Métaplan des leçons informatiques

Julie Parreaux

2018 - 2019

Table des matières

Leçon 901 : Structure de données. Exemples et applications.

Références pour la leçon

[1] Beauquier, Berstel et Chrétienne, *Éléments d'algorithmique*
[4] Cormen, *Algorithmique*

Développements de la leçon

Les B-arbres
La structure de donnée d'Union-Find

Motivation

Défense

Les structures de données jouent un rôle important dès que on souhaite stoker et manipuler (efficacement) des ensembles de données (qui sont généralement dynamique). La notion de type abstrait apparaît alors pour s'abstraire du langage de programmation. Elle intervient dans la conception des algorithmes qui est longue. On fait des raffinements successifs : la première version se fait indépendamment des structures de données ; la dernière version implémente dans un langage de programmation cette structure de données.

L'utilisation de cette structure abstraite se fait de deux manières différentes : la méthode ascendante qui part du langage et donne une structure abstraire ou la méthode descendante qui part de la structure abstraite et qu'on adapte au langage.

Le choix d'une structure de données est importante. Elle doit s'adapter au besoins et à la "sémantique" de l'algorithme. De plus, en fonction du problème (donc de l'algorithme), certaines structures de données sont plus efficaces que d'autre. On peut alors optimiser la complexité de nos algorithmes.

Remarque importante : Cette leçon n'est pas évidente : on peut très vite tombé dans un catalogue de structure de données sans aucune logique et le rapport du jury n'est pas nécessairement d'une grande aide. Mettre en avant les choix d'implémentations tout au long de la leçon et ne pas oublier de présenter des algorithmes utilisant ces structures. Cette leçon est fourre tout : toutes les notions d'algorithmique au programme (ou presque) peuvent se caser ici. Il y a alors un grand nombre de développements que l'on peut casé ici (on ne les a pas indiqué sur le plan...), il est donc important de soigné son plan.

Cette leçon se prête tout particulièrement aux dessins. Des dessins illustrant les structures de données classiques et leurs opérations pas nécessairement triviale peut s'avérer être intéressant (pensé à prendre le temps de les faire).

Ce qu'en dit le jury

Le mot algorithme ne figure pas dans l'intitulé de cette leçon, même si l'utilisation des structures de données est évidemment fortement liée à des questions algorithmiques. La leçon doit donc être orientée plutôt sur la question du choix d'une structure de données. Le jury attend du candidat qu'il présente différents types abstraits de structures de données en donnant quelques exemples de leur usage avant de s'intéresser au choix de la structure concrète. Le candidat ne peut se limiter à des structures linéaires simples comme des tableaux ou des listes, mais doit présenter également quelques structures plus complexes, reposant par exemple sur des implantations à l'aide d'arbres. Les notions de complexité des opérations usuelles sur la structure de données sont bien sûr essentielles dans cette leçon.

Métaplan

Tout au long de la leçon nous allons faire un va et vient entre la notion de type de donné abstrait et structure de données. Nous étudions notamment quelques types concrets pour les types abstraits que l'on présente.

- ⇝ *Objectif* : Comment choisir une bonne structure de données ? (phrase d'accroche)
- ⇝ *Motivation* : Stoker et manipuler (efficacement) un ensemble de données (dynamiques).
- ⇝ *Définition* [1, p.37] : Type abstrait de données
- ⇝ *Définition* [1, p.37] : Type concret
- ⇝ *Définition* [1, p.37] : Structure de données

I. Un type abstrait général (le type abstrait de base) Nous commençons par donner un type abstrait de base correspondant à la structure "optimale" que nous voulons construire. Elle contient toutes les opérations de base que nous souhaitons réaliser sur une structure de données afin de manipuler ces données. Nous allons ensuite données deux structures de données pour ce type abstrait. Dans la suite de la leçon nous réduisons le nombre d'opérations dans les structures de données afin de les spécialiser.

A. *Les opérations souhaitées* On présente dans cette section des structures de données implémentant ce type abstrait. Dans la suite de la leçon nous allons également nous intéresser des structures de données qui implémentent qu'une partie de ces fonctions pour répondre à des besoins précis.

Une liste d'opérations : create, union, find, insert, remove, find-earliest, find-min

B. *La table dynamique : une structure de données contiguë*

- ⇝ Type concret
- ⇝ Implémentation

C. *La liste (l'apparition des pointeurs)*

- ⇝ Type concret
- ⇝ Implémentation
- ⇝ *Théorème* : Comparaisons de la complexité des deux structures de données.

II. Les structures de données d'ordonnancement Les structures de données d'ordonnancement nous donne des structures qui nous permettent de ranger les données selon un certains ordre : de les trier.

A. *La pile et la file : des structures séquentielles*

- ⇝ Type abstrait
- ⇝ *Application* [1, p.102] : Parcours en profondeur de graphe
- ⇝ Type concret
- ⇝ Implémentation
- ⇝ *Proposition* : Complexité des opérations

B. *La file de priorité : une généralisation [4, p.140]*

- ⇝ Type abstrait
- ⇝ *Remarque* : Analogue en min
- ⇝ Implémentation
- ⇝ *Application* [4, p.600] : Algorithme de Dijkstra
- ⇝ Type concret : tas binaire (définition)
- ⇝ *Proposition* : Comparaison des complexités en fonction types concrets

C. *Le graphe : des relations d'ordres partiels*

- ⇝ Type abstrait
- ⇝ Types concrets :
- ⇝ *Application* [4, p.567] : Tri topologique
- ⇝ *Remarque* : Ordre partiel

III. Les structures de recherche : dictionnaires

Les structures de recherche et en particulier le dictionnaire sont des structures facilitant la recherche dans un ensemble de données. Pour cela, elles vont les trier de manière à faciliter la recherche. On cherche à optimiser la fonction de recherche et les fonctions insertion, suppression ne sont pas forcément optimiser.

Type abstrait : **dictionnaire** : insert, delete, find

A. *Les types concrets arborescents [4, p.267]*

- ⇝ *Définition* : Arbre de recherche
- ⇝ Implémentation
- ⇝ Type concret [4, p.368] : Arbres optimaux
- ⇝ *Proposition* : Complexité des opérations
- ⇝ Type concret [4, p.310] : AVL

B. *Un type concret séquentiel : la table de hachage [4, p.235]* Les problématiques autours du hachage parfait ne sont pas nécessaire dans cette leçon même si elles ne sont pas hors sujet. Maintenant, c'est une notion à garder en tête dès que l'on parle de hachage (même si le terme n'apparaît pas dans la leçon).

- ⇝ *Définition* : Table de hachage
- ⇝ Type concret
- ⇝ *Définition* : Collision
- ⇝ Implémentation

C. *Stoker et manipuler des données en grands nombres : la base de données relationnelle* La gestion des données sur un disque externe demande la minimisation du nombre d'accès au disque car cet accès est vraiment très lent (facteur d'un million). Dans cette leçon, il n'est pas nécessaire de parler d'arbre B+ mais il peut être intéressant de les garder en tête. Ils viennent généraliser les B-arbre dans le sens où ils permettent d'effectuer les opérations de recherche d'intervalles efficacement. Pour cela, on les construit comme les B-arbres sauf qu'on envoie une copie des clés présentes dans l'arbre dans ses feuilles.

- ⇝ *Objectif* : Manipuler des données sur un disque externe
- ⇝ Type abstrait : create, find, union
- ⇝ *Application* : Recherche sur des données stockées sur un disque externe
- ⇝ Type concret : B-arbre (Définition)
- ⇝ *Proposition* : Hauteur d'un B-arbre
- ⇝ *Application* : Complexité de la recherche et de l'insertion

IV. Partitionner un ensemble : Union-Find [4, p.519]

La structure d'union-find est une structure permettant de représenter un ensemble de données à l'aide d'une partition. C'est une des rares structures de données qui cherche à optimiser l'union de deux structures et la recherche. Notons que la structure de donnée générale présentée en début de leçon met des gyrophares sur cette structure de données.

- ⇝ *Objectif* : Trouver des éléments et unir des structures efficacement
- ⇝ Type abstrait : create, find, union
- ⇝ Type concret : Liste chaînée et union par rang
- ⇝ Type concret : Forêts et union par rang, compression de chemin
- ⇝ *Hypothèse* : Pointeurs sur les données
- ⇝ *Proposition* : Complexité pour la forêt
- ⇝ *Application* : Algorithme de Kruskal
- ⇝ *Proposition* : Complexité pour la liste chaînée

Leçon 903 : Exemples d'algorithmes de tri. Correction et complexité.

Références pour la leçon

[1] Beauquier, Berstel et Chrétienne, *Éléments d'algorithmique*
[4] Cormen, *Algorithmique*
[9] Froidevaux, Gaudel et Soria, *Types de données et algorithmes*

Développements de la leçon

Le tri par tas | Le tri topologique

Motivation

Défense

Le problème de tri est considéré par beaucoup comme un problème fondamental en informatique. Mais pourquoi trier ?

— Le problème de tri peut être inhérent à l'application : on cherche très souvent à classer des objets suivant leurs clés, comme lors de l'établissement des relevés bancaires.

— Le tri est donc souvent utilisé en pré-traitement dans de nombreux domaines de l'algorithmique : la marche de Jarvis (enveloppe convexe), l'algorithme du peintre (rendu graphique : quel objet je dois afficher en dernier pour ne pas l'effacer par d'autres ?), la recherche dans un tableau (dichotomie), l'algorithme de Kruskal (arbre couvrant minimal) ou encore l'implémentation de file de priorité.

— Le tri a un intérêt historique : de nombreuses techniques ont été élaborées pour optimiser le tri.

— Le problème de tri est un problème pour lequel on est capable de trouver un minorant (en terme de complexité).

— L'implémentation d'algorithme de tri fait apparaître de nombreux problèmes techniques que l'on cherche à résoudre via l'algorithmique.

Dans cette leçon, nous effectuons deux hypothèses importantes : les éléments à trier tiennent uniquement en mémoire vive et l'ordre sur ces éléments (sous lequel on tri) est total.

Ce qu'en dit le jury

Sur un thème aussi classique, le jury attend des candidats la plus grande précision et la plus grande rigueur.

Ainsi, sur l'exemple du tri rapide, il est attendu du candidat qu'il sache décrire avec soin l'algorithme de partition et en prouver la correction en exhibant un invariant adapté. L'évaluation des complexités dans le cas le pire et en moyenne devra être menée avec rigueur : si on utilise le langage des probabilités, il importe que le candidat sache sur quel espace probabilisé il travaille.

On attend également du candidat qu'il évoque la question du tri en place, des tris stables, des tris externes ainsi que la représentation en machine des collections triées.

Métaplan

I. Le problème de tri Les objets que l'on souhaite triés sont triés selon une clé (données satellites ou non)

- ⇝ *Problème* [1, p.122] : Problème du tri
- ⇝ *Hypothèses* : tri interne (tout est en mémoire vive) et ordre total
- ⇝ *Définition* [9, p.307] : tri stable
- ⇝ *Exemples* de tri dont un stable et l'autre non
- ⇝ *Définition* [4, p.136] : tri en place
- ⇝ Critère de comparaison des algorithmes de tri

II. Les tris par comparaison Ici la complexité de nos algorithmes peuvent être calculer en nombre de comparaisons effectuées puisque nos algorithmes de tris ne réalisent uniquement des comparaisons pour établir l'ordre. On s'autorise alors uniquement cinq tests sur les données à l'entrée : $=, <, >, \leq, \geq$. Il existe une borne minimale sur la complexité de ces algorithmes : on peut faire un premier classement des tris : ceux qui sont optimaux en moyenne, au pire cas ou pas du tout.

- ⇝ *Définition* [4, p.178] : Tri par comparaison
- ⇝ *Théorème* [4, p.179] : Borne optimale

Tri	Pire cas	En moyenne	Spatiale	Stable	En place
Sélection	$O(n^2)$	$O(n^2)$	$O(1)$	☑	☑
Insertion	$O(n^2)$	$O(n^2)$	$O(1)$	☑	☑
Fusion	$O(n \log n)$	$O(n \log n)$	$O(n)$	☒	☒
Tas	$O(n \log n)$	–	$O(1)$	☒	☑
Rapide	$O(n^2)$	$O(n \log n)$	$O(1)$	☒	☑
Dénombrement	$O(k+n)$	$O(k+n)$	$O(k)$	☑	☒
Base	$O(d(k+n))$	$O(d(k+n))$	$O(dk)$	☑	☒
Paquets	$O(n^2)$	$O(n)$	$O(n)$	☒	☒

A. *Les tris naïfs [9, p.310]* On commence par étudier les tris naïfs : ceux qui ne mettent en place aucun paradigme de programmation ni structures de données élaborées.

Tri par sélection

- ⇝ *Principe* : on cherche le minimum des éléments non triés et on le place à la suite des éléments triés
- ⇝ Algorithmes classique et tri à bulle
- ⇝ *Complexité* : $O(n^2)$
- ⇝ *Propriétés* : stable et en place

Tri par insertion (Le tri par insertion est aussi appeler la méthode du joueur de carte. Java implémente ce tri pour des tableau de taille inférieure ou égale à 7.)

- ⇝ *Principe* : On insère un à un les éléments parmi ceux déjà trié.
- ⇝ *Algorithme*
- ⇝ *Complexité* : $O(n^2)$
- ⇝ *Propriétés* : stable et en place
- ⇝ *Remarques* : très efficace sur des petits tableau ou sur des tableau presque trié.

B. *Diviser pour régner* On présente maintenant des algorithmes de tri basé sur le paradigme diviser pour régner : le premier découpe simplement le tableau de départ (tri fusion) tandis que le second combine facilement les deux sous tableau triés (tri rapide). Il existe des tris mixtes qui en fonctions de l'ensemble des données (taille, caractère trié) que l'on présente choisi l'un ou l'autre des algorithmes de tri.

Tri fusion [4, p.30]

- ⇝ *Principe* : Séparation facile, recollement avec fusion
- ⇝ *Algorithme*
- ⇝ *Complexité* (pire cas et moyenne) : $O(n \log n)$
- ⇝ *Application* : Calcul de jointure dans le cadre des bases de données

Tri rapide [4, p.157] Contre-exemple au Master theorem.

- ⇝ *Principe* : Séparation avec pivot, recollement facile
- ⇝ *Algorithme*
- ⇝ *Complexité* dans le pire cas : $O(n^2)$
- ⇝ *Complexité* en moyenne $O(n \log n)$
- ⇝ *Propriété* : en place ; le plus rapide en pratique

C. *Structure de données [4, p.140]* C'est un algorithme de tri qui se base sur les propriétés d'une structure de données bien précise : le tas. Elle permet de gérer les données. Le terme tas qui fut d'abord inventé dans ce contexte est aujourd'hui également utilisé dans un contexte d'exécution d'un programme : c'est une partie de la mémoire qu'utilise un programme. Un programme lors qu'il s'exécute utilise une mémoire de pile (qui donne les instructions suivantes à faire) et un tas (qui contient les valeurs des variables, de la mémoire auxiliaire pour faire tourner le programme). Ces deux mémoires grandissent l'une vers l'autre (l'erreur de *stack overflow* arrive quand la pile rencontre le tas). Le tri par tas est un tri en place et de complexité $O(n \log n)$ en moyenne. C'est donc un des meilleurs tri par comparaison que l'on possède.

- ⇝ *Définition* : un tas
- ⇝ *Principe* : Construction d'un tas avec les entrées et extraction du maximum.
- ⇝ Algorithme, correction et complexité
- ⇝ *Propriétés* : en place
- ⇝ *Application* : file de priorité

III. Les tris linéaires

On fait des hypothèses sur l'entrée de nos algorithmes afin d'obtenir un tri linéaire. De plus, ces algorithmes fond appel à d'autres opérations que les comparaisons : on fait abstraction de la borne de minimalité.

Tri par dénombrement [4, p.180]

- ⇝ *Hypothèse* : Valeurs de l'entrées $\in [\![0, k]\!]$ avec k fixé.
- ⇝ *Principe* : Combien sont inférieur à x.
- ⇝ *Méthode* : Utilisation d'un tableau
- ⇝ Algorithme et exemple
- ⇝ *Propriétés* : tri stable (Attention si valeurs égales)
- ⇝ *Complexité* : $O(n+k)$ (Linéaire : si $k = O(n)$)

Tri par paquets [4, p.185]

- ⇝ *Hypothèse* : Clés uniformément distribuées sur $[0, 1[$.
- ⇝ *Principe* : Divise $[0, 1[$ en n intervalles de même taille et tri par insertion
- ⇝ Algorithme
- ⇝ *Complexité espérée* : $O(n)$

Tri par base [4, p.182]

- ⇝ *Hypothèse* : d sous-clés entières bornées muni d'un poids
- ⇝ *Principe* : Application d'un tri par dénombrement (stable) sur les sous-clés
- ⇝ *Application* : Trier des cartes perforées
- ⇝ *Application* : Trier des dates
- ⇝ *Application* : trier des chiffres
- ⇝ *Propriétés* : tri stable ; pas en place
- ⇝ *Complexité* : $O(n)$

IV. Affaiblissement des hypothèses pour le tri

Hypothèse : on n'utilise maintenant plus qu'un ordre partiel : on utilise l'ordre donné par un graphe.

A. *Tri topologique [4, p.565]* On commence par étudier les tris naïfs : ceux qui ne mettent en place aucun paradigme de programmation ni structures de données élaborées.

- ⇝ *Problème* : Problème du tri topologique
- ⇝ *Remarque* : ordre partiel induit par le graphe
- ⇝ Algorithme du tri topologique et correction

B. *Tri externe* On s'autorise des données qui ne tiennent pas en mémoire vive.

Ouverture Tri de shell ; Réseau de tri

Leçon 907 : Algorithmique du texte. Exemples et applications.

Références pour la leçon

[2] Carton, *Langages formels, calculabilité et complexité*
[4] Cormen, *Algorithmique*
[5] Crochemore, Hancart et Lecroq, *Algorithmique du text*
[6] Crochemore et Rytter, *Text algorithms*

Développements de la leçon

L'automate minimal de la recherche de motif
Le calcul de l'alignement optimal

Motivation

Défense

Le texte est omniprésent en informatique. Des fichiers textes permette d'écrire des programmes informatique : on a dû développer des algorithmes afin de concevoir des logiciel capable de gerer du texte, mais égalent des logiciel capable de compiler nos programme qui sont sous la forme d'un texte. L'algorithmique du texte est ensuite apparue dans la bio-informatique où on a cherché à traiter les chaînes de l'ADN ou lors de traitement multimédia lorsque l'on recherche à compresser toutes ces données ou encore en sécurité avec la protection des mots de passes.

On a une algorithmique complexe et très variées qui touche à toutes sorte de problèmes.

Ce qu'en dit le jury

Cette leçon devrait permettre au candidat de présenter une grande variété d'algorithmes et de paradigmes de programmation, et ne devrait pas se limiter au seul problème de la recherche d'un motif dans un texte, surtout si le candidat ne sait présenter que la méthode naïve.

De même, des structures de données plus riches que les tableaux de caractères peuvent montrer leur utilité dans certains algorithmes, qu'il s'agisse d'automates ou d'arbres par exemple.

Cependant, cette leçon ne doit pas être confondue avec la 909, « Langages rationnels et Automates finis. Exemples et applications. ». La compression de texte peut faire partie de cette

Algorithme	Prétraitement	Recherche	Espace
Naïf	0	$O(tp)$	$O(1)$
Rabin–Karp	$\Theta(p)$	$O(tp)$	$O(1)$
Boyer-Moore	0	$O(tp)$	$O(1)$
Automate	$O(m\|\Sigma\|)$	$O(t)$	$O(m\|\Sigma\|)$
Morris–Pratt	$O(p\|\Sigma\|)$	$\Theta(t)$	$O(p)$
Knuth–Moris–Pratt	$\Theta(p)$	$\Theta(t)$	$O(p)$
Arbre	$O(t)$	$O(m)$	$O(n \log n)$

TABLE 1 – Récapitulatif des complexités pour les algorithmes de recherche de motifs.

leçon si les algorithmes présentés contiennent effectivement des opérations comme les comparaisons de chaînes : la compression LZW, par exemple, est plus pertinente dans cette leçon que la compression de Huffman.

Métaplan

- ⇝ *Motivation* : Compilation, traitement de texte, stockage des données
- ⇝ *Définition - Notation* : alphabet + mot + préfixe

I. Recherche de motif Lorsqu'on a un texte, on souhaite pouvoir retrouver rapidement un mot (ou un motif) dans celui-ci.

- ⇝ *Notations* : texte T et motifs P
- ⇝ *Remarque préliminaire* : si P est un motif dans T alors $|P| \leq |T|$.
- ⇝ *Applications* : fouille de données ; contrôle F sur un ordinateur ; ...

A. *Recherche d'un motif exact* On commence par recherche le motif exacte (on suppose qu'il n'y a pas d'erreur de frappe, d'orthographe dans le motif ou dans le texte). On va étudier toutes sorte d'algorithme étant plus ou moins efficace pour répondre à cette question. LPour les plus téméraires et s'il reste de la place, on peut évoquer la recherche par un arbre à suffixe [5, p.162] qui est une recherche basées sur un autre type de données.

L'algorithme naïf [4, p.905]

- ⇝ *Principe* : on teste tous les caractère jusqu'à ce qu'on trouve (fenêtre glissante)
- ⇝ Algorithme + Complexité en temps et en espace
- ⇝ On peut et on veut faire mieux

L'algorithme Boyer–Moore L'idée de cet algorithme est original : la lecture du motif en sens inverse peut conduire à une amélioration des performances.

- ⇝ *Principe* : on lit le motif dans l'autre sens
- ⇝ Complexité en temps et en espace

Utilisation d'un automate des occurrences [4, p.915], [5, p.182]

- — *Principe* : Trouver P dans T c'est trouver tous les T_i ayant P pour suffixe : $\{i : T_i \in \Sigma^* P\}$. On construit donc un automate des occurrences (exemple)
- — *Notations* : $\phi(T_i)$ l'état après la lecture de T_i et $\sigma(T_i)$ le plus long suffixe de P qui est préfixe de T_i
- — *Théorème* : $\sigma(T_i) = \phi(T_i)$ L'automate des occurrences est minimal pour reconnaître $\Sigma^* P$.
- — *Extension* : reconnaître plusieurs mots (automate en parallèle)

Utilisation de la notion de bords

- — *Définition* : bord + exemple [5, p.11]
- — *Principe* : on peut calculer d'où recommencer la recherche avec le bord du motif (on fait moins de comparaison, on est donc plus rapide)
- — *Algorithmes* : MP et KMP (qui améliore ce procédé) + complexités (inconvénient de la méthode)

B. *Cas d'une expression régulière (motif incomplet)* Lorsqu'on a un texte, on souhaite pouvoir retrouver rapidement un mot (ou un motif) dans celui-ci.

- ⇝ *Problème* : P peut être considéré comme une expression régulière
- ⇝ *Exemple* *.ml dans une liste de fichiers
- ⇝ *Application* : grep
- ⇝ *Application* : Analyse lexicale à l'aide de l'automate des motifs constitué d'expressions régulières

⇝ *Principe* : Construire l'automate fini qui la reconnaît, le déterminiser et l'appliquer.

⇝ *Complexité* : Exponentielle dans le pire cas (en pratique cela marche raisonnablement)

II. Comparaison de textes

Applications : correcteurs orthographiques, bio-informatique (étude du génome), traitement du signal, analyse de fichier (détection de virus), ...

A. *Plus longue sous-séquence commune [4, p.362]* L'algorithme de la plus longue sous-séquence commune est intéressant car il repose sur le principe de la programmation dynamique.

⇝ *Problème* : Spécification du problème et exemple ⇝ *Algorithme* : Algorithme et complexités

B. *Distance d'édition et recherche approchée [5, p.224]* La distance d'édition permet de comparer deux textes à priori distincts et de classer à quels points les deux textes sont distincts. Le calcul de telles distances peut-être délicat. On utilise alors des paradigmes issues de la programmation dynamique. Outre l'utilisation en bio-informatique (via les études sur l'ADN), les distances d'éditions sont également utilisées dans la recherche approchées de motifs.

⇝ *Définitions* : Fonctions de substitution, d'ajout et de suppression ; distance d'édition de Levenstein

⇝ *Théorème* : Caractérisation de la notion de distance

⇝ *Définition* : Alignement + exemple ⇝ *Définition* : Coût d'un alignement

⇝ *Algorithme* : Calcul de l'alignement optimal entre deux mots

⇝ *Application* : Recherche à k différences.

B. *Répétition dans un mot [5, p.211]*

⇝ *Problème* : Recherche de facteurs se répétant dans un mots

⇝ *Méthode* : automate des occurrences et recherche d'un état particulier.

III. Compression [6]

Lorsqu'on stocke un grand volume de données, on souhaite les compresser afin d'en réduire la place. La compression du texte est la première compression que nous avons réalisée : tout fichier informatique peut être vu comme du texte. Aujourd'hui chaque compression est spécifique au type des données que l'on stocke. On peut également parler de l'algorithme de LZH et de la notion de perte. De plus, à mon sens, il est important de garder en tête le fonctionnement du codage de Hoffmann même si le rapport du jury ne le mentionne pas explicitement car cette section ouvre des questions sur celui-ci.

⇝ *Remarque* : on ne peut pas compresser indéfiniment sans perte (théorème de Shannon)

⇝ Algorithme LZ + exemple + remarque sur son utilité actuelle

⇝ Algorithme de LZW

IV. Traitement d'un texte avec une grammaire [2]

Les informaticiens écrivent beaucoup de textes pour discuter avec une machine. Les langages de programmation sont les premiers à nécessité une transformation via un algorithme de texte.

⇝ Processus de compilation : analyse syntaxique

⇝ *Définition* : problème du mot + forme normale

⇝ *Théorème* : Décidabilité du problème du mot avec une grammaire algébrique sous forme FNC

⇝ *Algorithme* algorithme de CYK.

Ouverture

⇝ La compression fait appelle à de nombreux autres formats : ZIP ; Δ (git) ; ...

⇝ Recherche de motif incomplet ⇝ Le problème de l'encodage : UTF8 ; ASCII

Leçon 909 : Langages rationnels et automates finis. Exemples et applications.

Références pour la leçon

[1] Beauquier, Berstel et Chretienne, *Éléments d'algorithmique*
[2] Carton, *Langages formels, calculabilité et complexité*
[8] Floyd et Biegel, *Le langage des machines*
[22] Sakarovitch, *Éléments de la théorie des automates*
[26] Wolper, *Introduction à la calculabilité*

Développements de la leçon

Le problème PSA est NP-complet

L'automate minimal de la recherche de motif

Motivation

Défense

La théorie des langages reconnaissable apparaît dans les années 1940 avec une première modélisation de neurone grâce à des automates finis. Ensuite, dès 1956, la théorie des langages rationnels et reconnaissables sont uniformiser grâce aux théorème de Kleene (en 1956, on a également leur minimisation et en 1958, on a leur caractérisation). Il faudra attendre les années 1959 pour qu'un premier article posant les bases de la théorie des automates telles que nous la connaissons soit posé (cela à valu à leur auteur un prix Turing).

Les automates finis (qui sont les premières machines) ont été trouvé leurs essor dans les années 1980 pour commencer l'automatisation de certaine tâches comme l'analyse syntaxique lors de la compilation. Cependant leurs applications ne se limitent pas à l'informatique (avec la compilation ou le traitement de texte), ils apparaissent également dans la théorie linguiste en décrivant la morphologie d'une langue. En informatique théorique, ils sont très facile à manipuler. Les langages rationnels sont les premiers langages de la hiérarchie de Chomsky. Ils sont les langages les moins expressifs mais ils nous permettent de répondre à de nombreux problèmes de décisions. Il est difficile de trouver des problèmes de décision sur ces langages qui ne soit pas décidable.

Ce qu'en dit le jury

Pour cette leçon très classique, il importe de ne pas oublier de donner exemples et applications, ainsi que le demande l'intitulé.

Une approche algorithmique doit être privilégiée dans la présentation des résultats classiques (déterminisation, théorème de Kleene, etc.) qui pourra utilement être illustrée par des exemples. Le jury pourra naturellement poser des questions telles que : connaissez-vous un algorithme pour décider de l'égalité des langages reconnus par deux automates ? quelle est sa complexité ?

Des applications dans le domaine de l'analyse lexicale et de la compilation entrent naturellement dans le cadre de cette leçon.

Métaplan

— *Motivation* : Lien avec l'étude des langages (les comprendre) ; Automatiser des processus simples.

— *Cadre* : Σ alphabet fini ; L un langage sur cet alphabet.

I. Langages rationnels [2, p.38] On amène le monde de la syntaxe ici (on réintroduit un nouveau niveau). Les expressions rationnelles sont aux opérateurs rationnelles ce que sont les expressions arithmétiques pour leurs opérateurs.

- ⇝ *Définition* [2, p.17] : opérations rationnelles
- ⇝ *Exemple* : $\{a, b\}^*$.
- ⇝ *Définition* : langages rationnels
- ⇝ *Exemples* : (ne pas oublier le langage fini)
- ⇝ *Définition* : Expression rationnelles
- ⇝ *Remarque* : parenthésage et ambiguïté
- ⇝ *Proposition* : les langages rationnels sont ceux induits par les expressions rationnelles.

II. Langages reconnaissables [2, p.39]

A. *Automate fini* Les automates sont des machines de Turing particulière : sans mémoire, le ruban est en lecture seul en une seule passe.

- ⇝ *Définition* : automate fini
- ⇝ *Définition* : langage accepté
- ⇝ *Définition* : langage reconnaissable
- ⇝ *Application* : Décidabilité de Presburger

B. *Quelques automates particuliers*

- ⇝ *Définitions* [22, p.114] : Automate déterministe
- ⇝ *Définition* [22, p.74] : automate émondé
- ⇝ *Remarque* [1, p.299] : complexité de la déterminisation
- ⇝ *Exemple* : Automate des parties (optimal)
- ⇝ *Proposition* : PSA est NP-complet.
- ⇝ *Définitions* : Automate complet
- ⇝ *Proposition* [2, p.46] : Équivalence
- ⇝ *Définition* [8, p.555] : Problème PSA

C. *Équivalence de ces langages [2, p.40]*

- ⇝ *Théorème* : de Kleene
- ⇝ *Lemme* : d'Arden
- ⇝ *Preuve* : Automates vers expressions : McNaughton–Yamada ; Brzozowski–McCluskey ; élimination de Gauss. Expressions aux automates : Thomson
- ⇝ *Application* : recherche d'un motif par une expression régulière
- ⇝ *Application* : analyse lexicale

III. La classe des rationnels Cette famille de langage est très agréable à manipuler.On souhaite connaître quelques unes de ces propriétés. On souhaite également alors savoir quels langages appartiennent à cette famille.

A. *Stabilité* La classe des rationnels est très stable.

- ⇝ *Proposition* [22, p.66,p.97,p.118] : stabilité par intersection, passage au complémentaire, et opérations rationnelles
- ⇝ *Proposition* : stabilité par morphisme [2, p.58]
- ⇝ *Remarque* : L'inclusion n'est pas stable Σ^* est rationnel mais pas $a^n b^n$ ne l'est pas.

B. *Grammaire régulière (linéaire à droite) [26]* *Idée* : Les automates finis (langages rationnels) sont la première marche de la hiérarchie de Chomsky.

- ⇝ *Définition* : Grammaire / Langage engendré
- ⇝ *Définition* : Grammaire régulière
- ⇝ *Proposition* : Caractérisation des langages rationnels.

C. *Caractérisation* Mais qui y ait ?

- ⇝ *Lemmes* [22, p78] : de l'étoile
- ⇝ *Contre-exemple* : $a^n b^n$
- ⇝ *Théorème* [22, p.128] : Ehrenfeucht, Parikh, Rozenberg
- ⇝ *Lemme* : étoile par bloc
- ⇝ *Proposition* : caractérisation des langages reconnaissables

D. *Décidabilité et langages réguliers* *Idée* : il est très difficile de trouver des problèmes indécidables sur les langages rationnels.

- ⇝ *Problèmes décidables* : mot, vide, universalité, ...
- ⇝ *Remarque* : problème universelle est PSPACE-complet

IV. Minimisation [22, p.120] Existe-t-il un représentant canonique pour reconnaître un langage ?

A. *Morphismes d'automates* La relation d'ordre permettant de définir un automate minimal est formellement donnée par la notion de morphisme d'automate. Une bijection par morphisme d'automate définie un isomorphisme entre ces automates à numérotation près des états.

- ⇝ *Définition* [22, p.120] : Morphisme d'automate
- ⇝ *Proposition* : Morphisme surjectif
- ⇝ *Définitions* : relation d'ordre et minimisation

B. *Caractérisation de l'automate minimal* La notion de morphisme, même si elle donne un cadre formel à notre étude n'est pas facile à manipuler. Nous allons donc donner d'autres caractérisations de la minimalité d'un automate.

Automate des résiduels [1, p.312] Quelques fois appelé quotient : point de vu intrinsèque au langage.

- ⇝ *Définition* : automate des résidus
- ⇝ *Proposition* : caractérisation de l'automate minimal par les résidus

Équivalence de Nérode Autre point de vu : plus calculatoire.

- ⇝ *Définitions* : Équivalence de Nérode
- ⇝ *Définitions* : automate quotient
- ⇝ *Proposition* : équivalence des langages
- ⇝ *Définition* : congruence de Nérode
- ⇝ *Proposition* : caractérisation de l'automate minimale
- ⇝ *Exemple* : Automate des occurrences est l'automate minimal de $\Sigma^* P$
- ⇝ *Application* : recherche de motif dans un texte ((K)MP)

C. *Calcul de l'automate minimal* Comment calculer cet automate minimal ?

- ⇝ *Algorithme* [1, p.325] : Construction de Moore
- ⇝ *Algorithme* [1, p.325] : Algorithme de Hopcroft
- ⇝ Algorithme par renversement [22, p.125]
- ⇝ *Application* : Équivalence de langage
- ⇝ *Application* : Bi-simulation.

Ouverture Fermeture transitive

Leçon 912 : Fonctions récursives primitives et non primitives. Exemples.

Références pour la leçon

[1] Beauquier, Berstel et Chretienne, *Éléments d'algorithmique*
[2] Carton, *Langages formels, calculabilité et complexité*
[15] Lassaigne et Rougemont, *Logique et fondements de l'informatique. Logique du 1er ordre, calculabilité et λ-calcul.*
[26] Wolper, *Introduction à la calculabilité*

Développements de la leçon

Turing-calculable $\Rightarrow$ μ-récursive
Caractérisation des langages RE

Motivation

Défense

L'historique de ce modèle est moins claire que pour les machines de Turing. En effet, de nombreuses recherches ont été menée dans ce sens pour répondre au dixième problème de Hilbert (on est loin de l'idée novatrice de Turing). Godel est le premier à présenté les travaux sur les fonctions μ-récursive en 1934 (ils les appellent les fonctions récursives généralisées). Ces travaux se bases sur les fonctions primitives récursives (souvent associée à Kleene) dont les prémisses sont introduits dès les années 1920 et sur les travaux d'Ackermann en particulier qui pointent les limites de celles-ci. Le modèle apparaît donc juste avant les machines de Turing mais L'apparition des trois modèles de calculs classiques sont très proche car produit pendant les début de la recherche sur la théorie de la calculabilité.

Modélisation des fonctions récursives : les fonctions récursives sont un modèles proches des fonctions récursives classique (syntaxe proche du langages des fonctions récursives que l'on manipule). Le deuxième modèle (équivalent) qui modélise les fonctions récursives est le λ-calcul. Ce modèle se concentre sur la sémantique des fonctions récursives et sont à l'origine des sémantiques des langages de programmation purement fonctionnels (comme LISP) où tout est uniquement fonction.

Ce qu'en dit le jury

Il s'agit de présenter un modèle de calcul : les fonctions récursives. S'il est bien sûr important de faire le lien avec d'autres modèles de calcul, par exemple les machines de Turing, la leçon doit traiter des spécificités de l'approche. Le candidat doit motiver l'intérêt de ces classes de fonctions sur les entiers et pourra aborder la hiérarchie des fonctions récursives primitives. Enfin, la variété des exemples proposés sera appréciée.

Métaplan

- ⇝ *Pré-requis* : Autours des machines de Turing
- ⇝ *Définition* [15, p.115] : Un modèle de calcul.

I. Les fonctions primitives récursives Dans tout le plan on considère des fonctions d'arité quelconque mais qui dont à valeurs dans $\mathbb{N}$. Cependant dans le développement 1 par exemple nous avons besoin de fonction à valeur dans $\mathbb{N}^k$. Il faut donc soit préciser directement dans le plan qu'on peut facilement étendre cette notion (vue comme tuple de fonction) ou au le dire dans la défense de plan ou dans le développement.

A. *Syntaxe et sémantiques*

- ⇝ *Définition* : Syntaxe fonctions primitive récursive.
- ⇝ *Définition* : Sémantique fonctions primitive récursive.
- ⇝ *Définition* : Fonctions primitives récursives
- ⇝ *Exemples* [2, p.182] : Additions, multiplication
- ⇝ *Exemples* [26, p.123] : Flèche de Knuth, factorielle, Fibonacci
- ⇝ *Exemples* : Modulo, division

B. *Les prédicats [26, p.125]*

- ⇝ *Définitions* : Prédicats
- ⇝ *Exemples* : Paire, <
- ⇝ *Définition* : Fonction caractéristique
- ⇝ *Proposition* : Caractérisation des prédicats récursifs
- ⇝ *Exemples* : Égalité, zéro
- ⇝ *Application* : Fonctions à support finies
- ⇝ *Définition* [26, p.128] : Minimisation bornée
- ⇝ *Proposition* [26, p.128] : Minimisation d'une fonction primitive récursive

C. *Les limites des fonctions primitives récursives* Bel argument diagonal pour montrer que les fonctions primitives récursives ne permettent pas de délimiter l'ensemble des fonctions calculables. On donne ensuit un exemple d'une fonction calculable qui n'est pas primitive récursive. Attention, ici on n'a pas encore définit ce qu'est la calculabilité (elle vient avec la définition de la thèse de Church) donc la notion de fonction calculable par une machine de Turing est importante.

- ⇝ *Théorème* [26, p.129] : Existence d'une fonction calculable non primitive récursive
- ⇝ *Définition* [2, p.183] : Ackermann
- ⇝ *Proposition* [2, p.183] : Ackermann n'est pas primitive récursive
- ⇝ *Application* : Fonctions primitives et calculabilité
- ⇝ *Remarque* : Lien avec les langages de programmation

II. Les fonctions μ-récursive [26, p.131]

- ⇝ *Définition* : Minimisation non bornée
- ⇝ *Exemple* : Division par 2
- ⇝ *Définition* [26, p.136] : Fonction μ-récursive partielle
- ⇝ *Exemples* : Ackermann [2, p.183]
- ⇝ *Définition* : Fonction μ-récursive totale
- ⇝ *Définition* : Prédicats sûres
- ⇝ *Exemple* : Inverse de la fonction d'Ackermann
- ⇝ *Proposition* : Inverse d'une μ-récursive
- ⇝ *Remarque* : Lien avec les langages de programmation

III. Lien avec la calculabilité

A. *Lien avec la thèse de Church*

⇝ Thèse de Church pour les fonction μ-récursive

Étude des MT

⇝ *Théorème* : Fonctions μ-récursives $\Leftrightarrow$ Turing-calculable (DEV : $\Leftarrow$)

⇝ *Corollaire* : Fonction μ-récursive partielle $\Leftrightarrow$ Turing-calculable

⇝ *Application* : La thèse de Church pour les machines de Turing

Étude du λ-calcul [15, p.185]

⇝ *Définition* : Termes

⇝ *Exemple* : Entier de Church

⇝ *Exemple* : Addition par λ-calcul

⇝ *Théorème* : Fonctions μ-récursives $\Leftrightarrow$ λ-définissable

⇝ *Corollaire* : Fonctions Turing-calculable $\Leftrightarrow$ λ-définissable

B. *Les langages récursivement énumérables*

⇝ *Définition* : Langages R et RE

⇝ *Définition* : Langages décidables

⇝ *Remarque* : Lien entre langages et programmes

⇝ *Problème* : ARRÊT

⇝ *Théorème* : ARRÊT est indécidable et dans RE.

⇝ *Définition* : Réduction

⇝ *Théorème* : Utilisation des réductions

⇝ *Problème* : PAVAGE DE WANG

⇝ *Application* : PAVAGE DE WANG est indécidable

⇝ *Théorème* : Théorème de Rice

⇝ *Application* : Langage vide pour une MT

⇝ *Application* : Théorème de Rice pour fonctions μ-récursives

⇝ *Théorème* : Caractérisation des langages RE

Ouverture Hiérarchie de Chomsky

Leçon 913 : Machines de Turing. Applications.

Références pour la leçon

[15] Lassaigne et Rougemont, *Logique et fondements de l'informatique. Logique du 1er ordre, calculabilité et λ-calcul.*
[23] Sipser, *Introduction to the Theory of Computation.*
[24] Stern, *Fondement mathématiques de l'informatique.*
[25] Girard et Turing, *La machine de Turing*
[26] Wolper, *Introduction à la calculabilité*

Développements de la leçon

Turing-calculable $\Rightarrow$ μ-récursive — Théorème de Savitch

Motivation

Défense

Les machines de Turing on été introduites par Turing en 1936.

Contexte historique : répondre à la question d'Hilbert : Qu'est-ce qui est calculable ? Elle s'inscrive dans la formalisation des modèles de calculs formels qui tendent à répondre à cette question. On peut également cité les fonctions récursives et le λ-calcul ou la logique car calculer c'est prouver (Gödel).

Les modèle de calcul formel permettent d'appréhender les limites non-physiques mais bien conceptuelles du calcul et de l'informatique. Les fonctions récursives s'inscrivent dans une démarche qui identifie les fonctions non-calculables. Turing se tourne quant-a lui vers le calcul et tendent à répondre à la question comment calcule-t-on ? Les machines de Turing sont alors une réponse de logicien à l'action du calcul tel que réalisé par un humain. Elles servent aujourd'hui d'étalon dans la théorie de la complexité. Les machines de Turing semblent "stable" : à vouloir améliorer les Machines de Turing, on retombe toujours sur des machines reconnaissant la même classe de langages : elles semblent englober toute idée de procédure effective.

Ce qu'en dit le jury

Il s'agit de présenter un modèle de calcul. Le candidat doit expliquer l'intérêt de disposer d'un modèle formel de calcul et discuter le choix des machines de Turing. La leçon ne peut

se réduire à la leçon 914 ou à la leçon 915, même si, bien sûr, la complexité et l'indécidabilité sont des exemples d'applications. Plusieurs développements peuvent être communs avec une des leçons 914, 915, mais il est apprécié qu'un développement spécifique soit proposé, comme le lien avec d'autres modèles de calcul, ou le lien entre diverses variantes des machines de Turing.

Métaplan

Motivations :

- ⇝ Pourquoi avoir un modèle de calcul formel ? Répondre à la question de Hilbert : Qu'est-ce qui est calculable. Plusieurs modèles de calculs existent : les fonctions récursives, le λ-calcul, les machines de Turing, ... Ils tentent tous de répondre à la question.
- ⇝ Pourquoi les machines de Turing ? Même si elles n'ont pas été le premier modèle de calcul formelle, elle sont aujourd'hui considéré comme le modèle abstrait des ordinateurs (modèle RAM). De plus, même si elles n'ont pas apporté de nouvelles réponses à la question de Hilbert, par son approche novatrice elle a permis de classifier les fonctions calculables (et donc les problèmes).

I. *Les Machines de Turing : un modèle de calcul formel*

- ⇝ *Définition* [15, p.115] : Un modèle de calcul.

A. *Vocabulaire autours des machines de Turing [26, p.103]*

- ⇝ *Définition* : Machine de Turing déterministe
- ⇝ *Définition* : Configuration
- ⇝ *Application* [15, p.132] : Automates finies
- ⇝ *Définition* : Configurations suivantes
- ⇝ *Définition* : Exécution
- ⇝ *Définition* : Langage accepté
- ⇝ *Définition* : Langage décidé

B. *Les machines de Turing calculent [24, p.59]* On veut pouvoir calculer avec les machines de Turing.

- ⇝ *Exemple* : Additionneur
- ⇝ *Exemple* : Soustracteur
- ⇝ *Exemple* : Multiplicateur (par 2, de deux entiers)
- ⇝ *Définition* [26] : Fonction calculable par une machine de Turing.

II. *Justification de la thèse de Church* C'est une thèse (ni une hypothèse, ni un théorème). Nous ne pouvons pas le montrer formellement (facilement) car la notion d'algorithme ne possède pas de définition formelle. Cependant nous allons mettre en avant deux arguments qui nous conforte cette thèse

— Pas d'extension qui améliore le langage décidé (toute équivalente à une machine de Turing) (sous-section A) ;

— Équivalence avec les autres modèles de calculs évoqué plus tard (sous-section B).

- ⇝ *Thèse de Church* [26, p.109] : Les fonctions calculables par un algorithme sont les fonctions calculables par une machine de Turing.

A. *Les extensions d'une machine de Turing décident les mêmes langages*

Cas des machines à plusieurs rubans [26, p.112]

- ⇝ *Définition* : Fonction transition
- ⇝ *Définition* : Configuration
- ⇝ *Proposition* : Simulation par une machine à un ruban

Cas des machines à ruban bi-infini [26, p.110]

- ⇝ *Définition* : Configuration
- ⇝ *Proposition* : Simulation par une machine à deux rubans

Cas des machines non-déterministes [26, p.114]

- ⇝ *Définition* : Relation du transition
- ⇝ *Définition* : Configuration suivante
- ⇝ *Définition* : Langage accepté
- ⇝ *Proposition* : Simulation par une machine déterministe à trois rubans
- ⇝ *Application* : Satisfiabilité d'une formule du langage propositionnel
- ⇝ *Application* : Déterminisation d'un langage

B. *Les autres modèles de calcul décident les mêmes langages*

Cas des fonctions récursives [26, p.131]

- ⇝ *Définition* : Syntaxe des fonctions μ-récursive.
- ⇝ *Définition* : Sémantique des fonctions μ-récursive.
- ⇝ *Définition* : Fonction μ-récursive.
- ⇝ *Théorème* : Fonctions μ-récursives $\Leftrightarrow$ fonctions Turing-calculable. (DEV : $\Leftarrow$)
- ⇝ *Remarque* : Thèse de Church

Cas du λ-calcul [15, p.185]

- ⇝ *Définition* : Termes.
- ⇝ *Exemple* : Entiers de Church
- ⇝ *Théorème* : Fonctions μ-récursives $\Leftrightarrow$ fonctions λ-définissable
- ⇝ *Corollaire* : Fonctions Turing calculable $\Leftrightarrow$ fonctions λ-définissable

C. *La théorie de la calculabilité [26, p.139]* La calculabilité se fonde sur la thèse de Church-Turing ; elle permet aussi de montrer l'existence de fonctions non-calculables, par un simple argument de diagonalisation. Il existe une "équivalence" entre langages et programmes.

- ⇝ *Définition* [26, p.116] : Machines universelles
- ⇝ *Définition* : Langages R et RE
- ⇝ *Définition* : Langages décidables
- ⇝ *Problème* : ARRÊT
- ⇝ *Théorème* : ARRÊT est indécidable et dans RE
- ⇝ *Définition* : Réduction
- ⇝ *Théorème* : Utilisation des réductions
- ⇝ *Problème* : PAVAGE DE WANG
- ⇝ *Application* : PAVAGE DE WANG est indécidable
- ⇝ *Théorème* : Théorème de Rice
- ⇝ *Application* : Lien avec sémantique

III. *Les machines de Turing classent les fonctions calculables* Les machines de Turing servent d'étalon pour définir les classes de complexité. Dans cette partie, nous différencions les machines de Turing déterministes et non-déterministe (le facteur exponentielle existant dans la déterminisation ne nous permet plus d'exploité leur équivalence). Nous avons une première application des différentes variantes de nos machines de Turing.

- ⇝ *Définition* : Arbre de calcul
- ⇝ *Définition* : Machine décide un langage en temps/espace f
- ⇝ *Définition* : Famille (N)TIME et (N)SPACE
- ⇝ *Définition* : Classes P -> EXPSPACE
- ⇝ *Théorème* : Savitch
- ⇝ *Corollaire* : NPSAPCE = PSPACE
- ⇝ *Problème* : UNIVERSALITÉ
- ⇝ *Proposition* : UNIVERSALITÉ est dans PSPACE
- ⇝ *Définition* : NP-complétude
- ⇝ *Problème* : SAT
- ⇝ *Théorème* : Cook
- ⇝ *Définition* : classes (N)L avec des machines à plusieurs rubans
- ⇝ *Remarque* : Nécessité d'avoir plusieurs types de Machines de Turing.
- ⇝ *Problème* ACCESSIBILITÉ
- ⇝ *Proposition* : ACCESSIBILITÉ est dans NL.
- ⇝ *Remarque* : Si G est un graphe non-orienté alors le problème arrive dans L.

Ouverture

- ⇝ Hiérarchie de Chomsky
- ⇝ Autres modèles de calculs : les circuits booléens

Leçon 914 : Décidabilité et indécidabilité. Exemples.

Références pour la leçon

[2] Carton, *Langages formels, calculabilité et complexité.*
[12] Huth et Ryan, *Logic in Computer Science : Modelling and Reasoning About Systems.*
[26] Wolper, *Introduction à la calculabilité*

Développements de la leçon

Caractérisation RE	Indécidabilité du problème de validité dans la logique FO

Motivations

Ce qu'en dit le jury

Le programme de l'option offre de très nombreuses possibilités d'exemples. Si les exemples classiques de problèmes sur les machines de Turing figurent naturellement dans la leçon, le jury apprécie des exemples issus d'autres parties du programme : théorie des langages, logique,...

Le jury portera une attention particulière à une formalisation propre des réductions, qui sont parfois très approximatives.

Métaplan

I. La théorie de la décidabilité

A. *Des modèles de calcul* Des modèles de calculs qui donnent un cadre à la théorie.
- ⇝ *Définition* : MT
- ⇝ *Définition* : langages décidables
- ⇝ *Définition* : fonction calculables
- ⇝ *Théorèmes* : équivalence de ces modèles
- ⇝ Thèse de Church

B. *Des problèmes de décision* Ce sont les problèmes sur lesquels on décide l'indécidabilité ;)
- ⇝ *Définition* : Problème de décision
- ⇝ *Remarque* : Correspondance langages, fonctions

C. *Les classes R et RE* Ces classes permettent de définir la frontière entre le décidable et l'indécidable.
- ⇝ *Définition* : Classes R et RE
- ⇝ *Remarque* : RE est non vide
- ⇝ *Définition* : Énumérateur
- ⇝ *Définition* : Co-RE
- ⇝ *Théorème* : Caractérisation de RE

II. Prouver l'indécidabilité Comment savoir si un problème est indécidable ? Et surtout comment le montrer ?

A. *Un premier problème indécidable*

- ⇝ *Problème* : Arrêt
- ⇝ *Théorème* : Arrêt est RE et indécidable

B. *La technique de la réduction* Pour montrer qu'un problème est indécidable, nous utilisons une technique de preuve : la réduction. Pour appliquer le principe de réduction il nous faut connaître un premier problème indécidable.

- ⇝ *Définition* : Réduction
- ⇝ *Propriété* : Implications à la réduction
- ⇝ *Exemples* : réductions

III. Expressivité et décidabilité : un compromis Plus un modèle de calcul, une machine, un objet est expressif moins il est décidable.

A. *Une frontière a atteindre* Illustration de cette expressivité contre la décidabilité

Hiérarchie de Chomsky

- ⇝ Décidabilité des langages rationnels
- ⇝ Indécidabilité des langages algébriques
- ⇝ Machines de Turing : Rice
- ⇝ *Application* : Langage de programmation

Logique

- ⇝ Décidabilité dans la logique propositionnelle.
- ⇝ Indécidabilité de validité dans la logique FO.
- ⇝ *Application* : Bases de données
- ⇝ Décidabilité de Presburger
- ⇝ Indécidabilité de Peano

B. *Indécidabilité : rien n'est perdu*

- ⇝ *Idée* : Restriction des hypothèses
- ⇝ *Applications* : vérification, preuves

C. *Décidabilité : rien n'est gagné* Pas de limite théorique mais des limites "pratiques"

- ⇝ *Remarque* : Problèmes non élémentaires
- ⇝ Théorie de la complexité
- ⇝ *Remarque* : Déjà difficile pour NP alors pour EXPTIME

Leçon 915 : Classes de complexité. Exemples.

Références pour la leçon

[2] Carton, *Langages formels, calculabilité et complexité.*
[4] Cormen, *Algorithmique.*
[8] Floyd et Biegel, *Le langage des machines.*
[13] Kleinberg et Tardos, *Design algorithms.*
[20] Papadimitriou, *Computational complexity.*
[23] Sipser, *Introduction to the Theory of Computation.*
[26] Wolper, *Introduction à la calculabilité.*

Développements de la leçon

Le problème PSA est NP-complet

Théorème de Savitch

Motivation

Défense

Une première classification des problèmes en informatique est sa décidabilité ou non. Cependant, cette classification est grossière et une question naturelle a été de classifier les problème décidable en fonction de leur dureté : l'accessibilité dans un graphe et Presburger ce n'est pas tout à fait le même problème.

Les machines de Turing sont un outils permettant de réaliser cette classification : elles permettent de définir la complexité d'un problème. Nous allons donc étudier les classes de complexité en temps et en espace (les familles de problèmes qui ont des complexités équivalentes) et leurs hiérarchies. Remarquons que nous nous intéressons à la complexité d'un problème et non d'un algorithme et même si les deux notions sont très liées, l'étude précise d'une implémentation ne nous intéresse pas (on souhaite uniquement mettre le problème dans une case).

Ce qu'en dit le jury

Le jury attend que le candidat aborde à la fois la complexité en temps et en espace. Il faut naturellement exhiber des exemples de problèmes appartenant aux classes de complexité introduites, et montrer les relations d'inclusion existantes entre ces classes, en abordant le carac-

tère strict ou non de ces inclusions. Le jury s'attend à ce que les notions de réduction polynomiale, de problème complet pour une classe, de robustesse d'une classe vis à vis des modèles de calcul soient abordées.

Se focaliser sur la décidabilité dans cette leçon serait hors sujet.

Métaplan

— *Motivation* : Classer les problème décidable en fonction de leur difficulté (la difficulté est de définir la difficulté et la notion de problème plus dur qu'un autre).

— *Cadre* : Problème décidable.

— *Pré-requis* : Machine de Turing (déterministe, non déterministe, à plusieurs rubans (elles ne servent que pour NL (hors programme), si on en parle pas, on peut tout faire avec un seul ruban)), exécution, configurations.

I. Classes de complexité Pour définir les classes de complexité, nous devons définir les problèmes de décision et surtout soulever le problème du codage des entiers. Ensuite, nous définissons la complexité pour notre modèle de calcul qui sert d'échelon : les machine de Turing (déterministe, non déterministe, à plusieurs ruban).

A. *Problèmes de décision [2]* Un des points subtile dans la théorie de la complexité est le codage des entrées de notre problème. En effet, selon le codage choisi, la complexité de notre problème peut être différente (ou la preuve plus ou moins difficile).

- ⇝ *Définition* : Problème de décision
- ⇝ *Problème* : Codage des entrées
- ⇝ *Exemple* : Primes et UnaryPrimes

B. *Complexité d'une machines de Turing [2]* La complexité d'une machine de Turing est sa consommation d'une ressource donnée lors de son exécution. Cette ressource est temporelle ou spatiale. Comme on ne se concentre que sur des problèmes décidables, on peut supposer que les exécutions de notre machine sont finies. On utilise la taille des configurations : il nous faut faire attention dans le cadre d'une machine à plusieurs rubans.

- ⇝ *Définition* : Complexité d'une exécution.
- ⇝ *Définition* : Complexité d'une machine.
- ⇝ *Remarque* : Cas d'une machine de Turing à plusieurs rubans.
- ⇝ *Définition* : Complexité pour une entrée.
- ⇝ *Théorème* : Accélération temporelle et spatiale.

C. *Classes de complexité [2]* Nous allons maintenant définir la complexité d'un problème de décision et les classes de complexités usuelles. On choisit des machines de Turing à une seule bande. Grâce au théorème d'accélération, on sait que les constance ne sont pas nécessaire dans l'étude de la complexité. On commence à poser l'idée que la logique est une bonne porte d'entrée pour résoudre des problèmes difficile

- ⇝ *Définition* : Famille de problème en espace et en temps
- ⇝ *Définition* : Classes de complexité usuelles
- ⇝ *Proposition* : Hiérarchie
- ⇝ *Remarque* : Implication de l'encodage
- ⇝ *Définition* : Fonction constructible
- ⇝ *Exemple* : Logique dans les classes
- ⇝ *Proposition* : Rupture dans la hiérarchie
- ⇝ *Corollaire* : P$\neq$ EXPTIME et PSPACE$\neq$ EXPSPACE

II. Complexité en temps : P vs NP Nous allons nous intéresser à la complexité en temps et notamment aux classes P et NP. Nous allons étudier leur robustesse en fonction du modèle. Puis nous allons étudier la complétude et son impacte sur la question ouverte $P = NP$ ou non. Enfin, nous allons regarder au-delà avec la classe EXPTIME.

A. *Robustesse des classes temporelles [2, 26]* Les classes temporelles sont robustes à l'ajout ou à la suppression de ruban. Cependant, elles ne le sont pas au passage au non-déterministe. Cela implique que nous ne savons toujours pas si P égale ou non NP ou si EXPTIME égale ou non NEXPTIME : on ne peut donc pas déterminiser facilement sans impacter la complexité une machine non-déterministe.

- ⇝ *Proposition* : Équivalente machine plusieurs rubans.
- ⇝ *Interprétation* : Utilisation d'une machine à plusieurs bandes.
- ⇝ *Proposition* : Équivalence machine non-déterministe.

⇝ *Interprétation* : Les classes P $\neq$ NP ou EXPTIME $\neq$ NEXPTIME.

B. *Notion de NP-complétude et conséquences théoriques [2]* Afin d'étudier les spécificités de la classe NP, nous allons définir une famille de problème qui sont plus durs que tous les autres de cette classe. On définit alors la notion de complétude. Dans le cas de ces classes, nous devons définir la notion de réduction polynomiale.

⇝ *Définition* : Réduction polynomiale
⇝ *Définition* : Problème NP-dure et NP-complet
⇝ *Théorème* : Cook
⇝ *Application* : Autres problèmes NP-complet
⇝ *Exemple* [8] : Problème PSA est NP-complet
⇝ *Exemples* : 3-coloration, set-cover, ...
⇝ *Corollaire* : $p \in$ NP-complet et $p \in$ P implique P = NP.
⇝ *Corollaire* : P $\neq$ NP implique $\exists p \in$ NP $\setminus$ NP-complet.

C. *Conséquence de la NP-complétude en pratique [13]* Un problème NP-complet peut se traiter en pratique. On peut alors jouer : le temps, l'expressivité ou l'exactitude (si on a un problème d'optimisation).

⇝ *Méthode* : Backtracking ou Branch and bound
⇝ *Méthode* : Approximation
⇝ *Méthode* : Restriction des entrées
⇝ *Exemple* : DPLL
⇝ *Exemple* [4] : Approximation du problème TSP
⇝ *Exemple* : Logique de Horn

D. *Au delà de la NP-complétude : la classe EXPTIME [23]* Les classes P et NP ne sont pas les seules classes temporelles (même si elles sont très intéressantes). Les classes EXPTIME et NEXPTIME contiennent des problèmes dont la résolution n'est pas aisée car ce sont des tours d'exponentielle. Notons qu'il existe des problèmes encore plus difficile : les problèmes non-élémentaires : qui sont EXPTIME pour tout k de tour d'exponentielle. La généralisation de Presburer est un tel problème.

⇝ *Exemple* (admis) : Presburger
⇝ *Proposition* : Si EXPTIME $\neq$ NEXPTIME alors P $\neq$ NP
⇝ *Définition* : Problèmes non élémentaires
⇝ *Exemple* : Presburger et généralisation

III. Complexité en espace : *PSPACE* [23]

Nous allons nous intéresser à la complexité en espace et notamment aux classes PSPACE et NPSPACE. Nous allons étudier leur robustesse en fonction du modèle. Puis nous allons étudier la complétude. Enfin, nous étudierons l'impacte de la complexité en espace dans l'étude de la classe P.

A. *Robustesse des classes spatiales* Dans le cas de la complexité spatiale, les classes sont robustes si on déterminise la machine (théorème de Savitch). Cependant, elles ne le sont pas nécessairement face aux multibandes comme nous le verrons plus tard (même si ça ne change rien pour PSPACE).

⇝ *Théorème* : Savitch
⇝ *Corollaire* : PSPACE = NPSPACE
⇝ *Corollaire* : EXPSPACE = NEXPSPACE

B. *Notion de PSPACE-complétude et conséquences pratiques* La complétude dans PSPACE se défini à l'aide d'une réduction polynomiale. On peut alors définir des solveurs qui viennent résoudre l'ensemble des problèmes PSPACE.

⇝ *Définition* : Problème PSPACE-dur et PSPACE-complet.
⇝ *Proposition* : Le problème QBF est PSPACE-complet.
⇝ *Application* : Montrer que les problèmes sont PSPACE-complet.
⇝ *Exemple* : Le problème d'universalité algébrique est PSPACE-complet.
⇝ *Remarque* : De manière analogue aux SAT-solver, on produit des QBF-solver.

C. *Exploration de la classe P [20]* La complexité spatiale permet d'explorer la classe *P* est définissant une complexité log-space, la réduction qui va avec et les sous-classes qu'on en déduit. Nous définissons alors la classe NL, la NL-complétude et la P-complétude avec la réduction log-space.

⇝ *Définition* : Classes NL et L
⇝ *Définition* : Log-space réduction
⇝ *Exemple* : Accessibilité dans un graphe est dans NL
⇝ *Définition* : Problème NL-dur et NL-complet
⇝ *Proposition* : Le problème d'accessibilité dans un graphe est NL -complet.
⇝ *Application* : Montrer que d'autre problème le sont.
⇝ *Exemple* : Le problème 2-SAT est NL-complet
⇝ *Remarque* : Théorème de Savitch
⇝ *Définition* : Problème P-dur et P-complet
⇝ *Proposition* : Le problème Horn est P-complet

IV. Hiérarchie Nous pouvons conclure avec le dessin de la hiérarchie telle que nous la connaissons aujourd'hui.

$\rightsquigarrow$ Conclusion sur la hiérarchie des classes

Ouverture Machines de Turing alternantes ; circuits, Machines de Turing à oracle ; logique

Leçon 916 : Formule du calcul propositionnel : représentation, forme normale, satisfiabilité. Applications.

Références pour la leçon

[3] Cori et Lascar, *Logique mathématique, tome 1.*
[7] Duparc *Logique pas à pas.*
[16] Legendre et Schwarzentruber, *Compilation : Analyse lexicale et syntaxique du texte à sa structure en informatique.*
[24] Stern, *Fondement mathématiques de l'informatique.*

Développements de la leçon

Transformation de Tseitin	2-SAT est NL-complet (donc dans P)

Motivation

Défense

La logique propositionnelle est une logique très simple permettant d'exprimer des contraintes : c'est la première logique que nous utilisons.

Ce qu'en dit le jury

Le jury attend des candidats qu'ils abordent les questions de la complexité de la satisfiabilité. Pour autant, les applications ne sauraient se réduire à la réduction de problèmes NP-complets à SAT.

Une partie significative du plan doit être consacrée à la représentation des formules et à leurs formes normales.

Métaplan

I. Le langage de la logique propositionnelle On formalise la logique grâce à une syntaxe, une sémantique et un lien entre les deux donné par un système de preuve (sur la syntaxe). Nous allons également donner plusieurs représentations possibles (dans un ordinateur) de la syntaxe.

A. *Syntaxe [7, p.68]*

- ⇝ *Définition* : Langage avec connecteurs minimaux (induction)
- ⇝ *Notation* : Autres connecteurs
- ⇝ *Remarque* : Ambiguïté et parenthèses

B. *Représentations* Comment on représente les mots de ce langage dans un ordinateur ? Laquelle est la plus efficace (sous quels critères) ?

- ⇝ Représentation linéaire
- ⇝ Représentation arborescente
- ⇝ *Théorème*[3, p.57] : Lecture unique
- ⇝ Représentation BD
- ⇝ Représentation DAG (direct acyclique graph) : moins de mémoire
- ⇝ Représentation circuit

C. *Sémantique [7, p.83]* Comment donner du sens à ce qu'on écrit ? Que signifie ces formules ?

- ⇝ Valuation sur un modèle
- ⇝ *Application* : traduction du langage naturel
- ⇝ *Problèmes* : validité et satisfiabilité
- ⇝ Une représentation : table de vérité
- ⇝ *Remarque* : complexité de calcul
- ⇝ *Théorème* Cook
- ⇝ *Application* : réduction de problème pour NP-dureté (problème de séparation des automates)
- ⇝ *Définition* : tautologie / contradictions

D. *Théories et preuves* Vérifier que nous n'avons pas fait n'importe quoi avec ces définitions...

- ⇝ *Définition* : Théorie
- ⇝ *Théorème* : Compacité
- ⇝ *Applications* : Coloriage de sous-graphe, pavage, logique du premier ordre
- ⇝ *Définition* : CNF
- ⇝ *Définition* : Résolution
- ⇝ *Théorème* : Correction et complétude
- ⇝ Taille des arbres et certificats

II. Équivalence de formules

Afin de trouver des moyens de résoudre le problème de satisfiabilité, on peut chercher à mettre nos formules sous différentes forme qui sont équivalentes : elles expriment les mêmes contraintes. Pour cela nous avons besoins de la notions d'équivalence et de formes normales.

A. *Équivalence sémantique et équisatisfiabilité [7, p.106]* Comparer deux formules c'est connaître les modèles qui les discrimine (satisfait une des formule mais pas l'autre). Lorsque nous ne pouvons pas les différentier, nous parlons d'équivalence sémantique (c'est l'équivalence que l'on souhaite).

- ⇝ *Définition* : Équivalence sémantique
- ⇝ *Exemples* : Loi de Morgan
- ⇝ *Proposition* : Caractérisation avec la validité
- ⇝ *Conséquence* : Complexité de VALIDITE
- ⇝ *Définition* : Équisatisfiabilité
- ⇝ *Proposition* : Équivalent implique équisatisfiable

B. *Systèmes de connecteurs* L'ensemble des connecteurs logiques, appelé système de connecteurs, que nous choisissons pour écrire nos formules peuvent exprimer plus ou moins de choses. On en veut le moins de symbole possibles pour décrire notre logique.

- ⇝ *Définition* : Systèmes complets et minimaux
- ⇝ *Remarque* : $\{\neg, \wedge, \vee\}$ est un système complet mais pas minimal.
- ⇝ *Proposition* : Systèmes complets minimaux.

C. *Formes normales conjonctives ou disjonctives* Les formes normales sont une manière de représenté une formule par une autre formule dont les connecteurs logiques sont restreints et ordonnés selon un certain ordre. On présente leur mise en forme, leurs points forts et leurs limites

Forme normale négative Elle est à l'origine de toutes les autres

- ⇝ *Définition* : Forme normale négative (par sa grammaire)
- ⇝ *Proposition* : Transformation en une forme normale négative

Forme normale conjonctive

- ⇝ *Proposition* : Existence formule CNF équivalente
- ⇝ *Proposition* : Transformation de Tseitin
- ⇝ *Remarque* : Transformation exponentielle
- ⇝ *Corollaire* : CNF-SAT est NP-complet
- ⇝ *Remarque* : C'est une forme normale négative
- ⇝ *Remarque* : CNF-VALIDE est dans P

Forme normale disjonctive Les formules conjonctives (et uniquement conjonctives) permettent de représenter un modèle.

- ⇝ *Définitions* : Forme normale disjonctive DNF
- ⇝ *Proposition* : Existence d'une DNF équivalente
- ⇝ *Théorème* : DNF-SAT est dans P
- ⇝ *Conséquence* : Transformation exponentielle
- ⇝ *Exemple* : $\varphi_n = (a_1 \vee b_1) \wedge \cdots \wedge (a_n \vee b_n)$
- ⇝ *Remarque* : DNF-VALIDE est NP-complet

III. Résoudre la satisfiabilité en pratique On a vu que savoir si une formule est satisfiable est difficile. Cependant, en pratique on est souvent amené à résoudre cette question : on pose des contraintes (emplois du temps) ou issue de problème NP-complet. On aimerai alors un moyen de résoudre la satisfiabilité de manière efficace.

A. *Considérer des fragments de la logique* Limiter la logique (si le problème le permet) peut nous donner des algorithmes qui résolvent la satisfiabilité en temps linaire.

- ⇝ *Problème* : 3SAT -> NP-complet
- ⇝ *Problème* : 2SAT -> NL-dur (donc dans P)
- ⇝ *Application* : Problème d'ordonnancement multiprocesseurs avec 2 créneaux pour les tâches.
- ⇝ *Problème* : Horn -> P
- ⇝ *Application* [24] : Prolog avec les clauses de Horn
- ⇝ *Application* [16] : Analyse syntaxique (calcul de premier, LL(1), problèmes sur grammaires algébriques)

B. *Essayer de résoudre astucieusement* Si on provient d'un problème NP-complet, on peut mettre la formule sous forme CNF et résoudre à l'aide d'un algorithme dont le pire cas s'exécute en temps exponentiel mais dont on espère que celui-ci soit meilleur dans la pratique.

- ⇝ *Algorithme* : DPLL / CDLL
- ⇝ *Application* : SAT-solver

Leçon 918 : Systèmes formels de preuve en logique du premier ordre. Exemples.

Références pour la leçon

[7] Duparc *Logique pas à pas.*
[21] David, Nour et Raffali, *Introduction à la logique. Théorie de la démonstration.*
[24] Stern, *Fondement mathématiques de l'informatique.*

Développements de la leçon

Complétude de la déduction naturelle | Fonctionnement de Prolog

Motivation

Défense

La mathématicien Hilbert au début du $XIX^{\text{ième}}$ a cherché à formaliser et à comprendre la notion de preuve mathématiques. Il cherchait alors le nombre minimal d'axiomes mathématiques nécessaires pour prouver l'ensemble des mathématiques connus. Les logiciens se sont alors intéressés à la notion de preuve et ils ont tentés de répondre à la question : *qu'est-ce qu'une preuve* ?

Les informaticiens se sont ensuite emparés du sujet, notamment avec Curry et Howard dont la correspondance dit que programmer est en réalité prouver (pour cela, ils utilisent le λ-calcul simplement typé et la logique intuitionniste). De cette idée, on a cherché (et on cherche toujours) à automatiser au mieux les preuves à l'aide de la programmation logique ou sous-contrainte, en utilisant des assistants de preuves. Ces derniers sont devenus une aide aux mathématiciens pour certains théorème comme le théorème des quatre couleurs. Mais attention, comme pour tout système informatique, leur correction n'est pas simple à vérifier. Dans ce cas, ils ont tous un noyau qui n'a pas été prouvé : c'est le noyau de confiance de l'assistant auquel on se doit de faire confiance pour assurer la fiabilité des résultats.

Ce qu'en dit le jury

Le jury attend du candidat qu'il présente au moins la déduction naturelle ou un calcul de séquents et qu'il soit capable de développer des preuves dans ce système sur des exemples classiques simples. La présentation des liens entre syntaxe et sémantique, en développant en

particulier les questions de correction et complétude, et de l'apport des systèmes de preuves pour l'automatisation des preuves est également attendue.

Le jury appréciera naturellement si des candidats présentent des notions plus élaborées comme la stratégie d'élimination des coupures mais est bien conscient que la maîtrise de leurs subtilités va au-delà du programme.

Métaplan

Remarque : Il faut réfléchir au choix que nous devons faire : les règles de déductions peuvent elles ou non apparaître en annexe ? Si on les met en annexe, il faut les mettre dans un tableau, sinon ça peut prendre un peu de place dans une leçon il y a beaucoup de chose à dire.

Motivation :

- Hilbert : qu'est-ce qu'une preuve
- Programmer c'est prouver : correspondance de Curry–Howard
- Assistant de preuve

I. Logique du premier ordre Pourquoi l'étude de la logique du premier ordre ? C'est une logique assez puissante pour exprimer le langage naturel. Il est donc naturel lorsqu'on cherche à formaliser les preuves mathématiques (qui s'exprime via un langage humain) à se tourner vers cette logique. Cette section est une section de rappel : il n'est pas nécessaire d'en mettre trop (moins il en a, plus on a de systèmes de preuve et autres).

A. *Syntaxe de la logique du premier ordre [21, p.9]*

- ⇝ *Définition* : Langage de la logique du premier ordre
- ⇝ *Définition* : Terme et terme clos
- ⇝ *Définition* : Formule
- ⇝ *Remarque* : Lien avec le calcul propositionnel
- ⇝ *Définition* : Variable libre
- ⇝ *Définition* : Formule close
- ⇝ *Définition* : Théorie
- ⇝ *Définition* : Substitution

B. *Sémantique de la logique du premier ordre [21, p.75]* On définit le modèle au sens de modèle, on se passe de la définition de signature donc on met directement dans la définition de modèle la satisfiabilité.

- ⇝ *Définition* : Modèle
- ⇝ *Remarque* : Pour les théories
- ⇝ *Exemple* [24, p.220] : Modèle de Herbrand
- ⇝ *Définition* : Équivalence sémantique
- ⇝ *Définition* : Théorie contradictoire
- ⇝ *Problèmes* : Valide et T-Valide

II. Systèmes de preuves Les systèmes de preuve sont uniquement syntaxique. En effet, une preuve est basée et conduite par la syntaxe de la formule et non sa sémantique. Cependant à l'aide de théorème de complétude et de correction, on arrive à lier syntaxe et sémantique. Nos systèmes de preuve ne font donc pas n'importe quoi.

A. *La déduction naturelle* La déduction naturelle est un système de preuve dont toutes les déductions se fond sur le but : on passe tout dans le but pour les manipuler. Cela alourdit la manipulation et même si les règles sont relativement facile, il est délicat à automatiser.

Présentation du système de preuve [21, p.24] Nous donnons ici la présentation du système de preuve : les objets qu'il manipule ainsi que les règles de manipulation. On obtient un nombre fini de règles qui permettent de prouver l'ensemble des formules prouvables : ce qui n'était pas évident au départ.

- ⇝ *Définition* : Séquent
- ⇝ *Définition* : Règle
- ⇝ *Définition* : Séquent prouvable par déduction naturelle
- ⇝ *Définition* : Formule prouvable par déduction naturelle

Lien entre syntaxe et sémantique [21, p.79] Les systèmes de preuves sont des passerelles entre la syntaxe et la sémantique. Les preuves sont des objets purement syntaxique qui ne fond pas n'importe quoi sur la sémantique. Ce sont l'objets de théorème de correction et de complétude de la logique.

- ⇝ *Définition* : Théorie consistante, théorie complète
- ⇝ *Théorème* : Théorie consistante est non-contradictoire
- ⇝ *Corollaire* : Complétude et correction de la déduction naturelle
- ⇝ *Application* : Théorème de compacité

⇝ *Application* [21, p.99] : Théorème de Lowhein–Skolem

⇝ *Proposition* : Problèmes Valide et T-Valide indécidables

B. *Le calcul des séquents [21, p.185]* Le calcul des séquents est plus facile à manipuler que la déduction naturelle car on peut manipuler le contexte comme les conclusions (ce qui n'est pas possible en déduction naturelle) : on obtient des règles symétrique. Cela facilite leur manipulation (c'est souvent ce système de preuve qui est automatisé) même s'il est moins naturel que la déduction naturelle pour formaliser des preuves. Dans le contexte de la formalisation des mathématiques, la coupure est une règle qui nous permet de faire des lemmes et ainsi de formaliser les jolies preuves. Cependant, dans le cadre de l'automatisation cette règle est ingérable : comment choisir où couper ? Ce théorème implique que nous pouvons nous en passer, ce qui est un premier pas vers l'automatisation des preuves.

⇝ *Définition* : Séquent

⇝ *Définition* : Règle

⇝ *Définition* : Séquent et formule prouvable par calcul des séquents

⇝ *Théorème* : Équivalence des systèmes de preuves

⇝ *Théorème* (ADMIS) : Élimination des coupures

III. Automatisation des preuves [24, p.231]

On a ensuite cherché à automatiser les preuve via l'informatique avec l'idée que programmer c'est prouver. Cependant, même si on a un nombre fini de règles, leur automatisation n'est pas si simple (surtout dû au quantificateur existentiel) qui demande l'intervention de l'homme : ce sont les assistants de preuve. On a donc développé plusieurs stratégies basées notamment sur la résolution.

A. *Unification [21, p.248]* L'unification est un premier pas vers l'automatisation car elle permet de repérer deux termes syntaxiquement équivalents et d'appliquer une substitution afin de les rendre identiques.

⇝ *Définition* : Terme unifiable

⇝ *Définition* : Unificateur (principal)

⇝ *Définition* : Équations unifiables

⇝ *Algorithme* : Unification

⇝ *Théorème* : Correction de l'algorithme d'unification

B. *Résolution [21, p.264]* La résolution est un système de preuve car il ne possède que deux règles. Il est également très utile dans l'automatisation des preuves. On choisit d'écrire les deux règles et pas une seule

⇝ *Principe* : On cherche à montrer qu'un ensemble de formule est contradictoire

⇝ *Définition* : Règles

⇝ *Lemme* : Correction de la méthode de résolution

C. *Programmer c'est prouver* La preuve automatique par les programmes est une conséquence de la correspondance de Curry-Howard [7, p.527]. Nous allons étudier quelques exemples de cet adage même si en toute généralité nous devrons parler de λ-calcul simplement typé et logique intuitionniste.

⇝ *Définition* : Clause de Horn

⇝ *Proposition* : Résolution sur les clauses de Horn

⇝ *Application* : Prolog et la programmation logique

⇝ *Exemple* : Requête SQL sur une base de donnée

Leçon 921 : Algorithmes de recherche et structures de données associées.

Références pour la leçon

[1] Beauquier, Berstel et Chretienne, *Éléments d'algorithmique.*
[4] Cormen, *Algorithmique.*
[9] Froidevaux, Gaudel et Soria *Types de données et algorithmes.*
[14] Knuth, *The Art of Programming, vol 3*

Développements de la leçon

Les B-arbres

L'automate minimale pour la recherche de motifs

Motivation

Défense

La recherche est un enjeux clé en informatique. En effet, la recherche peut être une routine dans une application plus complexe. Il nous faut alors des algorithmes efficaces sur des ensembles de données qui peuvent être dynamique, hétérogène, ...

Dans le cadre de cette leçon, il faut faire attention à ce qu'on appelle un problème de recherche car tout problème peut être considérer comme un problème de recherche puisqu'on cherche une solution.

Ce qu'en dit le jury

Le sujet de la leçon concerne essentiellement les algorithmes de recherche pour trouver un élément dans un ensemble : l'intérêt des structures de données proposées et de leur utilisation doit être argumenté dans ce contexte.

La recherche d'une clé dans un dictionnaire sera ainsi par exemple l'occasion de définir la structure de données abstraite « dictionnaire », et d'en proposer plusieurs implantations concrètes. De la même façon, on peut évoquer la recherche d'un mot dans un lexique : les arbres préfixes (ou digital tries) peuvent alors être présentés. Mais on peut aussi s'intéresser à des domaines plus variés, comme la recherche d'un point dans un nuage (et les quad-trees), et bien d'autres encore.

Métaplan

⇝ *Motivation* : Stocker, exploiter et rechercher une donnée précise dans une collection est un problème récurent.

⇝ Problème de recherche L'élément que l'on cherche doit être bien "typé".

I. Recherche dans un dictionnaire

On se ramène à un problème de recherche dans l'ensemble des clés. Les structures de données utilisées pour stocker les clés vont donc être à la base de l'implémentation de notre dictionnaire. On implémente le dictionnaire avec ses données satellitaire : stocké à part des clés et accessible par un pointeur. Dans cette leçon, nous allons pas traiter le cas du nombre d'occurrence. Les algorithmes que nous présentons peuvent être "facilement" adapté pour ce problème (il faut les faire continuer à la place de s'arrêter comme nous le faisons).

Objectif : Recherche d'un élément quand $K = \{(cle, valeur)\}$ et E est un dictionnaire (au sens de la structure abstraite).

A. *La recherche suit une loi uniforme sur l'ensemble des clés*

On veut une complexité de la recherche qui ne dépend pas de la position de la clé. Les structures de données associées doivent être équilibré : la place de la donnée recherché ne doit pas influencer sur le temps de recherche. On a un compromis entre la complexité pour ajouter et celle pour la recherche. On choisit la structure de donnée en fonction de la propriété dynamique des données.

Objectif : Recherche d'un élément quand $K = \{(cle, valeur)\}$ où les clés sont à valeur dans un ensemble et E est un dictionnaire et k est uniformément distribué sur E.

Structure	Insertion	Suppression (sans recherche)	Recherche
Tableau (non triée)	$O(n)$	$O(n)$	$O(n)$
Tableau (triée)	$O(n \log n)$	$O(n)$	$O(\log n)$
Liste	$O(1)$	$O(1)$	$O(n)$
Liste triée	$O(\log n)$	$O(1)$	$O(\log n)$
ABR	$O(n)$	(dépend de la maintenance)	$O(n)$

Recherche naïve : On compare k à tous les éléments de E

⇝ Structure de données associée : liste tableau

⇝ Complexité temporelle dans le pire cas : $O(n)$.

⇝ + : facilité de la maintenance de la structure

⇝ - : complexité temporelle de la recherche

Recherche dichotomique [9, p.178]

⇝ Structure de données associée : utilisation d'un tableau **trié**

⇝ Complexité temporelle dans le pire cas : $O(\log n)$

⇝ + : complexité de la recherche

⇝ - : on doit triée et maintenir **triée** le tableau

Recherche supportée par un ABR équilibré La recherche dichotomique se rapproche à une recherche dans un ABR. Les ABR sont des structures de données facilitant l'implémentation du principe de la dichotomie (notamment dans le maintient de la structure).

⇝ *Définition* [9, p.197] : Arbre binaire de recherche (ABR)

⇝ *Définition* [9, p.221] : ABR est dit équilibré

⇝ *Implémentation* [9, p.224] : AVL

⇝ *Implémentation* [4, p.287] : Arbre rouge-noir

⇝ *Proposition* : Complexité temporelles dans le pire cas : $O(\log n)$

Quelques exemples en géométrie algorithmique

⇝ *Problème* : L'intersection de s droites par la méthode de balayage.

⇝ *Proposition* : Complexité pour trouver une intersection : $O(s \log s)$ dans le pire cas.

⇝ *Implémentation* : ABR équilibré (arbre rouge-noir)

⇝ *Problème* : La recherche d'un point dans un nuage.

⇝ S*Implémentation* : quad-tree

B. *La recherche n'est pas uniforme sur l'ensemble des clés*

Constations : Il existe des clés qui apparaissent plus souvent que d'autre dans les requêtes de la recherche : pour celles-ci nous devons minimiser les temps de recherche. Les complexité pour atteindre les clés les plus fréquentes doit être minimale.*Limites* : Il nous faut un ordre sur les clés. De plus, ces structure

l'accès concurrent n'est pas efficace. En effet, lors d'accès concurrents, le sous-arbre enraciné en le noeud sur lequel on ait ne peut plus être partager (des fois qu'on modifierait le sous-arbre).

Objectif : Recherche d'un élément quand $K = \{(cle, valeur)\}$ et E est un dictionnaire (au sens de la structure abstraite).

Recherche auto-adaptative : mettre en première position l'élément que l'on vient de rechercher

- ⇝ Structure de données associée [9, p.175] : liste
- ⇝ *Remarque* : Variante avec les tableaux
- ⇝ Remarque culturelle [14] : complexité temporelles dans le pire cas est en $O(\frac{n}{\log n})$
- ⇝ + : on n'a pas besoin de connaître la distribution des clés

Recherche supportée par un ABR optimal [4, p.368]

- ⇝ Pour construire l'arbre, on a besoin de connaître la distribution de probabilité : non dynamique.
- ⇝ *Définition* : ABR optimal
- ⇝ *Implémentation* : Arbre splay

C. *La recherche s'effectue sur un ensemble sur lequel on ne souhaite pas utiliser son ordre total* La recherche ne doit pas perturbée par l'ordre (ou le manque d'ordre) sur les clés. La table de hachage peut être utilisée de manière concurrente : on ne bloque que la case du tableau que l'on vient d'atteindre donc si la fonction n'est pas trop mauvaise, la majorité de nos clés restent accessibles.

- ⇝ *Objectif* : Recherche d'un élément quand $K = \{(cle, valeur)\}$ pour lequel on ne veut pas travailler avec l'ordre totale des clés.
- ⇝ *Principe* : Hachage
- ⇝ *Implémentation* [4, p.238] : table de hachage
- ⇝ Gestion des collisions : chaînage simple
- ⇝ *Définition* : Collision
- ⇝ Complexité de la recherche en $O(n)$ dans le pire cas.
- ⇝ *Remarque* : Choix de la fonction de hachage
- ⇝ *Définition* [4, p.258] : Fonction de hachage parfaite

D. *Le dictionnaire doit être stocké sur un disque externe* Dans ce contexte, le travail en mémoire vive est négligeable devant les accès au disque externe (en ms). Il nous faut donc minimiser le nombre d'accès au disque externe afin d'améliorer la complexité de la recherche. Exemple d'un cas de complexité non temporelle (on regarde le nombre d'accès au disque).

- ⇝ *Objectif* : Recherche d'un élément quand $K = \{(cle, valeur)\}$ où les clés sont à valeur dans un ensemble et E est un dictionnaire qui ne peut pas être dans la mémoire vive.
- ⇝ *Définition* [4, p.447] : B-arbre
- ⇝ *Proposition* : Complexité de la recherche et de l'insertion
- ⇝ *Remarque* : Extension d'un B-arbre en un B^+-arbre
- ⇝ *Utilisation* : Dans les bases de données relationnelles.

II. Recherche d'une chaîne de caractères (motif) dans un texte

Le motif ne peut être contenue dans le texte uniquement si sa taille est inférieure à celle du texte et s'ils sont définis sur un alphabet commun que l'on note Σ. Ce que l'on supposera dans la suite.Un problème analogue : le nombre d'occurrence.

- ⇝ *Objectif* : Recherche d'un élément lorsque K est l'ensemble des mots de $|\Sigma|^*$ (où Σ est un alphabet) et E est l'ensemble des facteurs d'un texte.
- ⇝ *Applications* : Recherche d'un mot dans un texte, d'une chaîne de caractères dans une séquence d'ADN
- ⇝ *Structure de données associée* : Tableau de longueur la taille de la chaîne de caractère considérée.
- ⇝ *Algorithme* [4, p.905] : Naïf
- ⇝ *Proposition* : Complexité dans le pire cas : $O((t-p+1)p)$

Algorithme	Prétraitement	Recherche
Naïf	0	$O((t-p+1)p)$
Rabin-Karp	$\Theta(p)$	$O((t-p+1)p)$
Automate fini	$O(p\|\Sigma\|)$	$\Theta(t)$
Knuth-Moris-Pratt	$\Theta(p)$	$\Theta(t)$
Boyer-Moore	0	$O(tp)$

A. *Algorithme de Rabin-Karp [4, p.905]* Les valeurs pour le texte sont calculées à la volée lors de la recherche. Le prétraitement permet de calculer la valeur du motif et la première valeur pour le texte. Le problème de recherche multiples motifs se résout avec un lourd prétraitement sur le texte T ou les multiples motifs P_i afin de comparer les multiples P_i à une

portion de texte une seule fois. On adapte l'algorithme de Rabin-Karp pour résoudre efficacement ce problème. (Autre algorithme pour ce problème : l'algorithme d'Aho-Corasick.)

- ⇝ *Hypothèse* : Motif fixe et connu
- ⇝ *Hypothèse* : $\Sigma = \{0, 1, \ldots, 9\}$
- ⇝ *Algorithme* : Rabin-Karp
- ⇝ *Proposition* : Complexité du prétraitement : $\Theta(p)$
- ⇝ *Proposition* : Complexité dans le pire cas : $O((t - p + 1)p)$
- ⇝ *Applications* : détection de plagiat ; comparaison d'un fichier suspect à des fragments de virus ; ...

B. *Algorithme de Knuth-Morris-Pratt [4, p.905]* *Constations* : Il existe des clés qui appairassent plus souvent que d'autre dans les requêtes de la recherche : pour celles-ci nous devons minimiser les temps de recherche. Les complexité pour atteindre les clés les plus fréquentes doit être minimale.*Limites* : Il nous faut un ordre sur les clés. De plus, ces structure l'accès concurrent n'est pas efficace. En effet, lors d'accès concurrents, le sous-arbre enraciné en le noeud sur lequel on ait ne peut plus être partager (des fois qu'on modifierait le sous-arbre).

- ⇝ *Hypothèse* : Motif fixe et connu
- ⇝ *Définition* : Fonction suffixe
- ⇝ *Définition* : Automate des occurrences
- ⇝ *Proposition* : Automate des occurrences minimal pour $\Sigma^* P$
- ⇝ *Algorithme* : Moris–Pratt
- ⇝ *Proposition* : Complexité temporelle dans le pire cas
- ⇝ *Idée* : Construction intelligente de l'automate
- ⇝ *Définition* : Fonction préfixe
- ⇝ *Algorithme* : Knuth-Morris-Pratt
- ⇝ *Proposition* : Complexité temporelle dans le pire cas

C. *Algorithme de Boyer et Moore [1, p.358]* La recherche ne doit pas perturbée par l'ordre (ou le manque d'ordre) sur les clés. La table de hachage peut être utilisée de manière concurrente : on ne bloque que la case du tableau que l'on vient d'atteindre donc si la fonction n'est pas trop mauvaise, la majorité de nos clés restent accessibles.

- ⇝ *Principe* : Vérification du motif de la droite vers la gauche.
- ⇝ *Exemples*
- ⇝ *Proposition* : Complexité : $O(pt)$

Ouverture

- ⇝ *Problématique* : Recherche du min et max
- ⇝ *Problème* : K est un ensemble ordonné et $E \subseteq K$. On cherche $k = \min_K E$ ou $k = \max_K E$.
- ⇝ *Problématique* : Union-find et partition d'un ensemble
- ⇝ *Problème* : K est l'ensemble des entiers, $E = \{(n, \bar{n})\}$ est un ensemble d'entier avec son représentant de classe d'équivalence.
- ⇝ *Problématique* : Recherche non-associative
- ⇝ *Application* : Base de données.
- ⇝ *Problématique* : Accessibilité dans un graphe
- ⇝ *Problème* : K est l'ensemble des états d'un graphe et E est l'ensemble des états accessibles de ce graphe.

Leçon 923 : Analyse lexicale et analyse syntaxique. Applications.

Références pour la leçon

[2] Carton, *Langages formels, calculabilité et complexité*
[16] Legendre et Schwarzentruber, *Compilation : Analyse lexicale et syntaxique du texte à sa structure en informatique.*

Développements de la leçon

$\mathcal{L}(G_{post})$ est $LL(1)$ — Construction des premiers

Motivation

Défense

La compilation permet de transformer un programme écrit dans un langage source en un programme sémantiquement équivalent écrit dans un langage cible. Généralement, on compile un langage de programmation comme C vers l'assembleur (le langage machine). Cependant, ce n'est pas la seule compilation que nous pouvons effectuer : on transforme du code latex en fichier en format pdf, ou en code html. De plus, pour compiler un langage de programmation comme OCaml ou Python, on utilise des langages intermédiaire comme C.

L'analyse lexicale et l'analyse syntaxique sont les premières étapes de la compilations. Dans un compilateur actuel, elles sont réalisées en étroite collaboration : l'analyse lexicale donnant à l'analyse syntaxique les mots dont elle a besoin pour continuer. Elles permettent de transformer un texte en un arbre de syntaxe abstraite qui sera la structure sur laquelle nous pourrons continuer la compilation. Cet arbre nous permet ensuite de réaliser une analyse sémantique (vérification de type, levée de certaines exceptions, ...) avant de produire un code intermédiaire au langage que nous souhaitons atteindre. Sur ce code intermédiaire, nous effectuons des opérations d'optimisation qui vise à rendre l'exécution plus rapide que nous l'avons écrite. On finit alors par traduire le bout qui manque.

Ces deux analyses reposent sur des outils théoriques simples et dont l'expressivité nous permet d'obtenir des calcul efficaces : les expressions rationnelles et les grammaires algébriques. En étudiant ces deux étapes de la compilation, on remarque que les automates finis sont pratiques pour couper un texte en mots mais qu'ils se révèle insuffisant pour ordonnancer les éléments en fonction de leurs opérandes. Pour cela (et afin de construire l'arbre de

syntaxe abstraite), nous sommes obligé d'utiliser la puissance d'expression des grammaires algébriques.

Ce qu'en dit le jury

Cette leçon ne doit pas être confondue avec la 909, qui s'intéresse aux seuls langages rationnels, ni avec la 907, sur l'algorithmique du texte.

Si les notions d'automates finis et de langages rationnels et de grammaires algébriques sont au cœur de cette leçon, l'accent doit être mis sur leur utilisation comme outils pour les analyses lexicale et syntaxique. Il s'agit donc d'insister sur la différence entre langages rationnels et algébriques, sans perdre de vue l'aspect applicatif : on pensera bien sûr à la compilation. Le programme permet également des développements pour cette leçon avec une ouverture sur des aspects élémentaires d'analyse sémantique.

Métaplan

- ⇝ *Motivation* : Chaîne de compilation
- ⇝ *Motivation* : Importance de la compilation

I. Analyse lexicale [16] On commence par la première étape de la compilation : l'analyse lexicale qui s'avère être la plus facile.

- ⇝ *Problème* : Transformer une suite de lettre en lexème
- ⇝ *Méthode* : Pattern-matching sur les expressions régulières
- ⇝ *Valeur ajoutée* : Erreur lexicale, filtrer les programmes, décoration des lexèmes produits

A. *Expressions rationnelles et automates finis [2, p.38]* On rappelle succinctement ces notions et leur équivalence. Celle-ci donnent les outils efficaces au cœur de l'analyse lexicale : les automates finis reconnaissants les expressions régulières caractérisant les mots autorisé dans le langage.

- ⇝ *Définition* : Expression régulière
- ⇝ *Définition* : Automate fini
- ⇝ *Théorème* : Théorème de Kleene
- ⇝ *Preuve* : Construction de Thomson
- ⇝ *Exemple* : Automate des motifs

B. *Analyseur lexical* On décrit maintenant les différentes méthodes d'analyse lexicale ainsi que leur complexité.

- ⇝ *Méthode* : Via les automates finis munis d'une priorité
- ⇝ *Remarque* : Détection d'erreurs
- ⇝ *Proposition* : Complexité au pire cas : $O(n^2)$
- ⇝ *Remarque* : Dans le cas d'un langage de programmation, on est généralement en $O(n)$.
- ⇝ *Méthode* : Via la programmation dynamique

II. Analyse syntaxique [16] L'analyse syntaxique est une étape plus compliquée à mettre en place qui demande l'utilisation d'outils plus conséquents : les grammaires algébriques.

- ⇝ *Problème* : Transformer une suite de lexème en arbre de syntaxe abstraite
- ⇝ *Remarque* : Problème qui est plus difficile
- ⇝ *Valeur ajoutée* : Erreur syntaxique
- ⇝ *Méthode* : ascendante ou descendante sur les grammaires

A. *Grammaire algébrique [2]* On rappelle succinctement les notions autours des grammaires algébriques. Ces notions vont nous être utile pour réaliser l'analyse syntaxique à partir de la grammaire de notre langage source.

- ⇝ *Définition* : Grammaire algébrique
- ⇝ *Définition* : Dérivation (gauche / droite)
- ⇝ *Définition* : Arbre de dérivation
- ⇝ *Remarque* : Dérivation vs arbre de dérivation
- ⇝ *Définition* : Ambiguïté
- ⇝ *Remarque* : Impact lors l'analyse syntaxique
- ⇝ *Définition* : Langage engendré
- ⇝ *Remarque* : Les rationnels sont algébriques.

B. *Analyse générique* Regardons les méthodes génériques : elles peuvent être utilisées pour toutes les grammaires algébriques.

- ⇝ *Remarque* : La méthode "naïve" qui utilise du backtracking est exponentielle

- ⇝ *Définition* : Grammaire de Chomsky
- ⇝ *Théorème* : Toute grammaire peut être mise sous forme de Chomsky
- ⇝ *Remarque* : Transformation peu coûteuse
- ⇝ *Proposition* : Le problème du mot est dans P
- ⇝ *Algorithme* : CYK et sa complexité
- ⇝ *Application* : Analyse syntaxique

C. *Méthode descendante* L'algorithme générique nous donne une analyse cubique. En réfléchissant sur la structure des grammaires des langages de programmation (qui sont pour la plus part agréable), nous pouvons grâce à des méthodes gloutonnes obtenir une analyse linéaire. Une première analyse est par méthode descendante dans la grammaire : on part de l'axiome pour obtenir le mot.

- ⇝ *Principe* :
- ⇝ *Définition* : Grammaire LL(1)
- ⇝ *Exemple* : $\mathcal{L}\left(G_{post}\right)$ est $LL(1)$.
- ⇝ *Proposition* : Caractérisation du premier
- ⇝ *Remarque* : Expressivité : langages de programmations concernés
- ⇝ *Algorithme*
- ⇝ *Limite* : Langages non $LL(1)$

D. *Méthode ascendante* Une première analyse est par méthode ascendante : on part des terminaux pour obtenir l'axiome.

- ⇝ *Principe* :
- ⇝ *Définition* : Grammaire LR(0)
- ⇝ *Définition* : Grammaire LR(1)
- ⇝ *Algorithme* :
- ⇝ *Remarque* : Expressivité : langages de programmations concernés
- ⇝ *Limite*

Conclusion : Récapitulation des différents types de grammaires et de leurs méthodes d'analyse.

Ouverture Analyse sémantique : λ-calcul simplement typé et correspondance de Curry-Howard.

Leçon 925 : Graphes : représentations et algorithmes.

Références pour la leçon

[1] Beauquier, Berstel et Chretienne, *Éléments d'algorithmique.*
[4] Cormen, *Algorithmique*
[9] Froidevaux, Gaudel et Soria, *Types de données et algorithmes.*

Développements de la leçon

Le tri topologique	Approximation ou non du problème TSP

Motivation

Défense

Les graphes sont des objets qui permettent de modéliser de nombreux problèmes informatiques et de la vie de tous les jours : réseaux (transport, internet, information, eau, électricité), relation entre des entité (réseaux sociaux, contraintes), circuit imprimé, des programmes ... Leurs études et les problèmes sont donc fondamentaux. Il existe également des problèmes calculatoires sur ces graphes. Lorsqu'on étudie les graphes, de nombreux paradigmes informatiques ressortent : on a différents paradigmes algorithmiques, du calcul de complexité, l'utilisation pertinente de structure de données ou encore l'apparition de quelques classes de problèmes.

Ce qu'en dit le jury

Cette leçon offre une grande liberté de choix au candidat, qui peut choisir de présenter des algorithmes sur des problèmes variés : connexité, diamètre, arbre couvrant, flot maximal, plus court chemin, cycle eulérien, etc. mais aussi des problèmes plus difficiles, comme la couverture de sommets ou la recherche d'un cycle hamiltonien, pour lesquels il pourra proposer des algorithmes d'approximation ou des heuristiques usuelles. Une preuve de correction des algorithmes proposés sera évidemment appréciée. Il est attendu que diverses représentations des graphes soient présentées et comparées, en particulier en termes de complexité.

Métaplan

I. La structure d'un graphe Un graphe peut être vu comme un objet mathématiques servant à modéliser ou bien à une structure de données quelconque. Ici nous présentons le graphe vu comme un objet mathématiques sur lequel on peut rajouter des propriétés mais avec une représentation afin de pouvoir écrire des algorithmes.

- ⇝ *Définition* [1, p.74] : graphe non-orienté et graphe orienté
- ⇝ *Définition* [4, p.545] : un poids (rend l'arbre pondéré)
- ⇝ *Définition* : l'arité et le degré d'un nœuds
- ⇝ *Définition* [1, p.74] : graphe biparti
- ⇝ Représentation des graphes : matrice d'adjacence, liste d'adjacence
- ⇝ *Remarque* : il en existe d'autre

Complexité	Liste d'adjacence	Matrice d'adjacence
Spatiale	$O(\|S\| + \|A\|)$	$O(\|S\|^2)$
Renvoyer la liste des voisins	$O(1)$	$O(\|S\|)$
Tester si deux sommets sont voisins	$O(\|S\|)$	$O(1)$
Parcours en largeur (Algorithme)	$O(\|S\| + \|A\|)$ (agrégat)	$O(\|S\|^2)$
Parcours en profondeur (Algorithme **??**)	$O(\|S\| + \|A\|)$	$O(\|S\|^2)$
Composantes fortement connexes (Algorithmes)	$O(\|S\| + \|A\|)$	$O(\|S\|^2)$
2-connexité (Algorithme)	$O(\|S\| + \|A\|)$	$O(\|S\|^2)$
Bellman–Ford	$O(\|S\|\|A\|)$	$O(\|S\|^3)$
Dijkstra	$O((\|S\| + \|A\|) \log \|S\|)$	$O(\|S\|(\|S\| + \|A\|) \log \|S\|)$
Floyd–Warshall	⊠	$O(\|S\|^3)$
Prim	$O(\|A\| \log \|S\|)$	
Kruskal	$O(\|A\| \log \|A\|)$	

II. Problèmes d'accessibilité Puis-je aller jusqu'à ce point ? Dans un graphe répondre à cette question est un problème en temps polynomiale. On va même aller plus loin en cherchant si tous les sommets peuvent aller jusqu'à n'import quelle autre sommets : c'est la connexité. On finira par parler un peu de robustesse du réseau avec la 2-connexité. Application : réseau électrique.

A. *Parcourir un graphe [4, p.549]*

- ⇝ Parcours en largeur : principe + algorithme + complexité
- ⇝ *Application* : Test le caractère biparti d'un graphe
- ⇝ Parcours en largeur : principe + algorithme + complexité
- ⇝ *Application* : Tri topologique

B. *Problème de la connexité [9, p.419]* La vérification de la connexité d'un graphe : peut-on aller de tous sommets vers tous sommets est un problème pratique important (exemple : réseau électrique).

- ⇝ *Définition* [9, p.140] : Composantes connexes
- ⇝ *Remarque* : Algorithme de parcours
- ⇝ *Définition* : Composantes fortement connexes
- ⇝ Algorithme de Kosaraju (parcours en profondeur)
- ⇝ Algorithme de Tarjan (parcours en profondeur)

C. *La robustesse d'un graphe [9, p.448]* Dans des réseaux de télécommunication, par exemple, il est intéressant que si un des noeuds tombe en panne, le reste du réseau reste fonctionnel (connexe). On se place dans le cadre d'un graphe non-orienté connexe et on souhaite savoir si il est robuste devant une panne.

- ⇝ *Hypothèses* : Graphe non-orienté et connexe
- ⇝ *Définition* : 2-connexité
- ⇝ *Définition* : Point d'articulation
- ⇝ *Définition* : Composante
- ⇝ *Algorithme*
- ⇝ *Proposition* : Complexité

III. Problèmes de routage dans un graphe Comment puis-je aller jusqu'à ce point ? Le problème de routage vient généraliser le problème d'accessibilité : en plus de savoir si deux noeuds sont accessibles on veut connaître un chemin entre ces deux nœuds. Souvent ce chemin vérifie une propriété. Application : internet

A. *Le plus court chemin [4, p.495]* Le plus court chemin est un problème minimisant une certaine quantité. Souvent, on traite se problème dans un graphe pondéré. Il existe plusieurs variantes de ce problème : à destination unique, pour un couple de sommet ou entre tout couples de sommets. On va étudier des algorithmes résolvant ces variantes sous certaines hypothèses. Le problème du chemin le plus long est NP-complet.

- ⇝ *Définition* : problème du plus cours chemin dans un graphe
- ⇝ *Remarque* : dans un graphe non pondéré un parcours en largeur suffit
- ⇝ Algorithme de Bellman–Ford + complexité
- ⇝ Algorithme de Dijkstra + complexité
- ⇝ Algorithme de Floyd–Warshall + complexité
- ⇝ *Application* à la fermeture transitive (application aux composantes connexes)

B. *Vers les problèmes de couverture* Les problèmes que l'on présente maintenant ne sont pas tout à fait des problèmes de routage : on ne cherche pas à relier deux points, ni tout à fait des problèmes de couverture car on cherche un chemin (ou un cycle) dans le graphe.

- ⇝ *Définition* : problème du chemin eulérien
- ⇝ *Proposition* : critère de la présence d'un chemin eulérien
- ⇝ *Définition* : problème du chemin hamiltonien
- ⇝ *Proposition* : chemin hamiltonien est NP-complet
- ⇝ *Définition* : problème du TSP + version géométrique
- ⇝ *Proposition* : TSP (géométrique) est NP-complet
- ⇝ Approximation gloutonne + non ϵ-approximation

IV. Problèmes de couverture

Un problème de couverture est un problème qui cherche à couvrir notre graphe à l'aide d'un objet (soit tous les sommets, soit toutes les arêtes sont dans cet objets sans en violer les propriétés) : coloration, arbre couvrant, ... Les problèmes de clique permettent de modéliser des problèmes de serveurs web et les problèmes d'indépendant set modélisent les activités compatibles ou non.

A. *Arbre couvrant minimal [4, p.583]*

- ⇝ *Définition* : arbre couvrant minimal
- ⇝ Algorithme de Prim (principe + complexité)
- ⇝ Algorithme de Kruskal (principe + complexité)

B. *Problème de couverture par sommet*

- ⇝ *Définition* : Problème couverture par sommet (éclairage)
- ⇝ *Proposition* : Problème NP-complet
- ⇝ Algorithme d'approximation

C. *Coloration d'un graphe*

- ⇝ *Définition* : Problème de coloration
- ⇝ *Proposition* : Problème NP-complet
- ⇝ Restriction du problème pour le rendre P

IV. Problème de flots

Quelle est la capacité de mon graphe ? Le problème Max Cut est NP-complet

- ⇝ *Définition* : Problèmes Max flot / Min Cut
- ⇝ Algorithme de Ford–Fulkerson (principe + complexité)
- ⇝ *Remarque* : Dualité
- ⇝ *Application* biparti

Leçon 926 : Analyse des algorithmes : Complexité. Exemples.

Références pour la leçon

[1] Beauquier, Berstel et Chretienne, *Éléments d'algorithmique.*
[2] Carton, *Langages formels, calculabilité et complexité.*
[4] Cormen, *Algorithmique.*
[9] Froidevaux, Gaudel et Soria, *Types de données et algorithmes.*

Développements de la leçon

Étude d'Union-Find pour la complexité Algorithme de Dijkstra

Motivation

Défense

Lors de la conception, puis de l'étude d'un algorithme, deux notions sont extrêmement importantes :

— la correction de l'algorithme : fait-il ce que l'on souhaite ?

— l'efficacité de l'algorithme : à quelle vitesse s'exécute-t-il ? ; est-il optimal ?

La complexité (temporelle ou spatiale) intervient alors pour comparer deux algorithmes corrects répondant à la même question.

Ce qu'en dit le jury

Il s'agit ici d'une leçon d'exemples. Le candidat prendra soin de proposer l'analyse d'algorithmes portant sur des domaines variés, avec des méthodes d'analyse également variées : approche combinatoire ou probabiliste, analyse en moyenne ou dans le pire cas.

Si la complexité en temps est centrale dans la leçon, la complexité en espace ne doit pas être négligée. La notion de complexité amortie a également toute sa place dans cette leçon, sur un exemple bien choisi, comme union find (ce n'est qu'un exemple).

Métaplan

I. Quantifier la complexité Définir la complexité d'un algorithme n'est pas facile. Intuitivement la complexité d'un algorithme est un indicateur de la difficulté pour résoudre le problème traité par l'algorithme. Mais cette vue de l'esprits n'est pas simple à quantifier. Nous allons alors définir la complexité comme une fonction qui en fonction des entrées sur notre programme donnera la consommation d'une certaine ressource.

A. *Qu'est-ce que la complexité ?*

- ⇝ *Définition* : Données d'entrée d'un algorithme
- ⇝ *Définition* : Taille d'une entrée : $f : \{entree\} \to \mathbb{N}^k$
- ⇝ *Définition* : Complexité $f : \{\text{entrée}\} \to \{\text{ressource}\}$
- ⇝ *Remarque* : En pratique : unités de bases

B. *Mesurer la complexité [4, p.40]* On ne peut pas toujours calculer la complexité exacte (avec les constantes) car généralement, elle dépend de l'implémentation que nous utilisons. Pour cette même raison le calcul de la complexité exacte peut s'avérer inutile.

- ⇝ *Définition* : Données d'entrée d'un algorithme
- ⇝ *Définition* : Taille d'une entrée : $f : \{entree\} \to \mathbb{N}^k$
- ⇝ *Exemples* : Tri bulle $O(n^2)$; B-arbre $O(\log n)$
- ⇝ *Proposition* : L'ordre de croissance

C. *Différentes approches de la complexité [9, p.19]*

- ⇝ *Définition* : Complexité pire cas
- ⇝ *Définition* : Complexité meilleur cas
- ⇝ *Définition* : Complexité en moyenne (probabilité)
- ⇝ *Remarque* : Distribution uniforme : simplification de l'écriture mais pas toujours vrai Attention
- ⇝ *Définition* : Complexité constante

II. Techniques de calcul de la complexité Maintenant que nous avons donné un cadre à notre théorie de la complexité, nous souhaitons calculer les complexités d'algorithmes usuels. Pour cela, nous allons présenter quelques techniques de calcul élémentaires.

A. *Le calcul direct* On peut parfois calculer directement la complexité en dénombrant les opérations que l'on doit calculer. Efficace dans le cas d'algorithmes itératifs

- ⇝ *Exemple* : Recherche de maximum dans un tableau
- ⇝ *Exemple* [2, p.198] : CYK, complexité : $O(n^3)$
- ⇝ *Exemple* [1, p.389] : Marche de Jarvis, complexité : $O(hn)$

B. *La résolution de récurrence [1, p.20]* On peut parfois exprimer la complexité pour une donnée de taille n par rapport à une donnée de taille strictement inférieure. Résoudre l'équation ainsi obtenue nous donne la complexité. Efficace dans le cas d'algorithmes récursifs.

- ⇝ *Proposition* : Suite récurrente linéaire d'ordre 1
- ⇝ *Proposition* : Suite récurrente linéaire d'ordre 2
- ⇝ *Proposition* : Master theorem
- ⇝ *Remarque* : Ne capture pas toutes les équations
- ⇝ *Exemples* Factorielle et Euclide
- ⇝ *Exemple* Fibonacci
- ⇝ *Exemples* Tri fusion et Strassen
- ⇝ *Contre-exemple* : Tri rapide

C. *Le calcul de la complexité moyenne par l'espérance*

- ⇝ *Remarque* : Distribution uniforme
- ⇝ *Exemple* : Tri rapide randomisé
- ⇝ *Remarque* : Hypothèse d'une distribution uniforme : introduction d'erreurs.

III. Raffinement de l'étude de la complexité : la complexité amortie La complexité amortie n'est pas une complexité moyenne ! La complexité amortie est une amélioration de l'analyse dans le pire cas s'adaptant (ou calculant) aux besoins de performance des structures de données.

- ⇝ *Définition* : la complexité amortie
- ⇝ *Exemple* [4, p.428] : table dynamique

A. *Méthode de l'agrégat* Dans cette méthode, le coût amortie est le même pour toutes les opérations de la séquence même si elle contient différentes opérations

- ⇝ *Principe* [4, p.418] : On calcul la complexité dans le pire cas d'une séquence qu'on divise par son nombre d'opérations.
- ⇝ *Exemple* : Table dynamique
- ⇝ *Exemple* : Union find + Kruskal
- ⇝ *Exemple* : Balayage de Gram

B. *Méthode comptable* Cette méthode se différencie de la précédente en laissant la possibilité que toutes les opérations ait un coup amortie différent.

- ⇝ *Principe* : On attribut à chaque opération un crédit et une dépense
- ⇝ *Exemple* : table dynamique
- ⇝ *Exemple* : KMP

C. *Méthode du potentiel* Cette méthode a été popularisé lors de la preuve de la complexité amortie de la structure de données Union Find, implémentée à l'aide d'une forêt et des heuristiques qui vont bien. Elle est moins facile que les autres à mettre en place.

- ⇝ *Principe* [4, p.424] : Au lieu d'assigner des crédits à des opérations, on va associer une énergie potentielle φ à la structure elle-même.
- ⇝ *Exemple* : Table dynamique
- ⇝ *Exemple* : Union Find

IV. Amélioration de l'étude de la complexité Plusieurs pistes existe afin d'améliorer la complexité d'un algorithme : utiliser une structure de données plus adaptée, utiliser un peu plus de mémoire ou au contraire se souvenir de moins de choses, ... Cependant, quelques fois nous sommes capable de mettre une borne minimale sur la complexité d'une famille de problème : quand nous avons atteint cette borne on sait que nos algorithmes sont optimaux.

A. *Borne minimale de la complexité sur une classe de problème*

- ⇝ *Proposition* : Borne minimal d'un algorithme de tri
- ⇝ *Exemple* : Tri par insertion $O(n^2) \geq O(n \log n)$
- ⇝ *Exemple* : Tri fusion $O(n \log n)$ atteint cette borne
- ⇝ *Remarque* : Optimiser les constantes

B. *Utilisation de structures de données adaptées* Une piste pour améliorer un algorithme : utiliser une bonne structure de données

- ⇝ *Exemple* : Tri par tas
- ⇝ *Remarque* : Dépend de la manière dont on implémente la structure
- ⇝ *Exemple* : Représentation d'un graphe
- ⇝ *Remarque* : Utilisation de ces représentation
- ⇝ *Exemple* : Prim (liste d'adjacence) + Floyd Warshall (matrice d'adjacence)
- ⇝ *Remarque* : On peut changer de structure de données
- ⇝ *Exemple* : Dijstra

C. *Compromis espace/temps*

- ⇝ *Principe* : Compromis espace/temps
- ⇝ *exemple* : Fibonacci
- ⇝ *Principe* [4, p.338] : La mémoïsation
- ⇝ *Exemple* : Découpage de barre

Ouverture Évaluer la complexité d'un algorithme (hors implémentation) n'est pas une tâche facile. Souvent plusieurs astuces sont nécessaire pour trouver la meilleure borne sur notre complexité possible. Quelque fois, un approximation grossière de notre complexité (évaluation de la complexité pour le tri par tas) suffit.

Leçon 927 : Exemples de preuves d'algorithmes : correction et terminaison.

Références pour la leçon

[2] Carton, *Langages formels, calculabilité et complexité.*
[4] Cormen, *Algorithmique.*
[19] Nielson et Nielson, *Semantics with applications.*

Développements de la leçon

Correction de l'algorithme de Dijkstra
Complétude de la logique de Hoare

Motivation

Défense

Lors de l'écriture d'un programme, savoir s'il termine et s'il est correcte est une question légitime et difficile. C'est pourtant l'essence de l'informatique : l'étude de ces objets (comme les nombres pour les mathématiques). En effet, par les théorèmes d'indécidabilité du problème de l'arrêt et de Rice nous savons que répondre à ces questions est indécidable. Leur automatisation ne peut être complète et pour la plus part des problèmes nous devons prouver au moins une partie de ces résultats à la main : ce qui peut être long, fastidieux et même délicat.

Ce qu'en dit le jury

Le jury attend du candidat qu'il traite des exemples d'algorithmes récursifs et des exemples d'algorithmes itératifs.

En particulier, le candidat doit présenter des exemples mettant en évidence l'intérêt de la notion d'invariant pour la correction partielle et celle de variant pour la terminaison des segments itératifs.

Une formalisation comme la logique de Hoare pourra utilement être introduite dans cette leçon, à condition toutefois que le candidat en maîtrise le langage. Des exemples non triviaux de correction d'algorithmes seront proposés. Un exemple de raisonnement type pour prouver la correction des algorithmes gloutons pourra éventuellement faire l'objet d'un développement.

Métaplan

- ⇝ *Motivation* : Étude des programmes : essence de la science informatique.
- ⇝ *Motivation* : Savoir si un programme termine ou s'il est correct est une notion importante.
- ⇝ *Motivation* : Indécidabilité de l'automatisation.
- ⇝ *Motivation* : Automatisation de la vérification : vérifier pas simple (souvent) long et fastidieux.
- ⇝ *Motivation* : Programmer c'est prouver (correspondance de Curry–Howard).
- ⇝ *Définition* : Spécification
- ⇝ *Remarque* : Le plus difficile à donner

I. Terminaison Le théorème de l'arrêt et son indécidabilité pose les limites de l'algorithmique : on ne peut obtenir un algorithme qui permettent de prouver la terminaison de tous les algorithmes. On étudie alors quelques techniques que nous utilisons à la main. Gardons en tête la preuve des programmes parallèles.

- ⇝ *Définition* : Terminaison d'un algorithme
- ⇝ *Remarque* : Boucle POUR préserve la terminaison
- ⇝ *Exemple* : Terminaison et boucle TANT QUE
- ⇝ *Définition* : Problème ARRÊT
- ⇝ *Théorème* [2] : ARRÊT est indécidable
- ⇝ *Exemple* : Syracuse
- ⇝ *Conséquence* : Pas de procédure effective mais heuristique pour certaine classe

A. *Ensemble bien fondé* La notion d'ensemble bien fondé est un outil puissant pour montrer la terminaison.

- ⇝ *Définition* : Ensemble bien fondé
- ⇝ *Exemple* : $\mathbb{N}$
- ⇝ *Contre-exemple* : $\mathbb{Z}$
- ⇝ *Proposition* : $\mathbb{N}^2$ muni de l'ordre lexicographique est bien fondé
- ⇝ *Proposition* : Caractérisation de l'ordre lexicographique

B. *Terminaison des algorithmes récursifs* Dans le cas des algorithmes récursifs, l'ensemble bien fondé suffit pour montrer la terminaison de tels algorithmes.

- ⇝ *Théorème* : Terminaison des algorithmes récursifs
- ⇝ *Exemple* : Calcul du pgcd via l'algorithme d'Euclide
- ⇝ *Exemple* : Fonction d'Ackermann

C. *Terminaison des algorithmes itératifs* Dans le cas des algorithmes itératifs, la notion d'ensemble bien fondé ne s'applique pas directement : on utilise la notion d'invariant de boucle.

- ⇝ *Définition* : Variant de boucle
- ⇝ *Théorème* : Terminaison des algorithmes itératifs
- ⇝ *Exemple* : Division euclidienne

II. Correction Le théorème de Rice et son résultat d'indécidabilité pose les limites de l'algorithmique : on ne peut obtenir un algorithme qui permettent de prouver la correction de tous les algorithmes. On étudie alors quelques techniques que nous utilisons à la main. De plus, via la correspondance de Curry–Howard, on remarque que programmer c'est prouver et donc programmer c'est prouver la correction.

- ⇝ *Idée* : Le programme (algorithme) vérifie se spécification
- ⇝ *Définition* : Correction (totale et partielle)
- ⇝ *Exemple* : Tri topologique est totalement correct
- ⇝ *Exemple* : Syracuse est partiellement correct

A. *Correction des algorithmes récursifs* Commençons par l'étude de la correction des algorithmes récursifs.

- ⇝ *Proposition* : Vérification par les prédicats
- ⇝ *Exemple* : L'algorithme du pgcd est correct
- ⇝ *Théorème* : Correction des algorithmes récursifs
- ⇝ *Exemple* : Correction du tri par insertion

B. *Correction des algorithmes itératifs* Pour les algorithmes itératifs, nous pouvons utiliser la notion d'invariant de boucle. Cet invariant permet de prouver la correction des boucles : ce sont des étapes vers la correction de l'algorithme.

- ⇝ *Définition* : Invariant de boucle
- ⇝ *Théorème* : Utilisation des invariants de boucles
- ⇝ *Exemple* : Correction de Dijkstra

C. *Programmer c'est prouver* La correspondance de Curry–Howard, notamment, nous dit que programmer c'est prouver et que cette preuve est en réalité la correction d'un énoncé logique (comme une spécification peut être vue comme une formule logique, on prouve le programme).

Preuve avant l'algorithme Dans certains cas la preuve de correction d'un algorithme vient par l'analyse de ce problème et de ces propriétés. Plus exactement, on en tire un algorithme. Dans d'autres cas, les structures de données que nous utilisons donne la correction de l'algorithme.

- ⇝ *Méthode* : Analyse du problème
- ⇝ *Exemple* : Alignement optimaux
- ⇝ *Exemple* : Tri par tas
- ⇝ *Application* : Programmation dynamique
- ⇝ *Méthode* : Structure de données

La correspondance de Curry–Howard Curry et Howard ont prouvé grâce au λ-calcul simplement typé et la logique intuitionniste, ils prouvent que programmer c'est prouver. Ici, on ne évoque simplement quelques faits culturels autours de ce paradigme.

- ⇝ *Théorème* : Correspondance de Curry–Howard
- ⇝ *Application* : Requête SQL dans une base de données

III. Automatisation

Même si l'automatisation de telles preuves est indécidable, nous souhaitons en automatiser une partie. Pour la correction la logique de Hoare s'avère être un outils puissant. Nous allons la définir pour un langage jouet IMP muni de sa sémantique naturelle (nous savons que la sémantique à petits pas et la sémantique dénotationnelles sont équivalentes à celle-ci pour ce langage). Nous pouvons donc choisir n'importe laquelle mais la naturelle est la plus simple pour faire ce que l'on souhaite.

A. *Langage IMP [19, p.7]*

- ⇝ *Définition* : Langage IMP
- ⇝ *Exemple* : Factorielle

B. *Sémantique naturelle [19, p.20]*

- ⇝ *Définition* : Règle de la sémantique naturelle
- ⇝ *Définition* : Fonction déterministe
- ⇝ *Théorème* : Cette sémantique est déterministe
- ⇝ *Exemple* : Arbre de dérivation de factorielle

C. *Logique de Hoare [19, p.213]*

- ⇝ *Définition* : Triplet de Hoare
- ⇝ *Définition* : Conséquence sémantique et preuve
- ⇝ *Définition* : wlp
- ⇝ *Exemple* :Factorielle par la logique de Hoare
- ⇝ *Remarque* : Rôle du wlp dans l'automatisation
- ⇝ *Définition* : Règle de la logique de Hoare
- ⇝ *Théorème* : Correction de la logique
- ⇝ *Théorème* : Complétude de la logique

Leçon 928 : Problèmes NP-complets : exemples et réductions.

Références pour la leçon

[2] Carton, *Langages formels, calculabilité et complexité.*
[4] Cormen, *Algorithmique.*
[8] Floyd et Biegel, *Le langage des machines.*
[13] Kleinberg et Tardos, *Algorithm design.*
[16] Legendre et Schwarzentruber, *Compilation : Analyse lexicale et syntaxique du texte à sa structure en informatique.*
[17] Moret et Shapiro, *Algorithms from P to NP.*
[18] Garey et Johnson, *Computers and Intractability : A Guide to the Theory of NP-Completness.*
[23] Sipser, *Introduction to the Theory of Computation.*
[24] Stern, *Fondements mathématiques de l'informatique.*

Développements de la leçon

Le problème PSA est NP-complet

Approximation ou non du problème TSP

Motivation

Défense

Thèse de Cobham–Edmonds

Ce qu'en dit le jury

L'objectif ne doit pas être de dresser un catalogue le plus exhaustif possible ; en revanche, pour chaque exemple, il est attendu que le candidat puisse au moins expliquer clairement le problème considéré, et indiquer de quel autre problème une réduction permet de prouver sa NP-complétude.

Les exemples de réduction polynomiale seront autant que possible choisis dans des domaines variés : graphes, arithmétique, logique, etc. Si les dessins sont les bienvenus lors du développement, le jury attend une définition claire et concise de la fonction associant, à toute instance du premier problème, une instance du second ainsi que la preuve rigoureuse que

cette fonction permet la réduction choisie et que les candidats sachent préciser comment sont représentées les données.

Un exemple de problème NP-complet dans sa généralité qui devient P si on contraint davantage les hypothèses pourra être présenté, ou encore un algorithme P approximant un problème NP-complet.

Métaplan

I. *Des problèmes NP-complets*

A. *Les classes P et NP [2]*

- ⇝ *Définition* : Classe de problème P
- ⇝ *Définition* : Classe de problème NP
- ⇝ *Proposition* : Caractérisation
- ⇝ *Proposition* : Propriété de la classe P (stabilité)
- ⇝ *Définition* : Vérifieur

B. *La NP-complétude [2, 18]*

- ⇝ *Définition* : Réduction en temps polynomial
- ⇝ *Définition* : Problème NP-complet
- ⇝ *Théorème* : Cook
- ⇝ *Exemple* : Clique serveur web, communauté
- ⇝ *Proposition* : Conséquences de la réduction
- ⇝ *Remarque* : Caractérisation de P = NP
- ⇝ *Exemple* : Vertex cover éclairage
- ⇝ *Exemple* : Independant set activité compatible

C. *Technique de preuve de la NP-complétude [18, 8]*

- ⇝ Restriction d'un problème : 3SAT à SAT
- ⇝ Gadget : 3SAT à 3-Col
- ⇝ Utilisation des classes supérieures : Regexp non universelle
- ⇝ Remplacement locale : SAT à 3SAT
- ⇝ Autre : Séparabilité d'un automate

II. *La NP-complétude en pratique* Quand on tombe sur un problème NP-complet, on n'a pas tout perdu...

A. *Restreindre des entrées [13]* *Idée* : Perte d'expressivité du problème

- ⇝ Réduction de la taille de l'entrée : 2SAT / 2Col
- ⇝ Logique de Horn
- ⇝ *Application* [24] : Prolog
- ⇝ Passage aux réels : problème du sac à dos réel
- ⇝ *Application* : Analyse syntaxique

B. *Approximer les solutions [4, 17]* *Idée* : Perte de précision

- ⇝ *Définition* : Problème d'optimisation
- ⇝ *Définition* : Approximation
- ⇝ Problème TSP (glouton)
- ⇝ Approximation de Sac à dos (programmation dynamique)
- ⇝ *Remarque* : Lien avec les problèmes de décision
- ⇝ *Définition* : Schéma d'approximation
- ⇝ Approximation de TSPE (Diviser pour régner)

C. *Explorer l'ensemble des solutions [13]* *Idée* : Perte de temps ;)

- ⇝ Backtracking : DPLL
- ⇝ Branch and Bound
- ⇝ Programmation linéaire

Leçon 929 : Le lambda-calcul comme modèle de calcul pur. Exemples.

Références pour la leçon

[10] Hankin, *Lambda Calculi : A Guide for Computer Scientists.*
[11] Hindley et Seldin, *Lambda-Calculus and Combinators : An Introduction.*

Développements de la leçon

μ-récursivité $\Rightarrow$ λ-définissable — Théorèmes de Scott–Curry et Rice

Motivation

Défense

Ce qu'en dit le jury

Il s'agit de présenter un modèle de calcul : le lambda-calcul pur. Il est important de faire le lien avec au moins un autre modèle de calcul, par exemple les machines de Turing ou les fonctions récursives. Néanmoins, la leçon doit traiter des spécificités du lambda-calcul. Ainsi le candidat doit motiver l'intérêt du lambda-calcul pur sur les entiers et pourra aborder la façon dont il permet de définir et d'utiliser des types de données (booléens, couples, listes, arbres).

Métaplan

- $\rightsquigarrow$ Introduction d'une logique alternative à la théorie des ensembles par Alonzo Church.
- $\rightsquigarrow$ Modéliser et formaliser les fonctions récursives via le calcul qui est omniprésent.
- $\rightsquigarrow$ Définir la sémantique des langages purement fonctionnels.

I. Le λ-calcul : présentation du modèle Le λ-calcul est un modèle de calcul dont la syntaxe utilise les λ. On présentera une manière de calculer à l'aide de ce modèle : la β-réduction. Pour finir, on évoque leur lien avec les langages de programmation.

A. *La syntaxe du λ-calcul* La syntaxe du λ-calcul nous permet de définir l'équivalence via une substitution : cette relation d'équivalence définie par induction permettant de rassembler les λ-termes représentant le même calcul. Avant d'introduire la substitution, nous sommes obligé d'introduire la notion de longueur d'un terme car c'est avec cette notion que nous montrons que tout est bien défini.

- ⇝ *Définition* : Langage du λ-calcul
- ⇝ *Remarque* : Non-ambiguïté des λ-termes
- ⇝ *Définition* : Variable libre et lié
- ⇝ *Définition* : Terme clos
- ⇝ *Définition* : Substitution
- ⇝ *Exemple* : Un λ-terme
- ⇝ *Définition* : Sous-terme sous une partie
- ⇝ *Exemple* : Variable libre et lié dans ce terme
- ⇝ *Définition* : Longueur d'un terme
- ⇝ *Définition* : α-conversion

B. *La β-réduction* La β-réduction nous permet de calculer grâce aux λ-terme (ce n'est pas la sémantique du λ-calcul en tant que logique mais sa sémantique calculatoire). Elle correspond à une exécution d'un programme.

- ⇝ *Définition* : β-réduction
- ⇝ *Définition* : Forme normale
- ⇝ *Remarque* : On n'a pas toujours existence d'une telle forme
- ⇝ *Théorème* : Théorème de Church–Rosser
- ⇝ *Corollaire* : Unicité de la forme normale
- ⇝ *Définition* : β-équivalence
- ⇝ *Théorème* : Théorème de Church–Rosser
- ⇝ *Définition* : Combinateur de point fixe

C. *Stratégies de réduction et lien avec les langages de programmation* Le λ-calcul est le cœurs de langages de programmation fonctionnel comme LISP ou Haskel. Cependant, une stratégie de réduction est nécessaire et donne le type de langage et ses spécificités.

- ⇝ *Hypothèse* : La restriction de la β-réduction
- ⇝ *Définition* : λ-abstraction
- ⇝ *Remarque* : Lien entre λ-abstraction et forme normale
- ⇝ *Définition* : Réduction par nom
- ⇝ *Application* : LISP
- ⇝ *Définition* : Réduction par valeurs
- ⇝ *Remarque* : Peu efficace sauf mémoïsation
- ⇝ *Application* : Haskel

II. Un modèle de calcul puissant Le λ-calcul est un modèle de calcul puissant puisqu'il est aussi expressif que les machines de Turing. De plus, pour pouvoir implémenter des programmes dans les langages de programmation, il nous faut pouvoir implémenter des structures des données grâce à ce modèle de calcul.

A. *Encoder des données* Le λ-calcul permet d'encoder différentes structures de données nécessaire à la programmation. Il est important de faire passer l'intuition sur ces modèles.

- ⇝ *Structure de données* : Booléen
- ⇝ *Application* : Conditionnelle
- ⇝ *Structure de données* : Paire
- ⇝ *Structure de données* : Liste
- ⇝ *Application* : Fonction récursive
- ⇝ *Structure de données* : Entier de Church
- ⇝ *Structure de données* : Entier de Barendregt

B. *Thèse de Church* La thèse de Church est vérifiée par ce modèle de calcul. Nous avons donc une équivalence aux fonctions μ-récursives et aux machines de Turing.

- ⇝ *Définition* : Fonction λ-définissable
- ⇝ *Définition* : Fonction μ-récursives
- ⇝ *Théorème* : Fonction λ-définissable $\Leftrightarrow$ fonction μ-récursive (DEV $\Leftarrow$)
- ⇝ *Théorème* : Fonction μ-récursives $\Leftrightarrow$ fonction calculable par une machine de Turing
- ⇝ *Corollaire* : Fonction λ-définissable $\Leftrightarrow$ fonction calculable par une machine de Turing
- ⇝ *Application* : Thèse de Church

C. *Théorie de la décidabilité* Grâce à la thèse de Church nous nous ouvrons les portes de la théorie de la décidabilité

- ⇝ *Définition* : Séparabilité
- ⇝ *Théorème* : Théorème de Scott–Curry
- ⇝ *Définition* : Fonction non triviale
- ⇝ *Corollaire* : Théorème de Rice
- ⇝ *Application* : Indécidabilité de l'existence d'une forme normale

Ouverture La correspondance de Curry–Howard pour le λ-calcul simplement typé.

Leçon 930 : Sémantique des langages de programmation. Exemples.

Référence pour la leçon

[19] Nielson et Nielson, *Semantics with applications.*

Développements de la leçon

Équivalence petit pas et grand pas

Complétude de la logique de Hoare

Motivation

Défense

Qu'est qu'un programme ? Un programme est une suite d'instructions qui à partir d'une entrée vérifiant une précondition donne une sortie vérifiant une postcondition. Ces deux conditions donnent alors la spécification de notre programme.

On s'intéresse alors à plusieurs problème. Le premier qui est le plus naturel est de savoir si un programme est correct, c'est-à-dire si celui-ci vérifie sa spécification (quand on lui donne une entrée vérifiant la précondition nous renvoie une sortie qui vérifie la postcondition). Le suivant est de savoir si un programme est équivalent à un autre : si on donne la même entrée aux deux programmes alors on obtient le même résultat. Le troisième problème et le dernier que nous abordons est la preuve de la correction d'une transformation de programmes qui intervient notamment lors de la compilation. On cherche alors à automatiser ces preuves en formalisant et en axiomatisant la sémantique d'un programme (en lui donnant un certain sens).

La sémantique d'un langage de programmation c'est un modèle mathématique permettant de raisonner sur le comportement attendu des programmes de ce langages. Une sémantique peut prendre des formes mathématiques variées et nous en présentants quelques unes ici.

Ce qu'en dit le jury

L'objectif est de formaliser ce qu'est un programme : introduction des sémantiques opérationnelle et dénotationnelle, dans le but de pouvoir faire des preuves de programmes, des preuves d'équivalence, des preuves de correction de traduction.

Ces notions sont typiquement introduites sur un langage de programmation (impératif) jouet. On peut tout à fait se limiter à un langage qui ne nécessite pas l'introduction des CPOs

et des théorèmes de point fixe généraux. En revanche, on s'attend ici à ce que les liens entre sémantique opérationnelle et dénotationnelle soient étudiés (toujours dans le cas d'un langage jouet). Il est aussi important que la leçon présente des exemples d'utilisation des notions introduites, comme des preuves d'équivalence de programmes ou des preuves de correction de programmes.

Métaplan

Motivation : Raisonner sur le comportement attendu des programmes d'un langage via des outils mathématiques ; plusieurs outils peuvent décrire un même langage.

I. Le langage IMP [19, p.7] On travaille sur un langage jouet impératif mettant en place la notion de mémoire via les affectation et la boucle while.

A. *Syntaxe du langage IMP*

- ⇝ *Définition* : Syntaxe et grammaire du langage IMP
- ⇝ *Exemple* : Factorielle et Fibonacci (arbre de syntaxe)

B. *Sémantique dénotationnelle des expressions* Les sémantiques, ou plus exactement les fonctions sémantiques que l'on définies ici, sont très générique, mais à l'aide des expressions légales on les spécialise comme on le souhaite. On suppose que toutes les variables sont définies et on les initialise à 0 ; même si elle pourrait être partielle, cela nous permet de faire l'impasse sur les cas d'erreurs du à la portée des variables.

Idée : Les sémantiques dénotationnelles modélisent le comportement par une fonction mathématiques.

- ⇝ *Définition* : Sémantique dénotationnelle des expressions
- ⇝ *Remarque* :$State$ totale et représentation avec liste finie.
- ⇝ *Remarque* : Totalité des fonctions $\mathcal{A}$ et $\mathcal{B}$
- ⇝ *Contre-exemple* : Ajout d'une opération binaire de division.

II. Sémantiques opérationnelles

Idée : Les sémantiques opérationnelles modélisent le comportement par un système de transitions dont les règles $\langle S, s\rangle \to s'$ modélisent le fait "l'exécution de S sur l'état s termine dans l'état s'".

A. *Sémantique opérationnelle à grand pas (sémantique naturelle) [19, p.20]*

- ⇝ *Définition* : Règles inductives.
- ⇝ *Définition* : Arbre de dérivation.
- ⇝ *Proposition* : Sémantique déterministe.
- ⇝ *Définition* : Équivalence de langage.
- ⇝ *Contre-exemple* : Instruction choose et IMP est non-déterministe
- ⇝ *Définition* : Fonction sémantique.
- ⇝ *Application* : IMP est déterministe.
- ⇝ *Exemple* : while b do $S \equiv$ if b then $(S;$ while b do $S)$ else skip (utile dans le DEV 1) et $S_1;(S_2;S_3) \equiv (S_1;S_2);S_3$ sont sémantiquement équivalents
- ⇝ *Contre-exemple* : $x := 3; x := x \oplus 1 \not\equiv x := x \oplus 1; x := 3$
- ⇝ *Limite* : La sémantique à grand pas ne permet pas d'exprimer les interruptions de programmes du à la division par 0 par exemple.

B. *Sémantique opérationnelle à petit pas (sémantique structurelle) [19, p.33]* La sémantique à petit pas ne traite de la terminaison que par la fonction sémantique qui n'est pas toujours facile à manipuler. La sémantique à petit pas vient combler ce manque. L'avantage est dans sa structure horizontale qui représente l'évolution temporelle de notre calcul et une structure verticale qui représente les sous calcul que nous devons effectuer.

Idée : La sémantique à petits pas est plus fine que la précédente : elle se concentre sur les étapes de l'exécution (affectations et tests).

- ⇝ *Définition* : Règles inductives.
- ⇝ *Définition* : Séquence de dérivation.
- ⇝ *Exemple* : Factorielle
- ⇝ *Proposition* : Langage déterministe sous la sémantique à petits pas
- ⇝ *Application* : IMP est déterministe
- ⇝ *Définition* : Fonction sémantique

- ⇝ *Proposition* : Propriétés de comportement de la sémantique à grand pas sur la séquence (DEV 1)
- ⇝ *Contre-exemple* : $< S_1; S_2, s >\Rightarrow^* < S_2, s' >$ n'implique pas $< S_1, s >\Rightarrow^* s'$.
- ⇝ *Définition* : Équivalence de langage ⇝ *Contre-exemple* : $S_1; S_2 \not\equiv S_2; S_1$.
- ⇝ *Exemples* $\mathsf{while}\ b\ \mathsf{do}\ S \equiv \mathsf{if}\ b\ \mathsf{then}\ (S;\ \mathsf{while}\ b\ \mathsf{do}\ S)\ \mathsf{else}\ \mathsf{skip}$; $S_1; (S_2; S_3) \equiv (S_1; S_2); S_3$

C. *Équivalence des deux sémantiques* On a défini deux sémantiques opérationnelles permettant de décrire le sens d'un programme, on voudrait maintenant savoir si l'une d'entre elles est plus expressive. En réalité, les deux sémantiques que nous avons définies sont équivalentes : elles expriment la même chose.

- ⇝ *Théorème* : Équivalence petits pas et grands pas
- ⇝ *Corollaire* : Caractérisation de la terminaison

III. Sémantique dénotationnelle [19, p.91]

- ⇝ *Idée* : On définie la sémantique dénotationnelle pour les instructions du langage IMP (modélisation par les fonctions compositionnelles). On utilise alors la théorie du point fixe afin de définir la sémantique du while.
- ⇝ *Remarque* : On commence par définir la sémantique pour l'ensemble des instructions. Puis, on montre que cette sémantique est bien définie en détaillant la théorie du point fixe.

A. *Sémantique dénotationnelle* On essaye (et on réussit) à faire passer l'intuition de pourquoi on en a besoin (car elle est uniquement compositionnelle, il nous faut donc des sous-terme strict ce que la boucle while ne peut nous garantir) et de la complexité de répondre à cette question (le nombre de choix possibles pour une instruction données car les contraintes sont uniquement sur les instructions qui terminent) ; on met le type de Fix qui se lit le plus petit points fixes.

- ⇝ *Définition* : Règles : introduction de la fonction F pour le while
- ⇝ *Exemple* : instruction simple avec deux points fixes possible
- ⇝ *Définition* : Équivalence de langage ⇝ *Exemple* : $S \equiv S; \mathsf{skip}$

B. *Théorie du point fixe* Permet de montrer que $FixF$ est bien définie et donc que notre sémantique est bien définie

- ⇝ *Définition* : Ordre partiel (exemple : $State \hookrightarrow State$) ⇝ *Définition* : Chaîne
- ⇝ *Définition* : CCPO ⇝ *Proposition* : $\bot$ est la borne inférieure de $State \hookrightarrow State$ sémantique
- ⇝ *Définition* : Fonctions monotones et continues ⇝ *Proposition* : $F(g)$ est continue
- ⇝ *Théorème* : Kleene (point fixe) ⇝ *Conséquence* : Bonne définition de la sémantique

C. *Équivalence des deux sémantiques*

- ⇝ *Théorème* : Équivalence avec la sémantique opérationnelle à petits pas
- ⇝ *Corollaire* : Équivalence avec la sémantique opérationnelle à grands pas

IV. Sémantique axiomatique (Hoare partiel) [19, p.213] Afin de faciliter la preuve automatiser, nous allons définir une logique particulière qui va représenter la sémantique.

- ⇝ *Idée* : Cette sémantique facilite la preuve de programmes et son automatisation. On se base sur les systèmes de déduction en logique que nous sommes capable de bien manipuler.
- ⇝ *Correction partielle* : on donne des garanties que si le programme termine.

A. *Triplet de Hoare* On définit la syntaxe et la sémantique de cette logique. Dans le Nielson [19], on définie la logique de Hoare comme une sur-logique : on met n'importe quelle logique dans ces prédicat. La définition syntaxe, sémantique, déduction est alors un peu bancale : il faut au moins en avoir conscience. On La sémantique que nous utilisons est celle à grands pas mais on pourrait le faire avec n'importe quelle autre (par l'équivalence).

- ⇝ *Définition* : Syntaxe des prédicats ⇝ *Définition* : Sémantique des prédicats
- ⇝ *Définition* : Triplet de Hoare ⇝ *Exemple* : Triplet pour la factorielle
- ⇝ *Définition* : Validité ⇝ *Définition* : Équivalence de programme
- ⇝ *Exemple* : Programme équivalent

B. *Logique de Hoare partielle* On présente ici le système de déduction de cette logique : cette sémantique.

- ⇝ *Définition* : Règles d'inférence de la logique ⇝ *Exemple* : Preuve d'un programme simple
- ⇝ *Définition* : Arbre de preuve ⇝ *Exemple* : Preuve d'un programme simple
- ⇝ *Proposition* : pour tout S et P, on a $\vdash_p \{P\}S\{\mathsf{True}\}$ ⇝ *Définition* : Prouvé sémantiquement

⇝ *Exemple* : S; skip et S prouvé sémantiquement

C. *Correction et complétude* La logique que nous avons définie ne fait pas n'importe quoi.

⇝ *Théorème* : Correction de la logique

⇝ *Définition* : Précondition la plus faible wlp.

⇝ *Lemmes* : Propriétés sur cette précondition

⇝ *Théorème* : Complétude de la logique

Ouverture

⇝ Les sémantiques offrent de nombreuses sémantiques.

∼ Certification d'un compilateur

∼ Vérification de la sécurité d'un programme

⇝ D'autres sémantiques : sémantique à continuation pour la sémantique opérationnelle à petit pas.

Leçon 931 : Schéma algorithmiques. Exemples et application.

Référence pour la leçon

[1] Beauquier, Berstel et Chretienne, *Éléments d'algorithmique.*
[4] Cormen, *Algorithmique.*
[13] Kleinberg et Tardos, *Algorithms Design.*
[17] Moret et Shapiro, *Algorithms from P to NP.*
[23] Sipser, *Introduction to the Theory of Computation*

Développements de la leçon

Étude de l'alignement optimal de deux mots Algorithme de Dijkstra

Motivation

Défense

La méthode naïve, le brute force n'est pas toujours efficace. Il existe plusieurs types d'algorithmes permettant de résoudre un problème. Selon la nature du problème (optimisation, récursif, ...) ces approches peuvent être plus ou moins efficaces. Nous allons en étudier quelques unes ici, et plus particulièrement des paradigmes de partitionnement et d'exploration.

Ce qu'en dit le jury

Cette leçon permet au candidat de présenter différents schémas algorithmiques, en particulier «diviser pour régner », programmation dynamique et approche gloutonne. Le candidat pourra choisir de se concentrer plus particulièrement sur un ou deux de ces paradigmes. Le jury attend du candidat qu'il illustre sa leçon par des exemples variés, touchant des domaines différents et qu'il puisse discuter les intérêts et limites respectifs des méthodes. Le jury ne manquera pas d'interroger plus particulièrement le candidat sur la question de la correction des algorithmes proposés et sur la question de leur complexité, en temps comme en espace.

Métaplan

⇝ Brute force : exemple et contre-exemple

I. Algorithmes de partitionnement

Une première approche de l'algorithmique est de partager le problème en sous problème dont ceux-ci sont à priori plus simple à calculer.

A. *Diviser pour régner [4, p.59]*

Diviser pour régner est plutôt un paradigme récursif. Le principe est de partager la résolution du problème en le divisant puis en recombinant les sous-instance ainsi résolu.

- ⇝ Paradigme
- ⇝ *Théorème* [1] : Master théorème
- ⇝ *Contre-exemple* : tri rapide
- ⇝ *Application* [17, p.480] : Approximation de TSPE
- ⇝ *Limites* : infinité de l'espace de la solution : pile d'appel (problème d'optimisation NP-complet)
- ⇝ *Limites* : infinité structurelle : recoupement (Fractale, Fibonacci)
- ⇝ *Exemple* : Tri fusion / enveloppe convexe
- ⇝ *Exemple* : Tri fusion
- ⇝ *Application* [23, p.335] : Théorème de Savitch

B. *Programmation dynamique [4, p.333]*

Cette approche vient rectifier les lacunes du à l'infinité structurelle du paradigme diviser pour régner en stockant dans des tables les résultats intermédiaires. Il s'applique généralement aux problèmes d'optimisation.

- ⇝ *Problème* : Quand doit-on l'utiliser ?
- ⇝ *Exemple* : Distance d'édition
- ⇝ *Exemple* : CYK
- ⇝ *Exemple* : Découpe de barre
- ⇝ Paradigme
- ⇝ *Exemple* : PLSC
- ⇝ *Définition* : Memoisation
- ⇝ *Application* [13, p.644] : Approximation de Sac à dos

C. *Algorithmes glouton : une heuristique*

Cette approche permet d'optimiser des problèmes en considérant une approche locale : elle vient rectifier les limite dû à l'infinité de l'espace des solutions. C'est une première heuristique simple qui peut parfois donner de bons résultats.

- ⇝ Paradigme
- ⇝ *Exemple optimal* : Prim Kruskal
- ⇝ *Exemple optimal* : Dijkstra
- ⇝ *Application* : Analyse syntaxique (LL(k), LR(k))
- ⇝ *Application* : Approximation de Set cover, TSP
- ⇝ *Application* : Approximation de TSP

II. Algorithmes d'exploration

Lorsque l'ensemble des solutions devient vraiment très important : le partage choisi par les algorithmes de partitionnement ne sont pas nécessairement les meilleurs. On utilise alors des algorithmes d'explorations qui d'une certaine manière partitionne l'espace des solutions mais sont l'exécution se base sur l'exploration d'un arbre. Remarque : on a aussi des algorithmes d'exploration sur les graphes. Dans les deux cas, on peut être amené à explorer l'ensemble de l'arbre qui nous sert de base.

A. *Backtracking*

Cette approche de l'exploration est plus efficace dans le cadre d'un algorithme d'exploration sur un problème de décision (où on cherche à construire une instance). On a des améliorations de cette approche nous permettant de ne pas explorer des parties de l'arbre (backjumping avec CDCL). On fait un parcours en profondeur de notre arbre des possibles.

- ⇝ Principe
- ⇝ *Application à la compilation* : analyse syntaxique pour toute grammaire
- ⇝ *Remarque* : On peut faire mieux avec CYK
- ⇝ *Application à la logique* : DPLL

B. *Branch and Bound*

Cette approche de l'exploration est plus efficace dans le cadre d'un algorithme d'exploration sur un problème d'optimisation. On se donne une fonction de coût, une heuristique facile à calculer, (qui majore ou minore celle du problème) qui nous permet de classer les solutions en fonction de leur affinité avec la solution.

- ⇝ Principe
- ⇝ Exemple

Leçon 932 : Fondements des bases de données relationnelles.

IMPASSE

Motivation

Ce qu'en dit le jury

Le cœur de cette nouvelle leçon concerne les fondements théoriques des bases de données relationnelles : présentation du modèle relationnel, approches logique et algébrique des langages de requêtes, liens entre ces deux approches.

Le candidat pourra ensuite orienter la leçon et proposer des développements dans des directions diverses : complexité de l'évaluation des requêtes, expressivité des langages de requête, requêtes récursives, contraintes d'intégrité, aspects concernant la conception et l'implémentation, optimisation de requêtes...

Bibliographie

[1] D. Beauquier, J. Berstel, and P. Chrétienne. *Eléments d'algorithmique*. Manuels informatiques Masson. Masson, 1992.

[2] O. Carton. *Langages formels, calculabilité et complexité*. Vuibert, 2008.

[3] R. Cori and D. Lascard. *Logique mathématique, tome 1*. Masson, 1994.

[4] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Algorithmique, 3ème édition*. Dunod, 2010.

[5] M. Crochemore, C. Hancart, and L. Lecroq. *Algorithmique du texte*. Vuibert, 2001.

[6] M. Crochemore and W. Rytter. *Text Algorithms*. Oxford University Press, Inc., New York, NY, USA, 1994.

[7] J. Duparc. *La logique pas à pas*. Presse polytechnicienne et université romandes, 2015.

[8] R. Floyd and R. Biegel. *Le langage des machines*. International Thomson Publishing, 1994.

[9] C. Froidevaux, M.C. Gaudel, and M. Soria. *Types de données et algorithmes*. McGraw-Hill, 1990.

[10] C. Hankin. *Lambda Calculi : A Guide for Computer Scientists*. Graduate texts in computer science. Clarendon Press, 1994.

[11] J. R. Hindley and J. P. Seldin. *Lambda-Calculus and Combinators : An Introduction*. Cambridge University Press, 2 edition, 2008.

[12] M. Huth and M. Ryan. *Logic in Computer Science : Modelling and Reasoning About Systems*. Cambridge University Press, New York, NY, USA, 2004.

[13] J. Kleinberg and E. Tardos. *Algorithm Design*. Pearson international Edition, 2006.

[14] D. Knuth. *The Art of Computer Programming, vol 3*. Addison-Wesley, 1998.

[15] R. Lassaigne and M. de Rougemont. *Logique et fondements de l'informatique. Logique du 1^{er} ordre, calculabilité et λ-calcul.* Hermes, 1993.

[16] R. Legendre and F. Schwarzentruber. *Compilation : Analyse lexicale et syntaxique du texte à sa structure en informatique*. Reference Sciences. Ellipses, 2015.

[17] B.M.E. Moret and H.D. Shapiro. *Algorithms from P to NP*. The benjamin / Cumming Publishing Company, 1990.

[18] D.S. Johnson M.R. Garey. *Computers and Intractability : A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1979.

[19] H. R. Nielson and F. Nielson. *Semantics with application*. Springer, 2007.

[20] C.H. Papadimitriou. *Computational complexity*. Pearson, 1993.

[21] C. Raffalli, R. David, and K. Nour. *Introduction à la Logique, Théorie de la démonstration (2nd édition)*. Sciences Sup. Dunod, 2004.

[22] J. Sakarovitch. *Éléments de la théorie des automates*. Vuibert informatique, 2003.

[23] M. Sipser. *Introduction to the Theory of Computation*. Cengage learning, 1133187811.

[24] J. Stern. *Fondements mathématiques de l'informatique*. Ediscience internationnal, 1990.

[25] A. Turing and J.-Y. Girard. *La machine de Turing*. Source du savoir, Seuil, 1991.

[26] P. Wolper. *Introduction à la calculabilité*. Dunod, 2006.