LYCÉE MARCEAU
Promo MP 2023-2024

Cours d'informatique Tronc Commun

ANTHONY PREUX

Table des matières

BASES DE DONNÉES : INTRODUCTION

Sommaire

I Introduction

I.1 Définition

Une base de données est un ensemble structuré de données. Il peut être vu comme une énorme "bibliothèque" dont chacun des livres stockerait des données telles que des nombres, des noms, des booléens, des dates des mots, etc...
La base est enregistrée sur un support accessible par ordinateur, capable de satisfaire simultanément plusieurs utilisateurs. Une base de données doit pouvoir être lue rapidement et on doit pouvoir en extraire rapidement les données.

I.2 Première approche

Une première idée qu'on pourrait se faire d'une base de données consiste à enregistrer toutes les données dans un même tableau. Par exemple, dans le cas suivant, où on s'intéresse aux notes de Mathématiques d'un groupe d'élèves (le devoir comporte deux parties notées sur 10) :

Nom	Prenom	Devoir	type	coefficient	part_1	part_2
ALBIN	Felix	1	DS	2	1	4
CHERRAJ	Assia	1	DS	2	10	9
GOMES	Quentin	1	DS	2	3	8
LEGER	Maia	1	DS	2	10	5
DECOUPY	Victor	1	DS	2	4	6
GOMIS	Jean-Marie	1	DS	2	1	10
POUBANNE	Lisa	1	DS	2	9	3
HAMELIN	Dimitri	1	DS	2	8	4
HAMELIN	Axel	1	DS	2	5	7
LE GUILLOUS	Cyrielle	1	DS	2	1	4
ALBIN	Felix	2	DM	1	7	3
CHERRAJ	Assia	2	DM	1	9	3
GOMES	Quentin	2	DM	1	9	2
LEGER	Maia	2	DM	1	2	4
DECOUPY	Victor	2	DM	1	4	4
GOMIS	Jean-Marie	2	DM	1	8	1
POUBANNE	Lisa	2	DM	1	4	2
HAMELIN	Dimitri	2	DM	1	6	7
HAMELIN	Axel	2	DM	1	7	5
LE GUILLOUS	Cyrielle	2	DM	1	8	7
ALBIN	Felix	3	Interro	1	5	4
CHERRAJ	Assia	3	Interro	1	9	6
GOMES	Quentin	3	Interro	1	2	3
LEGER	Maia	3	Interro	1	2	8
DECOUPY	Victor	3	Interro	1	2	8

Tableau de données

Ce tableau n'est pas simple à exploiter :

- certains noms peuvent être mal orthographié à certaines lignes car ils sont souvent répétés ;
- on peut mettre ou non l'accent, à "Félix" par exemple ;
- on peut avoir écrit "DM ou "Devoir maison", "interro" ou "interrogation" et pourtant le devoir est du même type.

S'il y a que très peu de données, il est possible de traiter les données à la main. Par exemple, retrouver toutes les notes de "Félix". Mais si le jeu de données devient trop important ?
Imaginez-vous la situation suivante : un braquage a eu lieu à Saint-Pourçain-sur-Sioule. Les voleurs sont partis dans une voiture. Un témoin est formel : c'est une Renault rouge coupé dont la plaque commençait par un "A" et avec le sigle de la concession automobile "Bony Automobiles". La police appelle Renault pour que la firme lui donne toutes les voitures correspondant à ce signalement. Pensez-vous qu'une personne va regarder dans la base de données les véhicules vendus par Renault un par un pour trouver tous les véhicules correspondants ? Pour rappel, Renault vend plus de 2 millions de véhicule par an...

Une base de données doit avoir plusieurs bonnes propriétés :

- **Intégration :** les données ne doivent pas être redondantes et rester cohérentes (par exemple, le changement du nom d'une personne doit être rapide et se faire partout où cela est nécessaire).
- **Indépendance :** plusieurs propriétés sont regroupées par ce terme : l'indépendance physique : un changement de disque dur pour stocker les données ne doit pas être ressenti ; l'indépendance logique : si on change la logique de l'organisation des données, on n'a pas besoin de changer les programmes d'application ; indépendance du moyen d'accès aux données : l'utilisateur demande simplement d'accéder aux données et on lui renvoie les données demandées.
- **Disponibilité :** les données doivent pouvoir être traitées rapidement et par plusieurs utilisateurs.
- **Sécurité :** protection contre les pannes mais aussi l'incohérence des données (si une lettre est rentrée dans la colonne des dates par exemple et que le format est du type 03/04/12).

Les bases de données sont stockées au sein d'une SGBD (Système de Gestion de Bases de Données). C'est un logiciel visant à simplifier la tâche des usagers en proposant un niveau d'abstraction (c'est-à-dire que le fonctionnement peut s'adapter à différents types de données). Il doit gérer les représentations physique (les "tables" de données) et logique (les liens entre les tables) des données. Pour cela il utilisera un langage dédié à la manipulation des données.

Plusieurs SGDB existent : ORACLE, ACCESS, OPEN OFFICE etc..., mais ils utilisent tous le langage normalisé SQL (Structured Query Language) créé en 1974. Plusieurs SGDB (libres et gratuites...) ont d'ailleurs un nom construit sur cet acronyme : SQLite, MySQL, sqliteman..

II Base de données relationnelle

II.1 Exemple

Dans une base de données relationnelle, les informations seront contenues dans plusieurs tableaux comme suit :

rowid	élèves	Devoir	part_1	part_2
1	1	1	1	4
2	**1**	**2**	**7**	**3**
3	1	3	5	4
4	1	4	4	9
5	1	5	9	3
6	1	6	3	8
7	1	7	6	6
8	1	8	8	5
9	1	9	9	4
10	1	10	9	7
11	1	11	3	3
12	1	12	4	9
13	1	13	7	9
14	1	14	6	8
15	2	1	10	9
16	2	2	9	3
17	2	3	9	6
18	2	4	3	9
19	2	5	8	1
20	2	6	6	1
21	2	7	7	5
22	2	8	6	10
23	2	9	2	7
24	2	10	1	8
25	2	11	1	9
26	2	12	9	9
27	2	13	5	7
28	2	14	5	8
29	3	1	3	8
30	3	2	9	2
31	3	3	2	3
32	3	4	4	2
33	3	5	6	2
34	3	6	4	5

notes

rowid	Nom	Prenom	année	homme
1	ALBIN	Felix	1998	1
2	CHERRAJ	Assia	1998	0
3	GOMES	Quentin	1997	1
4	**LEGER**	**Maia**	**1998**	**0**
5	DECOUPY	Victor	1997	1
6	GOMIS	Jean-Marie	1997	1
7	POUBANNE	Lisa	1998	0
8	HAMELIN	Dimitri	1998	1
9	HAMELIN	Axel	1998	1
10	LE GUILLOUS	Cyrielle	1998	0

identité

rowid	type_num	coefficient
1	1	2
2	**2**	**1**
3	3	1
4	1	1
5	2	2
6	1	3
7	2	1
8	3	2
9	1	1
10	2	2
11	1	3
12	2	1
13	1	1
14	2	1
15	1	2
16	1	3

devoir_config

rowid	type
1	DS
2	**DM**
3	Interro

devoir_type

Au lieu d'un seul tableau on en a 4. Ils contiennent des données sur des sujets différents. Un tableau contient les notes des élèves sur chaque partie du devoir. Un autre contient l'identité des élèves et quelques informations sur eux (année de naissance, genre...). Un troisième tableau contient les informations relatives aux devoirs et un dernier les types de devoirs qu'on peut trouver. On remarque tout de suite qu'il n'y a pas plus de redondance : les mots ne sont plus répétés, il n'y a plus de risque de mal les orthographier. Les données prennent moins de place qu'avec la méthode précédente : stocker un entier est moins couteux que le nom et prénom d'un élève. On appelle ces tableaux **des tables ou des relations.**

Exercice 1.1

Que représente la ligne 22 du tableau "notes" ?

E

II.2 Vocabulaire

- **tuple :** les lignes sont les **entrées** ou **tuples** ou encore **entités**. Ces tuples ne sont pas nommés. Exemple : (1,ALBIN,Felix,1998,1).
- **attribut :** les colonnes sont appelées les **champs**, les **attributs** ou encore les **propriétés** : dans

l'exemple précédent, les attributs de la table *identité* sont *rowid* (numéro de la ligne), *Nom*, *Prenom*, *année*, *homme*.

- **table :** tableau contenant les données. C'est un sous-ensemble du produit cartésien de plusieurs attributs. Par exemple, la table identité contient les éléments (1,ALBIN,Felix,1998,1), (2,CHERRAJ,Assia,1998,0), etc... On l'appelle également relation (langage algébrique) ou type entité.

- **domaine :** sous SQL, un domaine est décrit par un type (INT,FLOAT,etc...), auquel on peut ajouter des contraintes (tous différents, positif, etc...). Les types les plus usuels sont :

  → Le type VARCHAR (ou CHAR VARYING) permet de coder des chaînes de longueur variables, mais la longueur maximale est fixée. Par exemple VARCHAR(20) désigne le type des chaines alphanumériques formées d'au plus 20 caractères.

  → CHAR désigne le type des chaînes alphanumériques de longueur fixe, par exemple CHAR(10) désigne le type des chaînes alphanumériques formées de 10 caractères.

  → NUMERIC (ou DECIMAL ou DEC) qui permet de coder des nombres décimaux de façon exacte.

  → Le type FLOAT qui permet de coder des décimaux en base 2 de façon approchée (mais les calculs seront plus rapides qu'avec le type NUMERIC).

  → INT (ou INTEGER) qui permet de coder des entiers.

  → TEXT qui permet de coder des textes (éventuellement longs) de longueur indéterminée.

  → DATE et TIME pour la date et l'heure.

  Il faut se souvenir que chaque attribut possède un type qui lui est propre (éventuellement son domaine peut être restreint par des contraintes). Par exemple, dans la relation *identité*, l'attribut *homme* est un booléen (1 si c'est un homme, 0 si c'est une femme).

  Remarque 1.2. *Les attributs ne sont pas ordonnés, par contre ils possèdent tous un nom et peuvent être appelées par celui-ci. Les lignes ne sont pas ordonnées également mais elles doivent être toutes différentes.*

- **clé primaire :** Une clé primaire permet d'identifier de manière univoque chaque tuple d'une relation. C'est un sous-ensemble des attributs de la relation qui permet de distinguer chaque entrée de la relation, mais telle que, si on retire n'importe lequel des attributs de ce sous-ensemble, alors on ne peut plus distinguer certaines entrées de la relation. Par exemple dans la relation identité de notre exemple, à laquelle on aurait enlevé l'attribut *rowid*, on aurait pu choisir le sous-ensemble constitué des attributs *Nom* et *Prenom*, mais deux personnes peuvent avoir le même nom dans certains cas. Le sous-ensemble contenant que *rowid* convient dans tous les cas. En pratique la clé est souvent limitée à un seul attribut, qui prend des valeurs distinctes pour chaque entrée de la relation. Cet attribut sera identifié par le terme Cle ou Identifiant ou Id (éventuellement suivi de l'objet à identifier : Villes, Nom, Étudiant, Professeur.)

- **clé étrangère :** Une clé étrangère permet de référencer les entrées de certaines relations avec la clé (primaire) d'une autre relation : dans notre exemple, l'attribut *élèves* de la relation *notes* est une clé étrangère qui référence la clé primaire de la relation *identité*.

- **schéma relationnel :** Le schéma d'une relation est la donnée du nom de la relation et de chacun de ses attributs (avec ou sans mention de leurs domaines). Dans nos exemples, nous avons les deux schémas : *identité(rowid,Nom,Prénom,année,homme)*. Notez que l'on utilisera un symbole pour distinguer les clés primaires…

- **schéma de base de données** : est la donnée des différentes relations de la base, chacune avec ses attributs, et des références entre clés étrangères et clés primaires des différentes relations d'une base de données.

Exercice 1.3

Effectuer le schéma de base de données de l'exemple de ce cours :

E

Définition 1.4: entité

On appelle entité un objet (concret ou abstrait) qui peut être reconnue distinctement et qui est caractérisée par son unicité.

Exemple 1.5

un élève (Albin félix, Poubanne Lisa, etc..), la copie d'un élève à un devoir (la copie de Lisa au devoir 3), etc ...

Définition 1.6: type entité

Un type entité désigne un ensemble d'entités qui possèdent une sémantique et des propriétés communes.

Exemple 1.7

des élèves, des copies de devoir (plus précisément, les notes de ces copies) , des notes d'interrogation, etc..

Définition 1.8: association

On appelle **association** un lien entre plusieurs entités

Exemple 1.9

L'**obtention** de la note de 5/20 par Albin Félix au premier devoir

Définition 1.10: type association

Un type association désigne un ensemble d'associations qui possèdent les mêmes caractéristiques. Le type association décrit un lien entre plusieurs types entités. Les associations de ce type association lient les entités des tables.

Exemple 1.11

L'obtention des notes aux devoirs entre le type entité **élèves** et le type entité **notes de devoirs**.

Définition 1.12: cardinalité

On appelle **cardinalité** d'un lien reliant un type association à un type entité le nombre de fois minimal et maximal d'interventions d'une entité du type entité dans une association du type association.

Remarque 1.13. *Même si on connait le nombre maximal de fois qu'on utilise une entité (par exemple ici, si on sait qu'il y a 12 devoirs, on utilise donc au plus 12 fois un élève) on ne marquera pas ce nombre mais la variable n. En effet, comme les tables ne sont pas figés, ce nombre pourrait changer dans le temps (par exemple si on effectue un 13^e devoir, alors on devrait passer de 12 à 13).*

Remarque 1.14. *La cardinalité minimale admise est soit 0 (on peut ne pas utiliser toutes les entités) soit* 1 *(on doit toutes les utiliser au moins une fois). La cardinalité maximale admise est soit* 1 *(on l'utilise au plus une fois), soit n (on l'utilise autant de fois qu'on veut). Il y a donc* 4 *possibilités :*

- *0,1 : une entité peut être utilisée au plus qu'une seule fois et peut ne pas être utilisé.*
- *0,n : elle peut être utilisée sans limitation.*
- *1,1 : elle peut être utilisée une et une seule fois.*
- *1,n : elle doit être utilisée au moins une fois et sans limitation.*

Exemple 1.15

Le type association **obtention des notes aux devoirs** liant le type entité **élèves** au type entité **notes aux devoirs** et de cardinalité 0,n pour les élèves, 1,1 pour les notes de devoirs.

BASES DE DONNÉES : OPÉRATIONS SUR UNE TABLE

Sommaire

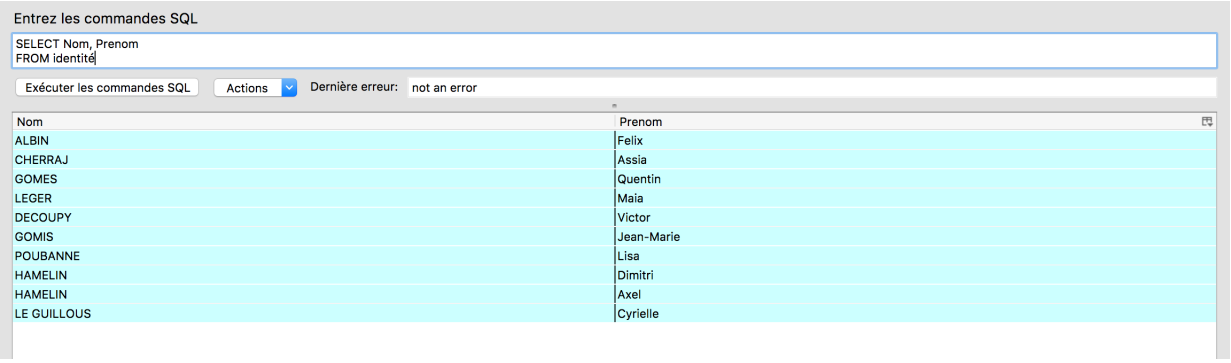
I Opérations ensemblistes

I.1 Projection

La projection est l'opération permettant de ne choisir que certains attributs (donc certaines colonnes) d'une relation. Par exemple lorsque l'on projette la relation *identité* en ne gardant que les attributs *Nom* et *Prénom,* la requête SQL est :

SQL

```
SELECT Nom, Prenom
FROM identité;
```



Cette requête s'écrit en algèbre relationnelle :

$$\pi_{Nom,Prenom}(\text{identité})$$

Remarque 2.1. *Le caractère ";" permet de fermer une requête SQL.*

Pour éviter des redondances dans les résultats, on peut utiliser le mot-clé SELECT DISTINCT au lieu de SELECT.

La clause LIMIT est à utiliser dans une requête lorsqu'on souhaite spécifier le nombre maximum de lignes dans le résultat. Par exemple,

SQL

```
SELECT Nom, Prenom
FROM identité
LIMIT 4;
```

ne donnera que les 4 premières lignes du résultat précédent.

La clause OFFSET permet d'ignorer les premières lignes. Par exemple,

SQL

```
SELECT Nom, Prenom
FROM identité
LIMIT 4 OFFSET 3;
```

renverra les lignes de LEGER à POUBANNE (les trois premières lignes sont ignorées et on récupère 4 lignes maximum).

Enfin, le mot clé ORDER BY permet d'ordonner le résultat selon un attribut.

SQL

```
SELECT Nom, Prenom
FROM identité
ORDER BY Nom;
```

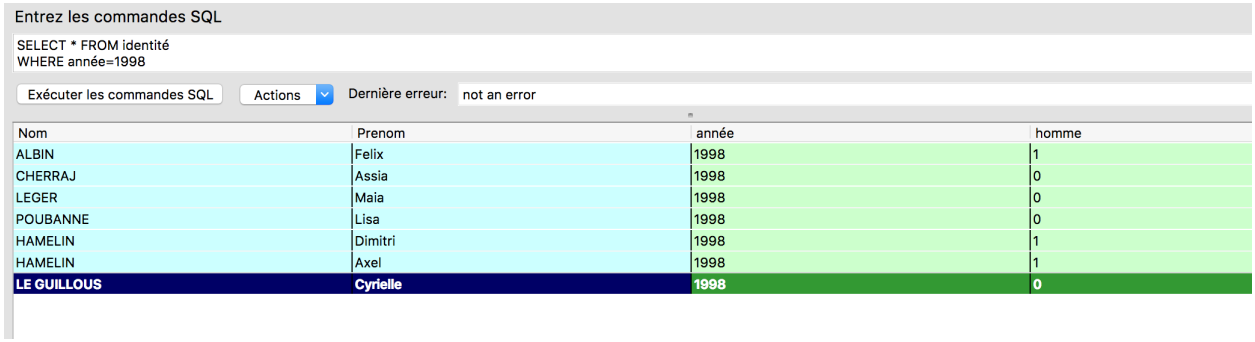
Les lignes seront ordonnées selon les noms des élèves.

I.2 Sélection

La **sélection** est l'opération permettant de ne choisir que les entrées (donc les lignes) d'une relation vérifiant une certaine condition. Par exemple si l'on cherche les élèves nés en 1998, la requête SQL est :

SQL

```
SELECT *
FROM identité
WHERE année=1998;
```



Nom	Prenom	année	homme
ALBIN	Felix	1998	1
CHERRAJ	Assia	1998	0
LEGER	Maia	1998	0
POUBANNE	Lisa	1998	0
HAMELIN	Dimitri	1998	1
HAMELIN	Axel	1998	1
LE GUILLOUS	**Cyrielle**	**1998**	**0**

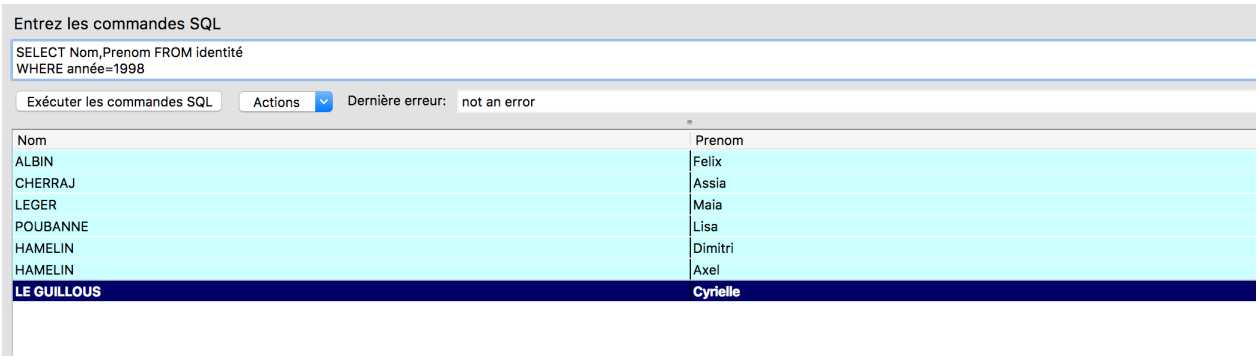
Cette requête s'écrit en algèbre relationnelle :

$$\sigma_{\text{année}=1998}(\text{identité})$$

On peut évidemment combiner projection et sélection :

SQL

```
SELECT Nom, Prenom
FROM identité
WHERE année=1998;
```



Nom	Prenom
ALBIN	Felix
CHERRAJ	Assia
LEGER	Maia
POUBANNE	Lisa
HAMELIN	Dimitri
HAMELIN	Axel
LE GUILLOUS	**Cyrielle**

Cette requête s'écrit en algèbre relationnelle :

$$\pi_{Nom,Prenom}\left(\sigma_{\text{année}=1998}(\text{identité})\right)$$

I.3 Union

La réunion est l'opération algébrique classique. Lors d'une requête, les attributs projetés doivent être les mêmes. Exemple : si on recherche les élèves nés en 1998 ou les hommes :

SQL

```
SELECT *
FROM identité
WHERE année=1998
UNION
SELECT *
FROM identité
WHERE homme=1;
```

```
1 SELECT *
2 FROM identité
3 Where année=1998
4 UNION
5 SELECT *
6 FROM identité
7 Where homme=1
```

Table | Formulaire

Nombre de lignes chargées : 10

	Nom	Prenom	année	homme
1	ALBIN	Felix	1998	1
2	CHERRAJ	Assia	1998	0
3	DECOUPY	Victor	1997	1
4	GOMES	Quentin	1997	1
5	GOMIS	Jean-Marie	1997	1
6	HAMELIN	Axel	1998	1
7	HAMELIN	Dimitri	1998	1
8	LE GUILLOUS	Cyrielle	1998	0
9	LEGER	Maia	1998	0
10	POUBANNE	Lisa	1998	0

Cette requête s'écrit en algèbre relationnelle :

$$\sigma_{\text{année}=1998}(\text{identité}) \cup \sigma_{\text{homme}=1}(\text{identité})$$

Remarque 2.2. *On peut effectuer cette exemple autrement avec une sélection :*

SQL

```
SELECT *
FROM identité
WHERE année=1998 OR homme=1;
```

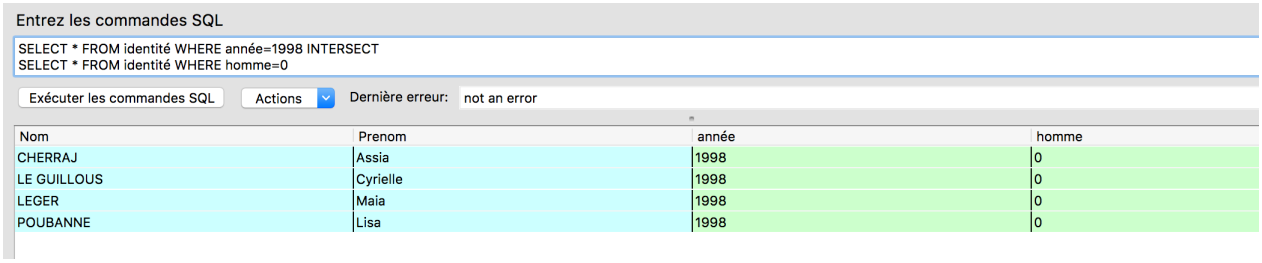
Cependant, on verra dans le chapitre suivant que la réunion peut s'effectuer entre tables différentes.

I.4 Intersection

L'intersection est l'opération algébrique classique. Lors d'une requête, les attributs projetés doivent être les mêmes. Exemple : si on recherche les élèves nés en 1998 qui sont des femmes :

SQL

```
SELECT *
FROM identité
WHERE année=1998
INTERSECT
SELECT *
FROM identité
WHERE homme=0;
```



Cette requête s'écrit en algèbre relationnelle :

$$\sigma_{\text{année}=1998}(\text{identité}) \cap \sigma_{\text{homme}=0}(\text{identité})$$

Remarque 2.3. *On peut effectuer cette exemple autrement avec une sélection :*

SQL

```
SELECT *
FROM identité
WHERE année=1998 AND homme=0;
```

Cependant, on verra dans le chapitre suivant que l'intersection peut s'effectuer entre tables différentes.

I.5 Différence ensembliste

La différence ensembliste est l'opération algébrique classique. Elle concerne les entrées uniquement. Lors d'une requête, les attributs projetés doivent être les mêmes. Exemple : si on recherche les élèves nés en 1998 qui ne sont pas des femmes :

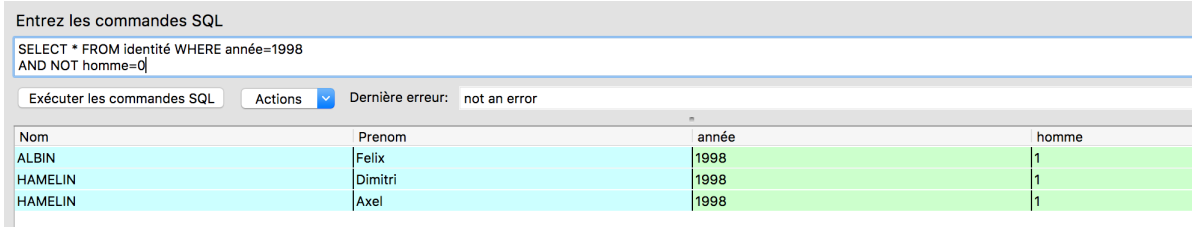
SQL

```
SELECT *
FROM identité
WHERE année=1998
EXCEPT
SELECT *
FROM identité
WHERE homme=0;
```

qu'on peut également obtenir avec :

SQL

```
SELECT *
FROM identité
WHERE année=1998
AND NOT homme=0;
```



Cette requête s'écrit en algèbre relationnelle :

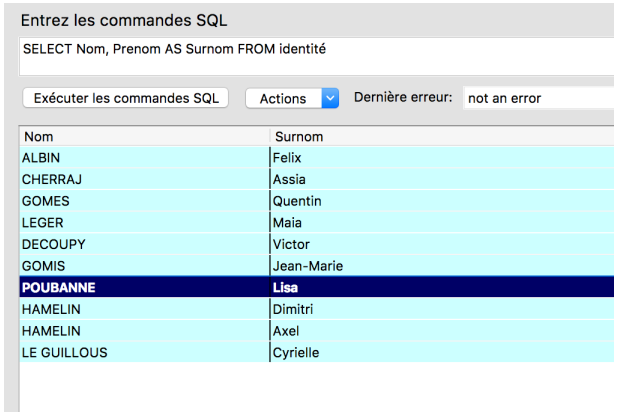
$$\sigma_{\text{année}=1998}(\text{identité}) \setminus \sigma_{\text{homme}=0}(\text{identité})$$

I.6 Renommage

Le renommage est l'opération algébrique consistant à donner un autre nom à une relation ou à un attribut.

SQL

```
SELECT Nom, Prenom AS Surnom
FROM identité;
```



Cette requête s'écrit en algèbre relationnelle :

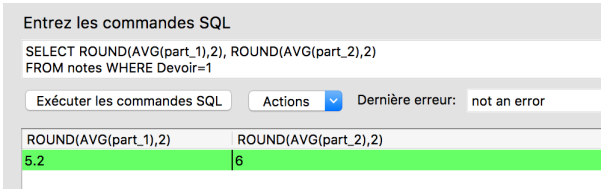
$$\rho_{\text{Prenom}\to\text{Surnom}}\left(\pi_{\text{Nom,Prenom}}(\text{identité})\right)$$

I.7 Fonctions d'agrégation

Les fonctions d'agrégation permettent de faire des calculs sur un groupe d'entrées sélectionnées. Par exemple, si on veut calculer la moyenne sur la partie 1 et 2 du premier devoir :

SQL

```
SELECT ROUND(AVG(part_1),2), ROUND(AVG(part_2),2)
FROM notes
WHERE Devoir=1;
```



Voici quelques fonction d'agrégation usuelle :

→ AVG calcule la moyenne ;

→ COUNT calcule le nombre de tuple sélectionnés

→ MAX et MIN pour le maximum et minimum

→ SUM pour le calcul de la somme

Certaines fonctions dont la valeur de retour est booléenne et qui servent à des critères de sélections d'autres requêtes : IN, ALL, ANY (ou SOME), EXISTS, NOT EXISTS. Elles apparaissent dans des sous-requêtes que nous verrons plus loin.

La fonction ROUND sur un attribut permet d'arrondir un résultat numérique. Cette fonction permet soit d'arrondir sans utiliser de décimal pour retourner un nombre entier (c'est-à-dire : aucun chiffre après la virgule), ou de choisir le nombre de chiffre après la virgule.

I.8 Opérateurs

Les opérateurs +, /, - et * peuvent également être utilisés. Par exemple part_1+part_2 permet de sommer la colonne part_1 et la colonne part_2 tuple par tuple (ce qui permet d'obtenir la note globale de chaque devoir de chaque élève).

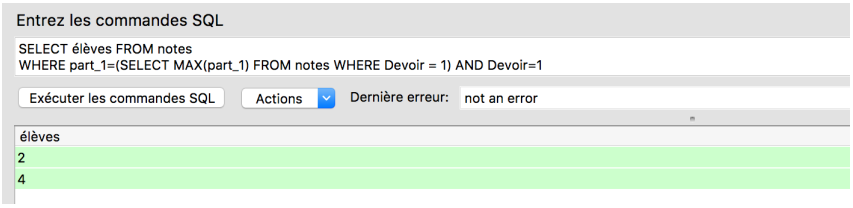
II Sous-requêtes

Il peut-être intéressant d'utiliser le résultat d'une requête R_1 à l'intérieur du critère d'une autre requête R_2 : on dit que la requête R_1 est une **sous-requête** de la requête R_2.

Par exemple, si on recherche dans la table *notes* les élèves ayant eu la meilleure note à la première partie du premier devoir :

SQL

```
SELECT élèves FROM notes
WHERE part_1=(SELECT MAX(part_1) FROM notes WHERE Devoir = 1) AND Devoir=1;
```

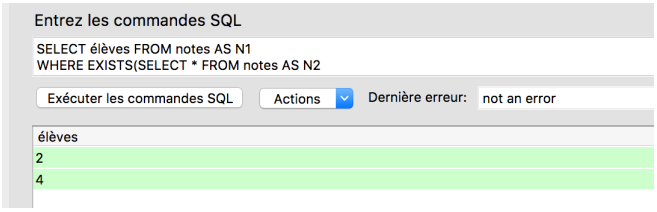


Utilisations du résultat d'une sous-requête avec ALL, EXISTS, ANY ou IN

On souhaite connaitre les élèves ayant eu la même note à la première partie du devoir 1 que l'élève numéro 2 :

SQL

```
SELECT élèves FROM notes AS N1
WHERE EXISTS(SELECT * FROM notes AS N2
WHERE N1.part_1=N2.part_1 AND N2.élèves=2 AND N1.Devoir=1 AND N2.Devoir=1);
```



En utilisant la fonction EXISTS, qui fournit le booléen Vrai si le résultat de la sous-requête est non vide et Faux sinon. Dans l'exemple précédent, on va donc chercher pour chaque élève de la table N1 si la table N2 est non vide, la table N2 est une table où on sélectionne que l'élève 2 et le devoir 1 et si la note de la partie 1 est la même que celle de l'élève de la table N1, elle est donc non vide que si l'élève de la table N1 a la même note que l'élève 2.

De même il existe la fonction *ALL* ou *ANY* qui signifie "vrai pour toutes les entrée" et "vrai pour au moins une des entrées" mais ce n'est pas le cas dans SQLite... Il faut passer par le négation *NOT* etc...

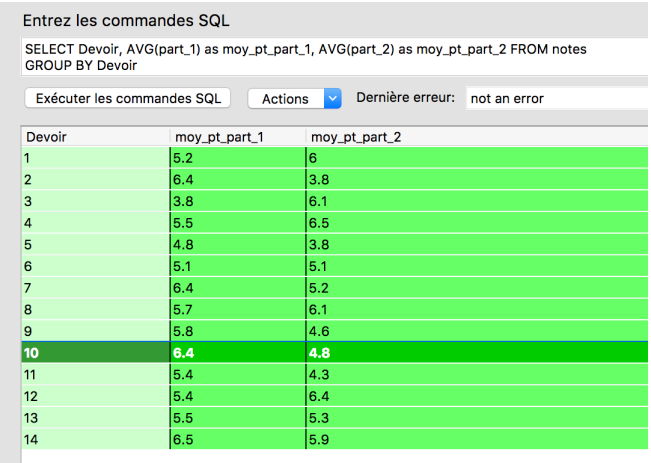
III Groupement de Tuples

III.1 GROUP BY

Cette fonction permet de partitionner les tuples renvoyés par une requête SELECT en différents groupes pour appliquer une fonction d'agrégation à chaque groupe.
Par exemple, si on veut calculer les moyennes de chaque partie de chaque devoir :

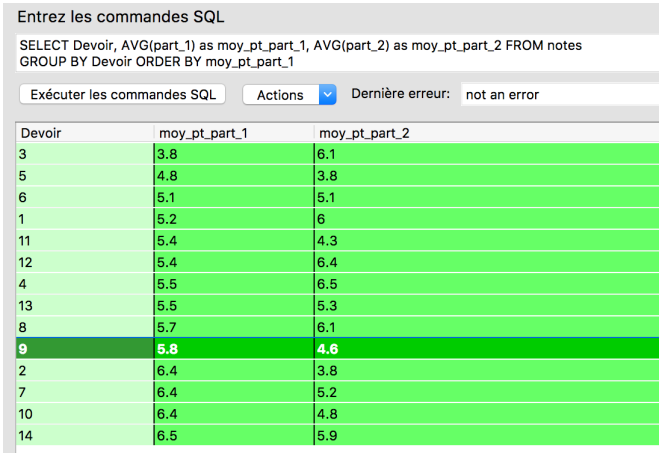
SQL

```sql
SELECT Devoir, AVG(part_1) as moy_pt_part_1, AVG(part_2) as moy_pt_part_2
FROM notes
GROUP BY Devoir;
```



Devoir	moy_pt_part_1	moy_pt_part_2
1	5.2	6
2	6.4	3.8
3	3.8	6.1
4	5.5	6.5
5	4.8	3.8
6	5.1	5.1
7	6.4	5.2
8	5.7	6.1
9	5.8	4.6
10	6.4	4.8
11	5.4	4.3
12	5.4	6.4
13	5.5	5.3
14	6.5	5.9

De plus, la fonction *ORDER BY* permet d'ordonner les lignes selon un critère :



Devoir	moy_pt_part_1	moy_pt_part_2
3	3.8	6.1
5	4.8	3.8
6	5.1	5.1
1	5.2	6
11	5.4	4.3
12	5.4	6.4
4	5.5	6.5
13	5.5	5.3
8	5.7	6.1
9	5.8	4.6
2	6.4	3.8
7	6.4	5.2
10	6.4	4.8
14	6.5	5.9

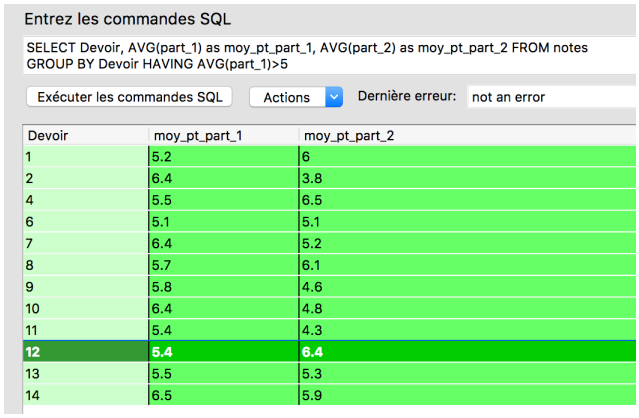
On rajoute DESC après l'attribut si on veut ordonner du plus grand au plus petit (par exemple ORDER BY moy_pt_part_1 DESC)

III.2 HAVING

Cette fonction effectue la même sélection que WHERE mais elle s'applique au groupe sélectionné via GROUP BY et non aux tuples eux-mêmes. Par exemple, si on veut garder dans l'exemple précédent les devoirs où la moyenne de la première partie dépasse 5 :

SQL

```sql
SELECT Devoir, AVG(part_1) as moy_pt_part_1, AVG(part_2) as moy_pt_part_2
FROM notes
GROUP BY Devoir
HAVING AVG(part_1)>5;
```



Devoir	moy_pt_part_1	moy_pt_part_2
1	5.2	6
2	6.4	3.8
4	5.5	6.5
6	5.1	5.1
7	6.4	5.2
8	5.7	6.1
9	5.8	4.6
10	6.4	4.8
11	5.4	4.3
12	5.4	6.4
13	5.5	5.3
14	6.5	5.9

BASES DE DONNÉES : OPÉRATIONS ENTRE TABLES

Sommaire

Dans ce chapitre, on reprend les tables présentées le chapitre 2, auxquelles on a rajouté une table nommé identité2 que voici :

rowid	Nom	Prenom
1	LE GUILLOUS	Cyrielle
2	MOLLE	Robin
3	BLANCHET	Lorry
4	LIN	Yiman

I Opérations ensemblistes entre tables

I.1 Union

L'union est une opération ensembliste que l'on peut réaliser sur des relations/tables à condition qu'elles aient la même structure, c'est-à-dire le même nombre d'attribut. Par exemple, si je veux trouver tous les noms des élèves apparaissant dans l'une des deux tables :

SQL

```
SELECT Nom, Prenom
FROM identité
UNION
Select Nom, Prenom
From identité2;
```

ce qui donne :

Nom	Prenom
ALBIN	Felix
BLANCHET	Lorry
CHERRAJ	Assia
DECOUPY	Victor
GOMES	Quentin
GOMIS	Jean-Marie
HAMELIN	Axel
HAMELIN	Dimitri
LE GUILLOUS	Cyrielle
LEGER	Maia
LIN	Yiman
MOLLE	Robin
POUBANNE	Lisa

I.2 Intersection

On peut également réaliser des intersections entre relations/tables ayant la même structure. Par exemple, si on cherche le (Nom,Prenom) des élèves apparaissant dans les deux tables :

SQL

```
SELECT Nom, Prenom
FROM identité
INTERSECT
Select Nom, Prenom
From identité2;
```

ce qui donne :

Nom	Prenom
LE GUILLOUS	Cyrielle

I.3 Produit cartésien

Il a le même principe qu'en algèbre, c'est-à-dire que si vous avez un ensemble A de tuples $(a_1, ..., a_n)$ et un ensemble B de tuples $(b_1, ..., b_m)$, le produit cartésien de ces deux ensemble donne l'ensemble des tuples $(a_1, ..., a_n, b_1, ..., b_m)$ avec $(a_1, ..., a_n)$ dans A et $(b_1, ..., b_m)$ dans B. Voici un exemple :

SQL

```
SELECT *, devoir_conf.rowid as numéro_devoir
FROM Devoir_conf
JOIN Devoir_type;
```

ce qui donne :

	type_num	coefficient	type	numéro_devoir
1	1	2	DS	1
2	1	2	DM	1
3	1	2	Interro	1
4	2	1	DS	2
5	2	1	DM	2
6	2	1	Interro	2
7	3	1	DS	3
8	3	1	DM	3
9	3	1	Interro	3
10	1	1	DS	4
11	1	1	DM	4
12	1	1	Interro	4
13	2	2	DS	5
14	2	2	DM	5
15	2	2	Interro	5
16	1	3	DS	6
17	1	3	DM	6
18	1	3	Interro	6
19	2	1	DS	7
20	2	1	DM	7
21	2	1	Interro	7
22	3	2	DS	8
23	3	2	DM	8

Remarque 3.1. *Le produit cartésien n'a pas vraiment d'intérêt ainsi. Dans le cas précédent, on remarque que cela aurait un intérêt de sélectionner que les triplets ou l'attribut type représente bien le type de devoir du devoir concerné. Autrement dit, il faut sélectionner les triplets où le numéro de la ligne de Devoir_type correspond au numéro indiqué dans l'attribut type_num. Le devoir numéro 1 a pour valeur dans type_num : 1, c'est donc un DS, il faut donc garder le triplet (1,2,DS,1) et supprimer tous les triplets de la forme (a,b,c,1), c'est le principe de la jointure que nous verrons un peu plus loin.*

I.4 Division cartésienne

La division cartésienne est l'opération inverse du produit cartésien : diviser une relation A par une relation B consiste à trouver une relation C contenant tous les n-uplets t tels que pour tout n-uplet t' de B, le n-uplet (t, t') soit dans A. Il n'y a pas d'opérateur permettant de faire cette opération directement. On la crée via les autres opérations selon les cas.

I.5 Jointure

L'opération de jointure est une opération permettant de joindre deux tables en une seule en suivant un critère donné. Formellement, c'est la composition d'un produit cartésien des deux tables qu'on veut joindre et de la sélection des n-uplets de la table issue du produit qui vérifient un critère.

Par exemple, si on reprend le produit cartésien précédent, en conservant les triplets qui on un sens :

SQL

```sql
SELECT *, devoir_conf.rowid as numéro_devoir
FROM Devoir_conf
JOIN Devoir_type ON devoir_type.rowid=devoir_conf.type_num;
```

	type_num	coefficient	type	numéro_devoir
1	1	2	DS	1
2	2	1	DM	2
3	3	1	Interro	3
4	1	1	DS	4
5	2	2	DM	5
6	1	3	DS	6
7	2	1	DM	7
8	3	2	Interro	8
9	1	1	DS	9
10	2	2	DM	10
11	1	3	DS	11
12	2	1	DM	12
13	1	1	DS	13
14	2	1	DM	14
15	1	2	DS	15
16	1	3	DS	16

Ce qui permet d'indiquer directement le type de chaque devoir (l'attribut type_num correspond bien à l'attribut type).

Deuxième exemple, si on veut retrouver le nom et le prénom des élèves avec les notes :

SQL

```
SELECT Nom, Prenom, devoir, part_1, part_2
FROM notes
JOIN identité ON identité.rowid=notes.élèves;
```

ce qui donne :

Nom	Prenom	Devoir	part_1	part_2
ALBIN	Felix	1	1	4
ALBIN	Felix	2	7	3
ALBIN	Felix	3	5	4
ALBIN	Felix	4	4	9
ALBIN	Felix	5	9	3
ALBIN	Felix	6	3	8
ALBIN	Felix	7	6	6
ALBIN	Felix	8	8	5
ALBIN	Felix	9	9	4
ALBIN	Felix	10	9	7
ALBIN	Felix	11	3	3
ALBIN	Felix	12	4	9
ALBIN	Felix	13	7	9
ALBIN	Felix	14	6	8
CHERRAJ	Assia	1	10	9
CHERRAJ	Assia	2	9	3
CHERRAJ	Assia	3	9	6
CHERRAJ	Assia	4	3	9
CHERRAJ	Assia	5	8	1
CHERRAJ	Assia	6	6	1
CHERRAJ	Assia	7	7	5
CHERRAJ	Assia	8	6	10
CHERRAJ	Assia	9	2	7
CHERRAJ	Assia	10	1	8

II Exercices

Exercice 1 Joindre les attributs Nom et Prenom à la table : notes. Puis, à l'aide de l'opérateur + permettant de sommer deux colonnes, calculer les notes de chaque élève à chaque devoir.

Exercice 2 Trouver, pour chaque élève, la plus mauvaise note qu'ils ont obtenue à un devoir. La table doit renvoyer le nom, le prénom, le numéro du devoir et la note obtenue.

Exercice 3 Trouver, pour chaque élève, la plus mauvaise note qu'ils ont obtenue pour chaque type de devoir. La table doit renvoyer le nom, le prénom, le numéro du devoir, le type de devoir et la note obtenue.

Exercice 4 Calculer la moyenne obtenue par chaque étudiant selon le type de devoir. La table doit renvoyer le nom, le prénom, le type du devoir, la moyenne obtenue sur ce type.

Exercice 4 Calculer la moyenne obtenue par chaque étudiant. La table doit renvoyer le nom, le prénom et la moyenne obtenue.

Exercice 5 Le professeur, légèrement compatissant, décide de ne pas compter dans la moyenne de l'élève sa plus mauvaise note dans chaque type de devoir. Recommencer l'exercice précédent en tenant compte de cette information. Pour cela, on pourra utiliser l'opérateur IN qui renvoie TRUE si un n-uplet est dans une table, FALSE sinon.

CONTENEURS ET FONCTIONNEMENT DES DICTIONNAIRES

Sommaire

I Généralités

Nous allons voir dans les grandes lignes comment sont implémentés les principaux conteneurs de Python.

I.1 Définition

Définition 4.1: conteneurs

Les conteneurs sont des objets abstraits qui permettent de stocker des objets informatiques sous forme organisée. Ces conteneurs peuvent être implémentés de différentes façons, et en conséquence, suivre certaines règles. Ces différentes manières de les implémenter entrainent que les actions qui en sont liées ont des complexités en temps et en espace différentes.

I.2 Propriétés générales

Trois propriétés sont fondamentales pour un conteneur :

- **l'accès :** c'est-à-dire la manière d'accéder aux différents éléments du conteneur ;
- **le stockage :** c'est-à-dire la manière dont sont stockés les différents éléments ;
- **le parcours :** c'est-à-dire la manière de parcourir les différents éléments du conteneur.

Différentes méthodes peuvent être implémentées :

- créer un conteneur vide ;
- ajouter des objets au conteneur ;
- supprimer un ou des objets du conteneur ;
- accéder aux objets du conteneur ;
- connaître le nombre d'objet du conteneur.

II Les principaux conteneurs en informatique

II.1 Liste chaînée

Définition 4.2: liste chaînée

Une liste chainée est un conteneur, éventuellement vide, dont chaque élément ou cellule contient une donnée et l'adresse (mémoire) de la cellule suivante, on dit alors qu'elle pointe sur l'élément suivant. L'accès aux éléments d'une liste se fait de manière séquentielle : chaque élément permet l'accès au suivant (contrairement au tableau dans lequel l'accès se fait de manière directe, par adressage de chaque cellule dudit tableau).

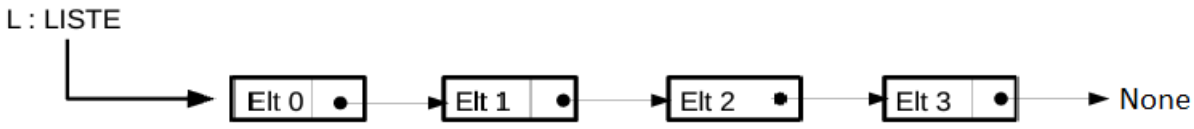


ILLUSTRATION D'UNE LISTE CHAÎNÉE

Cette structure supporte (la plupart du temps) les opérations minimales suivantes :

- création d'une liste vide ;
- ajout d'un élément dans la liste ;
- recherche d'un élément dans la liste ;
- suppression d'un élément dans la liste ;
- insertion d'un élément dans une position donnée ;
- accès à la cellule de tête ;
- accès au successeur d'une cellule (s'il y a un successeur).

Remarque 4.3. *En python, le type liste n'est pas réellement une liste chaînée, mais plutôt un tableau dont on peut changer la taille (cf sous-sections suivantes). L'accès au successeur n'est pas implémenté pour les listes python, alors qu'on peut accéder à l'élément d'indice k (il faudrait normalement partir de la cellule de tête et réaliser k fois l'opération successeur)*

II.2 Tableau

Définition 4.4: Tableau

Une structure de tableau est un conteneur possédant un nombre fixe d'éléments de même type.

- le nombre et le type des éléments sont définis à la création du tableau (un tableau est donc immuable contrairement à une liste) ;
- chaque composant du tableau est directement accessible en un temps $O(1)$, ce qui suppose que les données soient stockées dans des emplacements mémoire contigus et que l'élément d'adresse T [i] soit directement accessible.

Remarque 4.5. *Les tableaux ont l'avantage qu'on accède plus facilement à un élément que dans une liste chaînée. Par contre, on ne peut rajouter ou enlever un élément comme dans une liste chaînée. L'implémentation des listes en python semble être un hybride des deux conteneurs précédents, car le temps d'accès à une cellule n'est pas proportionnel à son indice.*

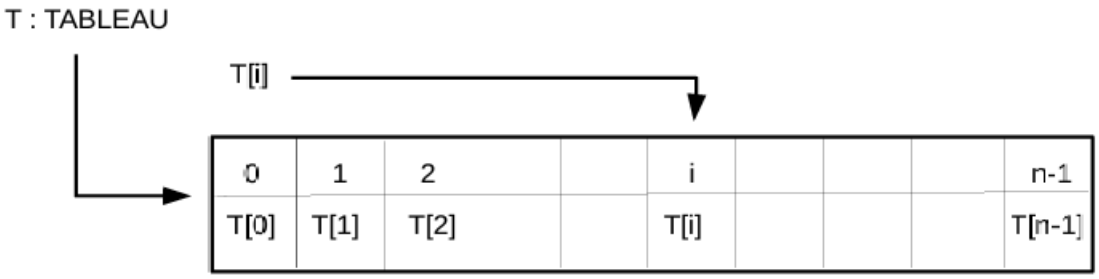


ILLUSTRATION D'UN TABLEAU

II.3 Tableau associatif ou dictionnaire

II.4 Définition et propriétés

Définition 4.6

Un tableau associatif ou dictionnaire est un conteneur dont les éléments sont des couples clef-valeur. C'est une application associant un ensemble de clefs à un ensemble de valeurs.

Remarque 4.7. *La recherche d'un élément se fait sur la clef (alors que dans un tableau elle ne se fait pas sur l'indice qui joue le rôle de la clé mais sur les éléments associés aux indices)*

Les tableaux associatifs supportent :

- la création d'une structure vide ;
- l'insertion d'un couple clé-valeur ;
- la suppression d'une clé (et de la valeur associée) ;
- la recherche d'une clé en un temps $O(1)$ en moyenne.

III Implémentation des dictionnaires

Nous allons voir un peu plus en détails comment sont implémentés les dictionnaires.

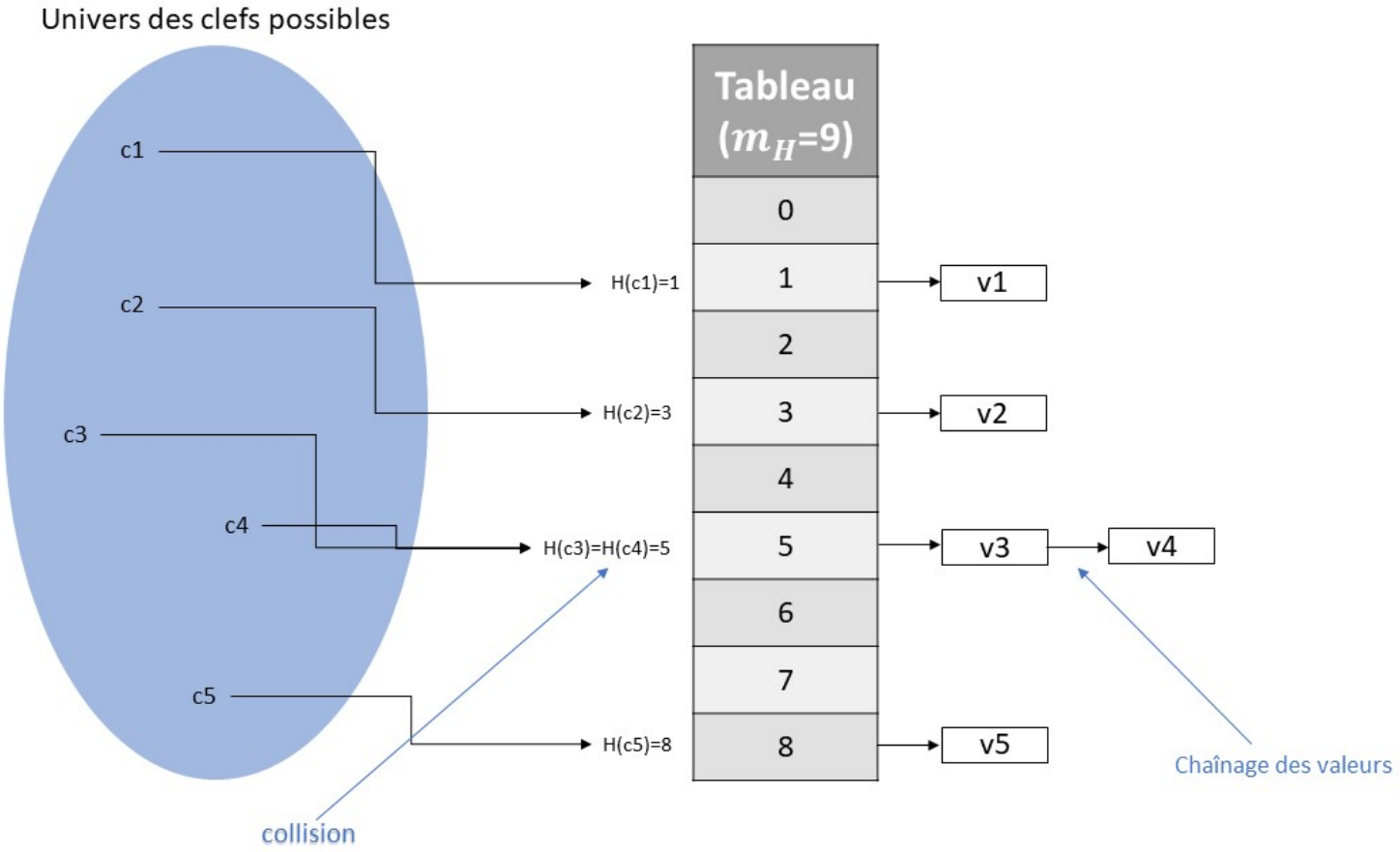
III.1 Fonctionnement et tables de hachage

L'implémentation d'une structure de dictionnaire procède généralement de la façon suivante :

1. On détermine un ensemble de clés envisageables (selon le langage). Par exemple, python accepte comme clé la plupart des types d'objets (les entiers, flottants, chaînes de caractères) sauf les listes ;
2. On se donne une famille de fonctions H (appelée **fonction de hachage**) de mon ensemble de clés possibles dans un segment d'entiers $[\![0, m_H - 1]\!]$;
3. Au moment de créer un dictionnaire une fonction de hachage est choisie et un tableau de taille m_H est réservé. On appelle **alvéole** les éléments de ce tableau.
4. Pour insérer un couple clef-valeur, H(clef) est calculé. C'est un entier $i \in [\![0, m_H - 1]\!]$
5. On place dans les alvéoles des pointeurs (adresses) vers des listes chaînées dans lesquelles les couples clefs-valeurs sont ajoutées (ou retranchés) au fur et à mesure.

Remarque 4.8.

- *On pourrait se demander pourquoi on insère pas directement la valeur de la clef dans l'emplacement H(clef). Si on effectue cela, il faut alors que pour tout clef2 différente de clef1, H(clef1) $\neq$ H(clef2), autrement dit que la fonction H soit injective. Mais ceci est totalement impossible car l'univers des clefs possibles est infini. Et quand bien même on se limiterait à des clefs d'une certaine taille mémoire, il serait bien trop grand... Plusieurs clefs doivent avoir la même image par H (valeur de hachage). Lorsqu'on utilise deux clefs ayant la même image par H, on parle de* ***collision****.*
- *Lorsque le nombre de clefs insérées augmente fortement, il est parfois nécessaire de redimensionner le tableau. En effet, pour que le dictionnaire conserve ses bonnes propriétés, il est nécessaire que les listes chaînées ne soient pas trop grandes. Lorsqu'on redimensionne le tableau, on change également de fonction de hachage.*



REPRÉSENTATION D'UNE TABLE DE HACHAGE, LA CLEF ci EST ASSOCIÉ À vi

C'est cette structure que l'on appelle table de hachage. Sa performance dépend de la fonction choisie. On souhaitera en général limiter le nombre de collisions tout en optimisant le taux d'occupation des alvéoles que l'on appelle aussi facteur ou taux de remplissage : $\alpha = \frac{n}{m_H}$ où n est le nombre de clés (ou de couples clé-valeur) insérées.

III.2 Quelques exemples de fonctions de hachage simples

En pratique, les objets qui sont des clefs possibles sont d'abord transformés en entier, puis on utilise cette valeur avec la fonction de hachage. Voici un exemple pour transformer les chaînes de caractères en entiers.

Transformation d'une chaîne de caractères en entier

On propose ici un exemple simple de pré-traitement des chaînes de caractères avant hachage. Nous allons nous servir du codage ASCII qui est donné sous Python par la fonction `ord(c)` qui renvoie le code ASCII sur 8 bits du caractère c. Si `ch` est une chaîne de n caractères du code ASCII , on associe à `ch` la valeur : $\sum_{k=0}^{n-1} \text{ord}(\text{ch}[k])256^k$. La fonction ainsi définie est bijective de l'ensemble des chaînes de caractères du code ASCII à valeurs dans $\mathbb{N}$.

```python
def chaine_en_entier(ch):
        r=0
        for k in ch:
                r=256*r+ord(k)
        return r
```

Par exemple, la chaine "Lycee Marceau" renvoie l'entier 605890819975307780371850811019 7.

Première fonction de hachage

C'est la fonction la plus simple, pour une table avec m alvéoles on choisit

$$H : c \mapsto c \bmod m$$

La qualité de ce hachage dépend fortement du choix de m si les clés ont une répartition qui n'est pas aléatoire. En pratique on choisit des nombres premiers éloignés des puissances de 2.

Seconde fonction de hachage

Cette méthode consiste à construire des fonctions de hachage de la façon suivante :

- on se donne un réel (ou un flottant) $\theta \in]0,1[$ et un entier $m > 0$.
- on définit

  $$H : e \in \mathbb{N}^* \mapsto \lfloor m \times d(e \times \theta) \rfloor \in [\![0, m-1]\!]$$

  où $d : x \in \mathbb{R} \mapsto x - \lfloor x \rfloor \in [0,1[$. Pour que cette fonction répartit uniformément les clefs, il faudrait prendre θ irrationnel. Mais comme tout représentant des réels est un décimal, il faut que sa représentation soit de la forme $\frac{p}{q}$ avec q très grand. En effet, si $\theta = \frac{p}{q}$ et $e = b \times q + r$ avec $0 \leq r < q$, alors $d(\theta e) = d(\theta r)$ ce qui implique qu'il n'y a que q valeurs possibles.

III.3 Opérations des dictionnaires en python

Opération	Résultat
`D={}`	affecte à la variable D un dictionnaire vide
`D={'a':1,'b':23}`	affecte à la variable D un dictionnaire donc les couples clefs-valeurs sont 'a'-1 et 'b'-23
`D['a']`	renvoie la valeur associée à la clef `'a'`
`D['a']=32`	affecte à la clef `'a'` la valeur entière 32
`del D['a']`	supprime l'association clef-valeur liée à la clef `'a'` dans D
`x in D, x not in D`	teste si x est une clef de D ou non
`list(D), set(D)`	renvoie la liste (resp. l'ensemble) des clefs de D
`D.keys(), D.values()`	permet de créer un itérateur sur les clefs ou les valeurs (pour les boules itératives)
`D.items()`	permet de créer un itérateur sur les couples clefs-valeurs
`D.copy()`	permet de copier D (sans dépendance)
`D={k:k**2 for k in range(5)}`	renvoie `{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}`

III.4 Exercices

Exercice 1. *Écrire une fonction `occurrence(L)` qui prend en argument une liste L et renvoie un dictionnaire où les clefs sont les éléments de L et les valeurs le nombre d'occurrence de la clef dans L.*

Exercice 2. *Écrire une fonction* `occurrence2(texte)` *qui prend en argument une chaîne de caractères* `texte` *et renvoie un dictionnaire où les clefs sont les lettres de* `texte` *et les valeurs le nombre d'occurrence de la clef dans texte.*

Exercice 3. *Écrire une fonction* `diviseurs(L)` *qui prend en argument une liste L d'entiers naturels et qui renvoie le dictionnaire dont les clefs sont les éléments de L et les valeurs la liste des diviseurs entiers naturels de la clef.*

Exercice 4. *On suppose que M est la matrice (liste de listes) d'adjacence d'un graphe dont les m sommets sont numérotés de 0 à* $m-1$*. Écrire une fonction python* `adj(M)` *qui prend cette matrice et renvoie un dictionnaire dont les clefs sont les sommets et les valeurs une liste contenant les voisins du sommet clef.*

PROGRAMMATION DYNAMIQUE

Sommaire

En informatique, la programmation dynamique est une méthode algorithmique pour résoudre des problèmes d'optimisation. La programmation dynamique consiste à résoudre un problème en le décomposant en sous-problèmes, puis à résoudre les sous-problèmes, des plus petits aux plus grands en stockant les résultats intermédiaires.

I Définition, programmation dynamique versus méthode "diviser pour mieux régner"

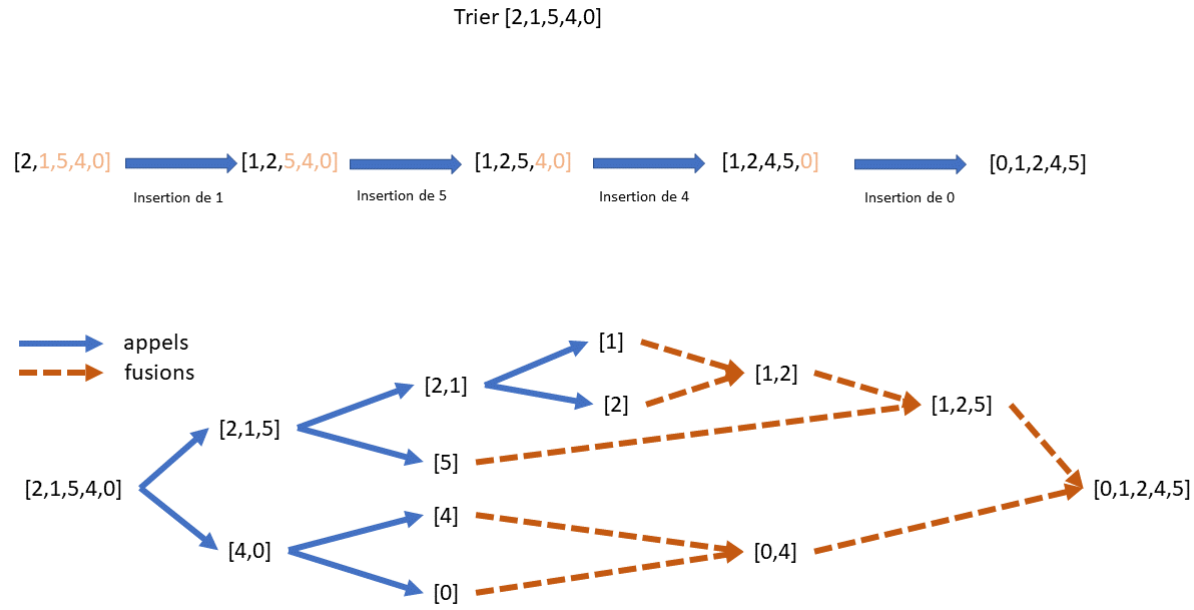
Le principe de la programmation dynamique est de décomposer le problème en sous-problème "de taille plus petite" dans un sens qu'il faut préciser selon le problème. La méthode "diviser pour mieux régner" consiste à diviser le problème en sous-problème indépendant pour ensuite résoudre le problème original de manière récursive.

Exemple du tri d'une liste

On peut voir la méthode de *tri par insertion* comme une méthode de programmation dynamique, le *tri fusion* est, quant à lui, utilise la méthode "diviser pour mieux régner".
Le tri par insertion résout les problèmes :

- je tri le premier élément ;
- je tri les deux premiers en insérant le second
- je tri les trois premiers en insérant le troisième
- etc...
- je tri la liste en insérant le dernier

A contrario, le tri fusion consiste à couper la liste en deux parts égales (ou presque) et à demander de les trier par appel récursif. Lorsque deux listes sont triées, on les fusionne avec une méthode linéaire.



EN HAUT LE TRI PAR INSERTION ET EN BAS LE TRI FUSION.

II Programmation ascendante, descendante

Il existe deux manières de programmer par la méthode de programmation dynamique :

Programmation ascendante

On utilise une méthode itérative pour passer des problèmes les plus petits aux problèmes les plus gros.

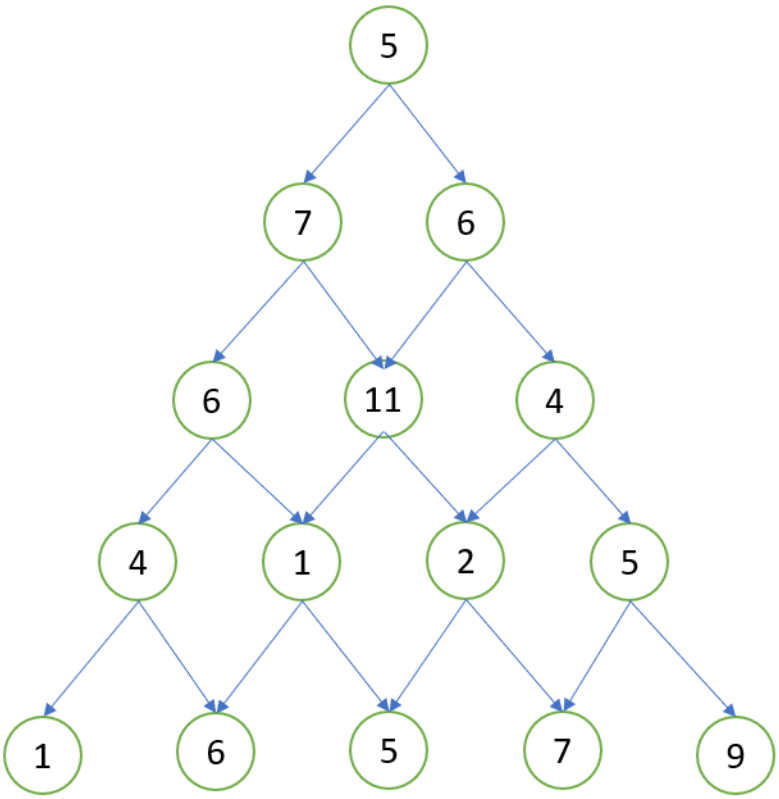
Programmation descendante

On utilise une méthode récursive pour résoudre les problèmes les plus gros à l'aide des problèmes les plus petits. Lorsque le graphe de dépendance des différents problèmes n'est pas un arbre (c'est-à-dire lorsqu'on a besoin de faire plusieurs fois le même appel récursif), on mémorise les résultats des appels récursifs : c'est la **mémoïsation**.

III Exemples

III.1 La pyramide de nombres

Dans une pyramide de nombres, on cherche, en partant du sommet de la pyramide, et en se dirigeant vers le bas à chaque étape, à maximiser le total des nombres traversés :

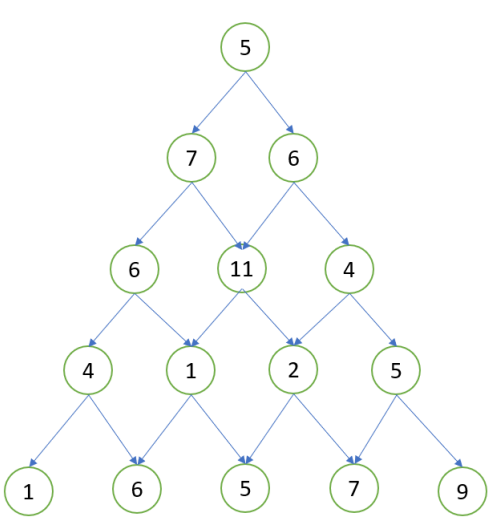


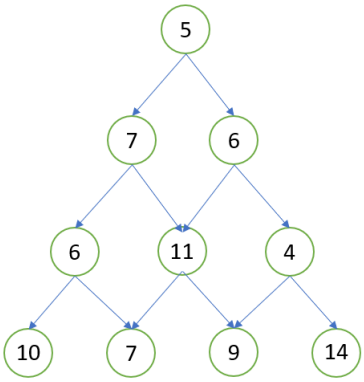
Une idée naïve consisterait à faire toutes les trajectoires possibles et de calculer le total pour chaque trajectoire, or, si il y a n niveau dans la pyramide, le nombre de trajectoire possible est de 2^{n-1} ce qui donne un algorithme de complexité exponentielle en temps.

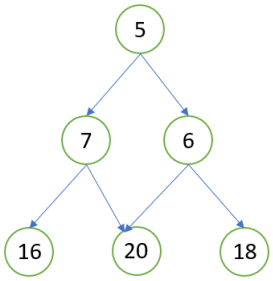
L'utilisation de la programmation dynamique permet d'obtenir un algorithme efficace en définissant des sous-problèmes, en écrivant une relation de récurrence, puis en donnant un algorithme (avec méthode ascendante ou descendante) :

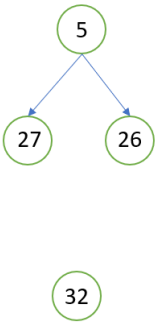
Pour toute position x dans la pyramide, notons $v(x)$ le nombre écrit à cette position et $c(x)$ le total des nombres traversés dans un chemin dont le total est maximal et partant de x. Les sous-problèmes consistent à calculer les valeurs de $c(x)$ pour tout x. Le problème initial consiste à calculer $c(x)$ lorsque x est le sommet de la pyramide. Pour cela, on va calculer les valeurs $c(x)$ niveau par niveau en partant du rez-de-chaussée :

$c(x) = v(x)$ pour toute position x situé au rez-de-chaussée de la pyramide $c(x) = v(x) + \max(c(g(x)), c(d(x)))$ pour toute autre position x, où $g(x)$ et $d(x)$ sont les positions inférieurs gauche et droite sous la position x.

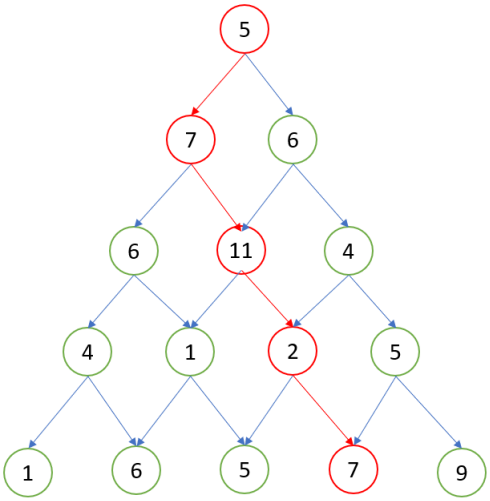








meilleur chemin :



Si on cherche à calculer directement par la définition récursive, on évalue plusieurs fois la même valeur par exemple, le meilleur chemin pour arriver à 11 de l'étage 2 est calculé pour 7 et 6 de l'étage 3. Il faut donc faire de la mémoïsation pour éviter de calculer plusieurs fois la même valeur. En ascendant, on stocke les valeurs dans, par exemple, une liste de listes, en partant du rez-de-chaussée au sommet.

III.2 Produit de matrice et parenthèsage optimal

Nous savons que la multiplication matricielle est associative c'est à dire que, lorsque cela a un sens, les produits $A \times (B \times C)$ et $(A \times B) \times C$ sont égaux ce qui fait qu'on note $A \times B \times C$ ou ABC sans préciser la place des parenthèses lorsqu'on ne s'intéresse qu'au produit lui-même. Par contre, les choses changent quand il s'agit de réaliser le calcul. Lorsque $A \in \mathcal{M}_{p,q}, B \in \mathcal{M}_{q,r}, C \in \mathcal{M}_{r,s}$, on a toujours $A(BC) = (AB)C \in \mathcal{M}_{p,s}$ mais le premier calcul aura coûté $pqs + qrs = qs(p + r)$ multiplications (de réels, complexes ou flottants) alors que le second en aura coûté $pqr + prs = pr(q + s)$.

$$\begin{pmatrix} * & * \\ * & * \\ * & * \\ * & * \end{pmatrix} \times \begin{pmatrix} * & * & * & * & * \\ * & * & * & * & * \end{pmatrix} \times \begin{pmatrix} * & * & * & * \\ * & * & * & * \\ * & * & * & * \\ * & * & * & * \\ * & * & * & * \end{pmatrix}$$

Dans cet exemple, la première méthode donne $qs(p+r) = 72$ alors que $pr(q+s) = 120$. Il est donc intéressant de planifier l'ordre des calculs, pour effectuer le moins d'opérations possibles. On peut montrer que le nombre de parenthèsages possibles pour n matrices est :

$$c_n = \sum_{i=0}^{n-1} c_i \times c_{n-1-i} = \frac{1}{n+1} \begin{pmatrix} 2n \\ n \end{pmatrix} \underset{n\to+\infty}{\sim} \frac{4^n}{\sqrt{\pi} n^{3/2}}$$

Ce sont les nombres de **Catalan** ($c_0 = 1$).

Considérons donc une suite de n matrices $(A_0, A_1, \ldots, A_{n-1})$ telle que les produits $A_i A_{i+1}$ soient définis. Pour $0 \leq i \leq n-1$, on note ℓ_i le nombres de lignes de A_i et on pose $\ell_n = c_{n-1}$, nombre de colonnes de A_{n-1} ; on a donc $A_i \in \mathcal{M}_{\ell_i, \ell_{i+1}}$. Nous voulons savoir quels parenthésages permettront d'optimiser le nombre de multiplications pour calculer le produit des A_i.

Imaginons qu'un tel parenthésage optimal conduise à effectuer comme dernière opération le produit de deux matrices $(A_0 \ldots A_k) \times (A_{k+1} \ldots A_{n-1})$. Ce dernier produit matriciel demandera à lui seul $\ell_0 \times \ell_{k+1} \times \ell_n$ multiplications. Le nombre total des multiplications sera donc égal à la somme de trois termes :
- nombre de multiplications pour le calcul de $(A_0 \ldots A_k)$;
- nombre de multiplications pour le calcul de $(A_{k+1} \ldots A_{n-1})$;
$-\ell_0 \times \ell_{k+1} \times \ell_n$.

Si cette somme de trois termes est optimale, le premier et le second terme sont aussi obtenus avec des parenthésages optimaux. En effet, si le parenthésage pour calculer $(A_0 \ldots A_k)$ n'est pas optimal, on peut le remplacer par un meilleur choix ce qui n'a pas d'incidence sur les façons de calculer $(A_{k+1} \ldots A_{n-1})$ ou le dernier produit. On améliore ainsi le score global ce qui contredit le fait que notre parenthésage pour $A_0 \ldots A_{n-1}$ est optimal.

Cette propriété est fondamentale, elle va nous permettre d'élaborer une stratégie pour déterminer les parenthésages optimaux. Notons $m(i,j)$ le nombre minimal de multiplications pour calculer $A_i \times \ldots \times A_j$ lorsque $0 \leq i < j \leq n-1$, et posons $m(i,i) = 0$. Pour calculer le produit $A_i \ldots Aj$ on peut placer une) après A_k pour $i \leq k < j$ et calculer séparément les deux produits $(A_i \ldots A_k)$ et $(A_{k+1}..A_j)$. Ce choix étant fait, le nombre minimal de multiplications est :

$$m(i,k) + m(k+1,j) + \ell_i \ell_k \ell_{j+1}$$

$m(i,j)$ est donc la valeur minimale parmi les $m(i,k) + m(k+1,j) + \ell_i \ell_k \ell_{j+1}$.

Il ne reste plus qu'à résoudre à l'aide de la programmation dynamique :

- soit par une méthode de programmation ascendante : on calcule les $m(i,j)$ lorsque $i = j$, puis $j = i+1$, $j = i+2$ et ainsi de suite ce qui revient à remplir la partie supérieure d'un tableau en partant de la diagonale.
- soit par une méthode de programmation descendante : on calcule les $m(i,j)$ en appelant récursivement la valeur de $m(i,k) + m(k+1,j) + \ell_i \ell_k \ell_{j+1}$ avec $k \in [\![i, j-1]\!]$. L'algorithme se

termine car $m(i,j)$ fait appel à des $m(u,v)$ tel que $|u-v| < |i-j|$. Le graphe de dépendance n'est clairement pas un arbre, il faut faire de la mémoïsation.

Il faudra stocker (dans les deux cas), le parenthèsage optimal en conservant quel élément donne le minimum.

Ces problèmes ont une chose en commun : la solution optimale pour un problème de taille n (par exemple, choisir un parenthésage optimal pour un produit de n matrices) se construit à partir de solutions également optimales pour des problèmes de tailles inférieures. On dit que les sous-problèmes sont indépendants : modifier la solution de l'un d'eux n'impose pas de modifier les autres pour conserver la solution globale. On remarquera que tous les problèmes d'optimisation ne présentent pas cette propriété.

IV Problèmes éligibles à la programmation dynamique

Comme on l'a vu dans les exemples, pour qu'un problème d'optimisation soit éligible il faut que la solution de l'optimisation d'un problème de taille n s'obtienne avec la ou les solutions solutions de l'optimisation d'un problème de taille inférieur ou égale à $n-1$.
C'est bien le cas, par exemple, dans le problème de parenthèsage des matrices, où la taille correspond aux nombres de matrices.

Théorème 5.1: Principe d'optimalité de Bellman

une solution optimale d'un problème s'obtient en combinant des solutions optimales à des sous-problèmes.

Il faut remarquer que ce n'est pas toujours le cas, prenons comme exemple simple celui de la recherche de chemins optimaux. On considère un graphe orienté $G = (S, A)$ et on s'intéresse à :
- la recherche d'un chemin avec un minimum d'arcs reliant un sommet u à v ;
- la recherche d'un chemin sans boucle avec un maximum d'arcs reliant u à v.

Exercice 5.2: Expliquer pourquoi, dans le premier cas, un chemin optimal qui relie u à v en passant par x est tel que les chemins $u \to x$ et $x \to v$ sont optimaux

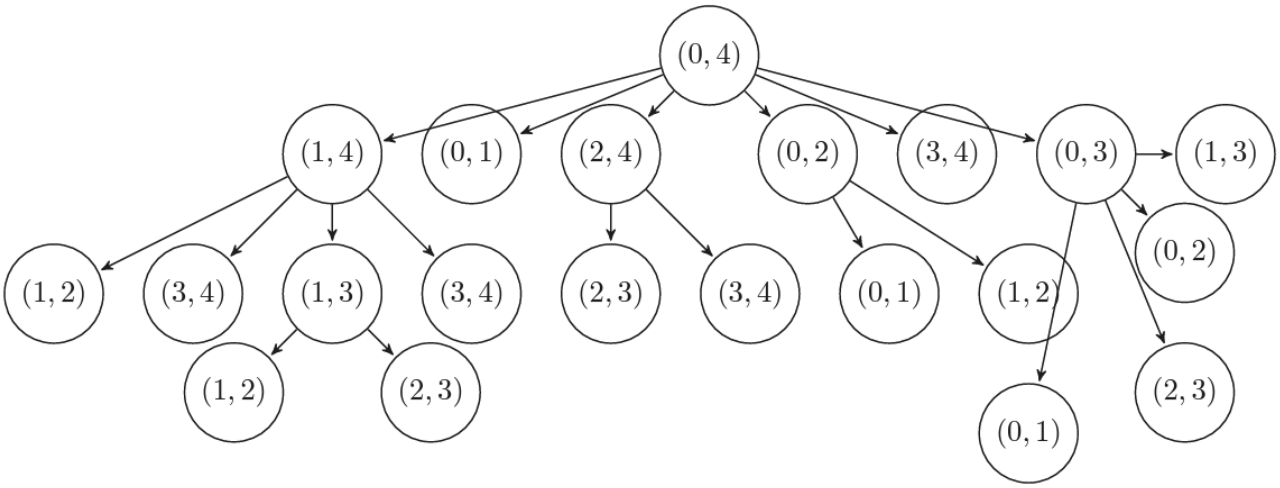
E

E

Exercice 5.3: Donner un exemple dans le second cas qui contredit la propriété de l'exercice précédent.

E

Dans la programmation dynamique, les sous-problèmes n'ont pas besoin d'être indépendant, il faut juste stocker les données pour éviter de recalculer les mêmes appels, par exemple, dans le cas $n = 5$ d'un produit de matrices, on obtient l'arbre d'appel suivant :



Il y a clairement des appels effectués plusieurs fois (et n est petit), il faut faire de la mémoïsation.

Dans le cas d'un algorithme "diviser pour mieux régner, les sous-problèmes d'un même niveau d'appel doivent être indépendants, c'est le cas dans l'arbre d'appel pour trier la liste [2,1,5,4,0] par tri fusion (cela serait aussi le cas avec quick-sort qui demanderait de trier [1,0] et [5,4] avec comme pivot 2, le premier élément.).

ALGORITHME POUR L'ÉTUDE DES JEUX

Sommaire

Le but de ce chapitre et de voir une manière usuelle de trouver des stratégies gagnantes à des jeux à deux joueurs au tour par tour.

I Représentation par des graphes, vocabulaire

Définition 6.1: prédécesseurs, successeurs

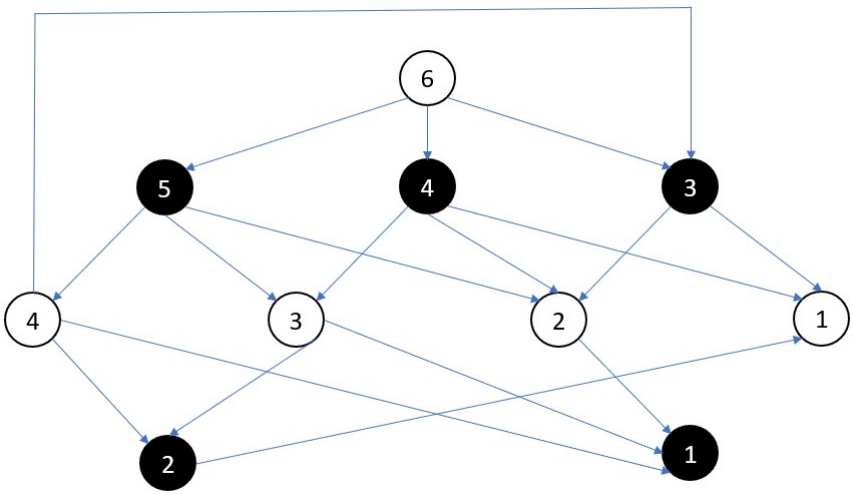
Soit $G = (S, A)$ un graphe orienté. Pour tout $s \in S$, les successeurs de s sont les extrémités des arcs dont s est l'origine. Les prédécesseurs de s sont les origines des arcs dont s est l'extrémité. Un sommet est terminal s'il n'a pas de successeur (c'est à dire si son degré sortant est nul).

Définition 6.2: graphe biparti

Soit $G = (S, A)$ un graphe (orienté ou pas, étiqueté ou pas). On dit que G est biparti s'il existe deux sous-ensembles de sommets, S_0 et S_1, formant une partition de S et tels que pour tout arc ou arête $a \in A$, l'origine et l'extrémité sont dans des S_i différents.

Exemple 6.3

Le graphe représentant une partie de Nim est un graphe biparti : si on note S_0 et S_1 les ensembles de sommets contrôlés par les joueurs J_0 et J_1 respectivement, on constate qu'il n'y a aucun arc de S_i vers lui-même.



GRAPHE DE LA PARTIE POUR UN NOMBRE DE BÂTONS DE SIX. LE NUMÉRO DU SOMMETS CORRESPOND AU NOMBRE DE BÂTONS RESTANTS. SI LE JOUEUR J_0 JOUE EN PREMIER, S_0 EST L'ENSEMBLE DES SOMMETS EN BLANC, S_1 L'ENSEMBLE DES SOMMETS EN NOIR.

Notation 6.4. *Par la suite, on notera xN le sommet de numéro x en noir, xB pour celui en blanc.*

Définition 6.5: graphe de jeu ou arène pour deux joueurs

Un graphe de jeu à deux joueurs ou arène est une structure $\mathcal{G} = (G, S_0, S_1)$ où $G = (S, A)$ est un graphe fini et biparti, S_0, S_1 sont comme dans la définition précédente. Par convention, on associe à tout graphe de jeu deux joueurs J_0 et J_1, et on dit que S_i est l'ensemble des sommets contrôlés par le joueur J_i.

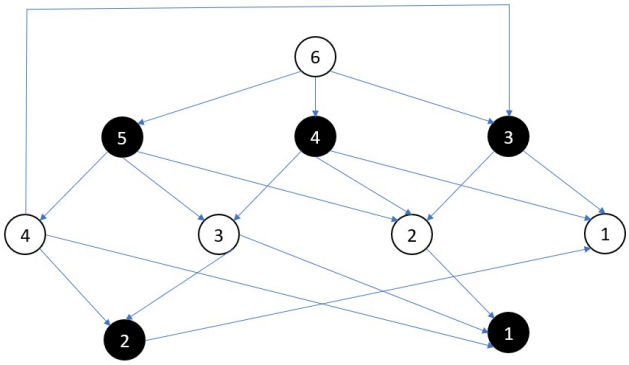
Exemple 6.6

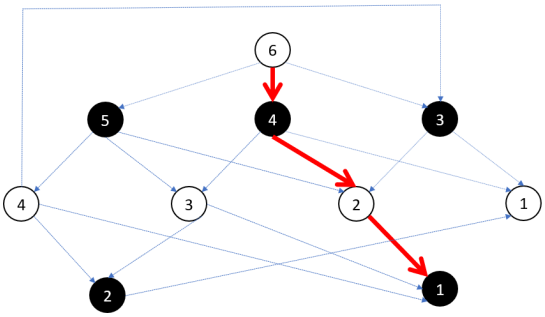
Exemples : Revenons au jeu de Nim : le joueur J_0 joue le premier et contrôle S_0, l'ensemble des sommets blancs. J_1 contrôle les autres sommets. Pour J_i, contrôler un sommet ou un état, signifie que c'est à son tour de jouer lorsque la partie est dans cet état.

Les graphes de jeux bipartis tels que nous les avons définis permettent de représenter des jeux dans lesquels deux joueurs jouent en alternance, où chaque coup conduit à une position contrôlée par le joueur adverse. Maintenant que le déroulement du jeu est en place (l'ensemble des coups permis est égal à A), il nous reste à définir formellement les **parties**, les **conditions de victoire**. Nous pourrons alors parler de **parties gagnantes,** de **positions gagnantes** et de **stratégies**.

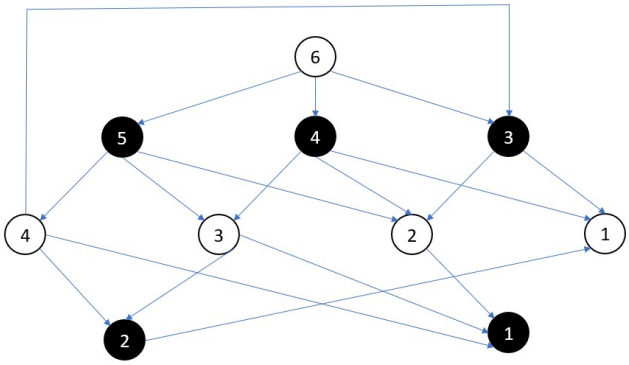
Définition 6.7

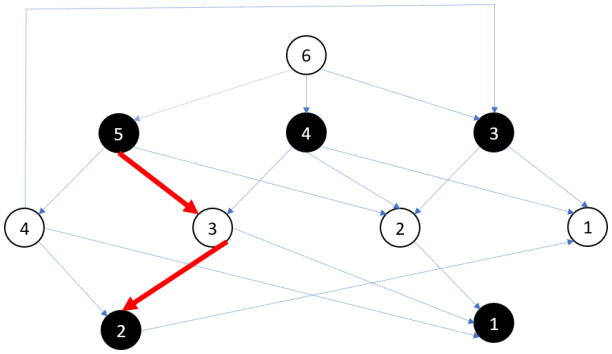
- Un chemin dans un graphe orienté G, est **maximal** lorsqu'il est infini ou lorsque l'extrémité de son dernier arc est terminale
- Soit $\mathcal{G} = (G, S_0, S_1)$ un graphe de jeu. On appelle "partie débutant en s_0" un chemin maximal de G et dont le premier arc a pour origine s_0.
- Une **partie partielle** débutant en s_0 est un chemin fini dont le premier arc a pour origine s_0.





EXEMPLE DE CHEMIN MAXIMAL DANS LE GRAPHE ORIENTÉ PRÉCÉDENT : (6B,4N,2B,1N) EST UN CHEMIN MAXIMAL CAR 1N EST TERMINAL. C'EST UNE PARTIE DÉBUTANT EN 6B





LE CHEMIN MIS EN ÉVIDENCE EST UNE PARTIE PARTIELLE DÉBUTANT ÉGALEMENT EN 5N.

Notation 6.8. *Nous noterons $\mathcal{P}$ l'ensemble des parties, $\mathcal{P}^*$ l'ensemble des parties partielles, $\mathcal{P}_0, \mathcal{P}_1$ les ensembles formés des parties partielles qui aboutissent à un sommet/état/position appartenant à S_0 ou S_1 respectivement. On pourra représenter une partie par la succession des sommets visités : $P = (s_0, s_1, \ldots, s_{p-1})$ représente donc le chemin $(s_0, s_1), (s_1, s_1), \ldots, (s_{p-2}, s_{p-1})$.*

Définition 6.9: stratégies, stratégies sans mémoire

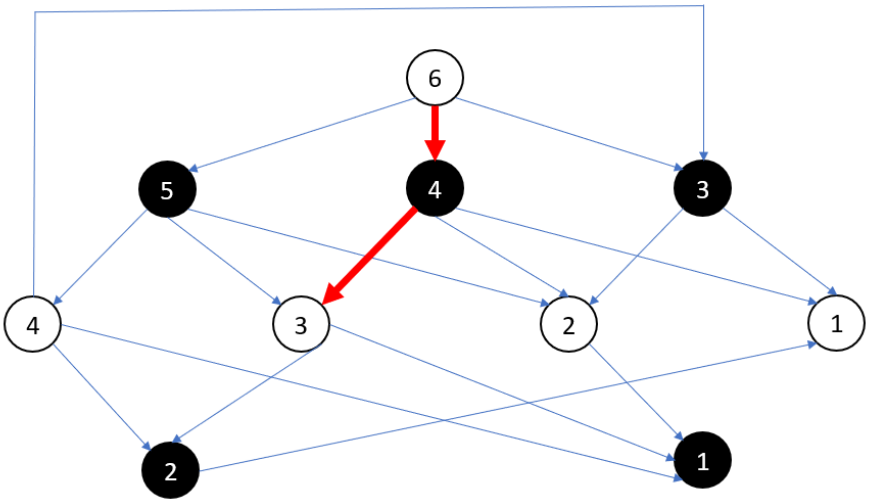
Soit $((S, A), S_0, S_1)$ un graphe de jeu.

- Une stratégie pour le joueur J_i est une application $\sigma : \mathcal{P}_i \longrightarrow S_{1-i}$ qui à toute partie partielle $(s_0, s_1, \ldots, s_{p-1})$ aboutissant en $s_{p-1} \in S_i$, associe un sommet s_p tel que $(s_{p-1}, s_p) \in A$. (s_{p-1}, s_p) est donc un coup jouable pour J_i.
- Une partie $(s_0, s_1, \ldots, s_t)$ est conforme à une stratégie pour le joueur i, si pour tout k tel que $0 \leq k < t$ et $s_k \in S_i$ on a $s_{k+1} = \sigma(s_0, s_1, \ldots, s_k)$.
- Une stratégie sans mémoire est une stratégie $\sigma : \mathcal{P}_i \longrightarrow S_{1-i}$ telle que pour tout couple de parties partielles $(s_0, s_1, \ldots, s_k), (v_0, v_1, \ldots, v_\ell) \in \mathcal{P}_i^2$,
$$s_k = v_\ell \Rightarrow \sigma((s_0, s_1, \ldots, s_k)) = \sigma((v_0, v_1, \ldots, v_\ell))$$

Disposer d'une stratégie σ permet de savoir quel coup (unique, car σ est une application) jouer en toute circonstance du jeu. Dire qu'une stratégie est sans mémoire c'est dire que pour jouer un coup dans une partie conforme, seul l'état actuel du jeu est à prendre en considération. Il existe une règle aux échecs qui nous dit qu'une partie est nulle si une même position des pièces sur l'échiquier se répète trois fois. Si on considère que les états sont les différentes positions des

pièces, il n'y a donc pas de stratégie utile sans mémoire.

Si on continue avec l'exemple du jeu de Nim, imaginons qu'on a effectuer la partie partielle (6B,4N,3B).



Une stratégie σ pour le joueur J_0, va être de choisir en connaissant la partie partielle (6B,4N,3B) le choix à effectuer pour le joueur. Ici, le joueur doit prendre deux bâtons pour gagner la partie, d'où une stratégie "intéressante" sera d'avoir $\sigma((6B, 4N, 3B)) = 1N$, ce qui est possible car $(3B, 1N) \in A$.
De manière générale, une "bonne" stratégie pour le jeu de Nim consiste à faire en sorte qu'il reste toujours un multiple de 4 + 1 bâtons sur la table, ce qui est toujours possible lorsqu'on débute (si on est le joueur J_0 ici). Cette stratégie est sans mémoire : elle n'utilise que l'état actuel du jeu (le nombre de bâtons restants sur la table).

Définition 6.10: condition de gain, position gagnante

Soit $((S, A), S_0, S_1)$ un graphe de jeu.

- Une condition de gain pour le joueur J_i dans ce jeu est un ensemble de parties $\Omega_i \subset \mathcal{P}$: J_i gagne une partie π ssi $\pi \in \Omega_i$.
- Une stratégie est gagnante pour le joueur J_i si toute partie conforme à la stratégie est gagnée par J_i.
- Une stratégie est gagnante à partir d'une position d pour le joueur J_i si toute partie conforme à la stratégie et qui débute en d est gagnée par J_i (appartient à Ω_i). On dit que d est une position gagnante pour J_i.

L'ensemble des parties gagnantes pour J_0 est l'ensemble des parties dont le dernier sommet est 1N :

- (6B,5N,4B,1N)
- (6B,5N,3B,1N)
- (6B,5N,2B,1N)
- (6B,4N,2B,1N)
- (6B,4N,3B,1N)
- (6B,3N,2B,1N)
- (6B,5N,4B,3N,2B,1N)

II Calcul des attracteurs

II.1 Jeu d'accessibilité

Définition 6.11: jeu d'accessibilité

Soit $\mathcal{G} = ((S, A), S_0, S_1)$ un graphe de jeu à deux joueurs, et Ω_i une condition de gain sur $\mathcal{G}$ pour le joueur J_i. On dit que le jeu ainsi défini est un jeu d'accessibilité lorsque

- il existe un ensemble de sommets terminaux F_i (ensemble cible) tel que les parties gagnantes pour J_i sont celles qui se terminent sur un sommet de F_i, on a donc
$$\Omega_i = \{(s_0, \ldots, s_t) \in \mathcal{P}; s_t \in F_i\}$$
- il n'y a pas de match nul (c'est à dire que $\Omega_{1-i} = \mathcal{P} \backslash \Omega_i$).

C'est le cas pour le jeu de Nim, toute partie mène aux sommets 1N ou 1B car dans une partie, le numéro du sommet décroit strictement.

II.2 Attracteurs et pièges

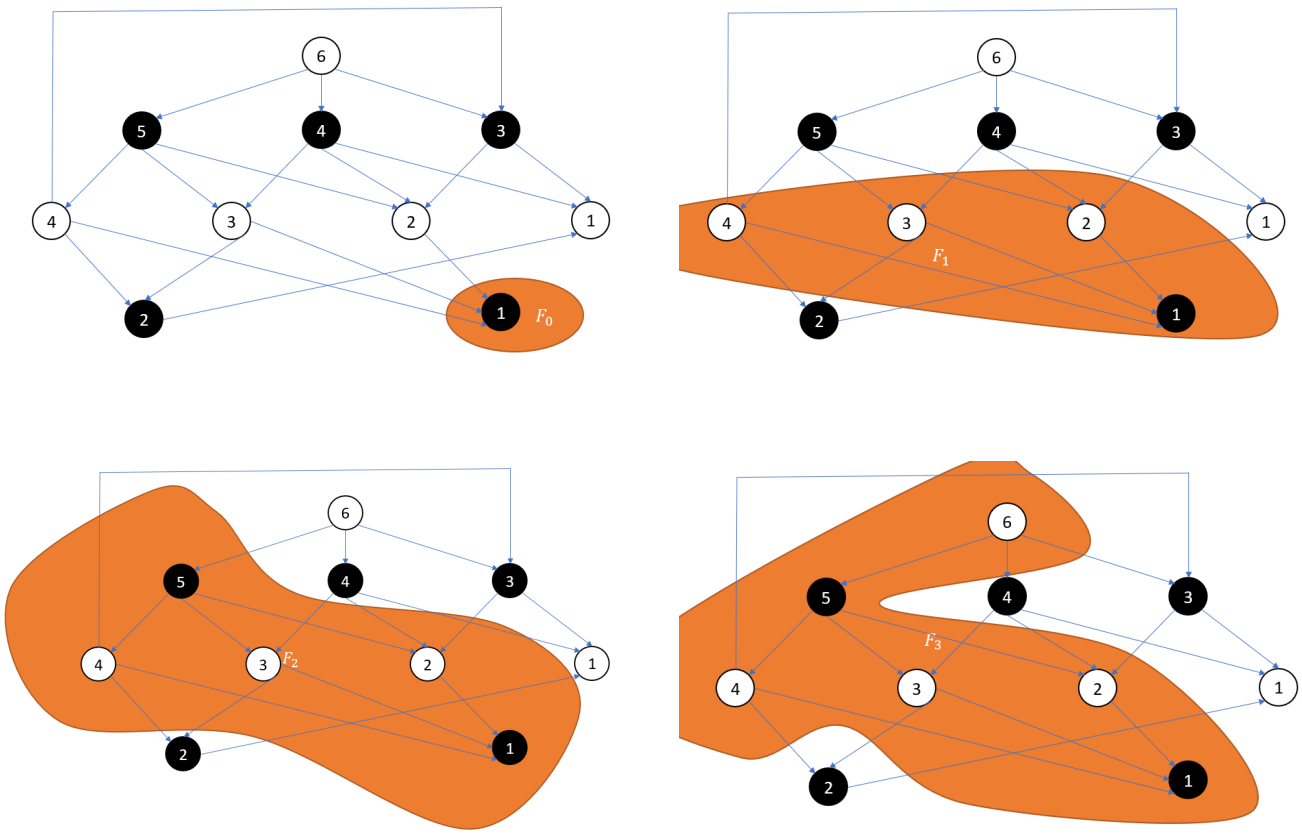
Considérons un jeu d'accessibilité dans lequel la condition de victoire pour le joueur J_0 est que la partie aboutisse à F_0. Pour une position donnée, J_0 est à un coup de gagner avec certitude, soit si c'est à son tour de jouer et s'il peut choisir un coup qui le conduit à F_0, soit si c'est à son adversaire de jouer et que tous les coups jouables mènent à F_0.

Cela nous incite à construire l'ensemble des positions gagnantes dans un tel jeu de la façon suivante :

1. On définit deux applications de des parties de S dans lui-même, en posant pour tout ensemble de sommets, $X \subset S$:
$$\mathrm{Pr}(X) = \left\{s \in S_0, \exists \left(s, s'\right) \in A, s' \in X\right\} \cup \left\{s \in S_1, \forall \left(s, s'\right) \in A, s' \in X\right\}$$
et $\mathcal{F}(X) = X \cup \mathrm{Pr}(X)$
$\mathrm{Pr}(X)$ est l'ensemble des positions qui permettent au joueur J_0 ou qui imposent au joueur J_1 d'amener le jeu en X.

2. On définit alors par récurrence une suite d'ensembles en posant :
   (a) $A_0 = F_0$;
   (b) $A_{n+1} = A_n \cup \mathrm{Pr}\left(A_n\right) = \mathcal{F}\left(A_n\right)$.

Observons que, lorsque $s \in A_i$, le joueur J_0 est à i coups au plus de la victoire. La suite $(A_i)_i$ est croissante pour l'inclusion et comme S est fini, elle est stationnaire : il existe un rang n_0 à partir duquel $A_{n_0} = A_{n_0+p}$ pour tout $p \in \mathbb{N}$. On note $\mathrm{Attr}\left(F_0, J_0\right) = A_{n_0}$.



CALCUL DE L'ATTRACTEUR DU COUPLE (F_0, J_0) QUI EST F_3

Définition 6.12: attracteur et rang

- L'ensemble Attr (F_0, J_0) est le bassin d'attraction (ou attracteur) de F_0 pour le joueur J_0.
- Pour tout $s \in$ Attr (F_0, J_0), on définit le rang des : $rg(s) = \min\{i \in \mathbb{N}, s \in A_i\}$. Lorsque $s \notin$ Attr (F_0, J_0), on pose : $rg(s) = \infty$.

Théorème 6.13: bassin attraction = position gagnante

Soit $\mathcal{G} = (S, A, S_0, S_1)$ un graphe de jeu d'accessibilité à deux joueurs pour lequel F_0 est la condition de gain pour le joueur J_0. Alors,

- $W_0 =$ Attr (F_0, J_0) est l'ensemble des positions gagnantes pour le joueur J_0.
- $W_1 = S \setminus$ Attr (F_0, J_0) est l'ensemble des positions gagnantes pour le joueur J_1.

Remarque 6.14. *Ce théorème nous dit que le bassin d'attraction* Attr (F_0, J_0) *est l'ensemble des positions gagnantes pour* J_0. *Un algorithme, avec une complexité linéaire en* $|S| + |A|$, *permet de le calculer.*

III Algorithme du minimax, heuristiques

Nous envisageons maintenant l'étude des jeux en déployant l'ensemble des parties possibles comme un arbre. Un nœud représentant un état, les branchements, les différents coups permis au joueur qui contrôle cet état.

III.1 Arbres

Nous donnons deux définitions équivalentes de la structure d'arbre. La première les présente comme des graphes particuliers, la seconde liée à la théorie des ensembles.

Définition 6.15

Un arbre est un graphe orienté dans lequel

- il existe un sommet et un seul, appelé racine, n'ayant pas d'antécédent ;
- tous les autres sommets admettent un prédécesseur et un seul.

Définition 6.16

On appelle arbre un ensemble $\mathcal{A}$ non vide, dont les éléments seront appelés nœuds, sur lequel est définie une relation $\mathcal{P}$ ($x\mathcal{P}y$ se dit "x est parent de y") telle que :

- il existe un nœud, r, et un seul, appelé racine, n'ayant pas de parent (c'est-à-dire, tel que $\forall x \in \mathcal{A}, \text{non}(x\mathcal{P}r)$) ;
- tous les autres nœuds admettent un(e) parent et un (e) seul(e) $(\forall y \in \mathcal{A}, \exists! x \in \mathcal{A}, (x\mathcal{P}y))$

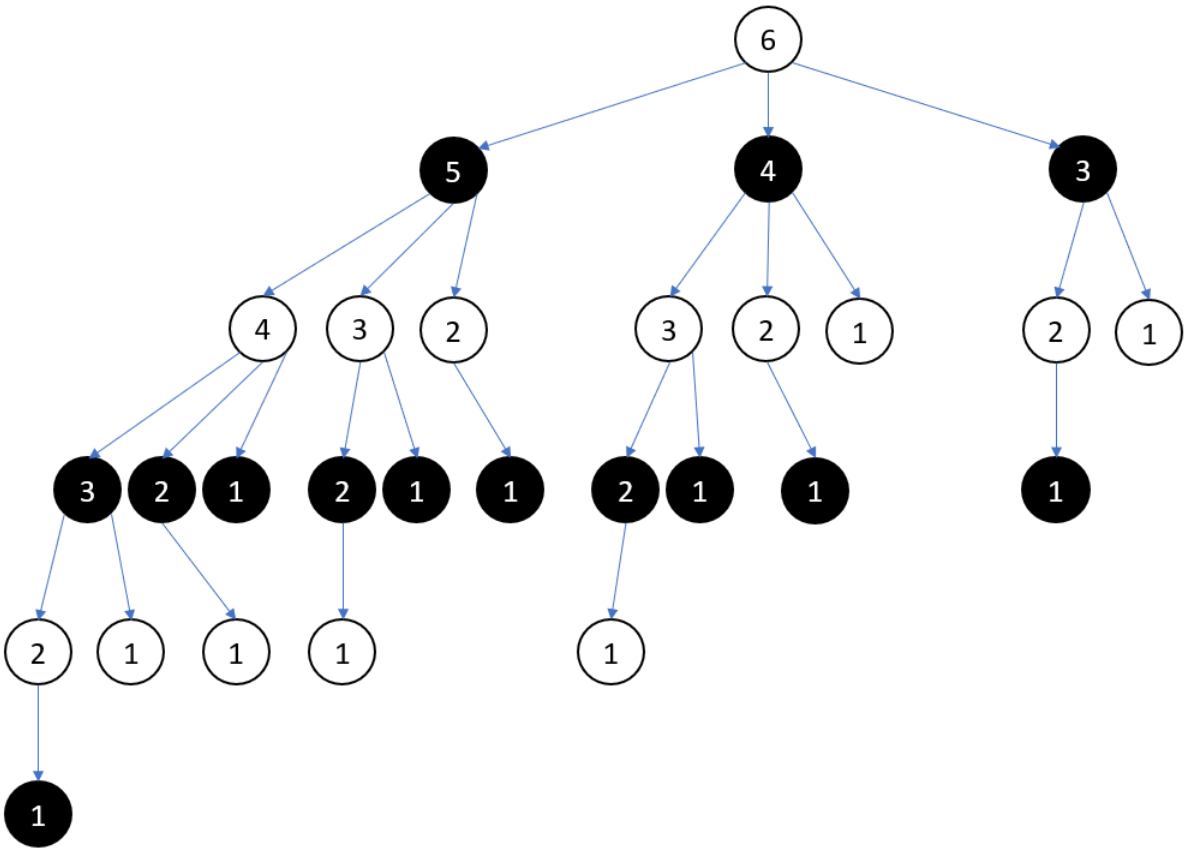
Théorème 6.17

Il existe un unique chemin qui nous permet de remonter d'un nœud quelconque jusqu'à la racine. Formellement : pour tout noeud $x \in \mathcal{A}\backslash\{r\}$, il existe une suite finie $x_0 = r, \ldots, x_p = x$ telle que pour tout i tel que $0 \leq i \leq p-1$, $(x_{i-1}\mathcal{P}x_i)$ (on dit que les x_i sont les ascendants de x).

Définition 6.18

Nous serons amenés à utiliser le vocabulaire suivant :

- les nœuds sans descendant sont les feuilles de l'arbre ou nœuds terminaux ;
- le degré d'un nœud est le nombre de ses descendants ;
- la profondeur d'un nœud est le nombre de ses ascendants stricts ;
- la hauteur d'un arbre est la profondeur maximale de ses nœuds ;
- un arbre est étiqueté lorsqu'à chaque nœud on associe une information, appelée étiquette.



ARBRE DE L'ENSEMBLE DES PARTIES POSSIBLES POUR LE JEU DE NIM DÉBUTANT À 6 BÂTONS, LES SOMMETS SONT ÉTIQUETÉS SELON LE JOUEUR QUI CONTRÔLE (COULEUR) ET LE NOMBRE DE BÂTONS RESTANTS.

III.2 L'algorithme du minimax

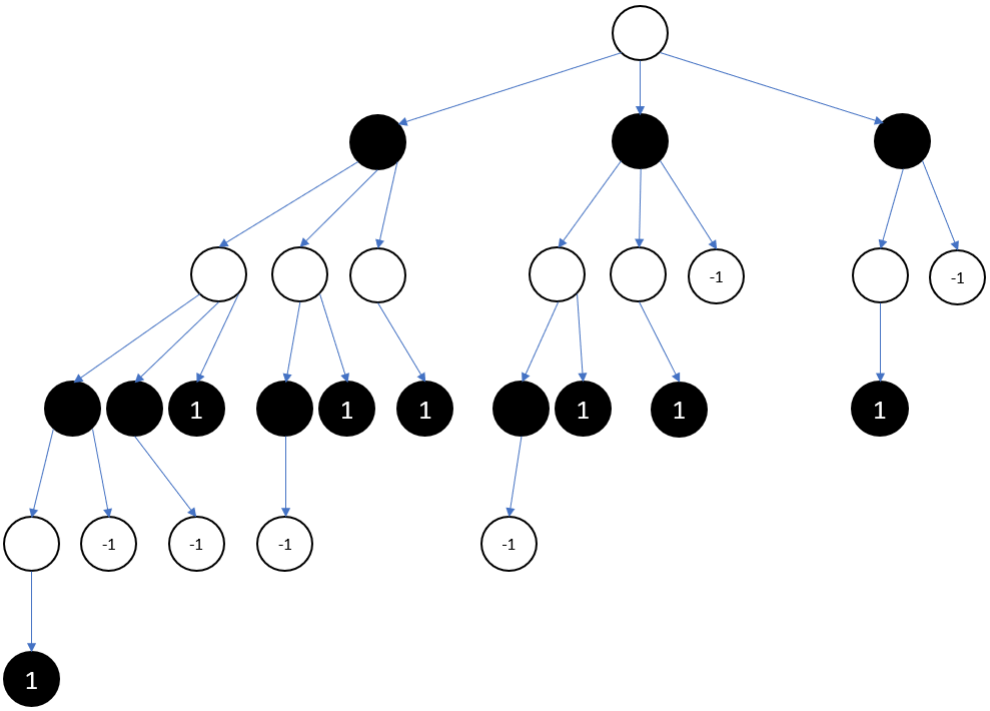
Théorème 6.19: L'algorithme du minimax

On considère, lorsque la complexité du jeu le permet, l'ensemble des parties possibles. Les feuilles d'un arbre de jeu représentent le résultat d'une partie (match gagné par J_0, perdu par J_0 ou nul), on leur attribue donc un score ou une valeur (respectivement $+1$, -1 ou 0 dans les cas où l'on ne considère que l'issue, $+g$, $-g$, 0 si les gains sont variables).

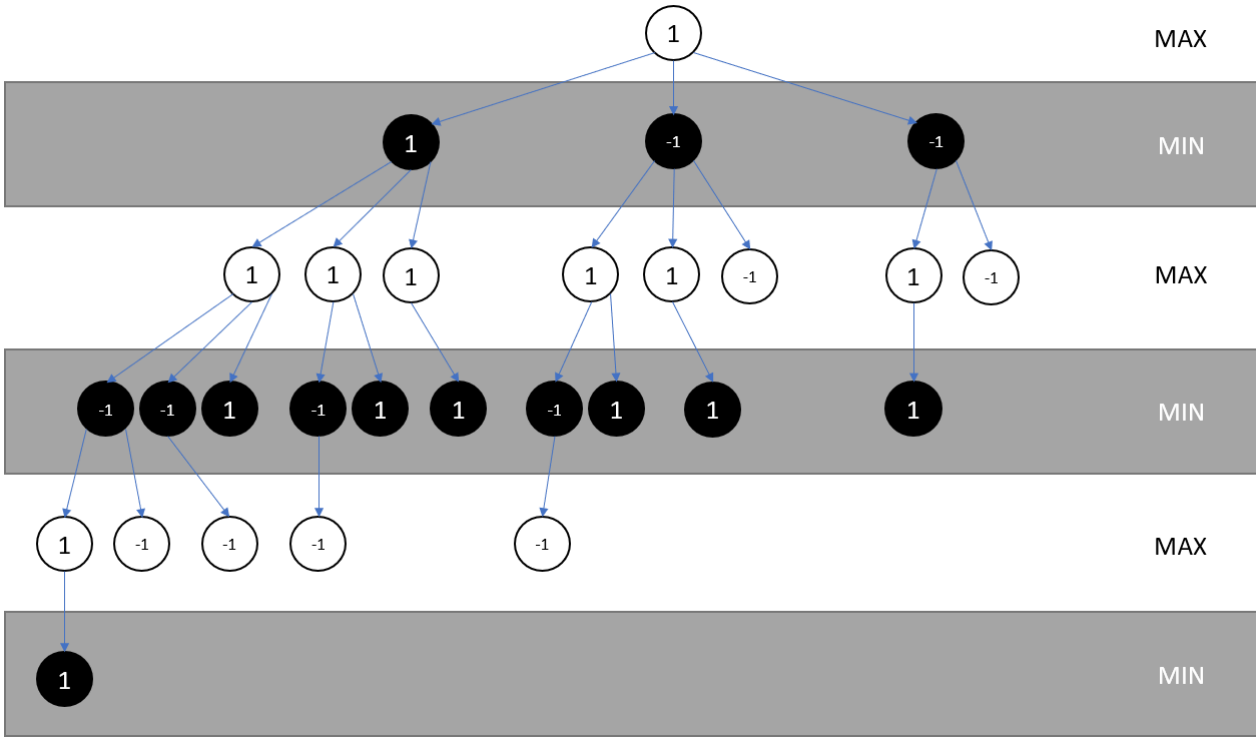
A partir de là, on définit récursivement une valeur pour chaque nœud de l'arbre :

- S'il est contrôlé par J_0, ce sera le maximum des valeurs des sous-arbres ;
- S'il est contrôlé par J_1, ce sera le minimum des valeurs des sous-arbres ;

Alors, lorsqu'un nœud a une valeur positive pour le joueur J_0, celui-ci peut imposer une partie conduisant à la victoire avec ce gain. Le score est calculé en supposant que l'adversaire joue tous ses coups de façon à minimiser ses pertes.



SCORE INITIALE DANS LE CADRE DU JEU DE NIM À 6 BÂTONS, COMME IL N'Y A PAS DE GAIN, LES SOMMETS TERMINALS SONT INITIALISÉS À ± 1



APPLICATION DE L'ALGORITHME DU MINMAX AU JEU DE NIM À 6 BÂTONS, LES VALEURS DANS LES SOMMETS SONT LES SCORES ENREGISTRÉS PAR L'ALGORITHME.

Théorème 6.20: L'algorithme du minimax avec fonction d'évaluation

On construit ou explore l'arbre du jeu à partir d'une position donnée en limitant la profondeur des sous-arbres à explorer (3 coups, 5 coups d'avance...). L'algorithme explore donc un arbre de jeu partiel dont les feuilles ne sont plus nécessairement des fins de parties.

On se donne une fonction d'évaluation qui est susceptible d'attribuer une valeur à tous les états du jeu. Aux échecs, elle dépendra de la nature des différentes pièces, de leurs positions (l'occupation du centre de l'échiquier ayant une meilleure valeur stratégique), etc. L'algorithme n'est pas modifié dans sa conception : il opère sur un arbre de jeu partiel et sa fonction d'évaluation est heuristique.

Ce dernier algorithme est utile lorsque l'arbre est beaucoup trop grand pour être parcouru totalement, c'est le cas, par exemple, lors d'une partie d'échec.

Définition 6.21: Heuristique

En informatique, l'heuristique est une technique conçu pour résoudre un problème plus rapidement lorsque les méthodes classiques sont trop lentes, ou pour trouver une solution approximative lorsque les méthodes classiques ne trouvent de solution exacte. Elle permet de trouver une solution dans un délai raisonnable, même si celle-ci n'est pas toujours optimale.

ALGORITHMES POUR L'ÉTIQUETAGE ET LA CLASSIFICATION

Sommaire

Nous allons, exposer des méthodes de classement et de classification automatique. Nous commençons par en préciser la problématique et les notions sous-jacentes.

I Classement et classification automatique

Définition 7.1: Partition

Soit E un ensemble. I et J des ensembles quelconques

- Une partition de E est la donnée d'une famille $(A_i)_{i\in I}$ de parties ou sous-ensembles non-vides de E, telle que : $\begin{cases} \bigcup_{i\in I} A_i & = E \\ A_i \cap A_j & = \emptyset, \text{ pour tous } i \neq j \end{cases}$
- Soient $(A_i)_{i\in I}$ et $(B_j)_{j\in J}$ deux partitions du même ensemble E. On dit que $(B_j)_j$ est plus fine que $(A_j)_j$ ssi $\forall j \in J, \exists i \in I, B_j \subset A_i$.

Proposition 7.2

- Si R est une relation d'équivalence sur un ensemble E, ses classes d'équivalence forment une partition de E.

- Pour toute partition P de E il existe une relation d'équivalence sur E telle que P est l'ensemble des classes d'équivalence de R. Elle peut trivialement se définir par « $R(x, y)$ ssi x et y appartiennent à un même élément de P ».

Définition 7.3: Classer ou étiqueter

Classer, étiqueter les éléments d'un ensemble (documents écrits, images, individus, lieux géographiques, événements ou données quelconques) revient à placer ces éléments dans des classes qui forment une partition de cet ensemble. Dans une autre formulation, cela revient à associer à chaque élément une étiquette appartenant à une liste prédéfinie (les éléments d'une même classe se voyant affublés d'une même étiquette). Les propriétés des partitions assurent que l'opération de classement d'un élément e est toujours définie de façon unique.

Définition 7.4: Classifier

Classifier est l'opération qui vise à définir une partition ou un système de partitions pour un ensemble de données. On imagine sans peine qu'en pratique on visera à obtenir une classification dans laquelle les individus appartenant à une même classe seront le plus ressemblants/proches possibles, les individus appartenant à des classes distinctes devront être le moins ressemblants/proches possible. Cette notion de ressemblance ou de proximité dépendra du type des données et nous serons amenés à considérer des distances ou des mesures de similarité adaptées à chaque problème.

Exemple 7.5

- Classifier des mails en "spams" et "non-spams"
- Classifier les réels : les positifs et les négatifs stricts.
- Classifier des romans : "science-fiction", "romantique", etc..
- Classifier un ensemble de portraits selon si ce sont des portraits de chats, de chiens, d'humains, etc..

II Distance ou similarité

II.1 Distance

Définition 7.6: distance

Soit E un ensemble, une distance sur E est une application de $E \times E$ dans $\mathbb{R}_+$ telle que pour tout $(x, y, z) \in E^2$:

1. $d(x, x) = 0$ et $d(x, y) = 0 \Rightarrow x = y$ (axiome de séparation) ;
2. $d(x, y) = d(y, x)$ (symétrie) ;
3. $d(x, y) \leq d(x, z) + d(z, y)$ (inégalité triangulaire).

Exemple 7.7

On peut en particulier définir une distance sur une partie d'un espace vectoriel à partir d'une norme en posant $d(X,Y) = \mathcal{N}(X-Y)$. Ainsi, dans $\mathbb{R}^n$,

$$d(X,Y) = \|X-Y\|_1 = \sum_{i=1}^{n} |x_i - y_i|$$
$$d(X,Y) = \|X-Y\|_2 = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$
$$d(X,Y) = \|X-Y\|_\infty = \max_{1\leq i\leq n} |x_i - y_i|$$

Exemple 7.8: Distance de Levenshtein

Soient $X = X_0$ et Y deux mots ou chaînes de caractères. On s'autorise les opérations élémentaires $X_p \longrightarrow X_{p+1}$ suivantes :

- X_{p+1} est obtenue à partir de X_p par suppression d'un caractère ;
- X_{p+1} est obtenue à partir de X_p par insertion d'un caractère de Y ;
- X_{p+1} est obtenue à partir de X_p par remplacement d'un caractère par un caractère différent provenant de Y.

La distance de Levenshtein entre X et Y est égale à la longueur minimale d'une suite d'opérations du type de celles précédemment décrites (on dira de types S, I_Y ou R_Y) qui permet de transformer X en Y. On la note $\mathrm{Lev}(X,Y)$.

On peut démontrer que Lev est bien une distance.

II.2 Mesures de similarité

Les distances usuelles dans les problèmes numériques ou qui ont une modélisation géométrique ne sont pas adaptées à toutes les situations. Nous ferons alors intervenir d'autres distances ou, à défaut, des mesures ou indices de similarité dont nous donnons ici quelques exemples.

Exemple 7.9: Indice et distance de Jaccard

On définit a priori l'indice de Jaccard comme un mesure de similarité entre deux ensembles finis A et B (l'un d'eux étant non vide) en posant

$$J(A,B) = \frac{|A\cap B|}{|A\cup B|} \text{ qui a les propriétés } \begin{cases} 0 \leq J(A,B) \leq 1 \\ J(A,B) = 1 \Leftrightarrow A = B \\ J(A,B) = 0 \Leftrightarrow A\cap B = \emptyset \end{cases}$$

Nous pouvons lui associer la distance de Jaccard, définie par

$$d_J(A,B) = 1 - J(A,B) = 1 - \frac{|A\cap B|}{|A\cup B|} = \frac{|A\Delta B|}{|A\cup B|}.$$

où $A\Delta B$ désigne la différence symétrique entre A et B : $A\Delta B = (A\cup B)\backslash(A\cap B)$. Cet indice intervient lorsque nous voulons comparer des éléments caractérisés par des attributs booléens/binaires (présence ou non de certaines propriétés). Supposons que nous voulions comparer des individus selon qu'ils possèdent ou non des propriétés P_i. Chaque individu est

représenté par un vecteur de $\{0,1\}^n$ avec $x_i = 1$ ssi $P_i(x)$:

$$X = \begin{bmatrix} 1 \\ 1 \\ 0 \\ 1 \\ 0 \end{bmatrix} \leftarrow \text{ ne vérifie pas } P_2, \quad Y = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \end{bmatrix} \leftarrow \text{ vérifie } P_2.$$

La distance de Jaccard est nulle lorsque les vecteurs X et Y sont identiques, égale à 1 lorsque $\forall i, x_i \neq y_i$.

Exemple 7.10: Comparaisons de textes, matrices termes-documents

Pour évaluer la proximité entre documents textes issus d'un même corpus on commence par en donner une représentation sommaire en repérant les mots utiles du corpus et en associant à chaque document D_j un vecteur colonne X_j dont le terme $x_{i,j}$ dépend de la fréquence du mot ou du terme t_i dans le document et dans le corpus $\mathcal{C}$. Le nombre d'occurrences du terme pourrait servir à comparer grossièrement des documents de même taille.

Cependant, on préfère le plus souvent le codage tf-idf (tf : fréquence ou plutôt nombre d'occurrences des termes dans les documents, idf : inverse du nombre de documents contenant un terme) défini par les formules

$$x_{i,j} = f_{t_i,d_j} \times \ln\left(\frac{|\mathcal{C}|}{df_{t_i}}\right) \text{ avec } \begin{cases} f_{t_i,d_j} & : \text{ fréquence de } t_i \text{ dans } D_j \\ df_{t_i} & : \text{ nombre de docs. contenant } t_i \end{cases}$$

Un indice de similarité entre deux documents est alors donné par la valeur du cosinus de "l'angle" entre X_j et X_k comme vecteurs de $\mathbb{R}^n$ avec n taille du lexique retenu (de quelques milliers à quelques dizaines de milliers de termes en pratique) :

$$\text{sim}\left(D_j, D_k\right) = \frac{\sum x_{i,j}x_{i,k}}{\left\|X_j\right\|_2 \left\|X_k\right\|_2} = \frac{\left(X_j \mid X_k\right)}{\left\|X_j\right\|_2 \left\|X_k\right\|_2}$$

où $\left(X_j \mid X_k\right)$ désigne le produit scalaire canonique dans $\mathbb{R}^n$. L'exercice qui suit permet de vérifier que deux documents similaires ont un indice de similitude proche de 1 , que deux documents très différents ont un indice de similitude proche de 0.

Exercice 7.11

On suppose que les documents d'un corpus sont représentés comme par des vecteurs X_j tels que $x_{i,j} = f_{t_i,d_j} \times \ln \frac{|\mathcal{C}|}{df_{t_i}}$.

1. On suppose que le mot ou le terme t_i est présent dans tous les documents. Que vaut alors $x_{i,j}$?
2. On suppose que le mot ou le terme t_i est présent dans le seul document D_j. Que vaut alors $x_{i,j}$?
3. On pose $\delta\left(D_j, D_k\right) = 1 - \text{sim}\left(D_j, D_k\right)$. Que dire des documents D_j et D_k tels que $\delta\left(D_j, Dk\right) = 0$? (Penser au cas d'égalité de Cauchy-Schwarz). S'agit-il d'une distance ?
4. δ vérifie-t-elle l'inégalité triangulaire ? Indication : on pourra réécrire une relation $\delta(X,Z) \leq \delta(X,Y) + \delta(Y,Z)$ en introduisant les notations

$$\cos\alpha = \frac{(X \mid Y)}{\|X\|_2, \|Y\|_2}, \cos\beta = \frac{(Y \mid Z)}{\|Y\|_2, \|Z\|_2}, \cos\gamma = \ldots$$

puis s'aider d'un dessin (on se souviendra que trois vecteurs de $\mathbb{R}_+^n$ sont coplanaires).

5. Que peut on dire de deux documents pour lesquels $\delta\left(D_j, D_k\right) = 1$?

III Inertie d'une partition

On définit l'inertie d'une partie finie de $\mathbb{R}^n$, puis d'une partition d'un ensemble, notion essentielle dans l'évaluation des algorithmes de classification.

Définition 7.12: Inertie

- Si $A = \left(X_j\right)_{0\leq j\leq m-1}$ est une partie finie de $\mathbb{R}^n$, le barycentre et l'inertie de A sont définis par :

$$G_A = \frac{1}{m}\sum_{j=0}^{m-1} X_j$$

$$I_A = \sum_{j=0}^{m-1} \left\|X_j - G_A\right\|_2^2$$

  On appelle aussi variance de A, la moyenne $\frac{I_A}{|A|}$.

- Soit $(A_k)_{0\leq k\leq K-1}$ une partition d'un ensemble fini E contenu dans $\mathbb{R}^n$, on appelle inertie totale de la partition la somme des inerties de chacune de ses classes :

$$I = \sum_{k=0}^{K-1} I_{A_k} = \sum_{k=0}^{K-1}\sum_{X_j\in A_k} \left\|X_j - G_{A_k}\right\|^2$$

Remarque 7.13. *Comme une variance en mathématique, plus la variance est petite, plus les points sont proches du barycentre, cela revient à dire ici que les éléments (d'une même partition) se "ressembleront". Le but sera donc d'essayer de rendre l'inertie totale la plus faible possible.*

IV Intelligence artificielle

L'intelligence artificielle (IA) est un terme souvent utilisé de manière large, englobant divers concepts, parfois de façon quelque peu floue, surtout lorsqu'il est abordé par des journalistes. Dans l'ensemble, on peut définir l'IA comme un "ensemble de théories et de techniques mises en œuvre en vue de réaliser des programmes informatiques qui visent à 'simuler' l'intelligence humaine". Bien que cela puisse être interprété de différentes manières, il existe un consensus général sur l'inclusion d'algorithmes, de méthodes, ou de programmes spécifiques dans le domaine, en fonction soit du problème à résoudre, soit des techniques mises en œuvre.

Certains problèmes abordés par l'IA sont souvent caractérisés par leur complexité, rendant difficile l'application d'une méthode exacte de bout en bout, ou parfois, ces problèmes ne se prêtent tout simplement pas à une formulation précise.

Caractérisation par les problèmes

- traitement du langage : utilisation d'algorithmes d'apprentissage automatique pour identifier la langue d'un texte en fonction de modèles linguistiques, diviser le texte en tokens

de manière "intelligente", analyser la structure grammaticale d'une phrase, trouver les sentiments exprimés dans le texte, les générateurs de texte en particulier les LLM (Large Language Model comme chatGPT).

- classement, étiquetage automatique (cf le début du chapitre) ;
- classification automatique, clustering (cf le début du chapitre) ;
- certains programmes de jeux dans lesquels interviennent des heuristiques...

Caractérisation par les méthodes

- systèmes experts : un outil capable de reproduire les mécanismes cognitifs d'un expert, dans un domaine particulier ;
- La programmation logique est une forme de programmation qui définit les applications à l'aide :
  - d'une base de faits : ensemble de faits élémentaires concernant le domaine visé par l'application,
  - d'une base de règles : règles de logique associant des conséquences plus ou moins directes à ces faits,
  - d'un moteur d'inférence (ou démonstrateur de théorème) : exploite ces faits et ces règles en réaction à une question ou requête.

  Cette approche se révèle beaucoup plus souple que la définition d'une succession d'instructions que l'ordinateur exécuterait.
- l'inférence bayésienne : est l'ensemble des techniques permettant d'induire les caractéristiques d'un groupe général à partir de celles d'un groupe particulier, en fournissant une mesure de la certitude de la prédiction, tout cela en utilisant principalement la formule de Bayes.
- apprentissage supervisé (cf la suite) ;
- apprentissage non supervisé (cf la suite) ;
- apprentissage profond (réseaux de neurones) ;

Cette diversité de caractérisations souligne la richesse et la variété des approches au sein du domaine de l'intelligence artificielle.

IV.1 Apprentissage supervisé

La problématique de l'apprentissage supervisé est la suivante :

- On dispose de données déjà classées ou étiquetées (s'il s'agit d'un problème de classement) ou de données sous la forme d'entrées-sorties. C'est l'ensemble d'apprentissage.
- On souhaite pour des données nouvelles du même type, attribuer une étiquette ou une valeur de sortie en se basant sur ce que l'on sait déjà grâce à l'ensemble d'apprentissage.

D'une façon générale, on se donne un modèle pour la fonction de prédiction f : entrées $\longrightarrow$ sorties, et on cherche à approcher f à partir des couples d'entrées-sorties fournis par l'ensemble d'apprentissage. Les différentes méthodes reposent sur des raisonnements probabilistes ou statistiques.

Exemple 7.14

- Reconnaissance de forme ;
- Reconnaissance vocale ;
- vision par ordinateur (transformation d'images visuelles en descriptions du monde qui ont un sens pour les processus de pensée et peuvent susciter une action appropriée, par exemple pour qu'un rover puisse se déplacer sur Mars).

IV.2 Apprentissage non supervisé

On parle d'apprentissage non supervisé lorsqu'il s'agit de découvrir les paramètres d'un modèle de structure sous-jacente à un ensemble de données en l'absence de connaissance a priori.

Exemple 7.15

- Partitionnement des données pour, par exemple, les compresser ou extraire des connaissances, pour faire émerger des sous-ensembles et sous-concepts éventuellement impossibles à distinguer naturellement.
- La réduction de la dimensionnalité (ou réduction de (la) dimension) : c'est un processus qui consiste à prendre des données dans un espace de grande dimension, et à les remplacer par des données dans un espace de plus petite dimension. Pour que l'opération soit utile il faut que les données en sortie représentent bien les données d'entrée.

V Classement supervisé, k-plus proches voisins

V.1 Définitions

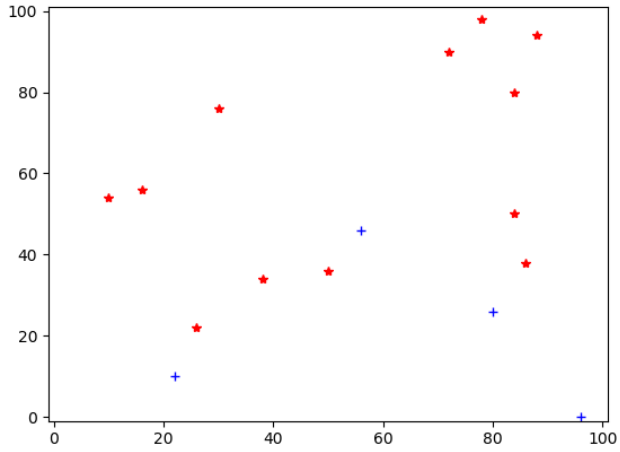
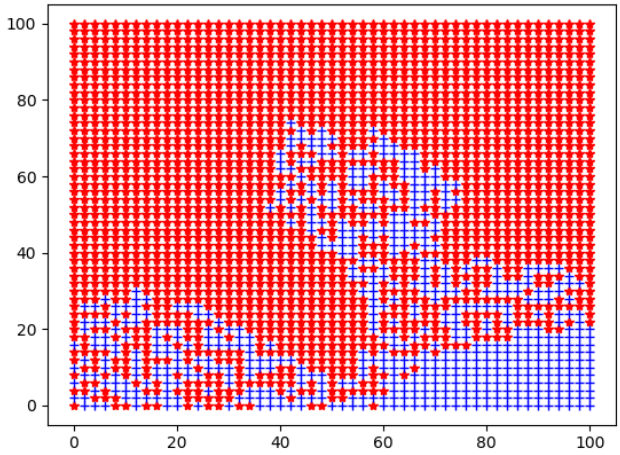
Théorème 7.16: Algorithme des k-plus proches voisins

On veut classer ou étiqueter des données appartenant à un ensemble muni d'une distance ou d'une mesure de similarité δ. Les étiquettes sont déterminées et on dispose d'un ensemble d'apprentissage A dans lequel chaque donnée est étiquetée ou classée.

Algorithme 7.17: k plus proche voisins

L'idée de l'algorithme est la suivante : Pour prédire l'étiquette d'un élément X'

1. On calcule les k éléments de l'ensemble d'apprentissage qui sont les plus proches de X' (au sens de δ) ;
2. On attribue à X' l'étiquette la plus fréquente dans ce voisinage.

EXEMPLE D'UTILISATION DE L'ALGORITHME DES K-VOISINS VU EN TP AVEC LA DISTANCE EUCLIDIENNE.

V.2 Tests et matrice de confusion

En pratique on teste de façon systématique un algorithme supervisé. On procède en partant d'un ensemble de données étiquetées ou pour lesquelles les sorties sont connues, X. On sélectionne un sous-ensemble des données qui sera l'ensemble d'apprentissage, $A \subset X$. On compare alors les prédictions que fournit l'algorithme pour des éléments $x \in X \backslash A$ aux étiquettes connues. La matrice de confusion de ces deux étiquetages rassemble tous ces éléments de comparaison.

Définition 7.18: matrice de confusion

Soient X un ensemble de données, $\mathcal{L} = \{e_0, e_1, \ldots\}$ un ensemble fini d'étiquettes et f_1, f_2 deux applications $f_i : X \to \mathcal{L}$. La matrice de confusion de f_1, f_2 est la matrice M de taille $|\mathcal{L}| \times |\mathcal{L}|$ telle que $M[i,j] = \left|\{x \in X / f_1(x) = e_i, f_2(x) = e_j\}\right|$.

Remarque 7.19. *Si la matrice est diagonale, alors f_1 et f_2 étiquettent de la même manière. En pratique, si on a un jeu de tests, il sera important que la fonction f obtenue par l'algorithme ressemble à l'hypothétique et que la matrice de confusion des deux fonctions soit diagonale ou quasi-diagonale.*

VI Classification non supervisée, algorithme des k-moyennes

L'algorithme de classification que nous présentons ici est un algorithme non supervisé : aucune information préalable sur une partition, aucun exemple de partie ne sont fournis en donnée. Seul le nombre de classes dans la partition que l'on veut obtenir est précisé.

Théorème 7.20: Algorithme des k-moyennes

- On dispose de M documents notés $(D_0, D_1, \ldots, D_{M-1})$; ces documents peuvent être des textes, des images, des relevés statistiques provenant de mesures biologiques comme de comportements d'achats, etc.
- Chaque document est représenté par un vecteur $X_j \in \mathbb{R}^N$;
- On suppose que l'on dispose d'une fonction d'écart entre les documents dist (D_i, D_j) qui vérifie l'inégalité triangulaire.

Alors, la fonction : `k_moyennes(X,k)` prend en arguments un ensemble de documents $X = (X_j)_j$, un entier $k \geq 1$ et renvoie un tuple (P, C) où P est une liste de k parties formant une partition de l'ensemble des documents et C est la liste des k centres de gravité de ces parties.

Algorithme 7.21

- On se donne ou on choisit aléatoirement k documents dans $(D_j)_j$ pour initialiser la liste C. Ils sont notés $C_0, \ldots, C_{k-1}$.
- On réserve un ensemble P de k listes vides ;
- Pour chaque document représenté par X_j on calcule les k distances dist (D_j, C_ℓ) , $\ell = 0, \ldots, k-1$. Si la distance minimale est dist $(D_j, C_{\ell_{min}})$, on place D_j dans la classe $\ell_{\min}$ (on place donc j dans la liste $P[\ell_{\min}]$).
- On dispose d'une première partition, on calcule le centre de gravité de chaque classe. La liste C est mise à jour avec les k centres de gravité : $C = [C_0, \ldots C_{k-1}]$.
- On itère le processus jusqu'à ce que les classes ne soient plus modifiées ou en utilisant une autre propriété à définit selon l'utilisation.

RÉSOLUTION D'ÉQUATIONS DIFFÉRENTIELLES

Sommaire

Le but de ce cours est de construire des méthodes numériques pour résoudre des équations différentielles. Les équations différentielles interviennent dans différentes filières en CPGE (maths, physique, chimie ou encore SI) et ne sont pas toujours résolubles (problèmes des trois corps par exemple). Cependant, il existe plusieurs méthodes numériques permettant d'en approcher les solutions. Nous allons, ici en regarder quelques unes.

I Équation différentielle d'ordre 1

Dans un premier temps, nous allons nous intéresser aux équations différentielles d'ordre 1 mis en problème de Cauchy : On cherche à trouver y dérivable sur $[t_0, T]$ telle que :

$$\begin{cases} y'(t) = F(t, y(t)) \\ y(t_0) = y_0 \end{cases} \tag{8.1}$$

sur $[t_0, T]$ avec F une fonction continue de $[t_0, T] \times \mathbb{R}$. Cette écriture regroupe à la fois les équations dites "linéaires", en prenant $F : (t, y) \to a(t)y$ ce qui donne :

$$y'(t) = a(t)y(t)$$

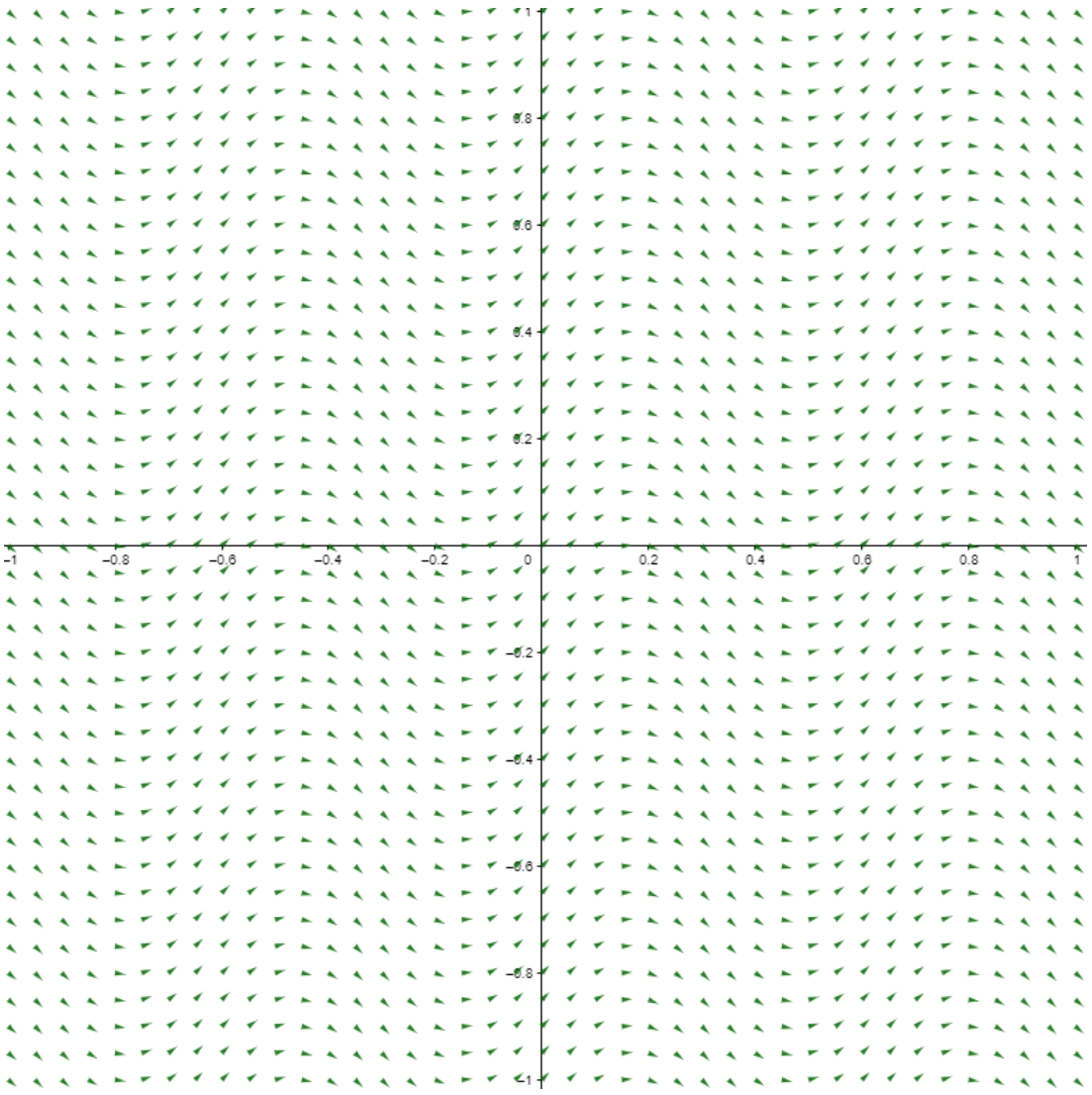
mais également des équations plus difficiles à résoudre et non linéaires, en prenant, par exemple $F(t, y) \to \cos(ty)$ ce qui donne :

$$y'(t) = \cos(ty(t)).$$

I.1 Principe

Exercice 8.1: On s'intéresse à l'équation $y'(t) = \cos(10t)$ sur le segment $[-1, 1]$, sur le graphe suivant, chaque vecteur a pour direction la même direction que la tangente à la courbe de la fonction solution de l'équation précédente et passant par ce point. Sans la calculer, dessiner en ligne brisée la fonction solution valant 0 en 0 (on placera un point tout les 0.1 unité)

E



Principe 8.2: Approximation avec la donnée de la dérivée

Soit y une fonction dérivable sur un intervalle $[t_0, T]$, il existe une fonction ϵ de limite nulle en 0 tel que :

$$y(t_0 + h) = y(t_0) + hy'(t_0) + \epsilon(h)$$

ainsi, si h est "petit" on peut dire :

$$y(t_0 + h) \approx y(t_0) + hy'(t_0)$$

et plus h est petit, plus l'approximation devient précise.

I.2 Méthode d'Euler Explicite

Si y est une solution du problème de Cauchy 8.1, l'idée est de construire point par point une approximation de la solution en partant du seul point qu'on connait de la courbe représentative de y, c'est-à-dire (t_0, y_0). On détermine la valeur de $y'(t_0)$ grâce à l'équation différentielle et on approxime la valeur de la solution en $t_0 + h$ avec la méthode précédente :

$$\boxed{y(t_0+h) \approx y(t_0) + hF(t_0, y(t_0))}$$

En effet, on rappelle que $y'(t_0) = F(t_0, y(t_0))$. On note alors $y_0 = y(t_0)$ et y_1 la valeur approchée de $y(t_0+h)$ c'est-à-dire :

$$y_1 = y_0 + hF(t_0, y_0)$$

en connaissant, y_1 une valeur approchée de $y(t_0+h)$ on connait également une valeur approchée de la dérivée car $y'(t_0+h) = F(t_0+h, y(t_0+h))$ on alors $y'(t_0+h) \approx F(t_0+h, y_1)$. On peut alors construire y_2 une approximation de $y(t_0+2h)$ de la manière suivante :

$$\boxed{y_2 = y_1 + hF(t_0+h, y_1)}$$

et ainsi de suite. Pour construire une solution sur l'intervalle $[t_0, T]$ de notre équation, on choisit h, qui est le pas temporelle entre deux points du graphe de notre solution, de manière équirépartie sur tout l'intervalle : si on veut construire N points de notre solution on prend le pas $h = \dfrac{T-t_0}{N}$. La méthode d'Euler est une méthode itérative : à partir des valeurs t_0, T, $y(t_0)$, N et de la fonction F, on construit deux suites (t_n) et (y_n) comme suit :

$$\begin{cases} t_{n+1} = t_n + \dfrac{T-t_0}{N} = t_n + h \\ y_{n+1} = y_n + hF(t_n, y_n) \end{cases}$$

La valeur de h est alors appelé le pas de discrétisation en temps.

Définition 8.3: Méthode d'Euler explicite avec un pas régulier

Soit y une solution de l'équation :

$$\begin{cases} y'(t) = F(t, y(t)) \\ y(t_0) = y_0 \end{cases}$$

sur $[t_0, T]$ avec F une fonction continue de $[t_0, T] \times \mathbb{R}$. Soit $N \in \mathbb{N}^*$ Si on note le pas de temps $h = \dfrac{T-t_0}{N}$, la **méthode d'Euler explicite** permet de construire un $(N+1)$-uplet $(y_k)_{k\in[\![0,N]\!]}$ approchant les valeurs du $(N+1)$-uplet $(y(t_0+kh))_{k\in[\![0,N]\!]}$. La suite est définie par récurrence comme suit :

$$\begin{cases} t_{n+1} = t_n + h \\ y_{n+1} = y_n + hF(t_n, y_n) \end{cases}$$

I.3 Exemples d'utilisation

Approcher l'exponentielle

Commençons par un problème simple où nous approchons la fonction exponentielle sur $[0, 1]$ via l'équation différentielle pour laquelle elle est solution :

$$\begin{cases} y'(t) = y(t) \\ y(0) = 1 \end{cases} \tag{8.2}$$

Exercice 8.4: On note $h = 1/N$ où N est le nombre de points crée par la méthode d'Euler explicite. Quelle la suite $(y_k)_{k\in[\![0,N]\!]}$ approchant les valeurs du $(N+1)$-uplet $(\exp(h/N))_{k\in[\![0,N]\!]}$ ainsi crée ? Spécifier surtout y_N approchant le nombre de Neper.

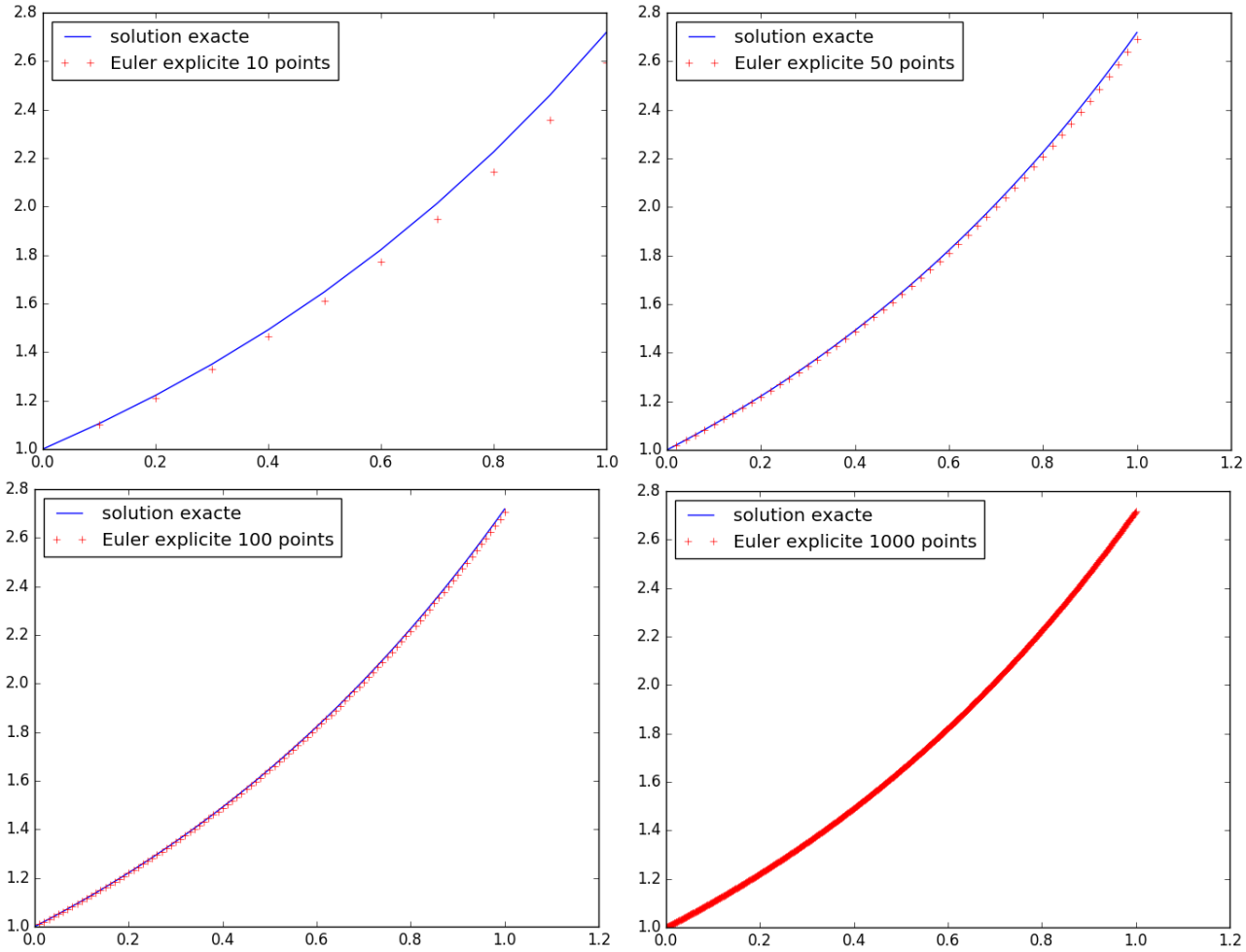
E

E

En voici le code :

méthode d'Euler explicite

```
import numpy as np
import matplotlib.pyplot as plt
t_0=0
y_0=1
T=1
N=1000
h=(T-t_0)/N
list_ye=[y_0]
list_t=[t_0]

y=y_0
t=t_0
ye=y_0
for k in range(N):
    ye=ye+h*ye
    t=t+h
    list_ye.append(ye)
    list_t.append(t)

sol=np.exp(list_t)

b,=plt.plot(list_t,sol,'-b',label='solution exacte')
c,=plt.plot(list_t,list_ye,'+r',label='Euler explicite '+str(N)+' points')

plt.legend(handles=[b,c],loc=2)
plt.show()
```



Approcher la charge du circuit RC

Le second problème est d'approcher la charge d'un circuit RC sur $[0, 1]$ via l'équation différentielle pour laquelle elle est solution :

$$\begin{cases} y'(t) = \dfrac{1}{RC}\left(E - y(t)\right) \\ y(0) = 0 \end{cases}$$

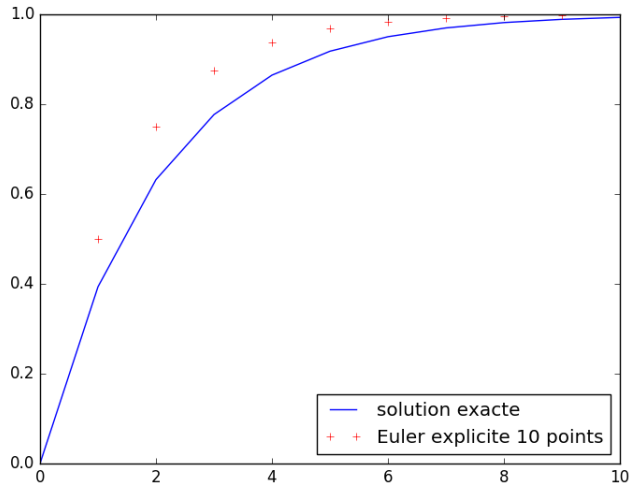
Le pas de temps est alors $h = 1/N$ où N est le nombre de points crée par la méthode. La méthode d'Euler explicite est alors $y_0 = 0$ et $t_0 = 0$ puis :

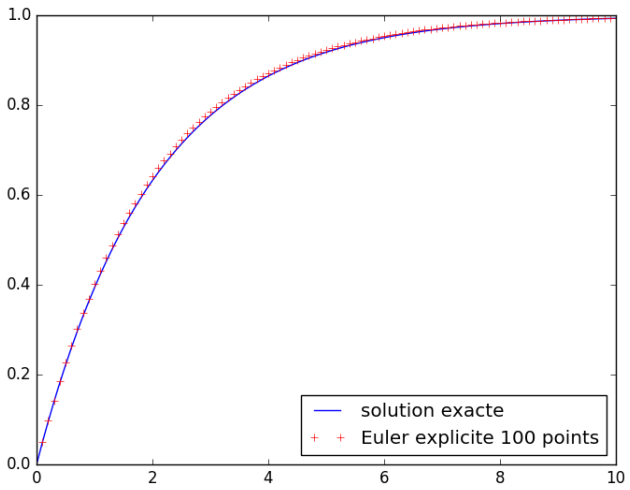
$$\begin{cases} t_{n+1} = t_n + \dfrac{1}{N} = t_n + h \\ y_{n+1} = y_n + \dfrac{h}{RC}\left(E - y_n\right) \end{cases}$$

En voici le code :

méthode d'Euler explicite

```
import numpy as np
import matplotlib.pyplot as plt
E=1;RC=2;t_0=0;y_0=0;T=10;N=10
h=T/N
list_y=[y_0]
list_t=[t_0]

y=y_0
t=t_0
for k in range(N):
    y=y+h*(E-y)/RC
    t=t+h
    list_y.append(y)
    list_t.append(t)

sol=E*(1-np.exp(-np.array(list_t)/RC))

b,=plt.plot(list_t,sol,'-b',label='solution exacte')
c,=plt.plot(list_t,list_y,'+r',label='Euler explicite '+str(N)+' points')

plt.legend(handles=[b,c],loc=4)
plt.show()
```





Approcher une solution qu'on ne connait pas formellement

Si on connait des méthodes de calculs qui nous permettent de connaître la solution formelle des deux dernières équations, ce n'est pas le cas de la suivante :

$$\begin{cases} y'(t) = \cos(ty(t)) \\ y(0) = 0 \end{cases}$$

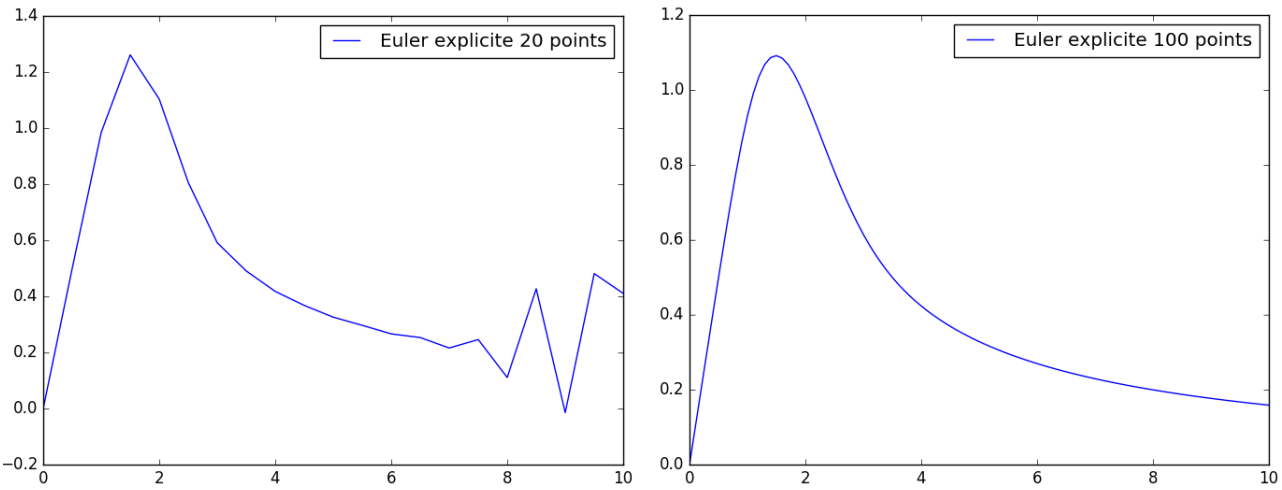
On cherche à résoudre cette équation sur $[0, 10]$. Le pas de temps est alors $h = 10/N$ où N est le nombre de points crée par la méthode. La méthode d'Euler explicite est alors $y_0 = 0$ et $t_0 = 0$ puis :

$$\begin{cases} t_{n+1} = t_n + \dfrac{10}{N} = t_n + h \\ y_{n+1} = y_n + h\cos(t_n y_n) \end{cases}$$

dont voici le code :

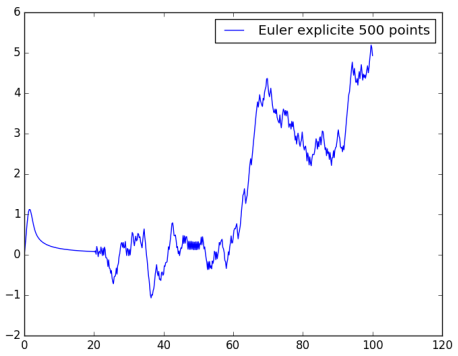
méthode d'Euler explicite

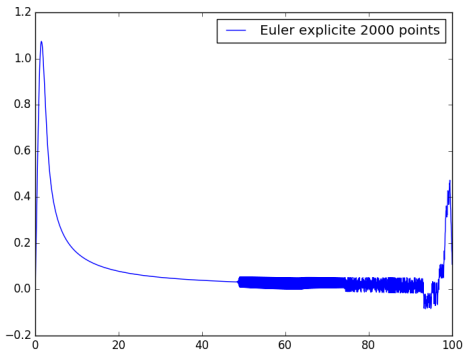
```
from math import cos
import numpy as np
import matplotlib.pyplot as plt
t_0=0;y_0=0;T=10;N=20;h=T/N
list_y=[y_0]
list_t=[t_0]

y=y_0
t=t_0
for k in range(N):
    y=y+h*cos(t*y)
    t=t+h
    list_y.append(y)
    list_t.append(t)



c,=plt.plot(list_t,list_y,'-',label='Euler explicite '+str(N)+' points')

plt.legend(handles=[c],loc=1)
plt.show()
```



I.4 Limite de la méthode

Bien que cet algorithme semble fonctionner à merveille sur les exemples énoncés, il faut tout de même faire attention : cet algorithme ne donne pas toujours de bonne voire d'approximation (tout court) de la solution. L'étude de la convergence de tel algorithme est une part importante de la recherche internationale en Mathématiques (Analyse numérique et Équations aux dérivées partielles). Pour illustrer ces propos, on peut déjà remarqué que, sous certaine condition, l'algorithme donne des résultats "étranges" pour la dernière équation. Par exemple, dans le cas où $T = 10$ et $N = 500$ ou $N = 2000$ (voire les figures ci-dessous).





HP Voir : consistance et stabilité d'un schéma ou encore la condition Courant–Friedrichs–Lewy (CFL)

I.5 Méthode d'Euler implicite

Le principe est similaire à la méthode d'Euler explicite, sauf qu'on utilise le taux d'accroissement pour la dérivée en t_{n+1} et non t_n ce qui donne :

$$\boxed{y(t_{n+1}) \approx y(t_n) + hy'(t_{n+1})}$$

la méthode d'Euler implicite en découle : $y_0 = y(t_0)$ et $t_0 = t_0$ puis :

$$\begin{cases} t_{n+1} = t_n + \dfrac{T - t_0}{N} = t_n + h \\ y_{n+1} = y_n + hF(t_{n+1}, y_{n+1}) \end{cases}$$

l'expression de y_{n+1} n'est plus donnée directement. Un calcul supplémentaire est donc nécessaire pour trouver y_{n+1} (souvent, résoudre un système linéaire).

Définition 8.5: Méthode d'Euler implicite avec un pas régulier

Soit y une solution de l'équation :

$$\begin{cases} y'(t) = F(t, y(t)) \\ y(t_0) = y_0 \end{cases}$$

sur $[t_0, T]$ avec F une fonction continue de $[t_0, T] \times \mathbb{R}$. Soit $N \in \mathbb{N}^*$ Si on note le pas de temps $h = \dfrac{T - t_0}{N}$, la **méthode d'Euler explicite** permet de construire un $(N+1)$-uplet $(y_k)_{k \in [\![0,N]\!]}$ approchant les valeurs du $(N+1)$-uplet $(y(t_0 + kh))_{k \in [\![0,N]\!]}$. La suite est définie par récurrence comme suit :

$$\begin{cases} t_{n+1} = t_n + h \\ y_{n+1} = y_n + hF(t_{n+1}, y_{n+1}) \end{cases}$$

Par exemple, dans le cas de l'exponentielle on a alors le schéma suivant : $y_0 = 1$ et $t_0 = 0$ puis :

$$\begin{cases} t_{n+1} = t_n + \dfrac{1}{N} = t_n + h \\ y_{n+1} = y_n + hy_{n+1} \end{cases}$$

ce qui revient à :

$$\begin{cases} t_{n+1} = t_n + \dfrac{1}{N} = t_n + h \\ y_{n+1} = \dfrac{y_n}{1 - h} \end{cases}$$

II Équations différentielles du second ordre

La méthode d'Euler peut être également utilisée pour résoudre des équations différentielles du second ordre. L'idée est de transformer cette équation en un système d'équations du premier ordre. On souhaite résoudre sur $[t_0, T]$ le problème dit de "Cauchy" suivant :

$$\begin{cases} y''(t) = F(t, y(t), y'(t)) \\ y(t_0) = y_0 \\ y'(t_0) = z_0 \end{cases}$$

où F est une fonction continue de $[t_0, T] \times \mathbb{R}^2$. On le transforme en un système de deux équations du premier ordre avec

$$\begin{cases} y'(t) = z(t) \\ z'(t) = F(t, y(t), z(t)) \\ y(t_0) = y_0 \\ z(t_0) = z_0 \end{cases}$$

On résout alors via la méthode d'Euler explicite vu précédemment, on contruit trois suites (t_n), (y_n) et (z_n) tel que :

$$\begin{cases} t_{n+1} = t_n + \dfrac{T - t_0}{N} = t_n + h \\ y_{n+1} = y_n + hz_n \\ z_{n+1} = z_n + hF(t_n, y_n, z_n) \end{cases}$$

II.1 Exemple : tension aux bornes du condensateur sur un circuit RLC

$$\begin{cases} y''(t) = \dfrac{E}{LC} - \dfrac{R}{L}y'(t) - \dfrac{1}{LC}y(t) \\ y(t_0) = 0 \\ y'(t_0) = 0 \end{cases}$$

avec $t_0 = 0, R = 1000\Omega, L = 0.1H, C = 10^{-9}F$ et $T = 0.001$. On rappelle que la solution analytique de ce problème est donnée par :

$$y : t \to E - Ee^{\frac{R}{2L}t}\left(\cos(\omega t) + \frac{R}{2L\omega}\sin(\omega t)\right)$$

avec $\omega_0 = \frac{1}{\sqrt{LC}}$, $Q = \frac{1}{R}\sqrt{\frac{L}{C}}$ et $\omega = \omega_0\sqrt{1 - \frac{1}{4Q^2}}$.

Exercice 8.6: Construire les suites (t_n), (y_n) et (z_n) permettant d'approcher les solutions de cette équation

Démonstration. Déterminons une solution numérique grâce à la méthode d'Euler explicite : on construit alors les suites (t_n), (y_n) et (z_n) vérifiant :

E

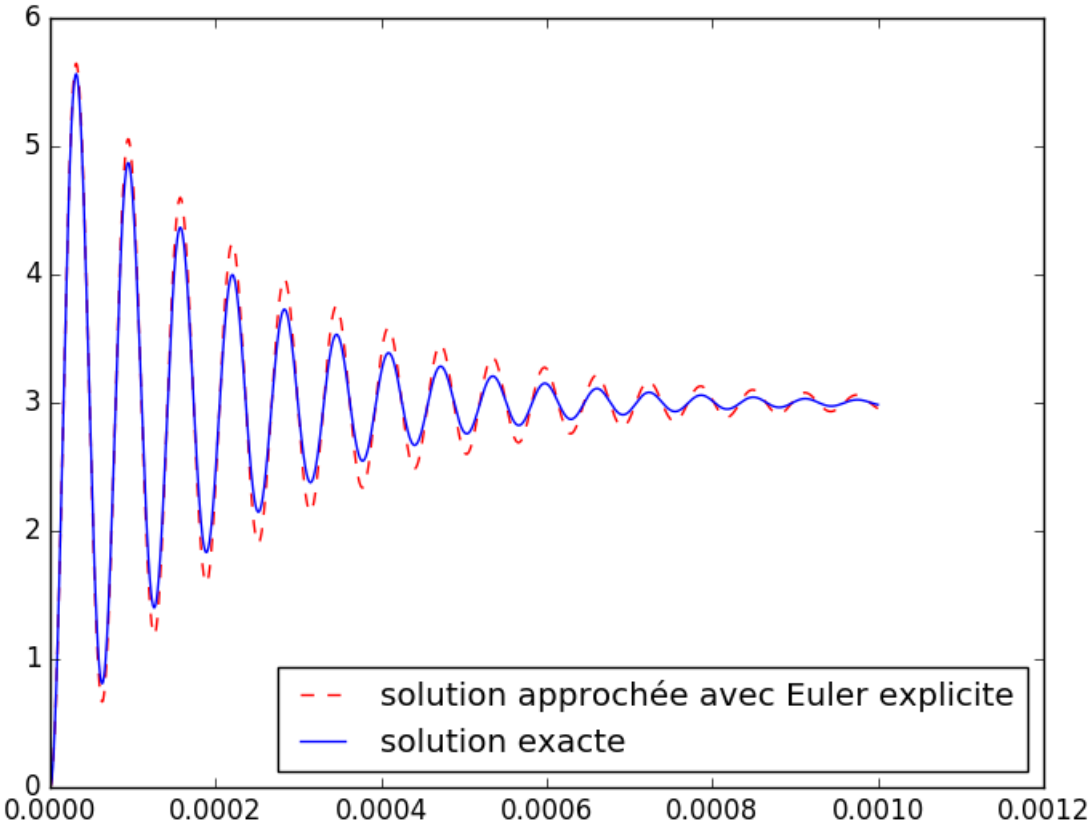
$$\begin{cases} t_{n+1} = t_n + \dfrac{T - t_0}{N} = t_n + h \\ y_{n+1} = y_n + hz_n \\ z_{n+1} = z_n + h\left(\dfrac{E}{LC} - \dfrac{R}{L}z_n - \dfrac{1}{LC}y_n\right) \end{cases}$$

□

On obtient le programme python suivant (voir page suivante)

méthode d'Euler explicite

```
import numpy as np
import matplotlib.pyplot as plt
E=3;R=1000;L=0.1;C=10**(-9)
t_0=0;y_0=0;z_0=0
T=0.001;N=5000;h=T/N

# constantes
om0=1/(np.sqrt(L*C))
Q=(1/R)*np.sqrt(L/C)
om=om0*np.sqrt(1-1/(4*Q**2))
list_y=[y_0]
list_z=[z_0]
list_t=[t_0]

y=y_0
t=t_0
z=z_0
for k in range(N):
    temp=y
    y=y+h*z
    z=z+h*((E/(L*C))-(R/L)*z-(1/(L*C))*temp)
    t=t+h
    list_z.append(z)
    list_y.append(y)
    list_t.append(t)

list_t=np.array(list_t)

sol=E-E*np.exp(-(R/(2*L))*list_t)*(np.cos(om*list_t)+(R/(2*L*om))*np.sin(om*list_t))

a,=plt.plot(list_t,list_y,'--r',label='solution approchée avec Euler explicite')
b,=plt.plot(list_t,sol,'-b',label='solution exacte')

plt.legend(handles=[a,b],loc=4)
plt.show()
```



II.2 Équation du pendule

Exercice 8.7: Expliquer la formule d'Euler explicite sur cet exemple et donner un algorithme permettant de calculer une solution approchée via cette méthode.

$$\theta'' + \frac{g}{l}\sin(\theta) = 0$$

avec $l = 50$ cm par exemple, et $\theta(0) = \pi/4$ et $\theta'(0) = 0$.

E

E

III Utilisation de odeint

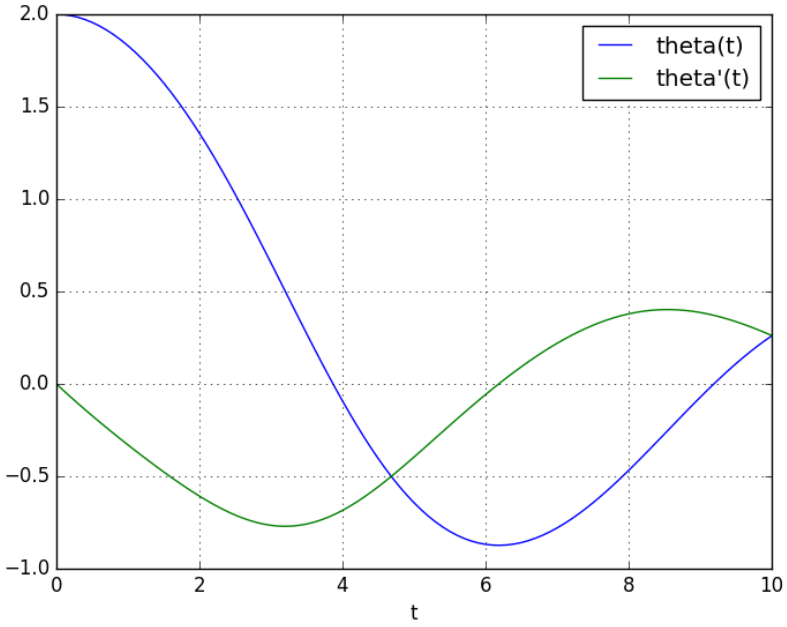
Odeint est une fonction de scypy qui permet de calculer une solution approchée d'une équation différentielle avec une méthode plus élaborée que Euler explicite ou implicite. En voici un exemple d'utilisation sur l'équation :

$$\theta''(t) + b\theta'(t) + c\sin(\theta(t)) = 0$$

avec $b = 0.25$, $c = 0.4$, $\theta(0) = 2$, $\theta'(0) = 0$, $t_0 = 0$ et $T = 10$.

```
def pend(y, t, b, c): #fonction renvoie le vecteur [theta',theta''] en fonction
        theta, thetaprim = y   #de  [theta,theta']
        dydt = [thetaprim, -b*thetaprim - c*np.sin(theta)]
        return dydt

b =0.25 #paramètres
c=0.4
y0=[2, 0.0] # listes des valeurs initiales de theta et theta'
t = np.linspace(0, 10, 101) #répartitions des points représentants la solution
# t_0=0, T=10, N+1=101
from scipy.integrate import odeint
sol = odeint(pend, y0, t, args=(b, c)) #calcule la solution

import matplotlib.pyplot as plt
plt.plot(t, sol[:, 0], 'b', label='theta(t)') #sol est un tableau
# la première colonne contient theta
```

```
plt.plot(t, sol[:, 1], 'g', label="theta'(t)") #la seconde theta'
plt.legend(loc='best')
plt.xlabel('t')
plt.grid() # permet de faire une grille
plt.show()
```



Les arguments à donner la fonction `odeint` sont les suivants (et dans l'ordre) :

- La fonction associée au problème de Cauchy `F(y,t)`. Attention, la variable y doit être avant la variable t. Il est possible de rajouter des arguments à F, il faut alors le spécifier à la fonction `odeint` comme dans l'exemple ci-dessus.
- La liste des conditions initiales `y0`
- La liste des temps pour l'extrapolation `t`

IV Fonctions Euler explicite sur Python

De manière générale, on pourra réécrire et utiliser les fonctions python suivante en TP :

La fonction suivante prend en argument la fonction F, les réels t_0, y_0, T et l'entier N renvoie le $(N+1)$-uplet $(y_k)_{k\in[\![0,N]\!]}$ approchant les valeurs du $(N+1)$-uplet $(y(t_0+kh))_{k\in[\![0,N]\!]}$ et le $(N+1)$-uplet des temps $(t_k)_{k\in[\![0,N]\!]}$. On rappelle que la suite est définie par récurrence comme suit :

$$\begin{cases} t_{n+1} = t_n + h \\ y_{n+1} = y_n + hF(t_n, y_n) \end{cases}$$

Fonction Euler Explicite

```
def EulerExp(F, t_0, y_0, T, N):
    """
    Méthode d'Euler explicite pour résoudre une équation différentielle d'ordre 1.

    Arguments:
    - F: fonction qui prend comme arguments t et y et
            retourne la dérivée de y par rapport à t
    - t_0: temps initial
    - y_0: valeur initiale de la solution
    - T: temps final
    - N: nombre de pas

    Retourne:
    - list_t: liste des temps
    - list_y: liste des solutions
    """
    list_y = [y_0]
    list_t = [t_0]
    h = (T - t_0) / N
    y = y_0
    t = t_0
    for k in range(N):
        y = y + h * F(t, y)
        t = t + h
        list_y.append(y)  # ou list_y=list_y+[y]
        list_t.append(t)  # ou list_t=list_t+[t]
    return list_t, list_y
```

Un exemple d'utilisation lié à l'équation :

$$\begin{cases} y'(t) = \cos(ty(t)) \\ y(0) = 0 \end{cases}$$

Exemple

```
from math import cos
import numpy as np
import matplotlib.pyplot as plt
t_0=0;y_0=0;T=10;N=2000;h=T/N

def Fonc(t,y):
    return np.cos(t*y)

list_t,list_y=EulerExp(Fonc,t_0,y_0,T,N)

c,=plt.plot(list_t,list_y,'-',label='Euler explicite '+str(N)+' points')
plt.legend(handles=[c],loc=1)
plt.show()
```

Attention 8.8

La fonction donnée en argument à la fonction `EulerExp` doit avoir deux arguments `t,x` même si la fonction dépend que de x. Par exemple, pour l'équation différentielle $y' = y$, la fonction a donnée en argument est :

Exemple

```
def Fonc(t,y):
    return y
```

IV.1 Pour les équations d'ordre 2

Euler explicite pour l'ordre 2

```
def EulerExp2(F, t_0, y_0, z_0, T, N):
    """
    Calcule la solution approchée d'une équation différentielle d'ordre 2
    en utilisant la méthode d'Euler explicite.

    Arguments :
    F : fonction dérivée de y'' = F(t, y, z) qui représente l'équation différentielle
    t_0 : temps initial
    y_0 : valeur initiale de y
    z_0 : valeur initiale de y' (la dérivée de y par rapport au temps)
    T : temps final jusqu'auquel la solution est calculée
    N : nombre de pas de discrétisation pour l'algorithme d'Euler

    Retourne :
    list_t : liste des temps discrétisés
    list_y : liste des valeurs de la solution y
    list_z : liste des valeurs de la dérivée de la solution z = y'

    Remarque :
    La fonction retourne trois listes : les temps, la solution y et la dérivée z.
    """
    # Initialisation des listes pour stocker les valeurs
    list_y = [y_0]
    list_z = [z_0]
    list_t = [t_0]

    # Calcul du pas de temps
    h = (T - t_0) / N

    # Initialisation des variables
    y = y_0
    t = t_0
    z = z_0

    # Boucle principale de l'algorithme d'Euler
    for k in range(N):
        # Stockage temporaire de la valeur précédente de y
        temp = y
```

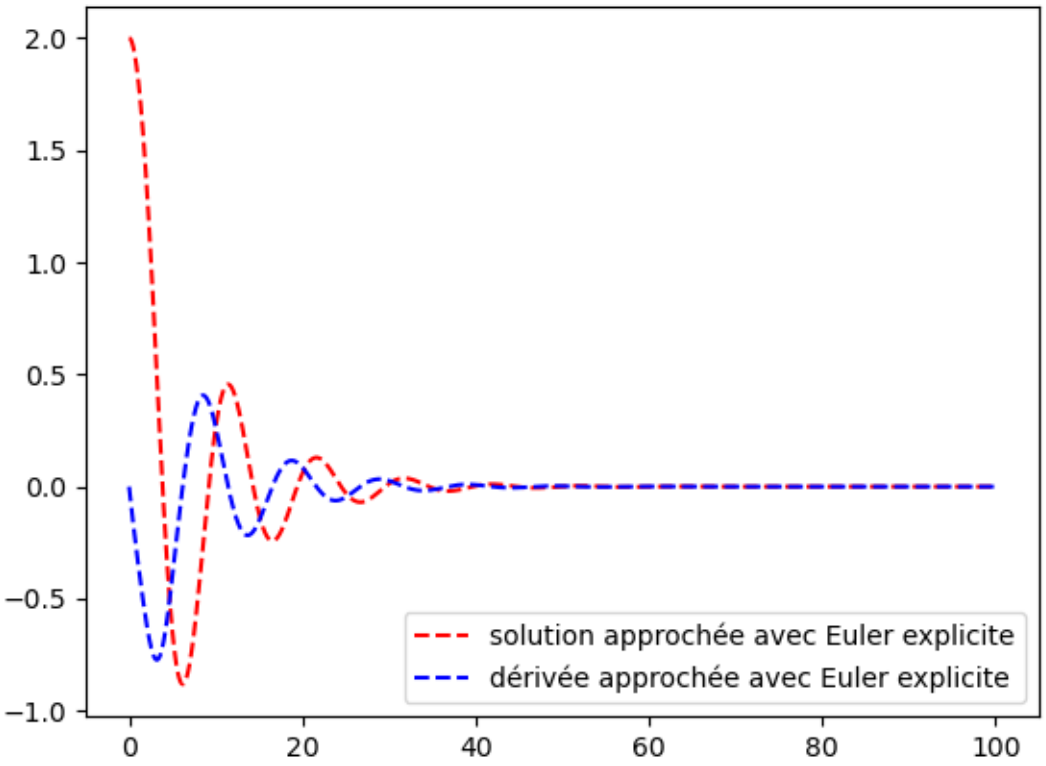
```
        # Mise à jour de y et z à l'aide de la méthode d'Euler
        y = y + h * z
        z = z + h * F(t, temp, z)

        # Mise à jour du temps
        t = t + h

        # Ajout des valeurs calculées aux listes
        list_z.append(z)
        list_y.append(y)
        list_t.append(t)

    return list_t, list_y, list_z
```

Voici un exemple d'utilisation pour

$$\theta''(t) + b\theta'(t) + c\sin(\theta(t)) = 0$$

avec $b = 0.25$, $c = 0.4$, $\theta(0) = 2$, $\theta'(0) = 0$, $t_0 = 0$ et $T = 10$.
Ici, $F(t, y, z) = -bz - c\sin(y)$ ne dépend pas de t.

```
import numpy as np
import matplotlib.pyplot as plt
b=0.25;c=0.4;y_0=2;z_0=0;t_0=0
T=100; N=10000; h=(T-t_0)/N

def Fonc(t,y,z):
    return -b*z-c*np.sin(y)

list_t,list_y,list_z=EulerExp2(Fonc,t_0,y_0,z_0,T,N)

a,=plt.plot(list_t,list_y,'--r',label='solution approchée avec Euler explicite')
b,=plt.plot(list_t,list_z,'--b',label='dérivée approchée avec Euler explicite')
plt.legend(handles=[a,b],loc=4)
plt.show()
```


2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
0
20
40
60
80
100
solution approchée avec Euler explicite
dérivée approchée avec Euler explicite