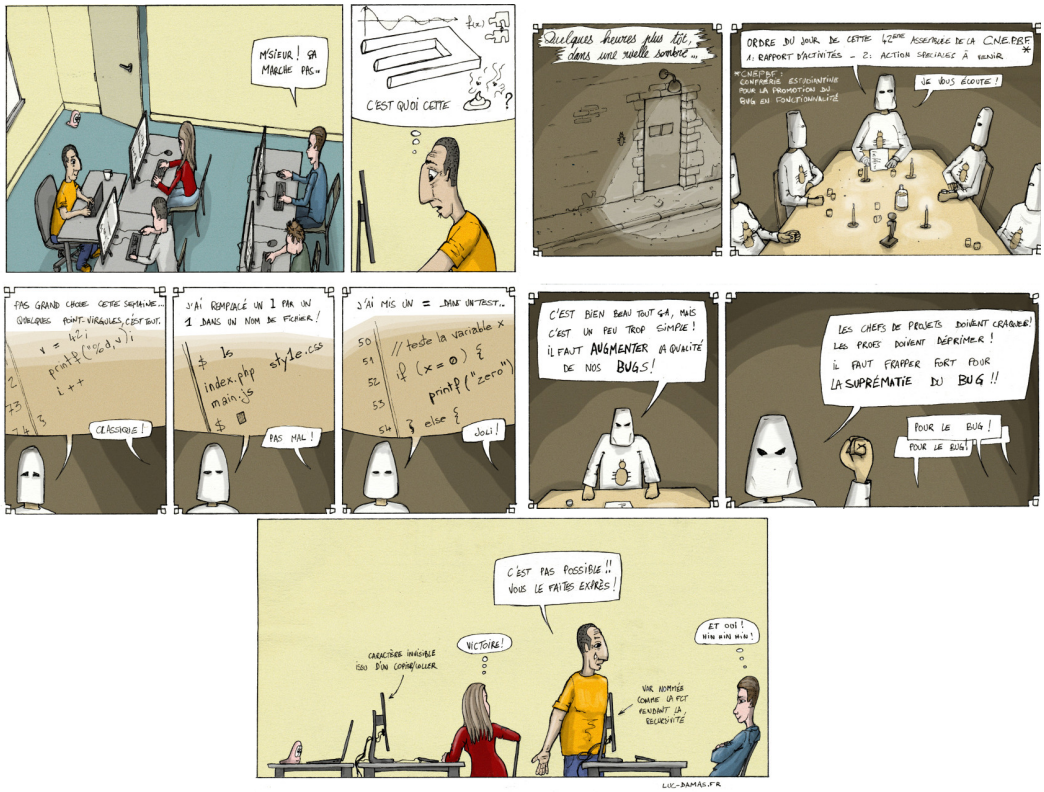
Informatique commune

MP



Lycée Sainte Geneviève

2024-2025

Table des matières

1 Bases de données **3**
- 1 Les bases de l'algèbre relationnelle . . . 4
- 2 Commandes simples . . . 7
- 3 Clefs . . . 11
- 4 Opérations avancées . . . 15

2 Révisions sur les graphes **22**
- 1 Notions théoriques sur les graphes . . . 23
- 2 Parcours d'un graphe . . . 27
- 3 Recherche d'un plus court chemin . . . 32

3 Algorithmes de tri **37**
- 1 Quelques définitions . . . 38
- 2 Petit point sur Python . . . 39
- 3 Algorithmes de tri . . . 40

4 Dictionnaires et programmation dynamique **49**
- 1 Tableaux et listes chaînées . . . 50
- 2 Piles et files . . . 53
- 3 Dictionnaires et fonctions de hachage . . . 58
- 4 Programmation dynamique . . . 64

5 Apprentissage supervisé et théorie des jeux **69**
- 1 Jeux sur un graphe simple : le Chomp . . . 70
- 2 Algorithme `min-max` . . . 75
- 3 Intelligence artificielle et apprentissage . . . 78

ITC 1

Bases de données

Sommaire

1 Les bases de l'algèbre relationnelle 4
1.1 Les limites de la représentation plane 4
1.2 Définitions 5
2 Commandes simples 7
2.1 Projection 7
2.2 Sélection simple 8
2.3 Opérateurs ensemblistes usuels 9
2.4 Renommage 10
2.5 Filtrage des résultats 10
3 Clefs 11
3.1 Définition 11
3.2 Clef primaire et clef étrangère 12
3.3 Associations 13
4 Opérations avancées 15
4.1 Produit cartésien 15
4.2 Jointure 15
4.3 Agrégation 18

1 Les bases de l'algèbre relationnelle

1.1 Les limites de la représentation plane

On appelle *représentation plane* la présentation de données sous la forme d'un tableau à deux entrées. Prenons pour exemple une liste de véhicules ci-dessous.

Marque	Modèle	Carburant	Cylindrée (cm^3)
Citroën	C4 Picasso	Diesel	1997
	C4 Picasso	Essence	1598
Volkswagen	Jetta	Essence	1197
	Jetta	Diesel	1598
Porsche	911 Carrera	Essence	3436
	911 GT3 RS	Essence	3996

En Python, on pourrait représenter cette liste sous la forme d'une liste de listes.

Code Python

Mode éditeur

```
voitures=[
["Citroen",
      ["C4 Picasso", "Diesel", "1997"],
      ["C4 Picasso", "Essence", "1598"]],
["Volkswagen",
      ["Jetta", "Essence", "1197"],
      ["Jetta", "Diesel", "1598"]],
["Porsche",
      ["911 Carrera 4S", "Essence","3800"],
      ["911 GT3 RS", "Essence","3996"]]
]
```

Application

ITC1.1 : Liste des véhicules-constructeur

.

Écrire une fonction permettant d'afficher la liste des véhicules d'un constructeur donné.

Application

ITC1.2 : Liste des véhicules-carburant

.

Écrire une fonction permettant d'afficher la liste des véhicules utilisant un carburant donné.

Pour simplifier cette dernière fonction, on aurait pu choisir une liste dans laquelle les voitures étaient triées par carburant et pas par marque. En revanche, le listing par constructeur devenait alors plus complexe.
La représentation plane nous a forcé à donner la priorité à une des caractéristiques (ici, le constructeur) des voitures considérées.

1.2 Définitions

Dans le modèle relationnel, on fait abstraction de toute priorité de caractéristique. On représente chaque voiture par un n-uplet :

(Marque, Modèle, Carburant, Cylindrée)

Une telle représentation est appelée un **schéma relationnel** noté S.
Les différents éléments de cette relation (marque, modèle,...) sont appelés les **attributs** A_i de la relation et sont distincts deux à deux. Le schéma relationnel peut ainsi se mettre sous la forme :

$$S = (A_1, \dots, A_n)$$

Remarque importante

Les différents attributs peuvent prendre leurs valeurs dans des ensembles appelés **domaines** des attributs notes $\text{dom}(A_i)$. Il est d'usage de représenter un schéma relationnel en rappelant le domaine de chaque attribut :

$$S = ((A_1, \text{dom}(A_1), \dots, (A_n, \text{dom}(A_n))$$

Ainsi, dans le cas des listes de véhicules, le schéma relationnel serait :

$$S = ((\text{Marque}, \text{texte}), \dots, (\text{Cylindrée}, \mathbb{N}))$$

On notera $B \in S$ pour signifier que B est un des attributs de S.
De même, on notera $X \subset S$ pour signifier que X est un sous-n-uplet de S.

Exemple

On considère le schéma relationnel :

$$S = (\text{Marque, Modèle, Carburant, Cylindrée})$$

- $B = (\text{Modèle})$ est un attribut de S,
- $X = (\text{Marque, Modèle})$ est un sous-n-uplet de S.

On appelle **relation**, ou **table**, associée à un schéma relationnel S, un ensemble de n-uplets correspondant aux différentes valeurs prises par les attributs. Cette relation est notée $R(S)$.

Exemple

Dans le schéma relationnel S donné ci-dessus, la relation $R(S)$ associée au tableau des voitures est représentable par :

voitures1

marque	modèle	carburant	cylindrée
Citroën	C4 Picasso	Diesel	1997
Citroën	C4 Picasso	Essence	1598
Volkswagen	Jetta	Essence	1197
Volkswagen	Jetta	Diesel	1598
Porsche	911 Carrera	Essence	3436
Porsche	911 GT3 RS	Essence	3996

On notera $e \in R(S)$ un élément de la relation $R(S)$ et $e.A_i$ la valeur de l'attribut A_i associé à l'élément e.

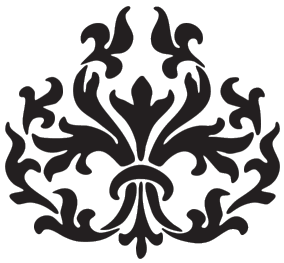
Exemple

Dans la relation définie précédemment, si on numérote les 6 lignes de 1 à 6, on a :

3.cylindrée = 1197

On notera $e(X)$ l'opération particulière qui consiste à exprimer tous les attributs de l'élément e pris dans le sous-ensemble X des attributs :

$$X = (B_1, \dots, B_m) \subset S, \quad e(X) = (e.B_1, \dots, e.B_m)$$



2 Commandes simples

2.1 Projection

2.1.1 Formalisme relationnel

La projection d'une relation consiste à ne conserver que certains attributs pour tous les éléments. Soit $R(S)$ une relation de schéma S et $X \subset S$. On appelle **projection** de R selon X la relation :

$$\pi_X(R) = \{e(X) | e \in R\}$$

On voit, qu'une projection ne contient pas forcément autant de valeurs de la relation de départ : certains éléments sont fusionnés si leurs valeurs après projection sont identiques.

2.1.2 Commande SQL

Un **s**ystème de **g**estion de **b**ase de **d**onnées (S.G.B.D.) est un logiciel (ou un ensemble de logiciels) permettant de manipuler les données d'une base de données. Manipuler, c'est-à-dire sélectionner et afficher des informations tirées de cette base, modifier des données, en ajouter ou en supprimer (ce groupe de quatre opérations étant souvent appelé *"CRUD"*, pour *Create, Read, Update, Delete*).

Un langage spécifique a été développé pour interagir avec les bases de données. Il s'agit du langage SQL (*Structured Query Language*). Il permet de traduire simplement les opérateurs de l'algèbre relationnelle en utilisant des mots clefs explicites. Il est utilisé par de nombreux S.G.B.D. (*MySQL, MariaDB, PostgreSQL, SQLite,...*).

Point d'attention

Toutes les requêtes SQL commencent par le mot clef `SELECT` et terminent par un point virgule.

La projection consiste à sélectionner certains attributs d'une relation, ce qui se traduit en langage SQL par :

Il est possible d'obtenir tous les éléments d'une table avec $*$:

Application

ITC1.3 : Projection

Projeter la relation *voitures1* selon les attributs marque et modèle, et comparer au résultat de la requête SQL correspondante.

2.2 Sélection simple

2.2.1 Formalisme relationnel

La sélection consiste à choisir les éléments de la relation vérifiant une certaine condition. Soit $R(S)$ une relation de schéma S, $A \in S$ et a une valeur possible de A. On appelle **sélection** de R selon $A = a$ la relation :

$$\sigma_{A=a}(R) = \{e \in R | e.A = a\}$$

Remarque importante

Si le domaine de A le permet (nombres entiers ou flottants), la sélection peut également porter sur une inégalité :

$$\sigma_{A \leqslant a}(R) = \{e \in R | e.A \leqslant a\}$$

Application

ITC1.4 : Sélection

Dans la relation *voitures1*, que donne la sélection des voitures diesel ? De même pour la sélection des voitures de plus d'1,7 L de cylindrée.

2.2.2 Commande SQL

On utilise encore la commande `SELECT` associée au mot clef `WHERE` pour imposer la (ou les) condition(s).

Point d'attention

Le mot clef `SELECT` permet de réaliser à la fois les projections et les sélections.

Application

ITC1.5 : Sélection-SQL

Écrire les requêtes SQL correspondant aux deux sélections effectuées dans l'application précédente.

On peut alors combiner aisément la sélection avec une projection, pour afficher par exemple la liste des modèles existant en version diesel :

```
SELECT modèle FROM voitures WHERE carburant="Diesel";
```

2.3 Opérateurs ensemblistes usuels

2.3.1 Formalisme relationnel

Si deux relations ont le même schéma relationnel, il est possible de leur appliquer des opérateurs ensemblistes. Parmi ceux-ci, on aura principalement recours à l'union, l'intersection et la différence. Pour cette partie, on considère deux relations $vendeur_1$ et $vendeur_2$ contenant les listes de véhicules proposés par deux concessionnaires.

$vendeur_1$

marque	modèle	carburant	cyl.
Mercedes	C200	Diesel	2143
Subaru	WRX STI	Essence	2457
Bugatti	Chiron	Essence	7993

$vendeur_2$

marque	modèle	carburant	cyl.
Mercedes	C63 AMG	Essence	6208
Subaru	WRX STI	Essence	2457
Fiat	500	Essence	1242

- L'union $vendeur_1 \cup vendeur_2$ est une relation comprenant l'ensemble des éléments appartenant à $vendeur_1$ **ou** $vendeur_2$:

$vendeur_1 \cup vendeur_2$

marque	modèle	carburant	cyl.
Mercedes	C63 AMG	Essence	6208
Mercedes	C200	Diesel	2143
Subaru	WRX STI	Essence	2457
Bugatti	Chiron	Essence	7993
Fiat	500	Essence	1242

- L'intersection $vendeur_1 \cap vendeur_2$ est une relation comprenant l'ensemble des éléments appartenant à $vendeur_1$ **et** $vendeur_2$:

$vendeur_1 \cap vendeur_2$

marque	modèle	carburant	cyl.
Subaru	WRX STI	Essence	2457

- La différence $vendeur_1 - vendeur_2$ est une relation comprenant l'ensemble des éléments appartenant à $vendeur_1$ mais pas à $vendeur_2$:

$vendeur_1 - vendeur_2$

marque	modèle	carburant	cyl.
Mercedes	C200	Essence	1796
Bugatti	Chiron	Essence	7993

2.3.2 Commande SQL

- L'union est réalisée avec le mot clef `UNION` :

- L'intersection est réalisée avec le mot clef `INTERSECT` :

- La différence est réalisée avec le mot clef `EXCEPT` :

2.3.3 Application à la sélection composée

Comme chaque sélection donne une nouvelle relation, on peut utiliser les opérations ensemblistes sur les sélections pour exprimer des conditions complexes.

Application

ITC1.6 : Opérateurs ensemblistes

On souhaite obtenir la liste des véhicules essence de cylindrée inférieure à 1,5 L proposés par le $vendeur_1$.
Donner deux requêtes SQL permettant d'effectuer cette opération : l'une à l'aide d'opérateurs ensemblistes, et l'autre à l'aide d'opérateurs logiques.

2.4 Renommage

Le renommage sert à changer **temporairement** le nom d'un attribut. Ce renommage n'affecte que la requête en cours (en vue de l'affichage) mais ne modifie pas la structure relationnelle.

Exemple

Pour afficher la liste des véhicules proposés par le vendeur 1 avec des noms de colonne en anglais. La requête suivante donnerait :

```
SELECT marque AS brand, modèle AS model, carburant AS fuel FROM vendeur1;
```

$vendeur_1$

brand	model	fuel	cyl.
Mercedes	C200	Diesel	2143
Subaru	WRX STI	Essence	2457
Bugatti	Chiron	Essence	7993

2.5 Filtrage des résultats

À la toute fin d'une requête il est possible de trier les résultats et de n'en renvoyer qu'une partie. Ces opérations se notent :

- `ORDER BY a ASC` pour trier suivant l'attribut `a` par ordre croissant (on utilise le mot-clef `DESC` pour un tri par ordre décroissant) ;
- `LIMIT n` pour limiter la sortie à `n` enregistrements ;
- `OFFSET` pour débuter à partir du `n`-ième enregistrement.

Si l'on souhaite utilise simultanément ces fonctions de filtrage, elles doivent apparaître dans cet ordre.

Application

ITC1.7 : Filtrage

Q.1. Écrire une requête permettant d'obtenir l'ensemble des véhicules diesel dans la table *voiture1* triés par ordre alphabétique croissant de la marque.

Q.2. Écrire une requête permettant d'afficher les 3 voitures ayant la plus grosse cylindrée triées par ordre décroissant

Il est possible d'effectuer un tri sur plusieurs attributs en les séparant par une virgule :
`SELECT * FROM relation ORDER BY attr1,attr2 ASC;`
Dans ce cas, les données sont d'abord triées selon l'attribut *attr1*, les cas d'égalité sont triés avec l'attribut *attr2*.

3 Clefs

3.1 Définition

Une **clef** pour une relation est un ensemble d'attributs qui permet d'identifier chaque élément de la relation de manière unique.
Soit $R(S)$ une relation de schéma S et $K \subset S$. On dit que K est une clef pour R si et seulement si pour tous éléments $e_1, e_2 \in R$, on a :

$$e_1(K) = e_2(K) \Rightarrow e_1 = e_2$$

Exemple

Considérons la relation *voitures1* ci-dessous.
L'attribut *marque* ou l'ensemble (*marque, modèle*) ne constituent pas des clefs pour la relation car deux voitures ont pour attributs (*Volkswagen, Jetta*).

voitures1

marque	modèle	carburant	cylindrée
Citroën	C4 Picasso	Diesel	1997
Citroën	C4 Picasso	Essence	1598
Volkswagen	Jetta	Essence	1197
Volkswagen	Jetta	Diesel	1598
Porsche	911 Carrera	Essence	3436
Porsche	911 GT3 RS	Essence	3996

En revanche, l'ensemble (*modèle, carburant*) est bien une clef pour la relation.

Remarque importante

La relation ci-dessus pourrait être complétée en ajoutant les différentes finitions disponibles pour un même modèle (et même motorisation).

voitures

marque	modèle	carburant	cylindrée	finition
Citroën	C4 Picasso	Diesel	1997	Business
Citroën	C4 Picasso	Diesel	1997	Confort
Citroën	C4 Picasso	Diesel	1560	Business
Citroën	C4 Picasso	Diesel	1560	Confort
Citroën	C4 Picasso	Essence	1598	Confort
Volkswagen	Jetta	Essence	1197	Trendline
Volkswagen	Jetta	Diesel	1598	Confrotline

L'ensemble (*modèle, carburant*) ne suffit plus et il faudrait considérer, par exemple, l'ensemble (*modèle, carburant, cylindrée, finition*) pour obtenir une clef.
Pour éviter d'utiliser des clefs à rallonge, on ajoute usuellement attribut d'identification (noté *Id*) dont la valeur est un entier auto-incrémenté. Cet entier constitue une clef à lui seul.

3.2 Clef primaire et clef étrangère

Une **clef primaire** pour une relation est un seul attribut qui permet d'identifier chaque élément de la relation de manière unique.
Soit $R(S)$ une relation de schéma S et $A \in S$ un attribut de S. On dit que A est une clef primaire pour R si et seulement si pour tous éléments $e_1, e_2 \in R$, on a :

$$e_1.A = e_2.A \Rightarrow e_1 = e_2$$

Exemple

voitures2

Id	marque	modèle	carburant	cylindrée
1	Citroën	C4 Picasso	Diesel	1997
2	Citroën	C4 Picasso	Diesel	1997
3	Citroën	C4 Picasso	Diesel	1560
4	Citroën	C4 Picasso	Diesel	1560
5	Citroën	C4 Picasso	Essence	1598
6	Volkswagen	Jetta	Essence	1197
7	Volkswagen	Jetta	Diesel	1598

Dans la relation ci-dessus, l'attribut *Id* est bien une clef primaire pour la relation.

L'intérêt d'une clef primaire est de faciliter les liens entre deux relations d'une même base de données.

Exemple

véhicules

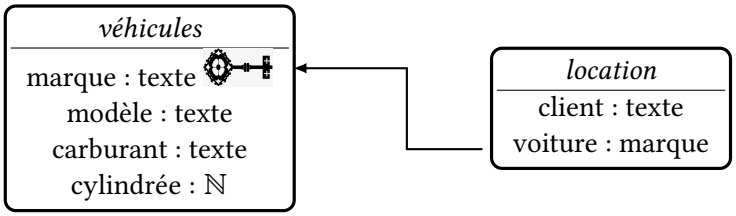
marque	modèle	carburant	cylindrée
Ariel	Atom	Essence	1998
Mercedes	C200	Diesel	2143
Subaru	WRX STI	Essence	2457
Bugatti	Chiron	Essence	7993

location

client	voiture
Alpha	Mercedes
Beta	Bugatti

L'attribut *marque* de la table *véhicules* constitue une clef primaire. Ainsi, en faisant simplement référence à la marque dans la table *location*, on peut avoir les caractéristiques détaillées de la voiture louée par chaque personne.

En revanche, dans l'exemple cité ci-dessus, rien n'empêche de créer un élément dans la table *location* qui emprunte une voiture qui n'existe pas. Une solution existe : c'est la **clef étrangère**.



Dans l'exemple, l'attribut *marque* constitue une clef primaire pour la relation *véhicules*. Si on impose que le domaine de l'attribut *voiture* de la table *location* soit constitué par les marques apparaissant dans la table *véhicules*, alors *marque* est une clef étrangère pour la table *location*. Il sera alors impossible d'insérer une voiture empruntée qui ne fasse pas partie de la liste des véhicules disponibles.

3.3 Associations

L'introduction de la clef étrangère a été l'occasion de présenter une première *association* entre deux tables : au lieu de mettre toutes les informations «Individu / Véhicule loué» dans une seule table, on sépare les informations dans deux tables distinctes.
Considérons une nouvelle table un peu plus remplie :

cinéma

titre	annee	nom_realisateur	prenom_realisateur
Alien	1979	Scott	Ridley
Vertigo	1958	Hitchcock	Alfred
Psychose	1960	Hitchcock	Alfred
Pulp Fiction	1995	Tarantino	Quentin
Titanic	1997	Cameron	James
Inglourious Basterds	2009	Tarantino	Quentin
Reservoir Dogs	1992	Tarantino	Quentin
Avatar	2009	Cameron	James

Comme on peut le voir, certaines informations (nom et prénom du réalisateur) sont répétées plusieurs fois pour les réalisateurs prolifiques. Pour alléger la base de donnée, on peut séparer cette table en deux :

films (clef étrangère : id_rea)

titre	annee	id_rea
Alien	1979	1
Vertigo	1958	2
Psychose	1960	2
Pulp Fiction	1995	5
Titanic	1997	6
Inglourious Basterds	2009	5
Reservoir Dogs	1992	5
Avatar	2009	6

réalisateurs (clef primaire : id)

id	nom	prenom
1	Scott	Ridley
2	Hitchcock	Alfred
5	Tarantino	Quentin
6	Cameron	James

L'attribut `id` constitue une **clef primaire** de la table *réalisateurs*. Au contraire, l'attribut `id_rea` n'est pas une clef primaire de la table *films*. Mais comme cet attribut fait référence à la clef primaire d'une autre table, `id_rea` est une **clef étrangère** (référençant l'attribut `id` de la table *films*). Dans ce schéma relationnel, un réalisateur peut être associé à plusieurs films : on dit qu'on a une **association 1-***.
Plus compliqué maintenant : certains films ont deux réalisateurs. On doit donc faire correspondre plusieurs films à plusieurs réalisateurs : c'est une **association** *-*.
Une première solution à ce problème serait d'ajouter une seconde colonne `id_rea2` pour mettre éventuellement l'identifiant du second réalisateur… Et le jour où on doit ajouter un film ayant trois réalisateurs, il faut tout recommencer.
Seconde solution, dupliquer les films ayant plusieurs réalisateurs comme sur le schéma ci-dessous. Cette solution oblige à multiplier les entrées pour certains films. On a séparé la table *cinéma* en deux pour éviter les répétitions des réalisateurs, ce n'est pas pour se mettre à répéter les films…

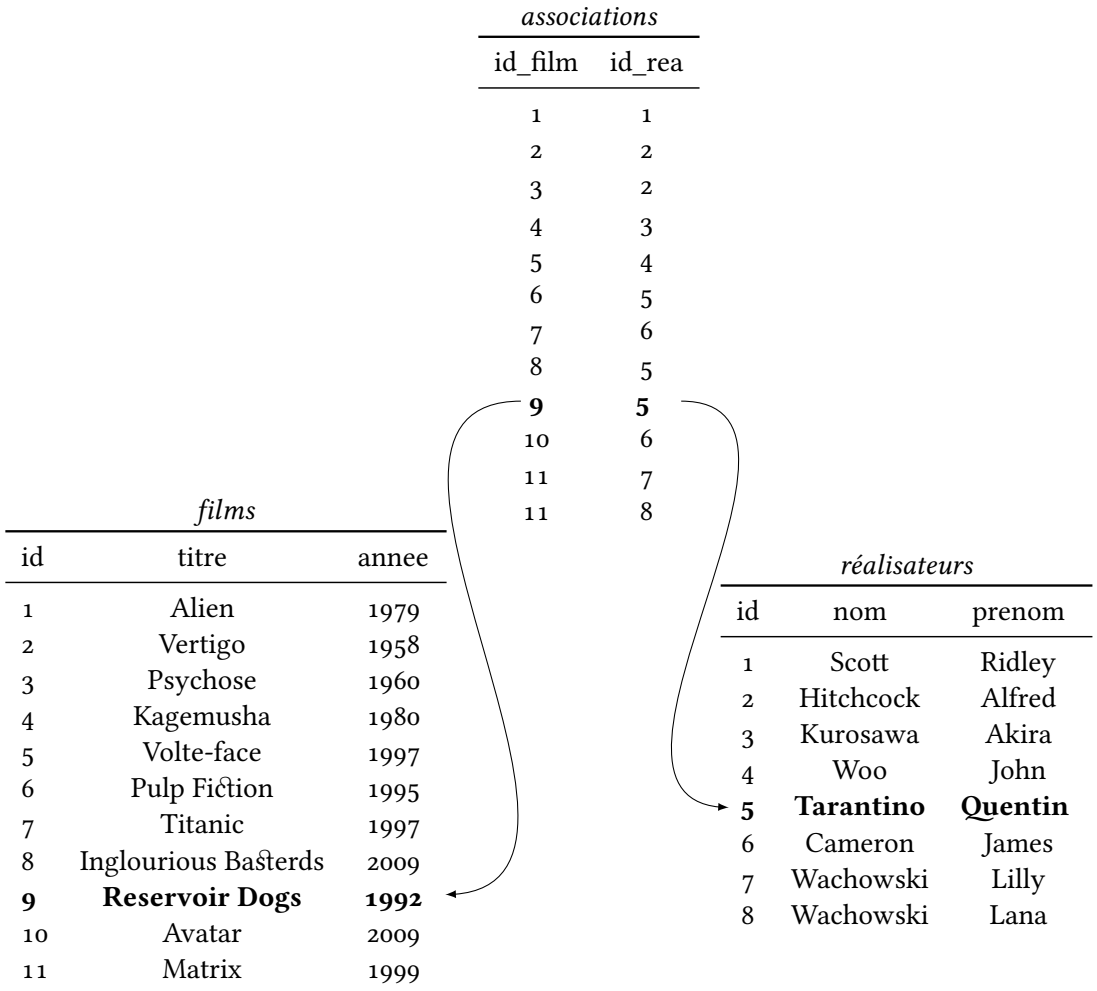
films

titre	annee	id_rea
Alien	1979	1
Vertigo	1958	2
Psychose	1960	2
Pulp Fiction	1995	5
Titanic	1997	6
Inglourious Basterds	2009	5
Reservoir Dogs	**1992**	**5**
Avatar	2009	6
Matrix	1999	7
Matrix	1999	8

réalisateurs

id	nom	prenom
1	Scott	Ridley
2	Hitchcock	Alfred
5	**Tarantino**	**Quentin**
6	Cameron	James
7	Wachowski	Lilly
8	Wachowski	Lana

La solution est de couper cette association *-* en deux associations 1-* à l'aide d'une table intermédiaire comme illustré sur la figure ci-dessous.



associations

id_film	id_rea
1	1
2	2
3	2
4	3
5	4
6	5
7	6
8	5
9	**5**
10	6
11	7
11	8

films

id	titre	annee
1	Alien	1979
2	Vertigo	1958
3	Psychose	1960
4	Kagemusha	1980
5	Volte-face	1997
6	Pulp Fiction	1995
7	Titanic	1997
8	Inglourious Basterds	2009
9	**Reservoir Dogs**	**1992**
10	Avatar	2009
11	Matrix	1999

réalisateurs

id	nom	prenom
1	Scott	Ridley
2	Hitchcock	Alfred
3	Kurosawa	Akira
4	Woo	John
5	**Tarantino**	**Quentin**
6	Cameron	James
7	Wachowski	Lilly
8	Wachowski	Lana

Dans cette situation, les attributs `id` des tables *films* et *réalisateurs* sont deux clefs primaires, et la table d'association contient les deux clefs étrangères référençant les éléments qu'elle relie. Un seul film peut être lié à plusieurs éléments de la table intermédiaire : c'est bien une association 1-*. De même, un seul réalisateur peut être relié à plusieurs éléments de la table intermédiaire.

4 Opérations avancées

Les opérations définies jusqu'à présent n'étaient applicables qu'à une seule relation ou à deux relations de même schéma relationnel. Les opérateurs suivants permettent de croiser les informations présentes dans plusieurs tables de formats différents.

4.1 Produit cartésien

4.1.1 Formalisme relationnel

Le produit cartésien $R \times R'$ de deux relations R et R', de schémas quelconques, consiste à associer tous les éléments de R avec tous les éléments de R'.

élèves

nom_élève
Alpha
Beta
Gamma

profs

nom_prof
Aleph
Beth

élèves×profs

nom_élève	nom_prof
Alpha	Aleph
Alpha	Beth
Beta	Aleph
Beta	Beth
Gamma	Aleph
Gamma	Beth

Remarque importante

La division cartésienne existe en algèbre relationnelle existe pour des raisons de complétude mais elle est absente des langages de requête.

4.1.2 Commande SQL

Le produit cartésien est obtenu très simplement en langage SQL en utilisant une projection de deux relations :

4.2 Jointure

4.2.1 Formalisme relationnel

La jointure sert à recoller deux relations différentes ayant au moins un attribut en commun.
Soient $R(S)$ et $R'(S')$ deux relations de schémas relationnels disjoints.
Soient $A \in S$ et $A' \in S'$ tels que $\text{dom}(A) = \text{dom}(A')$.
L'opération :

$$R[A = A']R' = \{e \in R \times R' | e.A = e.A'\}$$

est appelée **jointure symétrique** de R et R'. Elle peut également être notée $R \bowtie R'$, mais l'attribut servant à faire la jointure n'est pas précisé avec cette notation.

Exemple

Considérons les deux relations suivantes :

voitures

marque	modèle	carburant	cyl.
Mercedes	C200	Essence	1796
Subaru	Impreza	Diesel	1998
Bugatti	Chiron	Essence	7993
Fiat	500	Diesel	1248

constructeurs

constructeur	pays	groupe
Citroën	France	PSA
Porsche	Allemagne	Porsche
Mercedes	Allemagne	Daimler
Bugatti	France	Volkswagen AG
Subaru	Japon	Subaru

L'attribut `marque` de la table *voitures* correspond à l'attribut `constructeur` de la table *constructeurs*. On va alors réaliser la jointure symétrique selon (`marque = constructeur`) qui donnera une nouvelle relation.

voitures [marque = constructeur] *constructeurs*

marque	modèle	carburant	cylindrée	constructeur	pays	groupe
Mercedes	C200	Essence	1796	Mercedes	Allemagne	Daimler
Subaru	Impreza	Essence	1994	Subaru	Japon	Subaru
Bugatti	Chiron	Essence	7993	Bugatti	France	Volkswagen AG

Remarque importante

Les attributs servant à la jointure sont dupliqués. Une projection bien placée saura résoudre ce problème

Point d'attention

La jointure étudiée est ici une jointure symétrique, aussi appelée **jointure interne** en langage SQL. Elle exige qu'il y ait des données de part et d'autre de la jointure, c'est-à-dire que l'élément e sur lequel on fait la jointure doit exister dans les deux relations R et R'.
Si un élément de R n'a pas de correspondance dans R' (et réciproquement), cet élément n'apparaîtra pas dans la jointure.

4.2.2 Commande SQL

La jointure est obtenue avec le mot clef `JOIN ... ON`. La jointure de l'exemple ci-dessus s'écrit :

Remarque importante

Lorsqu'on les attributs servant à joindre deux tables ont le même nom, on peut utiliser le mot clef `USING` au lieu de `ON`.

Si les deux relations *constructeurs* et *voitures* étaient sous cette forme :

voitures

marque	modèle	carburant	cyl.
Mercedes	C63 AMG	Essence	6208
Subaru	WRX STI	Essence	2457
Bugatti	Chiron	Essence	7993

constructeurs

marque	pays	groupe
Mercedes	Allemagne	Daimler
Bugatti	France	Volkswagen AG
Subaru	Japon	Subaru

On pourrait réaliser la jointure sur la marque avec : `SELECT * FROM voitures JOIN constructeurs USING marque;`

4.2.3 Autojointure

Même si l'intérêt n'apparaît pas de prime abord, il peut être intéressant de réaliser la jointure d'un table avec elle-même.
La syntaxe est la même que précédemment à l'exception de la nécessité de renommer les copies de la table pour différencier les attributs.

Remarque importante

La syntaxe d'une autojointure est la suivante : `SELECT * FROM table [AS] table1 JOIN table [AS] table2 ON condition`
le mot-clef `[AS]` *est indiqué entre crochet car il est ici facultatif.*

Application

ITC1.8 : Autojointures

On se donne la base de données `windsor` suivante.

windsor

id	nom	mere	pere	conjoint
1	Elisabeth II			2
2	Philip			1
3	Charles III	1	2	5
4	Diana			
5	Camilla			3
6	William	4	3	7
7	Kate			6
8	George	7	6	
9	Charlotte	7	6	
10	Henry	4	3	
11	Anne	1	2	
12	Andrew	1	2	
13	Edward	1	2	
14	Meghan			10
15	Archie	14	10	
16	Louis	7	6	
17	Lilibet	14	10	

Écrire des requêtes SQL qui permettent d'afficher :

Q.1. le nom de chaque membre de la famille royale ainsi que celui de sa mère

Q.2. les enfants de Diana

Q.3. les parents de William

Q.4. les frères et sœur de Charles

4.3 Agrégation

Les opérations étudiées jusqu'à présent sont des fonctions scalaires : elles s'appliquent à chaque ligne indépendamment. Les fonctions d'agrégation regroupent les lignes pour effectuer une opération sur ce regroupement.

4.3.1 Formalisme relationnel

Pour toutes les explications qui suivent, on va travailler avec la relation suivante :

voitures3

marque	modèle	carburant	cylindrée
Mercedes	C63 AMG	Essence	6208
Mercedes	C200	Essence	1796
Mercedes	C200	Diesel	2143
Subaru	WRX STI	Essence	2457
Subaru	Impreza	Essence	1994
Subaru	Impreza	Diesel	1998
Bugatti	Chiron	Essence	7993
Renault	Twingo	Essence	1149
Abarth	500	Essence	1368
Fiat	500	Essence	1242
Fiat	500	Essence	1248

Lors de la réalisation d'une agrégation, on doit distinguer deux opérations distinctes :

L'application de la fonction d'agrégation Les fonctions d'agrégation classiques sont :

- comptage(A) : compte le nombre d'éléments d'attribut A,
- max(A) (resp. min(A)) : renvoi le plus grand (resp. le plus petit) élément de A,
- somme(A) : donne la somme des éléments de A,
- moyenne(A) : donne la valeur moyenne des éléments de A.

L'agrégation de la relation R en appliquant la fonction f à l'attribut A se note de la manière suivante :

$$\gamma_{f(A)}(R)$$

Il est possible de créer, par agrégation, une relation qui résulte de l'application de plusieurs fonctions d'agrégation. Soient $B_1, \dots, B_n \in S$ et $f_1, \dots, f_n$ des fonctions d'agrégation. La relation obtenue par application des fonctions $f_1, \dots, f_n$ aux attributs $B_1, \dots, B_n \in S$ est notée :

$$\gamma_{f_1(B_1),\dots,f_n(B_n)}(R)$$

Exemple

Si on cherche la plus grande cylindrée, ou les cylindrées extrémales dans la relation *voitures*, on peut écrire :

$\gamma_{\max(\text{cylindrée})}(voitures)$

max(*cylindrée*)
7993

$\gamma_{\max(\text{cylindrée}),\min(\text{cylindrée})}(voitures)$

max(*cylindrée*)	min(*cylindrée*)
7993	1149

Le regroupement des valeurs Au lieu d'appliquer la fonction d'agrégation à la totalité de la table, il est possible de regrouper les éléments selon un ou plusieurs attributs pour ensuite appliquer la fonction à chacun des regroupements.
Commençons par un exemple pour voir ce que peut donner un tel regroupement.
Soient $A_1, \dots, A_m, B_1, \dots, B_n \in S$ et $f_1, \dots, f_n$ des fonctions d'agrégation. L'application des fonctions $f_1, \dots, f_n$ aux attributs $B_1, \dots, B_n \in S$ regroupés par rapport aux attributs $A_1, \dots, A_m$ est notée :

$$A_1,\dots,A_m \gamma_{f_1(B_1),\dots,f_n(B_n)}(R)$$

Exemple

Si on cherche les cylindrées moyennes pour chaque marque, il faut donc appliquer la calcul de la moyenne aux éléments ayant comme attribut commun la *marque* :

$_{\text{marque}}\gamma_{\text{moyenne(cylindrée)}}(\textit{voitures})$

marque	moyenne(*cylindrée*)
Mercedes	3382,33
Subaru	2149,67
Bugatti	7793
Renault	1149
Abarth	1368
Fiat	1245

Remarque importante

Il est tout à fait possible de composer l'agrégation avec une sélection :

- sélection en amont : $A_1,\dots,A_m \gamma_{f_1(B_1),\dots,f_n(B_n)}(R) \circ \sigma_P$ La sélection σ_P est effectuée **avant** l'agrégation. Elle porte donc sur la relation R et la condition P porte sur les attributs de R.
- sélection en aval : $\sigma_P \circ A_1,\dots,A_m \gamma_{f_1(B_1),\dots,f_n(B_n)}(R)$ La sélection σ_P est effectuée **après** l'agrégation. Elle porte donc sur la relation γ et la condition P porte sur les attributs de γ.

4.3.2 Commandes SQL

La traduction de $\gamma_{f(A)}(R)$ en langage SQL se fait avec :

où les fonctions `f` sont :

fonction d'agrégation	langage SQL
comptage	`COUNT`
maximum	`MAX`
minimum	`MIN`
moyenne	`AVG`
somme	`SUM`

L'expression de ${}_{A_1,\dots,A_m}\gamma_{f_1(B_1),\dots,f_n(B_n)}(R)$ est légèrement plus complexe : il faut recourir au mot clef `GROUP BY` pour réaliser le regroupement.

Application

ITC1.9 : Fonctions d'agrégation

Écrire les requêtes SQL correspondant aux exemples donnés dans la partie précédente (cylindrée maximale, cylindrées extrémales et cylindrée moyenne).

Remarque importante

- La sélection en amont se fait comme une sélection classique, c'est-à-dire avec le mot clef `WHERE`.

  $${}_{A_1,\dots,A_m}\gamma_{f_1(B_1),\dots,f_n(B_n)}(R) \circ \sigma_{P_1}$$

  s'obtiendrait avec :

- La sélection en aval se fait avec le mot clef `HAVING`.

  $$\sigma_{P_2} \circ {}_{A_1,\dots,A_m}\gamma_{f_1(B_1),\dots,f_n(B_n)}(R)$$

  s'obtiendrait avec :

Point d'attention

Attention à l'ordre des mots clefs. `WHERE` porte sur la relation R. Il doit donc être juste après l'instruction `FROM R` et avant l'instruction de regroupement.
Au contraire, `HAVING` porte sur la relation agrégée. Il doit donc être placé à la fin de l'instruction.

Application

ITC1.10 : Sélections en aval et en amont

Q.1. Écrire une requête SQL qui permet de connaître le nombre de voitures par marque dont la cylindrée est supérieure à 2,0 L.

Q.2. Écrire une requête qui retourne les valeurs moyennes de cylindrées par marque lorsqu'elles sont supérieures à 2,5 L.

ITC 2

Révisions sur les graphes

Sommaire

1 Notions théoriques sur les graphes 23
1.1 Vocabulaire général 23
1.2 Listes et matrices d'adjacence 25
2 Parcours d'un graphe 27
2.1 Structures de données adaptées 28
2.2 Parcours en profondeur 29
2.3 Parcours en largeur 30
3 Recherche d'un plus court chemin 32
3.1 Présentation du problème 32
3.2 Cas d'un graphe non pondéré 33
3.3 Cas d'un graphe pondéré : algorithme de Dijkstra 34

1 Notions théoriques sur les graphes

1.1 Vocabulaire général

Graphe non-orienté

Définition

Un graphe non-orienté $G(S,A)$ est la donnée d'un ensemble fini S et d'un ensemble A de parties de S à 2 éléments. Les éléments de S sont appelés **sommets** (ou nœuds) du graphe. Les éléments de A sont appelés **arêtes** du graphe.

Graphe orienté

Définition

Un graphe orienté $G(S,A)$ est la donnée d'un ensemble fini S et d'une partie A de $S\times S$. Les éléments de S sont appelés **sommets** (ou nœuds) du graphe. Les éléments de A sont appelés **arcs** du graphe.

Degré

Définition

Si $G(S,A)$ est un graphe orienté, pour tout arc $(s,s')\in A$, on dit que s est un **prédécesseur** de s' et que s' est un **successeur** de s. Pour tout sommet $s\in S$ on note :
- $d_+(s)$ le nombre de successeurs de s : c'est le **degré sortant** de s ;
- $d_-(s)$ le nombre de successeurs de s : c'est le **degré entrant** de s.

Si $G(S,A)$ est un graphe non orienté, pour tout sommet $s\in S$, on dit que s' est un **voisin** de s si $\{s,s'\}\in A$. On note $d(s)$ le nombre de voisins de s : c'est le **degré** de s.

Chemin et chaîne

Définition

Un arc qui part et arrive au même sommet est une **boucle**.
Un **chemin** (respectivement une **chaîne**) est une suite finie $(s_0,\dots,s_k)$ d'éléments de S telle que pour tout $i\in[\![0,k-1]\!]$, $(s_i,s_{i+1})\in A$ (resp. $\{s_i,s_{i+1}\}\in A$).
La **longueur** du chemin est le nombre k d'arcs utilisés pour relier s_0 à s_k.
Le chemin (resp. une chaine) est dit **simple** s'il ne passe pas deux fois par un même arc (resp. arête).
Un **circuit** (resp. **cycle**) est un chemin (resp. chaîne) $(s_0,\dots,s_k)$ tel que $s_0=s_k$. On le qualifie d'élémentaire s'il est simple.

Connexité

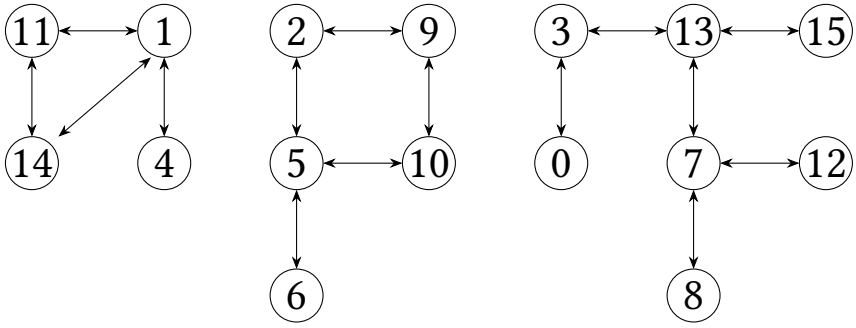
Définition

Une **composante connexe** d'un graphe non orienté (S,A) est un sous-ensemble S' de S tel que pour tous $s,t\in S'$ il existe une chaîne reliant s à t dans le graphe en ne passant que par des sommets de S'.
Une **composante fortement connexe** d'un graphe orienté (S,A) est un sous-ensemble S' de S tel que pour tout couple $(s,t)\in S'\times S'$ il existe un chemin de s à t dans le graphe en ne passant que par des sommets de S'.
Un graphe non orienté est dit **connexe** si l'ensemble de ses sommets forme une composante connexe.
Un graphe orienté est dit **fortement connexe** si l'ensemble de ses sommets forme une composante fortement connexe.

Application

ITC2.1 : Graphes non orientés

Considérons le graphe ci-dessous.



Q.1. Donner une chaîne de longueur 3 reliant le sommet 3 au sommet 12.

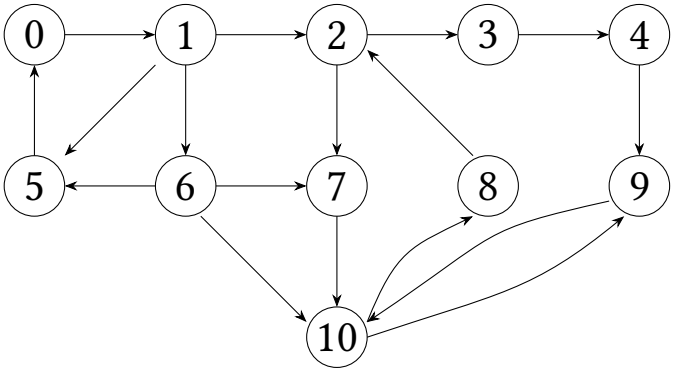
Q.2. Donner les degrés maximal et minimal de sommets que l'on peut trouver dans ce graphe.

Q.3. S'il en contient, donner les composantes connexes de ce graphe.

Application

ITC2.2 : Graphes orientés

Considérons le graphe ci-dessous.



Q.1. S'il en contient, donner les composantes fortement connexes de ce graphe.

Répondre par vrai ou faux aux questions suivantes :

Q.2. Si a et b sont dans une même composante fortement connexe de G, alors il existe un chemin de b vers a et un chemin de a vers b.

Q.3. Si a et b sont dans une même composante fortement connexe alors il existe un circuit qui passe par a et b.

Q.4. Si a et b sont dans une même composante fortement connexe alors il existe un circuit élémentaire qui passe par a et b.

Q.5. S'il existe un circuit qui passe par a et b alors, a et b sont dans une mème composante fortement connexe.

1.2 Listes et matrices d'adjacence

Liste d'adjacence

Définition

Soit $G = (S, A)$ un graphe éventuellement orienté et pondéré. Une représentation par **listes d'adjacence** consiste à associer à chaque sommet du graphe la liste de ses successeurs (ou voisins).

Matrice d'adjacence

Définition

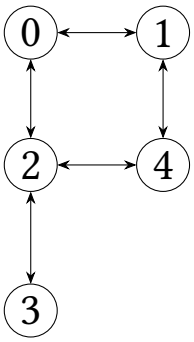
La **matrice d'adjacence** associée au graphe non orienté $G = (S, A)$ est la matrice (symétrique) $M_G \in \mathcal{M}_n(\mathbb{R})$ définie par $\begin{cases} m_{ij} = m_{ji} & = 1 \text{ si } \{i,j\} \in A \\ m_{ij} & = 0 \text{ sinon.} \end{cases}$.

La définition s'étend naturellement aux matrices orientées.

Application

ITC2.3 : Listes et matrices d'adjacence

On considère le graphe non orienté suivant :



Q.1. Donner une représentation par listes d'adjacence de ce graphe.

Q.2. Donner une représentation par matrice d'adjacence de ce graphe.

On considère le graphe orienté défini par les commandes suivantes :

```
S = [k for k in range(0,5)]
A = [(0,1), [1,4] ,[0,2], [4,2], [2,3]]
```

Q.3. Donner une représentation par liste d'adjacence de ce graphe.

Q.4. Donner une représentation par matrice d'adjacence de ce graphe.

Q.5. Représenter graphiquement cet objet.

Nombre de chemins entre i et j

Soit M^n la puissance usuelle n-ième (avec $n > 1$) de la matrice d'adjacence M d'un graphe (S, A). Alors le coefficient $m_{i,j}^{(n)}$ de M^n contient le nombre de chemins distincts de longueur n du i-ième sommet du graphe au j-ième sommet du graphe.

ITC2t1

Application

ITC2.4 : Décompte de chemins

On reprend le graphe de l'application 3.

Q.1. Déterminer le nombre de chemins de longueur 1,2 et 3 permettent de relier 2 à 0.

Q.2. En calculant les puissances n-ième de la matrice d'adjacence, vérifier le résultat précédent (on pourra se servir de sa calculatrice ou de Python).

2 Parcours d'un graphe

Différents parcours d'un graphe

Définition

Un **parcours en profondeur** consiste à partir d'un sommet et à explorer d'abord un chemin en ne visitant que des sommets non encore visités, jusqu'à être bloqué et à revenir au sommet précédent pour tenter de découvrir d'autres sommets non encore visités, explorant ainsi un nouveau chemin.
Un **parcours en largeur** consiste à partir d'un sommet et à explorer d'abord tous ses successeurs, avant de visiter tous les successeurs (non encore visités) des successeurs visités lors de la première étape, et ainsi de suite.

Application

ITC2.5 : Parcours de graphe

On considère la situation suivante où vous souhaitez acheter des mangues en utilisant un réseau de connaissances (vos amis, les amis de vos amis, etc.). Suivant les jours, seuls certains d'entre eux sont en mesure de vous vendre le produit désiré.

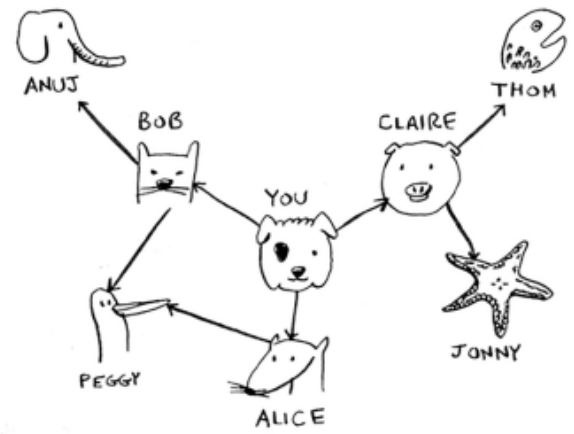


Figure 2.1 – Représentation du réseau relationnel

On considère les 3 situations suivantes :

1. Alice est la seule vendeuse de mangues ;
2. Peggy est la seule vendeuse de mangues ;
3. Anuj et Claire sont les deux seuls vendeurs de manques.

Q.1. Donner, dans le cas d'un parcours en profondeur, le nom du vendeur sélectionné dans chacun des cas ainsi que le nombre de tests effectués. On considère que les parcours sont faits par ordre lexicographique dans un niveau donné.

Q.2. Donner, dans le cas d'un parcours en largeur, le nom du vendeur sélectionné dans chacun des cas ainsi que le nombre de tests effectués.. On considère que les parcours sont faits par ordre lexicographique dans un niveau donné.

2.1 Structures de données adaptées

Pile et file

Définition

Une **pile** est une structure de données dans laquelle des éléments figurent dans un ordre précis, de sorte que seul le dernier élément ajouté, nommé le **sommet** de la pile, est accessible en le retirant de la pile.
Une **file** est, à l'instar d'une pile, une structure de données dans laquelle des éléments figurent également dans un ordre précis, mais cette fois-ci seul le premier élément ajouté, nommé la **tête** de la file, est accessible en le retirant de la file. Le dernier élément mis en place est nommé la **queue** de la file.

En Python, on peut réaliser une structure de pile en utilisant des listes qu'on s'interdit de manipuler autrement que par l'initialisation à `[]`. L'ajout se fait en écrivant : `l=[]+l` et nécessite une recopie en $O(n)$ tandis que la lecture est à coût constant.
Pour les files, l'ajout en fin de file se fait en $O(1)$ à l'aide de la méthode `append` ou `l=l+[]`. En revanche, nous verrons que l'accès en lecture au dernier élément a un coût linéaire.

Un objet (au programme) présent dans la bibliothèque `collections` est bien plus intéressant et permet une manipulation aisée des files et pile : les **`deque`** (qu'on traduit en français par dèque ou file à double entrée) dont le nom vient de *double-ended queue*, c'est-à-dire une file où les opérations d'ajout et de retrait sont disponibles (et optimisées) à la fois en tête et en queue. On utilisera les commandes suivantes :

Code Python

Mode éditeur

```
help(deque.append)
help(deque.appendleft)
help(deque.pop)
help(deque.popleft)
```

```
Help on method_descriptor:

append(...)
    Add an element to the right side of the deque.

Help on method_descriptor:

appendleft(...)
    Add an element to the left side of the deque.

Help on method_descriptor:

pop(...)
    Remove and return the rightmost element.

Help on method_descriptor:

popleft(...)
    Remove and return the leftmost element.
```

2.2 Parcours en profondeur

On souhaite maintenant écrire un code Python permettant d'afficher tous les sommets d'un graphe représenté par une matrice d'adjacence M (codée par une liste de listes) en partant d'un sommet arbitraire [a].
En premier lieu, il faut réfléchir à la structure de données appropriée :

ITC2t2

Complétons le code Python suivant :

Code Python

Mode éditeur

```
def parcours_profondeur(M,s0):
    """
    M: matrice d'adjacence sous forme d'une liste de listes
    s: sommet de départ
    sortie: aucune
    affiche tous les sommets de M, en partant du sommet i
    """
    n=len(M)
```

a. on considère que les n sommets du graphe sont les entiers $0, \dots, n-1$; l'utilisation d'un dictionnaire permet de facilement se rapporter à ce cas.

2.3 Parcours en largeur

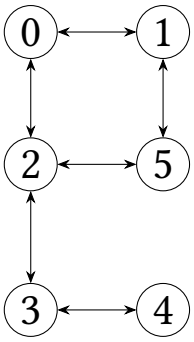
Le code à utiliser est ici très proche du précédent :

Code Python

Mode éditeur

```
def parcours_largeur(M,s0):
    """
    M: matrice d'adjacence sous forme d'une liste de listes
    s: sommet de départ
    sortie: aucune
    affiche tous les sommets de M, en partant du sommet i
    """
    n=len(M)
```

On peut valider les résultats des deux parcours sur l'exemple suivant :



Deux propriétés importantes découlent des parcours précédents.

Connexité d'un graphe non orienté .

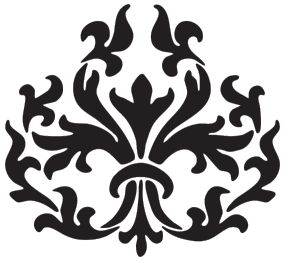
Remarque importante

Un graphe non orienté est connexe si, et seulement si, un algorithme de parcours depuis n'importe quel sommet renvoie un objet dont la taille est le nombre de sommets.

Présence d'un circuit dans un graphe orienté .

Remarque importante

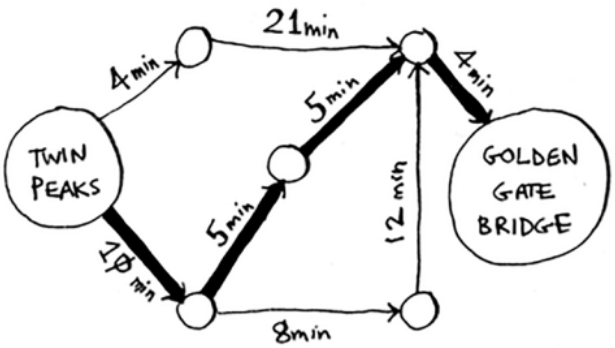
Un graphe orienté possède un circuit, si et seulement si, un algorithme de parcours depuis au moins un sommet renvoie un objet contenant ce sommet.



3 Recherche d'un plus court chemin

3.1 Présentation du problème

On considère le plan de transports en commun très simplifié suivant, indiquant de façon sommaire les durées entre les arrêts.



Deux situations peuvent se présenter :

- on souhaite minimiser le nombre de correspondance
- on souhaite minimiser le temps de parcours (on considérant que le temps de correspondance est pris en compte dans les durées présentées)

ITC2t3

3.2 Cas d'un graphe non pondéré

Minimiser le nombre d'étapes pour aller d'un point A à un point B est une légère variation du parcours en largeur mené précédemment : il suffit de tester pour chaque sommet rencontré si c'est celui que l'on recherche.
On donnera deux versions du code : la première se contente de tester si un chemin existe entre deux sommets passés en arguments, la seconde retourne le nombre d'étapes minimal entre ces deux sommets.

Code Python

Mode éditeur

```
def minimise_etapes(M,s_a,s_b):
    """
    M: matrice d'adjacence sous forme d'une liste de listes
    s_a: sommet de départ
    s_b: sommet d'arrivée
    sortie: bool. qui détermine si un chemin existe de A à B
    """
    n=len(M)

def minimise_etapes_compte(M,s_a,s_b):
    """
    M: matrice d'adjacence sous forme d'une liste de listes
    s_a: sommet de départ
    s_b: sommet d'arrivée
    sortie: nombre d'étapes pour aller de A à B
    """
    assert minimise_etapes(M,s_a,s_b)
    n=len(M)
```

3.3 Cas d'un graphe pondéré : algorithme de Dijkstra

3.3.1 Notion de graphe pondéré

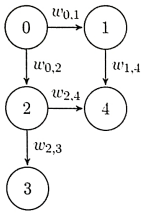
Avant de détailler l'algorithme proprement dit, il nous reste à introduire la notion de graphe pondéré.

Graphe pondéré

Définition

Un graphe $G(S,A)$ **pondéré** est un graphe dont les arcs (ou les arêtes) sont étiquetés par un nombre entier appelé **poids**. Cette donnée supplémentaire peut se représenter par une fonction de A vers $\mathbb{Z}$.
Le **poids d'un chemin** dans un graphe pondéré est la somme des poids des arcs qui forment ce chemin.

La représentation par matrice d'adjacence permet très facilement d'indiquer les poids de chaque arc :



$\leftrightarrow$

$$\begin{bmatrix} \star & w_{0,1} & w_{0,2} & \star & \star \\ \star & \star & \star & \star & w_{1,4} \\ \star & \star & \star & w_{2,3} & w_{2,4} \\ \star & \star & \star & \star & \star \\ \star & \star & \star & \star & \star \end{bmatrix}$$

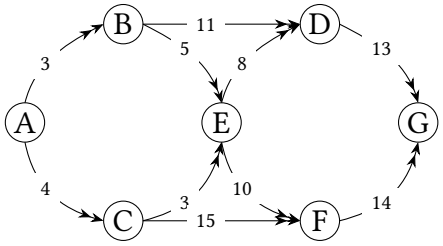
3.3.2 Principe de la méthode

L'algorithme de Dijkstra permet de trouver le chemin le plus court entre deux sommets d'un graphe non orienté pondéré lorsque **tous les poids sont positifs**. Il suit le principe d'un parcours en largeur, mais au lieu de prendre le sommet suivant dans la file, on traite d'abord celui dont la distance à l'origine est la plus petite. Ainsi, une fois qu'un sommet est traité, sa distance à l'origine ne peut plus diminuer (car les poids sont positifs). Dit autrement : si pour aller de A à B le chemin le plus court passe par I, alors la partie du chemin entre A et I est le plus court chemin entre A et I et la partie du chemin entre B et I est le plus court chemin entre B et I. On optimise donc le choix à chaque étape : c'est un algorithme glouton.

Application

ITC2.6 : Mise en œuvre à la main de l'algorithme de Dijkstra

On considère le graphe pondéré suivant (pour la lisibilité, les sommets sont nommés par des lettres, mais la conversion en entier est immédiate).



Étape	A	B	C	D	E	F	G
0	0	∞	∞	∞	∞	∞	∞
1							
2							
3							
4							
5							
6							
7							

Q. Remplir le tableau suivant correspondant aux six itérations de la méthode en considérant que le sommet d'origine est le A.

3.3.3 Code

Nous allons ici nous contenter d'une version naïve qui utilise des listes.
Dans la première version du code, on veut simplement retourner le poids d'un chemin reliant deux sommets.

Code Python

Mode éditeur

```
def dijkstra(M,s_a,s_b):
    """
    M: matrice d'adjacence pondérée (liste de listes)
    s_a: sommet de départ
    s_b: sommet d'arrivée
    sortie: coût minimal pour aller de s_a à s_b
    """
    assert minimise_etapes(M,s_a,s_b)
    n=len(M)
    poids_sommets={i:float('inf') for i in range(n)}
    dic_sommets_minimises={i:0 for i in range(n)}
```

Dans une seconde version, on veut retourner le chemin suivi pour aller de A à B. Pour cela il suffit de modifier légèrement le programme précédent, en construisant au fur et à mesure une liste des antécédants des points traités.

Code Python

Mode éditeur

```
def dijkstra_chemin(M,s_a,s_b):
    """
    M: matrice d'adjacence pondérée (liste de listes)
    s_a: sommet de départ
    s_b: sommet d'arrivée
    sortie: coût minimal pour aller de s_a à s_b,chemin suivi
    """
    assert minimise_etapes(M,s_a,s_b)
    n=len(M)
    poids_sommets={i:float('inf') for i in range(n)}
    dic_sommets_minimises={i:0 for i in range(n)}
    dic_antecedants={i:0 for i in range(n)}
```

Remarque : On peut améliorer le code en utilisant une variante des files : les **files de priorité**. Les défilements des éléments ne sont plus décidés par leur ordre d'enfilement, mais par un entier naturel associé : leur priorité.
Pour nous, cette priorité est la distance du sommet à l'origine : on veut traiter en priorité le sommet restant dont la distance à l'origine est la plus faible. On peut aussi mettre à jour la priorité des éléments au fur et à mesure de l'avancement.
La fonction `remove` du type `deque` permet ce type de manipulation facilement.

ITC 3

Algorithmes de tri

Sommaire

1 Quelques définitions 38

2 Petit point sur Python 39

2.1 Fonctions natives de tri 39

2.2 Conventions syntaxiques 39

3 Algorithmes de tri 40

3.1 Tri par sélection 40

3.2 Tri par insertion 41

3.3 Tri fusion (merge-sort) 42

3.4 Tri rapide (quicksort) 44

L'objectif de ce chapitre est de réviser les tris introduits dans le cours de première année, tout en réinvestissant la notion de complexité.
Nous allons voir dans un premier temps que `Python` propose des fonctions qui effectuent un tri directement, des sortes de « boîtes noires » dont on ne sait pas grand-chose. Ensuite, nous aborderons les principales techniques de tri que vous avez à connaître et nous discuterons de leur complexité en temps (puisqu'un algorithme sera d'autant plus efficace qu'il trie rapidement).
Le caractère aléatoire de l'organisation initiale de la liste à trier va nous conduire à discuter des complexités dans le meilleur et dans le pire des cas : il est possible, selon l'organisation initiale de la liste à trier, que la performance d'un algorithme censé être plus lent qu'un autre s'inverse.

1 Quelques définitions

Voici quelques notions qu'il faut savoir caractériser :

- clefs : c'est ce qui est utilisé pour trier des éléments. Par exemple :
  - on peut trier des mots à l'aide de leur première lettre. La clef est ainsi la première lettre
  - on peut trier des couples $(4,5)(1,2)(1,3)(2,3)(3,1)$ selon leur premier terme, ce sera donc la clef, ou selon leur deuxième terme , ou selon une combinaison des deux ...
- tri comparatif : tri fondé sur la comparaison entre les clefs des éléments pour les trier ;
- tri itératif (ex : tri insertion) : tri fondé sur un ou plusieurs parcours itératifs de la liste à trier ;
- tri récursif (ex : tri rapide, tri fusion) : tri fondé sur une méthode récursive ;
- tri en place : tri qui n'utilise une quantité constante d'éléments en mémoire ;
- tri avec listes auxiliaires : tri qui crée des listes auxiliaires pour réaliser le tri (quantité de mémoire utilisée non constante) ;
- tri stable : tri qui conserve l'ordre relatif des éléments de même clef.

Application

ITC3.1 : Caractérisation de tris

.

On se donne la liste suivante : `l=[(4, 5),(1, 2),(1, 3),(2, 3),(3, 1)]` Caractériser un algorithme de tri qui retourne :

Q.1. `[(1, 2),(1, 3),(2, 3),(3, 1),(4, 5)]`

Q.2. `[(1, 3),(1, 2),(2, 3),(3, 1),(4, 5)]`

2 Petit point sur Python

2.1 Fonctions natives de tri

Introduisons ici deux fonctions que bscPython propose nativement. et pour cela, regardons le résultat d'exécution des fonctions suivantes.

Code Python

Mode éditeur

```
n = 15
L = [rd.randint(-n,n) for i in range(n)]
L1= L.copy()
print(L)
L.sort()
L2 = sorted(L1)
print(L,L==L2)
```

```
[4, -4, -10, -10, 15, -2, -3, -3, -6, -15, 4, 3, -11, 6, -15]
[-15, -15, -11, -10, -10, -6, -4, -3, -3, -2, 3, 4, 4, 6, 15] True
```

Il est fort probable que ces deux fonctions soient interdites d'utilisation le jour du concours ...

2.2 Conventions syntaxiques

Les PEP (*Python Enhancement Proposal*) sont des documents qui décrivent de nouvelles fonctionnalités proposées pour Python et en documentent les aspects.

Les conventions syntaxiques sont regroupées dans la PEP 8. Elles précisent par exemple les normes d'indentation (4 espaces), les règles de nommage des variables composées (utilisation du _), les règles de saut de ligne (pas à l'intérieur d'une fonction), ...

Les PEP 484 et PEP 3107 ont introduit et précisé le mécanisme d'annotation de fonctions qui permettent de préciser le type des fonctions et de leurs valeurs de retour. Cet alourdissement de syntaxe possède un intérêt pour l'utilisation dans des projets lourds utilisant les potentialités de contrôle des I.D.E., pour la vérification automatique par des fonctions tierces, pour des utilisations inter-langages, ...
Voici quelques exemples élémentaires utiles :

Code Python

Mode éditeur

```
def greeting(name: str) -> str:
    return 'Hello ' + name

# Appel de fonction
print(greeting('MP'))

# Accès aux annotations utilisateur
print(greeting.__annotations__)
```

```
Hello MP
{'name': <class 'str'>, 'return': <class 'str'>}
```

3 Algorithmes de tri

Nous allons aborder quelques manières de trier que vous avez à connaître. Nous verrons ici les principes, et en TP vous les coderez directement. Appelons L la liste à trier et L la liste triée.

3.1 Tri par sélection

Le tri par sélection est sûrement le plus simple des tris. L'idée est de placer récursivement en tête de liste le minimum de sous-listes successivement triées. Le tri s'effectue ici très facilement en place.

Code .

ITC3t4

Caractéristiques Ce tri est :

ITC3t5

Complexité .

ITC3t6

3.2 Tri par insertion

Principe Le tri insertion consiste à prendre chaque valeur d'une liste à trier, de la comparer à la valeur précédente, et d'intervertir les deux si la valeur actuelle est plus faible que la valeur précédente. On procède alors ainsi de suite jusqu'à ce que la valeur déplacée initialement soit supérieure à la précédente. On commence à partir de la seconde valeur dans la liste. Il n'est pas nécessaire de créer une seconde liste, on trie directement la liste initiale.

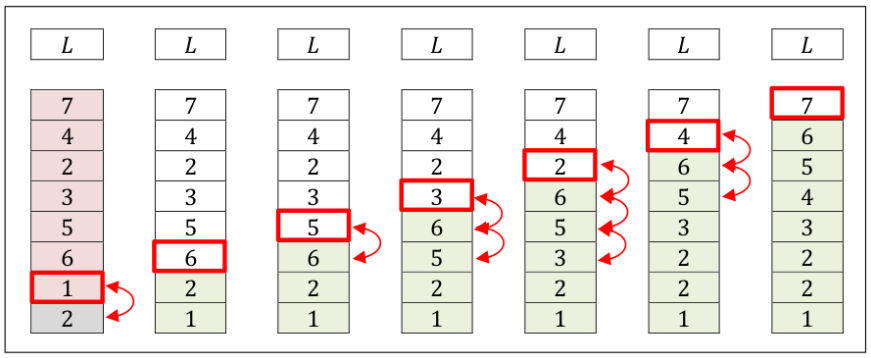


Figure 3.1 – Illustration d'un tri insertion

Code .

ITC3t7

Caractéristiques Ce tri est :

ITC3t8

Complexité La première boucle réalisé n itérations dans tous les cas.
Dans le pire des cas, la seconde boucle réalise j $<$ n itérations, soit une complexité quadratique.
Dans le meilleur des cas(liste triée) la deuxième boucle est de coût constant, donc la complexité est linéaire.

3.3 Tri fusion (merge-sort)

Cet algorithme emploie une stratégie dite de *diviser pour régner*. Il consiste à diviser récursivement une liste en deux puis à fusionner les sous listes obtenues en les triant.
Les étapes successives sont les suivantes. Soit une liste ℓ à trier :

- traiter le cas de base : si ℓ ne contient qu'un terme, elle est triée ;
- partager ℓ en 2 listes ℓ_1 et ℓ_2 de tailles identiques (à 1 près) ;
- appliquer récursivement la procédure aux listes ℓ_1 et ℓ_2 pour les trier ;
- en supposant que ℓ_1 et ℓ_2 ont été triées à l'étape précédente, les fusionner de manière ordonnée.

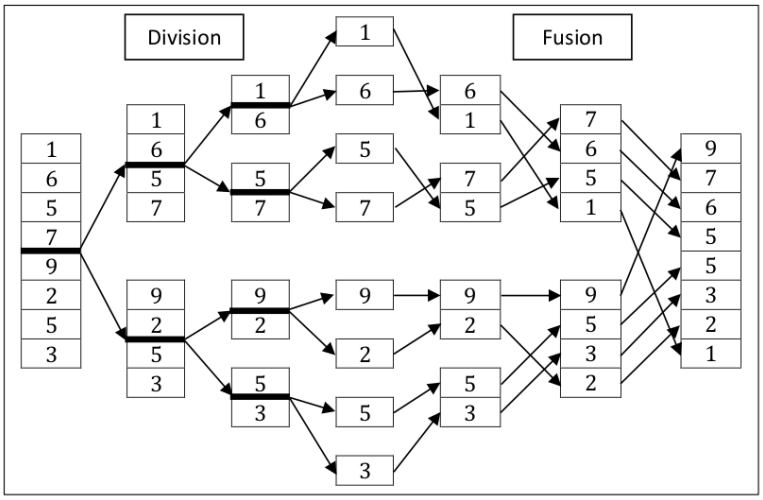


Figure 3.2 – Illustration d'un tri fusion

Code .

Remarque Le tri en place n'est pas présenté ici ; le décalage de tous les termes de la fusion n'étant pas du tout évident à mettre en place sans liste auxiliaire. On donne l'illustration ci-dessous d'une façon dont procéderait dans un tel tri.

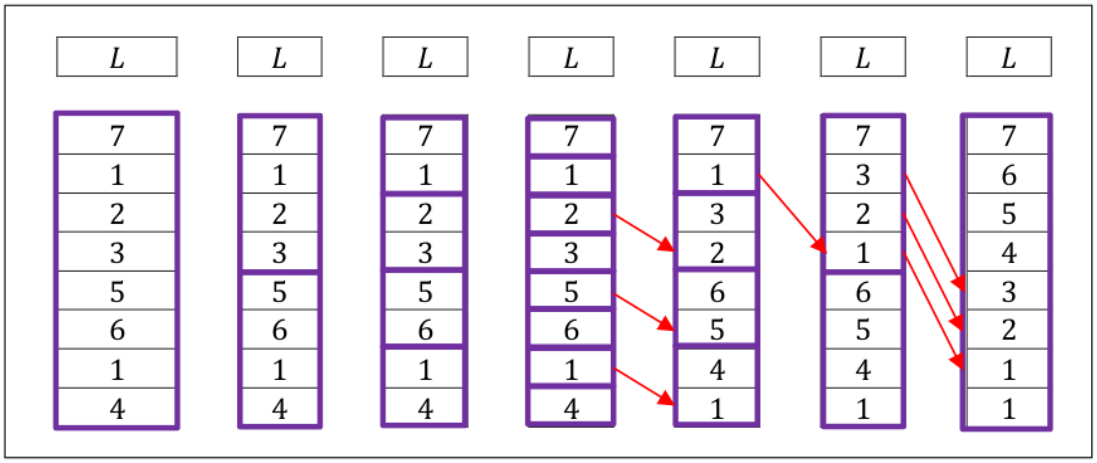


Figure 3.3 – Principe de tri fusion en place

Complexité Commençons par montrer que la complexité à l'étape n est en $O(n)$ puis trouvons la complexité globale.

ITC3t9

Il n'y a pas de distinction à faire entre meilleur et pire des cas puisque les déplacements sont tous effectués.

3.4 Tri rapide (quicksort)

Ce tri repose sur le choix d'un pivot (une des valeurs de la liste à trier) et le partage des autres valeurs en fonction de celui-ci. Naïvement, on considère que le pivot est la première valeur de la liste. On peut effectuer un autre choix, nous en parlerons lors de l'étude de la complexité de ce tri.

3.4.1 Tri rapide avec listes auxiliaires

Le tri rapide a été inventé par Hoare vers 1960. Le principe est de partager une liste en 2 listes telles que dans la première, toutes les valeurs prises soient inférieures à celles de la seconde. On retrouve ici une stratégie dite *diviser pour régner* où un problème est décomposé en deux problèmes plus simples. À la fin, on regroupe les résultats de chaque sous-problème pour arriver au résultat.
Il est alors possible d'appliquer récursivement cette démarche afin d'obtenir, à la fin, une liste triée :

- traiter le cas de base : si la liste est vide, la renvoyer.
- choisir un élément P de L appelé « pivot » (naïvement, première valeur).
- créer les listes L_1 et L_2 telles que $\begin{cases} \forall i \neq 0, L_1[i] \leqslant P \\ \forall i \neq 0, L_2[i] > P \end{cases}$
- appliquer récursivement le tri aux listes L_1 et L_2 pour obtenir deux listes L'_1 et L'_2 triées
- renvoyer les listes combinées dans l'ordre : L'_1, P, L'_2.

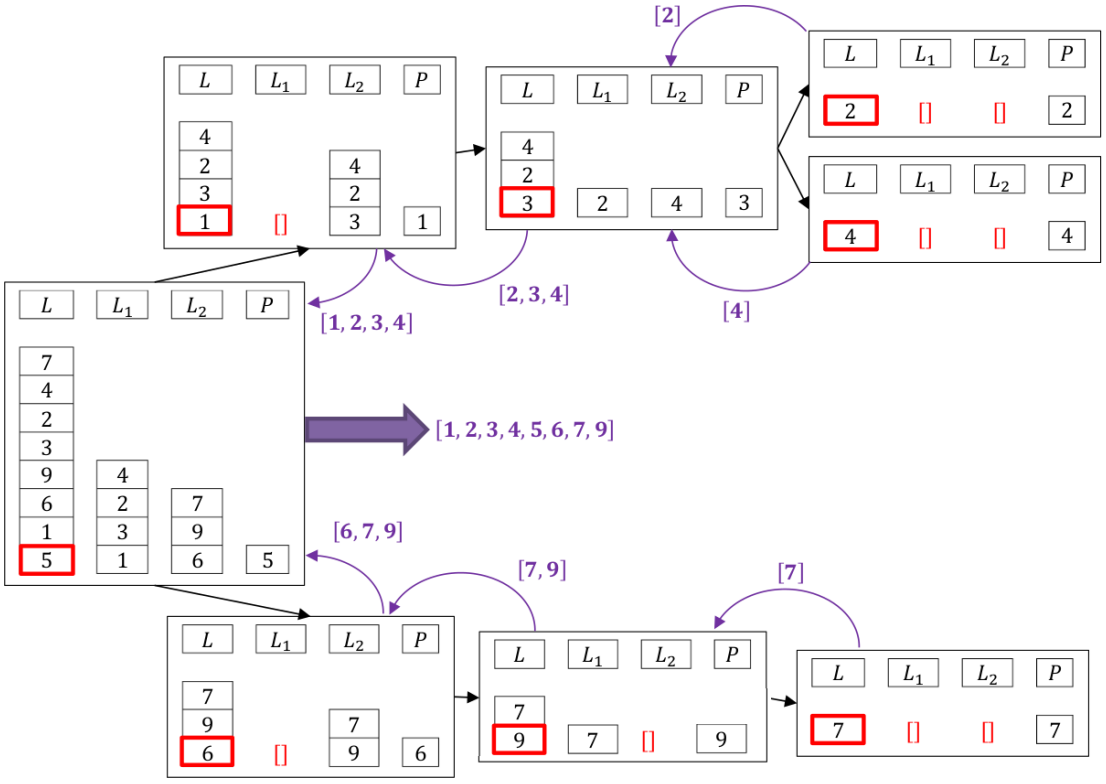


Figure 3.4 – Illustration d'un tri rapide avec listes auxiliaires

Code .

ITC3t10

Commentaire Ce tri n'est ni stable, ni en place. La taille de la pile de récursivité peut poser problème dans le cas de listes de grande taille. Une version en place est donc à préférer.

3.4.2 Tri en place

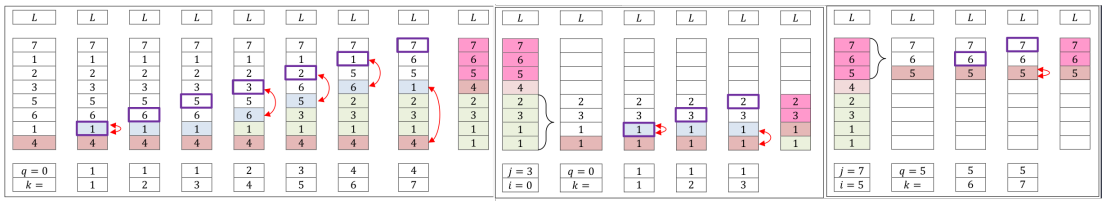
On peut réaliser un tri rapide en place, c'est-à-dire sans utiliser de mémoire additionnelle pour stocker des listes auxiliaires.
Le principe de réalisation de ce tri consiste à appliquer la démarche suivante par récurrence :

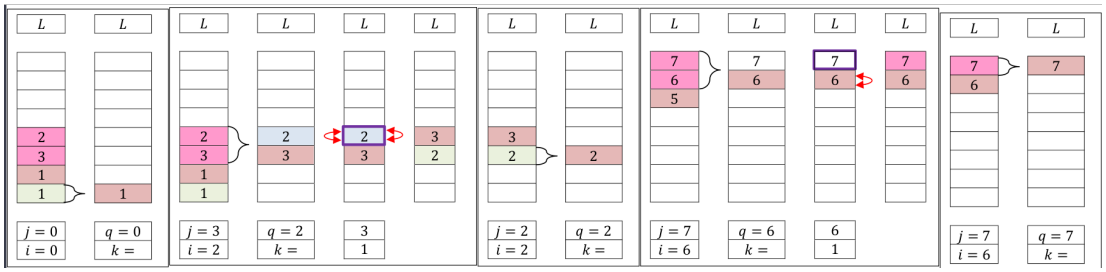
- considérer une portion de liste entre les indices `i` et `j` inclus. À la première itération, c'est la liste entière.
  Si cette sous liste contient un seul terme, ne rien exécuter, elle est déjà triée ;
- sinon :
  - ◇ choisir le pivot `P` : naïvement, le premier. Sinon, en choisir un et l'échanger avec la première valeur de la portion de liste traitée
  - ◇ appeler `p` l'indice du pivot : `p=i`
  - ◇ définir un indice `q` qui au départ vaut `p`
  - ◇ étudier chaque valeur de la sous-liste considérée pour un indice `k` variant entre les indices `i+1` et `j` inclus et procéder ainsi :
    - * si `L[k]>P`, ne rien faire
    - * sinon (`L[k] = P`) :
      - · `q=q+1`
      - · `L[q],L[k] = L[k],L[q]`
  - ◇ à la fin, échanger le pivot avec le terme à la position `q`: `L[p],L[q] = L[q],L[p]`
  - ◇ appeler récursivement cette procédure sur les « sous » listes de part et d'autre du pivot de la portion de liste étudiée

Après cette étape, on obtient un pivot définitivement placé, et deux sous listes avant et après le pivot telles que :

- toutes les valeurs situées avant le pivot lui sont inférieures ;
- toutes les valeurs situées après le pivot lui sont strictement supérieures ;

Il suffit alors de procéder de même sur les deux sous-listes et ainsi de suite par récursivité. Il n'y a alors aucun renvoi, chaque sous liste étant directement triée.





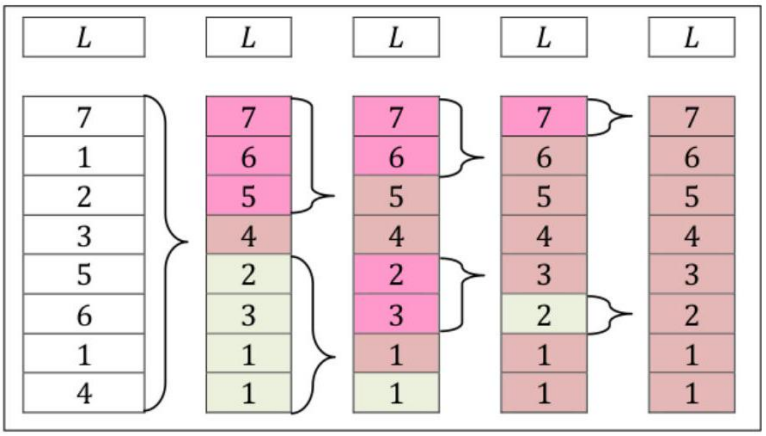


Figure 3.5 – Illustration résumée d'un tri rapide

Code .

ITC3t11

Commentaire De base, ce tri n'est pas stable. Prenons l'exemple d'un pivot en première position égal à l'un des autres termes de la liste ; il sera replacé après les termes avec lesquels il est égal.
Une adaptation mineure de ce code permet de trouver la médiane d'une liste : on ne traite à chaque étape que la sous-liste contenant l'indice milieu de la liste jusqu'à ce que le pivot se retrouve à cette position qui est la médiane de la liste.

Complexité Dans le meilleur des cas, à chaque division, on a autant de termes dans chacune des sous listes. Ainsi, à chaque étape, il s'appelle 2 fois à l'ordre $n/2$. Pour chaque exécution à l'ordre n, il parcourt la liste de ses n éléments. La complexité à l'ordre n est donc $O(n)$.
La complexité globale est donc comme précédemment quasi-linéaire : $C(n) = n \ln n$.

Dans le pire des cas, à chaque division, une sous-liste possède 0 termes, l'autre les $n-1$ restants.
Les n appels de la liste mènent à une complexité quadratique : $C(n) = O(n^2)$
Le pire des cas est atteint lorsque la liste est déjà presque triée.
Une pré-étude de la liste avant de la trier, permet de choisir un pivot en milieu de liste si l'on voit qu'elle déjà organisée.

Comparaison finale On peut comparer ces deux derniers algorithmes ainsi :

- le tri rapide effectue des travaux de tri autour du pivot puis appelle récursivement ce travail de tri : on parle de **récursivité sur les résultats** ;
- le tri fusion appelle récursivement une fonction qui divise le problème et trie à la fin, puis recombine les résultats : on parle de **récursivité sur les données**.

ITC 4

Dictionnaires et programmation dynamique

Sommaire

1 Tableaux et listes chaînées 50
- 1.1 Tableau 50
- 1.2 Liste chaînée 50
- 1.3 Le type `list` en Python 51

2 Piles et files 53
- 2.1 Rappels 53
- 2.2 Opérations sur les piles et files 53
- 2.3 Applications 54
- 2.4 Implémentations en Python 57

3 Dictionnaires et fonctions de hachage 58
- 3.1 Problématique 58
- 3.2 Exemple et définitions 58
- 3.3 Fonction de hachage et gestion des collisions 59
- 3.4 Les dictionnaires en pratique 62

4 Programmation dynamique 64
- 4.1 Un problème posé par la programmation récursive 64
- 4.2 Algorithme de Floyd-Warshall 67

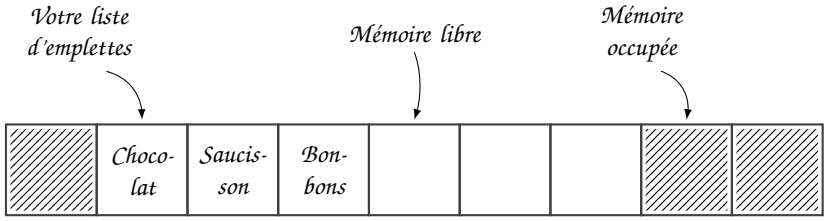
En informatique, une structure de données est une structure logique destinée à contenir des données, afin de leur donner une organisation permettant de simplifier leur traitement. Derrière cette définition triviale se cache une problématique fondamentale : comment ranger les données en mémoire pour en faciliter l'accès.

1 Tableaux et listes chaînées

Oublions pour l'instant les listes `Python` pour revenir à un cas plus général. Lorsque l'on souhaite stocker plusieurs informations en mémoire, les alternatives usuelles sont les tableaux et les listes chaînées.

1.1 Tableau

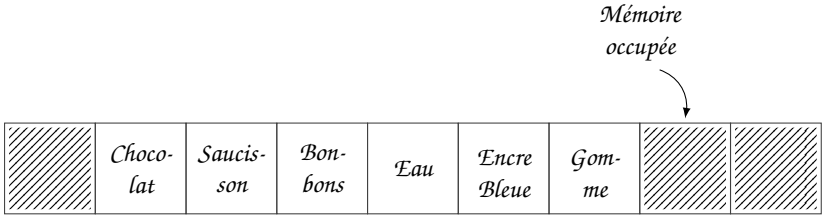
Imaginons qu'il faille enregistrer en mémoire une liste d'emplettes au Champial. Dans un tableau, les éléments sont enregistrés à des cases mémoires contiguës.



Pour lire un élément dans le tableau, il suffit de connaître l'adresse mémoire du début du tableau et l'index de l'élément à lire, ce que l'on peut simplifier en disant que :

adresse de l'élément = adresse du tableau + index de l'élément × *taille d'un élément*,

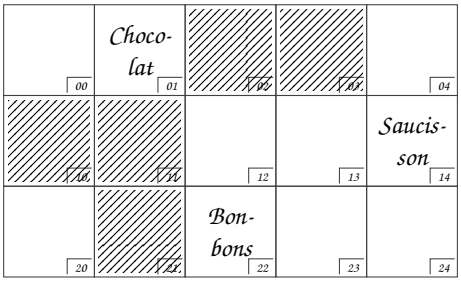
On considère par ailleurs que l'accès est toujours de coût unitaire (c'est-à-dire en $O(1)$). Pour ajouter un élément au tableau, il suffit de l'enregistrer dans le premier emplacement mémoire libre. Dans l'exemple ci-dessus, on peut ajouter trois éléments à la liste d'emplettes. Mais imaginons maintenant qu'il faille ajouter un septième élément : c'est impossible, car la case mémoire suivante est déjà occupée !



La solution est de déplacer **toutes** les données du tableau vers une zone mémoire contenant plus de cases libres adjacentes. L'ajout d'un élément à un tableau déjà complet nécessite un déplacement de toutes les données et coûte donc $O(n)$ où n est la taille du tableau.

1.2 Liste chaînée

Avec une **liste doublement chaînée**, les données peuvent être enregistrées à n'importe quel emplacement mémoire libre.



Chaque élément pointe alors vers l'adresse mémoire de son successeur et de son prédécesseur et on ne retient que l'adresse du premier et du dernier élément.

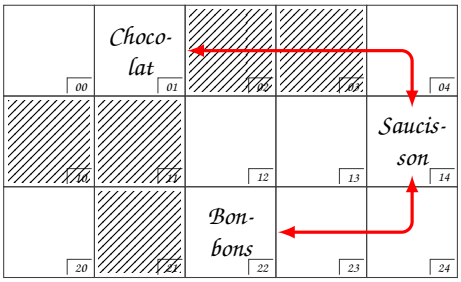


Figure 4.1 – Exemple de liste doublement chaînée. La tête de liste est à l'adresse 01 tandis que la fin de liste est à l'adresse 22.

Ajouter un élément en tête ou en fin de liste est extrêmement simple :

- on l'enregistre n'importe où en mémoire,
- on met à jour les pointeurs :
  - pour un ajout en tête de liste, le nouvel élément devient le prédécesseur de l'ancienne tête de liste et son adresse devient l'adresse du début de la liste (figure 4.3),
  - pour un ajout en fin de liste, le nouvel élément devient le successeur de l'ancienne fin de liste et son adresse devient l'adresse de la fin de la liste (figure 4.2).

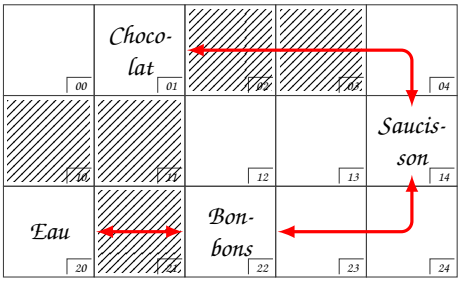


Figure 4.2 – Exemple d'insertion d'un élément en fin de liste. L'adresse de tête de liste reste 01 tandis que l'adresse de fin de liste devient 20.

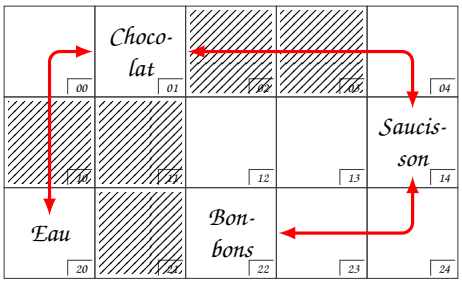


Figure 4.3 – Exemple d'insertion d'un élément en tête de liste. L'adresse de fin de liste reste 22 tandis que la tête de liste est à l'adresse 20.

Pour résumer :

	Tableau	Liste chaînée
Lecture		
Écriture		

1.3 Le type `list` en Python

Le type `list` utilisé par Python est en réalité une structure de tableau, ce qui assure un coût d'accès unitaire à chaque élément. La limitation en taille est masquée par le langage.

Pour simplifier, lors de la création d'une liste à n éléments, l'interpréteur réserve $2n$ emplacements mémoires. Tant que le tableau n'est pas rempli, l'ajout d'un élément avec la méthode `append` a un coût unitaire $O(1)$. Lors de l'ajout du $2n+1^{e}$ élément, le tableau est plein et

ses $2n$ éléments sont recopiés dans un espace mémoire contenant $4n$ emplacements libres adjacents. Cet `append` a donc une complexité en $O(n)$.
Tâchons de caractériser le coût de création d'une liste par ajout successif de ses $N = 2^n$ éléments.

ITC4t12

Ajout d'un élement à une liste

On dit de la méthode d'ajout[a] `append` d'un élément qu'elle a un **coût amorti constant**.

a. En pratique, l'interpréteur `Python` ne réserve pas un espace mémoire égal à $2n$ mais suit la loi suivante : $k = n + \left\lfloor \frac{n}{8} \right\rfloor + \begin{cases} 3 \text{ si } n < 9 \\ 6 \text{ sinon.} \end{cases}$ ce qui laisse toujours de la marge pour ajouter des éléments à la liste.

2 Piles et files

2.1 Rappels

Imaginons que vous souhaitiez organiser votre travail de la semaine : vous conservez la liste des choses à faire sous la forme d'une pile de post-it sur un coin du bureau.

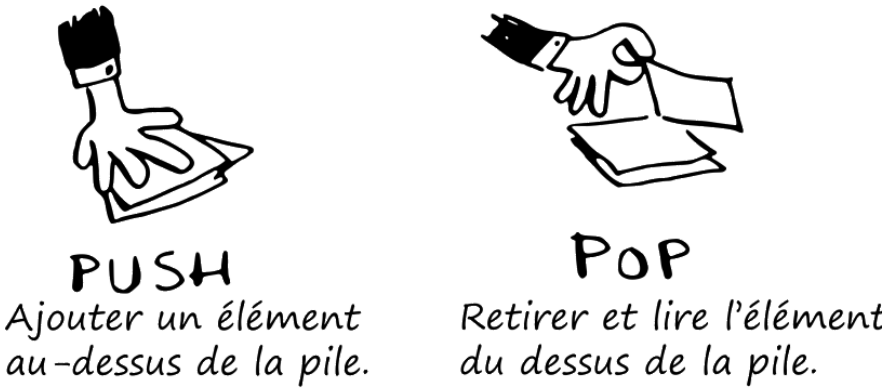
S'il y a une nouvelle tache à faire, vous ajoutez un post-it sur le dessus de la pile. Si vous cherchez quoi faire après avoir terminé le DM de math, vous prenez le post-it suivant : le TD de physique. Quant aux pauvres révisions de chimie des solutions situées en bas de pile, elles n'ont que peu de chances de voir le jour.

Pour répondre à cette dernière remarque, on peut utiliser une structure de données sur le principe du « premier arrivé, premier sorti » : il s'agit de la structure de file. À l'image d'une file d'attente de clients à un guichet, lorsqu'un élément est ajouté à la file, il prend place à la fin. L'élément en tête de file, c'est-à-dire celui qui est dans la file depuis le plus longtemps, est toujours sorti en premier.

2.2 Opérations sur les piles et files

Les deux principales opérations possibles sur une pile sont :

- empiler (push) un élément au sommet de la pile,
- dépiler (pop) l'élément du dessus de la pile pour le lire.



De manière analogue, les deux opérations principales sur une file sont :

- enfiler (enqueue) un élément à la fin de la file,
- défiler (dequeue) le premier élément de la file pour le lire.

D'autres opérations sont disponibles : créer une pile/file (vide ou non), vérifier si la pile/file est vide.
Ces deux structures de données s'implémentent simplement avec la structure de liste doublement chaînée présentée précédemment :

- dans le cas d'une pile, on ajoute et retire toujours le dernier élément de la liste chaînée ;
- dans le cas d'une file, on ajoute l'élément à la fin de la liste, mais on retire l'élément du début.

Comme on se limite au premier et au dernier élément pour l'écriture comme pour la lecture, ces opérations ont un coût unitaire pour les piles et les files.

2.3 Applications

Bien que plus restrictive que les listes en termes de possibilités d'accès, les piles et files ont plusieurs applications classiques :

- dans un navigateur web, une pile sert à mémoriser les pages Web visitées. L'adresse de chaque nouvelle page visitée est empilée et l'utilisateur dépile l'adresse de la page précédente en cliquant le bouton Afficher la page précédente ;
- l'algorithme de parcours en profondeur utilise une pile pour stocker les nœuds à visiter ;
- l'algorithme de parcours en largeur utilise une file pour stocker les nœuds à visiter ;
- lors de l'exécution d'un programme `Python`, les différentes fonctions appelées sont stockées sur une **pile d'appel**. Tant que la dernière fonction exécutée n'est pas terminée, elle n'est pas retirée de la pile ;

Considérons la fonction récursive suivante :

Code Python

Mode éditeur

```
def fibonacci(n, u0, u1):
  if n == 0:
    return u0
  else:
    return fibonacci(n-1, u1, u0+u1)

print(fibonacci(3,1,1))
```

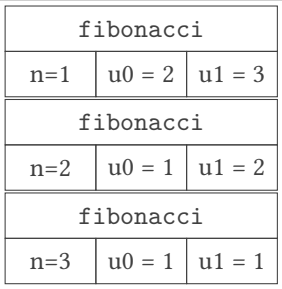
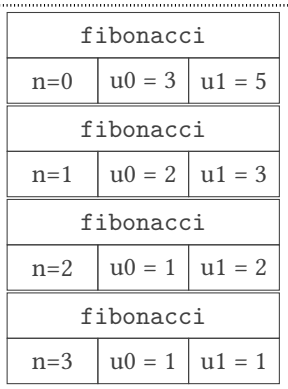
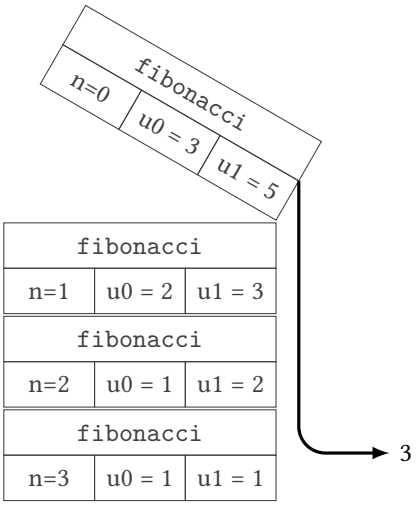
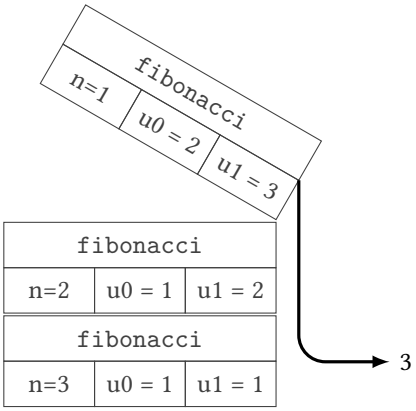
```
3
```

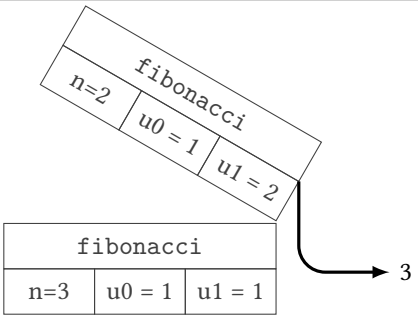
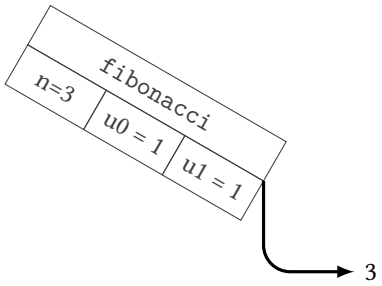
Lorsqu'on exécute `fibonacci(3,1,1)`, cette fonction est placée au sommet de la pile d'appel avec la valeur de ses paramètres d'entrée.

fibonacci		
n=3	u0 = 1	u1 = 1

Représentons en parallèle l'exécution de la fonction récursive et l'état de la pile d'appel.

Code	Remarques	Pile d'appel
`if n==0:`	La condition est fausse	fibonacci: n=3, u0 = 1, u1 = 1
`fibonacci(n-1, u1, u0+u1)`	L'appel à `fibonacci(2, 1, 2)` est ajouté sur la pile.	fibonacci: n=2, u0 = 1, u1 = 2 ; fibonacci: n=3, u0 = 1, u1 = 1
`if n == 0:`	La condition est fausse	

Code	Remarques	Pile d'appel
`fibonacci(n-1, u1, u0+u1)`	`fibonacci(1, 2, 3)` est empilé	
`if n == 0:`	La condition est fausse	
`fibonacci(n-1, u0, u0+u1)`	`fibonacci(0, 3, 5)` est empilé	
`if n == 0:`	La condition est vraie	
`return u0`	La valeur de `u0` est renvoyée	
`return`	Les appels à `fibonacci` sont dépilés car la fonction a terminé.	

Code	Remarques	Pile d'appel
`return`	La valeur 3 remonte les dépilages pour être renvoyée.	
`return`	Elle correspond au terme u_3 de la suite de Fibonacci.	

2.4 Implémentations en Python

Les piles et files ne font pas partie des structures de données de base de Python, mais plusieurs bibliothèques permettent d'en créer. On peut citer la structure `deque` de la bibliothèque `collections`. Sur un élément de type `deque` nommé `15`, les principales méthodes applicables sont :

- `p.append(e)` : ajoute l'élément `e` à droite (ie à la fin) de `15`,
- `p.appendleft(e)` : ajoute l'élément `e` à gauche (ie au début) de `15`,
- `p.pop()` : supprime et renvoie l'élément de droite (dernier élément) de `15`,
- `p.popleft()` : supprime et renvoie l'élément de gauche (premier élément) de `15`.

Ainsi, si on se limite aux méthodes `append()` et `pop()`, on utilise `15` comme une pile tandis que si on utilise les méthodes `append()` et `popleft()`, on utilise `15` comme une file.

Code Python

Mode éditeur

```
from collections import deque

pile = deque([10,30,20])
pile
pile.pop(),pile
pile.append(20),pile
pile.popleft(),pile
```

Application

ITC4.1 : Opérations en notation polonaise inverse

.

En notation polonaise inverse[a], les opérations sont placées après leurs opérandes.
Ainsi, l'expression $2+3$ se note $2,3,+$, et l'expression $2+3\times 4$ peut s'écrire $2,3,4,\times,+$ ou $3,4,\times,2,+$. On voit donc l'intérêt de cette notation : les parenthèses deviennent inutiles.
Dans la suite, les expressions arithmétiques en notation polonaise inverse sont représentées par des tableaux contenant des nombres et des caractères. Par exemple, $2+3\times 4$ peut être représentée par : `[2,3,4,'×','+']`.
L'affichage de la calculatrice HP 48 était constitué de 4 lignes qui donnaient les 4 premiers éléments d'une pile, chaque opération était appliquée au deux premières lignes.

Figure 4.4 – Exemple d'affichage lors de la réalisation de l'opération $2+3\times 4$.

On simule l'affichage de cette calculatrice par une pile nommée `calc` qui contient les opérandes et les résultats des opérations.
Écrire une fonction `RPN(l)` qui prend en argument une liste selon le modèle précédemment introduit et qui affiche les différentes valeurs prise par la pile `calc` au cours de l'exécution. On se limitera aux opérations d'addition et de multiplication.

a. cette notation est utilisée par les anciennes calculatrices HP

3 Dictionnaires et fonctions de hachage

3.1 Problématique

Reprenons les structures de tableau et de liste présentées au début de ce chapitre et imaginons que l'on cherche un élément particulier dans cette structure.
On rencontre alors deux situations :

1. Les éléments ne sont pas triés. On parcourt alors la liste élément par élément jusqu'à trouver la valeur voulue. Cette recherche est linéaire en temps ($O(n)$).
2. Les éléments sont triés dans un tableau. La recherche d'une valeur dans la structure se fait alors par dichotomie avec une complexité logarithmique ($O(\log(n))$).

On souhaiterait améliorer ce temps de recherche d'une valeur. Un type de données répond expressément à cette problématique : le *dictionnaire*.

3.2 Exemple et définitions

Introduisons le problème à l'aide d'un exemple de la vie courante : un épicier possède un répertoire sur lequel il note le prix des différents articles qu'il vend.
Ce répertoire se comporte comme une **table d'association** qui fait correspondre **une clef** (le nom d'un article) avec **une valeur** (son prix).

Définition — *Dictionnaire*

Un **dictionnaire** (ou mieux une table d'association) est un type de données associant un ensemble de clefs à un ensemble de valeurs. Si $\mathcal{C}$ désigne l'ensemble des clefs et $\mathcal{V}$ l'ensemble des valeurs, un dictionnaire est un sous-ensemble $\mathcal{T}$ de $\mathcal{C} \times \mathcal{V}$ tel que pour toute clef $c \in \mathcal{C}$ il existe *au plus* un élément $v \in \mathcal{V}$ tel que $(c, v) \in \mathcal{T}$. Les éléments de $\mathcal{T}$ sont appelés des **associations**.

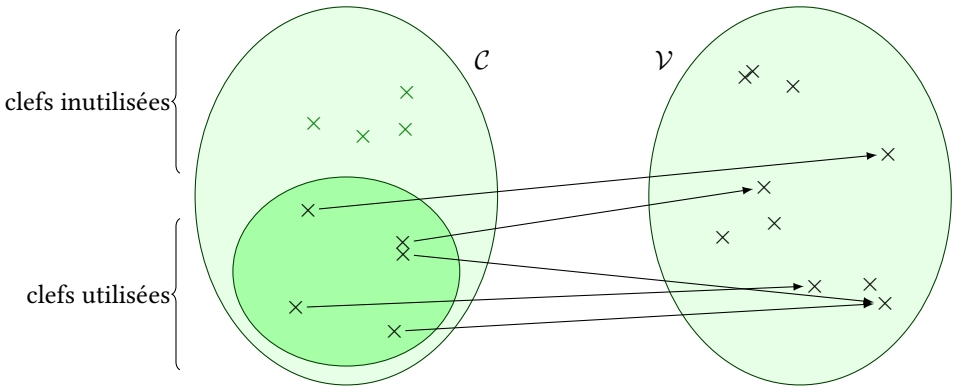


Figure 4.5 – Représentation informelle d'un dictionnaire.

On souhaite que le dictionnaire permette, en temps linéaire, de savoir si une clef est présente dans un dictionnaire, et le cas échéant, de retourner la valeur associée.

3.3 Fonction de hachage et gestion des collisions

Si les m clefs d'un dictionnaire étaient des entiers compris entre 0 et $m-1$, le problème serait simple : il suffirait d'utiliser un tableau de taille m. Comme ce n'est en général pas le cas (dans l'exemple précédent ce sont des mots), on se ramène à cette situation en utilisant une fonction $f : \mathcal{C} \mapsto [\![0, m-1]\!]$ qui associe aux différentes clefs possibles un entier.

ITC4t13

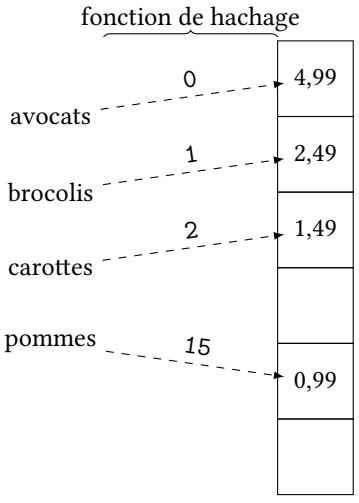


Figure 4.6 – Utilisation de la table de hachage en absence de collision

Le nombre de clefs possibles étant très important, le cardinal de $\mathcal{C}$ est beaucoup plus grand que m et la fonction de hachage n'est généralement pas injective : on dit que deux clefs dont la valeur de hachage est commune génèrent une collision .
Plusieurs solutions de traitement des collisions existent, on ne présentera ici que les deux méthodes les plus utilisées.

3.3.1 Chaînage

Lorsque deux clefs entrent en collision, on crée une liste chaînée à partir de l'adresse mémoire renvoyée par la fonction de hachage comme présenté dans la figure 4.7 : les couples (clef:valeur) correspondant à la même valeur de hachage sont enregistrés successivement dans une liste chaînée.

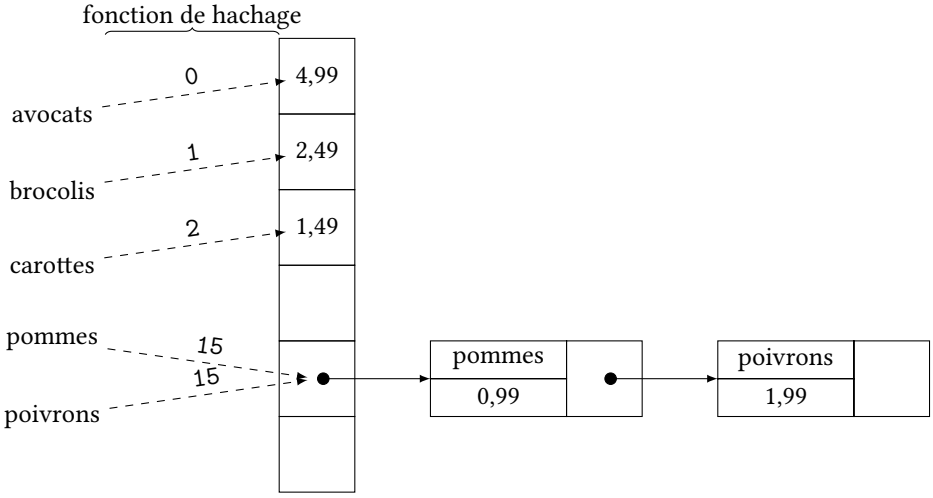


Figure 4.7 – Exemple de résolution d'une collision avec une liste simplement chaînée

Cette méthode a pour avantage de pouvoir accueillir un très nombre de clefs donnant la même valeur de hachage. En revanche, le coût d'accès peut devenir aussi mauvais que dans une liste (imaginons un magasin qui ne vende que des articles commençant par la lettre `A` ...), à savoir linéaire.

3.3.2 Adressage ouvert

L'adressage ouvert consiste, dans le cas d'une collision, à déterminer par *sondage* les emplacements libres puis à y stocker les paires clef–valeur dégénérées. Dans le cas de `Python`, ce sondage est aléatoire, c'est-à-dire que l'interpréteur cherche aléatoirement un emplacement mémoire libre.

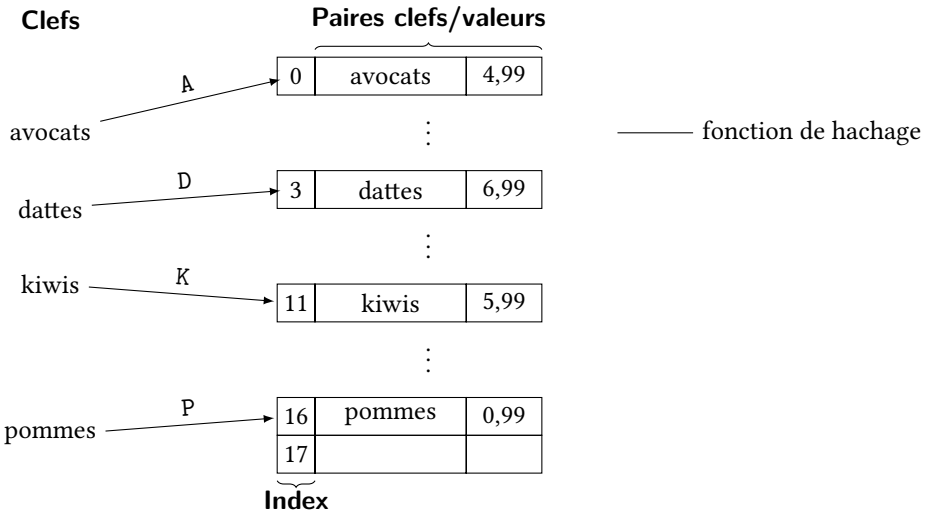


Figure 4.8 – Utilisation de la table de hachage en absence de collision

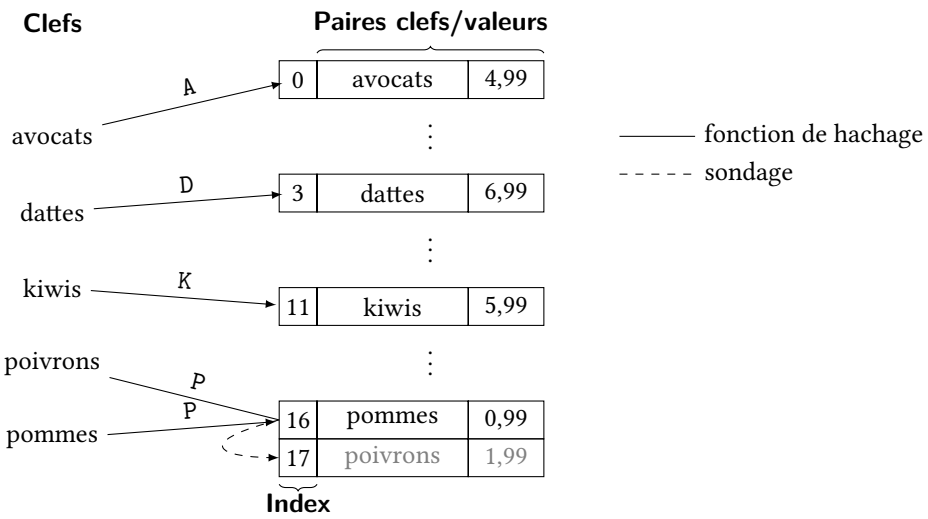


Figure 4.9 – Exemple de résolution d'une collision par sondage

Lors d'une recherche, si l'adresse obtenue par hachage direct ne permet pas d'obtenir la bonne clef, une recherche sur les adresses obtenues par une méthode de sondage est effectuée jusqu'à trouver une clef non attribuée.

3.3.3 Généralisation

Dans la réalité, la fonction de hachage travaille dans un intervalle d'arrivée beaucoup plus grand (de l'ordre de taille 2^{128} et doit posséder les caractéristiques suivantes :

- elle doit être facile à calculer,
- elle doit être stable : appliquée à une même clef, elle doit toujours renvoyer la même valeur,
- idéalement, elle renvoie deux valeurs différentes pour deux clefs distinctes afin d'éviter les collisions. Ceci étant impossible à garantir, on essaye de trouver une fonction de hachage dont la distribution est la plus uniforme possible.

La création d'une telle fonction de hachage est très complexe (et donc hors-programme). La fonction de hachage utilisée par `Python` est accessible par la fonction `hash(x)` où `x` est un objet immuable : ce peut être un nombre (entier ou flottant), une chaîne de caractères ou encore un tuple, mais pas une liste.

Code Python

Mode éditeur

```
hash(12345)
hash("12345")
hash((1,2,3,4,5))
# À ne pas faire
# hash([1,2,3,4,5])
```

Qu'en est-il pour le temps d'accès à un élément ?

ITC4t14

3.4 Les dictionnaires en pratique

3.4.1 Création d'un dictionnaire

En Python, un dictionnaire est délimité par des accolades, les valeurs sont affectées aux clefs avec « : ». On peut créer un dictionnaire vide à l'instar des listes.

Code Python

Mode éditeur

```
prix = {"carottes": 1.49, "brocolis":2.49, "pommes":0.99}
dico_vide = {}
dico_vide2= dict()
```

On accède à une valeur du dictionnaire en utilisant la clef entourée par des crochets.

Code Python

Mode éditeur

```
print(prix["pommes"])
```

```
0.99
```

Il est également possible d'ajouter une valeur pour une nouvelle clef en utilisant les crochets.

Code Python

Mode éditeur

```
prix["avocats"] = 4.99
print(prix)
```

```
{'carottes': 1.49, 'brocolis': 2.49, 'pommes': 0.99, 'avocats': 4.99}
```

Pour supprimer une paire clef-valeur, on utilise la méthode `pop(clef)`. Comme pour la méthode `pop()` appliquée à une liste, cette méthode renvoie la valeur de la paire qui a été supprimée.

Code Python

Mode éditeur

```
prix.pop("carottes")
print(prix)
```

```
{'brocolis': 2.49, 'pommes': 0.99, 'avocats': 4.99}
```

3.4.2 Parcours d'un dictionnaire

On peut parcourir un dictionnaire de trois manières différentes :

1. en ne parcourant que les clefs avec la méthode `keys()`

Code Python

Mode éditeur

```
for clef in prix.keys():
    print(clef)
```

```
carottes
brocolis
pommes
```

2. en ne parcourant que les valeurs avec la méthode `values()`

Code Python

Mode éditeur

```
for valeur in prix.values():
    print(valeur)
```

```
1.49
2.49
0.99
```

3. en récupérant à la fois les clefs et les valeurs avec la méthode `items()`

Code Python

Mode éditeur

```
for clef, valeur in prix.items():
    print("Le prix au kilo des", clef, "est :", valeur)
```

```
Le prix au kilo des carottes est : 1.49
Le prix au kilo des brocolis est : 2.49
Le prix au kilo des pommes est : 0.99
```

4 Programmation dynamique

4.1 Un problème posé par la programmation récursive

Nous allons nous intéresser au calcul du coefficient binomial $\binom{n}{p}$. Une solution consiste à utiliser la programmation récursive et la formule de Pascal, ce qui nous amène à écrire :

Tâchons de comprendre l'évolution désastreuse des temps d'exécution : l'arbre de calcul de $\binom{5}{2}$ montre que des appels récursifs identiques, donc superflus, sont réalisés.

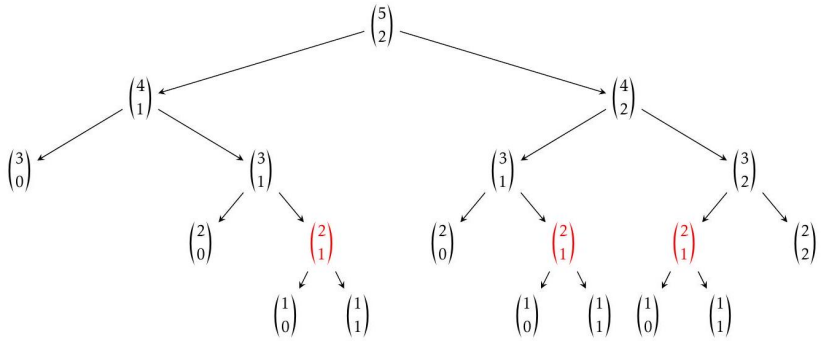


Figure 4.10 – Le calcul de $\binom{5}{2}$ fait appel par trois fois au calcul de $\binom{2}{1}$.

Point culture

Évaluation de la complexité temporelle de cette fonction

Notons $C(n,p)$ le nombre d'additions réalisées par la fonction précédente. Le nombre d'opérations effectuées est tel que :

$$C(n,0) = C(n,p) = 0$$
$$\forall m \in [\![1, n-1]\!], \quad C(n,m) = C(n-1,m-1) + C(n-1,m) + 1$$

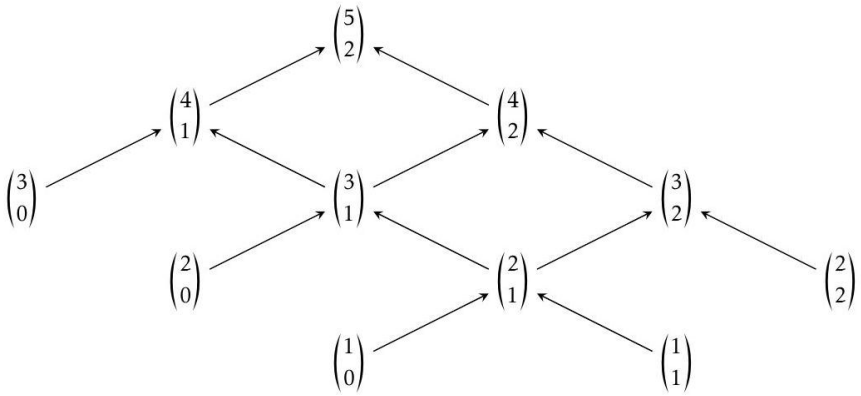
On démontre alors par récurrence sur $n \in \mathbb{N}$ que

$$\forall p \in [\![0,n]\!], \quad C(n,p) = \binom{n}{p} - 1$$

Ainsi, en utilisant la formule de Stirling : $C(2n,n) \sim \binom{2n}{n} \sim \frac{4^n}{\sqrt{\pi n}}$, le calcul de $\binom{2n}{n}$ se trouve être de complexité exponentielle[b].

b. l'expérience montre que le calcul de $\binom{30}{15}$ fait appel 40 116 600 fois au calcul de $\binom{2}{1}$.

Le problème à résoudre, ici le calcul de $\binom{n}{p}$, se ramène à la résolution de deux sous-problèmes interdépendants : le calcul de $\binom{n-1}{p-1}$ et celui de $\binom{n-1}{p}$.
La solution proposée par la programmation dynamique consiste à commencer par résoudre les plus petits des sous-problèmes, puis de combiner leurs solutions pour résoudre des sous-problèmes de plus en plus grands. Concrètement, le calcul de $\binom{5}{2}$ se réalise en suivant le schéma suivant :



Pour réaliser ce type de solution on utilise souvent un tableau, ici un tableau bi-dimensionnel $(n+1) \times (p+1)$ (dont seule la partie pour laquelle $i \geqslant j$ sera utilisée). Ce tableau sera progressivement rempli par les valeurs des coefficients binomiaux, en commençant par les plus petits.

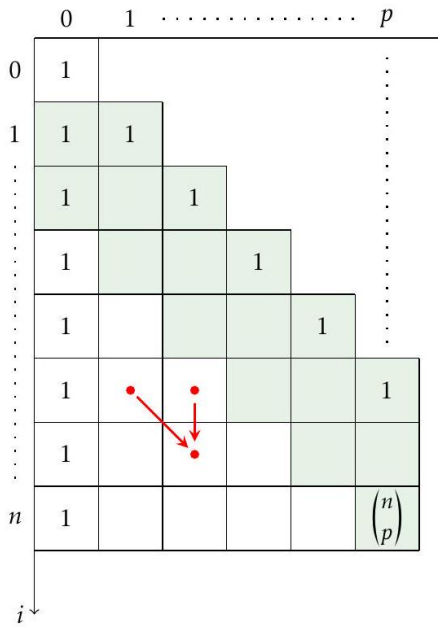


Figure 4.11 – Le schéma de dépendance du calcul de $\binom{n}{p}$: les flèches rouges modélisent l'ordre de remplissage : la case destinée à recevoir la valeur de $\binom{i}{j}$ ne peut être remplie qu'après celles contenant $\binom{i-1}{j-1}$ et $\binom{i-1}{j}$.

On peut ainsi donner une version itérative du code permettant de suivre cette approche :

Code Python

Mode éditeur

```
def binomIt(n, p):
    t = pl.zeros((n + 1, p + 1))
    #
    for i in range(0, n + 1):
        t[i, 0] = 1
    #
    for i in range(1, p + 1):
        t[i, i] = 1
    #
    for i in range(2, n + 1):
        #
        for j in range(1, min(p, i) + 1):
            t[i, j] = t[i - 1, j - 1] + t[i - 1, j]
    return t[n, p]
```

Déterminons la complexité temporelle d'un tel algorithme :

ITC4t15

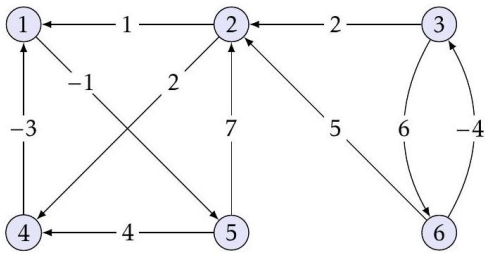
Un inconvénient majeur de cette approche réside dans la perte de lisibilité de l'algorithme : l'idéal serait de combiner l'élégance de la programmation récursive avec l'efficacité de la programmation dynamique.
La solution, déjà vue en première année, porte le nom de **mémoïsation**. Elle consiste à associer à la fonction un dictionnaire qui stockera les valeurs des termes évalués et ne réalisera les opérations que lorsqu'elles sont nécessaires.
Proposons un algorithme implémentant cette démarche :

L'affichage du dictionnaire permet de déterminer l'ordre dans lequel sont évalués les différentes valeurs.

4.2 Algorithme de Floyd-Warshall

Revenons pour terminer ce chapitre sur l'étude des graphes et cherchons à améliorer l'algorithme de recherche du plus court chemin vu en première année : nous nous étions alors limité au cas des arêtes de poids positif. Qu'en est-il quand des chemins de poids négatifs sont présents ?

Considérons donc un graphe orienté et pondéré représenté en machine par une matrice $M \in \mathcal{M}_n(\mathbb{Z})$, pour laquelle le coefficient $m_{i,j}$ sera égal au poids de l'arête reliant les sommets v_i à v_j si elle existe, ou sinon à $+\infty$.



$$M = \begin{pmatrix} 0 & +\infty & +\infty & +\infty & -1 & +\infty \\ 1 & 0 & +\infty & 2 & +\infty & +\infty \\ +\infty & 2 & 0 & +\infty & +\infty & 6 \\ -3 & +\infty & +\infty & 0 & +\infty & +\infty \\ +\infty & 7 & +\infty & 4 & 0 & +\infty \\ +\infty & 5 & -4 & +\infty & +\infty & 0 \end{pmatrix}$$

Figure 4.12 – Exemple de graphe pondéré et matrice d'adjacence associée.

L'algorithme de Floyd-Warshall a pour objet de calculer, pour chacun des couples de points $\left(v_i, v_j\right)$, le chemin de poids minimal reliant v_i à v_j s'il existe.

Pour ce faire, cet algorithme calcule la suite finie de matrices $M^{(k)}$, $0 \leqslant k \leqslant n$ avec $M^{(0)} = M$ et :

$$\forall k < n, \quad \forall (i,j) \in [\![1,n]\!]^2, \quad m_{ij}^{(k+1)} = \min\left(m_{ij}^{(k)}, m_{i,k+1}^{(k)} + m_{k+1,j}^{(k)}\right).$$

Afin de prouver que $m_{ij}^{(n)}$ est le poids minimal d'un chemin reliant v_i à v_j, nous pourrons établir le résultat suivant : si G ne contient pas de cycle de poids strictement négatif, alors $m_{ij}^{(k)}$ est égal au poids du chemin minimal reliant v_i à v_j et ne s'autorisant de passer que par des sommets parmi $v_1, v_2, \ldots, v_k$.

ITC4t16

En l'absence de cycle de poids négatif : $m_{i,k+1}^{(k)} = m_{i,k+1}^{(k+1)}$ et $m_{k+1,j}^{(k)} = m_{k+1,j}^{(k+1)}$.

La relation de récurrence peut donc indifféremment s'écrire :

$$\begin{aligned} m_{ij}^{(k+1)} &= \min\left(m_{ij}^{(k)}, m_{i,k+1}^{(k)} + m_{k+1,j}^{(k)}\right) \\ &= \min\left(m_{ij}^{(k)}, m_{i,k+1}^{(k+1)} + m_{k+1,j}^{(k)}\right) = \min\left(m_{ij}^{(k)}, m_{i,k+1}^{(k)} + m_{k+1,j}^{(k+1)}\right) \\ &= \min\left(m_{ij}^{(k)}, m_{i,k+1}^{(k+1)} + m_{k+1,j}^{(k+1)}\right) \end{aligned}$$

L'ordre dans lequel sont mises à jour les cases d'indices (i,j), $(i,k+1)$ et $(k+1,j)$ n'a pas d'importance et permet ainsi d'optimiser la complexité spatiale du code proposé en modifiant la matrice M au cours de l'exécution. Nous pouvons ainsi proposer une fonction permettant de déterminer la matrice $M^{(n)}$ associée à un graphe à n sommets :

La complexité temporelle de l'algorithme de Floyd-Warshall est en $O\left(n^3\right)$, où n est l'ordre du graphe. L'observation de la matrice obtenue permet de dire aisément quels sommets sont accessibles depuis un sommet donné. L'utilisation de booléens permet de répondre efficacement à cette question.

IIC 5

Apprentissage supervisé et théorie des jeux

Sommaire

1 Jeux sur un graphe simple : le Chomp 70
1.1 Présentation du jeu 70
1.2 Détermination des positions gagnantes 73
2 Algorithme `min-max` 75
2.1 Heuristique 75
2.2 Approche `min-max` 75
3 Intelligence artificielle et apprentissage 78
3.1 Apprentissage supervisé 78
3.2 Apprentissage non supervisé 81

Sous le terme d'intelligence artificielle se cachent un ensemble de techniques permettant à des machines d'accomplir des tâches et de résoudre des problèmes normalement réservés aux humains On associe presque toujours Grâce à l'apprentissage, un système intelligent peut améliorer ses performances avec l'expérience. On en distingue deux types : l'apprentissage supervisé, le plus fréquent, consiste pour un opérateur à montrer à la machines des milliers voire des millions d'exemples étiquetés avec leur catégorie. Une fois cette phase d'apprentissage terminée, la machine doit être capable de généraliser à des objets pas encore vu. L'apprentissage non supervisé, plus proche de notre propre modèle d'apprentissage, consiste à faire en sorte qu'à partir d'un ensemble de données la machine soit capable de créer ses propres catégories.
Une des prouesses marquantes de l'IA fut sa victoire au jeu de go[a]*. Nous allons ainsi nous intéresser à l'étude théorique et algorithmique de jeux à deux joueurs antagonistes jouant alternativement en information totale : chaque joueur a une vision complète de l'état du jeu (la plupart des jeux de cartes sont exclus, mais reste des jeux tels les échecs, les dames, le go, etc.).*
Dans un premier temps, nous allons nous intéresser à des jeux simples pour lesquels il est possible de déterminer une stratégie gagnante, puis nous verrons, pour les jeux les plus complexes, comment bâtir une stratégie à l'aide d'une heuristique.

a. AlphaGO vs. Ke Jie (n°1 mondial) en 2017

1 Jeux sur un graphe simple : le Chomp

1.1 Présentation du jeu

1.1.1 Introduction

Le premier jeu auquel nous allons nous intéresser se joue à l'aide d'une tablette de chocolat rectangulaire dont le coin supérieur gauche est empoisonné : chaque joueur choisit à tour de rôle un carré et le mange, ainsi que tous les morceaux situés à la droite et en dessous du carré choisi. Bien évidemment, le joueur[b] qui n'a plus d'autre choix que de manger le carré empoisonné a perdu.

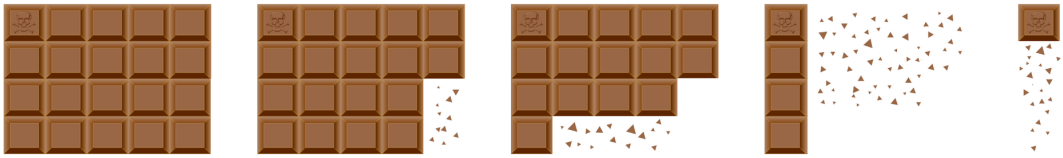


Figure 5.1 – Configuration initiale et exemple d'une partie du jeu de Chomp perdue par Adam

1.1.2 Modélisation et étude théorique

Pour modéliser un jeu, c'est à dire rendre compte des différentes possibilités qui se présentent à chaque tour, le formalisme de la théorie des graphes se révèle parfaitement adapté.

Définition — *Arène*

On nomme **arène** un graphe orienté (S, A) qui est tel que :

- chaque sommet de S représente une configuration du jeu (une position), l'une d'entre elles étant la position de départ
- une arête $a = (s_1, s_2) \in A$ reliant deux sommets indique la possibilité pour un joueur de passer de la position s_1 à la position s_2 en un coup.

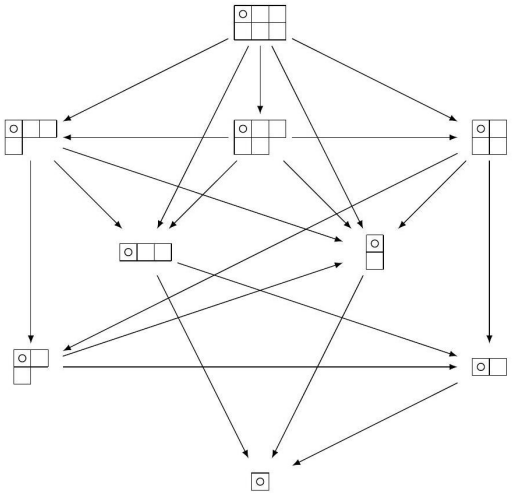


Figure 5.2 – Arène associée au Chomp $(2, 3)$

Une partie de Chomp correspond à un chemin d'origine s_0 dans l'arène.

Définition — *Jeu d'accessibilité*

Un jeu d'accessibilité est associé à une arène qui ne peut contenir de cycle : toute partie est finie. Il est caractérisé par un ensemble de positions (les **cibles**) qui sont sans successeurs.

b. Tout comme on nomme Alice et Bob les émetteurs et récepteurs en cryptographie, on nomme très souvent Adam et Ève les deux joueurs d'une partie, Adam étant, sauf mention contraire, celui qui joue le premier coup.

Cette structure ne rend pas compte du fait que le jeu implique deux joueurs dont les coups se succèdent sans annulation possible. Pour cela, on crée un nouveau graphe dit **biparti** dans lequel on double chacun des sommets (indexés par le nom du joueur), que l'on partitionne en deux sous-ensembles S_a et S_e tels que :

- S_i désigne l'ensemble des positions à partir desquelles le joueur i jouera ;
- $S = S_a \cup S_e$;
- $S_a \cap S_e = \emptyset$;
- les arêtes ne peuvent relier qu'un nœud de S_a à un nœud de S_e, et réciproquement.

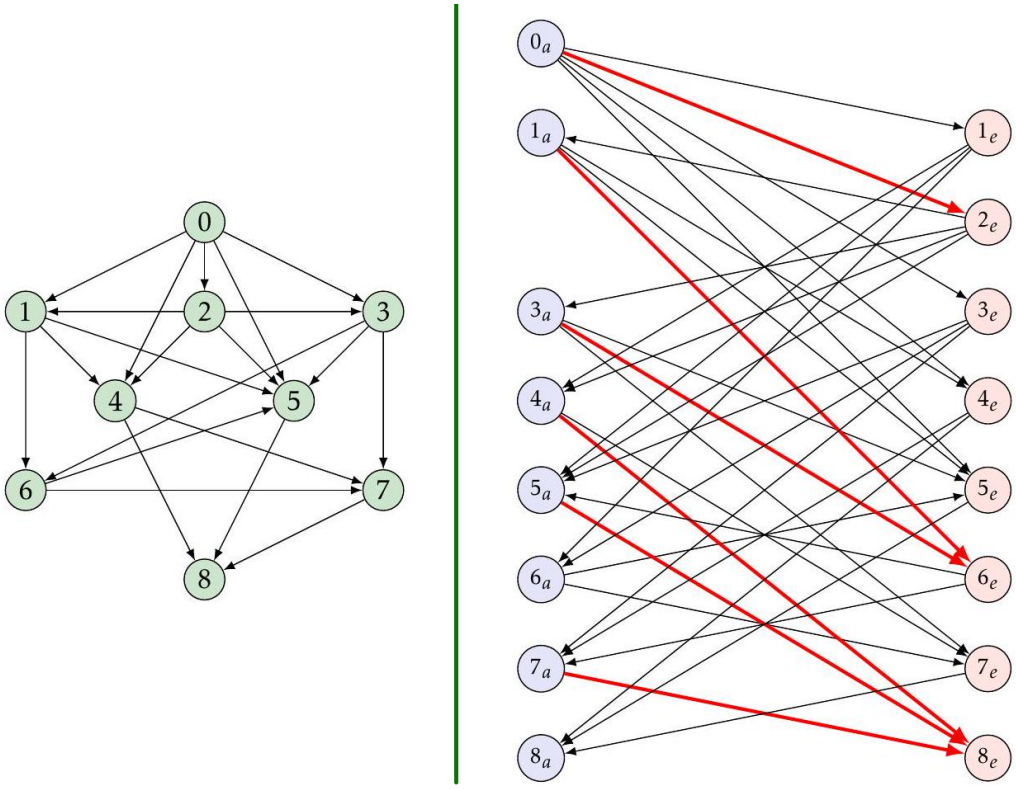


Figure 5.3 – Graphe, graphe biparti et stratégie pour un premier coup d'Adam en 2

Stratégies gagnantes

Stratégie et stratégie gagnante

Définition

Soit une arène (S, A) et $(\mathcal{S}, \mathcal{A})$ le graphe biparti associé.
Une **stratégie** pour Adam est une application

$$f : S'_a \subset S_a \to S_e \quad \text{telle que} \quad \forall s \in S'_a \quad (s, f(s)) \in \mathcal{A}$$

Une **partie** $s_0 \to s_1 \to \cdots \to s_n$ est dite **jouée suivant f** lorsque :

$$\forall k \in [\![0, n-1]\!] \quad s_k \in S'_a \Rightarrow s_{k+1} = f(s_k)$$

Une **stratégie** est dite **gagnante** pour Adam si toute partie jouée en suivant cette stratégie est gagnante pour Adam.

ITC5t17

Positions gagnantes

Définition — **Position gagnante**

Une **position** s est dite **gagnante** lorsqu'il existe une stratégie gagnante pour une partie débutant au sommet s.

Démontrons la propriété suivante : dans le jeu de Chomp (p, q) il existe une stratégie gagnante pour le premier joueur dès lors que $(p, q) \neq (1, 1)$.

ITC5t18

Ce type de preuve, appelé vol de stratégie, est malheureusement non constructif : savoir qu'une stratégie gagnante existe ne suffit pas à la déterminer.

1.2 Détermination des positions gagnantes

Considérons un graphe orienté acyclique (S, A) associé à un jeu d'accessibilité.
Montrons tout d'abord que dans un graphe acyclique il existe des sommets sans successeur.

ITC5t19

Définition — Stabilité, absorbance et noyau

Un sous-ensemble de sommets S' est dit **stable** si aucun sommet de S' n'a de successeur dans S'.
Un sous-ensemble S' de sommets est dit **absorbant** si tout sommet n'appartenant pas à S' possède au moins un successeur dans S'.
Un sous-ensemble S' de sommets est un **noyau** s'il est à la fois stable et absorbant.

On peut alors montrer que tout graphe orienté et acyclique possède un unique noyau.

ITC5t20

Cette preuve a le mérite de d'être constructive ; pour calculer le noyau de (S, A) il suffit de :

- chercher un sommet s de (S, A) sans successeur : il appartient au noyau ;
- supprimer le sommet s de (S, A) ainsi que tous ses prédécesseurs ;
- recommencer tant qu'il reste des sommets.

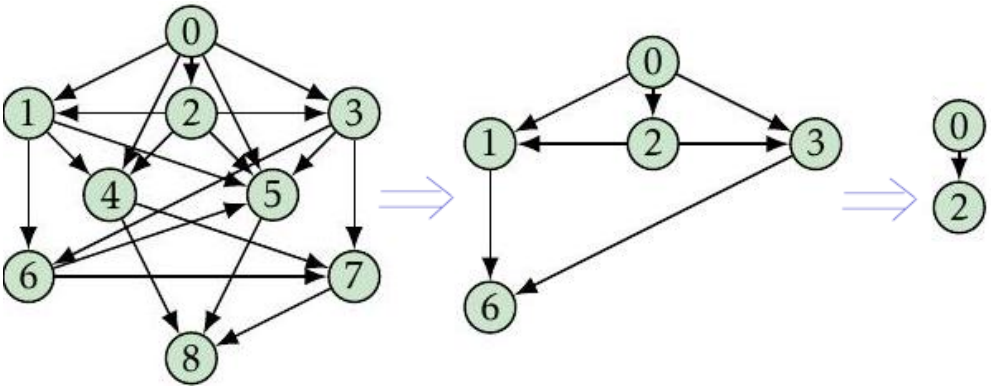


Figure 5.4 – Détermination du noyau d'un jeu de Chomp $(2, 3)$

Application

ITC5.1 : Algorithme de calcul du noyau

On considère un graphe acyclique (S, A). Il est représenté en machine par la donnée d'un dictionnaire d dont les clefs sont les sommets et les valeurs les successeurs des clefs (sous forme de liste).
On a donné à titre d'exemple une représentation du graphe associé au jeu de Chomp $(2, 3)$.

```
d = 0:[1,2,3,4,5],1:[4,5,6],2:[1,3,4,5],3:[5,6,7],4:[7,8],5:[8],
6:[5,7],7:[8],8:[]
```

Q.1. Rédiger une fonction `sans_successeur(d)` qui renvoie un sommet sans successeur.

Q.2. Rédiger une fonction `predecesseurs(d,j)` qui renvoie la liste de tous les prédécesseurs du sommet j.

Q.3. Rédiger une fonction `supprime_sommet(d,j)` qui supprime un sommet du graphe.

Q.4. En déduire[c] une fonction `noyau(d)` qui renvoie la liste des sommets constituant le noyau de (S, A).

Code Python

Node éditeur

```
print(sans_successeur(d),predecesseurs(d,8),noyau(d))
```

```
8 [4, 5, 7] [8, 6, 2]
```

c. L'algorithme proposé a une complexité en $O(n^3)$. Sachant que le jeu de Chomp (p, q) possède $n = C_p^{p+q}$ sommets, on conçoit que le calcul du noyau se révèle rapidement d'un coût rédhibitoire dès que p et q deviennent trop grands car pour $p = q$ on a $n \sim \frac{4^p}{\sqrt{\pi p}}$ soit une croissance exponentielle de la complexité avec p.

2 Algorithme `min-max`

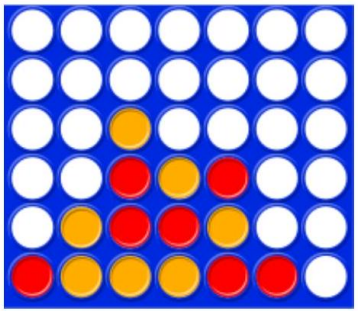
Le nombre de positions n que l'on peut rencontrer lors d'une partie est souvent extrêmement grand : il est estimé de l'ordre de 10^{32} pour les dames, entre 10^{43} et 10^{50} pour le jeu d'échecs, de l'ordre de 10^{100} pour le jeu de go. Le calcul du noyau de complexité cubique se révèle donc généralement impossible. Il devient donc nécessaire de s'appuyer non plus sur une évaluation exacte de la position, mais sur une estimation de la valeur de la position atteinte.

2.1 Heuristique

Dans la suite de cette section, nous supposerons posséder une fonction h, nommée heuristique, qui à toute position légale p du jeu associe un réel $h(p)$ tel que :

- plus $h(p)$ est grand, meilleure est la position pour Adam ;
- plus $h(p)$ est petit, meilleure est la position pour Ève.

Puissance 4 Pour illustrer cette section, nous allons prendre l'exemple du Puissance 4 : le but du jeu est d'aligner une suite de quatre pions de même couleur sur une grille comptant six rangées et sept colonnes.
Une heuristique simple consiste à attribuer à chaque case une valeur, par exemple le nombre d'alignements potentiels de quatre pions lorsqu'on place un pion à cet emplacement, puis à sommer positivement les valeurs de cases occupées par les jetons d'Adam, négativement par ceux d'Ève ; ceci confère à chaque état du jeu une valeur numérique.



3	4	5	7	5	4	3
4	6	8	10	8	6	4
5	8	11	13	11	8	5
5	8	11	13	11	8	5
4	6	8	10	8	6	4
3	4	5	7	5	4	3

Figure 5.5 – Une grille de Puissance 4 (Adam joue les jaunes) et valeur associée par case occupée

L'heuristique sera égale à $+\infty$ pour une position gagnante pour Adam, et à $-\infty$ pour une position gagnante pour Ève.

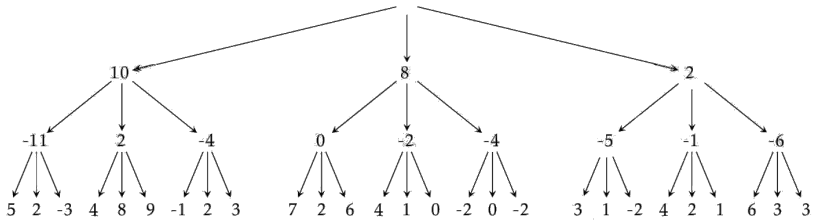
2.2 Approche `min-max`

Au moment où l'un des deux protagonistes doit jouer, plusieurs possibilités s'offrent à lui (entre une et sept pour le Puissance 4). Une solution simple pour choisir le coup à jouer consiste à calculer l'heuristique correspondant à chacune des configurations atteignables et à jouer celle d'heuristique maximale (pour Adam) ou minimale (pour Ève).

Application

ITC5.2 : Puissance 4

On représente ici l'arbre de choix qui s'offre à Adam qui dispose d'un coup à jouer.



Q.1. S'il ne raisonne que sur un seul coup, quel choix Adam doit-il opérer ?

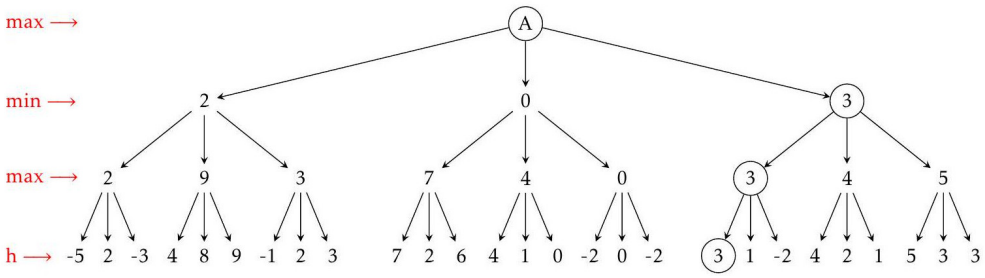
Q.2. S'il anticipe la réaction d'Ève, dont il suppose que le choix sera optimal au premier ordre, quel coup doit-il jouer ?

Q.3. S'il prend en compte les deux coups successifs qu'il pourra jouer, quel doit être ce fois le choix d'Adam ?

On voit que si ce raisonnement peut théoriquement être itéré, la croissance exponentielle du nombre de configurations à examiner nécessite de limiter la profondeur de la recherche.

L'algorithme min-max On peut donc proposer l'algorithme suivant pour déterminer le meilleur coup d'Adam (au sens d'optimal à n itérations) :

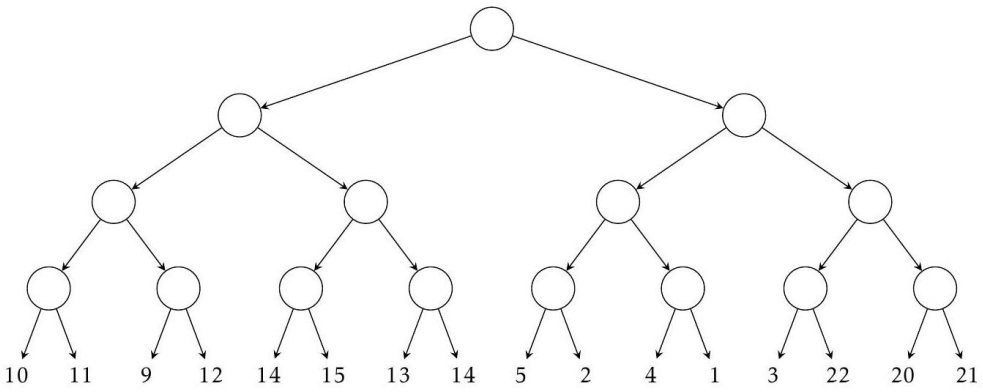
- calculer la valeur de l'heuristique de toutes les positions atteignables en n coups (les feuilles de l'arbre) ;
- en raisonnant sur le joueur qui joue le premier, définir si la position de départ est un maximum (pour Adam) ou un minimum (pour Ève). Associer ensuite alternativement à chaque niveau descendant un qualificatif `max` puis `min`, qui désigne si le parent a pour valeur le maximum ou le minimum de ses enfants ;
- en considérant un niveau rempli, remplir le niveau supérieur en attribuant au parent le maximum de la valeur de ses enfants si le niveau est `max`, le minimum sinon ;
- redescendre le chemin correspondant à l'obtention de la valeur maximale



Application

ITC5.3 : Mise en œuvre de l'approche min-max

Calculer la valeur de la position associée à l'arbre ci-dessous, dans le cas où le joueur qui cherche à maximiser l'heuristique commence. Faire de même si c'est son adversaire qui débute.



La traduction en Python de cette approche repose sur l'écriture de deux fonctions :

- `maximin(p,n)` (destinée à Adam) va chercher à maximiser l'heuristique après n coups en partant de la position p, en supposant que son adversaire joue au mieux ;
- `minimax(p,n)` (destinée à Ève) va chercher à minimiser l'heuristique après n coups en partant de la position p, en supposant que son adversaire joue au mieux.

Pour la rédaction des programmes, on suppose définies une fonction `h(p)` qui prend pour argument une position du jeu et renvoie la valeur de son heuristique, ainsi qu'une fonction `successeurs(p)` qui renvoie la liste des positions atteignables à partir de la position p.

Code Python

Mode éditeur

```
def minimax(p, n):
  if n == 0 or successeurs(p) == []:
    return h(p)
  mini = np.inf
  for pk in successeurs(p):
    s = maximin(pk, n - 1)
    if s < mini:
      mini = s
  return mini
```

Code Python

Mode éditeur

```
def maximin(p, n):
  if n == 0 or successeurs(p) == []:
    return h(p)
  maxi = -np.inf
  for pk in successeurs(p):
    s = minimax(pk, n - 1)
    if s > maxi:
      maxi = s
  return maxi
```

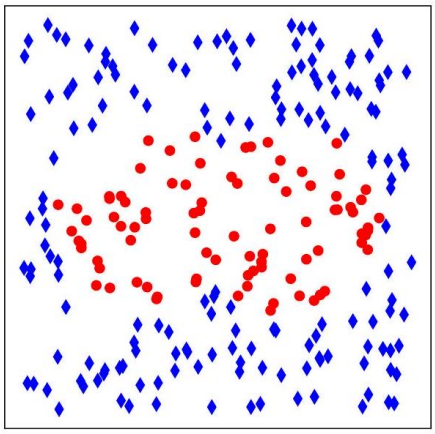
3 Intelligence artificielle et apprentissage

3.1 Apprentissage supervisé

Dans l'exemple que nous venons de présenter, le choix de l'heuristique détermine totalement l'efficacité de l'algorithme min-max. Le choix de cet heuristique dépend des règles du jeu, mais peut aussi être affecté par les caractéristiques du joueur adverse, la prise de décision irrationnelle, ... L'apprentissage d'une telle heuristique constitue une des notions de base de ce que l'on nomme « intelligence artificielle » (ou I.A.).

3.1.1 Un premier exemple

On se donne comme premier exemple un ensemble de 250 points de coordonnées $(x,y) \in [0,1]^2$, qui peuvent appartenir à deux catégories : les losanges bleus ou les disques rouges. Le but est d'affecter de façon la plus « logique » une couleur à un point tiré aléatoirement dans ce fermé.



Une façon de répondre à cette problématique est d'utiliser la **méthode des k plus proches voisins** : on attribue au point de coordonnées (x,y) la catégorie majoritaire parmi ses k plus proches voisins.
Pour ce faire, on se donne une liste `donnees` contenant les données d'apprentissage sous la forme (x,y,c) avec $(x,y) \in [0,1]^2$ et $c \in$ ["B","R"], ainsi qu'un entier impair k.
La fonction `plus_proches_voisins(M,k)` utilise une fonction de tri native de façon à ne conserver que les k points qui ont la distance au point $M(x,y)$ la plus faible. Tout point de $[0,1]^2$ peut être traité de la sorte.

Code Python

Mode éditeur

```
def plus_proches_voisins(M, k):
  x,y=M
  t = sorted(donnees, key=lambda d: (d[0]-x)**2 + (d[1]-y)**2)[:k
                                    ]
  r = 0
  for X,Y,col in t:
    if col == "R":
      r += 1
  if 2 * r > k:
    return "R"
  else:
    return "B"
```

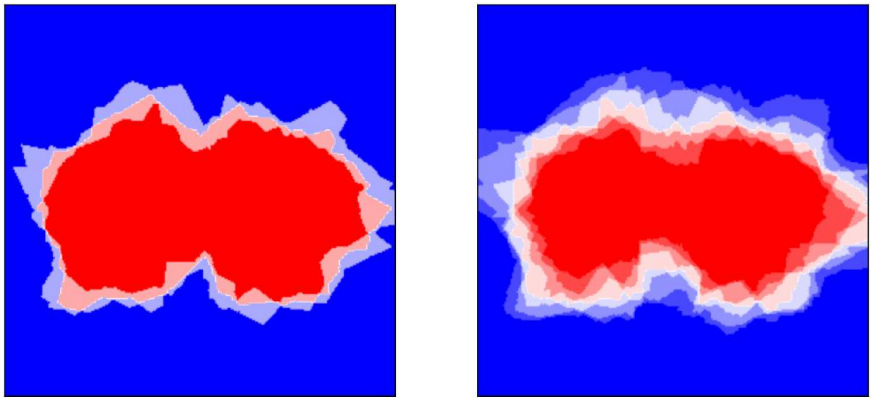


Figure 5.6 – Le résultat de `plus_proches_voisins` pour $k = 3$ et $k = 7$

Matrice de confusion Pour tester la qualité de l'apprentissage, on utilise un nouveau jeu de 100 autres données, toujours sous la forme (x, y, c) pour lesquels on compare la couleur obtenue par la méthode des k plus proches voisins et la couleur « réelle ». Le résultat permet de répartir les points tests en quatre catégories :

- les vrais positifs (les points rouges reconnus comme tels) ;
- les vrais négatifs (les points bleus reconnus comme tels) ;
- les faux positifs (les points bleus reconnus à tort comme rouges) ;
- les faux négatifs (les points rouges reconnus à tort comme bleus).

Ces quatre valeurs sont rangées dans la matrice de confusion $M_C = \begin{pmatrix} \text{VP} & \text{FN} \\ \text{FP} & \text{VN} \end{pmatrix}$.

Code Python

Mode éditeur

```
def construitMatriceConfusion(N,k):
    VP,VN,FP,FN=0,0,0,0
    for i in range(N):
        x,y=rd.uniform(-1,1),rd.uniform(-1,1)
        couleurPPV=plus_proches_voisins((x,y), k)
        if est_rouge(x,y) and plus_proches_voisins((x,y), k)=='R':
            VP+=1
        elif est_rouge(x,y) and plus_proches_voisins((x,y), k)=='B':
            FN+=1
        elif plus_proches_voisins((x,y), k)=='B':
            VN+=1
        else:
            FP+=1
```

Les résultats sont rassemblés dans le tableau suivant :

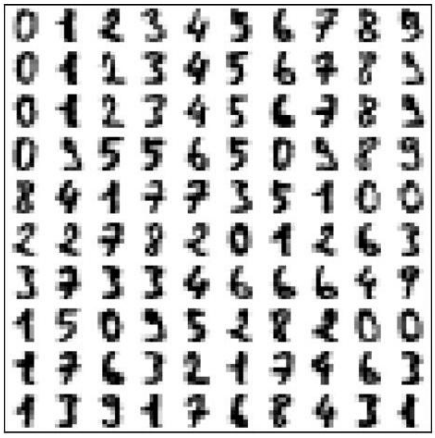
k	2	4	10	100
M_c	$\begin{pmatrix} 27 & 3 \\ 3 & 67 \end{pmatrix}$	$\begin{pmatrix} 25 & 0 \\ 2 & 73 \end{pmatrix}$	$\begin{pmatrix} 32 & 3 \\ 0 & 65 \end{pmatrix}$	$\begin{pmatrix} 25 & 5 \\ 3 & 67 \end{pmatrix}$

L'utilisation de la matrice de confusion permet de déterminer la valeur de k optimale.

3.1.2 Reconnaissance de caractères

Nous allons voir comment utiliser cette approche dans un nouveau problème : la reconnaissance optique de caractères (ou OCR).

Considérons que l'on dispose de 1797 images de 8×8 pixels en niveau de gris représentant des chiffres de 0 à 9 (un extrait de la base de données est présenté ci-contre).



Les données sont stockées dans deux tableaux de type `ndarray` :

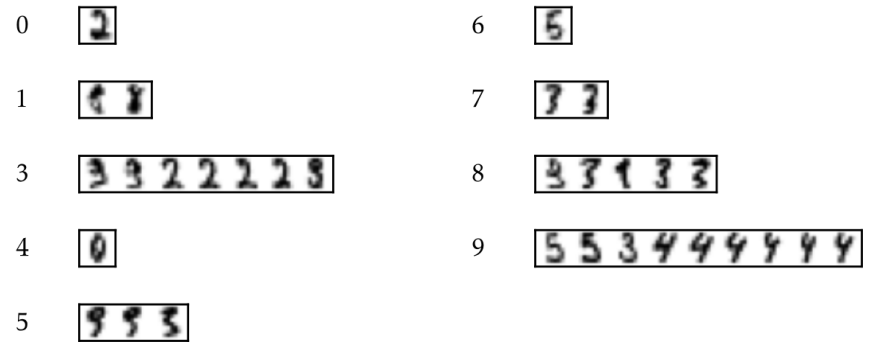
- X est une matrice 1797×64 ; telle que $X[k]$ est un vecteur de $\mathbb{R}^{64}$ qui représente l'image digitalisée d'un chiffre ;
- Y est un vecteur de $\mathbb{R}^{1797}$ qui identifie l'entier compris entre 0 et 9 représenté par $X[k]$.

L'approche est sensiblement la même que dans la partie précédente : une partie des données sert à l'apprentissage, le reste sert à son évaluation. La relation d'ordre entre deux images est définie comme la norme euclidienne associée à la comparaison des vecteurs. La fonction `plus_proches_voisins` renvoie la ou les étiquettes les plus présentes parmi les k voisins (sous la forme d'une liste).

Appliqué à une donnée de test, la fonction `plus_proches_voisins` peut renvoyer : la bonne valeur, une valeur fausse (la machine a tort) ou plusieurs valeurs (la machine est indécise).

L'expérience montre que pour $k = 3$, sur les 897 valeurs de test : la machine reconnaît le bon caractère 858 fois (soit un taux de réussite de plus de 95%) ; la machine se trompe 31 fois ; la machine est indécise 8 fois.

Des erreurs commises par la machine sont présentées ci-dessous.



La méthode présentée est simpliste, mais peut être améliorée de plusieurs façons, par exemple :

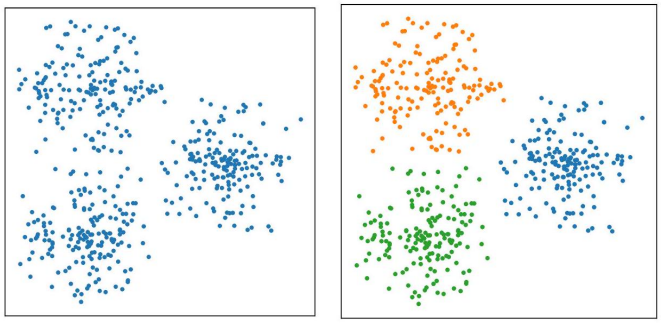
- au lieu de considérer les k plus proches voisins, on peut considérer tous les voisins contenus dans une boule de rayon ϵ centrée autour de l'objet à classer ;
- on peut pondérer la contribution de chaque voisin par un coefficient inversement proportionnel à la distance à l'objet à classer ;
- d'autres distances que la distance euclidienne peuvent être définies.

3.2 Apprentissage non supervisé

3.2.1 Partitionnement des données

Passons enfin à un problème plus complexe : comment comprendre la structure des données sans aider la machine en lui donnant des données d'apprentissage étiquetées ?
L'idée est que la machine regroupe les données proches au sein d'une même classe dont on souhaite qu'elles soient pertinentes.
Prenons l'exemple ci-dessous dans lequel nous voyons trois classes de points. Pour aider la machine, on lui impose le nombre k de classes $C_1, \dots, C_k$. On définit ensuite μ_j le barycentre de C_j et m_j son moment d'inertie :

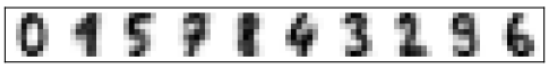
$$\mu_j = \frac{1}{\operatorname{card} C_j} \sum_{x \in C_j} x \quad \text{et} \quad m_j = \sum_{x \in C_j} \left|0x - \mu_j\right| 0^2$$



L'algorithme des k-moyennes Le calcul de la classification optimale, c'est-à-dire qui minimise la somme des moments d'inertie, est un problème très coûteux en temps, aussi emploie-t-on plutôt un algorithme glouton qui permet d'obtenir en un temps raisonnable une solution pas trop mauvaise.
L'algorithme des k moyennes consiste à réaliser la succession d'opérations suivantes :

1. on choisit aléatoirement k centres $\mu_1, \dots, \mu_k$;
2. chacun des points du nuage est associé au centre μ_j le plus proche dans la classe C_j ;
3. les barycentres $\mu_1, \dots, \mu_k$ sont recalculé au sein des classes constituées ;

Les étapes 2 et 3 sont répétées jusqu'à ce que l'algorithme converge : les centres $\mu_1, \dots, \mu_k$ ne se déplacent plus et les classes $C_1, \dots, C_k$ sont stabilisées[d].
En appliquant cette approche au problème précédent de la reconnaissance optique de caractères, on obtient les barycentres suivants pour les 10 classes imposées



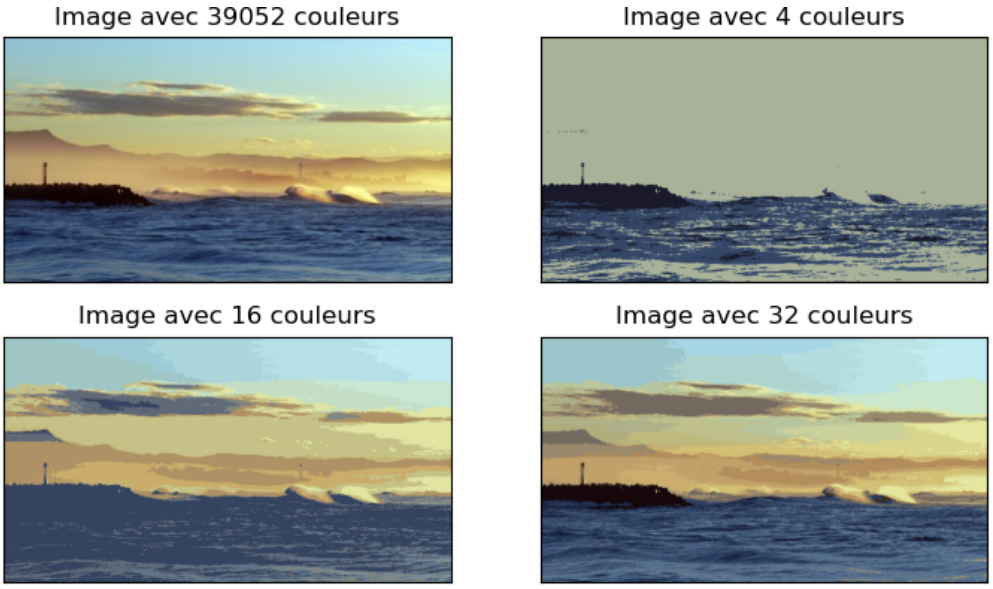
Le résultat des affectations montre quand même que des erreurs importantes sont commises :

classe	0	1	5	7	8	4	3	2	9	6
réussite	99%	60%	91%	85%	45%	98%	87%	85%	56%	97%

d. Attention, cet algorithme converge vers une configuration pour laquelle la somme des moments d'inertie est un minimum local, mais pas forcément le minimum global.

3.2.2 Application à la compression de données

On souhaite appliquer ce qui vient d'être vu à la compression de données : en choisissant intelligemment le nombre de couleurs d'une image, on peut en diminuer spectaculairement le nombre sans nuire excessivement à sa qualité.
Da façon générale, une image est une collection de pixels, vus comme un triplet d'entiers compris entre 0 et 255. Plus formellement, une image de taille $x \times y$ pixels est représentée par une matrice $M \in \mathcal{M}_{x,y}(K^3)$ avec $K = [\![0, 255]\!]$. Le nombre de couleurs théoriquement accessible est donc 16 777 216, soit 3 octets utilisés par pixel ; le fait de réduire le nombre de couleurs représentées à 16 permet de réduire le poids de l'image d'un facteur 2^{20}.



Application

ITC5.4 : Réduction du nombre de couleurs d'une image

Q.1. Écrire une fonction `trouve_couleur(img)` qui retourne le nombre de couleurs contenues dans une image (on rappelle que `img.shape` renvoie le triplet (x,y,3)).

Q.2. Rédiger une fonction `dist(p,q)` qui prend pour arguments deux pixels `p` et `q` (représentés par deux vecteurs de $\mathbb{R}^3$) et renvoie la distance euclidienne entre ces deux vecteurs.

Q.3. Rédiger une fonction `initialise(img,k)` qui prend pour arguments une image `img` un entier `k` et renvoie un tableau de `k` cases, chacune d'elles contenant un pixel tiré au hasard dans l'image.

Q.4. Écrire une fonction `barycentre(img, s)` qui prend pour arguments une image `img` et un ensemble `s` de coordonnées (x, y) et renvoie un pixel (une liste de 3 entiers) égal au barycentre des pixels de l'image dont les coordonnées appartiennent à `s`.

Q.5. Rédiger une fonction `plus_proche_pixel(p,mu)` qui prend pour arguments un pixel p et une liste de pixels `mu` $= [\mu_0, \dots, \mu_{k-1}]$ et qui renvoie l'indice j qui minimise la distance $\left|0p - \mu_j\right| 0$.

Q.6. Implémenter alors une fonction `k_moyennes(img,k)` qui prend pour arguments une image `img` et un entier `k` et qui renvoie un couple `(liste_pixels_couleur,couleurs)` où `liste_pixels_couleur` est un tableau de longueur `k` tel que chacun de ses éléments est une liste de coordonnées des pixels obtenues par l'algorithme des k-moyennes, et `couleurs` est la liste des pixels utilisés dans l'image finale.

Q.7. Proposer enfin une fonction `reduire(img,k)` qui prend pour argument une image et renvoie une nouvelle image dans laquelle seules `k` couleurs sont utilisées.