exécutant pendant un temps défini et en générant une réponse aléatoire s'il n'a pas terminé.

Remarque 11.13: Si on peut vérifier efficacement la validité du résultat, on peut également faire l'inverse.

Exemple 11.14: A la place de l'échec dans l'algorithme 26, on peut relancer l'algorithme. On obtient alors un algorithme trouvant toujours la médiane, la calculant en moyenne en $1,5n$ comparaisons, sans adversaires, ce qui est mieux que tout algorithme déterministe (qui ont nécessairement besoin d'au moins $2n$ comparaisons en moyenne).

2 Algorithmes d'approximation

I Définition

Définition 11.15: Un problème d'optimisation est un problème $\Pi = (I, S, c)$ où

- I est l'ensemble des instances
- $\forall i \in I, S(i)$ est l'ensemble des solutions pour i
- $c : I \times S \to R$ fonction d'évaluation.

Étant donnée une instance $i \in I$, l'objectif est de construire une solution $s^* \in S(i)$ vérifiant : $c(i, s^*) = \min\{c(x, s)/s \in S(i)\}$

Commentaire 11.2 *On peut aussi demander à ce que c soit calculable en temps polynomial mais ce n'est pas nécessaire. Les deux définitions coexistent.*

Remarque 11.16: On se ramène au max en prenant $-c$.

Définition 11.17: Une λ-approximation est un algorithme polynomial A donnant pour chaque instance

i de Π une solution tel que $\max\left(\left|\frac{A(i)}{OPT(i)}\right|, \left|\frac{OPT(i)}{A(i)}\right|\right) \leq \lambda$

II Exemples

Définition 11.18 (Couvertue par des sommets (Vertex-Cover)): Entrée : Un graphe $G = (S, A)$
Solution : Un ensemble $S \subset V$ tel que $\forall u, v \in E$, $v \in S$ ou $u \in S$

Algorithme 11.5 : Glouton Vertex Cover

$S \leftarrow \{\}$
tant que il existe une arête $(u, v) \in A$ tel que $u \notin S$ et $v \notin S$ **faire**
  Choisir (u, v) une telle arête
  $S \leftarrow S \cup \{(u, v)\}$
retourner S

Propriété 11.19: L'algorithme 28 est une 2-approx du problème de la couverture par les sommets.

Développement : Une $\frac{3}{2}$-approximation et une $\frac{7}{6}$-approximation gloutonnes pour le problème d'ordonnancement de tâches indépendantes sur 2 processeurs.

Définition 11.20 (Voyageur de commerce (TSP)): Instance : $G = (S, A, c)$ un graphe orienté complet pondéré
Solution : Un cycle hamiltonien sur G de poids minimal

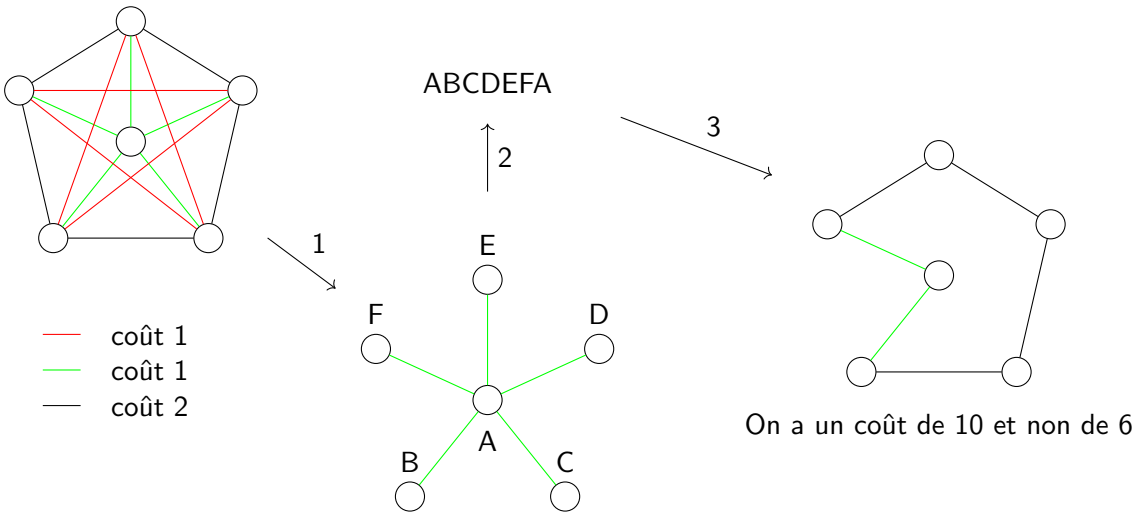
Théorème 11.21: Il n'existe pas d'algorithme d'approximation pour TSP, sauf si $P = NP$.

Algorithme 11.22 (Algorithme pour TSP):

1. $\mathcal{A} \leftarrow$ arbre couvrant de poids minimal de G
2. $P \leftarrow$ parcours en profondeur préfixe de $\mathcal{A}$
3. Renvoyer P

Propriété 11.23: L'algorithme 11.22 est une 2-approx si c vérifie l'inégalité triangulaire ($\forall x, y, z \in C$, $c(x, y) + c(y, z) \geq c(x, z)$)

Exemple 11.24:



3 Algorithmes d'approximation probabilistes

Il existe des algorithmes d'approximation probabilistes. Ce sont alors souvent des algorithmes de Monte Carlo.

I Max Sat

Définition 11.25: Instance : n variables $\{x_1, \dots, x_n\}$, p clauses $C_1, \dots, C_p$ sur $(x_1, \dots, x_n)$ contenant au moins k littéraux Solution : Trouver une valuation maximisant le nombre de clauses à vrai

Exemple 11.26: Si $\varphi = \bigwedge_{i=1}^{p} C_i$ est satisfiable, une solution optimale est une valuation satisfaisant φ

Algorithme 11.27: Renvoyer une valuation aléatoire.

Théorème 11.28: L'espérance du nombre de clauses satisfaites $\mathbb{E}(\varphi)$ vérifie $\mathbb{E}(\varphi) \geq p\left(1 - \frac{1}{2^k}\right) \geq \frac{c}{2}$

Remarque 11.29: On peut créer un algorithme déterministe qui est une $\left(1 - \frac{1}{2^k}\right)$-approx.

II Calcul de π

On peut calculer une valeur approchée de π par des méthodes probabilistes

Commentaire 11.3 *On peut aussi mettre un dessin ici pour expliquer ce qui se passe.*

Commentaire 11.4 *Si on veut parler plus précisément d'algorithme d'approximation, on prendrait dans la def, pour I la manière de représenter les flottants (nb de bits d'exposant, de mentisse, etc...) et pour S de tels flottants. La fonction de coût c serait alors la distance à pi (même si pas facilement calculable en temps polynomial)*

Algorithme 11.6 : $calcul\_pi(N)$

$i \leftarrow O$
répéter
  Tirer deux flottants x, y dans $[-1, 1]$ uniformément
  si $x^2 + y^2 < 1$ **alors**
    $i \leftarrow i + 1$;
// On a tapé dans le cercle d'aire π
jusqu'à N fois;
retourner $\frac{4i}{N}$; // Nombre de points dans le cercle / aire du carré = $\frac{\pi}{4}$

Remarque 11.30: On peut généraliser la méthode pour le calcul d'intégrales

Remarque 11.31: Cette méthode se généralise en une méthode de Monte Carlo où l'on échantillone le problème, on calcule la solution sur cette échantillon, puis on généralise la solution à partir de cet échantillon.

Leçon 12

Exemples d'algorithmes glouton et de retour sur trace

Auteur·e·s: Emile Martinez
Références :

Commentaire 12.1 *L'idée du plan, c'est de voir des gloutons exactes. Dans l'idée, on fait des choix locaux et ca marche. On finit par un problème qui nous dit que en fait des gloutons aussi simples, ne peuvent pas tout faire. Des fois, c'est impossible en temps polynomial. Donc, au lieu de faire des choix définitifs, on s'autorise à revenir sur nos choix, et donc à tout essayer. Mais ensuite, on se rend compte que tout essayer c'est trop long, donc on va s'autoriser à ne pas trouver la valeur exacte, mais seulement une solution approchée.*

Quand on aborde un problème à la main, on essaye souvent de construire des solutions au fur et à mesure, en faisant plein de choix locaux. Essayons de mettre cela en oeuvre algorithmiquement.

1 Algorithmes gloutons

I Définition

Définition 12.1: Un algorithme glouton construit une solution à un problème par choix successifs considérés localement optimaux, sans jamais revenir en arrière.

Avantage : C'est souvent peu coûteux, et potentiellement en ligne.

Définition 12.2 (Gymnases):
Instance : n évènements, leur date de début $\{d_i\}_{i\in\{1,\dots,n\}}$ et de fin $\{f_i\}_{i\in\{1,\dots,n\}}$
Problème : trouver un nombre minimal de gymnases pour organiser les évènements

Algorithme 12.3:

1. Trier les évènements par dates de début croissantes
2. Allouer successivement le premier gymnase disponible. En ouvrir un si nécessaire.

Propriété 12.4: L'algorithme 12.3 est optimal et en $O(n \log n)$

Démonstration. On prend une solution optimal, et on montre qu'on la transforme en notre solution gloutonne, sans augmenter le nombre de gymnases. □

Remarque 12.5: Ce type de preuve est souvent généralisable.

II Exemples

Définition 12.6 (Arbre couvrant de poids minimal):
Instance : Un graphe $G = (S, A)$ non orienté, $w : A \to \mathbb{N}$
Problème : Trouver un arbre couvrant S de poids minimal

Algorithme 12.1 : $Kruskal(G)$

Trier les arêtes par poids croissants
$L \leftarrow []$
pour a arête **faire**
  si $L \cup \{a\}$ ne contient pas de cycle **alors**
    Ajouter a à L
retourner L

Propriété 12.7: L'algorithme 30 est optimal

Remarque 12.8: Comment détécter l'absence de cycle ? En faisant les classes de connexité par union find.

Propriété 12.9: Si l'union find est implémenté avec le rang, on a O(—A— log(—A—))

Remarque 12.10: l'algorithme de Huffman est glouton

Définition 12.11 (Indep2):
Instance : n tâches de poids $\{p_i\}_{i\in\{1,\dots,n\}}$ indépendantes
Problème : Allouer les tâches sur 2 processeurs en minimisant la date de fin.

Algorithme 12.12: Allouer les tâches au fur et à mesure, au processus le moins plein

Propriété 12.13: Cet algorithme est en $O(n)$ et est en ligne.

Propriété 12.14: Si $pi = 1$, l'algorithme est optimal.
Dans le cas quelconque, l'algorithme n'est pas optimal.

Définition 12.15 (Somme Sous Ensmble (Subset Sum)):
Instance : n entiers $\{s_i\}$, $K \in \mathbb{N}$
Problème : Existe-t-il $I \subset \{1, \dots, n\}$ tel que $\sum_{i \in I} s_i = K$

Remarque 12.16: Savoir s'il existe un ordonnancement parfait sur 2 machines revient à résoudre SSE avec $K = \frac{\sum p_i}{2}$ or ce problème est dur à résoudre (NP-complet, c'est 2-Partition).

2 Retour sur trace

La stratégie gloutonne (notamment le fait de ne jamais revenir sur un choix) est parfois trop restrictive.

I Définition

Définition 12.17: Un algorithme de retour sur trace fait tous les choix possibles, jusqu'à pouvoir résoudre le problème.

Remarque 12.18: Si on enlève le tant que de la définition, on tombe sur un algorithme de force brute.

Algorithme 12.2 : $SSE(X, S)$

Entrées : X une famille de $\mathbb{N}$
$S \in \mathbb{N}$ la cible
Sorties : Booléen : $\ll \exists I : \sum_{i \in I} X[i] = S \gg$

si $S = 0$ **alors**
  retourner Vrai
si $S < 0$ ou $X =$ **alors**
  retourner Faux
Choisir $x \in X$
retourner $SSE(X \backslash \{x\}, S - x)$ ou $SSE(X \backslash \{x\}, S)$

Principe 12.19 (Généralisation du principe de retour sur trace):

Algorithme 12.3 : $RST(n)$

Entrées : n solution partielle
Sorties : booléen : Est-ce que on peut compléter n en une solution ?
si $est\_solution(n)$ **alors**
  retourner Vrai
si $impossible(n)$ **alors**
  retourner Faux
Choisir un choix
pour chaque option o possible **faire**
  si $RST(o)$ **alors**
    retourner Vrai
retourner Faux

II Construction de la solution au fur et à mesure

Commentaire 12.2 *Ici, on ne se limite plus implicitement comme précédemment aux problèmes de décision*

Remarque 12.20: Quand on parcourait récursivement les solutions, on modifie potentiellement les variables globales. Il faut alors penser à les remettre en état.

Algorithme 12.4 : $Sudoku(grille)$

```
Entrées : Tableau 9 par 9 sans contradiciton de Sudoku
si aucune case n'est à 0 alors
    retourner Vrai
i, j ← premier indice tel que grille[i][j] = 0 pour val de 1 à 9 faire
    si val n'est pas dans la ligne i, la colonne j ou le carré i/3 j/3 alors
        grille[i][j] ← val
        si Sudoku(grille) alors
            retourner Vrai
grille[i][j] ← 0    // rétablit la grille
retourner Faux
```

Remarque 12.21: Ici, on a seulement renvoyer si il y avait une solution, mais dans la grille on a une solution.

Exercice 12.22: Implémenter une solution du problèmes des N-reines en OCaml

Remarque 12.23: OCaml est adapté au retour sur trace : récursivité et structures immuables.

III Problèmes de terminaison

Définition 12.24 (Jeu du Taquin):
Instance : grille $n \times n$ contenant des entiers entre 1 et $n^2 - 1$ et un trou
Solution : La grille réordonnée (en ayant échanger le trou avec des voisins) avec si possible le trou en dernière position

Idée 12.25: L'algorithme 34 ne termine pas toujours.
Solution :

- Ajouter en entrée un nombre max d'itérations
- Mémoriser les positions déjà parcourues
- autre

Définition 12.26: Une **grammaire** est un quadruplet (Σ, V, R, S) avec :

- Σ : alphabet de terminaux
- V : ensemble fini de non terminaux

Algorithme 12.5 : $Taquin(grille)$

```
si grille est ordonnée alors
    retourner Vrai
i, j ← position du trou
pour i', j' case voisine de i, j faire // Au plus haut, bas, gauche et droite
    Échanger grille[i][j] et grille[i'][j']
    si Taquin(grille) alors
        retourner Vrai
    Échanger grille[i][j] et grille[i'][j']   // On rétablit la grille
retourner Faux
```

- $R \subset V \times (\Sigma \cup V)^*$: ensemble de règles
- $S \in V$: l'axiome

Une **dérivation** est une application successive de règles à des non terminaux du mot courant, en partant de l'axiome.
Si le mot ne contient plus que des terminaux, on dit que le mot est **accepté**.

Développement : Algorithme d'analyse syntaxique descendant par retour sur trace, illustrant les différents problèmes.

IV Point sur la complexité

Tous les algorithmes de retour sur trace précédents sont exponentiels. Pour certains problème on ne connaît pas d'algorithmes polynomial.

Commentaire 12.3 *On parle de cet algorithme car il est au programme et il paraît que l'endroit est tout indiqué pour le mentionner*

Définition 12.27 (SAT):
Instance : p clauses sur les variables $x_1, \dots, x_n$
Problème : Existe-t-il une valuation qui satisfait les p clauses.

Algorithme 12.6 : $Quine(C)$

```
Entrées : C liste de clauses
si C = [] alors
    retourner Vrai
si ⊥ ∈ C alors
    retourner Faux
x ← variable apparaissant dans C
// On affecte à x chaque valeur possible, en on simplifie les clauses avec
   cette valeur
si Quine(C[x ← ⊤]) alors
    retourner Vrai
si Quine(C[x ← ⊥]) alors
    retourner Vrai
retourner Faux
```

3 Algorithme d'approximation glouton

On peut alors (surtout pour des problèmes d'optimisation associés à des problèmes de décision NP-complet) se contenter d'approximation

I Définition

Définition 12.28: Un problème d'optimisation Π est un triplet (I, S, c) où :

— I est l'ensemble des instances

— $\forall i \in I, S(i)$ est l'ensemble des solutions réalisables

— $c : I \times S \in \mathbb{R}$ est une fonction d'évaluation

Le but est de trouver $\forall i \in I, s^* \in S(i)$ tel que $c(i, s) = \min\{c(i, s^*)/s \in S(i)\}$

Remarque 12.29: On se ramène à un problème de maximisation en prenant $c' \leftarrow -c$

Définition 12.30: Une λ-approximation est un algorithme polynomial $\mathcal{A}$ donnant pour chaque instance i de Π une solution telle que

$$\max\left(\left|\frac{\mathcal{A}(i)}{OPT(i)}\right|, \left|\frac{OPT(i)}{\mathcal{A}(i)}\right|\right) \leq \lambda$$

Exemple 12.31: L'algorithme 12.12 est une $\frac{3}{2}$-approximation dans le cas général.

Développement : Une $\frac{3}{2}$-approximation et une $\frac{7}{6}$-approximation pour Indep2.

II Exemples

Définition 12.32 (Couverture par des sommets):
Instance : Un graphe $G = (S, A)$ non orienté
Solution : $V \subset S$ tel que $\forall (u, v) \in A, u \in A$ ou $v \in A$ de taille minimale

Algorithme 12.7 : $Couverture\_par\_sommets\_glouton(G)$

$Couv \leftarrow \{\}$
tant que il existe $(u, v) \in A$ tel que $u \notin Couv$ et $v \notin Couv$ **faire**
  $Couv \leftarrow Couv \cup \{u, v\}$
retourner $Couv$

Propriété 12.33: L'algorithme précédent est une 2-approximation

Démonstration. $Couv$ est un ensemble d'arêtes indépendantes. L'optimal doit couvrir ces arêtes □

Définition 12.34 (Sac à dos):
Instance : n objets de poids $\{w_1, \ldots, w_n\} \in \mathbb{N}^n$, et de valeur $\{v_1, \ldots, v_n\} \in \mathbb{N}^n$. Une capacité $W \in \mathbb{N}$.

Problème : Trouver $I \subset \{1, \ldots, n\}$ tel que $\sum_{i \in I} w_i \leq W$ et qui maximise $\sum_{i \in I} v_i$

Algorithme 12.35: Algorithme glouton pour Sac à dos :

1. Trier les objets par $\frac{v_i}{w_i}$ décroissants
2. Prendre chaque objet s'il reste de la place

Propriété 12.36: Cette algorithme n'est pas une approximation

Algorithme 12.37: Prendre le maximum entre $\max_{i=1}^{n}(v_i)$ et le résultat de l'algorithme 12.35

Propriété 12.38: Cette algorithme est une 2-approximation.

Remarque 12.39: On peut également adapter l'idée du retour sur trace aux problèmes d'optimisation, en parcourant l'arbre des possibles, et en élaguant quand on a une meilleure borne. Cette stratégie s'appelle Séparation et Évaluation (Branch and Bound). Elle s'adapte bien au problème du sac à dos.

Leçon 13

Exemples d'algorithmes utilisant la méthode ≪ diviser pour régner ≫

Auteur·e·s: Emile Martinez
Références :

1 Introduction

Définition 13.1 (Paradigme Diviser pour Régner): Un algorithme de type Diviser pour Régner s'effectue en 3 étapes :

1. Division du problème en sous-problèmes indépendants
2. Résolution récursive des sous-problèmes
3. Construction d'une solution du problème global à partir des solutions des sous-problèmes

Principe 13.2 (Calcul de la complexité d'un tel algorithme.):
Trouver une fonction donnant la taille du problème (comme le nombre d'éléments pour une liste). Définir C la complexité maximale des instances de cette taille, puis trouver une relation de réccurence sur C, pour tenter de la résoudre.

Commentaire 13.1 *La méthode de résolution sera inculqué par l'exemple, et progressivement (d'abord le cas de base, où l'on découpe en seulement 2 avec l'exponentiation rapide, puis on complexifie)*

2 Applications au calcul formel

I L'exponentiation rapide

Problème : Étant donné un entier a et un entier positif n, calculer a^n.
Solution naïve : n multiplications

Algorithme 13.3 (Méthode D&R):

Algorithme 13.1 : $exponentiation\_rapide(a, n)$

si $n = 0$ **alors**
  retourner 1
$m \leftarrow m/2$; // étape 1
$x \leftarrow exponentiation\_rapide(a, m)$; // étape 2
si n est pair **alors** ; // étape 3
  retourner $x \times x$
sinon
  retourner $x \times x \times a$

Propriété 13.4 (Complexité): La complexité en nombre de multiplication par rapport à l'entier positif (noté $C(n)$) vaut $C(n) = O\big(\log(n)\big)$

Démonstration.

1. $C(n) = C\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + O(1)$
2. C est majoré par $D(n) = D\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + K$
3. D est croissante
4. $D\left(2^k\right)$ est une suite arithmétique donc $D\left(2^k\right) = O(k)$
5. $C(n) \underset{\uparrow 2}{\leq} D(n) \underset{\uparrow 3}{\leq} D\left(2^{\lfloor \log(n) \rfloor + 1}\right) \underset{\uparrow 4}{=} O(\log(n))$

□

II Multiplication matricielle

Problème : Étant donné $A = (a_{i,j})$ et B $= (b_{i,j})$ deux matrices de taille n, on cherche à calculer $A_{\times}B$
Solution naïve : $O(n^3)$

Algorithme 13.5 (Méthode D&R : Algorithme de Strassen):

1. Rajouter des 0 pour que A et B soient de tailles paires. Diviser alors A et B en matrices de taille $\frac{n}{2}$
$$A = \left(\begin{array}{c|c} A_{1,1} & A_{1,2} \\ \hline A_{2,1} & A_{2,2} \end{array}\right) \text{ et } B = \left(\begin{array}{c|c} B_{1,1} & B_{1,2} \\ \hline B_{2,1} & B_{2,2} \end{array}\right)$$
2. Calculer récursivement
$M_1 = (A_{1,1} + A_{2,2}) \times (B_{1,1} + B_{2,2})$
$M_2 = (A_{2,1} + A_{2,2}) \times B_{1,1}$
$M_3 = A_{1,1} \times (B_{1,2} - B_{2,1})$
$M_4 = A_{2,2} \times (B_{2,1} - B_{1,1})$
$M_5 = (A_{1,1} + A_{1,2}) \times B_{2,2}$
$M_6 = (A_{2,1} - A_{1,1}) \times (B_{1,1} + B_{1,2})$
$M_7 = (A_{1,2} - A_{2,2}) \times (B_{2,1} + B_{2,2})$
3. Calculer $A \times B = \left(\begin{array}{c|c} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ \hline M_2 + M_4 & M_1 - M_2 + M_3 + M_6 \end{array}\right)$

Exercice 13.6: Prouver la correction de l'algorithme de Strassen.

Propriété 13.7: Cette algorithme a une complexité en $O\left(n^{\log_2(3)}\right)$

Exercice 13.8: Prouver cela en reprenant la preuve pour l'exponentiation en considérant $\frac{D(2^k)}{7^k}$ à l'étape 4 de la preuve I

3 Application aux listes

I Recherche dichotomique

Problème : Rechercher un élément a dans une liste L d'éléments triés selon un ordre $\leq$

Algorithme 13.2 : $recherche\_dichotomique(l, a, debut, fin)$

Entrées : L une liste triée, a un élément
Sorties : i tel que $L[i] = a$ s'il existe, -1 sinon
si $fin < debut$ **alors**
  retourner -1
$m \leftarrow \left\lfloor \frac{debut + fin}{2} \right\rfloor$
si $l[m] = -a$ **alors**
  retourner m
si $l[m] < a$ **alors**
  retourner $recherche\_dichotomique(l, a, m + 1, fin)$
sinon
  retourner $recherche\_dichotomique(l, a, debut, m + 1)$

Exercice 13.9: Ecrire le code en plus de lignes et faire apparaître les 3 étapes de D&R

Propriété 13.10: La recherche dichotomique est en $O(\log |L|)$

Exercice 13.11: Écrire une version itérative de cet algorithme

II Tri fusion

Algorithme 13.3 : $fusion(L_1, L_2)$

Entrées : L_1, L_2 deux listes triées d'éléments comparables
Sorties : La fusion des listes L_1 et L_2 triée
$res \leftarrow []$
$i, j \leftarrow 0$
tant que $i < |L_1|$ et $j < |L_2|$ **faire**
  si $L_1[i] < L_2[j]$ **alors**
    res.ajouter($L_1[i]$)
    $i \leftarrow i + 1$
  sinon
    res.ajouter($L_2[j]$)
    $j \leftarrow j + 1$
Ajouter le reste de L_1 et de L_2 à res
retourner res

Algorithme 13.4 : $tri\_fusion(L)$

Entrées : Une liste L d'éléments comparables
Sorties : La liste L triée
$n \leftarrow |L|$
si $n \leq 1$ **alors**
  retourner L
$L_1, L_2 \leftarrow partionner(L)$; `// Étape 1`
$A \leftarrow tri\_fusion(L_1)$; `// Étape 2`
$B \leftarrow tri\_fusion(L_2)$ **retourner** $fusion(A, B)$

Propriété 13.12: Le tri fusion est $O(n \log n)$ avec $n = |L|$

Démonstration. Exercice □

Développement : Correction et terminaison du tri fusion

Application 13.13: Si l'on veut chercher si K entiers sont présents parmi N, on peut avec I et II faire cela en $O\big((N + K) \log N\big)$ au lieu de $O(k \times N)$

III Tri rapide

Objectif : Faire un tri D&R en place

Commentaire 13.2 *Ici, on peut avoir des questions sur le tri fusion en place, ce qui existe plus ou moins mais est assez pénible. Donc ca peut valoir le coup de se renseigner sur le sujet.*

Idée 13.14: Choisir un élément appelé pivot et mettre à gauche tous les éléments plus petit, à droite tous les plus grands, puis à trier cette partie à droite et à gauche.

Commentaire 13.3 *Par manque de place, on peut remplacer l'écriture de cet algorithme par un dessin expliquant l'idée, et mettre son écriture en exercice.*

Algorithme 13.5 : $tri\_rapide(L, debut, fin)$

```
si debut ≥ fin − 1 alors
    retourner L
pivot ← L[0]
i ← debut
j ← fin
tant que i < j faire
    si L[i + 1] ≤ pivot alors
        échanger L[i + 1] et L[i]
        i ← i + 1
    sinon
        échanger L[i + 1] et L[j]
        j ← j − 1
    tri_rapide(L, debut, i − 1)
    tri_rapide(L, i + 1, fin)
```

Propriété 13.15: Ce tri est dans le pire des cas en $O(n^2)$, dans le cas moyen en $O(n \log n)$. Il est néanmoins en place (en autorisant un espace non pas constant mais logarithmique).

Commentaire 13.4 *L'occasion de briller sur le fait que le tri rapide écrit comme ça n'est pas en place avec la définition première, mais l'est avec la définition élargie où on autorise $O(\log n)$ espace (utilisé par les appels récursifs)*

4 Application Géométrique

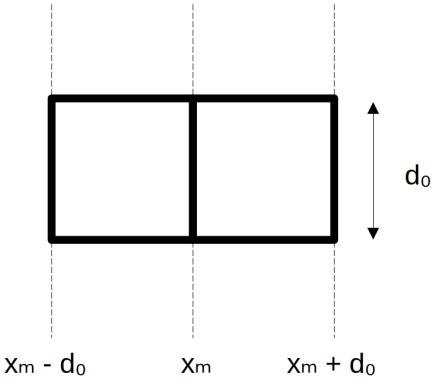
I Plus petite distance dans le plan

Problème : Étant donné n points de $\mathbb{R}^2$, donner la plus petite distance entre deux points
Solution naïve : n multiplications

Algorithme 13.16 (Solution D&R):

1. Choisir une abscisse x_m séparant les points en deux sous-ensembles P_1 et P_2 de taille égale (à + ou - 1)
2. Trouver les plus petites distances d_1 et d_2 de P_1 et P_2
3. Rechercher la plus petite distance d_3 entre deux points dans la bande d'abscisse $[x_m - d_0, x_m + d_0]$ pour $d_0 = \min d_1, d_2$

L'étape 3 peut se faire en ne regardant que les 7 points suivants (suivants au sens de l'ordonnée).



On a au plus 8 éléments dans ce rectangle, car dans chaque domaine, les points sont au moins à distance d_0. On a donc au plus 4 points par carré, d'où le résultat en mettant notre point sur le bas du rectangle.

Propriété 13.17: On a une complexité en $O\left(n(\log n)^2\right)$

II Arbre K-dimensionnel

Problème : Trouver les k plus proches voisins d'un point $y \in \mathbb{R}^K$ parmi un ensemble de n points $x_1, \ldots, x_n \in \mathbb{R}^K$

Solution initiale : Stocker nos k valeurs en cours dans une file de priorité et parcourir les n points, en mettant à jour la file de priorité. $\rightarrow O(n \log k)$

Algorithme 13.18 (Solution D&R): Faire un pré traitement où l'on stockera nos valeurs dans un arbre binaire de recherche, où l'on partitionnera récursivement les données alternativement sur chaque dimension. La recherche se fait alors en ne cherchant que d'un côté si le deuxième n'est pas nécessaire.

Développement : Présentation de la structure d'arbre K-dimensionnel

Propriété 13.19 (Complexité):

— prétraitement : $O(n \log(n))$

— Recherche des k plus proches voisins : $O(\log n \times k)$ en moyenne, $O(N \times K)$ dans le pire des cas

Remarque 13.20: On y perd pour une seule recherche, mais si on cherche les k-plus proches voisins de N points, on obtient du $O(n \log n + N \times k \times \log n)$ au lieu de $O(N \times n \times \log k)$.

Commentaire 13.5 *Si il reste beaucoup de places ou si on veut enlever un exemple au dessus (comme le tri rapide faisant doublons avec le tri fusion, même si il est très classique), on peut rajouter une section supplémentaire ici, et parler de l'additionneur à retenue anticipée par méthode D&R du développement *à compléter**

Leçon 14

Programmation dynamique

Auteur·e·s: Emile Martinez
Références :

Commentaire 14.1 *On présentera d'abord le principe de la méthode diviser pour régner. Ensuite on divise nos exemples, en deux : ceux qu'on verrait dans la leçon diviser pour régner car ils sont là spécifiquement pour illustrer le paradigme, et ceux que l'on verrait au cours de l'année, et s'appuyant sur le fait qu'on ait déjà vu la méthode diviser pour régner. Cette distinction vient du fait que dans un vrai cours, on ferait comme ça, et les problèmes de la dernière partie serait insérer dans autre chose, et on souligne alors le fait que on ne ferait pas simplement une liste d'algo, mais que pour que le paradigme soit assimiler, il faut mieux le présenter et y revenir régulièrement, plutôt que d'introduire plusieurs fois beaucoup de concepts. (et on met ainsi de la structure sans faire simplement une liste)*

1 Principe

I Motivation

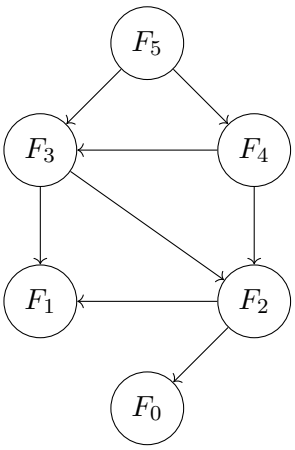
Algorithme 14.1: Implémentation naïve de la suite de Fibonacci

Algorithme 14.1 : $fibo(n)$

si $n = 0$ ou $n = 1$ **alors**
  retourner n
sinon
  retourner $fibo(n-1) + fibo(n-2)$

Propriété 14.2: Complexité en $O(2^n)$

Graphe des dépendances des sous problèmes pour $n = 5$



Les sous-problèmes se chevauchent : la méthode diviser pour régner est inefficace.

En programmation dynamique, on stocke les valeurs des sous-problèmes pour éviter de recalculer.

Algorithme 14.3: Fibonacci avec stockage.

Algorithme 14.2 : $Fibo(n)$

$F \leftarrow [0, 1]$
pour i allant de 2 à n **faire**
  Ajouter $F[i-1] + F[i-2]$ à F
retourner $F[n]$

II Définition

Définition 14.4: La programmation dynamique consiste à résoudre un problème en le décomposant en sous-problèmes, puis à résoudre les sous-problèmes, des plus petits aux plus grands en stockant les résultats intermédiaires.

Principe 14.5:

1. Complexifier le problème en créant plein de sous-problèmes (dont un sera celui que l'on veut résoudre)
2. Trouver une relation entre les sous-problèmes
3. Résoudre les sous-problèmes en partant du plus petit, en utilisant la relation :
   — Soit impérativement en ayant un tableau des sous-problème qui stockera leur résultat, et en le remplissant avec une (ou des) boucle (méthode ascendante)
   — Soit récursivement en mémoïsant. (méthode descendante)

Remarque 14.6: L'algorithme 14.3 utilise la méthode ascendante. On aurait pu utiliser uniquement deux variables et écraser les résultats intermédiaires.

Remarque 14.7: Dans le paradigme diviser pour régner, les sous pb sont indépendants. On peut donc se passer de la mémoïsation.

Remarque 14.8: Pour obtenir, en plus de la valeur de la solution optimale, quelle est cette solution, on peut mémoriser quel(s) sous-problème(s) on a utilisé pour l'obtenir.

Développement : Illustration du paradigme sur le problème du chemin dans la pyramide.

2 Algorithmes illustrant le principe

I Rendu de monnaie

Instance : n pièces $p_1, ...p_n$, $S \in \mathbb{N}$

Pb : trouver un n-uplet $T = (x_1, \ldots, x_n) \in \mathbb{N}^n$ te que $\sum_{i=1}^{n} x_i p_i = S$ et qui minimise $\sum_{i=1}^{n} x_i$. (i.e. trouver le nombre de pièces minimum pour rendre la monnaie)

Algorithme 14.9: Approche gloutonne :

1. Ajouter la pièce p_i de plus grande valeur $\leq S$
2. Recommencer avec $S - p_i$

Cet algorithme est-il optimal ?

Exercice 14.10: Avec les pièces $(4, 3, 1)$, trouver une somme S pour laquelle le glouton n'est pas optimal ($S = 6$ convient)

Algorithme 14.11 (Approche par programmation dynamique du rendu de monnaie):

1. On considère les sous-problèmes $R(s)$ pour $s \in [\![0, S]\!]$
2. Pour trouver la relation de récurrence, on regarde la dernière pièce rendue p_i. On a alors

$$R(S) = \begin{cases} +\infty & \text{si } S < 0 \\ 0 & \text{si } S = 0 \\ \min\limits_{i \in \{1,\ldots,n\}} (R(s - p_i) + 1) & \text{sinon} \end{cases}$$

3. Résolution des sous problèmes (méthode descendante)

Algorithme 14.3 : $rendu(P, S, m)$

Entrées : P est un tableau tel que $P[i] = p_i$
m tableau de mémoïsation initialisé à 0

si $S = 0$ **alors**
  retourner 0
si $m[S] > 0$ **alors**
  retourner $m[s]$
$n \leftarrow +\infty$
pour $p \in P$ **faire**
  $n \leftarrow \min(n, 1 + rendu(P, S - p, m))$
$m[S] \leftarrow n$
retournern

Complexité : $O(n \times S)$

Exercice 14.12: Écrire la méthode ascendante de résolution (i.e. trouver l'ordre de remplissage du tableau)

II Sac à dos

Instance : n objets de poids $\{w_1, \ldots, w_n\} \in \mathbb{N}^n$, et de valeur $\{v_1, \ldots, v_n\} \in \mathbb{N}^n$. Une capacité $W \in \mathbb{N}$.
Problème : Trouver $I \subset \{1, \ldots, n\}$ tel que $\sum_{i \in I} w_i \leq W$ et qui maximise $\sum_{i \in I} v_i$

Exercice 14.13: Proposer des algorithmes gloutons pour résoudre le pb du sac à dos. Sont-ils optimaux ?

Algorithme 14.14 (Approche par programmation dynamique du problème du sac à dos):

1. On considère les sous-problèmes $SD(i, w)$ réduit aux i premiers objets avec une capacité w. La solution qui nous intéresse est celle de $SD(n, W)$
2. $$T(i,w) = \begin{cases} 0 \text{ si } i = 0 \text{ ou } w = 0 \\ \max\Big(\underbrace{T(i-1,w)}_{\substack{\text{si l'optimal prend} \\ \text{pas l'objet } i}} , \underbrace{T(i-1, w-w_i) + v_i}_{\substack{\text{si l'optimal prend} \\ \text{l'objet } i}} \Big) \text{ sinon} \end{cases}$$
3. On remplit à i et w croissant.

Remarque 14.15: Le problème de décision associé au problème de sac à dos est NP-complet. Ici, l'algorithme résout le problème d'optimisation en $O(S \times n)$, or la taille de l'instance d'entrée est en $\log |S| + n$, donc notre algorithme n'est pas polynomial en la taille de l'instance.

3 Autres applications

I L'algorithme de Floyd-Warshall

Instance : Un graphe orienté $G = (S, A)$ sans circuit absorbant et une fonction de poids $w : A \to \mathbb{Z}$
Problème : Déterminer les valeurs des plus courts chemins entre toutes les paires de sommets de G

Algorithme 14.16 (Résolution du problèmes des plus courts chemins en découpant selon les sommets que l'on utilise):

1. On numérote les sommets de G : $S = \{1, \dots, n\}$. On s'intéresse aux sous problèmes $FW(i, j, k)$ qui correspond au plus court chemin de i à j empruntant des sommets intermédiaires dans $\{1, \dots, k\}$. Les problèmes qui nous intéressent sont $FW(i, j, n)$ pour $i \neq j$.
2. $$FW(i,j,k) = \begin{cases} w(i,j) \text{ si } k = 0 \\ \min\Big(\underbrace{F(i,j,k-1)}_{\substack{\text{si le plus court} \\ \text{chemin de } i \text{ à } j \\ \text{n'utilise pas } k}} , \underbrace{F(i,k,k-1)}_{\substack{\text{si le plus court} \\ \text{chemin de } i \text{ à } j \\ \text{utilise } k}} + F(k,j,k-1) \Big) \text{ sinon} \end{cases}$$
3. Résolution par la méthode ascendante

Algorithme 14.4 : $Floyd - Warshall(G)$

```
W ← matrice d'adjacence de G
pour k allant de 1 à n faire
    pour i allant de 1 à n faire
        pour j allant de 1 à n faire
            W[i,j] ← min(W[i,j], W[i,k] + W[k,j])
```

Complexité : $O(|S|^3)$

Remarque 14.17: On écrase la matrice au fur et à mesure car l'étape k ne dépend que de l'étape $k - 1$

Remarque 14.18: On aurait pu découper les sous problèmes différemment. Par exemple, on peut découper au chemin de taille au plus k. On retombe alors sur un algorithme de routage en réseau.

Développement : Terminaison et discussion autour de l'algorithme de Bellman-Ford

II Distance de Levenshtein (d'édition)

Définition 14.19: La **distance de Levenshtein** correspond au nombre minimum d'opérations (suppression, modification ou ajout d'une lettre) pour transformer un chaîne de caractère en l'autre.

Plus formellement, pour $a \in \Sigma$, $i \in \mathbb{N}$ on définit

- $\star$ $ins_{a,i} : \Sigma^* \to \Sigma^*$: insertion de la lettre a à la position i
- $\star$ $sub_{a,i} : \Sigma^* \to \Sigma^*$: modification de la lettre à la position i en a
- $\star$ $sup_i : \Sigma^* \to \Sigma^*$: suppression de la lettre à la position i

et alors $lev(w_1, w_2) = \min \left\{ k \in \mathbb{N} \quad / \quad \exists f_1, \dots, f_k \in \{ins_{a,i}, sub_{a,i}, sup_i / a \in \Sigma, i \in \mathbb{N}\} : w_2 = f_k \circ \cdots \circ f_1(w_1) \right\}$

Exercice 14.20: C'est une distance.

Commentaire 14.2 *Ici on ne donne que l'idée parce que on a pas la place et parce que l'élève peut avoir le recul pour trouver lui-même*

Algorithme 14.21: Idée de l'approche par programmation dynamique pour la distance de levenshtein

1. Considérer la distance d'édition entre tous les préfixes de w_1 et w_2
2. Considérer la modification sur la dernière lettre (rien, insertion, suppression ou modification)
3. Résoudre à préfixe croissant

Leçon 15

Exemples d'algorithmes d'apprentissage supervisés et non supervisés

Auteur·e·s: Emile Martinez
Références :

1 Introduction

I Définition

Définition 15.1: On dit que un algorithme apprend via un entraînement pour un ensemble de tâches et une mesure de performance si sa performance sur les tâches mesuré par la mesure de performance s'améliore après l'entraînement.

Définition 15.2: On considère deux familles de l'apprentissage machine :

1. l'apprentissage supervisé : on dispose d'espaces X d'entrée et Y de sortie, et d'un ensemble E d'exemples $(x_i, y_i) \in X \times Y$, représentant une fonction $f : X \to Y$. Le but étant alors de construire une fonction $\hat{h} : X \to Y$ approximant f.
2. l'apprentissage non supervisé : on dispose seulement de l'espace X dont on veut alors découvrir les structures sous-jacente

Exemple 15.3: Sur un ensemble de données sur des animaux, on peut :

1. Savoir lesquels sont des chiens, des chats, etc... (apprentissage supervisé)
2. Savoir quels animaux sont de la même espèce (apprentissage non supervisé)

Remarque 15.4: Il existe d'autres familles, comme l'apprentissage par renforcement qui consiste à maximiser un critère d'utilité par expériences successives.

II Évaluation d'un algorithme d'apprentissage

On cherche à évaluer la performance d'un algorithme d'apprentissage.

Définition 15.5: On découpe nos données d'entrées E en deux :

— Les données d'apprentissage, servant à entraîner l'algorithme

— Les données de validation, servant à imiter la prédiction, mais en comparant le résultat obtenu à celui attendu.

Remarque 15.6: Si l'on a trop peu de données, on peut également faire une validation croisée, consistant utiliser alternativement des données comme apprentissage et comme validation

Remarque 15.7: Cette phase est également utile pour calibrer les paramêtres de l'algorithme.

2 Apprentissage supervisé

On reprend les notations de la définition 15.2

Commentaire 15.1 *Dans un vrai cours, on réécrirait les définitions pour plus de clarté*

Définition 15.8:
Si Y est un ensemble fini de classes, on parle de problème de classification.
Si $Y = \mathbb{R}$, on parle de problèmes de régression.

Exemple 15.9: Sur un ensemble de données sur des animaux, on peut avoir :
— $Y = \{'chien', 'chat'\}$ (classification)
— $Y = \mathbb{R}$ représentant le poids de l'animal (régression)

Concentrons nous sur le problème de classification. On cherche à inférer de E, la classe de $\alpha \in X$. Notons C la partition de X représentant les différentes classes.

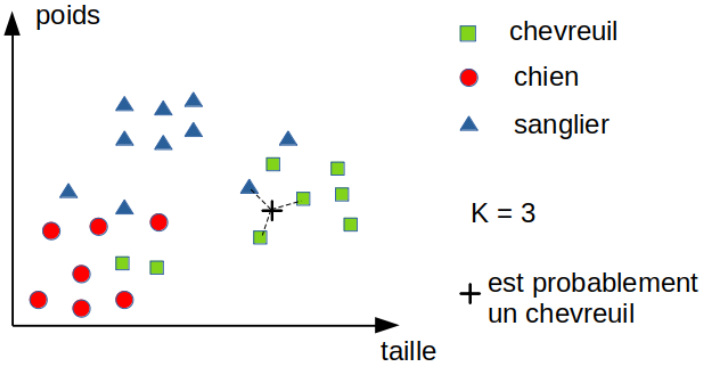
I K plus proches voisins

On s'intéresse ici au cas où $X = \mathbb{R}^d$

Algorithme 15.10: K plus proches voisins de α
1. Déterminer les K plus proches voisins de α dans E
2. Choisir la classe majoritaire parmi ces voisins.

Développement : Présentation de l'algorithme illustrant différents aspects de l'apprentissage machine.

Exemple 15.11:



Remarque 15.12: Ici pour choisir k, on utilise la méthode de mesure de performance du II en essayant plusieurs paramètres.

Exercice 15.13: Appliquer l'algorithme sur le jeu de données MINST.

Remarque 15.14: Pour un problème de régression, on ne choisirait pas la classe majoritaire à l'étape 2, mais on agrégerait les données (par exemple en faisant une moyenne).

Implémentation 15.15:

Solution initiale : Stocker nos K valeurs en cours dans une file de priorité (implémentée par un tas max) et parcourir les n points en mettant à jour la file de priorité

Complexité : $O(n \log k)$

Solution diviser pour régner : Faire un pré traitement où l'on stockera nos valeurs dans un arbre binaire de recherche (arbre d-dimensionnel), où l'on partitionnera récursivement les données alternativement sur chaque dimension. La recherche se fait alors en ne cherchant que d'un côté si le deuxième n'est pas nécessaire.

Complexité :

Prétraitement :$O(n \log n)$

Recherche des k voisins : $O(k \log n)$ en moyenne et $O(nk)$ dans le pire des cas.

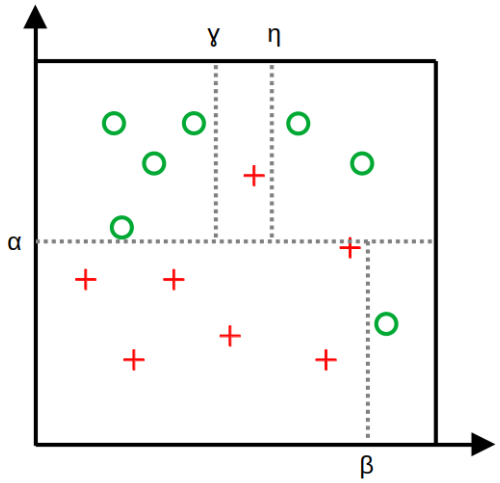
Développement : Présentation de la structure d'arbre d-dimensionnel.

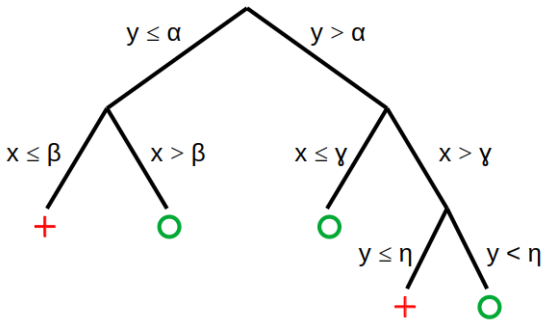
Remarque 15.16: On parle souvent d'arbre k-dimensionnel, mais on prend ici d pour éviter la confusion avec K.

II Arbre de décision

On s'intéresse ici au cas $X = \mathbb{R}^d$ (ou $\{O, 1\}^d$)

Idée 15.17: Partitionner récursivement X grâce à un arbre de décision où chaque feuille a une classe.





Définition 15.18: Définissons l'entropie d'une partie S de E : $H(S) = \sum_{c \in C} \frac{|S \cap c|}{|S|} \times \log\left(\frac{|S \cap c|}{|S|}\right)$

Remarque 15.19: Si tous les éléments ont la même classe, $H(S) = 0$

Définition 15.20: Le gain d'une partition S_1, S_2 de S est :

$$H(S) - \left(\frac{|S_1|}{|S|} \times H(S_1) + \frac{|S_2|}{|S|} \times H(S_2) \right)$$

Algorithme 15.21: On construit récursivement notre arbre de décision sur notre ensemble S de données restantes :

- Si S est vide : Choisir la classe la plus représentée du nœud parent
- Si toutes les données de S ont la même classe : en faire une feuille avec cette classe.
- Sinon, on choisit la coordonnée i et la valeur m tel que la partition

$$S_1 = \{(x_1, \ldots, x_n) \in S / x_i \leq m\} \text{ et } S_2 = \{(x_1, \ldots, x_n) \in S / x_i > m\}$$

maximise le gain

Remarque 15.22: On atteint très vite du sur apprentissage. Pour éviter cela, on peut élaguer le bas de l'arbre (algorithme de Cart)

Exercice 15.23: Application à détecter la langue d'une page wikipedia à partir de la matrice de fréquence de facteurs de 2 lettres.

Commentaire 15.2 *Ici pour ce TD, on peut mentionner à l'oral le fait qu'on peut utiliser ce truc pour de vrais, et surtout que on peut éventuellement faire réfléchir les élèves à comment modéliser le problème. Fréquence des mots, KNN avec distance de levenstein, fréquence des facteurs de 1, 2, 5 lettres ? Ce qui rend tout ca très intéressants selon moi. (suivant le nombre de pages en exemples, on peut prendre les facteurs d'un certain nombres de lettres (on aura un tableau de taille $27^{\text{taille des facteurs}}$) et ensuite faire du KNN, de l'arbre de décision, etc...) (Avec 4000 pages par langues pour 13 langues, et les facteurs de 2 mots, on classifie très bien).*

Remarque 15.24: Dans le cas d'une régression, on peut prendre comme mesure d'impureté la variance.

III Représentation de la qualité des classes

Quand on mesure la performance de notre algorithme on peut chercher à représenter la qualité de nos prédictions (pour classification).

Définition 15.25 (matrice de confusion): On associe Y à $\{1, \ldots, n\}$. La matrice de confusion est alors la matrice carrée M de taille n tel que $M_{i,j}$ est le nombre d'éléments de la classe i qui ont été classés à j

Propriété 15.26: Plus notre algorithme est correcte, plus la diagonale est dominante.

3 Apprentissage non supervisé

Remarque 15.27: Il existe plusieurs types d'apprentissage non supervisé. On peut par exemple penser à la réduction de dimension : On a $X \subset \mathbb{R}^n$ et on cherche $f : X \to \mathbb{R}^m$ avec $n < m$ et tel que $f(x)$ et $f(y)$ sont proches si x et y le sont aussi.

On se concentrera sur ce qu'on appellera regroupement, (clustering en anglais, ou classification non supervisée) qui consiste à trouver $f : X \to \{1, ..., k\}$ essayant de regrouper les éléments les plus proches.

Exemple 15.28: On a un manuscrit dans un alphabet inconnu, et on cherche à savoir quels lettres sont les mêmes.

Exercice 15.29: Dans l'exemple au dessus, quel X prendre ?

I Classification hiérarchique ascendante

Idée 15.30: Chacun est seul dans sa classe au début, et tant qu'on a k classes, on fusionne les classes les plus proches.

Remarque 15.31: C'est un algorithme glouton

Remarque 15.32: Pour définir la distance entre deux classes, on prendre prendre :

- $d(S_1, S_2) = \min_{x \in S_1, y \in S_2} (d(x, y))$
- $d(S_1, S_2) = \min_{x \in S_1, y \in S_2} (d(x, y))$
- $d(S_1, S_2) = \frac{1}{|S_1| \times |S_2|} \sum_{x \in S_1, y \in S_2} d(x, y)$

Exercice 15.33: Activité : Représenter l'exécution de l'algorithme sur un dendrogramme

II ALgorithme des k-moyennes

Idée 15.34: On cherche à trouver $S_1, \ldots, S_k$ une partition de X et $z_1, \ldots, z_k \in \mathbb{R}^d$ minimisant $\sum_{i=1}^{k} \sum_{x \in S_i} d(x, z_i)$

Propriété 15.35: Cet algorithme termine

Démonstration. La cible diminue à chaque étape, et ne peut prendre qu'un nombre fini de valeurs. □

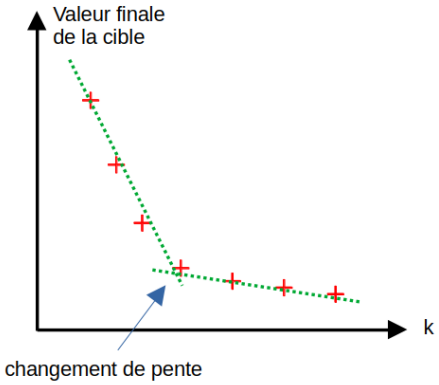
Remarque 15.36: On ne trouve pas toujours le minimum, on tombe souvent dans un minimum local. On relance alors plusieurs fois l'algorithme (d'où l'aléatoire au début).

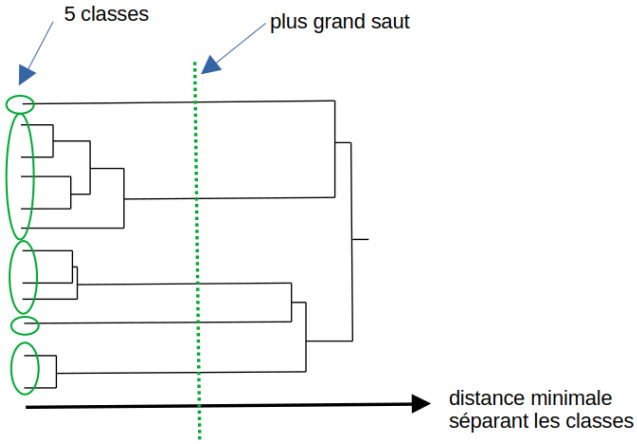
Algorithme 15.1 : K-mean

Assigner à chaque valeur une classe aléatoire
répéter
  pour $x \in X$ **faire**
    assigner à x la classe de $\arg\min_{i\in\{1,\dots,k\}} d(x, z_i)$
  pour i allant de 1 à k **faire**
    $z_i \leftarrow \frac{1}{|S_i|} \sum_{x\in S_i} x$
jusqu'à stabilisation;

Exercice 15.37: TP sur la réduction de palette d'une image

Remarque 15.38: Comment choisir k ? Pour la classification hierarchique ascendante, on s'arrête au plus grand saut sur le dendrogramme, pour les k-moyennes, on affiche la cible finale en fonction de k , et on choisit k au changement de pente.





Commentaire 15.3 *Si on a la place, on peut faire de cette remarque une sous-partie supplémentaire. Si on l'a pas, on peut enlever les dessins, et simplement le mentionner pour les k-moyennes, ou même l'enlever au pire.*

Leçon 16

Exemples d'algorithmes pour l'étude des jeux

Auteur·e·s: Emile Martinez
Références :

La théorie des jeux s'intéresse aux interactions entre des individus (joueurs) qui effectuent des choix selon les règles d'un jeu.

1 Jeux d'accessibilité à deux joueurs

I Définition

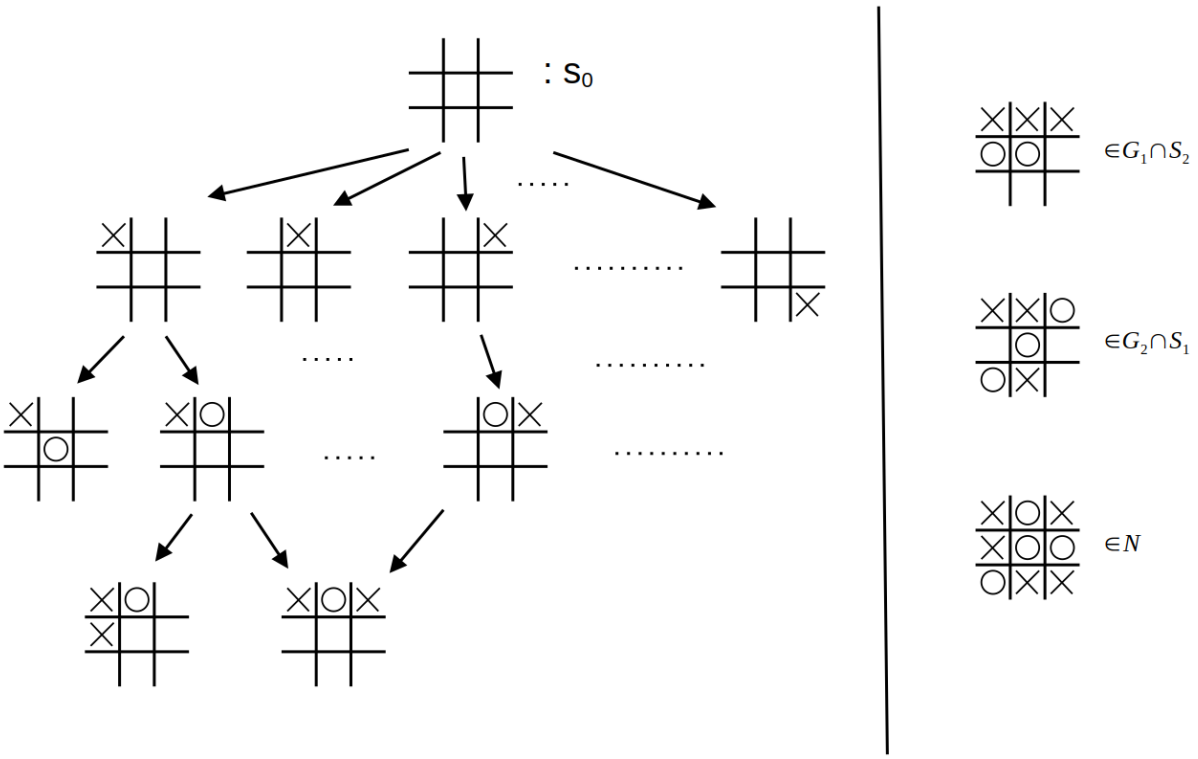
Notation : Pour $G = (S, A)$, on note $Fin(G) = \{v \in S / deg^+(v) = 0\}$

Définition 16.1: Un jeu à deux joueurs est :
- une arène : un graphe orienté biparti $G = (S_1 \sqcup S_2, A)$
- un sommet de départ s_0
- une partition de $Fin(G) = G_1 \sqcup G_2 \sqcup N$

Commentaire 16.1 *On peut mentionner ici qu'on peut se restreindre au graphes bipartie, car si quand on joue, c'est à nouveau à nous, on considère les deux coups à jouer d'affilée comme un seul. Mais y a des jeux où on peut rejouer, il faut alors travailler un peu pour modéliser.*

Idée 16.2: Les sommets de S_1 sont ceux où J1 joue. Un arc $x \to y$ représente un coup possible pour le joueur qui contrôle l'état x, qui déplace alors le jeu dans l'état y. G_i sont les états gagnants pour Ji, N ceux d'une partie nulle.

Exemple 16.3: Représentation d'un échantillon de la modélisation du Tic-Tac-Toe



Définition 16.4 (Partie): Une partie d'un jeu (G, s_0, G_1, G_2, N) est un chemin de s_0 à un sommet $s_f \in Fin(G)$.

Définition 16.5: On appelle stratégie pour le joueur $i \in \{1, 2\}$ toute fonction $\varphi : V_i \to V$ tel que $\forall u \in V_i \backslash Fin(G)$, $(u, \varphi(u)) \in A$.

Commentaire 16.2 *Ici on définit directement une stratégie sans mémoire, le programme se limitant à cela, et les jeux que nous considérerons ne nécessitant pas de stratégie avec mémoire.*

Définition 16.6 (Stratégie gagnante): φ est une stratégie gagnante pour le joueur i si pour toute partie $P = s_0, \ldots, s_f$,

$$\Big(\forall j \in [\![O, f-1]\!], s_j \in V_i \implies s_{j+1} = \varphi(s_j)\Big) \implies s_f \in G_i$$

s est une position gagnante s'il existe une stratégie gagnante depuis s.

Idée 16.7: φ est une stratégie gagnante si quand les coups de Ji sont ceux de φ, alors Ji gagne indépendamment de ce que joue l'autre joueur.

II Attracteurs

On se place du point de vue du joueur 1, mais la situation est symétrique avec le joueur 2.

Définition 16.8 (Attracteur): Pour une arène (G, s_0), et $F \subset V$ on note $Attr_i(F)$, l'ensemble des

sommets depuis lesquels le joueur J1 a une stratégie pour arriver dans F en au plus i étapes. On note $Attr(F) = \bigcup_{i=0}^{+\infty} Attr_i(F)$

Propriété 16.9: Pour un jeu (G, s_0, G_1, G_2, N), $Attr(G_1)$ est l'ensemble des position gagnante de J1.

Propriété 16.10:

- $Attr_0(F) = F$
- $Attr_{i+1}(F) = Attr_i(F) \cup \{u \in V_1 \,/\, \mathcal{N}^+(u) \cap Attr_i(F) \neq \emptyset\} \cup \{u \in V_2 \,/\, \mathcal{N}^+(u) \subset Attr_i(F)\}$

Propriété 16.11: Si G est fini alors $(Attr_i(F))$ est croissante bornée, donc stationnaire. Sa limite est alors $Attr(F)$.

Une stratégie gagnante depuis $Attr(F)$ est :

$$\begin{array}{rcl} \varphi : V_1 & \to & V \\ v & \mapsto & \begin{cases} \omega \in \mathcal{N}^+(v) \cap Attr_i(F) & \text{si } v \in Attr_{i+1}(F) \backslash Attr_i(F) \\ \omega \in \mathcal{N}^+(v) & \text{si } v \notin Attr(F) \end{cases} \end{array}$$

Développement : Stratégies gagnantes pour le jeu de Nim à 1 puis plusieurs tas.

2 Jeux Min-Max

I Algorithme Min-Max

On considère ici des jeux à deux joueurs, en reprenant la définition 16.1 en remplaçant la partition de $Fin(G)$ par une fonction de coût $c : Fin(G) \to \mathbb{Z}$.

Le joueur 1 (appelé Max ici) essaye alors de maximiser par ses coups la valeur finale, et le joueur 2 (ici Min) essaye de la minimiser.

Remarque 16.12: La partie précédente en est un cas particulier avec $c(G_1) = 1$, $c(N) = 0$ et $C(G_2) = 1$

Définition 16.13: Une stratégie optimale pour le joueur Max (resp. Min) est une stratégie maximisant (resp. minimisant) le coût.

Algorithme 16.14: On fait une recherche exhaustive de tous les coups, en prenant à chaque fois celui ayant le résultat maximum (resp.minimimum) quand Max (resp. Min) joue. Cela détermine une stratégie optimale (pour les deux joueurs).

Remarque 16.15: En pratique, c'est souvent infaisable tant le graphe est gros.

Exemple 16.16: Les echecs, avec +1000 victoire blanche, -1000 victoire noire et 0 nulle, le graphe a plus de 10^44 sommets.

Définition 16.17: Une heuristique $h : V \to \mathbb{Z}$ est une estimation de à quoi mènerait dans le meilleur cas cette position

Commentaire 16.3 *Ici on a une définition formelle avec l'intuition de son interprétation (et donc de son usage). (une heuristique, c'est simplement une fonction $V \to \mathbb{Z}$)*

Exemple 16.18: La fonction qui aux échecs donne la somme de la valeur des pièces

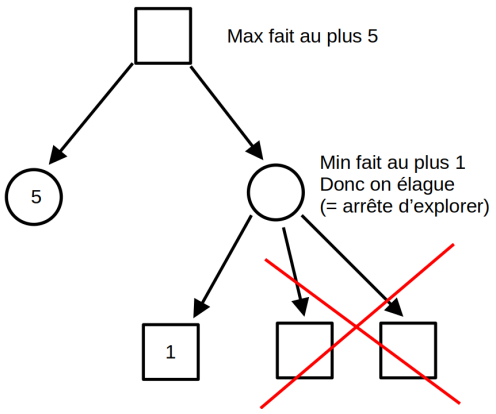
Idée 16.19: On fait l'exploration exhaustive en renvoyant l'heuristique quand on a fait suffisament d'étapes.

Algorithme 16.1 : $MinMax(j, prof, u)$

si $u \in Fin(G)$ ou $prof = 0$ **alors**
  retourner $h(u)$
si $i == 1$ **alors**
  $f \leftarrow \max$
sinon
  $f \leftarrow \min$
retourner $f\left(\left\{MinMax(3-j, prof-1, v) \ / \ v \in \mathcal{N}^+(u)\right\}\right)$

II Élagage α-β

Idée 16.20: Il n'est pas nécessaire d'explorer tout l'arbre : si je suis Min et que Max au coup d'avant peut faire 5 avec un autre coup, dès que je vois que je peux faire 1, je peux arrêter d'explorer, car je sais que Max ne fera pas ce coup.



Algorithme 16.2 : $Alphabeta(j, \alpha, \beta, u)$

si $u \in Fin(G)$ **alors**
  retourner $c(u)$
si $joueur = 1$ **alors**
  $res \leftarrow -\infty$
  pour v voisin de u **faire**
    $e = Alphabeta(2, \max(res, \alpha), \beta, v)$
    si $e > \beta$ **alors**
      retourner e `// Élagage`
    sinon
      $res \leftarrow \max(e, res)$
  retourner res
sinon
  `// Symétrique, à faire en exercice`

Idée 16.21: α (resp. β) est la valeur maximale (resp. minimale) que peut faire Max (resp. Min) parmi ce que l'on a explorer pour l'instant. (On appelle donc $alphabeta(1, -\infty, +\infty, s_0)$)

Remarque 16.22: Cet algorithme est exact. On peut, comme pour Min-Max ajouter une profondeur et une heuristique.

Exercice 16.23: Comparaison des temps d'exécution de Min-Max et de Alpha-Beta pour le Tic-Tac-Toe (exploration complète sans heuristique).

3 Les Jeux à un joueur

I Graphe d'état

Définition 16.24: Un graphe d'état est la donnée d'un graphe orienté $G = (S, A)$ pondéré par $c : A \to \mathbb{N}$, d'un état initial s_0 et d'un ensemble d'états finaux $F \subset S$

Remarque 16.25: S représente les configurations d'un jeu à un joueur, A les transitions d'une configuration à une autre en un coup, c le coût de cette transition, et F les configurations gagnantes.

Objectif : Trouver un chemin dans ce graphe entre l'état initial et l'un des états finaux de coût total minimal.

Exemple 16.26: Dans le jeu du taquin, les états correspondent aux dispositions possibles du plateau. L'état final est le plateau remis dans l'ordre. Chaque case a un degré sortant inférieur ou égal à 4 qui correspond aux déplacements possibles de la case vide (vers le haut, le bas, à gauche ou à droite). Tous les déplacements ont un coût unitaire.

Remarque 16.27: le graphe d'état est en général trop gros pour être stocké entièrement en mémoire. Il est donc nécessaire de mettre en place des stratégies ou des heuristiques pour orienter la recherche du chemin

II L'algorithme A*

Principe 16.28: L'algorithme A* est une variante de l'algorithme de Dijkstra pour calculer un plus court chemin entre un sommet initial s_0 et un sommet final s_f. On visite les sommets par estimation de leur proximité à s_f grâce à une fonction f définie par :

$f(s) = d(s) + h(s)$ où $d(s)$ est le coût d'un plus court chemin entre s_0 et s et $h(s)$ i une estimation (heuristique) du coût entre s et s_f

Exemple 16.29: dans le cas du taquin, on peut penser aux heuristiques suivantes :

- nombre de chiffres mal placés
- somme des distances de Manhattan des cases à leur position finale

Algorithme 16.3 : Algorithme A*

Entrées : W la matrice de poids du graphe ; h le tableau pour l'heuristique ; s_0 et s_f les sommets initiaux et finaux
Sorties : la distance d'un plus court chemin de s_0 à s_f

```
D ← tableau initialisé à ∞
D[s_0] ← 0
P ← file de priorité vide
Ajouter (s_0, h[s_0]) à P
tant que P non vide faire
    (s, _) ← extraire(P)
    si s = s_f alors
        retourner D[s]
    pour s' successeur de s faire
        c ← D[s] + W[s, s']
        si c < D[s'] alors
            D[s'] ← c
            Ajouter (s', c + h[s']) à P
retourner NonAccessible
```

Remarque 16.30: Si $h = 0$, on retrouve Dijkstra

Définition 16.31: Une heuristique est dite admissible si $\forall u \in S, h(u) \leq d(u)$

Théorème 16.32 (Correction): Si h est admissible, alors A* renvoie la distance d'un court chemin de s_0 à s_f.

Définition 16.33: Une heuristique est dite monotone si $\forall (u, v) \in A, h(u) \leq h(v) + w(u, v)$

Remarque 16.34: Dans un graphe avec les distances euclidiennes entre nœuds, la distance à vol d'oiseau est une heuristique monotone.

Propriété 16.35: Si h est monotone et $h(s_f) = 0$, alors A* est correct (h est admissible) et extrait chaque noeud au plus une fois.

Développement : Démonstration du théorème et de la propriété précédente.

Application 16.36: Calcul des itinéraires par un GPS

Leçon 17

Algorithmes d'ordonnancement de tâches et de gestion de ressources

Auteur·e·s: Emile Martinez
Références :

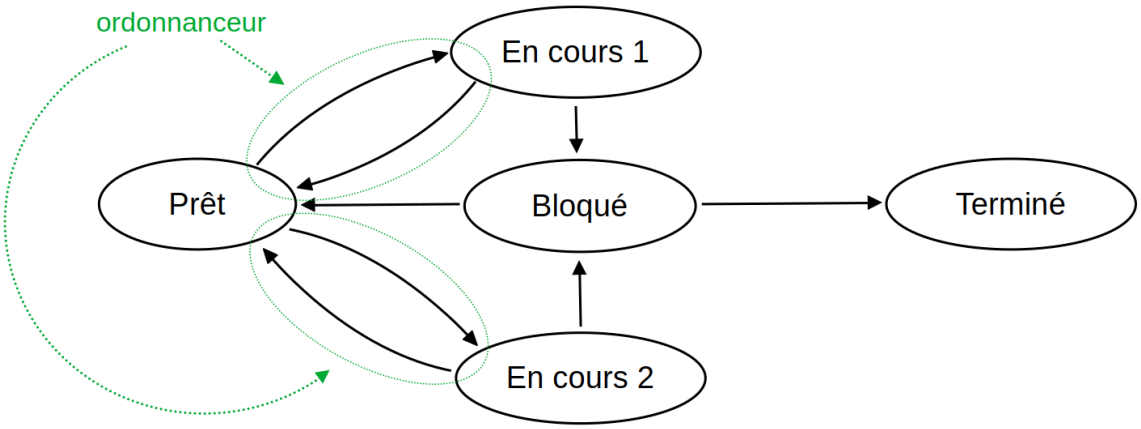
Métaphore filée : Ordonnancement des tâches dans une cuisine.

1 Motivation : le système d'exploitation

Remarque 17.1: Lorsque vous utilisez votre PC, vous éxécutez des dizaines de programmes "en même temps" (lecture de mail, taper au clavier, écouter de la musique...). Pourtant, votre PC a un nombre limité de processeurs.

Définition 17.2 (Execution concurrente et ordonnanceur): Le système d'exploitation peut interrompre un processus en cours pour exécuter du code qui lui est propre. Il peut alors, à intervalles réguliers, décider à quelle tâche en cours il rend la main.

Le rôle de l'ordonnanceur est de choisir le prochain processus à exécuter parmi une liste de processus candidats.



Cycle de vie d'un processus

Commentaire 17.1 *A faire. Mais à comparer avec le truc de Malory, parfois mieux que notre truc à nous.*

Leçon 18

Gestion et coordination de multiples fils d'exécution

Auteur·e·s: Emile Martinez
Références :

1 Gestion par la machine (Terminale)

I Motivation

Lorsque vous utilisez votre PC, vous éxécutez des dizaines de programmes ≪en même temps≫ (lecture de courriel, taper au clavier, écouter de la musique...). Pourtant, votre PC a un nombre limité de processeurs. Comment fonctionne cette illusion ?

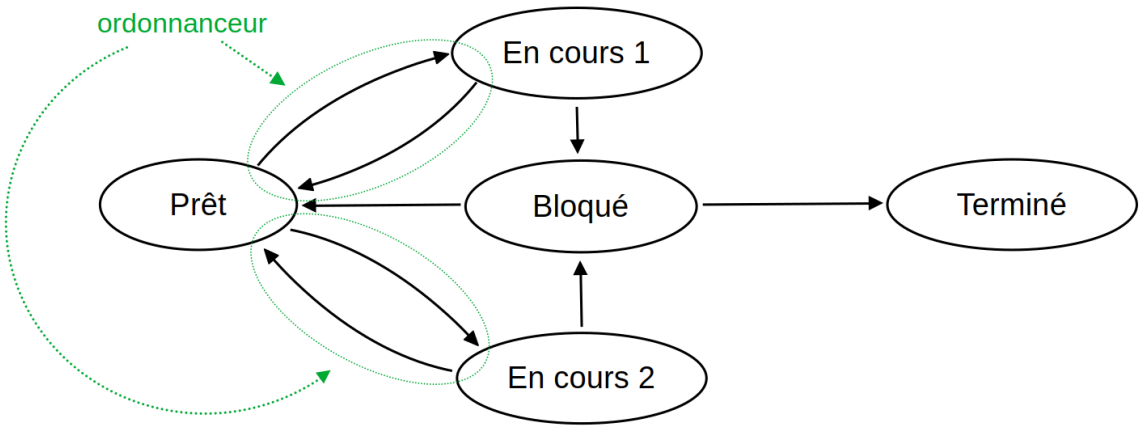
II Exécution concurrente

Définition 18.1: Un processus est un programme en cours d'exécution sur un ordinateur. Il dispose d'une zone mémoire en RAM. Le système d'exécution identifie les processus grâce à un numéro unique appelé PID.

Définition 18.2 (Exécution concurrente): Deux processus s'exécutent en concurrence si les intervalles entre leur commencement et leur fin respective sont non disjoints.

Principe 18.3 (Fonctionnement de l'exécution concurrente): Le système d'exploitation peut interrompre un processus en cours pour exécuter du code qui lui est propre. Il peut alors, à intervalles réguliers, décider à quelle tâche en cours il rend la main.

Le rôle de l'ordonnanceur est de choisir le prochain processus à exécuter parmi une liste de processus candidats.



Cycle de vie d'un processus

III L'ordonnanceur

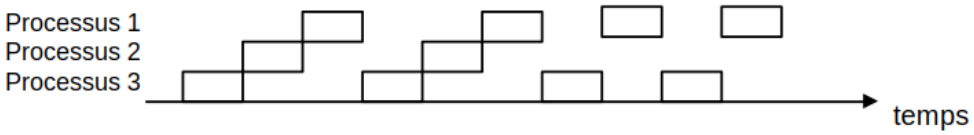
On peut vouloir des garanties sur la manière dont l'ordonnanceur choisit ses tâches.

Définition 18.4: On dit qu'il y a absence de famines (ou vivacité) si aucun processus n'attend indéfiniment.

Définition 18.5: On dit qu'il y a équité si aucun processus n'est favorisé.

Algorithme 18.6 (Algorithme du tourniquet): L'ordonnanceur définit un intervalle de temps τ.
Il place les processus en attente dans une file selon leur ordre d'arrivée (PAPS).
Tant que la file est non vide, il en sort le premier et l'exécute durant tau. Si besoin, il le ré-insert en queue de file.

Exemple 18.7: Exemple d'exécution sur 3 processus et un seul processeur



Propriété 18.8: L'algorithme du tourniquet garantie l'équité et l'absence de famine.

Remarque 18.9: Certains processus étant plus importants que d'autres, on peut adapter le temps tau à chaque processus, en fonction de leur priorité.

IV Commande unix de gestion de processus

`$ ps`	permet d'afficher les processus en cours, et leur taux d'occupation du processeur et de la mémoire.
`$kill 6555`	envoie un signal de terminaison au processus de PID 6555. (Pour une application graphique, c'est ce que fait la croix rouge)
`$top`	affiche la liste des processus prêts et en cours.

2 Gestion par l'utilisateur (prépa)

I Motivation

Définition 18.10: Un fil d'exécution (thread en anglais) est un sous processus qui partage la mémoire avec les autres fils du processus.

Implémentation 18.11: En C, un programme peut lancer d'autres fils d'exécution, de type `pthread_t` et que l'on lance avec la fonction `pthread_create`

Idée 18.12: On peut alors gérer des choses en parallèle

Exemple 18.13: Somme parallélisée d'un tableau T de taille 2m.

Fil 1 :

```
1 for(int i = 0; i < m; i++){
2   int s = res + T[2*i];
3   res = s;  }
```

Fil 2 :

```
1 for(int i = 0; i < m; i++){
2   int s = res + T[2*i + 1];
3   res = s;  }
```

On peut donc aller deux fois plus vite.

Question : Que se passe-t-il sur l'exécution F1 :1 → F2 :2 → F1 :3 → F2 :3 ?

II Les verrous

Définition 18.14: Le problème de l'exclusion mutuelle consiste à garantir que deux fils d'exécution n'essaieront pas d'exécuter simultanément un morceau de code prédéfini.

Exemple 18.15: Les lignes 2 et 3 pour les fils 1 et 2 de l'exemple 18.13.

Définition 18.16: Un verrou (ou mutex) est une structure de données permettant deux opérations :

- `prendre()` : appel bloquant qui demande l'accès au verrou
- `rendre()` : appel qui libère le verrou

Propriété 18.17: Une implémentation efficace des verrous devrait garantir l'exclusion mutuelle, l'absence de famine et l'équité.

Algorithme 18.18: L'algorithme de Peterson propose une implémentation des verrous pour deux fils vérifiant ces propriétés.

Développement : Présentation de l'algorithme de Peterson et preuve de fonctionnement.
Extension à n fils d'exécution :

Algorithme 18.19: Algorithme de la boulangerie de Lamport

```c
void prendre(int i ){
  Acq[i] = 1;
  int t = 0;
  for(int j = 0; j < n; j++)
    t = MAX(t, num[j]);
  num[i] = t + 1;
  Acq[i] = 0;

  for(int j = 0; j < n; j++){
    while (acq[j] == 1);
    while (num[j] == 1 && (num[j] < num[i]
      || (num[j] == num[i] && j < i)) );
  }
}

void rendre(int i){
  num[i] = 0;
}
```

Obtenir un numéro

Attendre son tour pour prendre le verrou

Analogie : On prend un ticket dans une boulangerie, et on attend que ce soit notre tour.

Remarque 18.20: L'attente ici est active (on effectue des opérations quand on attend)

Implémentation 18.21: En C : Les verrous sont disponibles dans la bibliothèque pthread.

type : `pthread_mutex_t`

initialisation : `pthread_mutex_init(pthread_mutex_t *m, NULL)`

prendre : `pthread_mutex_lock(pthread_mutex_t *m)`

rendre : `pthread_mutex_unlock(pthread_mutex_t *m)`

Remarque 18.22: Une mauvaise utilisation des verrous peut créer des interblocages.

Exemple 18.23:

Fil 1 :

```c
prendre(m1);
prendre(m2);
```

Fil 2 :

```c
prendre(m2);
prendre(m1);
```

Commentaire 18.1 *Ici on ne prend pas un exemple plus gros, comme le dîner des philosophes, car ici on ne veut pas s'étendre sur ça, simplement faire une ouverture, et mentionner le problème.*

3 Les sémaphores

Analogie : Tableau des clés d'un hôtel

Définition 18.24: Un sémaphore est un compteur qui propose les opérations suivantes :

- Initialiser à une valeur entière
- décrémenter : appel bloquant, décrémente le compteur s'il est positif, attend qu'il le soit sinon
- incrémenter : incrémente le compteur. S'il devient positif, cela libère un fil si un attendait

Application 18.25: Limiter l'accès à une ressource à n fils.

Commentaire 18.2 *On ne s'appesantit pas sur cette application au vu de sa complexité. Donc même si c'est l'application la plus directe et que dans un vrai cours, on aurait peut-être ici un petit programme l'utilisant pour illustrer le concept, on se concentre nous sur des choses plus importantes.*

Remarque 18.26: On peut implémenter un verrou par un sémaphore initialisé à 1.

Implémentation 18.27: Les sémaphores sont disponibles dans la bibliothèque semaphore.h.

- type : `sem_t`
- initialisation : `sem_init`
- décrémenter : `sem_wait`
- incrémenter : `sem_post`

Propriété 18.28: L'implémentation des sémaphore est faites (normalement) sans attente active.

Application 18.29 (Problème du rendez-vous): On a p fils qui doivent se synchroniser. Chacun travaillant en deux phases. Dans la première phase, tous les fils sont indépendants et peuvent s'exécuter simultanément. La deuxième phase d'un fil ne peut débuter que si tous les fils ont terminé la première.

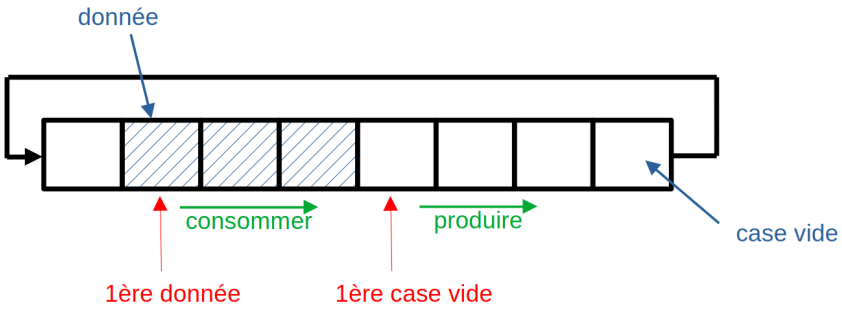
On peut résoudre ce problème à l'aide de sémaphore.

Développement : Solutions aux problèmes du rendez-vous

Application 18.30 (Producteur / Consommateur): On dispose d'un tampon de taille N, et on a deux types de fils :

- des producteurs qui produisent des ressources et les stockent dans le tampon
- des consommateurs qui consomment les données produites (consommer une donnée libère son emplacement)

Pour que la donnée consommée soit la plus vieille, le tampon est circulaire :



Problème :

- l'accès au tampon est partagé
- les consommateurs doivent attendre qu'une place soir produite

— les producteurs doivent attendre que de la place soit libérée dans le tampon.

Solution : On utilise un sémaphore indiquant le nombre données disponibles et un le nombre de cases disponibles.

```
sem_t mutex, vide, plein;
sem_init(&mutex, 0, 1);
sem_init(&vide, 0, N);
sem_init(&plein, 0, 0);
```

Producteur :

```
int item;
while(1){
  item = produire_item()
  // On attend un place
  sem_wait(&vide);
  sem_wait(&mutex);
  insert_item(item);
  sem_post(&mutex);
  // On la remplie
  sem_post(&plein);
}
```

Consommateur :

```
int item;
while(1){
  // On attend un element
  sem_wait(&plein);
  sem_wait(&mutex);
  item = remove_item();
  sem_post(&mutex);
  // On libere une place
  sem_post(&vide);
  consomer_item(item);
}
```

Commentaire 18.3 *Par manque de place, on peut moins s'étendre et ne pas écrire l'algorithme, se contenter de dire qu'on a un sémaphore qui compte les places vides et un les places pleines. (et éventuellement virer le dessin)*

Remarque 18.31: On peut implémenter un sémaphore par de l'attente active et des verrous. Donc naturellement, ce qu'apporte souvent les sémaphores, par rapport au verrou, c'est moins d'attente active.

Commentaire 18.4 *Cette remarque, car si on a des élèves un peu rapide, il pourrait trouver assez facilement des solutions n'utilisant que des verrous, et donc se poserait la question de la pertinence des sémaphores dans ce contexte. L'intérêt des sémaphores vient alors de la suppression de l'attente active.*

Commentaire 18.5 *On ne mentionne pas tous les problèmes possibles liés aux sémpahores, car ici dans cette leçon on doit parler de plein d'aspects différents. On se limite alors à en expliquer le fonctionnement global et à les illustrer.*

Leçon 19

Mémoire : du bit à l'abstraction vue par les processus.

Leçon 20

Problèmes et stratégies de cohérence et de synchronisation.

Auteur·e·s: Emile Martinez
Références :

1 Introduction

Définition 20.1: Un processus est un programme en cours d'exécution sur un ordinateur. Il dispose d'une zone mémoire en RAM (pour stocker la pile, les données de travail, etc.). Le système d'exécution identifie les processus grâce à un numéro unique appelé PID.

Définition 20.2: Un fil d'exécution (ou thread) est un sous processus qui partage la mémoire avec les autres threads du processus.

Implémentation 20.3: En C, un programme peut lancer d'autres fils d'exécution, de type `pthread_t` et que l'on lance avec la fonction `pthread_create`

Exemple 20.4: Somme parallélisée d'un tableau T de taille 2m.

Fil 1 :

```
1  for(int i = 0; i < m; i++){
2    int s = res + T[2*i];
3    res = s;  }
```

Fil 2 :

```
1  for(int i = 0; i < m; i++){
2    int s = res + T[2*i + 1];
3    res = s;  }
```

On peut donc aller deux fois plus vite.

Question : Que se passe-t-il sur l'exécution F1 :1 → F2 :2 → F1 :3 → F2 :3 ?

I Définitions générales

Définition 20.5: On appelle appel bloquant un appel à une fonction qui attendra que les conditions soit réunies avant de terminer.

Exemple 20.6: L'appel à `recv` en C qui attend que quelque chose arrive pour terminer.

Définition 20.7: On dit qu'il y a absence de famine (ou vivacité) si aucun fil n'attend indéfiniment.

Définition 20.8: On dit qu'il y a équité si aucun processus n'est favorisé.

2 Exclusion mutuelle et verrous

I Définition

Définition 20.9: Une section critique est un ensemble de morceau de code qui ne doit être exécute que par un seul fil à la fois.

Le problème de l'exclusion mutuelle est le problème consistant à garantir qu'on aura toujours au plus un fil dans une section critique.

Exemple 20.10: Les lignes 2 et 3 pour les fils 1 et 2 de l'exemple 20.4.

Remarque 20.11: Dans une section critique, les morceaux de code ne sont pas forcément les mêmes pour chaque fil (si un fil ne fais que lire quand l'autre ne fais qu'écrire dans une case partagée, leurs codes incompatibles n'est pas le même).

Solution : Les verrous

Définition 20.12: Un verrou (ou mutex) est une structure de données permettant deux opérations :

- `prendre()` : appel bloquant qui demande l'accès au verrou
- `rendre()` : appel qui libère le verrou

Propriété 20.13: Une implémentation efficace des verrous devrait garantir l'exclusion mutuelle, l'absence de famine et l'équité.

II Implémentation des verrous pour 2 fils

Définition 20.14: Une opération est dite atomique si elle ne peut pas être interrompue. Dans notre cas, une opération atomique correspond à une instruction en langage machine. On a notamment les opérations :

- lire une case mémoire
- écrire une case mémoire
- effectuer une opération arithmétique/logique

Remarque 20.15: Attention ! Une opération dans un langage de programmation n'est pas atomique en général.

Propriété 20.16: Si deux fils écrivent la même case mémoire simultanément, la case mémoire contiendra soit la valeur du premier fils soit celle du second. De même, si un fil lis dans une case quand un

autre écrit, la valeur sera écrite et le premier fil lira la valeur précédente ou la valeur actualisée.

Algorithme 20.17: Algorithme de Peterson pour un verrou à deux fils

```
tour = -1
veut_entrer = [false, false]

rendre(i):
        veut_entrer[i] = true
        tour = 1-i

        while(tour == 1-i && veut_entrer[1-i]) {}

prendre(i):
        veut_entrer[i] = false
```

Développement : Présentation de l'algorithme de Peterson.

III Généralisation à n fils

Algorithme 20.18: Algorithme de la boulangerie de Lamport

```
1  void prendre(int i ){
2    Acq[i] = 1;
3    int t = 0;
4    for(int j = 0; j < n; j++)
5    t = MAX(t, num[j]);
6    num[i] = t + 1;
7    Acq[i] = 0;
8
9    for(int j = 0; j < n; j++){
10     while (acq[j] == 1);
11     while (num[j] == 1 && (num[j] < num[i]
12     || (num[j] == num[i] && j < i)) );
13   }
14 }
15
16 void rendre(int i){
17   num[i] = 0;
18 }
```

Obtenir un numéro

Attendre son tour pour prendre le verrou

Analogie : On prend un ticket dans une boulangerie, et on attend que ce soit notre tour.

Remarque 20.19: Dans nos implémentations, l'attente est active.

Implémentation 20.20: En C : Les verrous sont disponibles dans la bibliothèque pthread.

type : `pthread_mutex_t`

initialisation : `pthread_mutex_init(pthread_mutex_t *m, NULL)`

prendre : `pthread_mutex_lock(pthread_mutex_t *m)`

rendre : `pthread_mutex_unlock(pthread_mutex_t *m)`

3 Synchronisation et sémaphores

Implémentation 20.21: Pour synchroniser des fils, la commande `pthread_join` de la bibliothèque pthread permet à un fil d'attendre la fin et de récupérer la valeur de retour d'un ou de n'importe lequel de ces fil fils.

Remarque 20.22: Néanmoins, on peut souhaiter plus de synchronisation entre les fils.

I Sémaphore et synchronisation

Bon on a un peu de redondances sur les titres là

Analogie : Tableau des clés d'un hôtel

Définition 20.23: Un sémaphore est un compteur qui propose les opérations suivantes :

- Initialiser à une valeur entière
- décrémenter : appel bloquant, décrémente le compteur s'il est positif, attend qu'il le soit sinon
- incrémenter : incrémente le compteur. S'il devient positif, cela libère un fil si un attendait

Application 20.24: Limiter l'accès à une ressource à n fils.

Commentaire 20.1 *On ne s'appesantit pas sur cette application au vu de sa complexité. Donc même si c'est l'application la plus directe et que dans un vrai cours, on aurait peut-être ici un petit programme l'utilisant pour illustrer le concept, on se concentre nous sur des choses plus importantes.*

Remarque 20.25: On peut implémenter un verrou par un sémaphore initialisé à 1.

Implémentation 20.26: Les sémaphores sont disponibles dans la bibliothèque semaphore.h.

- type : `sem_t`
- initialisation : `sem_init`
- décrémenter : `sem_wait`
- incrémenter : `sem_post`

Propriété 20.27: L'implémentation des sémaphore est faites (normalement) sans attente active.

Application 20.28 (Problème du rendez-vous): On a p fils qui doivent se synchroniser. Chacun travaillant en deux phases. Dans la première phase, tous les fils sont indépendants et peuvent s'exécuter simultanément. La deuxième phase d'un fil ne peut débuter que si tous les fils ont terminé la première.

On peut résoudre ce problème à l'aide de sémaphore.

Développement : Solutions aux problèmes du rendez-vous

II Cas classique d'utilisation

Application 20.29 (Problème du rendez-vous): On a p fils qui doivent se synchroniser. Chacun travaillant en deux phases. Dans la première phase, tous les fils sont indépendants et peuvent s'exécuter simultanément. La deuxième phase d'un fil ne peut débuter que si tous les fils ont terminé la première.

On peut résoudre ce problème à l'aide de sémaphore.

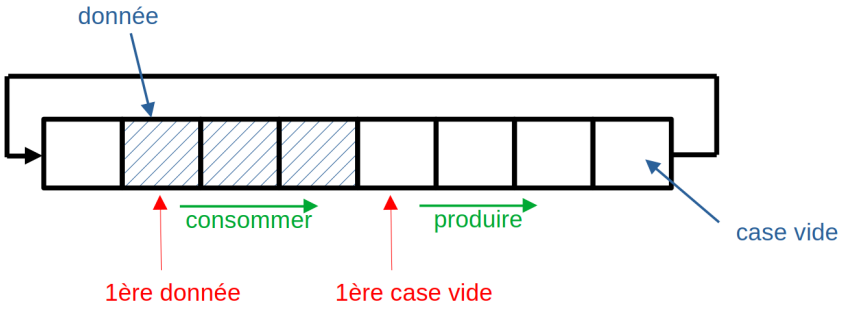
Développement : Solutions aux problèmes du rendez-vous

Application 20.30 (Producteur / Consommateur): On dispose d'un tampon de taille N, et on a deux types de fils :

— des producteurs qui produisent des ressources et les stockent dans le tampon

— des consommateurs qui consomment les données produites (consommer une donnée libère son emplacement)

Pour que la donnée consommée soit la plus vieille, le tampon est circulaire :



Problème :

— l'accès au tampon est partagé

— les consommateurs doivent attendre qu'une place soir produite

— les producteurs doivent attendre que de la place soit libérée dans le tampon.

Solution : On utilise un sémaphore indiquant le nombre données disponibles et un le nombre de cases disponibles.

```
sem_t mutex, vide, plein;
sem_init(&mutex, 0, 1);
sem_init(&vide, 0, N);
sem_init(&plein, 0, 0);
```

Producteur :

```
    int item;
    while(1){
      item = produire_item()
      // On attend un place
      sem_wait(&vide);
      sem_wait(&mutex);
      insert_item(item);
      sem_post(&mutex);
      // On la remplie
      sem_post(&plein);
  }
```

Consommateur :

```
    int item;
    while(1){
      // On attend un element
      sem_wait(&plein);
      sem_wait(&mutex);
      item = remove_item();
      sem_post(&mutex);
      // On libere une place
      sem_post(&vide);
      consomer_item(item);
  }
```

Commentaire 20.2 *Par manque de place, on peut moins s'étendre et ne pas écrire l'algorithme, se contenter de dire qu'on a un sémaphore qui compte les places vides et un les places pleines. (et éventuellement virer le dessin)*

Remarque 20.31: On peut implémenter un sémaphore par de l'attente active et des verrous. Donc naturellement, ce qu'apporte souvent les sémaphores, par rapport au verrou, c'est moins d'attente active.

Commentaire 20.3 *Cette remarque, car si on a des élèves un peu rapide, il pourrait trouver assez facilement des solutions n'utilisant que des verrous, et donc se poserait la question de la pertinence des sémaphores dans ce contexte. L'intérêt des sémaphores vient alors de la suppression de l'attente active.*

4 Interblocage

Définition 20.32: On dit qu'il y a interblocage quand tous les fils sont bloqués et ne seront jamais libérés.

Exemple 20.33:

Fil 1 :

```
1  prendre(m1);
2  prendre(m2);
```

Fil 2 :

```
1  prendre(m2);
2  prendre(m1);
```

Définition 20.34 (Diner des philosophes): On a cinq philosophes autour d'une table ronde et une fourchette entre chaque. Chaque philosophe alterne moment de faim et de réflexion. Quand il a faim, il attend de prendre ses deux fourchettes, puis il mange, les repose etc.

Solution naïve : Un verrou par fourchette, chaque philosopge attend de prendre sa fourchette gauche, puis sa droite

Problème : Interblocage

Exercice 20.35: L'implémenter pour observer cet interblocage

Solution :

- Avec des verrous : chaque philosophe essaye d'abord de prendre sa fourchette de plus petit indice
- Avec des sémaphores : On n'autorise que 4 philosophes à essayer de prendre des fourchettes grâce à un sémaphore

Commentaire 20.4 *Bon on pourrait aussi mettre dans cette partie un petit dessin.*

Leçon 21

Stockage et manipulation de données, des fichiers aux bases de données.

Auteur·e·s: Emile Martinez
Références : Balabonski, Barra

La mémoire d'un programme meurt avec lui. Néanmoins, on souhaite garder des données plus perennement.

La mémoire d'un ordinateur est alors une série de milliards de bits, parmi lesquels coexistent tout et n'importe quoi. On veut les organiser de façon à rendre leur accès le plus simple et rapide possible.

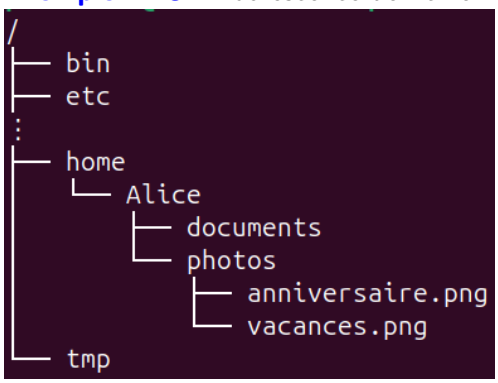
1 Fichiers

I Organisation et manipulation

Définition 21.1: Un fichier est un ensemble de données. C'est l'unité de stockage manipulé par l'utilisateur.

Définition 21.2: Pour organiser des données de manière persistante sur un disque on utilise une arborescence de fichier. La norme POSIX, suivie par la plupart des OS (linux, Mac, android) définit cette organisation et sa manipulation.

Exemple 21.3: Arborescence de fichier linux (obtenue par la commande `tree`)



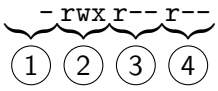
Définition 21.4: Le chemin d'accès vers un fichier est soit exprimé de manière absolu (depuis la racine) soit depuis le répertoire courant

Implémentation 21.5: On peut accèder et manipuler l'arborescence de fichier depuis un invite de commande (shell ou terminal) avec les commandes suivantes :

- pwd : affiche le repertoire courant
- cd chemin : change le repertoire courant pour la cible du chemin
- mkdir/touch : créer un dossier/un fichier
- cp/mv/rm : copie/ déplace et renomme / supprime
- ls : liste le contenu d'un repertoire

Exercice 21.6: Prise en main des commandes du shell avec appuie sur l'utilisation du man.

Implémentation 21.7: La commande ls -l permet de faire apparaitre des informations supplémentaires sur les fichiers : propriétaire, groupe prioritaire, taille, dernière modification et autorisations :



(1) : type de fichier :
- `d` : répertoire
- `l` : lien symbolique
- `-` : autre

(2)(3)(4) :
- `r` : read
- `w` : write
- `x` : execute

(2) : permissions propriétaire
(3) : permissions groupe
(4) : permissions tous

Remarque 21.8: Pour Linux, "tout est un fichier" : les codes sources, les exécutables, mais aussi les périphériques qui peuvent être lus (souris, clavier) ou écrits (écran) comme des fichiers

Propriété 21.9: Plusieurs systèmes de fichiers (volume dans la mémoire) peuvent cohabiter sur un même ordinateur, avec chacun leur racine

Exemple 21.10: Les C : :, D : :, E : :, etc. sous Windows sont autant de système de fichiers

Remarque 21.11: Sous Linux, tous les espaces de stockages montés partagent la même arborescence

II Stockage

Il existe plusieurs manières de faire un système de fichier (FAT32, ext4, ...) définissant des primitives de gestion des fichiers ainsi que des structures de données pour la gestion des espaces libres.

Définition 21.12: On découpe alors les fichiers en blocs de quelques ko. Le système de fichier ne manipulera alors que des blocs.

Définition 21.13: Un fichier étant souvent trop grand pour un unique bloc, il est séparé en plusieurs blocs :

— allocation contigüe : les blocs sont contigües sur le disque
→ accès séquentiel rapide mais fragmentation et difficulté à créer ou etendre des fichiers.

— allocation chaînée : les blocs peuvent être n'importe où, chaque bloc contenant l'adresse du suivant
→ bonne utilisation de la mémoire, création extension facile mais accès séquentiel lent
On dispose alors parfois d'une table d'allocation de fichiers

— l'allocation indexée : les adresses des blocs constituant un même fichier sont rangées dans une table, appelée index, elle-même con
tenue dans un ou plusieurs blocs.
→ bonne accès séquentiel et extension facile, mais taille de fichier maximales et utilisation de mémoire annexe (visible surtout sur les petits fichiers)

Définition 21.14: Chaque fichier se voit associé un numéro inode à un emplacement de stockage. L'inode permet de retrouver dans une table du périphérique de stockage des infos données par `ls -l`.

Définition 21.15: Pour économiser de la place, des liens peuventêtre crées entre des fichiers avec :
`ln` : lien physique, l'inode est partagé mais la suppression d'un des fichiers n'impacte pas l'autre
`ln -s` : lien symbolique, un nouvel inode est utilisé et le fichier ne contient que le chemin vers sa source.

2 Format

I Fichier texte

Définition 21.16: Un fichier texte représente uniquement une suite de caractères (type char en C ou en OCaml) codé en ASCII.

Remarque 21.17: C'est le format de fichier de base.

Remarque 21.18: Il arrive que l'on veuille représenter plus que les 128 caractères qu'autorise l'ASCII (ou 256 pour l'ASCII étendue). On peut alors utiliser des codages sur plus de bits (16 pour l'unicode par exemple).

Définition 21.19: Étant le format de fichier le plus basique, il est le plus simple à manipuler. On pourra alors accéder à ces fichiers en langage de programmation :

Fonction	En C	En OCaml
ouvrir un fichier	`fopen(chemin, mode)`	`open_in`
fermer un fichier	`fclose(fichier)`	`close_in`
écrire dans un fichier	`fprintf`	`input_line`
lire dans un fichier	`scanf`	`output_line`

Remarque 21.20: Lorsqu'on utilise `printf` en C, on écrit dans un fichier particulier : la sortie standard (stdout) qui correspond à l'invite de commande. `printf` est donc équivalent à `fprintf(stdout, ...)`. De même, `scanf` lit dans l'entrée standard (stdin).

Définition 21.21: Pour rediriger la sortie standard, on peut utiliser des commandes :

- `commande > filename` : la sortie standard de la commande est écrite ddans le fichier, qui est écrasé.
- `commande >> filemane` : même chose mais sans écraser le fichier.
- `commande < fichier` : le fichier devient l'entrée standard de la commande
- `commande1 | commande2` : la sortie standard de la première commande est reliée à l'entrée standard de la deuxieme.

Remarque 21.22: L'écriture étant lente, un tampo est utilisé. On peut forcer l'écriture des tampons avec `flush` en OCaml et `fflush` en C.

II Formats de fichiers

Pour représenter plus que des chaînes caractères, on a besoin de définir des formats de fichiers qui indiquent comment interpréter les bits de données.

Définition 21.23: Un format de fichier est une convention de représentation de données

Remarque 21.24: Pour gagner de l'espace, ces formats utilisent souvent des méthodes de compression, avec ou sans perte.

Exemple 21.25: Quelques formats particuliers :

- Le format png stocke et compresse les images sans perte
- Le format jpeg stocke des images compressées avec perte
- Le formats mp3 stocke des sons compressés avec perte
- Le format mp4 combine audio et vidéo
- Le format zip compresse sans perte des fichiers quelconques

Remarque 21.26: Le format zip utilise l'algorithme de compression LZW, mais aussi le codage de Huffman.

Développement : Présentation de l'algorithme LZW.

Remarque 21.27: Il y a toujours un compromis à faire entre différents objectifs (ex : compression et facilité d'utilisation). Ainsi la plupart des formats se spécialisent dans une utilisation :

— quand on ouvre une image en python, on la transforme en tableau de triplets, quand on la sauvegarde on la recompresse, dans un format moins manipulable

— Quand on édite une vidéo, on doit l'exporter à la fin pour passer d'un format manipulable à un format pour la lecture

— Quand on compresse des fichiers en .zip, on ne peut plus les modifier ou les lire

raison pour lesquelles on exporte quand on monte une vidéo, pour baser d'un format éditable à un format compressé adapté à la lecture).

Exemple 21.28: Le format CSV (pour comma separated values) est un format de texte brut permettant de stocker des données sous forme de table, permettant ainsi facilement la suppresion, l'ajout, etc.

Pour des commandes { produit : tomate, prix : 3 quantité : 50, client : Le navet naviguant, adresse : 13 rue du Swag à Tarbes }, { produit : patate, prix : 1, quantité : 30, client : Le navet naviguant, adresse : 13 rue du Swag à Tarbes }, ...

Pourrait-on mieux faire ?

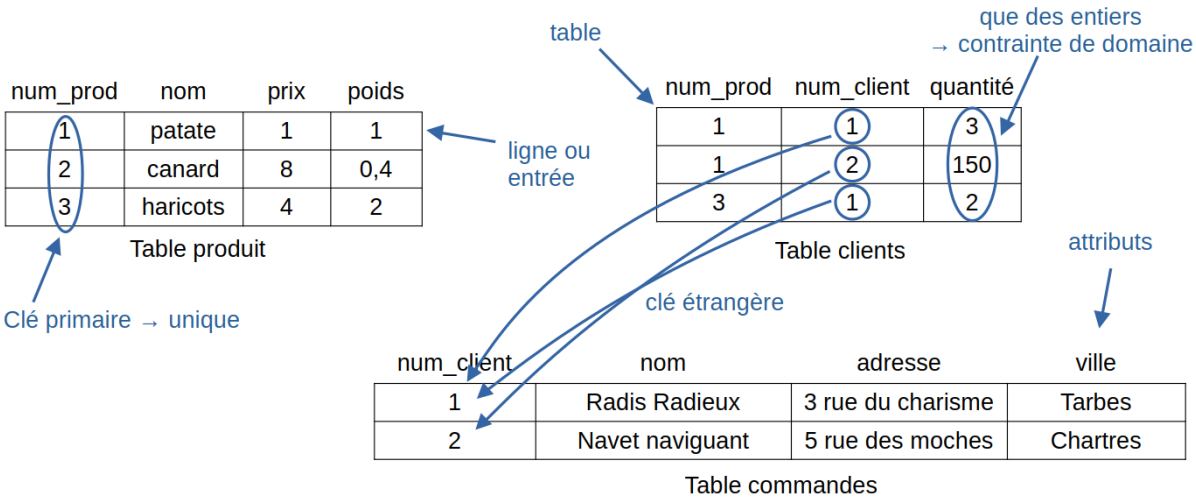
3 Bases de données

Souvent, les données d'une table ont des redondances, et des liens entre elles (cf exemple au dessus). Pour manipuler des gros volumes de données on ne se contente alors plus alors de fichiers en texte brut.

Définition 21.29: Le modèle relationnel est une manière de représenter les données en exploitant les relations entre elles.

Exemple 21.30: Un grossistes gérant des commandes.

`Produit(num_produit, nom, prix, poids)`
`Clients(num_client, nom, adresse, ville)`
`Commande(#num_produit, #num_client, quantite)`



Définition 21.31: Un SGBD (système de gestion de bases de données) est utilisé pour manipuler des données relationnelles, garantissant les propriétés ACID (atomicité, cohérence, isolation et durabilité).

On intéragit avec lui à travers le langage SQL.

Exemple 21.32: Le SQL permet de sélectionner certaines données, de les trier, de les filtrer selons certaines conditions, etc ...

Commentaire 21.1 *Si on a la place ici, on peut insérer des mots clés.*

Théorème 21.33 (Codd): SQL est suffisamment expressif pour quasiment tout ce que l'on souhaite faire

Exercice 21.34: Trouver différentes manières de calculer le max et la division.

Développement : Correction de l'exercice précédent.

Commentaire 21.2 *Ici on pourrait également défendre le fait de faire plutôt l'autre développement de requêtes SQL, introduisant les requêtes, les mots clés de bases, etc. Il s'insère parfaitement dans la leçon, (mieux que ce développement là où il faut justifier que on a introduit aucune syntaxe dans le cours, mais que en vrai ils la connaîtraient), mais c'est un développement moins poussé. Donc on montre moins la puissance de calcul.*

Leçon 22

Fonctions et circuits booléens en architecture des ordinateurs

Auteur·e·s: Emile Martinez
Niveau : Prépa / Première
Pré-requis : Représentation binaire et pour la première partie : induction, graphe
Références :

1 Cadre théorique (Prépa)

I Expression booléenne

Définition 22.1: On définit l'ensemble EB des expressions booléennes par induction avec la signature

- cas de bases : $\top$, $\bot$, V un ensemble de variables
- constructeurs :
  - $\star$ $\neg$ un constructeur unaire
  - $\star$ $\vee$, $\wedge$ deux constructeurs binaires

Informellement : $\top, \bot, x \in V$ sont des expressions booléennes, et si e_1 et e_2 le sont, alors $\neg e_1$, $e_1 \vee e_2$, et $e_1 \wedge e_2$ le sont.

Exercice 22.2: Définir inductivement les variables apparaissant dans une formule et en déduire une définition rigoureuse du nombre de variables d'une fonction.

Définition 22.3: Une valuation est une fonction $\sigma : V \to \{0,1\}$.

On définit alors $[\,]_\sigma : EB \to \{0,1\}$ par induction sur EB par :

- $[\bot]_\sigma = 0$, $[\top]_\sigma = 1$ et $[x]_\sigma = \sigma(x)$ pour $x \in V$
- $[e_1 \vee e_2] = \max([e_1]_\sigma, [e_2]\sigma)$
- $[e_1 \wedge e_2] = \min([e_1]_\sigma, [e_2]\sigma)$
- $[\neg e_1]_\sigma = 1 - [e_1]_\sigma$

Définition 22.4: Pour $a, b \in EB$, on dit que a et b sont équivalentes noté $a \equiv b$, si $\forall \sigma : V \to \{0,1\}, [a]_\sigma = [b]_\sigma$

Définition 22.5: On peut éventuellement définir le XOR (ou exclusif), avec $a \oplus b = (a \vee b) \wedge \neg(a \wedge b) \equiv (a \wedge \neg b) \vee (\neg a \wedge b)$

II Fonctions booléennes

Définition 22.6: Une fonction booléenne est une fonction de $\{0,1\}^n \to \{0,1\}^m$

Exemple 22.7: $f : \{0,1\}^{64} \to \{0,1\}^{64}$ prenant en entrée le codage (en double) d'un flottant x, et renvoyant le codage du flottant de e^x

Remarque 22.8: On se limite parfois à $f : \{0,1\}^n \to \{0,1\}$ (en décomposant $f : \{O,1\}^n \to \{0,1\}^m$ en m fonctions sur chaque dimension)

Définition 22.9 (Table de vérité): La table de vérité d'une fonction booléenne est la donnée de sa valeur sur toutes ses entrées possibles. On représente la table de vérité de $f : \{0,1\}^n \to \{0,1\}^m$ dans un tableau avec $n+m$ colonnes, et pour chaque éléments de $\{0,1\}^n$, une ligne avec la valeur du n-uplets, et la valeur du m-uplets de sortie.

Remarque 22.10: La table a 2^n lignes

Commentaire 22.1 *Bon là normalement il faudrait en mettre une, mais bon, il faut aussi que jeunesse se passe.*

Remarque 22.11: On se limite parfois à $f : \{0,1\}^n \to \{0,1\}$ (en décomposant $f : \{O,1\}^n \to \{0,1\}^m$ en m fonctions sur chaque dimension)

On peut définir une fonction booléenne :

— par extension (en donnant sa table de vérité)

— par intention (en donnant ses propriétés (ex : 1 ssi le nombre de variables à 1 est un nombre premier))

— par une expression booléenne.

Théorème 22.12: Toute fonction $f : \{0,1\}^n -> \{0,1\}$ est équivalente à une expression booléenne.
Plus rigoureusement, il existe une formule e dont les variables sont $x_1, \ldots, x_n$ et telle que pour toute valuation σ, $[e]_\sigma = f(\sigma(x_1), \ldots, \sigma(x_n))$

Développement : Preuve du théorème 22.12 et discussion sur la complexité.

III Représentation des expressions booléennes par des graphes orientés acycliques

Définition 22.13 (Circuit booléen): Un circuit est un graphe orienté acyclique (DAG) étiquetés par $V \cup \{\top, \bot, \neg, \wedge, \vee\}$ où un sommet d'étiquette x a :

- Un degré entrant 0 si $x \in V \cup \{\top, \bot\}$
- Un degré entrant 1 si $x = \neg$

- Un degré entrant 2 si $x \in \{\wedge, \vee\}$

Remarque 22.14: On peut ne pas prendre également des étiquettes supplémentaires (comme $\oplus$, le NAND, NOR, etc.)

Définition 22.15: On définit l'évaluation d'un graphe par une valuation $\sigma : V \to \{0, 1\}$ comme un étiquetage des nœuds où un nœud d'étiquette a vaudra :

- $[a]_\sigma$ si $a \in V \cup \{\top, \bot\}$
- $1 - y$ si $a = \neg$ et le sommet entrant de a est évalué en y
- $\min(y, z)$ (resp. $\max(y, z)$) si $a = \wedge$ (reps.$\vee$) et les sommets entrants sont évalués en y et z

Propriété 22.16: Cette définition est correcte (et ne boucle pas à l'infini)

Remarque 22.17:

- Les degrés sortants ne sont pas limités.
- Si les degrés sortants sont exactement 1 sauf pour un sommet où c'est 0, on obtient exactement les expressions booléennes (avec leur représentation sous forme d'arbres)
- Les circuits booléens correspondraient à un ensemble d'expressions booléennes où l'on aurait le droit de définir des alias

Conclusion 22.18: Les circuits booléens peuvent représenter tout ce qu'on veut calculer de finis et qu'on peut coder en binaire.

2 Dans un ordinateur

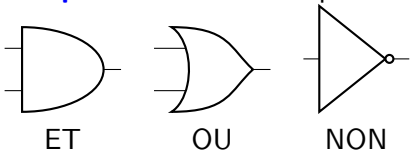
I Circuits combinatoires

Définition 22.19: Un bit est la plus petite unité d'information d'un ordinateur, ne pouvant prendre que deux valeurs (0V - +5V, 0 - 1, Ying - Yang, Ouvert - Fermé, ...). Fixons nous sur 0-1.

Dans les circuits électroniques, on arrive à manipuler des bits. Nous sommes par exemples capables de prendre un bit et de l'inverser, d'en prendre deux et de renvoyer 1 si au moins l'un des deux vaut 1, etc...

Définition 22.20 (porte logique): Une porte logique est un circuit électronique réalisant des opérations logiques sur une séquence de bits

Exemple 22.21: On est capable de faire les portes ET, OU, NON, ...



Définition 22.22 (circuits combinatoires): Un circuit combinatoire est alors une succession de portes logiques dont la sortie de certaines sont branchés sur l'entrée d'autres, et sans cycles.

Remarque 22.23: C'est l'implémentation physique des circuits booléens.

Conclusion 22.24: Les parties I-II, I-III et II-I nous donnent que l'on est capable électroniquement de calculer tous qui est représentable de manière fini par des bits.

3 Mesure d'un circuit

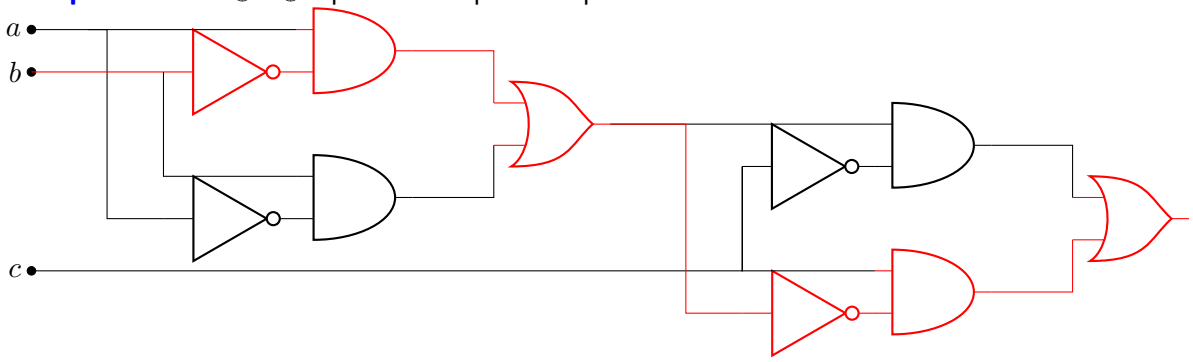
Idée 22.25: Quand on fait un circuit booléens, on peut vouloir maximiser différents critères :

- On veut, pour des raisons économiques, minimiser le nombre de transistors (et donc le nombre de portes)
- On veut tirer des câbles courts, et donc avoir un circuit compact
- Comme il y a un délai pour qu'une porte calcule sa sortie selon son entrée, on veut minimiser le délai total de mise à jour du circuit.

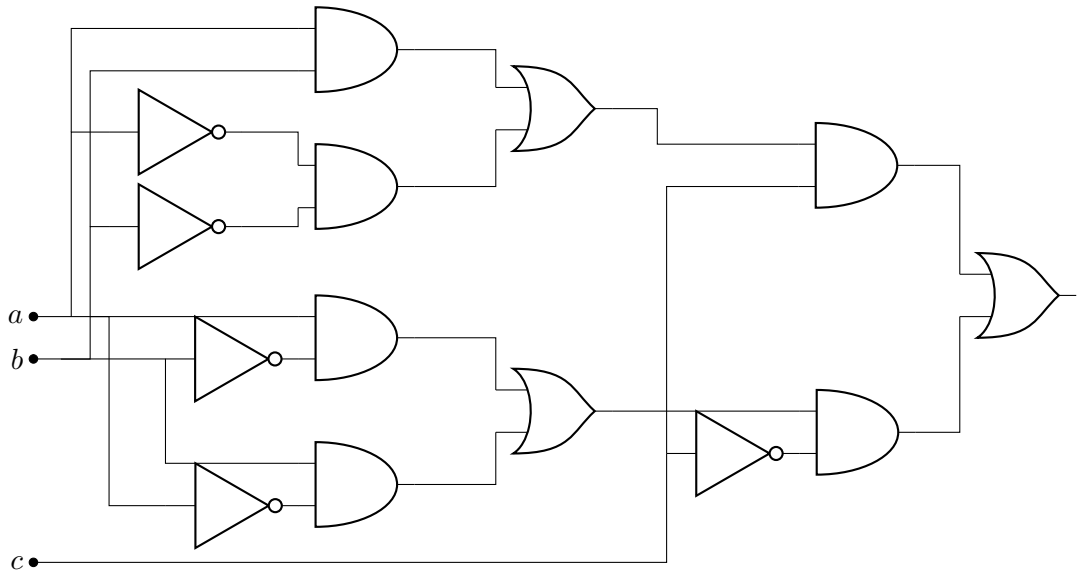
Définition 22.26 (chemin critique): Le chemin critique d'un circuit booléen (et donc d'un circuit combinatoire) est le (un) plus long chemin entre une entrée (degré entrant 0) et une sortie (degré sortant 0). Le graphe étant acyclique, c'est donc un plus long chemin.

Idée 22.27: Minimiser la longueur du chemin critique, qui est proportionnelle au délai que met un circuit combinatoire à effectuer un calcul.

Exemple 22.28: $a \oplus b \oplus c$ peut-être représenté par deux XOR successifs



On a alors 10 portes et un chemin critique de taille 6. On peut le diminuer à 5, en calculant directement la négation de $a \oplus b$, au lieu de réutiliser le résultat de $a \oplus b$



Néanmoins, on a maintenant 14 portes.

Commentaire 22.2 *Cet exemple sur un exemple non trivial le fait que on peut raccourci le chemin critique, et le fait que on peut le faire potentiellement au détriment du nombre de noeuds. (il faut voir le deuxième comme le premier, sauf qu'on calcule directement la négation du xor). Mais bon, cet exemple est un peu gros à mettre dans le plan quoi. Si on veut un truc plus cours, on peut passer de* $a \wedge (b \vee c) \wedge d$ *à* $(a \wedge d) \wedge (b \vee c)$ *mais on rajoute pas de portes, et cet exemple est bateau et donc moins intéressant que le XOR ternaire. Mais bon, il faut savoir vivre avec son temps.*

4 Des circuits particuliers

I Additionneur n bits

Algorithme 22.29: On reprend l'algorithme classique d'addition, adapté au binaires :

Algorithme 22.1 : $Addition(x, y)$

Entrées : x et y deux nombres de n chiffres en binaire

$r_{-1} \leftarrow 0$

pour $i = 0$ à $n - 1$ **faire**

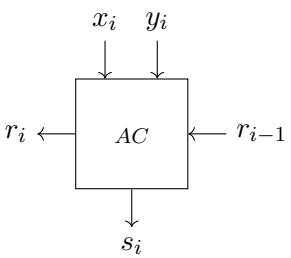
`// ` r_i `est la` i`-ème retenue,` s_i `le` i`-ème bit de sortie`

$r_i, s_i = AC(r_{i_1}, x_i, y_i)$

avec AC (additionneur complet) faisant l'addition de trois chiffres (0 ou 1) définie par la table de vérité :

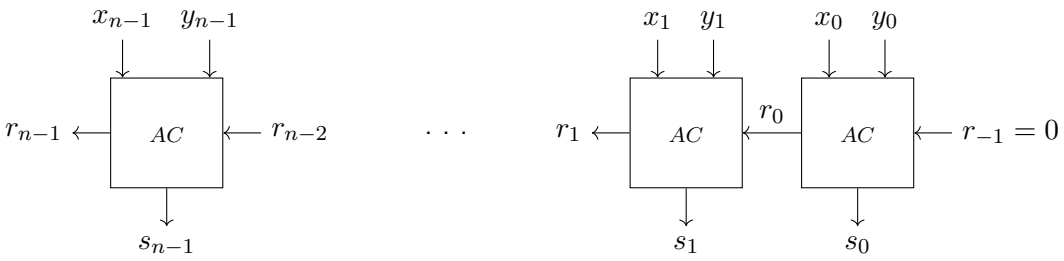
r_{i-1}	x_i	y_i	r_i	s_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

que l'on représente par



Exercice 22.30: Proposer un circuit booléen pour cette table.

En mettant plusieurs AC à la suite, on créer alors un additionneur n bits :



Problème 22.31: Le chemin critique est linéaire en n (car il faut propager la retenue).

Développement : Construction par une méthode D&R d'un additionneur n bits à retenue anticipée.

II Multiplexeur

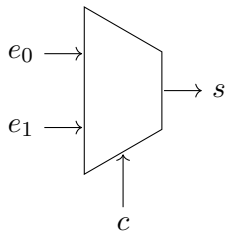
Définition 22.32: Un multiplexeur à deux entrées est un circuit booléen servant à sélectionner une entrée. Une troisième entrée de contrôle, determine si la sortie sera la première ou la deuxième entrée.

On lui associe la table de vérité

e_0	e_1	c	s
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

$(s = (c \wedge e_0) \vee (\neg c \wedge e_1))$

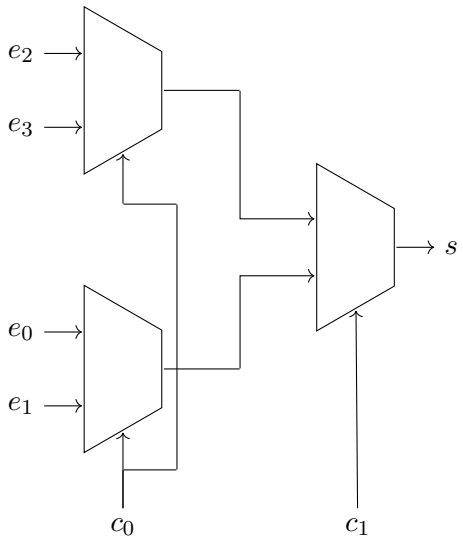
représenté par



Exercice 22.33: Construire un circuit booléen pour le multiplexeur à deux entrées.

Remarque 22.34: En combinant les multiplexeurs à deux entrées, on peut générer des multiplexeurs à 2^k entrées.

Exemple 22.35 (Multiplexeur à 4 entrées):



Remarque 22.36: Si on interprète c_1c_0 comme un nombre en binaire, sa valeur correspond à l'entrée sélectionné.

Application 22.37: On peut alors créer un sélectionneur d'adresse avec un chemin critique de taille $O(\log n)$ ce qui est optimal.

Commentaire 22.3 *Là si on a la place, pour insister sur l'aspect archi de la leçon, on peut parler plus longuement de cette application, en disant qu'on en met en parralèle pour avoir plus de données, qu'on le fait sur plus de bits d'adresse, sur le fait qu'on utilise qu'un nombre linéaire de portes.*

Exercice 22.38: Faire un démultiplexeur à 2^k entrées.

Leçon 23

Principes de fonctionnement des ordinateurs : architecture, notions d'assembleur.

Auteur·e·s: Emile Martinez
Niveau :
Pré-requis : Représentation des nombres en binaires
Références :

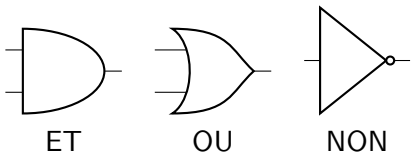
1 Circuits booléens

I Porte logique

Les circuits d'un ordinateur manipulent des bits qui correspondent en interne à des tensions électriques.

Définition 23.1: Une porte logique est une fonction qui prend un ou plusieurs bits en entrée et qui renvoie un bit en sortie.

Schéma 23.2:



Propriété 23.3: On peut composer les portes logiques, et scinder un fil : on crée alors des circuits booléens. Attention, on ne doit pas créer de boucles dans les circuits.

Exercice 23.4: Exprimer la porte OU à l'aide des portes NON et ET.

II Expressivité

Définition 23.5: On définit inductivement l'ensemble EB des expressions booléennes par :

Cas de base : $\top, \bot, x \in V$ (où V est un ensemble de variable)

Constructeurs : $\neg$ unaire, $\wedge$ et $\vee$ binaires

Remarque 23.6: Cela représente exactement les circuits booléens, où seuls les fils initiaux peuvent être dupliqués

Définition 23.7: Une valuation est une fonction $\sigma : V \to \{0,1\}$.

On définit alors $[\,]_\sigma : EB \to \{0,1\}$ par induction sur EB par :

- $[\bot]_\sigma = 0$, $[\top]_\sigma = 1$ et $[x]_\sigma = \sigma(x)$ pour $x \in V$
- $[e_1 \vee e_2] = \max([e_1]_\sigma, [e_2]\sigma)$
- $[e_1 \wedge e_2] = \min([e_1]_\sigma, [e_2]\sigma)$
- $[\neg e_1]_\sigma = 1 - [e_1]_\sigma$

Remarque 23.8: Cela revient à simuler l'exécution d'un circuit booléen.

Théorème 23.9: Pour toute fonction $f : \{0,1\}^n \to \{0,1\}$, il exsite une expression booléenne e ayant pour variables $\{x_1, \ldots, x_n\}$ tel que pour tout $(b_1, \ldots, b_n) \in \{0,1\}^n$, en prenant σ tel que $\sigma(x_i) = b_i$ on ait alors $[e]_\sigma = f(b_1, \ldots, b_n)$

Développement : Preuve du théorème précédent et discussion autour de la complexité

Conclusion 23.10: Les circuits booléens permettent d'exprimer toutes les fonctions que l'on pourrait vouloir calculer

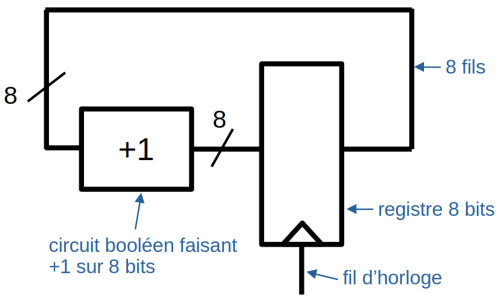
III Introduction du temps

Notre ordinateur est donc composé de circuits booléens. Néanmoins, on voudrait pouvoir brancher les circuits booléens entre eux (ce qui posent problème car les portes logiques ne changent pas de valeurs instantanément) et avoir de la rétroaction (ce qui est interdit).

Idée 23.11: On introduit alors des briques de mémoire dans un ordinateur (registre) et une horologe (un tic tac). Les liens entre les différentes circuits ne se font alors que à travers des registres, qui se mettent à jour en même temps grâce à l'horologe. Ainsi, on a jamais de réelles boucles.

Remarque 23.12: Les registres sont les plus petites unités de mémoire d'un ordinateur.

Exemple 23.13: Compteur sur 8 bits, faisant $+1$ à chaque tac du tic tac.



Remarque 23.14: La fréquence de l'horloge détermine donc le temps minimal pour faire une opération

dans un ordinateur. C'est ce que l'on dit quand on parle de processeur 4GHz (4 milliards de tac par secondes)

2 Modèle de Von Neuman

I Le modèle

Définition 23.15: Une instruction est une opération à effectuer par le processeur sur des éléments de mémoire de l'ordinateur (registre, cache, RAM, disque dur, etc...).

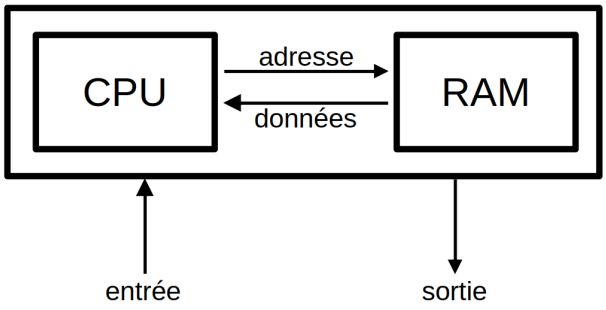
Principe 23.16: Un ordinateur passe alors son temps à exécuter des instructions (qui modifie l'état de la mémoire)

Définition 23.17: Le modèle de Von Neumann décrit le fonctionnement d'un ordinateur constitué :

- d'un processeur qui lit les instructions en mémoire et les exécute. Il accède à la mémoire par blocs appelés mots mémoire. Pour cet accès, le processeur utilise un registre appelé compteur ordinal (Program Counter ou PC) qui contient une adresse en mémoire.
- La mémoire RAM adressé qui contient les programme à exécuter et les données.
- Les périphériques d'entrée (clavier, souris, disque dur) et de sortie (écran, disque dur, haut parleur).

Remarque 23.18: Une des spécificités de ce modèle est que les instructions sont des données comme les autres.

Schéma 23.19 (Modèle de Von Neumann): CPU = processeur



Définition 23.20 (Cylce de Von Neumann):

1. Lire le mot mémoire qui commence à l'adresse PC
2. Interpréter ce mot mémoire comme une instruction et l'exécuter
3. Augmenter le PC pour passer à l'instruction suivante
4. Retourner au 1

Remarque 23.21: Quand vous avez plusieurs coeurs, il y a simplement plusieurs processeurs en parrallèle.

II Registres et mémoire adressable

Dans le modèle, la mémoire n'est pas dans le processeur. C'est la mémoire adressable (accessible par adresse).

Néanmoins, il y a aussi de la mémoire dans le processeur. Ce sont les registres. Quand un processeur veut exécuter une opération, il doit alors stocker les données dans des registres (les avoir en mémoire, sous la main), faire l'opération (et comme il doit stocker le résultat, le mettre dans un registre), puis renvoyer le résultat à la mémoire.

Remarque 23.22: Un processeur possède un registre stockant la valeur de PC.

3 Jeu d'instruction

I Définition

Définition 23.23: Une instruction machine est une séquence de bits que le processeur peut interpréter et exécuter. Un jeu d'instructions (ISA) définit quelles sont les instructions supportées par le processeur, et la façon dont elles sont représentées en mémoire. L'ensemble de ces instructions machines forment le langage reconnu par le processeur, appelé langage machine.

Remarque 23.24: Tous les codes dans n'importe quel langage de programmation (exemple Python) doivent être traduits en langage machine pour être exécutés par le processeur.

Propriété 23.25: En général, les jeux d'instructions gèrent les opérations suivantes :

- lire le contenu d'une case mémoire dans un registre, écrire le contenu d'un registre dans une case mémoire
- opérations arithmétiques ou logiques (addition, et bit à bit, etc...)
- se déplacer dans le programme que l'on exécute (sauts)

Remarque 23.26: Il exsite de nombreux ISA différents. On peut les diviser en deux catégories principales :

CISC (complex instruction set computer) : plus d'instruction pouvant faire plus de choses mais donc plus longues (ex. x86 sur la plupart des ordinateurs)

RISC (reduced instruction set computer) : moins d'instruction mais plus rapide et facile à implémenter (ex : RISC-V, ARM sur des téléphones).

II Langage assembleur

Une instruction est représentée en mémoire par un code en binaire. Par exemple, si les trois premiers bits sont des 0, on doit faire une opération arithmétique, puis si les deux suivants sont des 1, on fait une addition, etc. Néanmoins, cela est très peu lisible par l'être humain.

Définition 23.27: Le langage assembleur représente le langage machine sous une forme lisible par un humain. C'est le langage de programmation de plus bas niveau.

Exemple 23.28: Exemple d'instruction classique :

- `load` r_i `[`r_j`]` : mets le contenu à l'addresse contenu dans r_j dans le registre r_i
- `store` r_i `[`r_j`]` : mets le contenu du registre r_i dans la case d'adresse contenu dans r_j
- `add` r_i r_j r_k : ajoute le contenu des registres r_i et r_j pour le mettre dans le registre r_k
- `iload` r_i x : mets la valeur x dans le registre r_i
- `mv` r_i r_j : mets la valeur du registre r_i dans le registre r_j.

Exemple 23.29: Traduction en langage assembleur du code python «`z = x + y`»

```
iload r1 (adresse de x)
load r2 [r1]
iload r1 (adresse de y)
load r3 [r1]
add r1 r2 r3
iload r1 (adresse de z)
store r2 [r1]
```

Remarque 23.30: Pour pouvoir exécuter des boucles et les si, on introduit de nouvelles instructions :

- `bge` r_1 r_2 `N` : va à la ligne N si $r_1 \geq r_2$;
- `bgt`, `ble`, `beq`, `blt` (pour $>$, $\leq$, $=$, $<$)
- `jump` N : saute à la N-ième instruction du programme

Exemple 23.31: Pour `while (i < n): i = i + 1`

```
0. iload r1 (adresse de x)
1. iload r2 (adresse de n)
2. load r2 [r2]              // contient n
3. load r3 [r1]              // contient i
4. bge r3 r2 9
5. iload r4 1
6. add r3 r3 r4
7. store r3 [r1]             // mettre à jour i
8. jump 3
9.
```

Remarque 23.32: On peut faire beaucoup d'optimisation (par exemple en ne stockant i que à la fin). Ce sont là d'importants sujet d'études (en compilation)

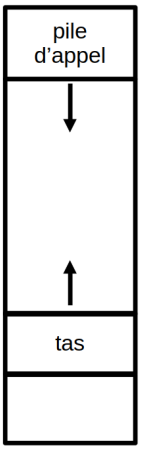
4 Gestion de la mémoire à plus haut niveau

Commentaire 23.1 *La raison de cette partie n'est pas uniquement de nous donner un développement mieux, mais également que c'est l'étape d'après dans la construction d'un ordinateur. Ici on commence à poser les premières briques du dessus. Surtout que c'est une étape cruciale de la compilation (passage de langage de plus haut niveau au code machine). Donc en tant qu'ouverture, tout en utilisant le langage assembleur ca parait cohérent.*

Quand on exécute un programme qui n'est pas dans un langage assembleur, on a souvent besoin d'instruction de plus haut niveau, nous permettant d'allouer des variables, de faire des appels de fonctions imbriqués, etc. ce qui n'est à priori pas disponibles dans le langage assembleur tel quel.

Il faut alors traduire ces instructions de plus haut niveau en langage assembleur (i.e. compiler)

Principe 23.33: Lors de l'exécution d'un processus, un espace mémoire en RAM lui est réservé.



Principe 23.34: Le tas gère les données accessibles depuis tout le programme (malloc).
Dans la pile, on met les variables locales à une fonction. A chaque nouvel appel, on empile de l'espace pour l'appel de fonction, que l'on enlève quand on return.

Développement : Explication de la pile d'appel et implémentation en assembleur

Leçon 24

Echange de données et routage. Exemples.

Y a une version sur les pads, mais bon, elle laisse peut-être un peu à désirer

Leçon 25

Client-serveur : des sockets TCP aux requêtes HTTP

Auteur·e·s: Emile Martinez
Niveau : Lycée
Pré-requis : Aucun
Références : Tannenbaum, Balabonski 1ère

1 Modèle Client-Serveur

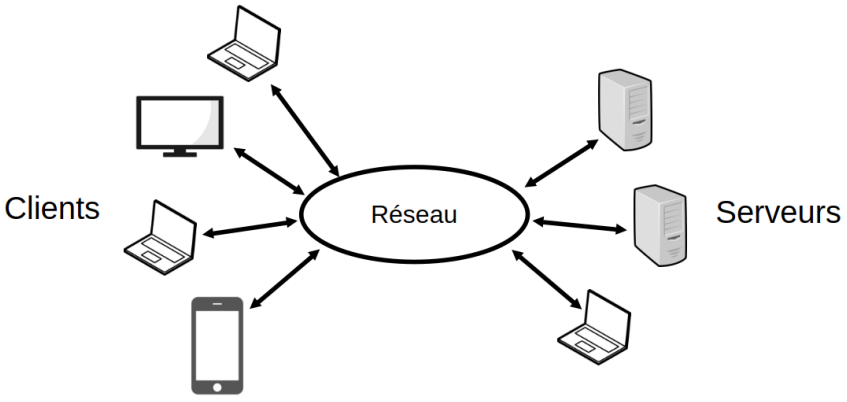
I Le modèle

Définition 25.1 (Client/Serveur): Dans un réseau informatique, les échanges de données entre deux ordinateurs sont souvent asymétriques. Un ordinateur, désigné comme le client, demande généralement des ressources (des données) à un autre ordinateur, appelé serveur, qui les fournit à tous ceux qui en font la demande.

Exemple 25.2: En tapant «http ://www.google.fr», votre machine va chercher à entrer en communication avec le serveur portant le nom «google.fr». Votre machine est ici client.

Remarque 25.3: N'importe quel type d'ordinateur peut jouer le rôle de serveur, mais dans le monde professionnel les serveurs sont des machines spécialisées conçues pour fonctionner 24h sur 24h.

Schéma 25.4:



Remarque 25.5: Un même ordinateur peut être à la fois plusieurs clients et plusieurs serveur

Remarque 25.6: Dans ce modèle les clients ne communiquent pas entre eux (ni les serveurs entre eux). Seuls des clients communiquent avec des serveurs

II Caractéristiques concrètes

Principe 25.7: Caractéristiques d'un serveur :

- ⋆ Son adresse IP est permanente, et le serveur attend une connexion entrante sur un ou plusieurs ports. On dit qu'il écoute.
- ⋆ A la connexion d'un client sur le port en écoute, il ouvre une socket (interface de connexion) [a]. Il n'est pas à l'initiative de la connexion
- ⋆ A la suite de la connexion, le processus serveur communique avec le client suivant un protocole établi préalablement. L'action réalisée par le serveur en réponse à la requête client est souvent appelée service.

a. traduction proposée par wikipédia, mais qui ne semble pas universellement admise

Principe 25.8: Caractéristiques d'un client :

- — Il est à l'initiative de l'établissement la connexion avec le serveur
- — Lorsque la connexion est acceptée par le serveur, il communique comme le prévoit la couche application du modèle OSI

Commentaire 25.1 *Bon ces deux caractérisations demandent à être améliorées, et éventuellement élaguées. De plus, on peut se poser la question si pour bien expliquer que ce n'est pas en tant que tel une machine qui est client ou serveur, mais une application/un programme, on pourrait pas mettre «caractéristiques d'un programme serveur /client». Mais un client pouvant être plus qu'un programme .... Tout cela demande réflexion.*

Le protocole d'échange se divise alors ici en deux couches : la couche transport et la couche application.

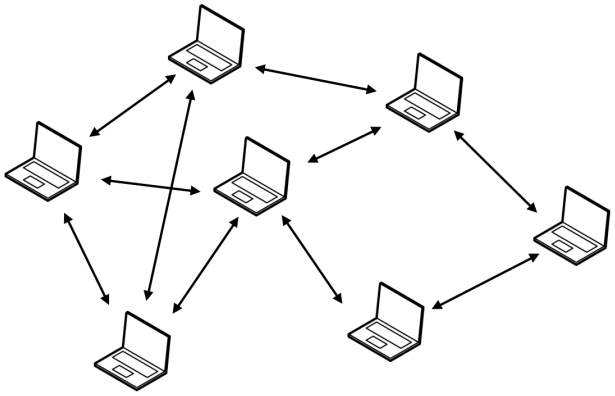
Définition 25.9 (Couche transport): C'est le trait de la machine vers le réseau dans le schéma 25.4. C'est le protocole qui met sur le réseau les données et qui les en récupèrent. (ex : TCP, UDP)

Définition 25.10 (Couche application): C'est la manière de choisir quelle données on veut échanger et de quelle manière on les envoie. Par analogie, choisir un protocole applicatif, c'est comme choisir une langue dans laquelle communiquer.

III Est-ce tout internet ?

Vous avez peut-être entendu parler de protocole pair-à-pair, sans serveurs, et où les données s'échangent entre clients

Schéma 25.11:



En réalité, nous sommes encore dans le modèle client serveur. En effet, chaque client joue alternativement le rôle de client et de serveur.

Remarque 25.12: L'ancien modèle de communication téléphonique fonctionnait en connectant directement les deux hôtes via un canal qui leur était réservé.

2 La couche transport

I Introduction

Définition 25.13 (numéro de port): Les protocoles de transport définissent un numéro de port, c'est à dire un identifiant numérique associé à une application particulière. Il permet à une même machine d'établir plusieurs communication réseau en parallèle.

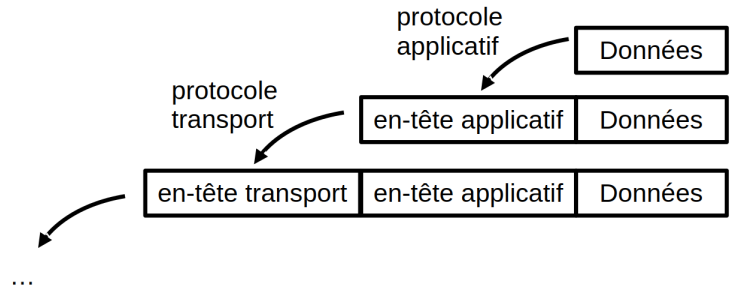
Exemple 25.14: Le numéro de port par défaut pour le web non sûr (protocole HTTP) est le 80, et celui pour le web sécurisé (HTTPS) est le 443. Ce ne sont cependant que des conventions, un serveur web peut attendre des connexions sur un autre port.

Principe 25.15 (Principe de Segmentation): Plutôt que d'envoyer toutes les données d'un seul tenant, la couche transport les découpe en plein de paquets plus petits qu'on envoie séparemment.

Intérêts 25.16:

- ⋆ Plusieurs paquets peuvent cohabiter sur un même lien (plus équitables)
- ⋆ En cas de paquet perdu ou corrompu, on peut ne renvoyer que le paquet en question (et pas tout le message)
- ⋆ On peut exploiter la redondance des liens du réseau en faisant emprunter plusieurs chemin à différents paquets

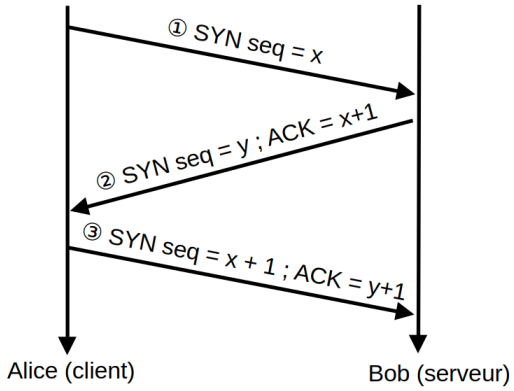
Principe 25.17 (Principe d'encapsulation): Chaque couche du réseau ajoute un en-tête aux données que l'on envoie, et considère les données d'un bloc (sans différencier les données d'éventuels en-tête de couches précédentes).



II Le protocole TCP

Principe 25.18: Le protocole TCP permet, après l'initialisation d'une connexion, d'envoyer des données de taille arbitraire, dans l'ordre, de détecter les erreurs de transmission et de retransmettre les fragments de données perdus ou corrompus.

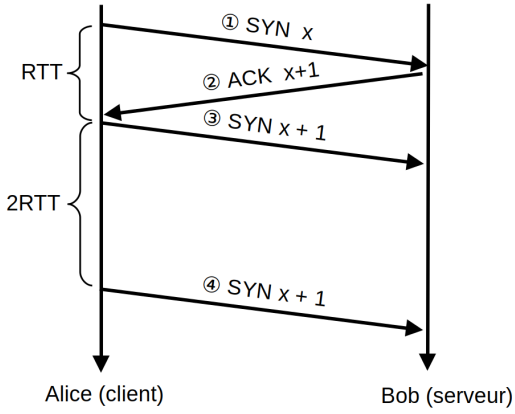
Étape 1 : Connexion en 3 temps



1. Le client choisit un numéro de séquence aléatoire et envoie un paquet SYN (pour synchronisation) indiquant ce numéro
2. Le serveur reçoit le paquet et choisit un autre numéro aléatoire et renvoie un paquet SYN - ACK (synchronized - acknowledgment) contenant le numéro de séquence du client incrémenté de 1 et son propre numéro de séquence.
3. Le client reçoit le paquet. Il est considéré comme connecté ! Il envoie «bien reçu» grâce au paquet ACK y+1

A la reception de ce paquet, le serveur est connecté

Étape 2 : Transfert de données et retransmission



1. Alice envoie un paquet de données identifié par un numéro de séquence.
2. Bob le reçoit et indique la bonne réception avec un paquet ACK
3. Alice n'a toujours pas reçu le ACK après 2 RTT = 2 fois le temps moyen d'un aller retour. Elle renvoie le paquet.

Remarque 25.19: Les numéros de séquences nous permettent d'envoyer le paquet suivants avant d'avoir reçu l'acquittement du précédent, tout en restant fiable (cf. principe 25.18) (car on sait alors les remettre dans l'ordre, et à quel paquet correspond un acquittement).

Développement : La fenêtre glissante dans le protocole TCP pour augmenter le taux d'envoi et éviter la congestion.

III Socket TCP en Python

Définition 25.20: Une socket est une interface logicielle, c'est à dire un objet qui représente le bout de la connexion entre deux machines.

Algorithme 25.21:

```python
# serveur TCP python
from socket import socket
serveur = socket() #creation de la socket
serveur.bind(('0.0.0.0', 9999)) #port 9999
serveur.listen()
while True :
    #sclient : socket du client, adclient : addresse IP du client
    (sclient, adclient) = serveur.accept()
    # On attend que le client lui envoie au moins 1000 octets de données
    donnees = sclient.recv(1000)
    while donnees :
        #traitement
        donnees = sclient.recv(1000)
    sclient.close()
```

```python
# client TCP python
from socket import socket
serveur = socket()
serveur.connect('127.0.0.1', 9999) #IP et port du serveur auquel se connecter
phrase = input()
while phrase != 'FIN':
    phrase = phrase + '\n'
    serveur.send(phrase.encode())
    phrase = input()
```

3 La couche application

Quand un client souhaite accéder à une page web, il tape un URL (ex : http ://wikipedia.fr/informatique) dans la barre du navigateur web. Cette URL se découpe en 3 morceaux :

— http ou https qui précise le protocole de la couche application utilisé

— le nom de domaine (ex : wikipedia.fr) ou l'adresse IP du serveur

— le document demandé au serveur (ex : la page informatique du serveur wikipedia)

I Le protocole HTTP (Hypertext transfer protocol)

Définition 25.22: Le protocole HTTP est un protocole applicatif qui définit les messages envoyés entre le navigateur (client) et le serveur. Les messages envoyés par le client sont des requêtes, ceux envoyés par le serveur des réponses.

Définition 25.23 (méthode): Les requêtes d'un client commence par un mot clef qui indique la méthode utilisée, c'est à dire la nature du message envoyé. Les méthodes les plus communes sont GET, HEAD et POST.

⋆ GET : Demande une ressource sans la modifier. C'est la méthode la plus courante pour demander d'afficher une page web par exemple

⋆ POST : transmet une donnée (ex : envoi de forumlaire)

⋆ HEAD : Cette méthode ne demande que des informations sur la ressource, sans demander la ressource elle-même

Exemple 25.24 (navigation vers l'URL http ://wikipedia.fr/informatique): Le client envoie la requête

```
GET /informatique HTTP/1.1
Host : www.wikipedia.fr
```

qui anonnce au serveur (wikipedia) que l'on utilise HTTP en version 1.1 pour obtenir la ressource informatique de wikipedia.

Le serveur répond alors

```
HTTP/1.1 200 OK #accepte de communiquer et de transmettre la ressource
Serveur: wikipedia.fr
Date: 27.07.1214
Content-Type: text/html

<html class ="client...
```

et le reste de la page wikipedia visible à `view-source:https://fr.wikipedia.org/wiki/Informatique`

Si jamais la ressource demandée n'existe pas ou est indisponible, la réponse sera

```
HTTP/1.1 404 Not Found #ressource non trouvée
Server: wikipedia.fr
Date: 27.07.1214
Content-type:
```

suivi de la page html décrivant l'erreur

II HTTPS

Le protocole HTTP est clair : si une personne intercepte les paquets de données sur le réseau entre un client et un serveur, elle peut lire le contenu des messages échangés, par exemple des mots de passe confidentiels. Le protocole HTTPS fonctionne comme le protocole HTTP, mais en chiffrant les messages.

Développement : Le S du protocole HTTPS : authentification et chiffrement des messages.

Leçon 26

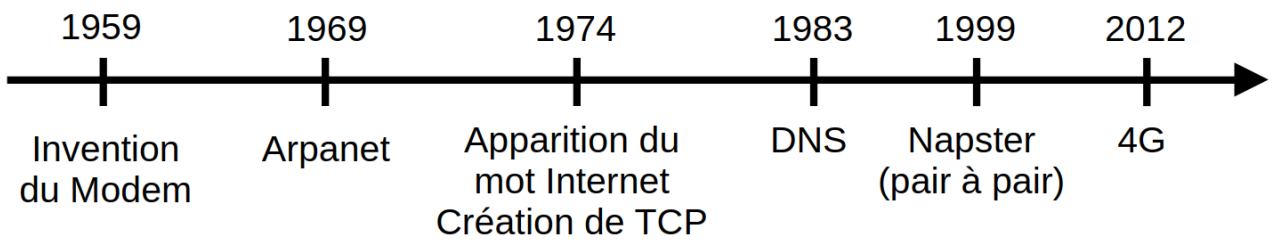
Architecture d'internet

Auteur·e·s: Emile Martinez
Niveau :
Pré-requis : Représentation binaire, graphes
Références : Tanenbaum, Balabonski 1ère, 2nd SNT Beaude Quenet

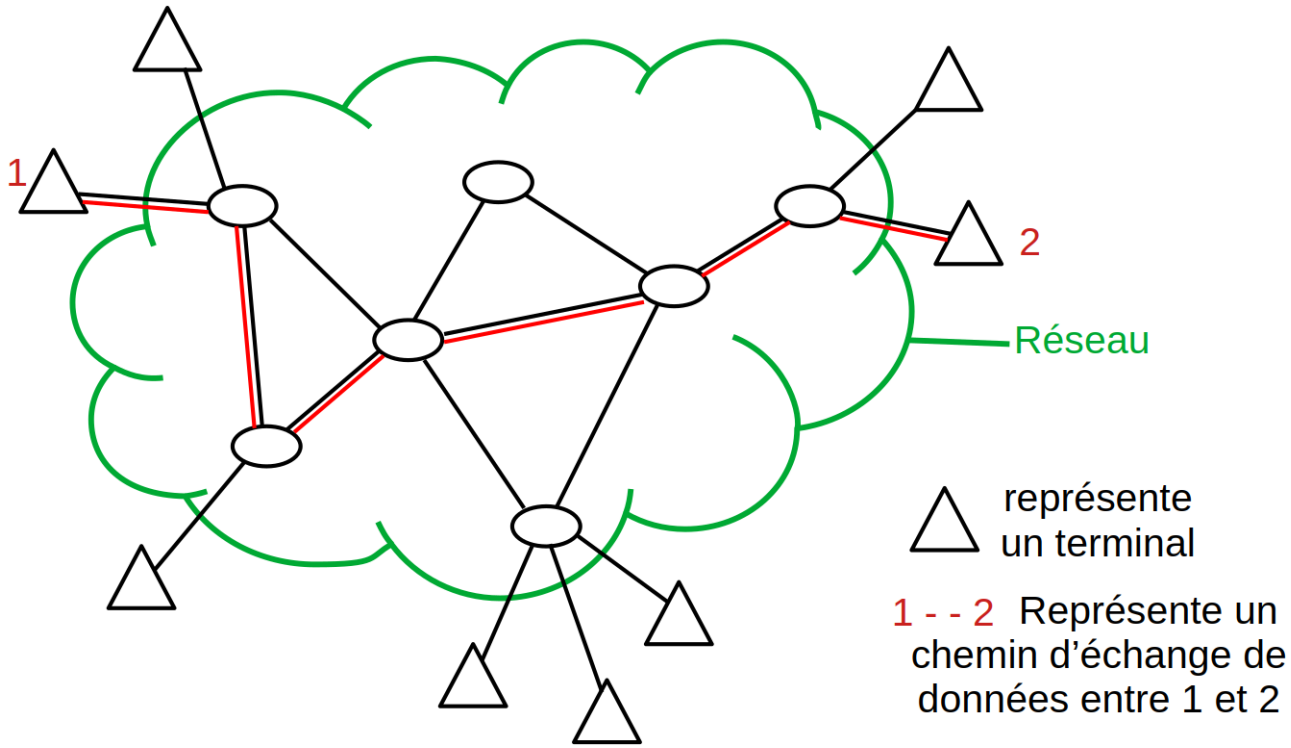
Commentaire 26.1 *On peut ici avoir une discussion sur le terme architecture. L'architecture c'est ≪Principe d'organisation d'un ensemble, agencement, structure≫. On peut donc penser à l'architecture physique. On parlerait alors des types de lien, de serveurs, de ou les placés, de qui controle quoi, de la mainère de faire des routeurs, etc. Néanmoins plus proches des notions du programme, on parlera plus ici d'architecture logicielle. Donc l'architecture conceptuelle, les différents protocoles qui s'empilent pour faire fonctionné cela.*

1 Mise en Contexte

I Historique



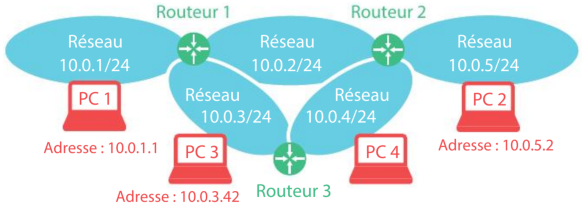
II Maintenant



Désormais, des milliards d'utilisateurs sont connectés à travers l'internet.

2 Modèles des couches OSI

Fais le lien entre les besoins d'une application et les messages transitant sur internet

Réseau	**Poste**
 (schém honteusement volé de ce livre https ://www.mesmanuels.fr/acces-libre/9782278094912)	
Couche Applicative	
Fais le lien entre les besoins d'une application et les messages transitant sur internet	Manière de lire et écrire une lettre. Langue dans laquelle on comunique
Couche Transport	
Centralise les données qu'envoient différentes applications d'une même machine et les mets sur le réseau. C'est le lien entre le PC et le réseau	poster le courrier, aller le chercher dans la boite aux lettres, et le repartir entre les différentes habitants
Couche Réseau	
S'occupe de faire passer les données d'un réseau à un autre. C'est là qu'on décide la direction que doit prendre la donnée. C'est ce qu'on appelle le routage.	Centre de tri (ou bureau de poste) mettant en sac postal et donnant le prochain lieu ou doit aller le sac postal
Couche Lien	
Achemine les données à l'intérieur d'un réseau	le kangoo jaune, le train entre deux gare de triage, le porteur qui amène le sac de lettre du quai au wagon postal, etc...

Remarque 26.1: La structure d'internet est assez indépendante de la technologie sous-jacente. Comme l'organisation de la poste ne dépend que à la marge du véhicule qu'utilise le facteur.

Conséquence 26.2: La technologie peut évoluer en conservant le travail des couches supérieures (et même plus généralement entre les couches)

Commentaire 26.2 *On parle plus de l'indépendance de la structure physique que des couches entre elles car le programme demande d'insister dessus.*

I Paquets

Principe 26.3 (Principe de segmentation): Plutôt que d'envoyer toutes les données d'un seul tenant, on les découpe en plein de paquets plus petits qu'on envoie séparément.

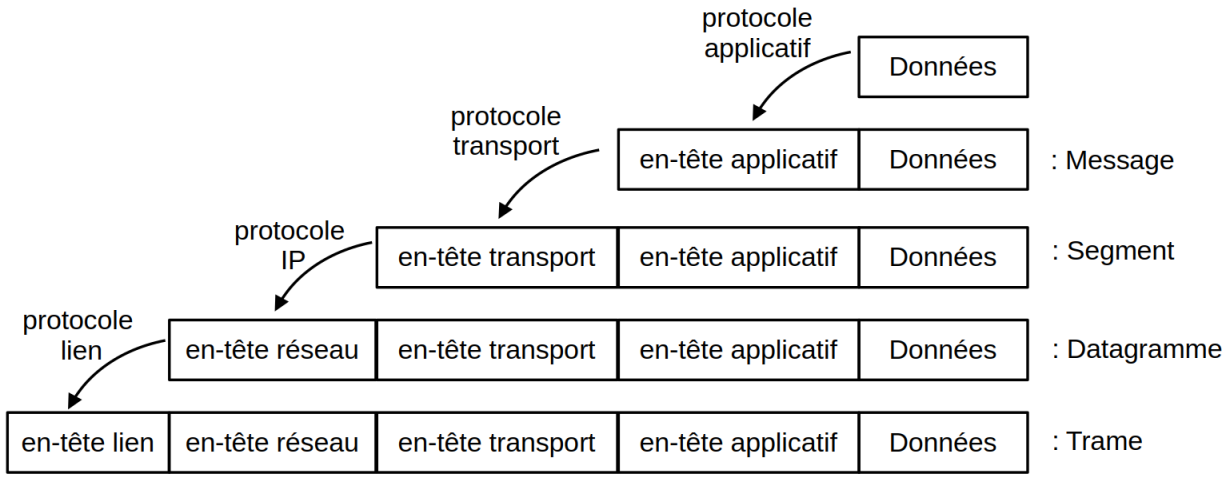
Intérêts 26.4:

— Plusieurs paquets peuvent cohabiter sur un même lien (il est facile de dire qu'on envoie un paquet de chaque application à tour de rôle par exemple)

— Si un paquet est corrompu (des données à l'intérieur sont fausses) ou se perd, on est pas obligé de tout renvoyer, on peut ne renvoyer que le morceau abîmée

— On peut maximiser la redondance des liens du réseau en faisant emprunter plusieurs chemin à différents paquets

Principe 26.5 (Principe d'encapsulation): Chaque protocole ajoute un en-tête à chaque paquet pour échanger de l'information, et considère l'en-tête d'avant comme de la données.



Commentaire 26.3 *Ce schéma pourrait être l'occasion de question avec les élèves, en expliquant plus longuement le principe, puis en leur demandant à leur avis à quelle couche le message est le plus grand (est ce que l'applicatif vient par dessus le transport ou l'inverse). Néanmoins il y a peu de chances qu'ils trouvent. On peut donc éventuellement plutôt le faire, et ensuite poser cette question en devoir.*

II Principe d'un protocole

Définition 26.6 (protocole): Un protocole est une manière de faire sur laquelle les différentes parties se mettent d'accord en amont.

Pour chaque couche et pour chaque chose que l'on veut faire dans cette couche, en réseau, on a des protocoles.

Exemple 26.7: Dans l'analgoie de la poste, la manière d'écrire une adresse sur une enveloppe est un protocole (que l'on pourrait placer à la couche réseau)

Exemple 26.8:

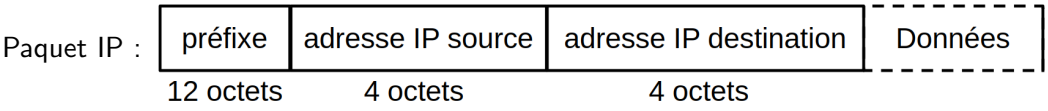
— NTP est un protocole applicatif permettant de syncrhoniser les horologes des ordinateurs
— FTP est un protocole applicatif permettant d'échanger des fichiers (comme des courriels)
— UDP est un protocole de la couche transport sans garantie utilisé pour transmettre des flux vidéos
— DHCP est un protocole de la couche réseau permettant à un nouvel utilisateur d'obtenir une adresse IP
— Ethernet est un protocole de la couche liaison décrivant comment se transmettre des paquets dans un câble coaxial

3 Des protocoles structurants

I Le protocole IPv4

C'est un protocole de la couche réseau ayant pour but d'identifier les différents hôtes d'un réseau. Chaque hôte est identifié par un nombre de 32 bits, découpé en 4 nombres de 8 bits :

`11000000101010000000101000101101` → `11000000.10101000.00010100.00101101` → `192.164.10.45`



Les routeurs (interfaces entre différents réseaux) garde alors une table de routage dans laquelle à chaque adresse IP est indiqué l'adresse IP du prochain routeur où aller.

Dveloppement : Convergence de l'algorithme de Bellman-Ford

Problème 26.9: Les tables sont beaucoup trop grandes

Solution 26.10: Deux machines proches recevront des IPs proches et ainsi on regroupe des machines en sous-réseau, dont les premiers bits des IPs sont les mêmes. Le nombre de bits identiques d'un sous-réseau est indiqué par un nombre après le /x (appelés masque de sous-réseau). Une table de routage n'a alors plus qu'a sauvegarder les sous-réseaux (sauf si il est dans le sous-réseau, où il garderait alors plus)

Exemple 26.11: `11000000.11000000.00001010.00101101/20` représente toutes les IP commençant par `11000000.11000000.0000`

Principe 26.12: Il arrive que l'on attribue des adresses d'un sous réseau ailleurs. On utilisera alors le principe de correspondance du plus grand préfixe (on va dans le sous réseau valide, où le plus de bits correspondent)

Remarque 26.13: Dans le champ préfixe on a des informations comme la version du protocole, un code de vérification (pour vérifier que le paquet n'est pas corrompu) ou encore le TTL (time to leave) indiquant le nombre de sauts restants autorisés pour ce paquets (lui évitant de boucler à l'infini).

II Le protocole TCP

Le protocole TCP est un protocole de la couche transport, décidant donc de qui va être envoyé, ré-envoyé, et quand.

Commentaire 26.4 *Si on veut s'étendre sur TCP, on peut reprendre ce qui est fait dans la leçon 25, et même éventuellement en récupérer le développement. Néanmoins ça parait moins structurant pour l'architecture globale d'internet.*

Principe 26.14:

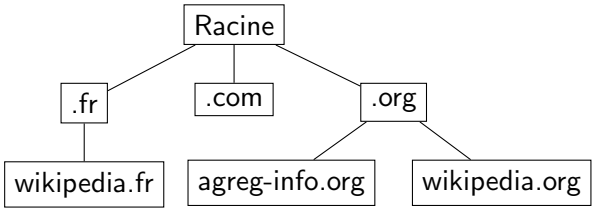
— Offrir des garanties sur les paquets : Pour cela il établit une connexion entre les deux hôtes (par une connexion en trois temps) puis envoie des acquittements pour chaque paquet (pouvant ainsi renvoyer ceux qui ne sont pas arrivés)

— Réguler le trafic : Pour cela, en fonction de la quantité de paquets perdus ou arrivant en retard, le protocole TCP adapte le taux d'envoi.

Commentaire 26.5 *On pourrait également dans le premier point rajouter le fait que on peut réordonner les paquets*

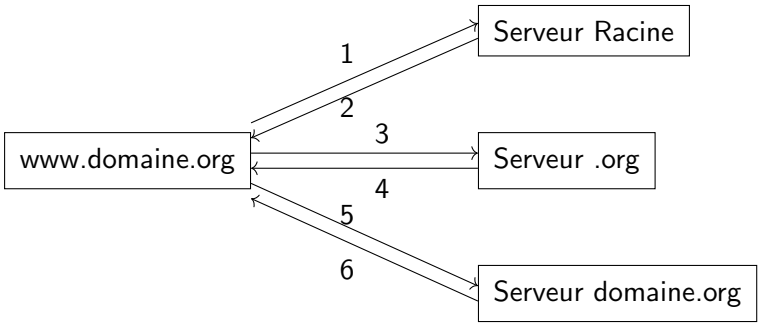
III Le protocole DNS

C'est un protocole de la couche application, parfois appelé ≪l'annuaire d'internet≫, qui fait correspondre des adresses IP à des URL lisibles par l'homme. Pour cela, des serveurs répondent l'adresse IP demandée.



Pour transformer un URL en adresse IP, on demande alors à chaque serveur l'adresse du serveur suivant.

Schéma 26.15 (**Résolution d'une requête DNS**):



IV HTTP

C'est un protocole omniprésent de la couche application. Il sert à échanger des pages web.

Son en-tête contient des informations comme le type de pages échangés, l'adresse correspondates, la date, etc. Il contient également un champ méthode précisant quel type d'opération l'on veut faire :

— GET pour obtenir une page WEB

— POST pour envoyer des données au serveur

— PUT qui modifie ce que contient le serveur

— pas de méthode coté serveur mais un code de réussite :

200 réussite de la réqueté
404 la page n'existe pas

Commentaire 26.6 *En vrai on montrerait une requête HTTP pour que l'on comprenne là c'est peu clair. On verrait alors en toute lettre les informations, puis le script HTML de la page WEB qui se déploie.*

Développement : Le S de HTTPS