Informatique de tronc commun deuxième année

Julien Reichert

2024/2025

Ces documents de cours suivent le nouveau programme d'informatique de tronc commun, introduit en deuxième année à la rentrée 2022.

Il est formé de trois chapitres, le premier étant une réécriture adaptée d'un chapitre de première année du programme antérieur et traitant des bases de données, le deuxième reprenant des éléments de l'ancien cours d'option sur la programmation dynamique, en plus d'une explication approfondie de la structure de dictionnaire déjà utilisée en première année, le troisième étant totalement neuf, par l'introduction de notions d'intelligence artificielle et de théorie des jeux en classes préparatoires.

En raison de la formation initiale de l'auteur de ces lignes, la section sur la théorie des jeux sera enrichie d'éléments non attendus au programme. Ceux-ci seront signalés comme dans les notes de cours de première année par leur couleur magenta, qu'on pourra retrouver sporadiquement ailleurs pour des remarques pour la culture mais non exigibles.

Puisque l'enseignement de deuxième année ne distingue plus forcément des séances de cours, de TD et de TP, les travaux pratiques proposés consisteront certes principalement en des exercices de programmation ou d'application du cours, mais comporteront également des exercices non moins importants et nécessitant de la rédaction. Leur quantité sera par ailleurs réduite et un TP recouvrira plutôt deux séances de cours, ou bien sera à travailler chez soi.

Le document lui-même n'a pas changé par rapport à l'année précédente, à la correction de coquilles près.

Table des matières

Première partie

Cours

Chapitre 1

Bases de données

Un des fondements et motivations de l'étude des bases de données relationnelles et, du point de vue théorique, de l'algèbre relationnelle est l'existence de problèmes algorithmiques plus ou moins concrets pour lesquels les structures de données étudiées auparavant ne sont pas aussi efficaces que l'on peut l'espérer.

Avant tout, une question en lien avec notre sujet : on souhaite trier un ensemble de personnes. Selon quels critères ?

Alors que les listes d'entiers se triaient dans presque tous les cas suivant l'ordre canonique (croissant ou décroissant), il n'y a pas d'ordre de référence pour les humains. C'est bien évidemment parce que chacun a ses particularités, qu'on représentera pour des objets quelconques par des attributs.

De tels attributs permettent un regroupement (on peut par exemple mettre ensemble toutes les personnes ayant des lunettes / des lentilles / subi une chirurgie au laser / etc.) voire un classement (tri par la taille, l'âge, etc.).

En pratique, le classement est toujours imaginable dans la mesure où on peut associer à une valeur d'un attribut une chaîne de caractères qui la décrit et considérer par exemple l'ordre lexicographique.

Le problème des structures de données que l'on connaît, c'est que si on tente de présenter des données regroupées selon un certain attribut, faire un regroupement selon un autre sera très complexe.

Imaginons qu'on mette les étudiants de SPE d'un lycée dans un tableau, ce tableau contenant un sous-tableau par classe et dans ce sous-tableau les élèves sont regroupés par classe l'année précédente.

Récupérer la liste des anciens PCSI 2 nécessite un parcours de chaque sous-tableau pour accéder au sous-sous-tableau de la PCSI 2. Bien entendu, à ce moment-là, récupérer la liste des PSI est rapide car elle est déjà fournie (on peut avoir besoin d'aplatir les sous-sous-tableaux suivant l'énoncé). L'idée est cependant de garder la même simplicité pour tous les problèmes de ce genre.

Ainsi, nous allons étudier dans ce chapitre les moyens d'exécuter des opérations (ou requêtes, sur un modèle concret de bases de données) sur des structures d'objets pourvus d'attributs.

1.1 Algèbre relationnelle

Cette section a été retirée du nouveau programme. Elle est laissée dans les notes de cours mais ne sera pas traitée en classe.

L'*algèbre relationnelle* est un modèle théorique développé dans les années 1970, en grand lien avec la théorie des ensembles. Nous allons nous appuyer sur cette dernière pour définir notre vocabulaire, et dans la section suivante un parallèle sera fait entre chaque opération présentée ici et chaque requête introduite dans le modèle concret des bases de données.

Soit un objet caractérisé par un certain nombre d'*attributs*. On va considérer cet objet comme le n-uplet formé par les valeurs de ces attributs dans un ordre fixé pour chaque objet.

Il est possible que des doublons soient alors créés dans un ensemble d'objets, ce qu'on autorise dès lors qu'on ne met pas en place une sorte d'identifiant sur lequel nous reviendrons.

Une *relation* est un ensemble fini d'objets (appelés ici *valeurs* ou *enregistrements*, dont le nombre est le *cardinal* de la relation, noté par un croisillon) ainsi définis. On précisera habituellement la structure de cette relation (voir deux définitions plus loin), tout comme on précise en théorie des ensembles sur quels ensembles une relation binaire est construite. L'homonymie n'est par ailleurs pas due au hasard.

Le *domaine* d'un attribut est l'ensemble de ses valeurs possibles (et non pas nécessairement l'ensemble des valeurs prises, tout comme une fonction réelle peut ne pas être surjective quand bien même son ensemble d'arrivée est défini comme étant $\mathbb{R}$ conventionnellement).

Un *schéma relationnel* est la précision de la structure d'une relation, sous la forme de n-uplet formé des couples (attribut, domaine) dans l'ordre (on pourra omettre le domaine dans certains cas).

Afin de permettre une représentation efficace d'une relation, on utilisera simplement un tableau (au sens habituel du terme), d'où le nom de *table* que l'on verra dans la section sur les bases de données.

Les opérations sur les relations, ou *opérateurs relationnels*, sont essentiellement des recherches d'enregistrements ou se fondent dessus en effectuant un traitement sur les résultats.

Produire un même résultat peut se faire de plusieurs façons différentes, mais pas nécessairement avec la même complexité, que ce soit du point de vue algorithmique ou de celui de la taille de la formule mathématique correspondante (les deux sont souvent en relation).

Soient deux relations R_1 et R_2 de même schéma. L'*union* (ou réunion) de R_1 et R_2, notée $R_1 \cup R_2$, est l'ensemble des enregistrements de R_1 et de ceux de R_2. Attention, un élément apparaissant au moins une fois dans R_1 et au moins une fois dans R_2 apparaîtra exactement une fois dans $R_1 \cup R_2$, d'après la convention classique.

Par analogie, on définit l'*intersection* de R_1 et R_2, notée $R_1 \cap R_2$, comme l'ensemble, lui aussi a priori sans doublon, des enregistrements communs à R_1 et R_2.

Enfin, la *différence* de R_1 et R_2, notée $R_1 - R_2$, est l'ensemble des enregistrements de R_1 qui ne sont pas dans R_2.

Dans les trois cas, le schéma de la relation « composée » est le schéma commun aux deux relations. Ceci étant, si on n'impose pas que les schémas soient communs mais seulement *compatibles* (c'est-à-dire les noms d'attributs peuvent ne pas être les mêmes, mais les domaines doivent être identiques, et à plus forte raison le nombre d'attributs doit être identique pour les deux relations).

On note que certaines propriétés de la théorie des ensembles ne sont plus forcément vraies en algèbre relationnelle : en raison de la présence possible de doublons, il n'est pas exclu que $\#(R_1 \cup R_2) < \max(\#R_1, \#R_2)$ et que $\#(R_1 \cap R_2) > \min(\#R_1, \#R_2)$ (si la convention est différente et que chaque égalité entre un élément de R_1 et un élément de R_2 donne un élément dans l'intersection). De même, la formule $\#(R_1 \cup R_2) = \#R_1 + \#R_2 - \#(R_1 \cap R_2)$ est potentiellement fausse.

Soient une relation R de schéma S et S' un sous-ensemble de S. La *projection* de R selon S', notée $\pi_{S'}(R)$, est la relation de schéma S' dont les enregistrements ne sont alors définis que par les valeurs pour les attributs dans S'. Ici, des doublons peuvent être créés, et le cardinal de la projection est identique au cardinal de R.

Soient R_1 et R_2 deux relations de schémas respectifs S_1 et S_2. Le *produit cartésien* de R_1 et R_2, noté $R_1 \times R_2$, est l'ensemble des couples d'enregistrements (e_1, e_2), où $e_1 \in R_1$ et $e_2 \in R_2$. Le schéma de ce produit cartésien est la réunion avec doublons de S_1 et de S_2, et son cardinal est le produit des cardinaux des relations.

Si de plus $S_2 \subseteq S_1$, on peut définir la *division cartésienne* de R_1 par R_2, notée $R_1 \div R_2$, par la relation, de schéma $S_1 \setminus S_2$, dont les enregistrements e sont tels que pour tout enregistrement e_2 de R_2, la fusion de e et de e_2 donne un enregistrement de R_1. On note que $(R_1 \times R_2) \div R_2 = R_2$, mais $(R_1 \div R_2) \times R_2 \subseteq R_1$ sans garantie d'égalité (et il faut aussi que $R_1 \div R_2$ existe, pour commencer). [1]

Soient une relation R de schéma S et A' un attribut de domaine D commun avec un attribut A dans S. Le *renommage* de A en A', noté $\rho_{A \to A'}(R)$, donne une nouvelle relation dont le schéma est S'. On peut imaginer un renommage multiple, par ailleurs. Puisque le nom de l'attribut n'a pas vraiment d'influence sur les enregistrements, on justifie ici le fait de nécessiter des schémas compatibles pour l'union, l'intersection et la différence, plutôt que nécessairement des schémas identiques.

Soient S un schéma relationnel et R une relation de schéma S. Étant donné un *prédicat* P, c'est-à-dire une fonction dépendant d'un certain nombre d'arguments et retournant un booléen, d'argument un enregistrement de R, la *sélection* de R selon P est l'ensemble des enregistrements e dans R tels que $P(e)$ soit vrai, qu'on écrit simplement $\sigma_P(R) = \{e \in R \mid P(e)\}$. Un prédicat de base est la comparaison de la valeur pour un certain attribut avec un certain élément de son domaine.

1. On peut faire un parallèle avec la division euclidienne. La relation $R_1 \div R_2$ est la plus grande relation R_q pour laquelle il existe une relation R_r telle que la réunion de R_r et de $R_q \times R_2$ soit R_1.

Remarque : Le rapport étroit entre la logique et la théorie des ensembles apparaît dans les propriétés suivantes.

— $\sigma_{P \text{ ET } Q}(R) = \sigma_P(R) \cap \sigma_Q(R)$ (privé des doublons) ;
— $\sigma_{P \text{ OU } Q}(R) = \sigma_P(R) \cup \sigma_Q(R)$ (idem) ;
— $\sigma_{\text{NON } P}(R) = R - \sigma_P(R)$ (idem) ;

La *jointure symétrique*[2] de R_1 et R_2 selon l'égalité d'attributs $A_1 = A_2$ (où chaque A_i est dans le schéma de R_i), notée $R_1 \bowtie_{A_1=A_2} R_2$, est l'ensemble des enregistrements du produit cartésien $R_1 \times R_2$ dont les valeurs pour les attributs A_1 et A_2 sont égales.

On notera qu'il n'est pas nécessaire que le domaine des attributs soient les mêmes, et que la jointure sera vide si l'intersection des domaines est vide, par exemple.

Par ailleurs, puisqu'on introduit ici une redondance, on peut se permettre de ne retenir qu'un attribut entre A_1 et A_2 dans le produit cartésien. C'est ce que font certains opérateurs de jointures dans le langage enseigné à la section suivante.

Ainsi, $R_1 \bowtie_{A_1=A_2} R_2 = \sigma_{A_1=A_2}(R_1 \times R_2)$, avec éventuellement l'application d'une projection retirant A_1 ou A_2 au résultat.

Restent les fonctions d'agrégation, que nous verrons plus particulièrement dans une section spécifique.

Toute l'algèbre relationnelle repose sur la composition d'opérations présentées ci-avant.

1.2 Bases de données relationnelles

Une grande partie du vocabulaire défini à la section précédente est à connaître du point de vue des bases de données. Il n'est pas rappelé ici mais présenté de manière adaptée en cours.

Une base de données est une implémentation d'un ensemble de relations (appelées ici *tables*) telles que décrites précédemment. Le stockage en lui-même de la base de données dépend du système de gestion, l'une des possibilités étant du texte clair, ou de façon plus pratique un tableur.

2. les autres jointures n'étant toujours pas au programme

Les opérations de l'algèbre relationnelle sont bien entendu implémentées elles aussi (du moins les principales), ainsi que des opérations de modification, telles que l'ajout d'un enregistrement (appelé ici *entrée*), sa suppression ou sa modification, de même que la modification de la structure d'une table (ce qui modifie au passage les entrées), voire l'ajout ou la suppression de tables, entre autres.

1.2.1 Clés

Dans une base de données, il peut être utile de disposer de contraintes sur les entrées assurant qu'on puisse identifier chacune par un index, notamment en vue de faire des relations entre tables, et surtout afin de garantir l'unicité d'une valeur pour un attribut, ou éventuellement seulement d'un n-uplet.

C'est le concept de *clé*. Une clé (en SQL, cela se repère par un champ `UNIQUE` dans la structure) est un sous-ensemble d'attributs tel que l'insertion ou la modification d'une entrée ne soit possible que si le n-uplet de valeurs pour ces attributs ne se retrouve dans aucun autre enregistrement.

Si la table n'a pas de doublon, son schéma est en pratique une clé, mais bien entendu on préfère que le nombre d'attributs soit aussi restreint que possible.

En pratique, on ajoute souvent à la structure d'une table un attribut spécial, dont le nom fait référence à un identifiant, prévu pour être une clé, on dispose même d'une fonctionnalité dite `AUTO INCREMENT`, grâce à laquelle on n'a pas besoin de renseigner la valeur de l'identifiant, car un compteur associé à la table sait quelle sera la prochaine valeur à affecter.

Une *clé primaire* est une clé pour laquelle la recherche est optimisée ; tout ceci se fait au sein de la base de données.

En particulier, rien n'impose qu'une clé primaire se limite à un attribut, et une clé limitée à un attribut n'est pas forcément primaire.

Le concept de *clé étrangère* est en lien avec celui que nous venons de présenter, à ceci près qu'il n'y a pas d'unicité dans la table où une telle clé figure. En fait, une clé étrangère est un attribut correspondant normalement à une clé (souvent la clé primaire, pour profiter de l'optimisation) dans une autre table, là aussi en vue de faire des associations.

Par exemple, si on dispose d'une table contenant une liste d'élèves et d'une table contenant une liste d'épreuves, on peut créer une troisième table avec trois attributs : une clé étrangère correspondant à l'identifiant d'un élève, une autre correspondant à l'identifiant d'un devoir, et finalement la note obtenue.

C'est le contenu de cette troisième table qui peut être étudié afin de faire des calculs, et les résultats seront traités à l'aide des deux premières tables : une fois la moyenne, le maximum, etc. calculés, il s'agit de retrouver à qui ou à quelle épreuve l'information obtenue correspond.

Ces trois tables se retrouveront dans la base de données utilisée dans le premier TD.

1.2.2 Le langage SQL

Introduction

Afin de communiquer avec un serveur de bases de données, un langage spécifique est nécessaire. Ce langage consiste à formuler des requêtes, reprenant les opérations de l'algèbre relationnelle, et à en récupérer les résultats.

Aux requêtes classiques de recherche et de gestion du contenu des tables s'ajoutent des requêtes d'administration, moins souvent utilisées (et qu'on préfèrera largement faire faire par un intermédiaire). Ces requêtes d'administration ne sont pas à connaître, pas même les manipulations de données modifiant le contenu de tables.

Un langage quasi incontournable pour exécuter ces requêtes est SQL [3].

En pratique, de nombreux systèmes de gestion de bases de données (ou SGBD) utilisent des surcouches de SQL, avec du sucre syntaxique. On citera les systèmes MySQL, PostgreSQL, Oracle, etc. Quant à SQLite, dont le nom est tout aussi connu [4], il ne s'agit pas d'un SGBD mais d'un moteur de bases de données (qu'il y a dans les SGBD, donc), nécessitant de mettre un peu plus les mains dans le cambouis.

3. pour *Structured Query Language*, qui se traduit par « langage de requête structurée »
4. et qui est parfois intégré à des environnements de développement intégrés pour Python, entre autres

Requêtes

Les requêtes principales concernant les données se regroupent en quatre catégories : insertion, recherche, mise à jour et suppression [5].

L'insertion se contente d'ajouter une entrée dans la table, en renseignant tous les attributs nécessaires (si des attributs manquent, il faut préciser quelles valeurs correspondent à quels attributs pour lever l'ambiguïté, ce qui permet également de renseigner les attributs dans l'ordre que l'on souhaite, heureusement).

Les autres requêtes nécessitent de préciser à l'aide de conditions sur quel(le)s entrée(s) l'opération doit être effectuée (il s'agit de la projection). Ces conditions portent sur les attributs de la table : comparaisons, tests d'égalité, voire recherche de sous-chaîne, etc. À ce titre, il est possible de faire appel à des fonctions prédéfinies dans le langage dont certaines sont données ci-après (fonctions d'agrégation).

La syntaxe est la suivante :

- Insertion : `INSERT INTO <table>(<attributs>) VALUES (<valeurs>)`, la partie renseignant les attributs entre parenthèses étant optionnelle si on les fournit tous ;
- Recherche : `SELECT <attribut>,...,<attribut> FROM <table>`, suivi de `WHERE <condition>` la plupart du temps, avec le joker `*` pour demander tous les attributs, ou la version sans doublon `SELECT DISTINCT` ;
- Mise à jour : `UPDATE <table> SET <attribut>=<valeur>,..., <attribut>=<valeur> WHERE <condition>` ;
- Suppression : `DELETE FROM <table> WHERE <condition>`.

Ordinairement, les attributs sont entourés d'accents graves et les valeurs sont entourées de guillemets ou d'apostrophes quand il s'agit de chaînes de caractères.

En pratique, utiliser systématiquement des guillemets est autorisé quel que soit le type, les conversions vers le bon type se font si besoin comme on l'observe en Python.

La partie commençant par `WHERE` est toujours optionnelle, et si elle est absente on fait la requête sur toutes les entrées. Un point-virgule, séparant deux requêtes consécutives est fréquemment mis en toute fin, mais il n'y est pas nécessaire.

5. Pour ceux qui font l'option informatique, notez le rapprochement avec des structures de données abstraites composées.

Attention, dans une requête de recherche la sélection vient du mot-clé `WHERE`, et `SELECT` induit une projection !

Exemples :

- `INSERT INTO ‘Etudiants‘(‘Nom‘,‘Prenom‘) VALUES ("Martin","Jean");` insère une entrée dans une table nommée `Etudiants` dont tous les champs sauf éventuellement `Nom` et `Prenom` précisés ici sont optionnels ;
- `INSERT INTO ‘Notes‘ VALUES (1,1,12);` insère cette fois une entrée dans une table à exactement trois champs ;
- `SELECT ‘Etudiant‘ FROM Notes WHERE ‘Epreuve‘ = 1 AND ‘Note‘ >= 10` récupère dans la table mentionnée les identifiants des étudiants ayant eu la moyenne à l'épreuve d'identifiant `1` ;
- `SELECT * FROM Etudiants WHERE Prenom REGEXP BINARY "J"` récupère tous les attributs des étudiants dont le prénom contient la lettre capitale `J`[6] ;
- `UPDATE ‘Notes‘ SET ‘Note‘ = ‘Note‘ + 2 WHERE ‘Epreuve‘ = 2` ajoute deux points à toutes les entrées correspondant à la deuxième épreuve ;
- `DELETE FROM Etudiants WHERE Nom = "Martin"` supprime tous les élèves dont le patronyme est `Martin`. Attention aux doublons ![7]

Syntaxe supplémentaire pour la présentation

Le langage SQL dispose de mots-clés impactant la façon de présenter les résultats de la requête, et certains des paramètres ci-après, ne relevant pas de l'algèbre relationnelle, sont parfois propres à un système de gestion de bases de données.

Pour trier les résultats selon un attribut spécifique, on ajoutera (obligatoirement après la condition, si elle existe) `ORDER BY <attribut>` (ordre croissant, `ASC` étant sous-entendu) ou `ORDER BY <attribut> DESC` (ordre décroissant), avec d'autres possibilités dont le notable `ORDER BY RAND()` pour laisser le hasard décider.

Il est également possible de n'afficher qu'un nombre limité de résultats (ce qui se combine bien avec ce qui précède) en écrivant `LIMIT <nombre> OFFSET <premier>`, ce qui donnera les résultats à partir de celui dont l'indice (qu'on comprendra comme un indice en Python, en commençant également à zéro) est précisé en "*offset*" et au nombre précisé en "*limit*" ou en s'arrêtant avant s'il n'y en a pas assez.

6. Le mot-clé `BINARY` permet que la recherche soit sensible à la casse, c'est-à-dire distingue les lettres capitales (majuscules) et en bas de casse (minuscule).

7. phpMyAdmin signale le nombre de lignes affectées par de telles requêtes, ce qui peut être utile pour arranger une catastrophe immédiatement.

Syntaxe supplémentaire pour l'administration

Les requêtes concernant la structure des tables, ou des bases de données en général, sont soumises à la même remarque que les requêtes d'administration (qui gèrent les utilisateurs, entre autres) : dans de nombreux cas, on ne les effectuera pas en les saisissant mais par exemple à l'aide d'une interface.

On mentionnera simplement `TRUNCATE <table>` pour vider une table, `DROP TABLE <table>` pour la supprimer, ainsi que `DROP DATABASE <BDD>` pour supprimer une base de données entière, `RENAME TABLE <table> TO <nom>` pour renommer une table, `ALTER TABLE <table> ORDER BY <attribut>` pour la trier et `ALTER TABLE <table> ADD PRIMARY KEY(<attribut>)` pour y ajouter une clé primaire.

Opérateurs pour les conditions

Les opérateurs à connaître pour l'arithmétique sont `+`, `-`, `x` et `/`. La division produit des flottants comme l'opérateur identique en Python.

Pour produire des booléens par des comparaisons, on retrouve `<`, `<=`, `>` et `>=`, mais le test d'égalité se fait avec `=` (d'autres opérateurs existent, mais sont hors programme) et le test de différence avec `<>`, sachant que `!=` est implémenté dans certaines versions de SQL, mais sera théoriquement compté faux au concours.

Les opérateurs booléens dans les conditions introduites par `WHERE` s'écrivent en toutes lettres en anglais : il s'agit de `AND`, `OR` et `NOT`, auxquelles on ajoute `XOR`, correspondant à soit ... soit ..., mais on regrette l'absence de `NAND` et `NOR`, qui s'obtiennent néanmoins aisément par composition.

Rassembler des tables

Si on souhaite plutôt faire une union, une intersection ou une différence de résultats de requêtes, il s'agit de combiner les requêtes en les séparant par les mots-clés `UNION`, `INTERSECT` ou `EXCEPT`.

La jointure utilise le mot-clé `JOIN` entre deux noms de tables, la condition associée s'exprimant juste après par `ON <égalités entre attributs séparées par AND>`. Le produit cartésien se fait par la mention de tables séparées par le mot-clé `JOIN`, car il s'agit en fait d'une jointure sans condition. Les autres conditions pour une jointure sont hors programme.

De manière analogue à ce qu'on observe en mathématiques, les produits cartésiens peuvent s'enchaîner (les jointures aussi), et un produit cartésien ou plus généralement une jointure peut concerner deux fois la même table.

Il est à noter qu'en faisant une jointure entre une table et elle-même, on crée une ambiguïté entre tous les attributs, ce qui nous amène à introduire le renommage.

Le renommage au sein d'une requête se fait par le mot-clé `AS` après un nom d'attribut. Ceci est notamment utile lorsque la requête fait intervenir des fonctions d'agrégation, où exploiter le résultat nécessite quasiment de lui donner un nom d'attribut pratique.

La même syntaxe permet de renommer une table pour l'exécution d'une requête, ce qui n'a donc aucun impact sur les données enregistrées. Dans les deux cas, écrire le vrai nom puis l'alias sans `AS` est aussi autorisé mais moins esthétique.

Puisque les attributs peuvent provenir de deux tables, les noms d'attributs étant parfois identiques d'une table à l'autre (à éviter sauf dans le cas de clés étrangères), d'éventuels noms d'attributs redondants sont à préfixer par le nom de leur table, il est même envisageable de tout préfixer.

Exemple de préfixage systématique : `SELECT classe.nom FROM classe JOIN edt ON classe.id = edt.classe WHERE edt.salle = 218 AND edt.date = "mardi" AND edt.heure = 8`, pour les noms des étudiants qui ont cours en 218 le mardi à 8 h.

En évitant la jointure : `SELECT nom FROM classe WHERE id_classe IN (SELECT classe FROM edt WHERE id_salle = 218 AND date = "mardi" AND heure = 8)`.

Agrégation

La notion d'*agrégation* réfère au fait qu'on applique une fonction à un nombre a priori indéterminé d'arguments, qui sont des valeurs d'attributs d'un certain nombre d'enregistrements rassemblés ; les enregistrements sur lesquels la fonction est appliquée forment ce qu'on appelle un *agrégat*.

Le regroupement se fait en écrivant `GROUP BY <attribut>`, après quoi il ne reste qu'une entrée pour chaque valeur ou n-uplet de valeurs du ou des attributs en question (agissant comme une clé), entrée qui est l'agrégat en question dont on peut sélectionner le résultat de l'application d'une fonction présentée dans la suite.

Le programme mentionne cinq fonctions d'agrégation à connaître. [8] Il s'agit du comptage (`COUNT`), du minimum (`MIN`), du maximum (`MAX`), de la somme (`SUM`) et de la moyenne (`AVG`).

L'agrégation crée de nouveaux attributs (dont on a dit qu'ils sont renommés pour faciliter leur utilisation). Elle peut alors se composer avec des sélections (ou regroupements) après avoir été appliquée, ce qui n'est pas la même chose que les regroupements ou sélections avant d'appeler la fonction.

Par exemple, si on veut trouver les étudiants qui ont au dessus de 12 en moyenne dans une table dont les attributs sont un identifiant d'étudiant, un identifiant d'épreuve et une note (cf. TD 1), il faut faire un regroupement des entrées avec le même identifiant d'étudiant, un calcul de moyenne puis une sélection dans la table obtenue, ce qui s'écrit par exemple `SELECT Etudiant FROM (SELECT Etudiant, AVG(Note) AS Moyenne FROM notes GROUP BY Etudiant) AS Id WHERE Moyenne > 12`. [9]

Une autre possibilité est de faire un filtrage en aval, dans la mesure où la clause introduite par `WHERE` est un filtrage en amont, donc avant le regroupement, donc sans pouvoir utiliser le résultat de fonctions d'agrégations.

Le mot-clé pour le filtrage en aval est `HAVING`, il concerne des champs agrégés en priorité. Évidemment, on met dans ce cas `HAVING` après `GROUP BY`, et la requête précédente se réécrit donc `SELECT Etudiant FROM notes GROUP BY Etudiant HAVING AVG(Note) > 12`.

Bien entendu, les fonctions d'agrégations peuvent être appelées plus simplement. Par exemple, le cardinal d'une table s'obtient par `SELECT COUNT(*) FROM <table>`, et l'enregistrement maximisant un attribut : `SELECT MAX(<attribut>) FROM <table>`, qu'on peut projeter avec des attributs « compatibles ».

Dans ce cas, tout se passe comme s'il n'y avait qu'un agrégat, composé de tous les éléments de la table, ou de tous les résultats qui ont été récupérés par un filtrage avec `WHERE`.

8. Une liste plus complète est donnée dans le mémento SQL accessible à la page `http://sql.sh/wp-content/uploads/2013/02/mysql-aide-memoire-sql-950px.png`.

9. La partie `AS Id` est imposée par MySQL, car il faut faire un renommage lorsque l'on utilise ce qu'il appelle une « table dérivée ». En SQL standard, ce n'est pas nécessaire, mais pour les travaux pratiques c'est MySQL qui est utilisé.

Notons pour terminer que, pour tous les mots-clés du langage SQL, les lettres capitales sont essentiellement esthétiques et donc non obligatoires, mais pour des raisons de lisibilité et de clarté on les retrouvera toujours.

TD 1 : Une base de données sommaire

Dans ce TD, nous allons travailler sur une base de données présentée en cours en première année, et effectuer des requêtes SQL, en plus de pouvoir procéder pour le travail chez soi à des manipulations à l'aide de la plate-forme PhpMyAdmin. L'adresse de la base de données est `http://phpmyadmin.online.net`, et l'identifiant est db86896.

La base de données peut être récupérée pour une utilisation individuelle sur un serveur personnel [10].

La structure associée est la suivante :

— Table Etudiants, avec les attributs Id (clé primaire avec auto-incrémentation, entier naturel), Classe (chaîne de caractères), Nom (idem) et Prenom (idem). On peut considérer que (Nom, Prénom) est également une clé.
— Table Examens, avec les attributs Id (clé primaire avec auto-incrémentation, entier naturel), Date (chaîne de caractères) et Coeff (entier naturel).
— Table Notes, avec les attributs Etudiant (entier naturel), Examen (entier naturel) et Note (entier naturel). Le couple (Etudiant, Examen) est une clé.

Dans chacun de ces exercices, on se contentera d'écrire une requête correspondante.

Exercice 1 : Déterminer qui a la meilleure note à l'examen numéro 2 parmi les MPSI 2.

Exercice 2 : Déterminer la moyenne du meilleur étudiant.

Exercice 3 : Déterminer combien de MPSI 1 ont sous la moyenne.

Exercice 4 : Déterminer quelle classe a la meilleure moyenne.

Pour les deux exercices suivants, on considère que les étudiants de toutes les classes passent les mêmes examens.

Exercice 5 : Déterminer dans quelle classe est l'étudiant ayant majoré le premier examen.

10. `http://jdreichert.fr/Enseignement/CPGE/PC/bdd_td_1.sql`

Exercice 6 : Déterminer le nombre de majors par classe.

Exercice 7 : Déterminer la classe avec le plus grand pourcentage d'étudiants ayant la moyenne.

TD 2 : Requêtes avancées en SQL

La base de données pour ce TD n'est pas disponible en ligne, privant de la possibilité de tester les requêtes ou donnant l'occasion de faire un exercice initial revenant à créer cette base de données, par exemple avec SQLite sur un IDE de Python au choix le permettant (voir aussi le TP 1 à ce sujet).

Inutile de faire de nombreuses tables pour donner la possibilité de se creuser la tête, une suffit. Cela permet en outre d'aborder les jointures entre une table et elle-même, qui **nécessite un renommage de la table à chaque occurrence, ainsi qu'un préfixage de tous les attributs devenant ici ambigus**.

On considère sur le TD entier une table `Resultats` pouvant matérialiser des épreuves quelconques. Les attributs sont limités à une date `Date` de type chaîne de caractères de taille limitée à 20, un identifiant `Id` et à un score `Score`, tous les deux de type entier. Le couple `(Date, Id)` est une clé primaire.

Exercice 1 : Écrire une requête qui permet de récupérer le score maximum réalisé, la date à laquelle il a été réalisé et l'identifiant de la personne qui l'a réalisé.

Exercice 2 : Écrire une requête qui permet de récupérer le score maximum réalisé à une date notée `d`.

Exercice 3 : Écrire une requête qui permet de déterminer le nombre de personnes ayant fait mieux qu'un score `s` donné, et ce à une date notée `d`.

Exercice 4 : Écrire une requête qui permet de déterminer le nombre de personnes ayant fait mieux que le score d'une personne dont l'identifiant `n` est donné à une date notée `d`.

Exercice 5 : Écrire une requête qui permet de récupérer l'ensemble des couples d'identifiants de personnes telles que le score de la première soit strictement supérieur au score de la seconde, le tout à une date notée `d`.

Exercice 6 : Déduire des exercices qui précèdent une requête qui permet d'obtenir le classement de l'épreuve à la date `d` en attribuant la même place à des égalités (l'ordre d'affichage en cas d'égalités n'importe pas). On comprendra que la place est à inclure dans le résultat de la requête.

Exercice 7 : Même exercice en arrêtant le classement à la moyenne.

Exercice 8 : Même exercice en arrêtant le classement à la médiane (en incluant tout de même les égalités).

Exercice 9 : Écrire une requête qui permet de récupérer les identifiants des personnes ayant participé à toutes les dates existantes.

Le dernier exercice revient à faire la division cartésienne de la table par sa projection selon l'attribut `Date` et à projeter le résultat selon l'attribut `Id`... mais on rappelle qu'il n'y a pas d'opérateur standard de division cartésienne.

Chapitre 2

Dictionnaires et programmation dynamique

Le premier semestre de la première année était l'occasion de se servir de la structure de *dictionnaire* comme un objet magique qu'on pouvait indexer par autre chose que des entiers consécutifs à partir de zéro, facilitant le travail par rapport à l'utilisation de listes.

À ce moment-là, il n'était pas question de parler des principes cachés derrière, et par commodité on pouvait considérer l'accès à une donnée comme immédiat.

Tout ceci sera clarifié dans ce chapitre, en introduisant tout d'abord la structure de données de *table de hachage* qui permet de réaliser concrètement un dictionnaire.

Par la suite, les dictionnaires en Python seront abordés, avec des subtilités propres par rapport à la structure de données abstraite dans le cas général, qui ne précise par nature pas comment indexer par des valeurs arbitraires.

Enfin, la programmation dynamique est présentée en fin de chapitre, en tant que paradigme de programmation fondamental. Déplacée depuis la première année de l'option informatique dans le nouveau programme, elle est désormais associée en tronc commun à l'utilisation de dictionnaires afin de gérer la mise en mémoire des résultats intermédiaires, qui est son principe fondamental, de manière plus pertinente.

2.1 Tables de hachage

2.1.1 Introduction

Commençons par motiver l'introduction de la structure étudiée dans cette section. Il s'agit de se poser la question de la façon d'organiser les éléments à mettre dans un dictionnaire, notamment de la façon de placer les clés afin de les retrouver efficacement.

Une simple liste redimensionnable, sans ordre sur les clés, forcerait un parcours potentiellement exhaustif pour toute opération, qu'il s'agisse de consulter, de modifier, de retirer ou d'ajouter une clé (dans ce dernier cas en raison de l'interdiction que deux clés identiques coexistent), pour localiser la clé en question. Ce choix aurait pu être retenu en première année, mais il est désormais exclu.

De manière plus élaborée, une liste triée en fonction des clés (en admettant que celles-ci soient prises dans un ensemble totalement ordonné) reste peu efficace pour les retraits et ajouts de clés, en raison du coût des décalages, mais permet au moins un accès en temps logarithmique grâce à l'algorithme de dichotomie.

Pour pouvoir faire des insertions sans décalages, le cours d'option informatique proposait dans l'ancien programme une implémentation des dictionnaires par une structure arborescente ayant une propriété sur l'ordre des éléments. Cette structure d'*arbre binaire de recherche* ne sera pas enseignée dans ce cours.

2.1.2 Définition

Aussi surprenant que cela puisse paraître, la structure employée sera cependant constituée d'une liste redimensionnable où les clés ne sont a priori pas dans l'ordre naturel. La subtilité est qu'il y a tout de même une information cruciale pour localiser celles-ci, par un calcul élémentaire.

Définition

Une table de hachage est la donnée d'une taille `n` *(usuellement un nombre premier ou une puissance de deux, donnant lieu à deux méthodes de hachage différentes), d'une liste* `tab` *de cette taille et d'une fonction* `h` *prenant en argument n'importe quel objet pouvant être considéré comme une clé et retournant un nombre entre* `0` *inclus et* `n` *exclu.*

Les éléments de la liste `tab` sont des couples `(clef, valeur)` placés à l'indice `h(clef)` et stockés d'une certaine manière, dans la mesure où `h` n'a aucune chance d'être injective et deux clés différentes peuvent avoir la même image. On appelle ce phénomène une *collision*, et plusieurs méthodes permettent de gérer les collisions.

2.1.3 Fonctions de hachage

Comme annoncé, la fonction de hachage doit prendre en argument une clé, qui peut être un entier, un flottant, une chaîne de caractères, un n-uplet de ces derniers...

Sans entrer dans le détail de la manière dont ces fonctions sont construites dans la pratique, et avec un aperçu dans la section suivante de ce que Python fait, on pourra imaginer que pour les entiers on fait simplement un résidu modulo `n`, et que pour les autres types on commence par transformer en entier avant de prendre ce résidu.

Pour les flottants, une idée revient à considérer ceux qui sont égaux à un entier comme l'entier lui-même, et de prendre la partie entière des autres multipliés par une puissance de deux assez grande pour séparer deux flottants proches (par exemple 2^{26} ou 2^{52}, suggérés par la taille de la mantisse avec 64 bits). À noter cependant que les flottants ne sont pas de bonnes clés pour une table de hachage, en raison du risque que deux flottants devant mathématiquement être égaux soient considérés comme différents par l'ordinateur.

Pour les chaînes de caractères, la position de chaque caractère dans la table de caractères utilisée permet d'obtenir une séquence d'entiers, pouvant donner lieu à une évaluation de fonction polynomiale dont les coefficients sont les positions en question, les degrés sont les indices (en mettant au choix le degré 0 au premier ou au dernier caractère) et la valeur pour laquelle on évalue la fonction est usuellement la taille de la table de caractères ou 1 (ce qui veut alors dire qu'on additionne les positions).

Si par hasard on sait qu'on ne met que des lettres non accentuées et minuscules (ou si on veut ignorer la casse), la table de caractères peut être remplacée par l'alphabet, pour une évaluation en 26. On peut même envisager une conversion directe depuis la base 36 si des chiffres peuvent coexister avec les lettres.

2.1.4 Gestion des collisions

Intéressons-nous maintenant à la façon dont on organise les éléments lorsque deux clés insérées dans une table de hachage ont la même image par la fonction de hachage.

La méthode la plus naturelle est de considérer que la liste `tab` est en fait composée de listes à chaque indice, ces listes contenant tous les couples `(clef, valeur)` partageant leur image par la fonction `h`.

On parle alors de *chaînage*, en référence à la structure de liste chaînée utilisée habituellement dans ce cas. Cette structure n'est pas au programme mais présente des similarités avec la pile, vue en première année. Par ailleurs, il n'est pas nécessaire d'utiliser des listes, et d'autres structures sont imaginables, sachant que le but reste de gérer les collisions quand il y en a, mais d'essayer de les éviter au maximum.

En pratique, la probabilité d'existence d'au moins une collision dépasse $\frac{1}{2}$ quand le nombre de clés dans la table est approximativement $\sqrt{n}$. Pour une formule plus précise que cette approximation, on pourra se renseigner sur le paradoxe de Von Mises, également appelé paradoxe des anniversaires, élément de culture mathématique fréquemment vulgarisé.

Ainsi, une table de grande taille avec peu de clés aura normalement très peu de collisions, et même beaucoup de places entre deux indices utilisés consécutifs, ce qui suggère une autre méthode, appelée *adressage ouvert*. Il s'agit de se décaler (unité par unité dans le cas simple, mais on peut faire autrement) de l'indice donné par la fonction `h` jusqu'à trouver un indice libre, et donc tester tous les indices rencontrés lorsqu'une clé est recherchée, avec la possibilité de perdre beaucoup de temps, voire de renoncer alors que la clé est présente, notamment si la table est densément remplie.

Quoi qu'il en soit, c'est en raison de l'existence possible de collisions qu'on ne peut pas dire en toute rigueur que l'accès à un élément d'une table de hachage est en temps $\mathcal{O}(1)$.

2.1.5 Tables de hachage dynamiques

Au fur et à mesure du remplissage d'une table de hachage et donc de l'apparition de collisions, il peut devenir pertinent de réorganiser tous les éléments dans une table plus grande, soit en prenant un autre nombre premier soit en multipliant la taille par une puissance de deux suffisante, suivant la nature de la taille précédente.

Une nouvelle fonction doit alors être mise en œuvre, car adaptée à la nouvelle taille, ce qui retirera des collisions existantes (et pourra en ajouter quelques unes dans le cas où la taille est un autre nombre premier) mais surtout nécessite une opération pour chaque élément déjà existant dans la table.

Pour aller plus loin, ce problème se pose pour des structures de taille modifiable, quand on alloue une zone mémoire contigue pour la stocker, et qu'ajouter un élément forcerait à empiéter sur une zone non libre.

Tout recopier dans une zone juste suffisante fait courir le risque que le prochain ajout force à recommencer, et mieux vaut réserver une zone de taille double à chaque fois, pour délayer d'autant le prochain besoin de réorganisation.

La notion de *coût amorti*, hors-programme, intervient pour justifier qu'on n'ait pas à se soucier d'un tel événement au moment d'évaluer le temps mis par des ajouts successifs d'un élément à une liste dans de telles implémentations.

Une table de hachage dynamique, au contraire des tables de hachages dites statiques, sera alors paramétrée pour augmenter sa taille le cas échéant, avec la fonction de hachage pour chaque taille déjà prête. Aucun détail d'implémentation de cette structure n'est à connaître.

2.1.6 Limitations

Dans la mesure où il est nécessaire de ne pas perdre des informations mises dans une table de hachage, la localisation des clés insérées ne doit pas changer de la moindre façon possible.

On exclut alors :
- Que la fonction change en cours de route (sauf cas dynamique, mentionné ci-avant).
- Que les éléments soient déplacés (même remarque).
- Que l'image des éléments par la fonction de hachage ne change par un autre moyen.

Ce dernier cas semble perturbant au regard des deux premiers mentionnés, mais il y a une subtilité.

Supposons qu'on utilise une structure mutable comme clé, par exemple une liste de nombres en Python.

En imaginant que les valeurs dans la liste sont utilisées pour calculer l'image de la liste elle-même par la fonction de hachage, comme on l'aurait fait avec un n-uplet contenant les mêmes éléments.

Cependant, une fois la liste mutée, son image par la fonction de hachage est amenée à changer, alors que le couple placé dans la table de hachage n'est pas déplacé en conséquence malgré le fait qu'un de ses éléments soit désormais différent et non adapté à l'endroit où il se situe.

Pallier ce problème en mémorisant pour chaque objet mutable créé quels sont les tables de hachage dans lesquelles il est utilisé comme clé n'est évidemment pas une option à envisager !

2.2 Dictionnaires en Python

Le type `dict` de Python est une implémentation par une table de hachage de la structure de données abstraite de dictionnaire. Derrière cette phrase se cachent plusieurs niveaux de compréhension des notions associées aux structures de données.

Voyons ce que la théorie de la section précédente donne dans la pratique.

2.2.1 Propriétés

Un dictionnaire en Python est un objet mutable, indexable par n'importe quel objet appelé clé, pourvu qu'il soit quant à lui immutable.

On le crée par une affectation `dico = {}` (dictionnaire vide) ou `dico = {cle1 : valeur1, cle2 : valeur2}` (initialisation, avec autant d'éléments qu'on le souhaite). Les ajouts peuvent se faire par la suite avec `dico[cle3] = valeur3`, écrasant un éventuel doublon. **En particulier, la méthode `append` ne fonctionnera pas !**

La fonction `dict` permet aussi d'initialiser un dictionnaire à condition de bien respecter sa syntaxe en lui donnant en argument un objet itérable formé d'objets décomposables en une clé et une valeur.

Il est possible d'utiliser la fonction `len` pour obtenir le nombre d'éléments du dictionnaire, et de parcourir le dictionnaire par une boucle `for`, sachant qu'écrire `for element in dico` donnera à `element` les valeurs successives des **clés** du dictionnaire, et les valeurs associées s'obtiendront à chaque tour de boucle par `dico[element]`, qui est même en-dehors d'une boucle la syntaxe pour accéder à un élément d'un dictionnaire.

2.2.2 Manipulations

Pour retirer une clé d'un dictionnaire, la syntaxe est la même que pour effacer un élément d'une liste (cette dernière opération effectuant un décalage de tous les indices au-delà, rappelons-le), à savoir `del dico[element]`, où `element` est la clé à retirer, ou `dict.pop(element)` qui retourne la valeur associée à la clé retirée. Sans argument, il est également possible de retirer le dernier élément ajouté à un dictionnaire (à condition que la version de Python soit assez récente), par la méthode `popitem`, renvoyant un couple `(clef, valeur)`.

L'accès à toutes les valeurs du dictionnaire peut se faire par l'appel à la méthode `values` sur ce dictionnaire, sachant que `dict.values()` n'est pas une liste mais un objet d'un type particulier. **Attention à ne pas oublier les parenthèses, comme pour les méthodes à suivre !** Cet objet n'en est pas moins itérable, et donc on peut le convertir en liste si on souhaite accéder à un élément par son indice.

L'ordre des éléments dans `dict.values()` est le même que l'ordre des clés dans son pendant `dict.keys()`, qui est également celui dans lequel les éléments apparaissent lors du parcours.

Pour obtenir directement les couples, il est plus rapide d'utiliser `dict.items()` que de combiner les deux objets mentionnés précédemment.

La vérification de la présence d'une clé se fait par l'opérateur `in`, qu'on considèrera comme donnant une réponse immédiate. Pour les valeurs, on peut tester l'appartenance à `dict.values()`, qui ne sera cependant pas en temps constant.

Dans les versions récentes de Python (depuis la version 3.7), les éléments d'un dictionnaire sont ordonnés, en fonction de la chronologie de leurs ajouts. C'est un gros avantage par rapport aux versions précédentes où l'ordre était imprévisible et dépendaient plutôt de la fonction de hachage associée.

Une remarque en passant : il est fortement recommandé de ne pas muter une liste pendant son parcours. C'est également vrai sur les dictionnaires, et Python se manifeste par une erreur dès que l'on tente d'ajouter ou de retirer des clés à un dictionnaire qu'on parcourt. En revanche, modifier les valeurs associées n'est pas du tout problématique, par exemple un dictionnaire contenant des nombres peut subir la multiplication par deux de chaque valeur qui y figure. En quelque sorte, c'est comme muter les valeurs dans une liste par une boucle sur ses indices.

2.2.3 La fonction `hash`

La fonction `hash` donne la valeur de hachage de n'importe quel objet compatible dans l'optique de s'en servir comme clé dans un dictionnaire. Au moment de la création d'un dictionnaire, la taille de la liste d'appui sera calculée afin de déterminer modulo combien cette valeur de hachage sera calculée, car il s'agit a priori d'un entier signé sur 64 bits.

Cette fonction a quelques propriétés remarquables : pour les entiers de $-2^{61} + 2$ à $2^{61} - 2$ la fonction retourne l'entier lui-même, pour les flottants égaux à un entier dans cette zone c'est la même chose, mais il faut savoir qu'aucun flottant n'est égal à $2^{61} - 2$ pour des raisons de précision.

Par ailleurs, relancer Python change l'image qu'elle donne pour des chaînes de caractères, au contraire des types numériques. Il s'agit en fait d'une sécurité de Python vis-à-vis d'attaques cryptographiques visant à engendrer un objet de même image qu'un objet ciblé. Comme le paradoxe mentionné au moment de parler des collisions, on parle ici d'attaque des anniversaires.

Tout objet immutable est a priori compatible avec la fonction `hash`, avec la réciproque qu'on peut considérer comme vraie. Pour la rigueur, c'est la compatibilité avec la fonction `hash`, donnant l'attribut `hashable`, qui fait qu'un objet peut être une clé de dictionnaire ou non, plutôt que l'immutabilité.

2.3 Programmation dynamique

Avant de présenter la programmation dynamique, intéressons-nous au contexte qui justifiera son emploi, en montrant sur des exemples comment optimiser un programme, avant de donner la définition formelle qui en deviendra bien plus compréhensible.

2.3.1 Problématique

L'explication technique sur les fonctions récursives a amené à présenter la notion de pile d'appels, et de mise en attente des calculs.

Imaginons une fonction récursive dont chaque appel peut en déclencher plusieurs dans les cas récursifs.

Même si la pile d'appels est amenée à se vider par moments à chaque cas de base atteint, pour n'occuper qu'un espace linéaire dans la plupart des algorithmes, le nombre total d'appels récursifs peut être bien plus élevé, par exemple exponentiel.

Ce qu'on doit chercher à éviter est l'apparition du même calcul à faire plusieurs fois dans cette pile d'appels.

L'exemple classique est le calcul des termes de la suite de Fibonacci, dont la complexité peut varier de manière spectaculaire suivant l'algorithme employé.

```
def fibomauvais(n):
    assert n >= 0
    if n < 2:
        return n
    else:
        return fibomauvais(n-1) + fibomauvais(n-2)
```

Soit f_n la valeur calculée par `fibomauvais(n)`. On constate pour de petites valeurs que f_4 nécessite de calculer f_3 et f_2, que f_3 nécessite de calculer f_2 et f_1 et que f_2 nécessite de calculer f_1 et f_0, le reste étant des cas de base.

Par conséquent, on a mis deux fois dans la pile d'appels un calcul de f_2 (pour f_3 et f_4), et à chaque fois on a mis dans la pile d'appels un calcul de f_1 et de f_0 ; par ailleurs, on a aussi mis dans la pile d'appels un calcul de f_1 lorsqu'on a demandé la valeur de f_3.

On en déduit que la complexité c_n en nombre d'additions du calcul de f_n est la suivante : $c_n = c_{n-1} + c_{n-2} + 1$, avec $c_1 = c_0 = 0$, et la complexité t_n en nombre d'appels récursifs est la suivante : $t_n = t_{n-1} + t_{n-2} + 2$, avec aussi $t_1 = t_0 = 0$, soit dans les deux cas un nombre exponentiel.

Il y a par ailleurs une chose bien précise qui révèle que le nombre d'appels est exponentiel : pour obtenir la valeur de `fibomauvais(n)`, on ne peut faire que des additions à partir de deux valeurs de base : `0` et `1`. Or, on sait à partir du cours de mathématiques calculer f_n, qui est de l'ordre du nombre d'or à la puissance de l'indice. Ainsi, engendrer ce nombre f_n nécessite de passer f_n fois par un appel à `fibomauvais(1)` et un nombre de fois du même ordre par un appel à `fibomauvais(0)` (exercice pour le lecteur : prouver que ce nombre est f_{n-1}).

Demandons-nous à présent comment faire mieux.

2.3.2 Intuition de la notion

Une version plus pertinente de l'algorithme stocke dans une liste ou dans un tableau les données calculées. Le principe dit de **mémoïzation** consiste à retenir les valeurs calculées et n'en calculer de nouvelles que si elles ne sont pas disponibles.

Dans l'idée, demander pour une fonction de genre la valeur de $f(x)$ se fait en deux étapes : on teste si $f(x)$ est stocké dans la mémoire en construction, et si oui on le retourne immédiatement, et sinon on le calcule en suivant la formule décrivant f et on le stocke pour tous les appels ultérieurs.

Pour se faciliter la vie et ne pas avoir à écrire un programme qui vérifie la disponibilité, non seulement on retiendra toutes les valeurs utiles par défaut, mais de plus on pourra les calculer dans un ordre pertinent, souvent à partir du cas de base et de façon monotone.

Ces approches sont dites *top-down* et *bottom-up* en anglais, non traduites de manière officielle en français mais on pourra dire « de haut en bas » ou « de bas en haut ».

Il s'agit généralement d'écrire une boucle demandant à calculer et stocker les valeurs, et l'ordre des calculs sera tel qu'on pourra savoir que tous les appels récursifs déclenchés par l'appel principal de la boucle se feront pour des valeurs déjà connues.

Ainsi, les deux optimisations suivantes sont en temps linéaire et en espace respectivement linéaire et constant : [1]

1. Pour être complet sur le sujet, il est possible de calculer f_n en temps logarithmique en n à l'aide de l'exponentiation rapide de matrices, mais ceci ne relève pas de ce chapitre.

```
def fibolin(n):
    assert n >= 0
    if n < 2:
        return n
    reponses = [0, 1]
    def aux(k):
        if k == n:
            return reponses[-1]
        reponses.append(reponses[-1] + reponses[-2])
        return aux(k+1)
    return aux(1)

def fibocstt(n):
    assert n >= 0
    if n < 2:
        return n
    def aux(fnm1, fnm2, k):
        if k == n:
            return fnm1 + fnm2
        return aux(fnm1 + fnm2, fnm1, k+1)
    return aux(1, 0, 2)
```

Il s'agit ici de deux programmes avec une approche bottom-up : on calcule les valeurs à partir de 0 et quand la fonction `aux` étudie l'indice `k`, il est garanti que les indices inférieurs ont déjà été calculés, sachant que dans le deuxième cas les indices inférieurs de trois unités peuvent être oubliés.

Un autre exemple plus élaboré est le calcul d'un coefficient binomial sans utiliser la factorielle. On peut se reposer sur une des deux formules suivantes :
$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$ (formule de Pascal) et $\binom{n}{k} = \frac{n}{k} \times \binom{n-1}{k-1}$ (formule dite du capitaine).

Pour la première formule, le cas de base est $\binom{p}{0} = \binom{p}{p} = 1$ pour tout $p \geq 0$.

En effet, dès qu'on cherche $\binom{n}{k}$ pour $k < n$, on va utiliser le résultat de $\binom{k}{k}$ et celui de $\binom{n-k}{0}$ dans les chemins extrêmes de calcul.

Pour la deuxième formule, le cas de base est $\binom{p}{0} = 1$ pour tout $p \geq 0$.

En effet, les deux paramètres sont diminués de 1 à chaque étape, et le premier est normalement supérieur au deuxième.

Pour garantir la terminaison et puisque par convention $\binom{n}{k} = 0$ si l'un des paramètres est négatif ou si $n < k$, on ajoute ceci aux cas de base.

Voici donc une version mauvaise en termes de complexité, puis une version écrite en se servant de la programmation dynamique :

```
def newton_P_mauvais(n, k):
    if n < 0 or k < 0 or n < k:
        return 0
    if k == 0 or k == n:
        return 1
    return newton_P_mauvais(n-1, k) + newton_P_mauvais(n-1, k-1)

def newton_P(n, k):
    if n < 0 or k < 0 or n < k:
        return 0
    if k == 0 or k == n:
        return 1
    ligne = [1] * (k+1)
    def aux(m):
        for i in range(min(m-1, k), 0, -1):
            ligne[i] += ligne[i-1]
        if m == n:
            return ligne[k]
        return aux(m+1)
    return aux(1)
```

On note que contrairement au cas où on souhaiterait toutes les valeurs de $\binom{n}{k}$ pour $0 \leq k \leq n$ [2], la formule « multiplicative » est plus efficace ici car elle demande de calculer un nombre linéaire de valeurs et d'en mémoriser une seule, tandis que la formule « additive » nécessite de calculer des valeurs formant un « rectangle » dans le triangle de Pascal, soit un nombre quadratique de valeurs à calculer (et un nombre linéaire à mémoriser, car une fois une ligne calculée, les précédentes sont inutiles).

2. Là, il faut calculer tout le triangle jusqu'à la ligne en question et additionner est plus rapide que multiplier.

Pour tout dire, selon certains la formule multiplicative ne relève pas tout à fait de la programmation dynamique, car chaque appel récursif n'en engendre à chaque fois qu'un autre, et il est tout aussi pertinent de faire une boucle.

```
def newton_cap(n, k):
    if n < 0 or k < 0 or n < k:
        return 0
    def aux(m):
        if m == 0:
            return 1
        return aux(m-1) * (n-k+m) // m
    return aux(k)
```

2.3.3 Formalisme

La programmation dynamique peut se formaliser ainsi : on considère un problème qu'on assimile au calcul de l'image par une fonction f de paramètres $p_1, \ldots, p_n$ (de n'importe quel type).

1. On cherche à établir une **formule de récurrence** (éventuellement donnée) liant $f(p_1, \ldots, p_n)$ à une ou plusieurs (toujours plusieurs en pratique) images par f d'autres paramètres **en garantissant la terminaison** (donc il doit s'agir de sous-problèmes).
2. On résout ces sous-problèmes par **mémoïzation** (donc en stockant tout ce qui peut être utile à la volée, car les sous-problèmes sont indépendants).
3. On **recombine les solutions**.

Il s'agit donc de profiter de la possibilité de mémoriser les résultats d'appels déjà effectués par une autre branche de l'exécution, quand les sous-problèmes se chevauchent. Plus ce chevauchement a lieu, et plus le gain en complexité en espace se ressent par rapport à un programme récursif naïf ou sa version itérative.

En voyant le formalisme précédent du point de vue informatique avec un problème à résoudre plutôt que mathématique avec une fonction à calculer, il s'agit d'établir que la solution au problème en question pour une certaine structure peut se déduire de la solution au même problème sur des sous-structures, d'où la mention dans le programme officiel de la propriété de sous-structure optimale.

Une belle illustration de la mémoïzation revient à refaire deux versions « jumelles » en espace linéaire de Fibonacci à l'aide d'un tableau géré comme une variable globale aux appels récursifs et locale à la fonction principale (raffinement : globale tout court, afin de pouvoir calculer plusieurs termes de la suite, mais il faudrait alors potentiellement redimensionner le tableau).

```
# Version avec effet de bord uniquement dans aux
def fibodyn(n):
    tab = [-1] * (n+1)
    tab[0] = 0
    tab[1] = 1
    def aux(i):
        if tab[i] == -1:
            aux(i-2) # effet de bord
            aux(i-1) # idem
            tab[i] = tab[i-1] + tab[i-2] # le voici !
        # sinon ne rien faire
        # aux(i) garantit que tab[i] contient la réponse
    aux(n)
    return tab[n]

# Version avec effet de bord et retour dans aux
def fibodyn2(n):
    tab = [-1] * (n+1)
    tab[0] = 0
    tab[1] = 1
    def aux(i):
        if tab[i] == -1: # il faut d'abord placer la valeur
            tab[i] = aux(i-2) + aux(i-1)
        return tab[i] # dans tous les cas
        # aux(i) garantit que tab[i] contient la réponse, et la retourne
    return aux(n)
```

Suivant le même principe, on peut calculer un coefficient binomial avec mémoïzation, soit par la méthode selon laquelle la fonction récursive fait uniquement un effet de bord comme dans le premier programme ci-avant, soit avec effet de bord et valeur de retour par une adaptation laissée en exercice :

```
def coeffbin(n, k):
    memoire = [[-1 for j in range(k+1)] for i in range(n+1)]
    for i in range(n+1):
        memoire[i][0] = 1
    for j in range(1, k+1):
        memoire[0][j] = 0
    def aux(nn, kk):
        if memoire[nn][kk] == -1: # en particulier nn et kk sont > 0
            aux(nn-1, kk)
            aux(nn-1, kk-1)
            memoire[nn][kk] = memoire[nn-1][kk] + memoire[nn-1][kk-1]
    aux(n, k)
    return memoire[n][k]
```

2.3.4 Remarques d'usage

Voyons un exemple supplémentaire : le problème du rendu de monnaie. On cherche à utiliser le moins de pièces (ou billets) possibles dans un système monétaire donné pour produire une somme donnée.

Les paramètres sont la somme en question et la liste des valeurs de pièces/billets possibles, et la fonction f retourne le minimum du nombre de pièces/billets nécessaires.

Pour les euros, on a donc la fonction f, en écrivant l la liste $[1, 2, \ldots, 20000, 50000]$ des valeurs exprimées en centimes, donnée par la formule de récurrence : $f(s, l) = 1 + \min\{f(s - v, l) \mid s - v \geq 0, v \in l\}$, avec $f(0, l) = 0$ et s est également exprimée en centimes.

Une remarque au passage : les soucis liés à l'utilisation des flottants font qu'on privilégie l'expression en centimes dans un programme effectivement écrit sur l'ordinateur, et il serait intéressant de tomber sur un cas d'échec d'un programme utilisant des flottants, par exemple en testant le rendu d'un euro avec uniquement des pièces inférieures à 20 centimes.

En mémorisant toutes les valeurs, on ne les calcule qu'une fois et la complexité est de l'ordre du produit de la valeur à rendre par le nombre de pièces et billets. La complexité en espace s'en ressent certes, d'où **la possibilité d'utiliser un dictionnaire pour les valeurs mémorisées**.

Si on sait que les valeurs mémorisées couvrent un intervalle d'entiers démarrant à zéro, l'utilisation d'un tableau, où toutes les cases seraient donc utiles, est cependant plus simple. Si l'intervalle ne démarre pas à zéro, il est là encore pertinent d'utiliser un tableau, et d'opérer un changement d'indices pour la reconstitution de la réponse.

Le problème du rendu de monnaie a, par ailleurs, déjà été abordé en première année comme illustration des algorithmes gloutons, où ces derniers sont optimaux pour les euros mais pas dans n'importe quel système monétaire a priori.

Chapitre 3

Algorithmique pour l'intelligence artificielle et l'étude des jeux

Ce chapitre, totalement inédit dans l'enseignement en classes préparatoires, fait écho au développement de l'intelligence artificielle comme domaine de l'informatique en plein essor, et donne également enfin aux bases de la théorie des jeux une place de choix.

Ce dernier domaine, pouvant relever à la fois des mathématiques, pour ce qui est de l'enseignement qui sera fait ici, et de l'économie, en guise d'application très courante et au prix de l'ajout de quelques notions basiques, n'est pas à comprendre de travers.

Il est en effet aussi faux de croire que la théorie des jeux se résume à dire comment gagner à tous les coups que de réduire l'informatique à la programmation de jeux vidéo.

Il s'agit plutôt de considérer que dans l'exécution d'un programme par exemple tout n'est pas connu à l'avance et des interactions favorables ou défavorables peuvent changer le résultat final, ce dont on pouvait déjà se rendre compte lorsque le hasard intervenait.

Pour autant, la combinaison adversaire + hasard, évidemment présente en théorie des jeux, ne sera pas abordée dans ce cours.

Il y a ici un objectif double : accroître la culture informatique et développer les compétences algorithmiques, voire donner des réflexes de mise en œuvre d'algorithmes à partir de problèmes dans n'importe quel contexte.

3.1 Intelligence artificielle

3.1.1 Notion d'apprentissage

L'intelligence artificielle, de plus en plus présente dans nos vies, vise à faire prendre par des algorithmes des décisions orientées à partir de données, si possible en quantité phénoménale (c'est le principe du "Big Data"), avec l'efficacité d'un cerveau humain qui peut raisonner.

Par ailleurs, enrichir son jeu de données par de nouvelles connaissances issues de tests précédents, avec une forte adaptabilité, de manière à limiter les erreurs ultérieures, ce qui revient à apprendre par l'expérience, est également un objectif à mettre en œuvre pour de tels algorithmes.

Deux notions seront en jeu dans cette section : l'apprentissage supervisé et l'apprentissage non supervisé.

Dans le premier, l'algorithme doit prendre une certaine décision, souvent un choix binaire, sur des données prises individuellement, sachant qu'il dispose d'une banque de données pour lesquelles la « bonne décision » est fournie.

Cela correspond à l'idée que l'humain aide l'ordinateur sur les premières instances avant de le laisser agir sur les données à traiter.

Malheureusement, quand bien même une formule générale existerait, le principe est de ne pas la fournir, et par ailleurs l'objectif n'est a priori pas de la faire déduire. Dans les applications de l'apprentissage supervisé, le recours à une formule est de toute manière exclu, d'où le fait que l'on se considère dans la même situation pour les cas basiques qui seront traités.

Concernant l'apprentissage non supervisé, les données sont cette fois traitées ensemble, et la décision sera plutôt à prendre en les comparant entre elles, sans la moindre information initiale.

L'algorithme dit des k *plus proches voisins* est l'exemple d'apprentissage supervisé retenu par le programme officiel, et l'algorithme dit des k-*moyennes* relève de l'apprentissage non supervisé.

3.1.2 Algorithme des k plus proches voisins

Présentation du problème

Commençons par un problème simplifié laissant introduire l'algorithme des k plus proches voisins.

Imaginons une forme géométrique dans un plan, de préférence convexe. Disposons sur le plan, dans une zone bien délimitée où la forme est entièrement contenue, un certain nombre de points. Certains seront connus pour faire partie de la forme, d'autres non, mais dans tous les cas la connaissance de l'appartenance ou non à la forme est supposée acquise.

Maintenant, disposons un ou plusieurs points dans le plan et demandons-nous s'ils font partie de la forme ou non. Pour répondre avec le moindre risque d'erreur possible, une possibilité est d'étudier les distances aux points de la forme et les distances aux autres points, puis de trancher. Il s'agit donc d'un apprentissage supervisé.

Considérons un entier naturel impair (c'est mieux !) k inférieur au nombre de points disponibles. Il est possible de calculer pour chaque point supplémentaire du plan les k points connus les plus proches de celui-ci. À partir de ces k points, la décision se fera en fonction de la majorité entre l'appartenance ou la non-appartenance à la forme.

Le fait que k soit impair évite d'avoir une égalité, et dans les modèles classiques les coordonnées sont des flottants, limitant le risque d'égalités rendant le choix des k plus proches voisins difficile.

Généralisation

Une généralisation du problème ici présenté revient à choisir n'importe quelle structure pour les données, pourvu qu'on puisse établir entre les données une fonction agissant comme une distance, sans qu'il n'y ait nécessairement besoin que cette fonction ait les propriétés mathématiques d'une distance, cependant. **Dans le programme officiel, on se contente de la distance euclidienne.**

De plus, le choix n'est pas nécessairement binaire, et avec une valeur suffisamment grande pour k, qui n'a alors plus besoin d'être impair, on peut toujours espérer tomber sur un choix ayant la majorité relative.

En outre, charge au programmeur de disposer d'une base d'apprentissage suffisamment fournie et de sélectionner une valeur pertinente pour k, suffisamment grande pour éviter trop d'arbitraire, mais suffisamment petite pour ne pas tenir compte de valeurs trop éloignées (si k s'approche de l'effectif total, la localisation du point n'a presque plus d'effet, par exemple).

À ce stade, une alternative au choix de k voisins pourrait être de prendre tous les voisins dans un certain rayon, avec un nombre minimum et un nombre maximum à prendre quoi qu'il en soit, mais cela déplacerait le problème au choix d'un bon rayon...

Une autre piste est de pondérer les effectifs par un coefficient décroissant en fonction de la distance et de déterminer quelle réponse a le meilleur score, afin de limiter un biais qui serait dû à la sous-représentation d'une des réponses, risquant autrement dans certains cas de ne jamais la faire apparaître.

Pour la culture, prendre $k = 1$ renvoie à la notion de diagramme de Voronoï.

L'algorithme

L'algorithme des k plus proches voisins se décompose en plusieurs étapes.

La représentation des données sous la forme d'une valeur permettant le calcul des distances n'est pas forcément à portée de main. Elle alors nécessiter un pré-calcul, où on pourra être amené à faire un choix entre plusieurs représentations possibles.

Ensuite, toutes les distances se calculent à partir de la formule mathématique établie et dépendant de la représentation retenue, entre le point à traiter et les données disponibles.

Cette étape est sujette à optimisation (possibilité d'exclure directement certaines données si on peut « savoir » en raison de propriétés particulières qu'elles ne feront pas partie des k plus proches voisins, et que la vérification permettant cette exclusion est bien plus rapide que le calcul de distance qu'elle permet d'éviter).

Une fois les distances disponibles, il faut trouver les k plus proches voisins. Si k est fixé, un algorithme classique mémorise à tout moment d'un parcours les k meilleures valeurs. Si k est grand ou arbitraire, on pourra aussi se poser la question de l'efficacité d'un tri puis de la récupération des k premiers résultats...

Finalement, le recensement parmi l'échantillon et la récupération de la valeur la plus fréquente se fait sans peine au regard du travail déjà accompli.

Matrice de confusion

Une bonne façon de voir si les données et la valeur de k risquent de donner des réponses acceptables est de commencer par tester les données déjà disponibles.

De manière générale, le programmeur peut faire un retour sur expérience en constatant parmi les deux réponses, dans un cas binaire, combien de fois chacune a été donnée à raison, et combien de fois elle a été donnée à tort.

Ce comptage permet de remplir les quatre cases d'un tableau à double entrée appelé la *matrice de confusion*, dont voici un exemple dans le cas présenté en début de section :

	Points de la forme	Points hors de la forme
Points signalés comme dans la forme	90	2
Points signalés comme hors de la forme	10	98

Dans les cas de construction usuels de matrices de confusion, une des réponses est considérée comme positive et l'autre négative.

On déduira des effectifs dans les cases des valeurs associées à l'instance de tests comme la « précision » (terme non conventionnel), calculée comme le nombre de réponses positives détectées à raison divisé par le nombre total de données où une réponse positive était attendue, ou la « correction » (même remarque), calculée comme le taux de succès de l'algorithme...

Pour l'instance ayant donné la matrice de confusion donnée en exemple, cela donne une précision de 90% et une correction de 94%.

Applications

Pour une banque de données de taille conséquente et un choix non binaire, on citera notamment la reconnaissance de chiffres manuscrits.

Ce problème a l'avantage de permettre de disposer d'une banque de données en accès facile, en l'occurrence la base de données MNIST dont les fichiers sont téléchargeables à l'adresse `http://yann.lecun.com/exdb/mnist/`, en ignorant l'éventuelle demande d'identifiants.

Ici, on considèrera des images en noir et blanc binarisées (ce qui donne l'occasion de revenir sur la manipulation d'images vue en TP au premier semestre de SUP), et de mêmes dimensions, en passant au besoin par une mise à l'échelle et un recentrage, et la distance entre deux images sera le nombre de pixels blancs dans une image et noirs dans l'autre (ou vice-versa donc).

Sélectionner les k images les plus proches et le chiffre qu'elles représentent (information fournie) permettra à un programme de prédire le chiffre qu'une image fournie représente.

Une telle application s'étendra à la reconnaissance de n'importe quel type d'images, avec une utilité dans la vie quotidienne pour les voitures autonomes devant identifier les panneaux de signalisation, mais aussi la reconnaissance d'émotions sur un visage.

3.1.3 Algorithme des k-moyennes

Présentation du problème

Les cas d'applications de l'algorithme des k-moyennes ressemblent assez à ce qui a été vu ci-avant, à ceci près que la base de données initiale n'est pas fournie.

Ainsi, l'ordinateur fait cette fois face à plusieurs données, sans information dans le moindre cas quant à une certaine étiquette qui a été attribuée à chacun, mais dans l'optique de rassembler les données en groupes cohérents auxquels on attribuera la même étiquette. Les données sont une fois de plus associées à une valeur prise dans un ensemble muni d'une distance, et on choisira cette fois-ci $\mathbb{R}^d$, pour une certaine dimension d, afin de pouvoir calculer des barycentres (en l'occurrence de simples moyennes car les coefficients seront tous égaux à 1).

La notion de cohérence pour les groupes revient à calculer la variance des distances au sein de chacun des groupes qui ont été formés, avec pour objectif de minimiser la somme de ces variances, sous la contrainte que le nombre de groupes soit fixé, précisément à la valeur k, pour ne pas baisser artificiellement la somme des variances en formant de nombreux groupes de faible variance, ou dans le cas extrême autant de groupes que de données, ne donnant que des variances nulles.

L'intuition est que des données correspondant à la même étiquette entraîneront des valeurs proches entre elles, mais éloignées des valeurs des données correspondant à une autre étiquette.

Ainsi, il est prévisible que « la » répartition par étiquettes qui minimise la somme des variances est l'attribution de la bonne étiquette à chaque donnée.

Plusieurs bémols s'imposent à cette phrase :

— Le mot « la » a été entouré de guillemets car on n'a pas de garantie d'unicité de la répartition qui minimise la somme des variances. Pour autant, on peut admettre que les égalités seront rares voire impossibles, notamment si on travaille avec des flottants.
— Il est prévisible, mais pas forcément garanti...
— Enfin, la valeur de k doit pour commencer être connue ou bien devinée, pour qu'il y ait le moindre espoir de trouver la bonne répartition. Ceci étant, parfois il est possible de se contenter d'une répartition compatible avec la bonne, qui fusionnera certes deux groupes, tant qu'aucune donnée d'un groupe ne figure à tort dans un autre groupe avec d'autres données par exemple.

Comme annoncé en introduction de la section, il s'agit d'un apprentissage non supervisé, et le programmeur n'a pas non plus à intervenir avant la fin de l'exécution.

L'algorithme

L'algorithme des k-moyennes existe avec plusieurs variations, notamment en termes d'initialisation et d'ajout d'une condition d'arrêt quantitative ou d'un nombre de tours maximal.

Une version simple sous forme de « recette de cuisine » figure ci-après.

```
Sélectionner k éléments distincts parmi les données, notés m1 à mk
  en tant que moyennes initiales (les stocker dans un tableau nommé m).
Créer k ensembles notés E1 à Ek (les stocker dans un tableau nommé E).
Créer le tableau mbis initialisé à rien.
Tant que mbis est différent de m {
  Remplacer mbis par m.
  Placer les points du plan dans E de sorte que pour tout i
    l'ensemble Ei soit formé des données qui ont mi pour donnée la plus proche
  Recalculer chaque mi en tant que moyenne des éléments de chaque Ei
  (Si un Ei est vide, prendre un point au hasard)
}
Renvoyer les Ei.
```

On remarquera que `mi` n'est plus forcément une donnée, mais peut être un point quelconque du plan.

Cet algorithme ne calcule pas la variance de chaque `Ei`, mais ces variances, ou plus précisément leur somme, jouent un rôle dans la compréhension de l'algorithme.

L'algorithme présenté ici a quelques particularités par rapport à ceux qui étaient enseignés jusque là : comme le suggèrent les bémols ci-avant, il n'est pas certain de donner la bonne réponse (mais celui des k plus proches voisins non plus), ni même de converger vers un ensemble minimisant la somme des variances.

En pratique, ce minimum existe simplement car il y a un nombre fini de valeurs possibles, issus d'un nombre fini de répartitions possibles. La preuve de convergence, autrement dit de sortie de la boucle conditionnelle, n'est pas au programme non plus, mais repose sur le fait qu'à chaque tour de boucle la somme des variances ne peut pas augmenter strictement, ainsi, elle diminue strictement **ou** l'algorithme s'arrête parce que les moyennes n'ont pas changé (en excluant le cas où les points changent mais que la somme des variances reste identique, par des arguments un peu plus forts). Le nombre fini de valeurs possible permet alors de conclure sur un arrêt de l'algorithme, une fois qu'il a atteint un minimum local.

Concernant le nombre de tours de boucles nécessaires, donnant la complexité, il n'est pas acquis qu'il soit borné par une fonction polynomiale en la taille de l'entrée, même s'il reste inenvisageable de remplacer l'algorithme des k-moyennes par une exploration exhaustive, qui est en temps exponentiel.

Applications

Allègeons la contrainte de ne rien connaître sur les étiquettes à attribuer à chaque donnée, mais faisons comme si elles devaient rester inconnues pendant l'exécution de l'algorithme. Une fois les groupes formés, le programmeur peut alors déduire que d'autres données, jusque là sans étiquette connue, partagent l'étiquette d'un autre élément connu avec lequel ces données ont été regroupées.

Une application à la reconnaissance faciale se dessine alors, en prenant une banque de photos dont très peu sont associées à une identité, au point d'altérer l'efficacité de l'algorithme des k plus proches voisins.

D'autres applications se retrouvent plutôt en biologie, même si ce n'est par exemple pas la méthode la plus classique pour la construction d'arbres phylogénétiques et la classification d'espèces en fonction de leur degré de parenté.

3.2 Bases de la théorie des jeux

Comme annoncé en introduction du chapitre, la théorie des jeux abordée ici sera traitée d'un point de vue mathématique, en reprenant la notion de graphe vue en première année et en introduisant des fonctions et ensembles.

En guise d'introduction, plaçons-nous dans la situation où quelqu'un qui comprend la notion de jeu de la vie réelle formule l'intuition de ce qu'est un jeu informatique...

Le principe est ici de modéliser en tant que jeu une situation ou deux adversaires veulent remplir un certain objectif, les objectifs de chacun étant non compatibles, de sorte qu'il est impossible que les deux gagnent. Une égalité est cependant possible.

Pour les jeux dits d'accessibilité, seuls au programme, des hypothèses restrictives seront faites.

Pour commencer, la structure d'appui sera un graphe fini, et l'information est dite parfaite, de sorte qu'aucun élément servant à la décision ne soit inconnu des joueurs. Nous ne pouvons ainsi pas modéliser des jeux de cartes où les mains sont cachées.

Ensuite, le hasard n'interviendra pas, y compris dans l'initialisation du jeu. *Exeunt* ainsi les dés et autres dominos.

Concernant les actions des joueurs, nous nous contenterons de jeux en tour par tour, à la différence des jeux dits concurrents, qui se retrouvent par ailleurs plutôt en économie. Cela exclut le Shifumi.

Enfin, nous ferons en sorte que toute prise de décision puisse se faire simplement en considérant la situation actuelle, sans tenir compte de tout ce qui a pu se passer avant, ce qui permet de faire la distinction entre stratégies positionnelles et stratégies à mémoire.

Le fait que l'information soit parfaite ne garantit pas que de telles stratégies soient efficaces, mais parfois la mémoire nécessaire est suffisamment limitée pour qu'on puisse considérer qu'elle est encodable dans le graphe d'appui.

Ainsi, aux échecs, l'historique de la partie a une très légère influence sur les coups autorisés, et peut donc pour cette raison suggérer un coup efficace uniquement parce qu'un certain événement s'est produit antérieurement, mais l'information nécessaire est facile à résumer, et il suffit qu'elle soit finie :

- Quels sont les roques encore autorisés ?
- Quel est le dernier mouvement dans l'optique d'une prise en passant ?
- Quels sont plus généralement les cinq derniers coups dans l'optique d'une annulation de la partie par répétition de position ?
- Dans combien de coups la partie pourra être annulée ?

Pour d'autres jeux que ceux que nous étudierons, en l'occurrence pour d'autres objectifs de gain, l'historique entier d'une partie peut être nécessaire avant de prendre une décision, car les informations utiles nécessiteraient de toute manière une mémoire illimitée, ne serait-ce que parce qu'on maintient à jour un compteur non borné.

3.2.1 Introduction générale aux jeux

Le fil rouge de cette sous-section sera le jeu, sans doute très connu, dit « des petits bâtons ». Il figurera en jaune par la suite.

En cas de besoin, voici un rappel des règles : vingt-et-un [1] bâtonnets sont alignés, et deux joueurs en retirent à tour de rôle un, deux ou trois. Celui qui prend le dernier a perdu.

1. le plus souvent, mais cela marche avec d'autres valeurs

Notations et introduction

Tout au long de cette section, nous utiliserons les notations suivantes : pour un ensemble X quelconque, X^* est l'ensemble des séquences finies d'éléments de X, X^ω est l'ensemble des séquences infinies d'éléments de X et $X^\infty = X^* \cup X^\omega$.

Les étudiants de l'option informatique auront l'occasion de reconnaître l'étoile de Kleene vue a priori plus tard dans l'année. Quant à ω, il s'agit d'une référence aux nombres ordinaux. On peut remplacer ω par $\mathbb{N}$ dans les notations.

On considère deux joueurs appelés Adam et Ève. Ces noms sont associés aux quantificateurs universel et existentiel, car on se place du point de vue d'Ève, pour laquelle on se demande s'*il existe* une stratégie telle que, *pour toute* stratégie d'Adam, Ève gagnera, en essayant par ailleurs de déterminer la stratégie.

Il est alors intéressant de constater qu'une preuve faisant intervenir des quantificateurs peut s'interpréter comme un jeu. Ceci permet éventuellement une meilleure compréhension de l'effet des quantificateurs !

Dans la littérature, on trouve également des mentions des joueurs 0 et 1, ou 1 et 2, voire Abélard et Éloïse, stylisés en ∀bélard et ∃loïse. Ces derniers sont cependant moins connus en-dehors de la France...

Définitions de base

Définition

Une arène est un triplet $\mathcal{G} = (G, S_A, S_E)$ où $G = (S, T)$ est un graphe orienté (fini en ce qui nous concerne) et (S_A, S_E) une partition de S. Les éléments de S_A sont appelés « sommets d'Adam » et les éléments de S_E sont appelés « sommets d'Eve ».

On notera l'utilisation de la lettre T pour l'ensemble des arcs, parfois rencontrée ailleurs dans la littérature, et utilisée ici en raison de risques ultérieurs de confusion. De toute manière cet ensemble ne sera plus évoqué par la suite.

Pour notre jeu, le graphe sera composé de quarante-quatre sommets, dont les noms sont les n_J pour n de 0 à 21 et J valant A ou E. Intuitivement, on comprend que S_A est composé de tous les sommets indexés par A et de même pour S_E.

Il existe un arc entre n_J et m_K si, et seulement si, $n - m$ est compris entre 1 et 3 et $J \neq K$. En particulier, les sommets 0_A et 0_E n'ont pas d'arc sortant.

Définition

Une condition de gain associée à une arène est un sous-ensemble de S^∞, où S est l'ensemble des sommets de l'arène.

La condition de gain de notre jeu sera détaillée ultérieurement, en parler ici serait prématuré.

Définition

Un jeu est un triplet $(\mathcal{G}, \Omega_A, \Omega_E)$, où $\mathcal{G}$ est une arène, Ω_A est une condition de gain associée à $\mathcal{G}$ et attribuée à Adam et Ω_E est une condition de gain également associée à $\mathcal{G}$ et attribuée à Ève, avec la contrainte que $\Omega_A \cap \Omega_E = \emptyset$.

Lorsque $\Omega_A \cup \Omega_E = S^\infty$, on peut se contenter de la condition de gain associée à Ève.

Définition

Une partie dans un jeu $(\mathcal{G}, \Omega_A, \Omega_E)$, est une suite, finie ou non, de sommets de l'arène, notée $s_0 s_1 \ldots$, construite de la façon suivante : s_0 est un sommet de départ fixé (on peut parler de partie depuis s_0), et pour tout i, le choix de s_{i+1} est laissé au propriétaire du sommet s_i parmi les sommets accessibles par un arc depuis s_i. La partie est gagnée par un joueur si elle appartient à sa condition de gain.

Pour notre jeu, une partie possible où Adam commence est la suite

$$21_A 18_E 17_A 16_E 13_A 11_E 9_A 7_E 5_A 4_E 1_A 0_E.$$

En considérant les nombres figurant dans le nom des sommets comme le nombre de bâtonnets restants, cela correspond à une partie réelle du jeu où Adam s'est retrouvé obligé de prendre le dernier bâtonnet et a perdu. La condition de gain pour Ève dans ce jeu est l'ensemble des parties qui se terminent en 0_E et celle, complémentaire, pour Adam est l'ensemble des parties qui se terminent en 0_A. Ici, en fait, il n'existe pas de partie infinie. Nous verrons ultérieurement qu'il s'agit de conditions d'accessibilité.

Stratégies, ensembles gagnants

Il est temps de détailler la méthode que chaque joueur emploie pour choisir le sommet accessible à chaque fois que c'est son tour.

Définition

Une stratégie σ pour un joueur dans un jeu $(\mathcal{G}, \Omega_A, \Omega_E)$ est une fonction prenant en paramètre le début d'une partie dont le dernier sommet, noté s, appartient au joueur en question et renvoyant un sommet accessible depuis s.

On dit qu'un joueur joue selon une stratégie σ si à chaque fois qu'il doit choisir un nouveau sommet il sélectionne l'image par σ du début de la partie jusqu'ici.

Si les deux joueurs jouent selon une certaine stratégie, la partie est entièrement caractérisée par le sommet de départ.

Définition

Une stratégie σ est dite positionnelle si pour deux débuts de partie P_1 et P_2 quelconques, le fait que P_1 et P_2 aient leur dernier sommet en commun implique que $\sigma(P_1) = \sigma(P_2)$. On peut alors voir une stratégie positionnelle d'Adam comme une fonction de S_A dans S et une stratégie positionnelle d'Ève comme une fonction de S_E dans S.

La notion hors-programme de stratégie mixte étend les stratégies dites déterministes et présentées ici. Au lieu de renvoyer un seul sommet, une stratégie mixte renvoie une distribution de probabilité sur les sommets possibles. Jouer selon une stratégie mixte revient à tirer au sort le sommet en respectant la distribution de probabilité en question.
Il existe également la possibilité d'introduire des stratégies non-déterministes, renvoyant un sous-ensemble de l'ensemble des sommets possibles.

Définition

Une stratégie σ pour un joueur est dite gagnante si pour toute stratégie σ' de son adversaire, toute partie démarrant dans un sommet quelconque et où chacun joue selon sa stratégie est gagnée par le joueur considéré.

On dispose aussi de la notion plus faible de *stratégie gagnante depuis un sommet s*, d'où la définition suivante.

Définition

Un sommet s est dit gagnant pour un joueur si celui-ci a une stratégie gagnante depuis s.

Dans notre jeu, on propose une stratégie gagnante depuis chaque sommet n_J tel que n ne soit pas congru à 1 modulo 4, consistant précisément à aller dans le seul sommet accessible qui soit justement congru à 1 modulo 4. Exception : depuis un sommet 0_J, la partie est terminée et donc il n'y a pas à définir de stratégie.

On déduit de ce qui précède une partition de S en $(W_A, W_E, W_\emptyset)$, respectivement l'ensemble des sommets gagnants pour Adam, l'ensemble des sommets gagnants pour Ève et l'ensemble des sommets pour lesquels aucun joueur n'a de stratégie gagnante, impliquant que chaque joueur dispose d'une stratégie permettant de gagner ou de faire une partie nulle quelle que soit la stratégie de son adversaire, ce qui suggère que les joueurs choisiront une stratégie optimale et que la partie qui en découlera sera nulle.

Il est important d'avoir à l'esprit que si l'humain réfléchit en cours de partie, suggérant une construction de sa stratégie au fur et à mesure, une stratégie est formellement déjà prête car elle tient compte de toutes les possibilités.

Ceci étant, une telle fonction pourrait ne pas être constructible ou stockable en mémoire, d'où le besoin de simplifier les calculs, et d'où la restriction aux stratégies positionnelles imposée par le programme.

Définition

Un jeu est dit déterminé si l'ensemble $W_\emptyset$ est vide, ce qui revient à dire que depuis n'importe quel sommet de départ, un joueur a une stratégie gagnante.

Dans certains cas, la modélisation des jeux fera que le sommet de départ est imposé. Dans ce cas, le fait que le jeu soit déterminé ou non ne nécessitera pas que chaque sommet soit gagnant pour un des joueurs, simplement celui de départ. Charge au joueur pour lequel le sommet de départ est gagnant de jouer selon une stratégie lui permettant de forcer la partie à ne visiter que des sommets gagnants pour lui.

Pour aller plus loin, un terme plus précis existe : un jeu est *positionnellement déterminé* s'il est déterminé et depuis tous les sommets le joueur ayant une stratégie gagnante a également une stratégie gagnante qui est positionnelle, sachant qu'une stratégie gagnante positionnelle doit être gagnante face à toutes les stratégies de l'adversaire, pas seulement celles qui sont positionnelles.

La plupart des jeux sous les restrictions que nous avons annoncées, et même beaucoup d'autres sans ces restrictions, sont déterminés. Des résultats très généraux dans ce sens ont été prouvés par David Gale (à qui l'on doit le problème des mariages stables) et Frank Stewart en 1953, puis par Donald Martin en 1975. Il s'agit pour ainsi dire de citations incontournables dans des articles de théorie des jeux.

Le jeu des bâtonnets est ainsi déterminé, et le joueur qui commence a une stratégie gagnante si, et seulement si, le sommet de départ n'est pas un multiple de quatre plus un. Une version plus standarde du jeu fait gagner le joueur qui prend la dernière, parce que dans l'esprit son adversaire ne peut plus jouer, et dans ce cas cela revient au jeu précédent avec un bâtonnet de plus. Pour la culture, il s'agit d'un cas simple de jeu de Nim.

Précisions sur les parties, contraintes du programme

Par toutes les définitions qui précèdent, les jeux ici sont *tour par tour*, ce qui signifie que les joueurs ne prennent pas de décision simultanément. Pour autant, cela ne signifie pas que tous les arcs partant d'un sommet d'Adam mènent à un sommet d'Ève et vice-versa.

En pratique, il est utile pour certaines classes de jeux que plusieurs sommets appartenant au même joueur s'enchaînent, mais dans notre cas il ne coûte rien de fusionner ces sommets successifs (exercice laissé au lecteur quand le moment viendra).

Le programme suggère ainsi que les jeux se jouent sur des graphes bipartis.

Les graphes de jeu peuvent même avoir une propriété supplémentaire plus forte que d'être bipartis, ils peuvent présenter une symétrie de sorte que S_A et S_E soient en bijection par une certaine application φ vérifiant que pour tout sommet s dans S_A, l'ensemble des successeurs de s (sous-ensemble de S_E) est l'image directe par φ de l'ensemble des successeurs de $\varphi(s)$ (sous-ensemble de S_A puisque $\varphi(s)$ est dans S_E).

On dit alors que le jeu est *impartial*, et il reviendrait au même de considérer que le graphe est réduit de moitié, en adaptant les arcs, et que les joueurs alternent leurs tours.

Si la partie est gagnée par un joueur, elle est considérée comme perdue par l'autre, mais si la partie n'appartient à la condition de gain d'aucun joueur, elle est déclarée nulle.

A priori, il n'y a aucune raison d'arrêter une partie sur un sommet quelconque. Une raison possible est qu'aucun arc ne parte du sommet courant, ou que la forme de la condition de gain garantisse que le gagnant de la partie est connu.

Dans le cas où une partie s'arrête, il est classique de déclarer systématiquement le joueur possédant le dernier sommet comme perdant, mais il est tout aussi imaginable de faire le contraire (les deux variantes existent, même au sein de la catégorie bien connue des jeux de Nim). Cela revient à dire que la condition de gain est entièrement caractérisée par le dernier sommet des parties finies, les parties infinies pouvant alors être déclarées nulles.

3.2.2 Jeux d'accessibilité

Les jeux d'accessibilité figurent à la fois parmi les plus classiques, que ce soit du point de vue de la théorie des jeux ou des jeux « réels » ainsi modélisés, mais aussi les plus simples, dans la mesure où la condition de gain associée contribue à ce que les parties étudiées soient finies.

Leur résolution dans le cadre de notre étude se fera par des algorithmes qui étendent les parcours de graphes vus en première année.

Les arènes considérées ici auront trois types de sommets : les sommets gagnants pour Adam, les sommets gagnants pour Ève et les autres. Cette idée de sommet gagnant sera associée à la condition de gain, comme vu plus loin.

Signalons qu'un sommet gagnant pour un joueur peut appartenir à l'autre, mais de toute manière nous pourrons admettre que les sommets gagnants pour un joueur n'ont pas besoin d'avoir d'arc sortant, donc leur propriétaire n'a pas de pertinence.

Le problème d'accessibilité

Commençons par fixer les idées en précisant par une définition formelle la notion intuitive d'accessibilité.

Quittons temporairement la théorie des jeux pour revenir à des structures « chaînées » quelconques, en commençant par des graphes.

Définition

Soit une structure munie d'un ensemble Q d'états et d'un moyen, a priori quelconque, de passer d'un état à un autre. Le problème d'accessibilité est la donnée d'un sous-ensemble F de Q et revient à poser l'une des deux questions suivantes :

- *Étant donné un élément q_0 de Q, existe-t-il une suite d'étapes permettant de passer de q_0 à un élément de F ?*
- *Quels sont les éléments de Q pour lesquels la réponse à la question précédente est oui ?*

La première question est un *problème de décision*, c'est-à-dire une question à laquelle la réponse est un booléen.

La deuxième demande une construction. En tout état de cause, le seul problème d'accessibilité ne demande pas une preuve par exhibition d'une suite possible d'étapes à mettre en œuvre pour atteindre un élément de F, ni même l'élément atteignable.

Le problème d'accessibilité peut se résoudre par un algorithme dont la complexité optimale déterminera la complexité du problème, mais sur certaines structures la complexité n'est pas connue, voire pire : parfois il n'existe pas de réponse.

Le problème indécidable de l'arrêt d'une machine de Turing est un exemple de problème d'accessibilité pour lequel on ne peut pas établir d'algorithme répondant forcément oui ou non sans se tromper et avec une terminaison garantie. Une autre structure pour laquelle il existe un algorithme est le réseau de Petri, mais la complexité optimale n'est pas encore connue, et on sait seulement qu'une tour d'exponentielles ne suffit pas dans le pire des cas.

Pour les structures qui nous intéressent en classes préparatoires, il n'y a pas de souci : le problème d'accessibilité dans un graphe se résout par un simple parcours, qui s'effectue en temps linéaire en la taille du graphe.

Il s'agit même d'un problème d'une classe de complexité plus restreinte, car il se résout en prenant un espace logarithmique en la taille du graphe par une machine de Turing non-déterministe. Pour vulgariser, il suffit d'écrire le code d'un état et un compteur, puis de transformer ce code en le code d'un état relié par un arc au précédent tout en incrémentant le compteur, en s'arrêtant si le code correspond à un état à atteindre (réponse OUI) ou si le compteur atteint le nombre de sommets (réponse NON). Si la réponse au problème d'accessibilité est OUI, il existe une série de transformations qui permet d'atteindre un sommet de F et la réponse sera OUI au moins une fois, ce qui suffit pour que la réponse globale soit OUI dans le cas non-déterministe.

Il est très intéressant de noter qu'une application du problème d'accessibilité se retrouve dans l'analyse des programmes informatiques : on peut voir l'ensemble des états comme l'ensemble des n-uplets de valeurs possibles des variables impliquées dans un programme assortis d'une information sur le point où l'exécution du programme se situe.

On peut alors se demander si une configuration particulière est atteignable, par exemple la dernière ligne du programme (c'est là aussi le problème de l'arrêt, indécidable dans le cas général) ou une valeur particulière pour une des variables.

De manière duale au problème d'accessibilité, le *problème de sûreté* revient à poser la question du problème de décision précédent et de nier la réponse. Il s'agit exactement du problème d'accessibilité dans un contexte où on souhaite éviter les éléments de F.

En ce qui concerne l'analyse des programmes, le problème d'accessibilité et le problème de sûreté peuvent tous deux se rencontrer.

De manière générale, les informaticiens s'intéressent à la complexité et à l'établissement d'algorithmes traitant le problème d'accessibilité dans des structures potentiellement infinies, ce qui se produit lorsqu'on associe par exemple des entiers à un ensemble fini d'états de contrôle. Bien entendu, sur ces mêmes structures, un jeu d'accessibilité pourra se construire, donnant lieu à d'autres problèmes et d'autres algorithmes.

Jeux d'accessibilité

On considère désormais deux sous-ensembles disjoints de S, notés F_A et F_E.

Définition

Un jeu d'accessibilité est un jeu $(\mathcal{G}, \Omega_A, \Omega_E)$ tel que Ω_A soit l'ensemble des parties finies dont le dernier élément est dans F_A et Ω_E soit l'ensemble des parties finies dont le dernier élément est dans F_E. La construction du jeu empêche de continuer une fois F_A ou F_E atteint, de sorte que Ω_A et Ω_E soient bien disjoints.

Il peut exister dans $\mathcal{G}$ des sommets sans arc sortant et qui ne soient ni dans F_A ni dans F_E. Il s'agit alors de sommets de match nul, et les atteindre arrête en particulier la partie.

De même, toute partie infinie est considérée comme nulle, pour les conditions de gain présentées ici.

Une variante des jeux d'accessibilité que nous considèrerons ici revient à ne conserver que F_E comme ensemble particulier, et Ω_E comme condition de gain d'accessibilité pour Ève, en définissant alors Ω_A comme le complémentaire de Ω_E dans S^∞, qui est en quelque sorte une condition de sûreté pour Adam. Dans ce cas, Adam gagne toute partie finie s'arrêtant hors de F_E et toute partie infinie, sachant que là aussi on considèrera qu'atteindre F_E arrête la partie.

Remarquons que le problème d'accessibilité est un cas particulier de jeu d'accessibilité avec $S_A = \emptyset$, en demandant si Ève a une stratégie gagnante pour atteindre F_E depuis un sommet ou l'ensemble des sommets gagnants d'Ève.
De même, le problème de sûreté revient à poser $S_E = \emptyset$, en demandant là aussi si Ève a une stratégie gagnante pour atteindre F_E... sachant qu'elle ne dispose en pratique que d'une stratégie consistant à ne rien faire.

Les jeux d'accessibilité s'adaptent à toute structure plus complexe que les arènes décrites ici, en levant les restrictions imposées par le programme. En particulier, les graphes infinis obtenus par produit cartésien entre un graphe fini et un ensemble $\mathbb{Z}^d$ ont servi de support aux jeux d'accessibilité étudiés au cours de ma thèse.

Attracteur

Dans cette section, nous allons voir comment construire une stratégie gagnante pour Ève dans un jeu d'accessibilité. Comme le point de vue d'Adam est analogue, il n'est pas nécessaire de le mentionner, et cela permet de couvrir également le cas où seule Ève a une condition de gain d'accessibilité, Adam ayant la condition complémentaire.

Premièrement, la victoire d'Ève est immédiate si la partie démarre dans F_E. Ensuite, tout sommet de S_E ayant un arc menant dans un sommet de F_E est gagnant pour elle en un coup.

En ce qui concerne Adam, un sommet de S_A peut être gagnant en un coup **pour Ève**, si **tous les arcs partant de ce sommet** mènent dans un sommet de F_E.

En poursuivant, Ève peut gagner en au plus deux coups depuis un de ses sommets si celui-ci a un arc menant dans un sommet, quel que soit son propriétaire, gagnant en un coup pour Ève voire gagnant tout court pour elle. S'il s'agit d'un sommet d'Adam, il faut là aussi que tous les arcs mènent à un sommet gagnant pour Ève, directement ou en un coup.

La généralisation est alors la suivante : l'ensemble gagnant pour Ève est la réunion croissante pour tout entier naturel n des sommets A_n pour lesquels Ève gagne en n coups, définie par une formule de récurrence donnée ci-après, et cet ensemble est appelé *l'attracteur de* F_E.

$$\mathcal{A} = \bigcup_{n\in\mathbb{N}} \mathcal{A}_n, \text{ avec } \mathcal{A}_0 = F_E \text{ et } \forall n \in \mathbb{N},$$
$$\mathcal{A}_{n+1} = \mathcal{A}_n \cup \{s \in S_E, \exists s' \in \mathcal{A}_n, (s,s') \in T\}$$
$$\cup \{s \in S_A, \exists s' \in S, (s,s') \in T \text{ ET } \forall s' \in S, (s,s') \in T \Rightarrow s' \in \mathcal{A}_n\}$$

En pratique, la formule de récurrence fait que si $\mathcal{A}_{m+1} = \mathcal{A}_m$, alors tous les $\mathcal{A}_n$ pour $n \geq m$ sont égaux à $\mathcal{A}_m$, et donc $\mathcal{A}$ aussi. Comme les $\mathcal{A}_n$ sont des sous-ensembles de S, on en conclut que la suite est stationnaire après au plus autant d'étapes qu'il y a de sommets, ce qui justifie la terminaison de l'algorithme calculant $\mathcal{A}$ et qui s'appuie directement sur la formule de récurrence.

Cet algorithme sera à écrire en TP, avec le calcul de complexité associé.

Théorème

Les sommets gagnants pour Ève sont exactement les sommets de $\mathcal{A}$, avec la construction précédente. Si Adam a également une condition de gain d'accessibilité, l'attracteur de son ensemble de sommets à atteindre F_A est l'ensemble de ses sommets gagnants. En particulier, dans ce cas-là, les deux attracteurs ainsi construits sont disjoints, mais ne couvrent pas forcément tous les sommets.

Ce théorème se prouve en construisant une stratégie positionnelle pour chacun des joueurs sur son ensemble de sommets respectifs, et cette stratégie est gagnante pour Ève sur un sommet si, et seulement si, le sommet est dans $\mathcal{A}$.

Le couple de stratégies est le suivant, sachant que chaque cas est à interpréter comme un « sinon » :

- Pour un sommet sans arc sortant, dont les sommets de F_A, F_E et les autres puits, il n'y a pas de décision à prendre.
- Pour un sommet $s \in S_E \cap \mathcal{A}$, soit $n = \min\{k \in \mathbb{N}, s \in \mathcal{A}_k\}$. Par hypothèse on a $n > 0$ et par construction de $\mathcal{A}_n$ il existe un $s' \in \mathcal{A}_{n-1}$ tel que $(s, s') \in T$, en remarquant que si $n > 1$, il n'existe aucun $i < n - 1$ ni aucun $s' \in \mathcal{A}_i$ tel que $(s, s') \in T$. Alors Ève doit aller dans s'.
- Pour un sommet $s \in S_A \cap \mathcal{A}$, peu importe car tous les sommets accessibles par un arc sont encore dans $\mathcal{A}$.[2]
- Dans le cas où Adam a une condition d'accessibilité, on pose $\mathcal{A}'$ son attracteur construit de manière analogue, et pour un sommet $s \in S_A \cap \mathcal{A}'$, on reprend le premier cas.
- Toujours dans ce même cas, pour un sommet $s \in S_E \setminus \mathcal{A}'$, Ève doit choisir n'importe quel sommet accessible par un arc et qui ne soit pas dans $\mathcal{A}'$, puisque par hypothèse il n'est pas possible de rejoindre $\mathcal{A}$.
- Pour un sommet $s \in S_E \setminus \mathcal{A}$, peu importe là aussi.
- Pour un sommet $s \in S_A \setminus \mathcal{A}$, Adam doit choisir n'importe quel sommet accessible par un arc et qui ne soit pas dans $\mathcal{A}$.

2. L'intuition voudrait qu'Adam décide d'aller dans le successeur commençant à appartenir à $\mathcal{A}_n$ pour le plus grand n possible, afin de retarder sa défaite, mais dans la vie réelle quelqu'un qui sait qu'il perdra si son adversaire joue optimalement peut chercher à jouer un coup qui laisse le plus de mauvais choix possibles à son adversaire, afin de lui donner une occasion de faire une erreur…

Si les deux joueurs choisissent une stratégie qui respecte ces principes, la partie qui en découle sera gagnée par Ève si, et seulement si, elle commence dans un sommet de $\mathcal{A}$, et on pourra plus précisément montrer par récurrence sur l'indice d'un coup dans cette partie que si la partie ne commence pas dans un sommet de $\mathcal{A}$, elle restera à jamais hors de $\mathcal{A}$, donc en particulier hors de F_E, et si elle commence dans un sommet de $\mathcal{A}_n$ pour un certain n, elle atteindra F_E en au plus n coups.

Il en va de même pour Adam s'il a une condition de gain d'accessibilité, et sinon il gagne pour sa condition de sûreté toutes les parties qui ne commencent pas dans un sommet de $\mathcal{A}$.

Il reste à signaler que si un des joueurs choisit une stratégie qui respecte ces principes mais pas son adversaire, le joueur qui joue suivant l'attracteur gagnera tout de même depuis les sommets gagnants du cas précédent mais potentiellement aussi depuis d'autres sommets, en cas d'erreur de l'adversaire pour ainsi dire.

Le couple de stratégies précédent peut être vu comme un bel exemple d'application de la programmation dynamique, et sera également à faire construire en TP.

Il peut être intéressant de construire l'attracteur pour le jeu des bâtonnets, en arrêtant le calcul à 9_J car à ce stade on aura compris la régularité de l'ensemble gagnant.

3.2.3 Algorithme minmax

Contexte

Nous allons rester ici dans la théorie des jeux mais nous éloigner des simples jeux d'accessibilité sur un graphe de taille raisonnable où l'aperçu global de l'arène permet de trouver une stratégie gagnante facilement.

L'algorithme *minmax* que nous allons présenter concernera alors des cas où le calcul de l'attracteur consommerait trop de ressources en temps et en espace pour qu'il soit effectué en pratique, notamment si le graphe de jeu est infini.

Une autre possibilité est qu'à la place d'une condition d'accessibilité le jeu soit associé à un paiement pour les joueurs, et on supposera pour simplifier que tout gain d'un joueur soit une perte pour l'autre (on parle alors de *jeux à somme nulle*).

Ainsi, on pourra introduire la notion de *paiement* d'une partie en tant que gain d'Ève (dont le gain d'Adam se déduit). Le but d'Ève sera de maximiser le paiement et celui d'Adam de le minimiser. Les parties ainsi considérées seront finies.

Dans les deux cas d'étude, on raisonnera sur l'arbre de toutes les parties possibles obtenu à partir du jeu et de la situation de départ.

Dans cet arbre, certains nœuds internes correspondront à une décision qu'Ève doit prendre et les autres à une décision qu'Adam doit prendre.

Les deux joueurs jouent au meilleur de leur intérêt, donc il s'agit de chercher une stratégie optimale, en tant que fonction qui prend en entrée un nœud interne appartenant au joueur et qui retourne un de ses fils.

Les feuilles signaleront l'arrêt de la partie et seront marquées du paiement. Dans le cas de la condition d'accessibilité, ce paiement sera de 1 si la partie s'arrête dans un sommet de F_E, -1 si elle s'arrête dans un sommet de F_A et 0 si elle s'arrête dans un sommet neutre.

Exclure les parties infinies revient ici à forcer une feuille marquée par un paiement nul à partir d'une certaine profondeur, si on peut conclure à l'impossibité pour les deux joueurs de rejoindre un des sommets qu'ils doivent atteindre.

Voyons désormais comment raisonner sur cet arbre si on n'est pas en mesure de le construire totalement.

Notion d'heuristique

L'intuition de la définition d'une *heuristique* a déjà été donnée en première année, lors de la présentation de l'algorithme A*.

Il s'agit d'une méthode de décision ou d'un algorithme pour lequel on privilégie la rapidité d'exécution au prix de l'optimalité de la réponse, par exemple en ne considérant pas toutes les caractéristiques d'un objet mais en lui attribuant une valeur numérique en tant que score, de sorte que le meilleur score soit privilégié.

À la lumière de cette explication, il est naturel de considérer qu'un algorithme glouton est un cas particulier d'heuristique.

Pour autant, toutes les heuristiques ne sont pas des algorithmes gloutons, car elles ne s'interdisent pas d'explorer plusieurs branches.

Les heuristiques sont employées par exemple pour jouer aux échecs, en se servant du score associé à la configuration du plateau après chaque coup possible et en sélectionnant un coup maximisant le score pour le joueur en question.

Ceci se généralise au cas précédent avec l'arbre de toutes les parties possibles. Il s'agit d'anticiper une valeur possible pour chaque nœud interne, selon des principes justifiés par les mathématiques au mieux, au feeling du programmeur au pire.

Pour l'évaluation d'un nœud, une technique courante est la méthode de Monte-Carlo ou ses variantes : pour chaque nœud à une profondeur donnée, un grand nombre de parties est engendré en faisant jouer les deux protagonistes au hasard (ceci suppose que les parties soient assez courte pour que le temps pris soit rentable).

Tous les nœuds à la profondeur en question peuvent alors être considérés comme des feuilles, auxquelles on attribue une valeur qui est la moyenne des paiements des parties ainsi engendrées, et cette valeur peut alors remonter selon le principe du minmax.

Le programme AlphaGo, connu pour avoir battu sans conteste un des meilleurs joueurs de Go du monde dans un match très médiatisé en 2016, s'appuyait dans ses premières versions sur la méthode de Monte-Carlo. Dans un tel cas, le pourcentage de victoires permet d'évaluer un nœud par la méthode de Monte-Carlo.

Pour des parties plus longues, ou pour pouvoir engendrer beaucoup de parties, il est possible de ne faire jouer que quelques coups au hasard et d'évaluer la situation, en considérant par exemple le nombre de pions pris au jeu de dames ou à celui d'Othello, le nombre de pièces restantes pondérées par leur valeur intrinsèque aux échecs, etc. Un exemple plus élaboré a donné lieu à un algorithme suggérant une stratégie pour le Mastermind, dû à Knuth dans les années 1970.

L'algorithme

L'évaluation des nœuds internes est une nouvelle application des techniques en programmation dynamique, en l'occurrence le calcul *bottom-up* des valeurs, qui s'écrit très facilement par récursivité.

Il est en effet possible de considérer le paiement de la partie construite à partir de stratégies optimales en remontant depuis les feuilles, à savoir :

- Le paiement d'une feuille est calculé en amont.
- Le paiement associé à un nœud interne où Adam doit prendre une décision est le plus petit paiement d'un enfant de ce nœud.
- Le paiement associé à un nœud interne où Ève doit prendre une décision est le plus grand paiement d'un enfant de ce nœud.
- Le paiement de la partie est le paiement de la racine de l'arbre. C'est ce qu'on obtient si chaque joueur joue au mieux de ses intérêts en supposant une connaissance globale du contenu de l'arbre.

La notion d'*élagage alpha-beta*, expressément hors-programme, est une accélération du calcul en retirant des branches qui sont certaines de ne pas influencer sur le paiement d'un nœud, par exemple pour un nœud n_0 d'Ève tel que l'un des successeurs, noté n_1, ait un paiement de 73 et l'autre, un nœud d'Adam noté n_2 ait un paiement en cours d'évaluation sachant qu'un des fils de n_2 a un paiement déjà calculé : 42. Alors on sait que le paiement de n_2 sera au plus 42 et qu'Ève ne choisira pas n_2 puisque n_1 a un meilleur paiement. Par conséquent, le calcul du paiement de n_2 est interrompu immédiatement pour gagner du temps.

Cet algorithme s'applique également aux jeux à information imparfaite, d'une part afin de déterminer par exemple par une analyse à cartes connues quel camp remporte une donne au bridge ou dans d'autres jeux de levées [3], d'autre part afin d'établir des stratégies pour un programme devant prendre une décision sans connaître toutes les cartes.

Terminons ce chapitre par une remarque personnelle : les deux thèmes abordés ici étaient l'intelligence artificielle et les jeux, mais très peu de choses sont à savoir sur la création d'intelligences artificielles pour les jeux, notamment ceux où l'apprentissage et l'utilisation de réseaux de neurones serait bien trop excessifs.

Il en va alors comme face à tout problème informatique : c'est au programmeur de montrer qu'il sait résoudre à la main une instance, et donc élaborer lui-même sa propre stratégie au cours d'une partie, puis de formaliser et de généraliser ceci en un programme bien structuré.

3. et ici un élagage est nécessaire tout comme la fusion de cas équivalents (par exemple des cartes consécutives d'une même main qui peuvent être jouées indifféremment l'une à la place de l'autre sans changer le cours de la donne)

Deuxième partie

Travaux pratiques

TP 0 : Programmation avancée

En attendant d'avoir matière à réaliser un TP sur le début du cours et pour reprendre contact avec Python en début d'année scolaire, voici un TP optionnel présentant des techniques de programmation utiles pour un usage personnel de Python, mais a priori superflues dans l'optique des concours.

1 Fonctions anonymes

Il est possible que des fonctions rencontrées en première année, notamment dans l'utilisation de l'informatique dans d'autres matières, avaient au moins une autre fonction parmi leurs arguments. Or, une telle fonction en argument pouvait être une expression relativement simple, par exemple une fonction polynomiale de bas degré.

Quand il s'agissait de l'exponentielle, par exemple, tout allait bien, on pouvait écrire `exp` comme argument, ce nom étant associé à un objet *appelable* ("callable").

Dans le cas général, pour ne pas être obligé d'écrire une ligne de définition pour une expression courte et utilisée une seule fois (conditions à respecter pour ne pas tout compliquer ni écrire du code moche), on peut recourir aux fonctions anonymes.

La syntaxe est la suivante : `lambda x1, x2, ..., xn : expression` s'évalue en une fonction à n arguments et ayant une valeur de retour qui en dépend.

Par exemple, la fonction carré peut s'écrire `lambda x : x ** 2`. Il est possible de stocker ceci dans une variable, mais presque tout le monde verra d'un mauvais œil la première de ces deux syntaxes équivalentes :

```
carre = lambda x : x ** 2

def carre(x):
  return x ** 2
```

Encore pire, on peut (et on ne doit pas) écrire en Python `(lambda x : x ** 2)(42)`. Encore encore pire : `(lambda x : (lambda y : x + y))(30)(12)`.

2 Ensembles

La structure d'ensemble a été rapidement mentionnée dans le chapitre 0 du cours de première année. Il s'avère qu'en Python un ensemble est syntaxiquement proche d'un dictionnaire. Dans les deux cas, le délimiteur est une accolade, mais les éléments d'un ensemble sont quelconques alors que ceux d'un dictionnaire s'écrivent `clef : valeur`, avec unicité des clés.

Les ensembles sont apparus tardivement en Python, et la syntaxe mentionnée ci-avant date de la version 3. D'ailleurs, l'ensemble vide doit encore être écrit `set()` car `{}` est le dictionnaire vide, qui n'est pas une instance du type ensemble.

Avec des noms intuitifs, voici quelques opérations fondamentales (à côté de `len` et `in` / `not in` déjà connues) :

```
E.add(x)
E.remove(x)
E.pop() # le premier élément de E est retiré et renvoyé
E1.issubset(E2) # c'est E1 qui est un sous-ensemble
E1.issuperset(E2)
E1.union(E2)
E1.intersection(E2)
E1.difference(E2)
E1.symmetric_difference(E2)
E.copy()
```

Pour rappel, un ensemble ne supporte pas l'indexation, donc on ne peut pas accéder à un élément particulier d'un ensemble, cependant on peut le parcourir (l'ordre n'est pas nécessairement l'ordre donné à la création) à l'aide d'un `for` comme on le ferait avec d'autres structures.

Le chapitre 2 présente la notion de hachage, et à la lumière du cours on comprendra qu'un ensemble n'est pas hachable. Pour cette raison, Python dispose aussi de la notion de *frozen set*, c'est-à-dire un ensemble qui n'est pas mutable et qui en devient alors hachable. Il se crée par la fonction `frozenset` et les fonctions de manipulation d'ensembles créent le résultat sans faire les effets de bord des fonctions précédentes.

3 Listes en compréhension

On peut créer une liste en compréhension, c'est-à-dire en mettant entre crochets un code raccourci qui aurait provoqué l'ajout d'éléments à la liste.

Dans ce cas, si on simule une double boucle, il faut faire attention à l'ordre d'écriture. Il est par ailleurs également possible de mettre d'utiliser des tests conditionnels.

```
# Deux façons de créer [1,3,3,3,5,5,5,5,5,7,7,7,7,7,7,7].
# On pourrait aussi utiliser range(1,9,2).

l = []
for i in range(8):
    if i % 2 == 1:
        for j in range(i):
            l.append(i)

ll = [i for i in range(8) if i % 2 == 1 for j in range(i)]
```

Cette syntaxe se rapproche de l'écriture mathématique des ensembles, elle peut fournir une alternative intéressante par sa concision (voire être plus facile à comprendre).

Attention : les deux codes présentés ne sont pas équivalents. En effet, dans le premier cas, les variables `i` et `j` ont une existence en-dehors de la boucle (c'est ainsi en Python), mais dans le deuxième cas elles n'existent pas en-dehors du crochet.

4 Les mots-clés break et continue

Les deux mots-clés présentés ici, et déjà mentionnés dans les TP du premier semestre en première année, sont à utiliser le moins souvent possible indépendamment du langage (presque aussi rarement que `goto`, qui est un tabou absolu heureusement absent de Python). Cependant, autant savoir qu'ils existent.

L'instruction `break` permet d'interrompre une boucle (la plus interne) immédiatement. L'interruption concerne bien une boucle et non un bloc, bien que le bloc soit également court-circuité dans la foulée. Pour sortir de plusieurs boucles à la fois, on utilisera si possible des techniques plus propres...

L'instruction `continue` permet d'aller directement au tour suivant d'une boucle (la plus interne), avec la même remarque.

Tester par exemple les codes ci-après :

```
for i in range(9):
    for j in range(9):
        print(i, j)
        if j == 5:
            break
            print("Ceci ne sera jamais lu")
        print("Ceci sera lu pour j < 5")

for i in range(9):
    for j in range(9):
        print(i, j)
        if j == 5:
            continue
            print("Ceci ne sera jamais lu")
        print("Ceci sera lu pour j != 5")

# Attention, horreur en approche !
for i in range(9):
    for j in range(9):
        print(i, j)
        if i == j == 5:
            break
            print("Ceci ne sera jamais lu")
        print("Ceci sera lu jusqu'à (5, 5)")
    else: # ignoré si et seulement s'il y a eu un break
        continue # seul moyen d'éviter le break suivant
    break # et donc le break précédent est répercuté
```

Une interruption encore plus violente est `exit`, mais il s'agit d'une fonction (donc suivie de parenthèses). Au contraire, le mot-clé `pass` se contente de ne rien faire. Il apparaît éventuellement pour éviter des erreurs de syntaxe, mais on peut toujours faire aussi proprement sans.

```
if n == 42:
    pass
else:
    print("Ce n'est pas la réponse !")

if n != 42: # équivalent
    print("Ce n'est pas la réponse !")
```

5 La fonction map

La fonction `map` s'applique à une fonction et à un itérable, et elle renvoie un « objet map » (quelqu'un qui reçoit une liste est en train d'utiliser la version 2 de Python) qui contient toutes les images des éléments de l'itérable par la fonction. L'ordre de parcours est préservé (donc attention aux ensembles, notamment).

Un objet engendré par `map` est itérable, mais non indexable et on ne peut pas non plus récupérer sa taille à l'aide de `len`, bien que la taille soit en pratique celle de l'objet de départ (les doublons ne sont pas supprimés). Pour cette raison, on appellera souvent la fonction `list` après avoir appelé `map`.

6 Générateurs

Nous avons déjà vu qu'une fonction ne pouvait avoir qu'une valeur de retour. Pour autant, il peut être intéressant de disposer de fonctions qui renvoient des objets « au fur et à mesure », ne serait-ce que pour économiser l'espace mémoire nécessaire pour stocker une liste énorme qui serait retournée, donc si toutes les valeurs étaient fournies d'un coup.

Python dispose en fait d'objets dits générateurs, qui sont engendrés par des fonctions utilisant non pas `return`, mais le mot-clé `yield` (produire, en anglais), qui renvoie un objet tout en n'interrompant pas l'exécution de la fonction.

Cette exécution n'est en pratique effectuée que quand on cherche l'objet suivant, par exemple au moment de parcourir un générateur.

Ainsi, simplement appeler une fonction qui retourne un générateur ne prend pas de temps, même si un million d'objets étaient à engendrer en cas de parcours.

Sans entrer dans les détails concernant le fonctionnement des générateurs, il faut savoir qu'une fonction ne peut pas avoir à la fois des `return` et des `yield`, même si la spécification garantit qu'un des deux n'est jamais rencontré (allez l'expliquer à Python !).

De plus, tout comme `return`, il est interdit d'écrire `yield` en-dehors d'une fonction.

On remarquera qu'en Python, de nombreux objets sont en fait des générateurs notamment les objets map et les range. Ceci explique l'accélération du code quand on fait un range sur un intervalle énorme.

7 Fonctions avec variables optionnelles

Au moment de définir une fonction, il est possible de rendre certaines des variables optionnelles en précisant une valeur par défaut si lors de l'appel de la fonction ces arguments ne sont pas fournis.

Mieux que cela : les noms de variables retenus pour chaque argument permettent lors de l'appel de la fonction de fournir les arguments dans l'ordre que l'on souhaite, à condition de préciser quel argument on est en train de renseigner.

Pour éviter toute ambiguïté, des règles s'imposent, notamment le fait que les arguments dits nommés figurent uniquement après les arguments non nommés, qui doivent, quant à eux, être fournis dans l'ordre et donc être les premiers arguments sans interruption.

Un exemple très classique de fonction avec variables optionnelles est `range`, et on peut observer des arguments nommés dans la fonction `print`.

Pour fixer les idées, un code permet de comprendre la fonctionnalité présentée ici.

Exécuter les lignes suivantes et deviner avant exécution le résultat donné par Python.

```
def poly(a=1, b=1, c=1):
    return a**3 + b**2 + c

poly()
```

```
poly(2)

poly(c=10, a=20, b=1)

poly(3, c=10, b=5)

poly(3, c=0)

poly(3, a=2, c=4) # Spoiler : erreur

poly(c=3, 2, a=-1) # Erreur aussi
```

Au passage, une astuce classique consiste à bricoler les arguments pour compenser leur absence, notamment en leur donnant des valeurs par défaut exceptionnelles, comme `None`.

Puisqu'on parlait de `print`, il a également été constaté qu'on peut donner un nombre arbitraire d'arguments à une fonction. Ceci utilise une syntaxe particulière, à l'aide d'un symbole * avant un nom d'argument.

En pratique, l'idée est que les arguments sont stockés dans une liste et la déconstruction permet elle-même de récupérer directement un nombre arbitraire d'éléments d'un itérable.

Seul un nom peut être précédé du symbole * pour éviter toute ambiguïté, et dans la définition d'une fonction ces arguments doivent être après les arguments obligatoires, de préférence après les arguments optionnels aussi.

Comparer :

```
l = list(range(42))

a, b, *c = l

print(a) # 1
print(b) # 2
print(c) # la liste du reste
```

```
def multadd(coeff=1, *valeurs):
    somme = 0
    for x in valeurs:
        somme += coeff*x
    return somme

multadd(1, 2, 3, 4, coeff=5) # erreur car le coefficient
# est déjà considéré comme la première valeur
multadd(1, 2, 3, 4, 5) # 14


def addmult(*valeurs, coeff=1):
    somme = 0
    for x in valeurs:
        somme += coeff*x
    return somme

addmult(1, 2, 3, 4, coeff=5) # 50, pas de souci
addmult(1, 2, 3, 4, 5) # 15 car coeff est considéré comme omis
```

Au passage, si on avait défini une fonction `temps_execution` pour mesurer le temps d'exécution d'une fonction, on observait que l'écriture imposait le nombre d'arguments de la fonction dont on voulait évaluer le temps d'exécution. Avec la syntaxe nouvellement définie, on peut écrire simplement :

```
from time import time

def temps_execution(fonction, *arguments):
    avant = time()
    fonction(*arguments)
    return time() - avant
```

8 Formats pour l'impression

Python dispose de nombreuses syntaxes pour imprimer des valeurs paramétrées. Celle qui est utilisée spontanément revient à utiliser `print` avec autant d'arguments qu'on le souhaite, en alternant les chaînes de caractères constantes et les variables.

Elle est plus pratique que de construire une chaîne par additions, ce qui est une façon plus coûteuse en raison du coût de l'opérateur `+`.

Une troisième syntaxe est commune à de nombreux langages et héritée du langage C : elle revient à utiliser une sorte de joker dont le type est précisé en plein milieu d'une chaîne, ce joker étant introduit par le symbole `%` (comme pour modulo). Ainsi, l'opérande de gauche est interprété en fonction de l'opérande de droite, qui est un objet quelconque ou un n-uplet contenant autant d'éléments que requis par les jokers apparaissant dans la chaîne, bien entendu dans l'ordre d'apparition.

Pour produire un entier, le joker est `%d` (car l'entier est écrit en décimal, mais on peut aussi utiliser `%i` s'il est signé). Pour un flottant, c'est `%f`, pour un caractère seul (surtout pratique si on donne la position dans la table de caractères du caractère en question), c'est `%c`, pour une chaîne de caractères, c'est `%s`... et pour imprimer un `%`, il faut bien trouver une solution, c'est simplement d'écrire `%%` (donc ne demandant pas à fournir un argument).

Une quatrième syntaxe est spécifique à Python, il s'agit de la méthode `format` appelée sur une chaîne comprenant des accolades et dont les autres arguments sont des valeurs disponibles. Entre les accolades, il est possible de ne jamais rien mettre, auquel cas les valeurs sont imprimées dans l'ordre d'appel, ou alors on peut mettre des entiers correspondant aux indices, dans l'ordre qu'on veut et en réutilisant un indice aussi souvent qu'on le souhaite, auquel cas les valeurs aux positions correspondantes, entre zéro et le nombre de valeurs disponibles moins un, sont imprimées. Ceci s'étend même à des arguments nommés :

```
"{} {} {}".format(1, 2, 3) # "1 2 3"
"{1} {0} {1}".format(1, 2, 3) # "2 1 2"
"{a} {c} {b}".format(a=1, b=2, c=3) # "1 3 2"
"{b} {b} {c}".format(a=1, b=2, c=3) # "2 2 3"
```

TP 1 : Python et SQL

Dans ce TP, nous allons simuler les requêtes les plus simples du langage SQL en Python. L'essentiel du travail effectué reprend le sujet d'informatique B du concours X-ENS 2018.

Concernant l'administration effective d'une base de données en Python, on pourra consulter la documentation de SQLite, et un autre lien explicatif figure ici :
`https://python.doctor/page-database-data-base-donnees-query-sql-mysql-postgre-sqlite`

La plupart des IDE de Python intègrent un moteur de bases de données, en pratique.

Consigne importante pour l'ensemble du TP : On considère des tables vues comme des listes en Python, contenant elles-même des listes (ce qui ne sera pas rappelé par la suite), une par enregistrement dans la table, dont les attributs seront renseignés dans l'ordre correspondant au schéma de la table supposé connu.

Pour simplifier, les valeurs seront obligatoirement des chaînes de caractères ou des nombres.

La base de données considérée sera celle du TD 1. Pour rappel, elle contient trois tables, dont les schémas ci-dessous imposeront l'ordre des attributs :

- `Etudiants`, attributs `Id`, `Nom`, `Prenom` et `Classe`.
- `Examens`, attributs `Id`, `Date` et `Coeff`.
- `Notes`, attributs `Etudiant`, `Examen` et `Note`.

Exercice 1 : Écrire une fonction `selection_constante(table, indice, valeur)` qui prend en entrée la représentation d'une table et qui retourne la liste des enregistrements pour lesquels l'attribut à l'indice demandé est égal à la constante fournie.

Par exemple, si la table `Notes` contient entre autres `(1, 1, 16)` et `(1, 2, 13)`, la fonction `selection_constante` appelée sur cette table avec l'argument `indice` valant `2` et l'argument `valeur` valant `16` renverra une liste contenant l'enregistrement représenté par `(1, 1, 16)` mais pas l'enregistrement représenté par `(1, 2, 13)`.

On attend donc un résultat similaire entre `SELECT * FROM t WHERE att = valeur` et `selection_constante(tp, i, valeur)` où `tp` est la liste en Python correspondant à la table `t` et `i` est l'indice de l'attribut `att` dans le schéma de `t`.

Exercice 2 : Écrire une fonction `selection_egalite(table, indice1, indice2)` qui prend en entrée la représentation d'une table en tant que liste et qui retourne la liste des enregistrements pour lesquels les deux attributs aux indices demandés ont la même valeur.

Exercice 3 : Écrire une fonction `projection(table, indices)` qui prend en entrée la représentation d'une table en tant que liste et qui retourne la liste, de même taille, des enregistrements dont seuls les attributs aux indices demandés sont fournis. On supposera que la liste `indices` est strictement croissantes et que ses valeurs seront comprises entre `0` inclus et le nombre total d'attributs exclu.

Écrire d'abord une fonction qui fait le travail de projection sur un seul enregistrement.

Exercice 4 : Écrire une fonction `produit_cartesien(table1, table2)` qui prend en entrée la représentation de deux tables (potentiellement les mêmes) en tant que listes et qui retourne leur produit cartésien.

La requête ici simulée est `SELECT * FROM t1 JOIN t2` sans contrainte.

Exercice 5 : Écrire une fonction `jointure(table1, table2, indice1, indice2)` qui prend en entrée la représentation de deux tables en tant que listes ainsi que deux indices d'un attribut de chaque table respective et qui retourne la jointure symétrique suivant l'égalité des attributs aux indices précisés.

Pour rendre les choses intéressantes, on ne mettra pas dans les enregistrements de la jointure l'attribut de la deuxième table qui est égal à celui de la première !

La requête ici simulée est `SELECT * FROM t1 JOIN t2 ON att1 = att2` suivie d'une projection simplifiant la redondance, en considérant le nom des attributs aux indices concernés.

Exercice 6 : Écrire une fonction `supprimer_doublons(table)` qui prend en entrée une liste et qui renvoie la même liste dont tous les doublons sont supprimés.

Exercice 7 : Déterminer la complexité des fonctions des exercices de 1 à 6 en fonction de la taille des tables et du nombre d'attributs qu'elles ont.

Pour les quatre exercices suivants, il faudra écrire une requête SQL répondant à la question puis une fonction en Python, sans argument, qui retournera le même résultat. On considèrera alors pour ces fonctions trois variables globales ETUDIANTS, EXAMENS et NOTES, les listes correspondant aux tables de la base de données

Exercice 8 : Trouver la note de l'étudiant d'identifiant 42 à l'examen d'identifiant 5.

Exercice 9 : Trouver le nom et le prénom de tous les étudiants qui ont eu 20 à l'examen numéro 1.

Exercice 10 : Trouver le nombre d'étudiants ayant eu au moins une fois 0. La fonction `len` pourra être utilisée en Python.

Exercice 11 : Trouver l'ensemble des classes où au moins un étudiant a eu 19 à l'examen du 19 décembre 2015.

TP 2 : Bases du hachage

Ce TP a pour but de simuler des tables de hachage et de découvrir les grands principes et les problématiques associés. Il s'agit alors de recréer la théorie soi-même à partir d'observations guidées.

Exercice 1 : Télécharger le fichier de faux-texte disponible sur `http://jdreichert.fr/Enseignement/CPGE/Divers/lorem_ipsum.txt`. L'ouvrir avec Python, récupérer son contenu et le découper en la liste des mots, une fois la ponctuation (points, virgules et points-virgules) retirée. Tout mettre en bas de casse puis retirer les doublons de cette liste avec une méthode au choix.

Il faudra conserver le résultat de l'exercice 1, il s'agira de l'ensemble (de taille `127`) des clés de tous les dictionnaires de chaînes de caractères.

Exercice 2 : Engendrer une liste de mille entiers aléatoires distincts entre `0` et `999999`. Il s'agira de l'ensemble des clés de tous les dictionnaires d'entiers. En profiter pour relever combien il a fallu d'essais pour ne plus avoir de doublons (suivant la méthode, et si l'absence de doublons était garantie, commenter la complexité).

Exercice 3 : Engendrer une liste de mille flottants aléatoires entre `0` (inclus) et `1000000` (exclu), en gardant six chiffres après la virgule.

Pour les exercices 4 à 7, on utilisera le chaînage pour stocker les valeurs dans les tables de hachage, donc les listes d'appui contiendront des listes. Il s'agira de détecter les collisions dans tous les cas.

Exercice 4 : Trouver une fonction bijective de l'ensemble des chaînes de lettres en bas de casse de taille deux vers l'ensemble des entiers de `0` à `675` inclus. L'écrire en Python. Utiliser cette fonction comme fonction de hachage pour la liste de l'exercice 1. Que remarque-t-on ?

Exercice 5 : Écrire une fonction « d'évaluation polynomiale d'une chaîne de lettres en bas de casse », en assimilant la lettre `a` à la valeur `1` (on pouvait aussi prendre `0`) et en considérant la première lettre comme plus haut degré (de même, on pouvait décider que c'est le plus bas degré). Prouver qu'on a bien une fonction injective (ce n'est pas le cas si on pose `a` valant `0`). Écrire ensuite une fonction prenant en argument supplémentaire un entier `n` et calculant le résidu de la fonction précédente modulo `n`.

Utiliser cette fonction comme fonction de hachage pour la liste de l'exercice 1 avec diverses valeurs de `n`, en s'intéressant particulièrement aux nombres premiers. Que remarque-t-on ?

Pour les exercices 6 et 7, il s'agira de mettre en commun les remarques avec des listes aléatoires différentes entre tous les étudiants. Il est aussi envisageable pour quelqu'un de recommencer l'expérience avec de nouvelles listes aléatoires.

Exercice 6 : Stocker la liste de l'exercice 2 dans des tables de hachage de tailles diverses en prenant pour fonction de hachage le résidu modulo la taille de la table de hachage. Que remarque-t-on ?

Exercice 7 : Stocker la liste de l'exercice 3 dans des tables de hachage de taille un million en prenant pour fonction de hachage la partie fractionnaire multipliée par `1000000`, arrondie à l'entier le plus proche en raison des soucis avec les flottants. Que remarque-t-on ? Même consigne avec la partie entière comme fonction de hachage.

Exercice 8 : Reprendre les exercices précédents en gérant les collisions par adressage ouvert.

Exercice 9 : Les tailles `t` des ensembles de clés et `n` des dictionnaires étaient liées par la formule $\texttt{n} = \texttt{t}^2$ dans l'exercice 7. Se renseigner sur le paradoxe de von Mises et vérifier par le calcul (assisté par ordinateur...) pour des valeurs raisonnables de `n` à partir de quelle valeur de `t` la probabilité d'une collision dépasse un demi.

Exercice 10 : Pour limiter les risques de collision, une fonction de hachage plus pertinente a été développée par Glenn Fowler, Landon Curt Noll et Kiem-Phong Vo dans les années 1990. Son écriture est par ailleurs relativement simple. Implémenter cette fonction pour des chaînes de caractères et constater l'optimisation (ou pas).

L'algorithme est le suivant pour les chaînes de caractères : soit `x` valant initialement `14695981039346656037`. Prendre chaque caractère `c` de la chaîne [4] et remplacer `x` par le « ou exclusif bit à bit » de lui-même avec `c`, puis par les soixante-quatre derniers bits du produit du résultat par le nombre premier `1099511628211`. Le haché est le résultat final, sur soixante-quatre bits. Autant dire que la taille de la table de hachage est rédhibitoire... mieux vaut ne pas créer un tableau d'appui.

4. vu comme son code ASCII sur huit bits

Une variante de l'algorithme revient à échanger l'ordre entre le produit par le nombre premier (aussi ramené à 64 bits) et le ou exclusif bit à bit.

Pour faire les tests, nous allons prendre soixante-dix-mille valeurs et considérer la version à trente-deux bits de l'algorithme, ce qui utilise pour valeur initiale de `x` le nombre `2166136261` et pour nombre premier `16777619`. Ainsi, puisque le nombre de valeurs à placer est de l'ordre de la racine du nombre de valeurs possibles, on pourra constater un taux de collisions pertinent.

Exercice 11 : Écrire une fonction qui prend en argument une table de hachage et qui la mute de sorte que sa taille change. Les arguments supplémentaires seront alors la nouvelle taille et la nouvelle fonction de hachage. On pourra faire la version avec chaînage ou avec adressage ouvert.

Ce dernier exercice peut être très artificiel, car une version revient simplement à reconstituer la liste des clés, puis à créer la nouvelle table de hachage qu'on recopie dans l'ancienne après destructions. Il y a certes un effet de bord, mais on ne peut pas appeler cela du travail en place. Pour autant, une version qui mute sur place serait possible dans le cadre du chaînage au prix de la gestion des éléments, qu'il faut reconnaître comme anciens ou nouveaux car les deux sont amenés à cohabiter si on ne veut pas consommer d'espace supplémentaire, ne serait-ce que pour stocker la liste des clés.

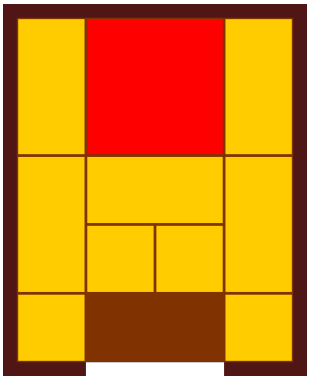
TP 3 : Dictionnaires appliqués en Python

Ce TP constitue une révision d'algorithmes de première année en forçant l'utilisation de dictionnaires pour chaque exercice (cette consigne ne sera pas rappelée mais est à respecter systématiquement).

Exercice 1 : Écrire en Python une fonction prenant en argument une liste de listes de nombres et déterminant l'élément le plus fréquent.

Exercice 2 : Écrire en Python une fonction réalisant un parcours en profondeur d'un graphe donné par sa représentation en liste d'adjacence et stocké dans un dictionnaire. On la testera sur un graphe créé en Python à partir d'un exemple dessiné au tableau. La fonction prendra en particulier un sommet de départ et un sommet d'arrivée en entrée et déterminera un chemin de l'un à l'autre.

Exercice 3 : En guise d'application de l'exercice précédent, écrire en Python une fonction résolvant une instance de l'âne rouge, présenté ci-après.



Le jeu de l'âne rouge consiste à faire se déplacer une pièce particulière (en rouge) à l'extérieur d'un carré ouvert uniquement au milieu dans la ligne du bas.

Les seuls mouvements possibles consistent à glisser des pièces d'une unité de déplacement horizontalement ou verticalement, toujours une à une et sur une case libre.

Comme le montre cette figure, la zone de jeu fait quatre unités de largeur pour deux unités de hauteur, ce qui donne cependant un graphe dont le nombre de sommets a cinq chiffres et le degré moyen entre trois et quatre.

La modélisation ici proposée est de numéroter chacune des pièces par un entier différent, et les cases libres par un autre entier, les configurations étant alors des séquences de vingt entiers, qui ne peuvent pas être des listes de listes car les listes n'ont pas le droit d'être des clés dans un dictionnaire.

Une variante du jeu de l'âne rouge, souvent illustrée avec des voitures dans un parking, implique des pièces de forme rectangulaire, de longueur supérieure ou égale à deux et de largeur un, dont les mouvements sont obligatoirement sur une droite, qui correspondrait au sens de la longueur.

Exercice 4 : Écrire en Python une fonction calculant le nombre de schémas de déverouillage de téléphone possibles en fonction de la position de départ et de la taille (dont on admettra qu'il s'agit d'un entier entre `4` et `9`) du schéma. Un schéma de déverouillage consiste à partir d'une des positions possibles d'un carré de taille trois et d'aller en ligne droite vers d'autres positions, relevées une et une seule fois. Il est alors interdit d'ignorer une position par laquelle on passerait pour la première fois, autrement dit la position en haut à gauche ne peut être suivie de la position en bas à gauche que si la position au milieu à gauche a déjà été visitée. En revanche, on peut repasser sur une position déjà relevée à condition d'être dans un cas tel que celui présenté à la phrase précédente.

Ces trois derniers exercices sont une occasion de réviser les graphes en tronc commun, revus par ailleurs à peu près au même moment en option.

Il est intéressant de remarquer que dans certains cas, le graphe se stocke en mémoire au fur et à mesure de sa découverte, par l'application de règles, par exemple des fonctions, déterminant les sommets voisins d'un sommet en cours d'exploration. Les sommets ainsi visités peuvent alors être stockés dans un dictionnaire (en tant que clés, les valeurs étant alors l'ensemble des voisins), et ce même dictionnaire permettra de ne pas visiter un sommet plusieurs fois. Cette méthode peut s'appliquer au jeu de l'âne rouge en écrivant une fonction annexe donnant la liste des déplacements possibles à partir d'une certaine configuration, ainsi qu'à l'exercice sur le déverouillage du clavier, en gérant l'évolution des mouvements possibles à partir d'une position en fonction des positions vues précédemment.

Reste à voir si la mémoire ne sature pas dans de tels cas. Le jeu du taquin par exemple, dispose d'un nombre factoriel de sommets possibles, ce qui exclut le parcours au profit d'une stratégie de résolution.

Exercice 5 : Écrire en Python une fonction prenant en argument un entier entre `1` et `3999` et qui retourne son écriture en chiffres romains. Écrire aussi sa réciproque.

Exercice 6 : Écrire en Python une fonction prenant en argument une chaîne de caractères formée des lettres `A`, `T`, `C` et `G`, donc une portion d'ADN, et qui construit successivement sa transcription en ARN messager (en tant que chaîne de caractères) puis la protéine obtenue (en tant que liste d'acides aminés), sachant que si aucune protéine ne peut être obtenue la liste retournée sera vide.

Petit rappel de biologie, avec des abus et erreurs éventuelles justifiés par l'algorithmique à travailler pour cet exercice :

- La transcription de l'ADN en ARN messager revient à substituer les bases azotées (les caractères, informatiquement) par leur complémentaire, selon la règle `A` $\to$ `U`, `T` $\to$ `A`, `C` $\to$ `G` et `G` $\to$ `C`.
- La construction de la protéine revient à étudier des séquences de trois bases azotées de l'ARN messager, en commençant par détecter ce qu'on appelle un codon initiateur (une séquence référencée, unique chez les eucaryotes, pas forcément pour cet exercice), au-delà duquel toutes les séquences de trois bases azotées, jusqu'à une séquence particulière appelée codon stop (il y en a plusieurs) seront traduites en un acide aminé en fonction d'un dictionnaire donnant les correspondances, ce dictionnaire pouvant être une variable globale et contenant également l'information des codons initiateurs et stop d'une manière ou d'une autre. Le codon stop, comme le codon initiateur, ne sont pas inclus dans la protéine.
- Remarque importante : il ne faut pas lire trois par trois les bases azotées de l'ARN messager dès le début, mais chercher la première occurrence d'un codon initiateur.

TP 4 : Programmation dynamique

Dans ce TP, des applications de la programmation dynamique sont présentées.

Exercice 1 : Écrire un programme dynamique déterminant le nombre minimal de multiplications de scalaires à faire pour multiplier n matrices de dimensions variées.

Par exemple, pour calculer $M_1M_2M_3M_4$, où les dimensions respectives des matrices sont $(4,6)$, $(6,2)$, $(2,10)$ et $(10,3)$, les multiplications peuvent être faites ainsi :
- $M_1(M_2(M_3M_4))$ ($2\times10\times3+6\times2\times3+4\times6\times3$ soit 168 multiplications) ;
- $(M_1M_2)(M_3M_4)$ ($4\times6\times2+2\times10\times3+4\times2\times3$ soit 132 multiplications) ;
- $M_1((M_2M_3)M_4)$ ($6\times2\times10+6\times10\times3+4\times6\times3$ soit 372 multiplications) ;
- $((M_1M_2)M_3))M_4$ ($4\times6\times2+4\times2\times10+4\times10\times3$ soit 248 multiplications) ;
- $(M_1(M_2M_3))M_4$ ($6\times2\times10+4\times6\times10+4\times10\times3$ soit 480 multiplications) ;

Pour information, le nombre de façons d'organiser les multiplications de n matrices est le n-ième nombre de Catalan, donné par les formules équivalentes

$$C_n = \frac{\binom{2n}{n}}{n+1} = \frac{(2n)!}{n!(n+1)!} = \binom{2n}{n} - \binom{2n}{n-1}.$$

Le nombre de Catalan se retrouve très souvent en combinatoire avancée.

Exercice 2 : Écrire un programme dynamique pour le problème du rendu de monnaie dans le cas général. Ce problème avait déjà été présenté dans le TP sur les algorithmes gloutons.

Exercice 3 (Ordonnancement de tâches pondérées - *weighted interval scheduling*) : Écrire un programme dynamique déterminant la valeur maximale d'un ensemble de tâches effectuables, les tâches étant définies par leur heure de début, leur heure de fin et leur valeur, et une seule tâche pouvant être accomplie à la fois.

Exercice 4 (Distance d'édition) : Soient deux chaînes de caractères s_1 et s_2. La distance d'édition de s_1 à s_2 correspond au coût minimal pour passer de s_1 à s_2 (coût qui n'est pas symétrique a priori) en nombre d'insertions de caractères, de modifications de caractères et de suppressions de caractères (n'importe où dans les trois cas). On associe à chacune de ces opérations un coût constant, mais pas nécessairement égal entre les opérations (on considère dans le cas simple que les coûts sont tous d'un). Écrire un programme dynamique qui calcule la distance d'édition entre deux chaînes de caractères dans le cas simple puis dans le cas général.

Exercice 5 : Écrire un programme dynamique pour le problème du sac à dos. Écrire par ailleurs un programme glouton et constater qu'il n'est pas optimal dans certains cas.

Le problème du sac à dos est le suivant : étant donné un ensemble d'objets d'un certain poids et d'une certaine valeur, comment remplir un sac à dos avec un ou plusieurs exemplaire(s) (certaines variantes excluent cependant d'en prendre plusieurs) de certains objets de sorte que la valeur transportée soit maximale et le poids soit inférieur à un seuil donné ?

L'argument de la fonction sera donc une liste de couples (poids puis valeur pour se mettre d'accord) et la valeur de retour la liste des objets retenus. Une autre possibilité est d'ajouter un nom aux couples, pour ne renvoyer que la liste des noms des objets retenus.

Le problème demande normalement de renvoyer la valeur maximale sans détailler les objets à transporter, mais le programme devra retourner les deux (ou au moins la liste des objets, dont poids et valeur seront déduits).

Exercice 6 (long, à faire chez soi) : Écrire un programme dynamique pour résoudre toutes les instances du jeu 24, détaillé ci-après.

Le jeu 24 se rapproche du coup de chiffres de l'émission « Des chiffres et des lettres ».

Seule la partie qui a un intérêt algorithmique est présentée ici (pour en dire légèrement plus, on utilise les 40 cartes numériques d'un paquet de 52 dans la pratique) : il s'agit d'obtenir le nombre `24` grâce aux quatre opérations arithmétiques usuelles (d'autres sont possibles dans des variantes) en partant de quatre entiers entre `1` et `10` fournis à l'avance et non nécessairement distincts.

Il est obligatoire d'utiliser les quatre entiers fournis, et il est autorisé de passer par des nombres négatifs voire non entiers.

En l'occurrence, comme les nombres resteront rationnels (hors variante...), on utilisera des couples d'entiers représentant tous les nombres impliqués, en tant que fraction simplifiée (le premier élément du couple représentera le numérateur et le deuxième, nécessairement strictement positif et premier avec l'autre, représentera le dénominateur).

Le travail se décomposera en plusieurs étapes :

- Écrire les opérations arithmétiques sur les couples d'entiers en tant que fonctions annexes.
- Créer un dictionnaire `d` dont les clés seront tous les 1-uplets, couples, triplets et quadruplets d'entiers (croissants pour éviter l'ambiguïté) entre `1` et `10` et les valeurs seront des dictionnaires vides.
- Initialiser les valeurs dans `d` pour tous les 1-uplets `(k, )` en y mettant `{ (k, 1) : "k"}`, signifiant qu'on peut obtenir le nombre `k`, représenté par le couple `(k, 1)`, en écrivant simplement ce nombre.
- En déduire les valeurs dans `d` pour tous les couples formés à partir de deux 1-uplets par l'application des opérations élémentaires.
- En déduire les valeurs dans `d` pour tous les triplets formés à partir d'un couple et d'un 1-uplet de la même façon.
- En déduire les valeurs dans `d` pour tous les quadruplets formés à partir de deux couples ou d'un triplet et d'un 1-uplet, mais cette fois-ci seul le couple `(24, 1)` sera intéressant à stocker.
- Retourner `d` dont seuls les quadruplets seront conservés en tant que clés.

TP 5 : Applications de kNN et k-means

Ce TP présente quelques problèmes simples où les algorithmes des k plus proches voisins et des k-moyennes seront à utiliser.

1 Le cours

Exercice 1 : Écrire de manière générique l'algorithme des k plus proches voisins pour un seul élément à étiqueter. Les arguments seront la liste des points placés en tant que couples formés par la valeur dans un espace muni d'une distance et l'étiquette attribuée (le nombre d'étiquettes différentes n'est pas nécessairement deux), la valeur de l'élément à étiqueter, la valeur de k et la fonction de distance. La fonction devra renvoyer l'étiquette attribuée à l'élément.

Exercice 2 : Écrire le cas particulier où l'espace est un certain $\mathbb{R}^d$, donc en retirant la fonction de distance.

Exercice 3 : Écrire de manière générique l'algorithme des k moyennes, prenant en argument une liste de points, tous des d-uplets de flottants car on travaille dans un certain $\mathbb{R}^d$, et la valeur de k. La fonction devra renvoyer une liste de k listes, qui sont les regroupements obtenus après la fin des itérations.

2 Extensions et applications de kNN

Exercice 4 : Modifier la fonction de l'exercice 1 pour que les voisins, toujours au plus k, soient pris dans un rayon de r au plus, le rayon étant un argument supplémentaire. S'il n'y a pas du tout de voisin dans le rayon en question, on sélectionne le plus proche.

Exercice 5 : Faire la modification « duale » : prendre au moins k voisins, mais aussi tous les points dans le rayon de r.

Exercice 6 : Écrire une fonction qui applique l'algorithme des k plus proches voisins pour une liste de points (dont la taille est en argument) obtenue par tirage aléatoire dans $]-\frac{3}{2},\frac{3}{2}[\times]-\frac{3}{2},\frac{3}{2}[$, les points étant étiquetés par 1 s'ils sont dans le disque délimité par le cercle trigonométrique et 0 sinon. Le point supplémentaire est également à prendre au hasard dans le carré en question. Répéter l'expérience plusieurs fois (nombre de fois en argument) et établir la matrice de confusion.

Exercice 7 : Dans le cadre de l'exercice prédédent, faire afficher le cercle trigonométrique, les points de base en noir et le point supplémentaire en vert s'il est considéré comme dans le cercle ou rouge sinon. Ici il ne faut faire qu'une fois l'expérience.

Cet exercice peut prendre plus de temps si la documentation du sous-module `pyplot` de `matplotlib` est à reconsulter. Dans ce cas, mieux vaut le faire chez soi en-dehors de la séance.

Exercice 8 : Prendre une image binarisée pertinente (par exemple une silhouette) et stocker son contenu dans une liste de listes. Écrire une fonction qui applique l'algorithme des k plus proches voisins pour une liste de couples d'indices dans la liste de listes et un couple d'indices supplémentaire. Là aussi on pourra afficher le résultat dans un graphique.

3 Extensions et applications de k-means

Exercice 9 : Écrire une fonction qui prend en argument une liste de points du plan et un rayon et qui retourne un nuage de points sous forme de liste de listes de points, de sorte que chaque liste contienne des points appartenant au disque du rayon fourni autour du point du même indice dans la liste des points donnés, en imposant au moins un point par disque. La taille du nuage de points est un argument supplémentaire, mais le nombre de points par liste peut être différent.

On s'efforcera lors de l'appel de cette fonction à ce qu'aucun cercle ne s'intersecte.

Là aussi, il serait intéressant de faire dessiner le nuage de points. En se débrouillant bien, dans le graphique les couleurs seront différentes dans chaque disque.

Exercice 10 : Écrire une fonction qui prend en argument une liste de nuages de points telle que retournée par la fonction de l'exercice précédent (sans tenir compte du regroupement) et qui applique l'algorithme des k-moyennes, où le nombre de clusters à constituer est la taille de la liste.

. . . et faire un graphique une fois de plus !

Exercice 11 : Écrire une fonction qui rassemble les deux exercices précédents et compte le nombre de points mal placés.

Exercice 12 : Modifier la fonction de l'exercice 10 pour ne plus tenir compte de l'information du nombre de clusters. À la place, un argument supplémentaire donne le nombre maximal de clusters autorisé, et il s'agira de choisir le nombre de clusters qui minimise la somme des variances.

Cette dernière fonction risque d'être moins spectaculaire qu'on pourrait l'imaginer. Quand la valeur de k augmente, la somme des variances aura tendance diminuer, ne serait-ce que parce qu'on peut prendre un regroupement optimal pour un k inférieur et extraire des points isolés de valeurs éloignées du barycentre.

TP 6 : Algorithmes autour des jeux d'accessibilité

À l'instar du TP précédent, nous allons ici écrire en Python les algorithmes donnés en cours, mais aussi étendre les simples conditions d'accessibilité et voir des exemples de jeux d'accessibilité connus des informaticiens voire du grand public.

Les arènes seront représentées en Python comme des listes dont les éléments, simulant les sommets, seront des couples dont le premier élément est le propriétaire du sommet (booléen valant `False` pour Ève et `True` pour Adam) et le deuxième est la liste des indices où figurent les couples représentant les sommets auxquels le sommet est relié.

Exercice 1 : Écrire une fonction prenant en entrée une arène et une condition de gain d'accessibilité pour Ève en tant que liste des sommets à atteindre, représentée par la liste des indices des sommets en question dans la liste, et calculant l'attracteur de cette liste de sommets, en tant que liste d'indices de sommets.

Pour ce premier exercice, il est recommandé de créer une liste de même taille que l'arène et contenant à chaque indice la liste des prédécesseurs des sommets à l'indice correspondant dans l'arène (le graphe est a priori orienté).

Exercice 2 : Écrire de même l'attracteur si la condition de gain est une condition d'accessibilité pour Adam. Le copier-coller est interdit (mais l'adaptation de la fonction précédente ajoutant un argument précisant le joueur voulant atteindre les sommets listés est autorisée).

Exercice 3 : Écrire finalement une version où les deux joueurs ont chacun un sous-ensemble cible, les sous-ensembles étant bien entendu disjoints. Une fois de plus, le copier-coller est interdit.

Exercice 4 : Déterminer la complexité des fonctions écrites, en vérifiant qu'elle est toujours la même (ce qui est à espérer au niveau de la programmation).

Exercice 5 : Adapter la fonction du premier exercice pour que la stratégie gagnante soit retournée, en tant que liste dont l'élément à chaque indice `i` est `"OK"` si le sommet à l'indice `i` de l'arène est dans la condition de gain, `-1` s'il appartient à Adam et n'est pas dans la condition de gain, `None` si ce sommet appartient à Ève mais n'est pas dans l'attracteur et un certain `k` sinon, sachant que le sommet `k` reste dans l'attracteur **et rapproche Ève de son objectif**.

Exercice 6 : Adapter la fonction du premier exercice pour une condition de gain d'accessibilité successive de plusieurs sommets pour Ève, ce qui signifie que la condition de gain sera représentée par une liste de listes et qu'Ève gagne une partie si, et seulement si, elle parvient à visiter un sommet dont l'indice est dans la première liste, puis (pas nécessairement immédiatement après) un sommet dont l'indice est dans la deuxième liste, et ainsi de suite. Adapter alors également la fonction de l'exercice précédent, en notant qu'il ne s'agira pas d'une stratégie sans mémoire, donc la structure de la stratégie est à repenser.

Clarification pour l'accessibilité successive : pour simplifier, on admettra que si un sommet est commun à deux listes consécutives, deux conditions peuvent être atteintes en même temps. Idéalement, on considèrera tout de même des listes disjointes de leurs voisines...

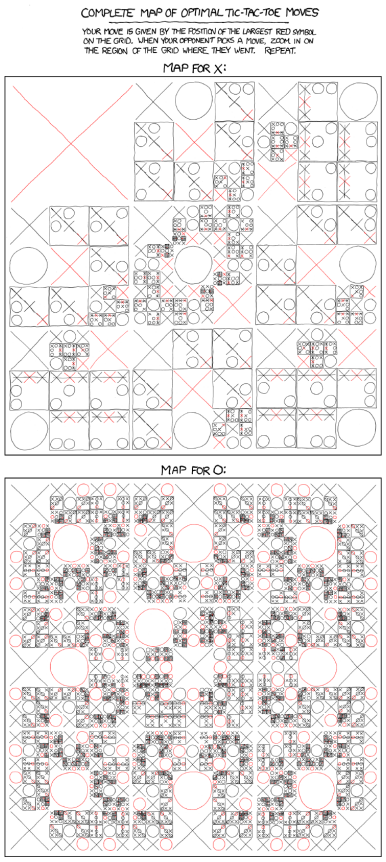
Exercice 7 : Adapter la fonction du premier exercice pour une condition de gain de Büchi pour Ève, ce qui signifie que la condition de gain sera représentée par une liste d'indices qu'Ève doit pouvoir visiter non pas une fois mais une infinité de fois dans une partie infinie. Il suffit en pratique de ne plus condidérer comme gagnants les sommets gagnants qui ne permettent plus l'accès en au moins une étape à un sommet encore gagnant au moment de l'étude, jusqu'à ce que l'ensemble se stabilise, puis de calculer l'attracteur de l'ensemble en question. Il faudra là aussi donner la stratégie gagnante, sachant que le coup d'Ève sur un sommet gagnant est à définir intelligemment.

Exercice 8 : Écrire une fonction calculant sans argument une stratégie gagnante pour le jeu du morpion, cette stratégie devant garantir la partie nulle contre un adversaire jouant optimalement et la victoire en cas de jeu douteux de l'adversaire. La valeur de retour devra être un dictionnaire dont les clés sont des triplets de triplets remplis de "X" (adversaire), "O" (soi-même) et "" (pas encore joué) correspondant à des configurations accessibles, et les éléments des coordonnées de cases où jouer, en tant que couple (ligne, colonne) avec l'indexation à partir de zéro.

Par exemple, pour la clé `(("X", "O", ""), ("X", "O", ""), ("", "", ""))`, on aura `(2, 1)` en tant que coup immédiatement gagnant.

Si le triplet est plein de chaînes vides, c'est qu'on commence, s'il contient un unique `"X"`, c'est que l'adversaire a déjà commencé...

Une petite référence à Randall Munroe qui a déjà fait le travail d'une certaine manière :



Vu que le nombre de configurations possibles, sans tenir compte des symétries et autres invariances, tient sur cinq chiffres, il est envisageable de construire le graphe de jeu en entier et de calculer l'attracteur.

Exercice 9 : Écrire une fonction calculant sans argument une stratégie gagnante pour le jeu de Marienbad.

Le principe du Marienbad est de disposer seize objets en quatre tas de tailles respectives 1, 3, 5 et 7 et de faire retirer à tour de rôle à chacun des deux joueurs autant d'objets qu'il souhaite d'un seul tas. Celui qui ne peut plus jouer a gagné (une variante plus cohérente au niveau de la théorie stipule le contraire, il s'agit d'un gain par blocage de l'adversaire).

On construira l'arène (en supposant qu'Adam commence) et on lancera le calcul de l'attracteur à partir de fonctions précédentes.

Exercice 10 : Reprendre l'exercice précédent pour en faire un jeu interactif (faire commencer l'ordinateur cependant, vu que la position de départ est gagnante pour le deuxième joueur).

Si l'envie prend d'utiliser une interface graphique (chez soi...), pourquoi pas !

TP 7 : Algorithmes autour des jeux et du minmax

1 Petit détour par la méthode de Monte-Carlo

Dans le TP 5, une application de l'algorithme des k plus proches voisins pour l'appartenance au disque délimité par le cercle trigonométrique était laissée en exercice. Nous allons reprendre cet exemple pour déterminer par une méthode de Monte-Carlo une approximation de la valeur de π.

Le principe est le suivant : le disque mentionné ci-avant a une aire de π et le carré $[-1, 1]^2$ a une aire de 4. Si on engendre des points du carré, la probabilité pour chacun d'être dans le disque est de $\frac{\pi}{4}$. On peut alors estimer qu'avec un très grand nombre de points, la fréquence d'appartenance au disque multipliée par quatre approche correctement π.

Exercice 1 : Réaliser cette expérience, avec le nombre de points en argument.

Pour changer, un dessin serait de bon ton !

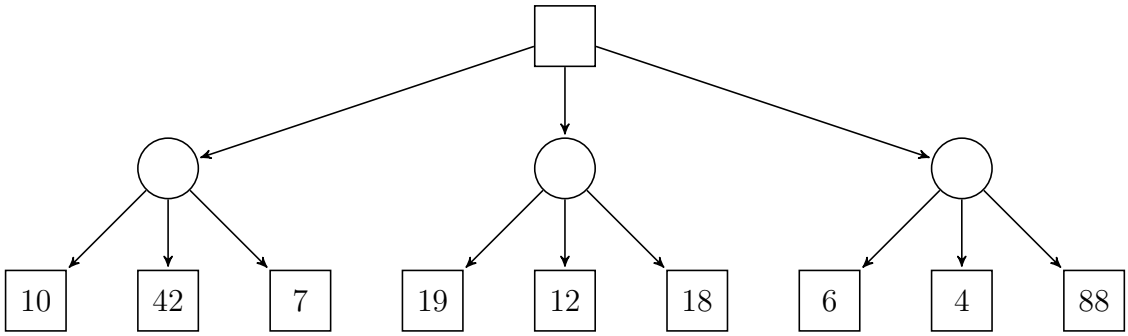
2 Algorithme minmax

Puisque l'algorithme minmax n'a pas été écrit directement dans le cours, c'est le moment de le faire ici, pour commencer dans sa version canonique.

La structure considérée est une arborescence quelconque, représentée sous la forme d'un tableau dont les éléments seront des triplets de la forme `(J, succ, val)`, où `J` est l'information du joueur, ici représentée comme un booléen, sachant que `True` correspondra au joueur voulant maximiser la valeur et `False` au joueur voulant la minimiser, `succ` est la liste des indices où se trouvent les successeurs de l'élément courant et `val` est la valeur de l'élément, initialement nulle sauf pour les éléments ayant une liste `succ` vide. On retrouve la structure du TP précédent en version un peu plus complexe.

L'élément de départ sera forcément à l'indice `0`, il ne pourra pas y avoir de cycle (ni même de confluence pour simplifier, en notant que cela ne changerait essentiellement rien) et les successeurs d'un sommet de n'importe quel joueur n'appartiendront pas forcément à un même joueur, donc en particulier l'alternance n'est pas garantie.

Exercice 2 : Écrire en Python la représentation de l'arborescence ci-dessous. Les carrés appartiendront au joueur `True` et les ronds au joueur `False`. L'ordre des éléments dans la liste n'est pas important, ni celui des successeurs dans les listes, tant que le branchement est cohérent.



Exercice 3 : Écrire l'algorithme minmax sous les conditions ci-avant. Il s'agit de renvoyer la valeur de l'élément de départ, sachant qu'on autorise à modifier la valeur associée à chaque élément intermédiaire par mutation de la liste.

Nous allons maintenant étudier un jeu permettant de mettre en situation l'algorithme minmax. Ce jeu a une longue histoire et a été originalement posé au détour d'une conversation avec un éminent professeur à l'ENS Cachan [5].

Considérons un gâteau circulaire et deux personnes s'apprêtant à le manger. La première a un couteau et découpe des parts selon le rayon du gâteau, de la forme qu'il souhaite. Puis la deuxième prend une part, qu'il mange, et chacun à son tour prend et mange une part disponible au sens où un de ses côtés est à l'air libre. Quand le gâteau est terminé, la personne ayant le plus mangé a gagné.

Bien entendu, découper un nombre pair de parts égales fait que chacun mangera la même chose. On observe aussi qu'a priori la personne choisissant la première part a tant de choix qu'elle semble avantagée, et effectivement sur un découpage arbitraire elle a une stratégie gagnante.

La question qui a été posée était de trouver un découpage du gâteau tel que la personne ayant procédé au découpage ait une stratégie gagnante pour manger **strictement** plus que l'autre.

5. Hubert Comon-Lundh

L'idée de considérer des parts epsilonesques et trois grosses parts dont deux allaient être prenables par la personne avec le couteau semblait prometteuse, mais ce n'est que des années plus tard qu'une solution a été proposée par un collègue [6], donnant l'occasion de vérifier par exploration exhaustive de l'arbre qu'elle était bien valide.

Cette solution ne sera pas écrite ici, et la difficulté rédhibitoire de la question fait qu'elle ne sera pas non plus demandée, mais à la place...

Exercice 4 : Écrire une fonction qui prend en entrée la liste des tailles des parts du gâteau, en admettant que le dernier et le premier élément de la liste sont des parts consécutives, et qui détermine si le joueur ayant procédé au découpage a une stratégie gagnante pour manger strictement plus que l'adversaire.

Cet exercice peut se découper en deux étapes : construire l'arborescence du jeu et lancer minmax dessus. Ici la valeur sur les feuilles dépend des prédécesseurs, comme si chaque nœud apportait une plus-value ou une moins-value.

À ce stade, on peut simplement considérer que c'est l'occasion de faire une version plus élaborée de minmax, et la version de base correspondrait au cas où toutes les plus-values sont à zéro.

En pratique, l'algorithme minmax n'est pas nécessaire pour déterminer qui gagne, car l'information des feuilles peut être remplacée par un booléen déterminant si c'est gagné ou perdu, auquel cas on se ramène à un jeu d'accessibilité simple car sur un arbre, mais on peut aussi chercher à gagner avec la plus grande proportion de gâteau possible, ce qui nécessite alors effectivement un minmax. Pour la solution proposée par mon collègue, je pense que la victoire se fait toujours par 2 unités d'avance en admettant que les joueurs jouent au mieux de leur intérêt, mais je n'ai pas vérifié...

3 Un jeu concurrent

Cette section vise à faire implémenter une partie de l'excellent travail réalisé par Nicky Case à l'adresse suivante : `https://ncase.me/trust/`. Nous considérons ici une version répétée du dilemme du prisonnier (pour la culture, se renseigner sur la version originale, qui n'a pas d'influence ici), avec un contexte différent, où l'objectif de chaque joueur est de maximiser sa fortune, indépendamment de celle des autres.

6. Nicolas Pécheux

À tout moment, deux des joueurs doivent faire un choix simultané entre offrir et garder. Un joueur qui offre perd une unité de fortune tandis que l'autre en gagne trois, un joueur qui garde ne change pas sa fortune ni celle de l'autre.

Il est apparent que garder est un meilleur choix quel que soit le choix de l'adversaire, même si paradoxalement la fortune totale est la plus grande si les deux joueurs offrent et gagnent donc deux unités chacun.

Pour la modélisation à venir, le choix d'un joueur sera un booléen (`True` pour offrir, `False` pour garder).

Exercice 5 : Écrire une fonction prenant en entrée deux listes de choix de même taille, une par joueur, et retournant le couple représentant la fortune de chaque joueur après l'expérience.

Les stratégies qui seront considérées ici seront les suivantes :

- La gentille : offre toujours.
- La méchante : garde toujours.
- La copieuse : offre la première fois et imite toujours le dernier mouvement de l'adversaire par la suite.
- La méfiante : offre toujours mais dès que l'adversaire garde une fois garde toujours jusqu'à la fin.
- L'aléatoire : joue toujours au hasard.

Exercice 6 : Écrire pour chaque stratégie une fonction prenant en entrée une liste de choix prédéfinie pour l'adversaire et retournant les choix faits par la stratégie en question.

Cette fonction ne sera pas utilisée telle quelle, mais permettra de détecter d'éventuelles erreurs dans ce qui suit.

Exercice 7 : Écrire pour chaque stratégie une fonction permettant d'affronter un joueur ayant cette stratégie et retournant la fortune des deux joueurs (au sens de l'exercice 5) après l'expérience. On mettra le nombre de coups à jouer en argument.

Il s'agit encore une fois d'une fonction non utilisée, mais pour une fois qu'on peut un peu s'amuser avec de l'interactivité...

Exercice 8 : Écrire une fonction prenant en entrée deux stratégies (représentation au choix) et un nombre de coups et simulant une partie entre les deux stratégies durant le nombre de coups précisé. La fonction retournera la liste de chaque joueur.

On branchera cette fonction sur celle du premier exercice de la section pour obtenir le gain de chaque stratégie. Ce gain va désormais être cumulé dans un enchaînement de tels affrontements.

Exercice 9 : Écrire une fonction prenant en entrée une liste de stratégies avec doublons possibles et un nombre de coups et simulant un tournoi où les stratégies s'affrontent toutes deux à deux. La fonction renverra la liste des stratégies avec leur fortune totale triée par fortune décroissante.

Exercice 10 : Écrire une fonction prenant en entrée une liste de taille paire de stratégies, un nombre de coups et un nombre de répétitions d'expérience et simulant un enchaînement de tournois où les stratégies s'affrontent toutes deux à deux. À la fin de chaque tournoi, la moitié des stratégies avec la fortune la plus basse est supprimée et l'autre moitié est dupliquée, avec sélection au hasard en cas d'égalité. La fonction renverra la liste des stratégies restantes à la fin.

Lexique

28 Adressage ouvert
17 Agrégation (fonctions)
8 Algèbre relationnelle
43 Algorithme des k plus proches voisins
43 Algorithme des k-moyennes
62 Algorithme minmax
51 Arène
60 Attracteur
8 Attribut (bases de données)

28 Chaînage
12 Clé (bases de données)
12 Clé étrangère
12 Clé primaire
27 Collision
52 Condition de gain

25 Dictionnaire
9 Différence (opérateur relationnel)
10 Division cartésienne (opérateur relationnel)
9 Domaine (d'un attribut)

8 Enregistrement (ou valeur)

63 Heuristique

9 Intersection (opérateur relationnel)

52 Jeu
62 Jeu à somme nulle
59 Jeu d'accessibilité
54 Jeu déterminé
11 Jointure symétrique

45 Matrice de confusion