IA Pas de Soucis !

Un projet de Evrard Maurice,
Lejeune Grégory,
Lejeune Lucas
Mathieu Louca,
Pluvinage Victor
et de Ralet Vincent.

22 février 2021

Table des matières

"Copier, puis coller"

– **Ralet, Vincent**

"Une couche de neurones d'entrées, plusieurs couches cachées, une couche de sortie suivie d'une fonction d'activation."

– **Ralet, Vincent (aussi)**

"J'ai perdu la passion"

– **Ralet, Vincent**

"Les instructions étaient pas claires"

– **Pluvinage, Victor**

1 Introduction.

Ce projet a été créé dans le cadre du cours de PESU maths, ce projet a pour objectif de présenter les différentes méthodes utilisées dans la création d'intelligences artificielles au cours des dernières décennies.

Notre travail se sépare en différentes parties, nous abordons d'abord l'aspect historique, ensuite une partie logique dans laquelle nous parlerons de tables de vérités, d'arbres de preuves, et du langage PROLOG.

La suite se compose de la partie algorithmique, qui abordera les thèmes de la récursion, de la complexité ainsi que de la notation de Landau, les exemples fournis dans cette partie seront écrits en Common Lisp.

Pour finir, nous expliquerons le fonctionnement du machine learning, nous définirons également la Data-Science et illustrerons son utilité via quelques utilisations de cette science dans la vie courante, nous donnerons également des exemples d'utilisation du langage Python, et présenterons les bibliothèques Numpy et Keras.

2 Histoire.

2.1 Origines.

Avant les premières découvertes scientifiques concernant l'intelligence artificielle, elle est déjà imaginée dans des oeuvres de fiction dans lesquelles des artisans seraient capables de mettre au point des êtres artificiels dotés d'une intelligence. Certains verront le jour sous la forme d'automates (Ex : Automates de Vaucanson, 18e siècle).

L'intelligence artificielle comme nous la connaissons aujourd'hui a d'abord été présentée par des philosophes de l'ère classique, Leibniz par exemple. Ils avançaient que la pensée humaine était un raisonnement mécanique explicable par les mathématiques. Ils n'ont cependant jamais tenté de créer une intelligence artificielle.

Entre les années 1930 et 1950 de nombreux progrès sont faits en neurologie, on comprend que le cerveau est un réseau de neurones qui envoient un signal électrique binaire (cf. Théorie de l'information de Claude Shannon). La théorie du calcul d'Alan Turing montre que toute forme de calcul peut être représentée numériquement. Additionnés au travail de Norbert Wiener sur la cybernétique, ces progrès poussent la communauté scientifique à croire en la possibilité de construire un cerveau artificiel.

2.2 Arrivée des premiers ordinateurs.

Les premiers ordinateurs modernes feront leur apparition durant la seconde guerre mondiale poussés par la quête d'informations protégées par des codes sur les stratégies des différents camps.

2.2.1 Premières concrétisations d'intelligences artificielles :

a. En 1949, Warren Weaver met au point un protocole qui vise à utiliser l'intelligence artificielle pour traduire automatiquement différents langages.

b. En 1950, Alan Turing publie un article dans lequel il imaginera le célèbre *Test de Turing* qui vise à définir ce qui est ou non une machine intelligente. Pour se faire, la machine doit pouvoir converser avec un humain sans que ce dernier se rende compte de sa vraie nature. Il sera réussi pour la première fois dans les années 60 par ELIZA, une IA simulant un psychothérapeute. Elle se contentait en réalité de reformuler les affirmations du"patient" en questions.

c. En 1955, Allen Newell accompagné de Herbert Simon de de Cliff Shaw crée le *Théoricien logique*, un programme capable de démontrer 38 des 52 premiers théorèmes des *Principia Mathematica* de Russell et Whitehead, et a même trouvé des démonstrations inédites et élégantes.

2.3 Officialisation.

En 1956 la conférence de Dartmouth réunit tous les plus grands scientifiques travaillant alors sur le sujet de l'intelligence artificielle. Elle est le vrai point de départ de l'intelligence artificielle moderne étant donné que c'est à ce moment qu'elle a défini ses objectifs et qu'elle devient officiellement une discipline scientifique.

2.4 L'âge d'or (1956-1974).

La conférence de Dartmouth organisée par Marvin Minsky, John McCarthy, Claude Shannon et Nathan Rochester marque le début de l'âge d'or de l'IA. En effet, entre 1956 et 1974 l'IA va voir un grand nombre de découvertes qui pousseront les investissements, ce qui entraînera un cercle vertueux. Cette période sera donc marquée par un optimisme presque excessif concernant les"futurs" progrès de L'IA.

La méthode alors utilisée par la grande majorité des programmes consiste à avancer pas à pas en faisant des essais-erreurs. Cette méthode montrera toutefois ses limites face aux *explosions combinatoires* (problèmes dont le nombre de chemins possibles vers une solution est astronomique).

2.5 L'hiver de l'IA (1974-1980).

L'année 1974 marque le début de la période creuse de l'histoire de l'intelligence artificielle. Cela étant dû au retour à la réalité des chercheurs en IA qui se retrouvaient face à des impasses, limités par le niveau technologique de l'époque. Cette baisse de vitesse dans les découvertes sera accompagnée de nombreuses critiques concernant les recherches menées jusqu'alors provoquera la déception et le retrait de nombreux investisseurs optimistes.

Cette période marque malgré tout l'introduction de la logique dans l'IA, notamment par l'invention du langage de programmation Prolog.

2.6 Le retour en force (1980-1987).

Cette période voit apparaître un nouveau genre de programmes d'IA appelé "Systèmes experts", ces programmes ont pour particularité d'être limités à des domaines très restreints. En effet, ce type de programme utilise un ensemble de règles logiques directement inspirées des connaissances des experts du domaine. Cette limite volontaire imposée à ces programmes les rendent plus facile à concevoir et à améliorer une fois déployés. Ces programmes sont les premiers à réellement se rendre utiles pour la société.

Le gouvernement japonais va permettre à l'IA de se relancer économiquement en investissant 850 millions de Dollars dans le but de développer des programmes de communication (Traduction, interprétation d'images, ...). Le Royaume-Uni lance un projet similaire qu'il finance à hauteur de 350 millions de livres.

2.7 Le nouvel hiver (1987-1993).

L'engouement qu'avait réussi à créer l'IA au début des années 80 grâce aux systèmes experts a en fait créé une bulle économique qui a fini par exploser à la fin de la décennie.
Cette période marquera tout de même l'apparition d'une nouvelle approche

de l'IA, une approche dans laquelle les intelligences artificielles devraient avoir conscience de"leur corps" et de l'environnement qui les entoure.

2.8 Depuis 1993.

Grâce à un niveau technologique et donc une puissance de calcul qui augmente de manière exponentielle (c.f. Loi de Moore), de nombreux objectifs de l'IA sont atteints. Les investissements n'affluent toujours pas, car on ne s'est pas encore approché du tout du rêve des années 60, c'est-à-dire une IA aussi intelligente qu'un humain.

Depuis 1993, l'IA continue de faire ses preuves au travers de nombreux exploits, on peut noter la victoire historique de l'IA Deep Blue aux échecs contre le champion du monde en titre d'échecs Garry Kasparov. On peut aussi parler des nombreux robots anthropomorphes comme le Asimo de Honda ou les robots militaires de Boston Dynamics dont les capacités ne cessent d'impressionner le grand public. Mais au delà de ces prouesses, l'intelligence artificielle a aujourd'hui surtout gagné une place proéminente dans notre quotidien : applis de traduction, publicités ciblées, GPS, recommandations d'amis sur les réseaux sociaux, ...

3 L'Approche Logique.

3.1 Les bases de la logique

3.1.1 Introduction à la logique propositionnelle.

L'importance de la logique propositionnelle est immense en mathématiques et en cryptographie, mais également, comme nous allons le voir, en informatique.

Voici une fameuse lapalissade, exemple typique d'utilisation d'une phrase ne découlant de rien d'autre que de cette logique :

"15 minutes avant sa mort, il était encore en vie." [1]

Évidemment, grâce à notre capacité déductionnelle, nous pouvons tous définir cette phrase comme vraie, c'est ici, une vérité dite "de langage".

3.1.2 Un peu de vocabulaire !

La logique propositionnelle possède son propre vocabulaire, il est presque indispensable de connaître son vocabulaire et sa syntaxe afin même de pouvoir en comprendre les concepts.

Tout d'abord, un **langage formel** est un ensemble de mots que l'on peut obtenir en utilisant un alphabet [2], un langage possède plusieurs **lois**, cet ensemble de lois sera appelé la **syntaxe**, qui définissent les différentes manières grâce auxquelles les éléments de l'alphabet (aussi appelés les **symboles**) peuvent se placer pour former quelque chose de cohérent au langage.

Par exemple, en français, vous n'écririez pas "Jème lai vwaturres", mais plutôt, "J'aime les voitures", aussi bête que ce petit exemple puisse paraître,

1. https ://fr.wikipedia.org/wiki/Lapalissade
2. Cela peut être un alphabet comme abcd...yz tout comme un alphabet composé uniquement de 1 et de 0.

il s'agit là de l'une des nombreuses fois où l'on se plie aux règles d'une grammaire logique.

Un mot respectant toutes les règles de syntaxe sera alors appelé un **mot bien formé**.

La logique propositionnelle se compose donc d'un langage formel, et de sémantiques donnant du sens aux mots bien formés, répondant au nom de **propositions**.

Les propositions logiques sont désignées par des lettres, comme $A, B, C...$, ou par des lettres indicées comme $A_2, B_4...$ Pour relier ces propositions, on utilise des connecteurs, répertoriés dans le tableau ci-dessous.

nom	**symbole**	**autre nom**
négation	$\neg$	NOT
conjonction	$\wedge$	AND
disjonction (I)	$\vee$	OR
disjonction (E)	$\oplus$	XOR
implication	$\Rightarrow$	IF..THEN
équivalence	$\Leftrightarrow$	IFF

En plus de ces connecteurs viennent s'ajouter les deux valeurs logiques à la base de tout, Vrai et Faux.

3.1.3 Pour quelques exemples de plus...

Voici quelques exemples d'énoncés de logique propositionnelle dans un cadre assez éloigné des mathématiques, en espérant que cela fasse sens au lecteur.

Supposons que A et B soient deux propositions logiques.
A : Je suis boulanger.
B : Je sais faire des gâteaux.

Nous pouvons ainsi relier ces deux propositions avec les connecteurs vus dans le tableau ci-dessus.
$\neg A$: Je ne suis pas boulanger.
$\neg B$: Je ne sais pas faire des gâteaux.
$A \wedge B$: Je suis boulanger et je sais faire des gâteaux.
$A \vee B$: Je suis boulanger ou je sais faire des gâteaux . (Les deux propositions peuvent être vraies, tout comme une seule des deux).
$A \oplus B$: Je suis boulanger ou alors je sais faire des gâteaux. (Une seule de ces deux propositions doit être vraie).
$A \Rightarrow B$: Je suis boulanger, donc je sais faire des gâteaux.
$A \Leftrightarrow B$: Je suis boulanger si et seulement si je sais faire des gâteaux.
...

3.1.4 Quelques tables de vérité.

Après tant d'exemples "instructifs", il serait temps de passer aux fameuses **tables de vérité**, ces tables seront d'une importance capitale lors de résolutions de problèmes, les voici donc :

Table 1 – Négation.

A	$\neg A$
F	V
V	F

Table 2 – Conjonction.

A	B	$A \wedge B$
F	F	F
F	V	F
V	F	F
V	V	V

Table 3 – Disjonction (OR).

A	B	$A \vee B$
F	F	F
F	V	V
V	F	V
V	V	V

Table 4 – Disjonction (XOR).

A	B	$A \oplus B$
F	F	F
F	V	V
V	F	V
V	V	F

Table 5 – Implication.

A	B	$A \Rightarrow B$
F	F	V
F	V	V
V	F	F
V	V	V

Table 6 – Équivalence.

A	B	$A \Leftrightarrow B$
F	F	V
F	V	F
V	F	F
V	V	V

Les tables 1 à 4 doivent sans doute paraître logique, je m'attarderai toutefois sur la table 5, en effet les deux phrases $F \Rightarrow V$ et $F \Rightarrow F$ sont toutes deux vraies. Cela est dû au **principe d'explosion**[3] : Du faux, on peut déduire absolument n'importe quoi ! (On verra plus tard que l'on peut noter ceci $A \wedge \neg A \models B$).

Pour ce qui est de la table 6, il faut savoir que certains notent $\Leftrightarrow$ comme étant $\equiv$, cette notation a l'avantage d'accentuer le fait que cette relation n'est autre que la relation d'équivalence.

3.1.5 Résolution de problèmes en logique propositionnelle.

Maintenant que nous avons acquis les bases de la logique propositionnelle, attaquons nous à quelques problèmes.

En voici un premier,

$$\boxed{\neg(A \wedge B) \Leftrightarrow \neg A \vee \neg B}$$

Pour cela nous utiliser une grande table de vérité.

A	B	$A \wedge B$	$\neg(A \wedge B)$	$\neg A$	$\neg B$	$\neg A \vee \neg B$	$\neg(A \wedge B) \Leftrightarrow \neg A \vee \neg B$
F	F	F	V	V	V	V	V
F	V	F	V	V	F	V	V
V	F	F	V	F	V	V	V
V	V	V	F	F	F	F	V

Ainsi, comme nous pouvons le constater, la dernière ligne est remplie de "V", cela veut donc dire que nous venons de prouver la relation $\neg(A \wedge B) \Leftrightarrow \neg A \vee \neg B$, aussi connue sous le nom de "la loi de DeMorgan", nous avons prouvé par la même occasion que $\neg(A \wedge B) \Leftrightarrow \neg A \vee B$ est **valide**, cela veut dire qu'elle sera toujours vraie, peu-importe les A, et les B.

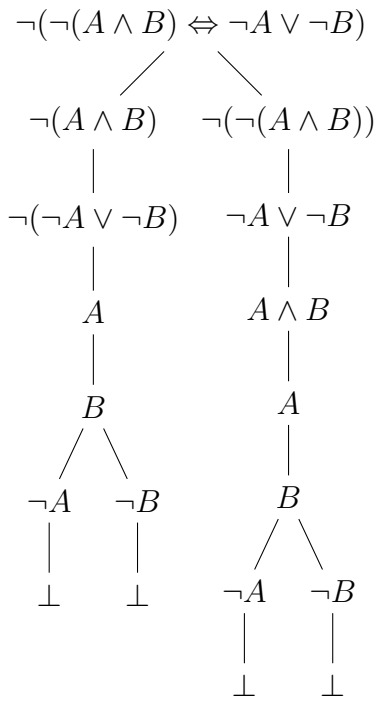
3. https ://fr.wikipedia.org/wiki/Principe_d%27explosion

Il existe toutefois une autre manière de faire, il s'agit d'utiliser un arbre, cette méthode requiert moins d'étapes et nous permettra de résoudre certains problèmes de manière plus simple. Toutefois, il y a quelques règles à respecter lors de l'utilisation d'un arbre, ces règles sont répertoriées dans le tableau ci-dessous.

$\neg(\neg\phi)$ $\vert$ ϕ	$\phi \wedge \psi$ $\vert$ ϕ ψ	$\neg(\psi \wedge \phi)$ $/ \quad \backslash$ $\neg\psi \quad \neg\phi$
$\phi \vee \psi$ $/ \quad \backslash$ $\psi \quad \phi$	$\neg(\phi \vee \psi)$ $\vert$ $\neg\phi$ $\neg\psi$	$\phi \Rightarrow \psi$ $/ \quad \backslash$ $\neg\phi \quad \psi$
$\neg(\phi \Rightarrow \psi)$ $\vert$ ϕ $\neg\psi$	$\phi \Leftrightarrow \psi$ $/ \quad \backslash$ $\phi \quad \neg\phi$ $\psi \quad \neg\psi$	$\neg(\phi \Leftrightarrow \psi)$ $/ \quad \backslash$ $\phi \quad \neg\phi$ $\neg\psi \quad \psi$

Le but du jeu avec un arbre logique, c'est de terminer chacune des branches de l'arbre par $\perp$ (C'est le symbole utilisé pour les contradictions). Ainsi, avec un arbre, nous commençons par utiliser l'opposé de notre hypothèse de base (ici, l'opposé de $\neg(A \wedge B) \Leftrightarrow \neg A \vee \neg B$, c'est $\neg(\neg(A \wedge B) \Leftrightarrow \neg A \vee \neg B)$), (en général, une contradiction arrive quand nous nous retrouvons avec A et $\neg A$ sur la même branche).

Passons maintenant à la preuve



Chaque branche de l'arbre fini bien par $\perp$, nous venons donc de prouver la loi de DeMorgan, avec l'aide de notre arbre de démonstration.

Ci dessous, le lecteur pourra s'essayer à la démonstration de certaines lois logiques célèbres, avec l'aide d'un tableau ou avec un arbre.

$$\neg\neg A \Leftrightarrow A \tag{3.1.1}$$

$$A \wedge \neg A \Leftrightarrow F \tag{3.1.2}$$

$$A \vee \neg A \Leftrightarrow T \tag{3.1.3}$$

$$A \wedge F \Leftrightarrow F \tag{3.1.4}$$

$$A \vee V \Leftrightarrow V \tag{3.1.5}$$

$$\begin{cases} A \vee B \Leftrightarrow B \vee A \\ A \wedge B \Leftrightarrow B \wedge A \end{cases} \tag{3.1.6}$$

$$\begin{cases} A \wedge (B \wedge C) \Leftrightarrow (A \wedge B) \wedge C \\ A \vee (B \vee C) \Leftrightarrow (A \vee B) \vee C \end{cases} \tag{3.1.7}$$

$$A \wedge (B \vee C) \Leftrightarrow (A \wedge B) \vee (A \wedge C) \tag{3.1.8}$$

3.2 Les ensembles de propositions.

Les ensembles de propositions, comme leur nom l'indique, sont des ensembles mathématiques, composés de **formules logiques**. Ces formules sont dites soit :

- **consistantes**, signifiant qu'il est possible d'en tirer du Vrai, par exemple $A \wedge B$ ou encore $\neg\neg A \Rightarrow A$.
- **inconsistances**, signifiant que l'on ne peut en tirer que du Faux, par exemple $A \wedge \neg A$, ces formules peuvent être notées $\bot$.
- **valides**, signifiant qu'elles ne peuvent être que Vraies, comme $A \vee \neg A$ (principe du tiers exclus), une formule **valide** est par définition toujours **consistante**, on appelle bien souvent ces formules valides des **tautologies**, ces formules pourront être notées $\top$.

- **contingentes**, impliquant que l'on peut tirer de la formule du faux, tout comme du vrai, une **tautologie** ne peut pas être formule contingente, un exemple de formule contingente serait $A \vee B$.

Les ensembles aussi ont leur propre terminologie, ainsi, si l'on prend l'ensemble noté S, il pourra être qualifié également de **consistant**, si il n'y a pas de contradictions au sein de S et qu'il n'y a aucune formule inconsistante contenue dans S, autrement S sera défini comme étant **inconsistant**.

Il y a plusieurs manières d'**inférer** quelque chose d'un ensemble logique.

Une première manière est de prendre les **prémisses** qui nous intéressent, et d'écrire

1. **Premisse_A**
2. **Premisse_B**
⋮
n. **Premisse_X**

$\therefore$ **Conclusion**

Prenons un ensemble **consistant** S composé des formules A et $A \Rightarrow B$.

On pourrait alors noter S comme étant $S = \{A, A \Rightarrow B\}$ (ce qui est parfaitement équivalent à écrire $S = A \wedge (A \Rightarrow B)$),

De ceci, nous allons utiliser l'opérateur de la déduction, $\vDash$, ceci nous permettra ainsi écrire $S \vDash A$, littéralement "De S, nous déduisons A".

Ce principe est encore plus flagrant quand nous utilisons la notation suivante : $A, A \Rightarrow B \models A$, où $A, A \Rightarrow B$ n'est autre que l'ensemble S, avec une notation légèrement différente.

Je n'ai pas choisi cet ensemble de manière anodine, car, grâce à celui-ci, nous allons pouvoir utiliser une **règle d'inférence** connue sous le nom du MP (Modus Ponens), le lecteur ne devrait toutefois pas s'inquiéter, un tableau recensant d'autres règles d'inférence sera présenté à la page suivante.

Une première manière de noter cette règle d'inférence serait de faire usage de la notation que nous avons vue plus haut.

1. A (De cet ensemble, nous savons A)
2. $A \Rightarrow B$ (De cet ensemble, nous savons que $A \Rightarrow B$)

$\therefore$ B

Une autre manière serait d'utiliser l'opérateur $\vDash$ (celui de la **déduction**), de la manière suivante : $A, A \implies B \vDash B$, nous pourrions même être tentés d'utiliser la **règle d'addition**, disant que si l'on a $S \vDash B$, alors, on peut rajouter la formule B à S. Et ainsi, notre ensemble S de base pourra être réécrit en $S = A, A \Rightarrow B, B$ Et maintenant, comme promis, voici un tableau comprenant toutes les règles d'inférence qui seront bien pratiques pour travailler avec les ensembles logiques.

Règle d'inférence	Tautologie	Nom de la règle d'inférence
$A, B \vDash A \wedge B$	$A \wedge B \Rightarrow A \wedge B$	Loi de combinaison
$A, B \vDash A$	$(A \wedge B) \Rightarrow A$	Loi de la simplification
$A, A \Rightarrow B \vDash B$	$A \wedge (A \Rightarrow B) \Rightarrow B$	Modus Ponens
$\neg B, A \Rightarrow B \vDash \neg A$	$\neg B \wedge (A \Rightarrow B) \Rightarrow \neg A$	Modus Tollens
$A \Rightarrow B, B \Rightarrow C \vDash A \Rightarrow C$	$(A \Rightarrow B) \wedge (B \Rightarrow C) \Rightarrow (A \Rightarrow C)$	Syllogisme hypothétique
$A \vee B, \neg A \vDash A$	$(A \vee B) \wedge \neg A \Rightarrow B$	Syllogisme disjonctif
$A \Rightarrow B \vDash A \Rightarrow (A \wedge B)$	$(A \Rightarrow B) \Rightarrow (A \Rightarrow (A \wedge B)$	Règle d'absorption
$A \Rightarrow B, C \Rightarrow B, A \vee C \vDash B$	$(A \Rightarrow B) \wedge (C \Rightarrow B) \wedge (A \vee C) \Rightarrow B$	Élimination disjonctive
$A, \neg A \vDash B$	$A \wedge \neg A \Rightarrow B$	Principe d'explosion

Le lecteur pourra s'exercer à démontrer la validité des tautologies présentées dans le tableau-ci.

3.2.1 Le besoin d'algorithme, présentation de l'algorithme de Quine.

Un ensemble de formules, tout comme une formule peut-être inconsistante, cela signifie que notre ensemble est équivalent à F,et comme nous l'avons vu dans le tableau ci-dessus, on peut déduire absolument n'importe quoi d'un ensemble inconsistant.

Le problème des tableaux de vérités, c'est qu'ils prennent de la place, et même, beaucoup de place. C'est pour cela qu'est venue la nécessité de créer des algorithmes de résolution d'ensembles de formules, afin de faciliter le travail des logiciens, et ainsi, de réduire le temps de calcul nécessaire à un ordinateur.

L'algorithme de Quine se déroule en plusieurs étapes, tout d'abord, nous allons simplifier, si possible, les formules logiques contenues dans notre ensemble de formules, par exemple, au lieu d'écrire $A \vee V$, nous pourrons écrire simplement V, et ainsi, supprimer ce V de notre ensemble, comme $B, V \models B$, il y a bien d'autres simplifications possibles, nous laisserons au lecteur le soin d'en trouver.

Une fois cette première étape passée, il suffit d'utiliser un arbre logique (ou éventuellement une table de vérité) et enfin, nous avons pu prouver des formules logiques avec l'aide de l'algorithme de Quine.

3.3 Une logique ? Des logiques !

Les logiciens, non contents de la seule logique propositionnelle, ont créé un très grand nombre de logiques, j'en liste quelques unes ci-dessous.

- La logique linéaire, créée par un français, est basée sur la gestion de ressources, c'est une des nombreuses logiques n'excluant pas le tiers exclus, en effet, en logique linéaire, $A \vee \neg A \nvDash V$, cela est simplement du au fait que nous "utilisons" la ressource A une fois, elle n'existe donc plus réellement, et donc $\neg A$ n'existe pas comme A est devenu indéterminé.
- La logique floue, dans laquelle une proposition est vraie selon un certain degré de probabilité.
- La logique de premier ordre, ou logique des prédicats, cette logique sera abordée dans le chapître sur Prolog.
- La logique booléenne, elle est basée sur les portes logiques, les circuits logiques, et les ensembles.

— La logique combinatoire, logique inventée pour formaliser la notion de fonction, et pour limiter le nombre d'opérateurs nécessaires pour définir le calcul des prédicats.
— La logique modale, ayant recours à des opérateurs comme "il est nécessaire que" ou "il est possible que".
— Et bien d'autres...

3.4 Introduction à Prolog et à la logique des Prédicats.

Dans cette nouvelle sous-section, nous allons nous intéresser à la logique des prédicats, connue également sous le nom de logique de premier ordre.

Tout d'abord, en logique des prédicats, nous aurons besoin de deux nouveaux **quantificateurs**.

Ceux-ci sont le quantificateur **universel**, noté $\forall$ [lisez "pour tout"], et le quantificateur **existentiel**, noté $\exists$ [lisez "il existe"].

A ces deux quantificateurs viennent s'ajouter :
— Des **connecteurs logiques**, qui ont été discutés dans la section précédente.
— Des **constantes**, celles-ci représentent un événement, une personne ou un objet en particulier, nous noterons ces constantes avec une majuscule comme première lettre et un nombre à la fin, par exemple "Turing_1" ou encore "Chaise_2".
— Des **variables**, celles-ci représentent un concept général ou un ensemble, par exemple, l'ensemble des mathématiciens, ou encore l'ensemble des chaises dans le monde. Nous noterons ces variables en minuscules, par exemple "mathématiciens" ou encore "chaise".
— Des **prédicats**, ceux-ci nous permettent d'établir des liens entre nos différentes variables et constantes, nous noterons nos prédicats avec une majuscule en première lettre, par exemple "**Mortel**(x)" ou encore "**Humain**(x)".
— Des **fonctions**, qui ont pour but de retourner une valeur, pouvant-

être autre chose que vrai ou faux. Nous noterons celles-ci en toutes minuscules.

3.4.1 Exemples.

Pour se faire une bonne idée, voici quelques phrases françaises “traduites” en logique des prédicats.

1. Tout les mathématiciens sont cools.
=> $\forall$x (**Mathématicien**(x) $\Rightarrow$ **Cool**(x)).

2. Alan Turing et Alonzo Church sont des mathématiciens.
=> **Mathématicien**(Turing_1) $\wedge$ **Mathématicien**(Church_1).

3. Il y a des chats qui ne sont pas noirs.
=> $\exists$x (**Chat**(x) $\wedge$ $\neg$**Noir**(x)).

3.4.2 Qu’est-ce que PROLOG ?

Cette petite introduction passée, concentrons-nous maintenant sur le coeur du sujet : PROLOG !

Prolog a été inventé en 1972 par les informaticiens français Alain Colmerauer et Philippe Roussel.

C’est un langage de programmation **logique** et son nom est un acronyme pour PROgrammation LOGique.

Prolog a été très utilisé en Europe et au Japon dans le domaine de l’Intelligence Artificielle, tout en étant basé sur la logique propositionnelle dont nous avons posé les bases juste au dessus.

Il existe de nombreuses distributions de PROLOG [4], nous utiliserons ici

4. https ://en.wikipedia.org/wiki/Comparison_of_Prolog_implementations

SWI-PROLOG, avant tout pour son côté open source et gratuit.

3.4.3 Introduction à la syntaxe de Prolog

En Prolog, contrairement aux règles que nous avions établies en logique des prédicats, les constantes (ici appelés Atomes) doivent commencer par une minuscule. Les variables commencent par une majuscule. A cela viennent s'ajouter les listes, dénotées par des [].

Faits et Règles.

En PROLOG, un fait s'écrit simplement

```
mathematicien(turing).
```

Un fait n'a pas de "corps", et tiendra toujours. Dans ce cas-ci, cela veut dire que Turing est un mathématicien, quoiqu'il arrive ce **fait** ne changera pas.

Maintenant, si nous essayons de reformuler l'exemple n°1 de notre dernière section, "tous les mathématiciens sont cools" en Prolog, cela donne ceci :

```
cool(X) :-
    mathematicien(X).
```

On remarque tout de suite que cela est plutôt facile à lire, de plus, si nous ouvrons notre interprète Prolog, voilà ce que nous obtenons :

```
?- cool(turing).
true.
```

Tout cela est très bien, mais, si j'essaye de demander à prolog si alonzo_church est cool, que se passe-t-il ?

```
?- cool(alonzo_church).
false.
```

Pas grand-chose comme nous le constatons, PROLOG préfère éviter de faire la moindre assomption, et répondra "Faux" dès qu'il ne sait pas.

Ensuite, Prolog nous permet de faire de l'arithmétique, regardons un peu cela, avec une fonction dont le seul rôle est d'additionner deux nombres :

```
ajouter(A, B, C) :-
    C is A + B.
```

Ouvrons maintenant l'interprète et regardons la magie opérer :

```
?- ajouter(3, 4, Res).
Res = 7.
```

Mais, si pour une obscure raison, je décidais de vouloir avoir B, juste en entrant A et C, comment faire ? Regardons d'abord comment notre première version réagirait à cela :

```
?- ajouter(3, B, 7).
ERROR: Arguments are not sufficiently instantiated
ERROR: In:
ERROR:     [9] 7 is 3+_9722
ERROR:     [7] <user>
ERROR:
ERROR: Note: some frames are missing due to last-call
   optimization.
ERROR: Re-run your program in debug mode (:- debug.) to get
    more detail.
```

Heureusement pour nous, Prolog possède un module basé sur la logique par contraintes, pour l'utiliser, il suffit d'ajouter [5]

```
:- use_module(library(clpfd)).
```

au dessus de votre fichier Prolog, maintenant, modifions légèrement notre prédicat :

```
ajouter_clp(A, B, C) :-
    C #= A + B.
```

Et voilà ! Essayons là maintenant avec notre interprète :

```
?- ajouter_clp(3, B, 7).
B = 4.
```

De la vraie magie !

5. https ://www.swi-prolog.org/man/clpfd.html

3.4.4 Mon arbre familial, avec Prolog !

Pour montrer la puissance de Prolog, ce chapître présente un petit exemple d'arbre familial (un exercice très classique pour apprendre Prolog), ici, l'arbre représenté est celui de notre famille royale belge[6] Le programme se sépare en deux étapes. 1. L'écriture de faits simples. 2. L'écriture des prédicats, afin de faire de l'inférence sur les liens familiaux. Voici pour l'écriture des faits :

```
% Ecrivons quelques faits.
fils(louis_philippe_de_belgique ,louise_d_orleans, leopold1
   ).
fils(leopold2, louise_d_orleans, leopold1).
fils(philippe_de_belgique, louise_d_orleans, leopold1).
fils(leopold_de_belgique, marie_henriette_d_autriche,
   leopold2).
fils(lucien_durrieux, blanche_delacroix, leopold2).
fils(philippe_durrieux, blanche_delacroix, leopold2).
fils(baudoin_de_belgique, marie_de_hohenzollern_sigmaringen
   ,
     philippe_de_belgique).
fils(albert1, marie_de_hohenzollern_sigmaringen,
     philippe_de_belgique).
fils(charles_de_belgique, elisabeth_en_baviere, albert1).
fils(leopold3, elisabeth_en_baviere, albert1).
fils(baudoin1, astrid_de_suede, leopold3).
fils(albert2, astrid_de_suede, leopold3).
fils(alexandre_de_belgique, lilian_baels, leopold3).
fils(philippe1, paola, albert2).
fils(laurent, paola, albert2).
fils(gabriel, mathile, philippe1).
fils(emmanuel, mathilde, philippe1).
fils(amedeo, astrid, lorenz).
fils(joachim, astrid, lorenz).
fils(maximilian, elisabetta_rosboch, amedeo).
fils(nicolas, claire, laurent).
fils(aymeric, claire, laurent).

fille(charlotte_de_belgique, louise_d_orleans, leopold1).
```

6. https ://fr.wikipedia.org/wiki/Arbre_g%C3%A9n%C3%A9alogique_de_la_famille_royale_belge

```
fille(louise_de_belgique, marie_henriette_d_autriche,
   leopold2).
fille(stephanie_de_belgique, marie_henriette_d_autriche,
   leopold2).
fille(clementine_de_belgique, marie_henriette_de_belgique,
   leopold2).
fille(henriette_de_belgique,
   marie_de_hohenzollern_sigmaringen,
      philippe_de_belgique).
fille(josephine_de_belgique1,
   marie_de_hohenzollern_sigmaringen,
      philippe_de_belgique).
fille(josephine_de_belgique,
   marie_de_hohenzollern_sigmaringen,
      philippe_de_belgique).
fille(marie_jose_de_belgique, elisabeth_en_baviere,
      albert1).
fille(josephine_charlotte, astrid_de_suede, leopold3).
fille(marie_christine_de_belgique, lilian_baels, leopold3).
fille(maria_esmeralda_de_belgique, lilian_baels, leopold3).
fille(astrid, paola, albert2).
fille(elisabeth, mathilde, philippe1).
fille(eleonore, mathilde, philippe1).
fille(maria_laura, astrid, lorenz).
fille(luisa, astrid, lorenz).
fille(laetitia, astrid, lorenz).
fille(anna_astrid, elisabetta_rosboch, amedeo).
fille(louise, claire, laurent).
```

Jusque là, rien de transcendant, toutefois, il est important de noter la simplicité et l'expressivité que nous offre Prolog. Ainsi, graĉe à celle-ci, il est possible pour un lecteur ne connaissant pas ce langage d'en lire un programme, et de comprendre ce qu'il s'y passe sans trop de soucis, ceci est un des plus gros points forts de Prolog.

Passons maintenant à l'écriture de notre premier prédicat :

```
% Maintenant, réflechissons à quelques lois.
parents(Mere, Pere, Enfant) :-
```

```
    fille(Enfant, Mere, Pere) ;
    fils(Enfant, Mere, Pere).
```

Ici, ";" représente l'opérateur de disjonction, nous pouvons lire le prédicat ci-dessus ainsi :

X et Y seront parents de Z si Z est fille de X et de Y ou Z est fils de X et de Y.

Les différents prédicats seront testés à la fin.

```
grands_parents(GrandMere, GrandPere, PetitEnfant) :-
    parents(Mere, Pere, PetitEnfant),
    (   parents(GrandMere, GrandMere, Mere)
    ;   parents(GrandMere, GrandPere, Pere)
    ).
```

Nous noterons ici l'apparition de ",", il s'agit simplement de l'opérateur de conjonction (ou le "et"). La syntaxe choisie pour les clauses est dictée selon cet article :
http ://www.covingtoninnovations.com/mc/plcoding.pdf

```
frere(Frere, FOS) :-
    dif(Frere, FOS), % FOS c'est pour Frère ou Soeur.
    fils(Frere, Mere, Pere),
    parents(Mere, Pere, FOS).

soeur(Soeur, FOS) :-
    dif(Soeur, FOS),
    fille(Soeur, Mere, Pere),
    parents(Mere, Pere, FOS).
fos(Enfant1, Enfant2) :-
    frere(Enfant1, Enfant2) ;
    soeur(Enfant1, Enfant2).
```

Ici, "fos" signifie "frère ou soeur", nous laisserons au lecteur le soin de trouver le fonctionnement des différents prédicats.

```
oncle(Oncle, Neveu) :-
    parents(Mere, Pere, Neveu),
    (   frere(Oncle, Mere)
    ;   frere(Oncle, Pere)
```

```
    ).


tante(Tante, Neveu) :-
    parents(Mere, Pere, Neveu),
    (   soeur(Tante, Mere)
    ;   soeur(Tante, Pere)
    ).
```

Les prédicats “tante” et “oncle” ne sont pas non plus biens compliqués à comprendre.

```
cousin_m(Cous1, Cousin) :-
    parents(Mere, _, Cous1),
    fos(Mere, Parent),
    (   parents(Parent, _, Cousin)
    ;   parents(_, Parent, Cousin)
    ).

cousin_p(Cous1, Cousin) :-
    parents(_, Pere, Cous1),
    fos(Pere, Parent),
    (   parents(Parent, _, Cousin)
    ;   parents(_, Parent, Cousin)
    ).

cousin(Cous1, Cousin) :-
    cousin_m(Cous1, Cousin) ;
    cousin_p(Cous1, Cousin).
```

Le prédicat cousin fut un peu plus fastidieux à écrire. Afin d’y parvenir, nous avons séparé ce prédicat en trois parties, dont deux “prédicats d’aide”. Tout d’abord, nous vérifions si les deux enfants sont cousins du côté maternel via le prédicat “cousin_m”, nous faisons de même, du côté paternel cette fois-ci, avec l’aide du prédicat “cousin_p”, par après, nous utilisons le prédicat cousin, qui nous signale si les deux enfants sont cousins du côté paternel, ou du côté maternel.

```
descendant(Parent, Descendant) :-
    parents(Parent, _, Descendant) ;
```

```
    parents(_, Parent, Descendant).

descendant(Parent, Descendant) :-
    parents(X, Y, Descendant),
    (   descendant(Parent, X)
    ;   descendant(Parent, Y)
    ).
```

Ce prédicat est un prédicat récursif, comme nous le constatons, “descendant” est appelé à l'intérieur de sa propre définition.
Comment cela marche ?
Nous verrons la réponse à cette question lors du prochain chapître sur l'approche algorithmique.

Ainsi, si nous appliquions la méthode du canard plastique[7], voici comment nous devrions lire ce prédicat :

a. Si P est père ou mère de D, D est descendant de P.

b. Si D est descendant de P et que D n'est pas un enfant de P, alors, un des parents de D est descendant de P.

Cela peut paraître assez bizarre pour le lecteur de prime abord, mais c'est à force d'habitude et de travail que l'aisance avec l'écriture de prédicats récursifs se fera ressentir.

Maintenant, passons au test de nos différents prédicats !

```
?- parents(X, Y, philippe1).
X = paola,
Y = albert2.

?- grands_parents(X, Y, philippe1).
X = astrid_de_suede,
Y = leopold3.

?- frere(X, philippe1).
```

7. https ://fr.wikipedia.org/wiki/M%C3%A9thode_du_canard_en_plastique

```
X = laurent ;
false.

?- soeur(X, philippe1).
X = astrid ;
false.

?- oncle(X, philippe1).
X = baudoin1 ;
false.

?- tante(X, philippe1).
X = josephine_charlotte ;
false.

?- cousin(albert1, X).
X = louise_de_belgique ;
X = stephanie_de_belgique ;
X = clementine_de_belgique ;
X = leopold_de_belgique ;
X = lucien_durrieux ;
X = philippe_durrieux ;
false.

?- descendants(albert2, X).
Correct to: "descendant(albert2,X)"? yes
X = astrid ;
X = philippe1 ;
X = laurent ;
X = elisabeth ;
X = eleonore ;
X = maria_laura ;
X = luisa ;
X = laetitia ;
X = anna_astrid ;
X = louise ;
X = gabriel ;
X = emmanuel ;
X = amedeo ;
```

```
X = joachim ;
X = maximilian ;
X = nicolas ;
X = aymeric ;
false.
```

Il est intéressant de noter lors des tests la puissance de l'interprète prolog, permettant même de suggérer des noms corrigés en cas de fautes de frappe !

4 L'approche algorithmique.

4.1 Qu'est-ce qu'un algorithme ?

Un algorithme est une suite d'instructions permettant de résoudre un problème.

Il faut savoir que nous utilisons des algorithmes bien plus souvent que ce que l'on pourrait croire. Par exemple, quand vous préparez un gâteau pour célébrer une quelconque occasion, vous aurez besoin d'une recette. Cette recette n'est autre que l'algorithme aidant à la préparation du gâteau, chaque étape de la recette n'étant qu'une instruction faisant partie de l'algorithme.

Un des tout premiers exemples d'algorithmes est "l'algorithme d'Euclide", celui-ci permettait de trouver le PGCD de deux nombres.
Voici comment celui-ci fonctionne :

- Tout d'abord, nous prenons deux nombres a et b.
- Si b = 0 alors, nous retournons a comme étant le pgcd.
- Sinon, nous écrivons que **pgcd**(a, b) = **pgcd**(b, a **mod** b)

Et en voici sa traduction en Common Lisp, le langage de programmation que nous utiliserons dans cette partie du dossier.

```
(defun pgcd (a b)
  (if (zerop b)
      a
      (pgcd b (mod a b))))
```

4.1.1 La complexité des algorithmes

Évidemment, tous les algorithmes ne se valent pas, certains sont bien plus lents que d'autres, et ce, pour une seule et même tâche.
Pour juger de la **complexité** d'un algorithme, nous utilisons la notation de

Landau (dite du “Big-O” en anglais).

Cette notation a pour but d'estimer l'évolution du nombre d'opérations qui seront effectuées par l'algorithme au cours du temps. Car au plus l'algorithme effectue d'opérations, au plus il sera coûteux en temps, ce qui n'est pas pour nous arranger.

Pour montrer l'importance d'avoir des algorithmes performants, nous avons implémenté en Lisp deux algorithmes de tris différents :

— Le **tri à bulles** de complexité $O(n^2)$
— Le **tri par fusion** de complexité $O(n \log n)$

Le **tri à bulles** fonctionne de la manière suivante :
— Prendre les deux premiers éléments de la liste.
— Si le premier est plus grand que le second, les échanger, sinon, les laisser en place.
— Faire de même jusqu'à la fin de la liste.
— Une fois arrivé à la fin de la liste, reprendre à partir du début de la liste.
— Continuer ce procédé jusqu'à ce que la liste soit parfaitement rangée.

Il est implémenté de la manière suivante en Common Lisp :

```
(defun bubble-sort (lat)
  "Implémentation non-récursive, avec copie,
   de l'algorithme du tri à bulles."
  (let ((arr (make-array (length lat) :initial-contents lat
   )))
    (loop for i from (1- (length arr)) downto 1
            do (loop for j from 0 to (1- i)
                     when (< (aref arr (1+ j)) (aref arr j))
                         do (rotatef (aref arr (1+ j))
```

```
                                    (aref arr j))))
    arr))
```

Le **tri par fusion** marche de manière assez différente :

- Si le tableau n'a qu'un seul élément, il est considéré comme déjà trié.
- Si il a plus d'un élément, séparer le tableau en deux parties à peu près égales.
- Trier les deux parties ainsi séparées.
- Fusionner les deux tableaux triés en un seul tableau trié.

La fonction merge étant prédéfinie en common Lisp, nous implémenterons notre tri par fusion ainsi.

```
(defun merge-sort (arr)
  "Implémentation du tri par fusion,
en utilisant la fonction merge, prédéfinie."
  (if (<= (length arr) 1)
      arr
      (let* ((middle (floor (length arr) 2))
             (left (subseq arr 0 middle))
             (right (subseq arr middle (length arr))))
          (merge 'list (merge-sort left)
                 (merge-sort right) #'<))))
```

Passons maintenant aux **tests de performances**.

Afin de mesurer le temps pris par chacun de mes deux algorithmes, j'ai utilisé "time", une macro bien pratique pour ce genre de tests. J'utilise le compilateur SBCL pour faire mes tests. En plus de cela, dans un souci de facilité, j'utilise la macro random-sample, créée par mes soins, me permettant de créer une liste contenant des nombres aléatoires, de taille fixée.

```
;;; Ma macro random-sample bien pratique.
(defmacro random-sample (x)
   `(loop for _ below ,x collect (random 1000)))
```

Sans plus attendre, voici le tableau montrant les résultats de notre petite expérience.

Nombre d'éléments à triers.	Temps Tri à Bulles.	Temps Tri par fusion.
10	0.0000008 secondes	0.000016 secondes
100	0.000089 secondes	0.000140 secondes
1000	0.007863 secondes	0.001188 secondes
10,000	0.318904 secondes	0.012558 secondes
100,000	31.164620 secondes	0.090092 secondes

Pour des raisons pratiques, je n'ai pas pu obtenir les résultats du Tri à Bulles pour le million d'éléments.

Comme nous pouvons le constater sur le tableau précédent, la complexité d'un algorithme (estimé par le Big-O) ne mesure pas le temps exact que l'algorithme prendra afin de trier une certaine liste, il s'agit simplement d'un indicateur, montrant comment évoluera le nombre de comparaisons au cours du temps, cet indicateur devenant extrêmement significatif quand n devient très grand.

4.2 Présentation de structures de données de base

Pour produire une intelligence artificielle efficace, il nous faut de bons algorithmes, mais il nous faut également les structures de données adéquates.

Tout d'abord, qu'est-ce qu'une structure de données ? D'après Wikipedia [8], une structure de données est une manière d'organiser les données, pour les traiter plus facilement.
On conclut donc que le choix judicieux d'une structure de données est indispensable à l'optimisation de la performance de notre IA.

Je présente ici trois structures de données de base.

a. Les listes simplement chaînées.

b. Les tableaux (ou arrays en anglais.)

8. https ://fr.wikipedia.org/wiki/Structure_de_donn%C3%A9es

c. Les tables de hachages (ou les dictionnaires.)

4.2.1 Les listes simplement chaînées.

Une **liste simplement chaînée** est une structure de données pouvant contenir plusieurs éléments. Chaque élément appartenant à la liste chaînée contient deux choses :

a. La valeur de l'élément.

b. Un pointeur pointant vers l'élément suivant.

Il est possible de représenter une liste simplement chaînée ainsi :

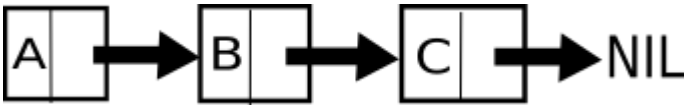


FIGURE 1 – Liste simplement chaînée.

En Lisp, une liste simplement chaînée peut être crée de la manière suivante.

```
(defparameter *my-linked-list* '(1 2 3 4 5))
```

Ainsi, si nous reprenions le dessin que nous avions utilisé ci-dessus, *my-linked-list* ressemblera donc à ceci :

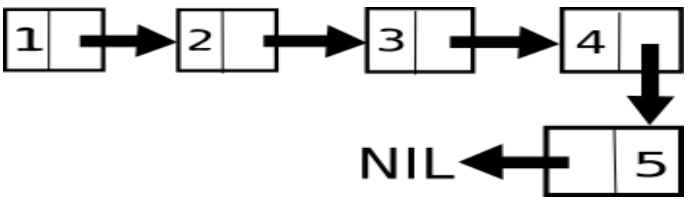


FIGURE 2 – *my-linked-list*

D'ailleurs, pour pouvoir travailler avec des listes chaînées, deux opérateurs bien pratiques s'offrent à nous : "car" et "cdr".

```
CL-USER> (car *my-linked-list*)
1
CL-USER> (cdr *my-linked-list*)
(2 3 4 5)
```

Nous pouvons également chaîner nos opérateurs "car" et "cdr", afin de nous assurer un contrôle total sur notre liste simplement chaînée.

```
CL-USER> (car (cdr *my-linked-list*))
2
CL-USER> (cadr *my-linked-list*) ; Il existe aussi une mani
    ère abrégée.
2
```

Il existe également la fonction "nth", permettant de récupérer le n-ième élément d'une liste chaînée. (Il ne faut pas oublier que l'indexing marche comme pour les tableaux, et commence à 0.)

```
CL-USER> (nth 2 *my-linked-list*)
3
```

4.2.2 Les tableaux.

Le tableau est une structure extrêmement importante en programmation, celle-ci nous permet de stocker un nombre fixe de données, et nous permet d'accéder à ces données de manière rapide. Chaque élément du tableau se retrouve collé en mémoire aux éléments qui lui sont adjacents, ainsi, pas besoin de traverser chaque élément du tableau lorsque l'on désire accéder au n-ième élément du tableau.

Voici comment créer un tableau en Lisp :

```
CL-USER> (defparameter *my-array* #(1 2 3 4 5))
```

Il existe également une manière alternative :

```
CL-USER> (defparameter *my-array* (make-array 5 :
  initial-contents '(1 2 3 4 5)))
```

Pour récupérer le n-ième élément d'un tableau, il nous suffit de faire :

```
CL-USER> (aref *my-array* 2)
3
```

4.2.3 Tableaux vs Listes simplement chaînées.

Pour en conclure avec ces deux structures de données, je me propose de faire un petit tableau comparatif final, afin que le lecteur puisse mieux comprendre les différences entre Tableau, et Liste simplement chaînée, (que je noterai SLL pour Single-linked list).

	Tableaux	SLL
Accession	O(1)	O(n)
Insertion	O(n)	O(1)
Délétion	O(n)	O(1)

Comme on le constate sur ce tableau, les arrays ont un réel avantage lorsque nous désirons uniquement lire un élément de notre structure de données, toutefois, si nous désirons modifier la structure de données en elle-même, alors, tout devient plus problématique, et cela, car notre tableau devra retrouver un autre emplacement libre pour s'y mettre en mémoire.

La liste chaînée quant à elle, n'a pas besoin de se repositionner entièrement lorsqu'on lui ajoute ou qu'on lui retire un élément, il lui suffit plutôt de rajouter un pointeur vers le nouvel élément, ce qui se fait en temps constant (O(1)), peu importe la taille de la liste. Toutefois, la liste chaînée a pour inconvénient d'être en croissance linéaire lors de la recherche d'un élément dans la liste, en effet, lors de la recherche d'un élément dans une liste chaînée, le programme devra d'abord passer par tous les éléments se trouvant avant celui que l'on cherche.

Pour démontrer l'importance d'utiliser la bonne structure au bon endroit, nous allons reprendre notre bon tri à bulles.

Pour ceux qui avaient oublié l'implémentation sur les tableaux, la voici :

```
(defun bubble-sort (lat)
  "Implémentation non-récursive, avec copie,
   de l'algorithme du tri à bulles."
  (let ((arr (make-array (length lat) :initial-contents lat
   )))
    (loop for i from (1- (length arr)) downto 1
            do (loop for j from 0 to (1- i)
                     when (< (aref arr (1+ j)) (aref arr j))
                         do (rotatef (aref arr (1+ j))
                                     (aref arr j))))
    arr))
```

Le lecteur attentif remarquera l'utilisation dans ce code de "make-array" et de "aref", tout deux synonymes de l'utilisation d'un tableau. Maintenant, transformons cette implémentation en une implémentation utilisant des listes simplement chaînées.
Et maintenant, voici le même algorithme, mais, sur les listes simplement chaînées cette fois-ci.

```
(defun sll-bubble-sort (lat)
  "Implémentation non-récursive, avec copie,
   de l'algorithme du tri à bulles,
  sur des listes simplement chaînées"
  (let ((lat-copy (copy-list lat)))
    (loop for i from (1- (length lat-copy)) downto 1
            do (loop for j from 0 to (1- i)
                     when (< (nth (1+ j) lat-copy) (nth j
   lat-copy))
                         do (rotatef (nth (1+ j) lat-copy)
                                     (nth j lat-copy))))
    lat-copy))
```

Et maintenant, comparons leurs temps respectifs sur des listes de tailles différentes avec l'aide d'un nouveau tableau, (mon mode opératoire reste le même, j'utilise SBCL, et la macro time, pour créer une liste contenant des nombres aléatoires, j'utilise toujours ma macro random-sample.)

	Tri à bulles (Tableaux)	Tri à bulles (SLL)
10	0.0000008 secondes	0.000009 secondes
100	0.000089 secondes	0.002111 secondes
1000	0.007863 secondes	0.750918 secondes
10000	0.318904 secondes	888.190640 secondes
100000	31.164620 secondes	>1000 secondes

La différence de performance entre nos deux implémentations du même algorithme est énorme, et pourtant, nous n'avons pas changé l'algorithme en lui-même ! J'espère que notre petite expérience montre bien au lecteur l'importance d'un choix de structure de données adéquat.

4.2.4 Les tables de hachages.

Une table de hachage (ou un dictionnaire en python) est une structure de données des plus utiles en ce qui concerne l'optimisation d'algorithmes.

Son utilisation est simple ; avec une table de hachage, nous relions des clés avec des valeurs.

Ainsi, si dans une table de hachage, je relie le mot "bonjour" avec le mot "hello", mon mot "bonjour" sera considéré comme étant la clé, et le mot "hello" comme étant la valeur.

En Lisp, pour créer une telle table de hachage, il faut faire :

```
CL-USER> (defparameter *my-dict* (make-hash-table))
*MY-DICT*
CL-USER> (setf (gethash 'bonjour *my-dict*) 'hello)
HELLO
CL-USER> (gethash 'bonjour *my-dict*)
HELLO
```

Ce procédé étant toutefois un peu fastidieux, nous utiliserons ici la bibliothèque lisp connue sous le nom de serapeum.

```
CL-USER> (ql:quickload :serapeum) ;; Ici, nous déclarons la
    bibliothèque.
To load "serapeum":
  Load 1 ASDF system:
    serapeum
; Loading "serapeum"
.
Switching to the BALLAND2006 optimizer

(:SERAPEUM)
CL-USER> (defparameter *my-dict* (serapeum:dict 'bonjour '
   hello))
*MY-DICT*
CL-USER> (gethash 'bonjour *my-dict*)
HELLO
```

Pour montrer un cas pratique d'utilisation de tables de hachages, nous allons créer un mini-programme qui permet de convertir des phrases en morse.

Ce programme se fait en deux parties, tout d'abord, nous définissons une table de hachage qui nous remet la traduction morse de chaque lettre et de chaque chiffre.

```
(ql:quickload :serapeum)

(defparameter *latin->morse*
  (serapeum:dict #\a "._" #\b "_..." #\c "_._." #\d "_.."
                 #\e "." #\f ".._." #\g "__." #\h "...."
                 #\i ".." #\j ".___" #\k "_._" #\l "._.."
                 #\m "__" #\n "_." #\o "___" #\p ".__."
                 #\q "__._" #\r "._." #\s "..." #\t "_"
                 #\u ".._" #\v "..._" #\w ".__" #\x "_.._"
                 #\y "_.__" #\z "__.." #\1 ".____" #\2 "..
   ___"
                 #\3 "...__" #\4 "...._" #\5 "....." #\6 "_
   ...."
                 #\7 "__..." #\8 "___.." #\9 "____." #\0 "
   _____"
```

```
                    #\SPACE " "))
```

Voici notre dictionnaire ! Ne soyez pas surpris par les “#’, c’est ainsi que nous définissons des caractères en common lisp.

Maintenant, la deuxième partie consistera simplement à créer une fonction chargée de transformer nos phrases en morse.

```
(defun convert-to-morse (sentence)
  "Cette fonction s'occupe de convertir
nos phrases vers du code Morse, grâce au dictionnaire
créé ci-dessus!"
  (let ((converted-list
         (loop for letter across sentence
               collect (gethash letter *latin->morse*))))
  (format nil "~{~A~}" converted-list)))
```

Il ne nous reste plus qu’à tester.

```
CL-USER> (convert-to-morse "ceci est une phrase")
"-.-.··-.-··· ···· · ··--·· ·--·····-··-····"
```

Et voilà comment notre programme marche !

Petite parenthèse au niveau de la complexité de notre fonction “convert to morse”. Tout d’abord, il faut savoir que pour chercher une valeur avec une clé dans un dictionnaire, la complexité temporelle est de O(1) ! Peu importe la taille du dictionnaire, cela prendra toujours aussi peu de temps de chercher une clef dedans, c’est la raison pour laquelle ceux-ci sont très utilisés de nos jours. Pour les lecteurs intéressés, nous conseillons fortement de vous renseigner sur la fonction SHA, c’est grâce à cette fonction de hachage que les dictionnaires marchent aussi bien.

Ainsi, pour en revenir à notre programme, nous appliquons une opération de complexité O(1) sur chaque élément de notre liste, soit, n fois. On peut donc en déduire que notre fonction “convert-to-morse” est donc de complexité O(n), car nous appliquons n fois une opération de complexité O(1).

4.2.5 Promenons-nous dans les bois.

Pour comprendre l'intérêt des arbres dichotomiques en programmation, intéressons nous à un petit problème assez simple.

Imaginez que je pense à un nombre, compris entre 1 et 1000, et que vous deviez deviner ce nombre avec le moins d'essais possible, à la même manière que dans le jeu du "Juste Prix", je vous dirai si votre essai est trop grand, ou trop petit.

La première stratégie évidente serait de commencer par 1 ; ensuite, si mon nombre n'est pas 1, essayer avec 2 ; puis, essayer avec 3, et ceci, jusqu'à ce que vous trouviez le nombre auquel je pense.

Cette première stratégie pourrait faire penser à la manière à laquelle il faut chercher un élément dans une liste simplement chaînée, en effet, si je pense au nombre 629, il va falloir passer par les 628-ièmes éléments se trouvant avant. Dans ce cas-ci, comme pour lire un élément dans une liste simplement chaînée, nous dirons que la complexité de cet algorithme sera de O(n).

Une deuxième manière de faire serait de faire ce que l'on appelle une recherche dite "par dichotomie". Tout d'abord, prenons un nombre au milieu entre 1 et 1000, ici, ce sera 500, si je dis "au dessus", alors, il suffira de prendre le milieu entre 500 et 1000, ici, ce sera 750, désormais, si je dis "en dessous", il faudra alors faire $\frac{500+750}{2} = 625$, ainsi, en appliquant cet algorithme jusqu'au bout, vous trouverez le nombre auquel je pense en utilisant un algorithme de complexité O(log n) !

Ainsi, vous trouverez grâce à cette technique le nombre auquel je pense en maximum $\log_2 n$ essais (où n est le nombre maximum, qui est ici 1000).

J'espère que cette petite explication aura pu expliquer au lecteur le fonctionnement d'une recherche par dichotomie. Maintenant, pour ce qui est de l'implémentation d'une telle recherche en Lisp :

```
(defun binary-search (l-bound u-bound &optional (tries 0))
  (let ((guess (floor (+ l-bound u-bound) 2)))
    (format t "~&~D?~&" guess)
    (case (read)
      (plus-haut (binary-search guess u-bound (1+ tries)))
      (plus-bas  (binary-search l-bound guess (1+ tries)))
      (t         (format t "Ton nombre a été trouvé en ~D
   essais" tries)))))
```

Plus qu'à la tester !

```
[1] CL-USER> (binary-search 1 1000)
500?
plus-haut
750?
plus-bas
625?
plus-haut
687?
plus-bas
656?
plus-bas
640?
plus-bas
632?
plus-bas
628?
plus-haut
630?
plus-bas
629?
oui
Ton nombre a été trouvé en 9 essais
```

Cette petite introduction à la recherche par dichotomie servait surtout à

donner une intuition au lecteur de l'importance des arbres binaires.

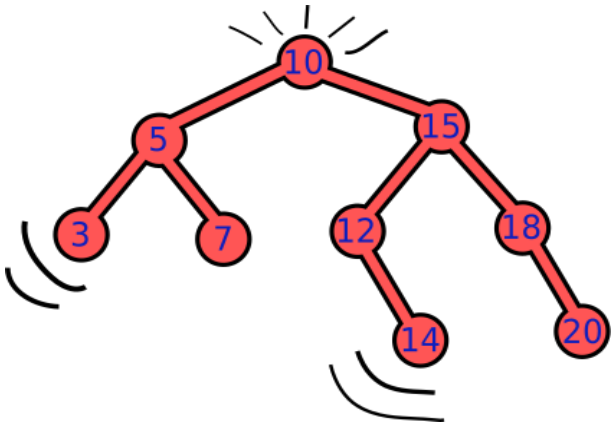
Ceux-ci fonctionnent d'une manière similaire à notre recherche dichotomique.

Sur un arbre binaire, chaque noeud est soit relié à rien (ou "nil"), soit relié à sa "gauche" à un noeud contenant une valeur plus petite que celle de son noeud parent, soit relié à sa "droite" à un noeud contenant une valeur plus grande que celle de son noeud parent. Nous pouvons donc représenter un arbre binaire ainsi :

en lisp, pour définir un arbre binaire, nous ferons simplement :

```
(defparameter *my-tree* '(10 (5 3 7) (15 (12 nil 14) (18
    nil 20))))
```

Celui-ci sera représenté ainsi :



4.2.6 Les tas

Nous faisons ci-dessous un petit aparté pour parler de tas.

Un tas n'est autre qu'une forme spéciale d'arbre où les valeurs sont ordonnées selon une certaine fonction "clef".

L'avantage du tas est avant tout que la complexité de la fonction pour récupérer l'élément au sommet du tas n'est autre que $O(1)$!

Un tas s'avère donc extrêmement pratique lorsque nous désirons garder des valeurs ordonnées, sans avoir à retrier la structure de donnée à chaque nouvel ajout de données.

Nous verrons comment utiliser les tas en Lisp dans la partie sur l'implémentation de l'algoithme A*.

4.3 Qu'est-ce que Lisp ?

Lisp est une famille de langages de programmation fonctionnels, inventés en 1958 par John McCarthy (l'homme ayant inventé le terme "intelligence artificielle").

Ceux-ci sont reconnaissables facilement grâce au très grand nombre de parenthèses présentes dans chacun des dialectes de Lisp.

Aujourd'hui, le dialecte Lisp le plus utilisé reste **Clojure**, toutefois, il en reste d'autres gardant toujours leur cote de popularité, pour n'en citer que quelques-un, nous avons

- **Scheme**, qui est un dialecte très minimaliste de Lisp, très utilisé au niveau académique.
- **Emacs Lisp**, un dialecte très pratique pour tous ceux désirant configurer l'éditeur de texte Emacs.
- **Racket**, un super-set de Scheme.
- **Common Lisp** est le dialecte que nous utiliserons ici, ce dialecte a toujours eu la réputation d'être plus orienté vers les applications pratiques, et n'a jamais reçu de grandes faveurs académiques. Common Lisp a toutefois le mérite d'avoir été standardisé, de bénéficier d'un système orienté-objet connu sous le nom de CLOS.

4.3.1 Les bases de Lisp.

Tout d'abord, pour commencer notre aventure avec Lisp, ouvrons donc notre REPL (cela peut-être SBCL, Clisp, ou encore CCL), et additionnons deux nombres.

```
CL-USER> (+ 3 5)
8
```

Cela peut sembler bien étrange pour le non-initié, et pourtant, tout cela est parfaitement logique.

En fait, lorsque l'on désire appeler une fonction (comme par exemple ici "+"), la syntaxe sera toujours "(**fonction arguments**)". Prenons quelques autres exemples avec quelques opérations arithmétiques.

```
CL-USER> (- 2 1)
1
CL-USER> (* 3 4 2)
24
CL-USER> (/ 5 6)
5/6
CL-USER> (/ 5 6.0)
0.8333333
```

Comme on peut le voir, la logique reste la même pour tous les opérateurs, aussi, cette syntaxe nous permet également de mettre à ces opérateurs autant d'arguments que l'on le souhaite, (l'exemple est donné ci-dessus avec l'opérateur de multiplication.).

Également, l'opération de division ne remet pas directement un nombre à virgule, mais plutôt un Ratio, si l'on ne lui donne pour argument uniquement des nombres entiers.

Pour déclarer une variable en Lisp, plusieurs choix s'offrent à nous, je présenterai uniquement ici "defparameter".

```
CL-USER> (defparameter *test* 'bonjour)
*TEST*
CL-USER> *test*
BONJOUR
```

J'en profite pour attirer l'attention sur l'utilisation de "*" autour du nom de la variable. Ceux-ci sont appelés les "cache-oreilles", il ne s'agit ici que d'une convention, il est tout à fait acceptable de ne pas utiliser de caches-oreilles lors de la définition d'une variable globale.

Maintenant, j'utiliserai la fin de cette petite introduction comme prétexte pour présenter "la récursion".

Pour montrer un exemple de fonction récursive, je présenterai ici la fonction factorielle, fonction récursive par excellence.

Tout d'abord, pour déclarer une fonction, nous utiliserons le mot-clé **defun**.

```
CL-USER> (defun factorial (x) ...)
```

Il faut ensuite réfléchir à ce qu'il faudra mettre au sein de notre fonction.

La récursion est un processus en 3 étapes.

a. Trouver la valeur de f(0) (ou de n'importe quel cas de base.)
b. Supposer que cette fonction remettra un nombre correct pour f(n-1).
c. Trouver la valeur de f(n) en fonction de f(n-1).

L'exemple ici est assez simple, et est souvent donné en introduction à la récursion. Voici donc une définition formelle de notre fonction factorielle.

$$\begin{cases} 0! = 1 \\ n! = n \cdot (n-1)! \end{cases}$$

Ainsi, uniquement grâce à cette simple définition, nous allons pouvoir coder

cette fonction en Common Lisp

```
CL-USER> (defun factorial (x)
                    (if (= x 0)
                        1
                       (* x (factorial (- x 1)))))
FACTORIAL
```

On constate directement la simplicité et l'élégance avec laquelle cette fonction peut être ainsi programmée. D'autres fonctions peuvent-être codée de manière récursive, l'exemple de l'algorithme de tri par fusion, ou encore l'agorithme d'Euclide pour calculer le pgcd sont deux exemples de fonctions récursives dont j'ai pu montrer l'implémentation ci-dessus.

Un autre exemple d'algorithme utilisant la récursion serait l'algorithme "quick-sort" dit du "tri-rapide", en voici son implémentation

```
(defun quick-sort (lat)
  "Implémentation récursive de l'algorithme de tri-rapide."
  (if lat
      (let ((partition (car lat))
            (rest (cdr lat)))
      (nconc
       (quick-sort (remove-if (lambda (a)
                                (>= a partition))
                              rest))
       (remove-if (lambda (a) (/= a partition)) lat)
       (quick-sort (remove-if (lambda (a)
                                (<= a partition))
                              rest))))
    nil))
```

Si le lecteur en sent l'envie, nous l'encourageons à essayer de coder la fonction récursive d'Ackermann, ou encore la McCarthy 91.

4.3.2 Métaprogrammons !

Cette petite partie a pour objectif de montrer en quoi Lisp est si bon : **La métaprogrammation** ! La métaprogrammation, c'est l'art de faire des programmes qui manipulent le code du programme même !

Passons directement aux exemples :

En Common Lisp, il est possible de créer des variables locales avec l'aide du mot-clé **let**. Par exemple, si je désire créer de manière non récursive la fonction :

$$f(n) = \sum_{i=0}^{n} i$$

Je devrai faire :

```
(defun f (n)
  (let ((buffer 0))
    (loop for i from 1 to n
          do (incf buffer i))
    buffer))
```

Toutefois, certains ne trouvent pas très esthétique le fait de devoir empiler des parenthèses à chaque fois qu'un très petit **let** est appelé.

Pour remédier à cela, il est possible de modifier la manière à laquelle notre code Lisp est lu !

Ainsi, en créant la macro :

```
(defmacro let-in (var val &body body)
  `(let ((,var ,val))
     ,@body))
```

Il devient possible de modifier notre fonction de base :

```
(defun g (n)
  (let-in buffer 0
    (loop for i from 1 to n
          do (incf buffer i))
    buffer))
```

Tout de suite, le code devient (pour certains) plus agréable à lire.

En réalité, notre nouvelle fonction est exactement la même que l'ancienne ! La macro que nous avions créée se fait remplacer par le préprocesseur.

```
CL-USER> (macroexpand '(let-in buffer 10 (+ buffer 5)))
(LET ((BUFFER 10))
  (+ BUFFER 5))
```

Le code ci-dessus illustre la manière à laquelle le préprocesseur évalue notre macro.

Un autre exemple de macro sont les Reader Macros, celle-ci ont pour objectif de complètement modifier la syntaxe même de Common Lisp.

Un exemple de Reader Macro, serait par exemple une pour créer des lambdas de manière plus simple :

Sans plus attendre, voici la macro :

```
(set-macro-character #\> (get-macro-character #\)))

(set-macro-character
 #\<
 (lambda (stream char)
   (declare (ignore char))
   (let ((contents (read-delimited-list #\> stream t)))
     `(lambda (_)
        ,contents))))
```

Et voici comment elle s'utilise :

```
CL-USER> (mapcar <1+ (* 2 _)> '(1 2 3 4))
(3 5 7 9)
CL-USER> (macroexpand '<1+ (* 2 _)>)
#'(LAMBDA (_) (1+ (* 2 _)))
T
```

Le lecteur intéressé pour s'essayer au fonctionnement des macros anaphoriques, qui ne sont pas présentées dans cette introduction.

4.3.3 Programme Lisp pour résoudre parfaitement le jeu de Nim.

Afin que le lecteur se fasse une idée de comment utiliser Lisp dans des cas plus concrets, nous allons vous présenter une intelligence artificielle, assez basique, gagnante au jeu de Nim à plusieurs tas. (Aussi connu sous le nom de jeu de Marienbad.), dans ce jeu, plusieurs allumettes sont disposées sur plusieurs piles, le but du jeu est de récolter la dernière allumette.

L'intelligence gagnante à ce genre de jeux est assez basique, en effet, le jeu de Nim à plusieurs tas est un jeu résolu parfaitement.

La stratégie gagnante pour ce jeu est simple, nous définissons une position gagnante une position où la"nim-somme" (qui est en réalité l'opérateur "xor" ou $\oplus$ appliqué à la représentation binaire de notre tas) est différent de 0, et à l'inverse, une position perdante est une position où la "nim-somme" appliquée aux tas est égale à 0.

Un exemple de position perdante serait la position où les allumettes seraient réparties sur 4 tas avec 1 allumette sur le premier tas, 3 sur le deuxième, 5 sur le 3ième, et 7 sur le dernier.

$$1 \oplus 3 \oplus 5 \oplus 7 = 0$$

Pour se le prouver, on peut utiliser notre interprète Lisp et faire

```
CL-USER> (logxor 1 3 5 7)
0
```

Le but de l'ordinateur sera donc de toujours se trouver en position de sécurité, et de toujours mettre le joueur en position perdante.

Le twist que nous avons apporté est le fait que le joueur peut décider des tas avec lesquels ils va jouer, et ainsi, mettre directement l'ordinateur en position de défaite, notre IA devra donc se montrer capable de jouer des coups, même en perdant.

```
(defparameter *heaps* nil
  "*heaps* représente les différents tas avec lesquels nous
    allons jouer.")
```

Ici, je ne fais que définir une variable globale *heaps*, celle-ci représentera les tas disponibles au travers de mon programme.

```
(defun unsafep (heaps)
  "Vérifie si la nim-somme des tas est égale à 0."
  (= 0 (reduce #'logxor heaps)))

(defun emptiedp (heaps)
  "Vérifie si les tas sont vides."
  (= 0 (reduce #'+ heaps)))

(defun winningp (num index heaps)
  "Vérifie si une position serait hypothétiquement gagnante
    ."
  (let ((safe-copy (copy-list heaps)))
    (decf (nth index safe-copy) num)
    (unsafep safe-copy)))

(defun heap-print (heaps)
  (format t "~&~{TAS ~D: ~D allumettes.~%~}"
          (loop for number from 1 to (length *heaps*)
                for heap in heaps
```

```
                collect number
                collect heap)))
```

Ces deux fonctions seront dites des “fonctions d’aide”, elles auront surtout pour utilité de ne pas avoir à surcharger les fonctions principales.

Le lecteur attentif remarquera que je termine le nom de ces fonctions par un “p”, cela est dû au fait que ces fonctions sont des **prédicats**, elles ne peuvent retourner que “t” ou “nil” (“vrai” ou “faux”), ceci n’étant également qu’une convention, si le lecteur désire écrire des prédicats sans laisser de “p” au bout, SBCL ne lui trouvera rien à redire.

Aussi, on peut remarquer dans les deux premières fonctions l’utilisation de “reduce”, il s’agit d’une fonction destinée aux **catamorphismes**, cela veut dire qu’elles permettent de passer d’une structure de données comme une liste, ou un arbre, vers un scalaire, comme un entier ou encore un flottant.

Egalement, j’utilise # , ce symbole nous permet d’expliciter que nous utilisons une fonction, et pas une variable.

```
(defun play-random-legal-move (heaps)
  "Si l'adversaire est en position gagnante, nous prendrons
    un nombre au hasard
d'allumettes sur le premier tas à disposition."
  (let ((unchanged t))
    (loop for heap in heaps
          for heap-number from 1
          while unchanged
          when (< 0 heap)
            do (let ((number-of-matches-c
                      (1+ (random heap))))
                 (format t "L'ordinateur a décidé de prendre
    ~D allumettes du tas ~D~%"
                         number-of-matches-c heap-number)
                 (decf (nth (1- heap-number) heaps)
   number-of-matches-c)
```

```
                (setf unchanged nil)))
    heaps))
(defun play-winning-move (heaps)
  "Si l'adversaire n'est pas en poistion gagnante, nous
   jouons alors le meilleur
coup disponible."
  (let ((unchanged t))
    (loop for heap in heaps
          for i from 0
          while unchanged
          do (loop for num from 0 to heap
                   when (winningp num i heaps)
                     do (progn
                          (format t "L'ordinateur a décidé
   de prendre ~D allumettes du tas ~D~%"
                                  num (1+ i))
                          (decf (nth i heaps) num)
                          (setf unchanged nil)))))
  heaps)
```

Nous voici enfin au cœur du programme. Avec nos fonctions "play-winning-move" et "play-random-legal-move" qui sont nos deux fonctions où est contenu notre joueur, les commentaires explicitant déjà leurs actions, je me concentrerai surtout sur l'implémentation.

Comme nous le remarquons, j'utilise **let** et **loop**, **let** me permet de déclarer des variables dites "locales", cela me permet d'avoir un code propre, où chaque chose reste à sa place.

loop est une macro, qui permet d'itérer, ou de répéter un nombre certain de fois une même action, je l'utilise ici en raison de la lisibilité de celle-ci, certains préfèrent la macro "do", je trouve cette dernière illisible, mais j'encourage le lecteur à se renseigner sur cette dernière si le coeur le lui en dit.

```
(defun c-move (heaps)
  "Cette fonction fait office d'aiguilleur,
| Les tas sont vides -> On arrête de jouer.
```

```
| Nous sommes dans une poisition perdante -> On joue un
   coup au hasard.
| Nous gagnons -> On joue le meilleur coup disponible."
  (cond
    ((emptiedp heaps)
     (format t "Félicitations, vous avez gagné!~%"))
    ((unsafep heaps)
     (play-random-legal-move heaps))
    (t (play-winning-move heaps))))
```

Je ne ferai remarquer ici que l'utilisation de la macro **cond**, une autre manière d'écrire des conditions en Lisp.

```
(defun heaps-repl ()
  "Cette fonction récursive est la fonction que le joueur
   voit lorsqu'il joue."
  (when (emptiedp *heaps*)
    (format t "Pas de chance, l'ordinateur a gagné.~%A une
   prochaine fois peut-être."))
  (format t "C'est votre tour, de quel tas désirez vous
   retirer des allumettes? ")
  (heap-print *heaps*)
  (princ 'PLAYER> )
  (let ((player-heap-choice (read)))
    (if (or (> player-heap-choice (length *heaps*))
            (>= 0 player-heap-choice))
        (progn (format t "Veuillez choisir un tas compris
   entre 1 et ~D~%" (length *heaps*))
               ;; Remarquez l'appel récursif à heaps-repl
               ;; si le joueur tente une mauvaise entrée.
           (heaps-repl))
      (progn
        (format t "Combien d'allumettes désirez-vous
   retirer de ce tas?~%")
        (princ 'PLAYER>)
        (let ((player-heap-choice (1- player-heap-choice))
              (number-of-matches (read)))
          (if (or (< (nth player-heap-choice *heaps*)
   number-of-matches)
                  (>= 0 number-of-matches))
```

```
                (progn (format t "Veuillez choisir un
    nombre correct d'éléments.~%")
                       ;; Encore un appel récursif à
    heaps-repl!
                       (heaps-repl))
                ;; Si nous parvenons à passer à travers
    toutes les conditions,
                ;; alors, nous modifions la valeur des tas.
                (decf (nth player-heap-choice *heaps*)
    number-of-matches)))))))
```

Cette très longue fonction fait office d'interface utilisateur. Nous avons déjà fait remarqué au lecteur en commentaires l'utilisation de la récursion.

Egalement, nous signalons **progn**, une fonction bien pratique pour définir des blocs de code.

```
(defun nim-game-repl ()
  "La fonction de base pour commencer une partie."
  (unless *heaps*
    (format t "Avec quels tas voulez-vous jouer? ")
    (setf *heaps* (read-from-string
                   (format nil "(~A)" (read-line))))
    (format t "Très bien, que le jeu commence!~%"))
  (heaps-repl) ;; <- Appel de heaps-repl (tour du joueur).
  (setf *heaps* (c-move *heaps*)) ;; <- Appel de c-move (
   tour de l'ordinateur).
  (unless (emptiedp *heaps*)
    (nim-game-repl)))
```

Nous finissons enfin notre programme avec cette fonction nim-game-repl, permettant de commencer une partie entre humain et ordinateur.

4.4 Le pathfinding.

Le "pathfinding" (ou la "recherche de chemin" en français) est un ensemble de problèmes dans le domaine de l'intelligence artificielle où l'objectif est de trouver le chemin optimal entre deux points dans un graphe.

4.4.1 Présentation du problème.

Ici, le problème que nous allons essayer de résoudre est de trouver le chemin optimal entre deux villes belges.

Les villes belges que nous utilisons ici sont répertoriées dans le document CSV présent à l'adresse suivante : https://github.com/spatie/belgian-cities-geocoded, comme vous le constatez, ce fichier fait plus de 2500 éléments, il va donc falloir être judicieux dans le choix des structures de données, et des algorithmes utilisés afin d'espérer avoir une implémentation un minimum correcte.

4.4.2 L'algorithme de Dijkstra.

Cet algorithme de recherche, inventé par Edsger Dijkstra dans les années 50, marche de la manière suivante :

Tout d'abord, pour représenter l'espace dans lequel notre algorithme doit chercher, nous utiliserons un graphe avec des poids, en voici un exemple :

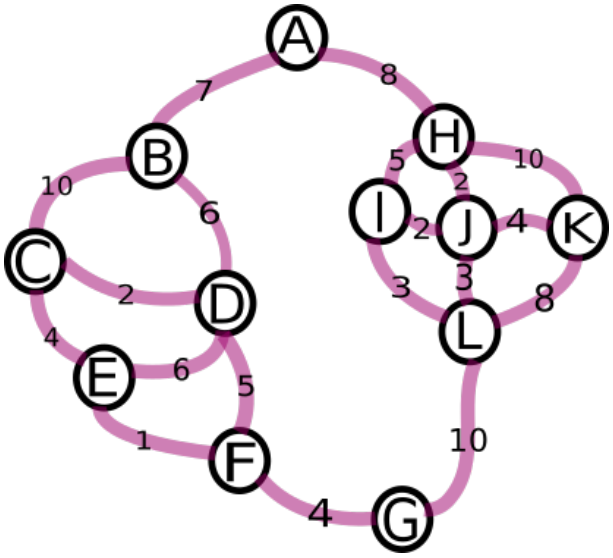


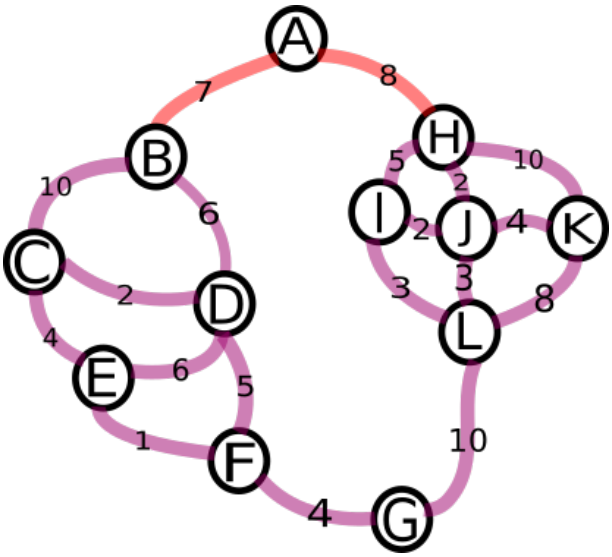
FIGURE 3 – Graphe avec poids.

Ainsi, imaginons que nous cherchons le chemin optimal entre la ville A et la ville G. Nous allons entreprendre les recherches avec l'aide de l'algorithme de Dijkstra.

Cet algorithme se déroule en plusieurs parties, tout d'abord listons l'ensemble des villes joignables directement depuis notre première ville.

Dans notre graphe ci-dessus, les villes B et C sont directement accessible via A.

Voici les chemins marqués sur le graphe :

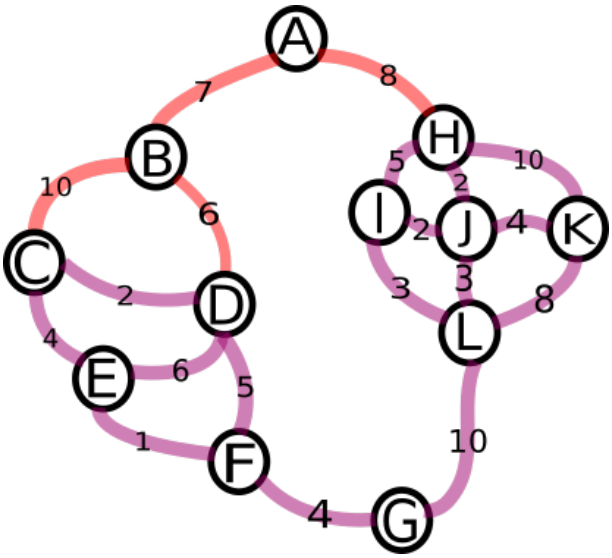


Ainsi, nous pouvons créer une file des priorités, reprenant les villes visitées et les triant en fonction du coût nécessaire pour y accéder (cette file des priorités sera implémentée avec l'aide d'un tas).

— Pour B, ce coût sera de 7.
— Pour H, ce coût sera de 8.

Il faut donc visiter en priorité la ville B !

À partir de B, nous pouvons visiter les villes D et C, voici les chemins empruntés :



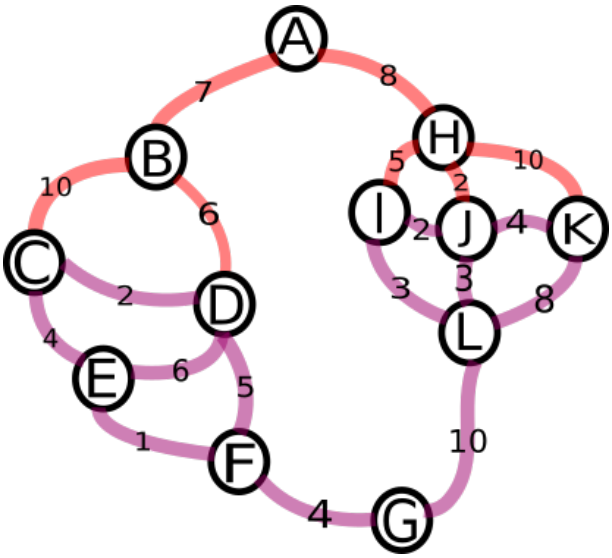
Nous pouvons désormais mettre à jour notre liste des priorités en retirant B, et en y rajoutant :

— C avec un coût de $7 + 10 = 17$
— D avec un coût de $7 + 6 = 13$

Notre liste des priorités sera ainsi composée dans l'ordre de H, puis de D, puis de C.

Il faudra donc visiter la ville H !

De H, nous pouvons visiter les villes I, J et K, voici encore une fois le graph des villes visitées :



Nous rajoutons ainsi à notre liste des priorités ces villes avec les coûts associés :

— I avec un coût de $8 + 5 = 13$
— J avec un coût de $8 + 2 = 10$
— K avec un coût de $8 + 10 = 18$

Il faudra donc visiter en priorité J !

Il faut ainsi répéter les étapes de cet algorithme, jusqu'à obtenir la solution finale : le chemin optimal !

Le lecteur pourra s'essayer à trouver le chemin optimal entre A et G, en appliqant scrupuleusement l'algorithme de Dijkstra, le graph montrant le chemin optimal sera présenté à la page suivante. Toutefois, on remarque vite un problème avec cet algorithme : toutes les villes ont du être visitées afin de pouvoir résoudre correctement notre problème !

Cela ne peut pas sembler être bien grave sur un si petit exemple, mais, quand il s'agit d'effectuer cet algorithme sur toutes les villes belges, il devient clair que nous avons une perte en performances.

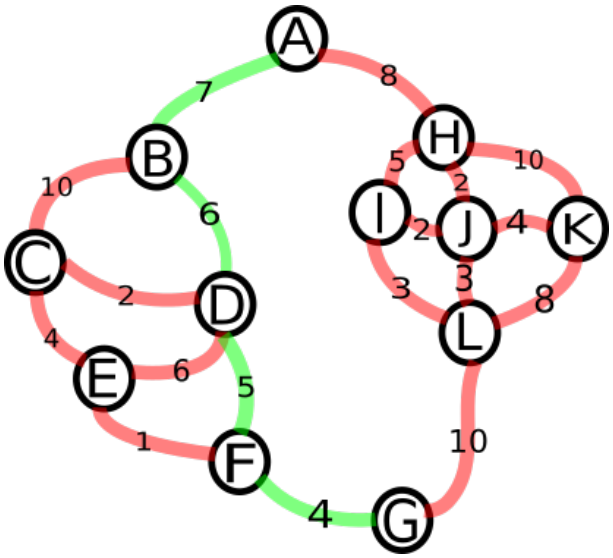


FIGURE 4 – Le chemin optimal !

Il va donc falloir ruser, et utiliser une heuristique afin d'avoir de meilleurs temps, et espérer pouvoir appliquer notre algorithme raisonnablement sur l'ensemble des villes belges.

Ces methodes d'heuristique seront justement discutées dans le prochain chapître à leur sujet.

4.4.3 Le besoin d'heuristiques.

Dijkstra a l'avantage d'être un algorithme donnant toujours la solution optimale, ainsi, peu importe le graphe dans lequel nous appliquons notre algorithme de Dijkstra, le chemin trouvé entre deux points de notre graph sera toujours le plus optimal.

En contrepartie, cet algorithme a pour défaut de bien souvent visiter un nombre assez important de noeuds sur notre graphe, ceci ayant pour conséquence de baisser la performance de notre algorithme.

Fort heureusement, des mathématiciens ont trouvé une solution afin de remédier à ce problème : **Créer une heuristique !**

Une heuristique est un outil nous permettant de prendre des décisions, de faire des choix. Une heuristique intéressante dans le cadre de notre problème avec un algorithme de Dijkstra, serait de privilégier uniquement certains noeuds nous rapprochant de la solution finale.

Il faut toutefois faire attention dans le choix de notre heuristique, une mauvaise heuristique pourrait nous faire retirer trop de noeuds d'une solution potentielle, et ainsi, produire des solutions sous-optimales.

4.4.4 Dijkstra avec de l'heuristique : A* !

L'algorithme A* est un algorithme de Dijkstra avec une heuristique intéressante.

Ainsi, pour chaque noeud de notre graphe, nous allons-y rajouter un "coût" heuristique. Celle qui sera choisie ici sera la distance euclidienne entre le noeud du graphe et l'objectif final.
Afin d'illustrer nos propos, nous avons ci-dessous implémenté l'algorithme A*.

```
(require :cl-csv)
(require :serapeum)
```

Tout d'abord, les deux librairies que nous utiliserons ici seront "cl-csv", similaire à pandas de python, cette librairie permet de traîter les fichiers csv (comma-separated values).

La seconde est serapeum, cette bibliothèque, au même titre que alexandria, permet d'étendre Common Lisp, afin d'en faire un outil bien pratique, nous nous servons de serapeum avant tout pour pouvoir utiliser les tas que cette librairie nous offre.

```
(defparameter *belgian-cities*
  (make-array 2787 :initial-contents
              (mapcar
                 (lambda (lat)
                   (let ((city-name
                            (second lat))
                          (city-longitude
                            (read-from-string (third lat)))
                          (city-latitude
                            (read-from-string (fourth lat))))
                     (make-city :name city-name
                                :longitude city-longitude
                                :latitude city-latitude
                                :cost -1
                                :heuristic 0
                                :prob-cost 0)))
                 (cdr (cl-csv:read-csv
                       #P"belgian-cities-geocoded/
   belgian-cities-geocoded.csv"))))
  "Récupère les éléments du CSV et les entrepose dans ce
   grand tableau.")
```

Ici, nous reprenons simplement les données du CSV de belgian-cities-geocoded et les entrepose dans un très grand tableaun, ici *belgian-cities*. Le lien du CSV est donné dans le doc-string accompagnant le paramètre global.

```
(defstruct city
  name longitude latitude cost heuristic prob-cost)
```

defstruct permet de créer des structures Common Lisp, il est possible de voir celles-ci comme les namedtuples de la bibliothèque collections de python.

```
(defun deg->rad (deg)
  (* deg (/ pi 180)))

(defun get-distance (city1 city2)
  "Nous donne la formule entre deux points à partir de
   leurs longitudes, via la formule de Haversine, cf.
https://stackoverflow.com/questions/27928/calculate
-distance-between-two-latitude-longitude-points
-haversine-formula"
```

```
    (let* ((earth-radius 6371)
           (diff-lat
             (deg->rad (- (city-latitude city1)
                          (city-latitude city2))))
           (diff-lon
             (deg->rad (- (city-longitude city1)
                          (city-longitude city2))))
           (a (+ (* (sin (/ diff-lat 2))
                    (sin (/ diff-lat 2)))
                 (* (cos (deg->rad
                          (city-latitude city1)))
                    (cos (deg->rad
                          (city-latitude city2)))
                    (sin (/ diff-lon 2))
                    (sin (/ diff-lon 2))))))
      (* earth-radius
         (* 2 (atan (sqrt a)
                    (sqrt (- 1 a)))))))
```

Ces deux fonctions permettent de calculer la distance entre deux villes en fonction de leur latitude et de leur longitude. La formule permettant de faire cela s'appelle "formule de Haversine" [9].

```
(defun neighbors (city)
  "Remet toutes les villes se trouvant
à proximité de notre ville d'origine."
  (remove-if
   (lambda (other-city)
     (or (< 20 (get-distance other-city city))
         (equal other-city city)))
     *belgian-cities*))
```

Cette fonction prend une ville en argument, et retourne chacune des villes se trouvant à un rayon de 20km de celle-ci.

On peut remarquer l'utilisation d'un **remove-if**, cette fonction d'ordre supérieur filtre tous les éléments d'une structure de données ne respectant pas un certain prédicat.

9. https ://fr.wikipedia.org/wiki/Formule_de_haversine

```
(defun heuristic-comp (city)
  "La fonction heuristique en elle-même,
celle-ci nous permet d'affecter une
priorité aux éléments du tas."
  (- 0 (+ (city-heuristic city)
          (city-cost city))))
```

Comme expliqué dans le docstring, cette fonction a pour objectif d'affecter une priorité à chaque élément du tas, en fonction du coût heuristique et du coût réel.

Il est intéressant de noter que pour réaliser un algorithme wA*, la seule modification à faire au programme est de modifier **heuristic-comp** en

```
(defun weighted-heuristic (city weight)
   (- 0 (+ (* weight (city-heuristic city))
           (city-cost city))))
```

Passons aux trois prochaines fonctions :

```
(defmacro get-max (heap)
  `(serapeum:heap-extract-maximum ,heap))

(defun make-path (curr path)
  "Reconstruis le chemin à partir des noeuds visités."
  (reverse (cons curr path)))

;; Macro anaphorique, applique la fonction function
;; sur le set, entrepose chacun de ces résultats
;; dans "it".
(defmacro for-it-over (function set &body body)
  `(loop for it across (map 'vector
                            #',function
                            ,set)
         ,@body))
```

Celles-ci n'ont malheureusement pas grand chose de très intéressant, elles ont été crées afin de limiter la taille du programme, et de rendre le code propre,

Il est toutefois intéressant de s'intéresser à for-it-over, qui est une macro anaphorique, cela veut dire que celle-ci se permet de garder le résultat des

calculs effectués à l'intérieur dans une variable, passant celle-ci au reste du programme.

Les macros anaphoriques sont toutefois à doubles-tranchants, bien qu'elles puissent faire gagner un programme en élégance, elles ont le don de faire en sorte que le lecteur s'y perde, et ne comprenne pas d'où proviennent les variables "it" se promenant librement dans le programme sans avoir été déclarées explicitement.

```
(defun shortest-a*-path (start goal)
  "Implémentation de l'algorithme A* en Common Lisp."
  ;; Création de la file de priorité,
  ;; et de la liste des noeuds visités.
  (let ((closed-list nil)
        (priority-queue (serapeum:make-heap
                         :key #'heuristic-comp)))
    ;; Le départ est bien souvent
    ;; le premier noeud à devoir être visité.
    (serapeum:heap-insert priority-queue start)
    (loop when (serapeum:heap-maximum priority-queue)
            do (let
                 ((new-city (get-max priority-queue)))
               (if (equal new-city goal)
                   (return (make-path new-city closed-list)
   ))
                   (for-it-over (lambda (c)
                                  (setf (city-heuristic c)
                                        (get-distance c
                                        goal))
                                  (setf (city-prob-cost c)
                                        (+ (get-distance c
                                            new-city)
                                           (city-cost
                                            new-city)))
                                  c)
                                (neighbors new-city)
                       ;; !! Les "it" s'expliquent
                       ;; par l'utilisation de la
                       ;; macro anaphorique
```

```
                        ;; for-it-over.
                        when (and
                               (not (member it closed-list))
                               (or  (eql -1 (city-cost it))
                                    (<  (city-prob-cost it)
                                        (city-cost it))))
                        do (progn
                             (setf (city-cost it)
                                   (city-prob-cost it))
                             (serapeum:heap-insert
                                 priority-queue it))
                finally (setf closed-list
                              (cons new-city closed-list))
    )))))
```

Et enfin, voici l'algorithme A* en lui même !

Remarquons l'utilisation assez forte de "loop", donnant un style assez impératif à cette implémentation.

Il s'agti d'ailleurs d'un avantage de Lisp, le langage est multiparadigme, et permet de coder de la manière à laquelle l'utilisateur le souhaite !

4.5 Des programmes et des jeux !

Dès les débuts de l'intelligence artificielle, une envie de battre des humains avec des ordinateurs à des jeux de société s'est fait sentir.

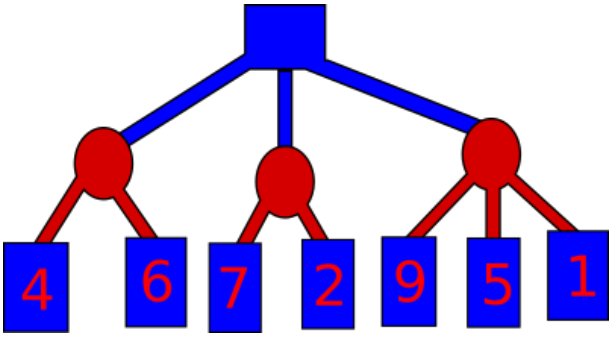
Un exemple historique de cela est la défaite de Kasparov face à Deep Blue d'IBM.

Le but de ce chapître est de présenter comment un ordi peu gagner contre un humain à un jeu de société, et quelles stratégies doivent être appliquées.

Nous implémenterons une IA capable de battre un humain à puissance 4.

4.5.1 L'agorithme Minimax.

À la base de cette section se trouve l'algorithme minimax. Pour comprendre celui-ci, nous allons jouer à un jeu !
Prenons comme plateau l'arbre suivant :



Le but du jeu est simple :

À chaque case bleue sur laquelle nous nous trouvons, vous avez le droit de déplacer le joueur où vous le voulez, dès que nous sommes sur une case rouge, c'est à mon tour de déplacer le joueur !

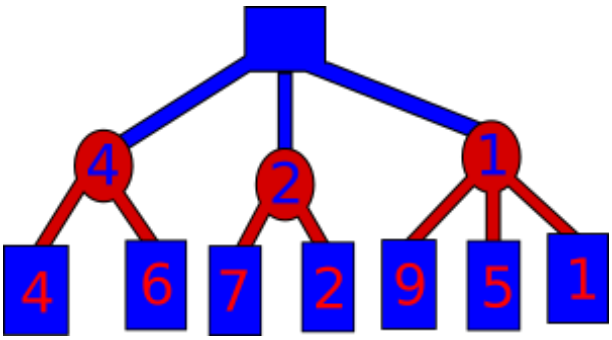
Votre but est d'atterrir sur la case la plus grosse possible, le mien est de vous faire atterrir sur la case détenant la plus petite valeur possible.

Comme première stratégie, vous pourriez être alléché par le 9, et vous déplacer directement vers la case rouge permettant d'y accéder.

Le défaut de cette stratégie est que si vous désiriez appliquer cette stratégie, alors, je déplacerai simplement le joueur vers le 1, et vous vous retrouveriez avec le pire résultat possible.
Il va donc falloir innover.

La stratégie du minimax consiste à dire que je jouerai à chaque fois le coup le plus minimisant, et ainsi, il est possible de remplacer chacun de mes choix par la valeur de la case la plus petite qui me serait accessible depuis la case où nous nous trouvons.

Ainsi, nous pourrions représenter notre plateau ainsi :



De cela, il devient très clair que le meilleur score que vous puissiez espérer avoir est 4.

Ainsi, c'est par une alternance de cycle où l'on minimise les scores, et de cycles où on les maximise que se déroule le minimax (en effet, si je devais prédire votre coup, je dirais que vous iriez toujours vers là où la case est la plus haute.)

Le défaut du minimax est quand le nombre de positions dans la partie devient énorme. C'est le cas du puissance 4, ou encore du jeu d'échecs, il faut donc pouvoir construire des heuristiques ou alors poussez l'optimisation jusqu'au bout.

Le reste de ce chapître présente quelques méthodes d'optimisations de l'algorithme minimax.

4.6 Un programme de jeu.

Nous voici enfin au cœur du sujet, avec l'implémentation de notre algorithme Minimax en Common Lisp. Le code se sépare en trois parties :

a. La partie Logique de Jeu

b. La partie Minimax

c. La partie interface utilisateur

sans plus tergiverser, analysons de suite le code.

```
(defparameter *board*
  (make-array 7 :initial-element
              (make-list 6 :initial-element nil))
  "Le plateau de puissance 4 en lui-même.")

(defparameter *moves* nil
  "Reprend la liste des coups ayant été jouée.")
```

Ici, nous utilisons deux variables globales, toujours en respectant la convention des cache-oreilles. Il est intéressant de noter qu'il est en général conseillé de limiter le nombre de variables globales se promenant dans un code au strict minimum, en effet, celles-ci empêchent une lecture linéaire, et peuvent mener à ce qui est connu sous le nom de "code spagetthi". Dans notre code de puissance 4, nous n'en utiliserons au total trois.

```
;;; LOGIQUE DE JEU
(defun make-new-board ()
  "Permet de créer un plateau de jeu tout neuf."
  (setf *board* (make-array 7 :initial-element
                            (make-list 6 :initial-element
   nil)))
  (setf *moves* nil))
```

Avec cette fonction permettant de construire un nouveau tableau, nous entamons notre première partie concernant la logique de jeu.

Il est intéressant ici de noter la structure de donnée représentant le tableau de jeu. Un tableau comprenant des listes simplement chaînées. Nous avons choisi une structure de donnée si originale simplement car nous aimions

la manière à laquelle celle-ci se laissait représenter en mémoire.

```
(defmacro count-to (x)
  "Une macro similaire à iota de la bibliothèque alexandria
   ."
  (let ((i (gensym)))
    `(loop for ,i from 1 to ,x
           collect ,i)))

(defun playerify (board-case)
  "Permet d'afficher propreememnt chaque case."
  (case board-case
    (yellow 'o)
    (red    'x)
    (t      '+)))

(defun show-board (board)
  "Permet d'afficher le plateau."
  (let ((row nil))
       (loop for i from 1 to 6
           do (progn
                   (setf row (map 'list #'car board))
                   (format t "~{~A ~}|~D~%"
                           (mapcar #'playerify row) i)
                   (setf board (map 'vector #'cdr board)))
           finally (format t "~{~D ~}~%"
                           (count-to 7)))))
```

count-to et **playerify** servent juste à alléger la syntaxe à certains endroits. Nous faisons remarquer avant tout l'utilisation de **gensym** dans notre macro, **gensym** a pour rôle de créer un symbole au nom unique, ainsi, les variables étant définies dans la macro ne peuvent pas être appelées en dehors de la macro (**gensym** existe également en prolog). **case** est l'équivalent de **switch** en C.

show-board nous permet d'afficher notre structure de donnée à l'écran de manière convenable.

```
(defun fullp (i board)
```

```
  "Vérifie si une colonne est vide."
  (car (aref board i)))

(defun get-legal (board)
  (loop for i from 0 to 6
        if (not (fullp i board))
          collect i))

(defun tiedp (board)
  "Vérifie si la position est nulle."
  (notany #'null
          (map 'list #'car board)))
```

Voici trois fonctions d'aide, celles-ci ont pour but d'alléger grandement le corps des fonctions plus grandes, se situant plus loin dans le code. Nous recommandons grandement la lecture de **The Little Schemer** afin de comprendre pourquoi de telles fonctions sont très importantes.

```
(defun play (i color board &optional (mark nil))
  "Fonction permettant de jouer un jeton sur le plateau."
  (when mark
    (setf *moves* (append *moves* (list i))))
  (let ((c-board (copy-seq board)))
    (labels ((push-until (lat token)
               (cond
               ((eq 1 (length lat)) `(,token))
               ((cadr lat) (cons token (cdr lat)))
               (t (cons (car lat)
                        (push-until (cdr lat) token))))))
        (setf (aref c-board i) (push-until (aref c-board i)
    color))
    c-board)))
```

Notons ici l'utilisation de **&optional**, permettant d'utiliser des paramètres optionnels. Et **labels** permettant de définir des fonctions à l'intérieur d'autres fonctions.

```
(defun maximum (lat)
  "Donne le nombre maximum dans une liste simplement chaîné
   e."
  (reduce (lambda (a b)
```

```
            (if (> a b)
                a
                b))
          lat))

(defun max-subseq (color seq)
  "Donne le nombre maximal d'élément se ressemblant dans
   une liste.
e.g => (max-subseq 'red '(red red yellow red red red yellow
    yellow)) == 2"
  (let ((maxi 0)
        (temp 0))
    (loop for token in seq
          if (eq color token)
            do (progn
                 (incf temp)
                 (setf maxi (max temp maxi)))
          else
            do (setf temp 0))
    maxi))
```

Encore deux fonctions d'aides, celles-ci seront très pratiques pour vérifier le nombre de connections entre les jetons sur notre plateau de puissance 4.

```
(defun vertical-connections (color board)
  "Donne le nombre maximal de connection verticale."
  (labels ((number-in-vertical (column colour &optional (
   buffer 0))
             (cond
               ((null column) buffer)
               ((not (car column)) (number-in-vertical (cdr
    column) colour 0))
               ((eq (car column) colour)
                (number-in-vertical (cdr column) colour (1+
    buffer)))
               (t buffer))))
    (maximum (map 'list (lambda (column)
                          (number-in-vertical column color)
   )
                  board))))
```

```
(defun horizontal-connections (color board)
  "Donne le nombre maximal de connections horizontales."
  (labels ((row-finder (the-board)
             (loop for row from 0 to 5
                   collect (map 'list #'car the-board) into
    rows
                   do (setf the-board (map 'vector #'cdr
    the-board))
                   finally (return rows))))
    (maximum (mapcar (lambda (row)
                       (max-subseq color row))
                     (row-finder board)))))
```

Ici, nous définissons enfin des fonctions permettant de vérifier le nombre de connections verticales et horizontales sur notre plateau. Attention au faux-amis ! **return** en Common Lisp n'est utile que dans les boucles, pour retourner une valeur à partir d'une fonction, il n'est nécessaire de préciser return que lorsque nous désirons faire sortir une valeur d'une boucle.

Le code pour vérifier les connections horizontales et verticales reste relativement simple. Intéressons-nous maintenant au code permettant de vérifier les connections en diagonale.

```
(defun make-left-diagonal (the-case
                           &optional (max-row 5))
  "Donne les diagonales allant d'en haut à gauche,
jusqu'à en bas à droite."
  (let ((column (car the-case))
        (row    (cdr the-case)))
    (loop for c from column downto 0
          for r from row to max-row
          collect (cons c r))))

(defun make-right-diagonal (the-case
                            &optional
                              (max-row 5)
                              (max-column 6))
  "Donne les diagonales allant d'en bas à gauche,
```

```
jusqu'en bas à droite."
  (let ((column (car the-case))
        (row    (cdr the-case)))
    (loop for c from column to max-column
          for r from row to max-row
          collect (cons c r))))

(defconstant top-right-corner '((3 . 0) (4 . 0) (5 . 0)
                                (6 . 0) (6 . 1) (6 . 2))
  "Coordonnées des points se trouvant en haut à droite.")

(defconstant top-left-corner '((3 . 0) (2 . 0) (1 . 0)
                               (0 . 0) (0 . 1) (0 . 2))
  "Coordonnées des points se trouvant en haut à gauche.")

;; Cette macro permet d'écrire les fonctions
;; vérifiant le nombre de connections diagonales
;; de manière bien plus simple et concise.
(defmacro max-in-some-diagonal
    (function-name color board helping-function
   starting-corner)
  `(defun ,function-name (,color ,board)
     (let* ((diagonals (mapcar #',helping-function ,
   starting-corner))
            (diag-values (mapcar
                          (lambda (diags)
                            (mapcar
                             (lambda (coords)
                               (let ((column (car coords))
                                     (row (cdr coords)))
                                 (nth row (aref ,board
   column))))
                             diags))
                          diagonals)))
       (maximum (mapcar
                 (lambda (diagonal)
                         (max-subseq ,color diagonal))
                 diag-values)))))
```

```
;;; Définition des fonctions
;;; left-diagonal-connections et
;;; right-diagonal-connections.
(max-in-some-diagonal left-diagonal-connections
                      color board  make-left-diagonal
   top-right-corner)
(max-in-some-diagonal right-diagonal-connections
                      color board make-right-diagonal
   top-left-corner)
```

Ici, remarquons la définition de constantes. Les constantes sont à différencier des variables globales telles que *moves* de par le fait qu'elles sont parfaitement immutables, et ainsi, plus facilement optimisables par le compilateur et laissent au code sa lisibilité.

L'utilisation d'une macro dans le cas de **max-in-some-diagonal** aurait put être remplacé par une simple fonction d'aide. Nous aimions juste la liberté supplémentaire au niveau de la syntaxe dans la définition des fonctions du dessous que nous offrait la macro.

```
(defun connections (color board)
  "Donne le nombre maximal de connections sur le plateau."
  (max
   (vertical-connections       color board)
   (horizontal-connections     color board)
   (left-diagonal-connections  color board)
   (right-diagonal-connections color board)))

(defun winningp (board)
  "Si le nombre maximal de connection sur le plateau est
   supérieur
à trois, un joueur a gagné."
  (or (> (connections 'yellow board) 3)
      (> (connections 'red    board) 3)))
```

Voici enfin les deux dernières fonctions composant notre partie de la logique de jeu. **connections** est encore une fois une fonction d'aide, il est intéressant de noter ici l'utilisation de **max** comme fonction polyvariadique.

```
;;; PARTIE MINIMAX
```

```
(defun evaluation (board)
  "Fonction permettant d'attribuer une valeur à chaque
   position."
  (cond
      ((tiedp board) 0)
      (t (flet ((eval-num (a) (case a (0 0) (1 4) (2 550) (
   3 3500) (t 50000000))))
           (let ((vert (vertical-connections       'red
   board))
                 (hori (horizontal-connections     'red
   board))
                 (l-di (left-diagonal-connections  'red
   board))
                 (r-di (right-diagonal-connections 'red
   board))
                 (vero (vertical-connections       'yellow
    board))
                 (horo (horizontal-connections     'yellow
    board))
                 (l-do (left-diagonal-connections  'yellow
    board))
                 (r-do (right-diagonal-connections 'yellow
    board)))
           (- (reduce #'+ (mapcar #'eval-num (list vert
   hori l-di r-di)))
              (reduce #'+ (mapcar #'eval-num (list vero
   horo l-do r-do)))))))))
```

Notre partie Minimax commence enfin avec la fameuse fonction d'évaluation. Celle-ci marche avec la formule suivante :

$$eval = 1R*1+2R*550+3R*3500+4R*50000000-1Y*1-2Y*550-3Y*3500-4Y*50000000$$

Celle-ci est bien évidemment perfectible, (lors des scéances de tests, celle-ci apporta à notre algorithme un comportement fort sadique).

```
(declaim (inline max-to-color))
(defun max-to-color (n)
  (if (eql n 1)
      'red
```

```
      'yellow))

(defun get-max-min-seq (lst key)
  "Donne la séquence de coups ayant la meilleure évaluation
   , selon la couleur."
  (let ((max-seq
         (reduce (lambda (a b)
           (if (funcall key (cdr a) (cdr b))
               a
               b))
           lst)))
         (nbutlast (car max-seq))
         max-seq))

(defun group-by (list key)
  "Groupe les éléments d'une liste selon un certain pré
   dicat/fonction."
  (declare (optimize (speed 3) (safety 0) (debug 0)))
  (declare (type cons list) (type function key))
  (labels ((grouper (lat the-key acc)
            (if (null lat)
                acc
                (let ((test-val (funcall the-key (car lat)))
   )
                  (grouper (remove-if (lambda (a)
                                        (equal (funcall
   the-key a) test-val))
                                      lat)
                           the-key
                           (cons (remove-if-not (lambda (a)
                                                  (equal (
   funcall the-key a) test-val))
                                                lat)
                                 acc))))))
    (grouper list key nil)))

(defun max-len-filtr (big-list &key (fn 'identity))
  "Retourne toutes les listes de tailles maximales,
ainsi que la longueur de ces listes, et une liste contenant
```

```
    les listes de tailles inférieures."
  (let ((max-len    0)
        (acc      nil)
        (rejected nil))
    (loop for list in big-list
          do (cond ((> (length (funcall fn list)) max-len)
                    (setf max-len (length (funcall fn list)
 ))
                    (loop for list in acc
                          do (push list rejected))
                    (setf acc (make-list 1 :initial-element
 list)))
                   ((= (length (funcall fn list)) max-len)
                    (push list acc))
                   (t (push list rejected))))
    (values acc max-len rejected)))
```

Toutes les fonctions suivantes ne sont que des fonctions d'aide, nous n'allons pas tergiverser énormément dessus. La chose importante à noter toutefois est l'utilisation de **(declaim (inline f))** permettant de définir une fonction comme étant inline (il s'agit simplement d'une technique d'optimisation, les fonctions inlines étant en dehors de la portée de ce projet, nous encourageons le lecteur à se renseigner sur celles-ci de lui-même).

```
(defun make-moves (moves)
  "Permet de trouver une position uniquement en fonction
   des coups joués."
  (let ((first-player 'red)
        (board (make-array 7 :initial-element
                           (make-list 6 :initial-element
 nil))))
    (flet ((get-opp (color)
             (case color
               (red 'yellow)
               (t   'red))))
      (loop for move in moves
            do (setf board (play move first-player board))
               (setf first-player (get-opp first-player)))
      board)))
```

```
(defun eval-sequence (moves)
  "Permet d'évaluer une position uniquement en fonction des
    coups joués."
  (evaluation (make-moves moves)))
```

Ces fonctions permettent de créer un tableau de puissance 4, uniquement en fonction d'une suite de coups. **flet** a la même fonction que **labels**.

```
(defun parity (num)
  "Fonction retournant -1 si le nombre est pair, 1 sinon."
  (if (evenp num) -1 1))

(defun flatten (nested)
  "Permet d'aplatir une liste."
  (reduce #'nconc nested))
```

Deux fonctions d'aide, leur intérêt est assez explicite grâce aux doc-strings les accompagnants.

```
(defun minmax (move-seq depth)
  "Retourne toutes les suites de coups possibles, ainsi que
    l'évaluation leur étant attribuée."
  (let ((board (make-moves move-seq)))
    (cond
      ((tiedp board)
       (push (cons move-seq 0) *all-possible-moves*))
      ((winningp board)
       (push (cons move-seq (* +inf (parity (length
   move-seq))))
             *all-possible-moves*))
      ((= depth 0)
       (push (cons move-seq (eval-sequence move-seq))
             *all-possible-moves*))
      (t (loop for move in (get-legal board)
               do (minmax (append move-seq (list move)) (1-
    depth))))))
  *all-possible-moves*)

(defun best-play (depth moves)
  "Retourne le meilleur coup possible en fonction de la
   profondeur."
```

```
(minmax moves depth)
(mapc (lambda (lst)
        (setf (car lst)
              (nthcdr (length moves)
          (car lst))))
  *all-possible-moves*)
(loop (multiple-value-bind (valuable max-len rejected)
          (max-len-filtr *all-possible-moves* :fn #'car)
        (when (= max-len 0)
          (format t "An error Occured, caught!") (return)
 )
        (when (= max-len 1) (return))
        (setf *all-possible-moves*
           (mapcar (lambda (a)
                     (get-max-min-seq a
                         (lambda (c d) (if (evenp max-len
 ) (< c d) (> c d)))))
                   (group-by valuable (lambda (a) (butlast
  (car a))))))
        (setf *all-possible-moves* (append rejected *
 all-possible-moves*))))
    (caar (reduce (lambda (a b) (if (> (cdr a) (cdr b))
 a b))
          *all-possible-moves*)))
```

Voici enfin venir la fin de notre partie minimax.

Nous nous intéresserons ici à l'intérêt de **multiple-value-bind**, en effet, en Lisp, il est possible de retourner plusieurs valeurs d'une fonction grâce à l'utilisation de **values**. Il faut tout de même un moyen de récupérer toutes les valeurs sorties de la fonctions, et savoir relier ses valeurs à des variables. C'est ici qu'entre en jeu **multiple-value-bind**, dont l'intérêt est justement de récupérer les valeurs sorties par **values** et les rattacher à des variables locales.

```
;;; PARTIE INTERFACE UTILISATEUR
(defmacro player-repl (board player-turn context-name)
  (let ((other-player (if (eql player-turn 2) 1 2))
        (color (if (eql player-turn 1) 'yellow 'red)))
```

```
    `(progn
        (when (winningp ,board)
          (return (format nil "Le joueur ~D a gagné." ,
    other-player)))
        (format t "Au tour du joueur ~D!~%~A~%Dans quelle
    colonne souhaitez vous jouer? "
                ,player-turn (show-board ,board))
        (let* ((column (read))
               (i (1- column)))
          (cond
            ((not (<= 1 column 7))
                  (format t "Veuillez jouer un numéro de
    colonne correct!~%")
                  (,context-name))
             ((fullp i ,board)
              (format t "Cette colonne est déjà remplie!~%"
    )
              (,context-name))
             (t (setf ,board (play i ',color ,board t))))))
    ))

(defun play-against-player ()
  "L'interface utilisateur, permettant de jouer contre un
   autre joueur."
  (loop while (not (or (winningp *board*)
                       (tiedp *board*)))
        do (player-repl *board* 1 play-against-player)
        do (player-repl *board* 2 play-against-player)))

(defun play-against-computer ()
  (loop while (not (or (winningp *board*)
                       (tiedp *board*)))
        do (setf *board* (play (best-play 4 *moves*) 'red *
    board* t))
           (setf *all-possible-moves* nil)
           (player-repl *board* 1 play-against-computer)
        when (winningp *board*)
          do (return "Vous avez gagné!")))
```

Voici en fin venir la fin du programme Lisp, avec la partie interface utili-

sateur, le programme permet de jouer contre un autre joueur, ou encore de jouer contre l'IA.

4.6.1 $\alpha\beta$-élagage.

L'alpha-bêta élagage est une technique d'optimisation inventée par nul autre que John McCarthy (l'inventeur du Lisp, en personne !).

Nous ne rentrerons pas dans les détails du fonctionnement de l'$\alpha\beta$-élagage. En clair, celui-ci permet de réduire grandement le nombre de recherches à faire dans l'arbre de décision sur lequel nous appliquons notre algorithme minimax.

En effet, il n'est pas nécessaire de visiter les positions où l'évaluation n'est pas assez favorable.

L'arbre de jeu étant tellement énorme, il est extrêmement pratique de ne pas avoir à tout visiter.

4.6.2 Du dynamisme bon sang !

Le principe de la programmation fondateur dynamique est assez simple : Il est plus facile de mémoriser que de calculer.

La programmation dynamique se base sur le principe du Divide and Conquer (ou "diviser pour mieux régner" en français). Il faut tout d'abord répartir le problème en plusieurs petites tâches plus simples à résoudre, par la suite, nous stockons les résultats intermédiaires, et grâce à ces résultats stockés, nous pouvons arriver à une solution finale plus rapidement que si nous devions à chaque fois tout recalculer à partir de 0.

Nous utiliserons une stratégie s'appelant la **mémoization** afin d'optimiser notre algorithme du Minimax.

La **mémoization** est une stratégie typique en programmation fonctionnelle.

Le principe est simple : Chaque fois que nous appelons une fonction, nous stockons le résultat associé à la valeur entrée dans une table de hachage, dès que nous rappelons la fonction, nous vérifions si le paramètre de celle-ci se trouve dans notre table de hachage.
Si c'est le cas, nous retournons directement la valeur associée à la clé, n'oublions pas que cette opération est de complexité O(1), ce qui explique l'intérêt de stocker le résultat plutôt que d'avoir à le recalculer entièrement.

Nous utilisons cette stratégie dans notre puissance 4 avant tout car il est possible d'atteindre une même position, à partir de suites de coups différentes, il est donc intéressant de ne pas avoir à réévaluer la position inutilement.

5 Le Machine Learning.

5.1 Définition du Machine Learning

Le Machine Learning, ou Apprentissage Automatique, est un type d'intelligence artificielle qui analyse et s'entraine avec des données, permettant dès lors aux ordinateurs d'apprendre par expérience (sans avoir été explicitement programmé à cet effet ou par intervention humaine). Cela consiste en algorithmes d'apprentissage qui améliorent leur performance à exécuter des tâches au fil du temps grâce à de l'expérience.

Le machine learning est présent dans la vie de tous les jours. Par exemple, nos boîtes mail sont équipées d'algorithmes capables d'apprendre avec le temps. Prenons les spams, leurs émetteurs sont capables de s'adapter, de changer leur manière d'agir, il est donc impératif que l'algorithme de défense contre les spams change lui aussi. Dans les e-mails, le machine learning se cache aussi derrière les réponses automatiques, en 2018, 6,7 milliards des mails envoyés étaient des réponses automatiques. On peut aussi trouver des IA dans les réseaux sociaux (reconnaissance faciale entre amis Facebook,...), blocage automatique des cartes de crédit volées, et bien d'autres.

5.2 Les Maths dans le Machine Leanring.

De nombreux data scientists (chargés de la gestion, de l'analyse et de l'exploitation des données au sein d'une entreprise) considèrent le machine learning comme un apprentissage statistique car celui-ci s'appuie sur des algorithmes (principalement statistique) permettant à une machine d'apprendre grâce à des échantillons ou bases d'apprentissages.

Dans le machine learning, il y a en grande partie des maths qui permettent à la machine de fonctionner, on retrouve surtout des algorithmes

et des statistiques. D'ailleurs, de nombreux Data scientists considèrent que le machine learning n'est qu'un apprentissage statistique. De plus, ce qu'on appelle machine learning est pour certains la rencontre des statistiques avec la puissance de calcul disponible aujourd'hui (mémoire, processeurs, cartes graphiques).

On retrouve aussi les fonctions, les variables, les équations, les graphiques, et d'autres outils mathématiques de ce genre dans le machine learning. Pour les fonctions, elles sont utiles à l'analyse des séries temporelles (suite de valeurs numériques représentants l'évolution d'une quantité spécifique au cours du temps).

5.3 Les résaux de neurones

Une couche de neurones d'entrées, plusieurs couches cachées, une couche de sortie suivie d'une fonction d'activation. Chaque neurone possède une valeur obtenue par une fonction de combinaison étant la somme des valeurs des neurones de la couche précédente, chacune multipliée par un poids spécifique $z = x_1 \cdot p_1 + x_2 \cdot p_2 + \cdots + x_n \cdot p_n$

Une fois la (ou les) valeur(s) de la couche de sortie obtenue, on applique à celle-ci une fonction d'activation qui transforme la valeur en fonction d'un seuil. Si en dessous du seuil, inactif (0/-1), aux environs du seuil, phase de transition, et au-dessus du seuil, actif (1/>1). Le type de fonction varie d'un cas à l'autre, mais les plus récurrentes sont la fonction sigmoïde $\frac{1}{1+e^{-x}}$ la fonction tangente hyperbolique $\frac{2}{1+e^{-2x}} - 1$ ou encore la fonction ReLU $\begin{cases} 0 & x < 0 \\ x & x >= 0 \end{cases}$

Durant l'apprentissage, les poids sont des valeurs prises au hasard. Il faut donc les ajuster pour fournir une réponse qui se rapproche au mieux de la

réalité. Comme on entraîne notre réseau, on connaît la vraie valeur finale. On va donc appliquer une fonction de coût afin de calculer le gradient d'erreur entre la valeur réelle et la valeur prédite $\frac{1}{2}(y_r - y_p)^2$, et ainsi mettre à jour les poids par rétropropagation (! il y a des maths plus compliquées derrière). A chaque nouvelle donnée injectée lors de l'apprentissage, le réseau est plus performant.

5.4 Intro au langage Python

Python est un langage de programmation créé par Guido van Rossum dont la première version est parue en 1991 mais est exposée au grand public au Stichting Mathematisch Centrum en 1994 (fondation du centre mathématique), en Hollande. Ce langage est dit open source, c'est-à-dire qu'il est gratuit et libre d'utilisation.

Ce langage est le successeur de quelques langages de programmation comme ABC, C, PERL et bien d'autres. Python est un langage à typage fort, cela veut dire qu'il n'y a pas de conversion automatique entre deux types distincts. Ainsi, il n'est pas possible en python d'écrire.

```
>>> 3 + '3'
```

sous peine de recevoir une erreur.

C'est aussi un langage dynamique, c'est-à-dire que les variables ne sont pas attachées à leur type, ainsi, il est possible d'écrire

```
>>> a = 3
>>> a = "foo"
```

Enfin le typage de python est dit implicite, il ne faut pas obligatoirement préciser le type de variable, bien qu'il soit possible de déclarer le type des variables depuis la version 3.5.

```
>>> a = "foo"
>>> a: str= "bar"
```

Sont ainsi deux manières tout à fait correctes de déclarer une variable en python.

Le nom du langage vient de la série Monty Python's flying Circus dont Guido Van Rossum était fan. Mais l'image du serpent paraissant plus évidente pour tout le monde, il a décidé d'utiliser celle-ci comme symbole du langage.

Le langage python a été mis à jour de nombreuses fois et est passé d'une version 1.5 en 1999 jusqu'à 3.9 en 2020, nous nous intéresserons surtout aux versions python 2 et python 3 qui ont été ls plus marquantes.

5.4.1 Python 2 :

Par exemple, des erreurs de syntaxe, des manques de ' :', des manques d'une lettre, et d'autres, toutes des erreurs d'analyse de code. Nous pouvons aussi trouver des exceptions qui, quand utilisées dans notre programme, nous permettent de contrer les erreurs pouvant être écrites par la suite comme les 'ZeroDivisionError' ou en français, erreur de division par zéro, mais aussi les 'ValueError' qui nous annonce une valeur incorrecte et encore bien d'autres.

Il existe des moyens de convertir du code de python 2 vers python 3 mais ce n'est pas très fiable et de toute façon, ce n'est plus trop utile non plus car de plus en plus de développeurs conseillent python 3 pour sa facilité d'utilisation, de plus le support de la version 2 a été abandonné en 2020.

5.4.2 Python 3 :

Comme expliqué ci-dessus, Python 3 est plus simple d'utilisation que Python 2, la syntaxe y est plus simple et plus facilement compréhensible. Prenons un exemple simple avec le print, pour Python 2, il n'y a pas l'utilisation de la parenthèse, juste des guillemets. Ainsi, quand en Python 2 il faut écrire :

```
>>> print "Hello World!"
Hello World!
```

EN PYTHON 3, IL FAUDRA ÉCRIRE CELA

```
>>> print ("Hello World!")
Hello World!
```

Le stockage est dit unicode, c'est-à-dire que l'échange de texte est fait dans différentes langues, la langue la plus utilisée est bien sûr l'anglais. Dans Python 3, la valeur des variables utilisées ne change jamais, si vous prenez x=3, la valeur reste x=3.

Certaines fonctions utilisées dans python 2 comme xrange() on été remplacées dans python 3 par range(), tous deux servent à créer une liste et aussi des itérations(répétitions). Certaines bibliothèques(ensemble logiciel de modules) de python 3 ne peuvent pas être créées avec Python 2.

a. **Python 2 :**

```
>>> print "Hello World"
Hello World!
>>> print 7/3
2
>>> a = "éèà"
>>> a
'\xc3\xa9\xc3\xa9\xc3\xa0'

>>> i = 4
>>> my_list = [i for i in xrange(5, 1, -1)]
>>> my_list
[5, 4, 3, 2]
>>> i
2
```

b. **Python 3 :**

```
>>> print ("Hello World!")
Hello World!
```

```
>>>print (7/3)
2.3333333333333335
>>> a = "ééà"
>>> a
'éèà'
>>> i=4
>>> my_list = [i for i in range(5,1,-1)]
>>> my_list
[5, 4, 3, 2]
>>> i
4
```

5.4.3 Un programme d'exemple avec le second degré !

```
from math import sqrt

# J'importe la fonction sqrt, permettant de calculer les
   racines carrées.
def second_degre(
    a, b, c
) -> float:  # Je défini une fonction qui me permettra en
   mettant uniquement les valeurs de a, b, c de résoudre
   une équation du second degré du type ax^2 + bx + c=0
    p = b ** 2 - 4 * a * c  # p est ici le déterminant de
   mon équation.
    if (
        p > 0
    ):  # If permet la création de conditions, si le dé
   terminant est supérieur à 0, nous aurons deux racines.
        x1 = (-b + sqrt(p)) / 2 * a
        x2 = (-b - sqrt(p)) / 2 * a
        return (
            f"les valeurs sont {x1} et {x2}"
        )  # j'affiche les valeurs du x1 et du x2 à l'écran
   .
    elif p == 0:  # Si P = 0, il n'y a qu'une seule
   solution.
        x = -b / (2 * a)
        return (
```

```
            f"la valeur est égale à {x}"
        )  # j'affiche la valeur de x grâce au return et à
    la fonction f string
     return "il n'y a pas de valeur de x"


a = int(input("introduire une valeur de a :"))
b = int(input("introduire une valeur de b :"))
c = int(input("introduire une valeur de c :"))
# La fonction input a pour but de demander une valeur à l'
    utilisateur du programme..
# La fonction int pour convertir la chaîne de caractères
    obtenues en nombre.
print(second_degre(a, b, c))
```

5.5 le Data Science

La Data science (sciences des données) est un domaine scientifique qui regroupe et relie toutes les disciplines relatives à l'utilisation de données. Son but à terme est de récolter et transformer des données brutes (Ex : Âge et classement des participants à une compétition) en informations concrètes et utilisables (Ex : Influence estimée de l'âge d'une personne sur ses résultats à cette compétition). Cette transformation commence par leur tri, pour garder uniquement ce qu'on peut utiliser, ensuite, on structure ces données selon certains modèles qui rendent leur exploitation possible, on commence donc à exploiter ces données à l'aide d'un logiciel ou d'un algorithme, et il ne reste plus qu'à rendre les informations obtenues communicables avec, par exemple, un graphique.

Comme mentionné ci-dessus, la data science utilise diffèrents algorithmes. Par exemple, la régression linéaire, qui sert à établir la relation entre une variable explicative, aussi appelée variable à expliquer, et une ou plusieurs variable expliquée (utilisée pour décrire les variables dépendantes). La régression linéaire permet par exemple de calculer des phénomènes économiques mais aussi d'autres genres de phénomènes. Par exemple, si l'on mesure la consommation d'une voiture en fonction de sa vitesse, la consommation est considérée comme la variable expliquée, tandis que la vitesse est considérée comme la variable explicative, à chaque valeur de la vitesse correspond une et une seule valeur de la consommation. Exactement comme dans une fonction où, à chaque valeur de x (qui pourrait ici, être la vitesse) correspond une seule valeur de Y (pouvant ici être la consommation).

L'intelligence artificielle est une des disciplines qui demande la plus grande connaissance en data science étant donné que les données en sont le carburant, ce dont toute intelligence a besoin pour se structurer. Cependant, la data science étant un domaine extrêmement large, elle est utile dans bien d'autres domaines que la conception d'IA.

5.6 La bibliothèque NumPy

5.6.1 NumPy : Qu'est-ce que c'est et pourquoi c'est génial ?

NumPy est une bibliothèque très importante de Python utilisée dans le machine learning.

Elle permet de créer et de manipuler des tableaux à une ou plusieurs dimensions. Elle rajoute aussi plusieurs méthodes afin d'utiliser des opérations entre les tableaux.

L'avantage du tableau dans NumPy est qu'il est beaucoup plus performant, prenant moins de place : un tableau à une dimension ressemblera par exemple à une liste Python mais prendra moins de place que celle-ci. Il est par conséquent plus rapide mais il est aussi beaucoup plus précis et pratique permettant d'éviter des taches inutiles grâce à la grande variété de méthode que la bibliothèque NumPy offre.

5.6.2 Création d'un tableau

Pour créer un tableau, plusieurs façons peuvent être utilisées.

Une façon assez classique est d'utiliser la fonction array avec laquelle, il est possible de créer un tableau. Si par exemple, on veut créer un tableau à une dimension contenant les chiffres 1, 2 et 3, il suffit d'utiliser la fonction array suivie d'un tuple ou d'une liste contenant ces chiffres.

```
>>> import numpy as np
>>> x = np.array([1, 2, 3])
>>> print(x)
[1 2 3]
```

Attention à ne pas oublier de mettre ces chiffres entre [ ] ou entre ( ) pour que le tableau se crée.

Si maintenant on veut rajouter une dimension contenant les chiffres 4, 5, 6, il suffit de faire la même chose en rajoutant des ( ) ou des [ ] pour délimiter

les éléments d'une même dimension et une virgule entre les ( ) ou [ ] pour les séparer.

```
>>> import numpy as np
>>> x = np.array([(1, 2, 3), (4, 5, 6)])
>>> print(x)
[[1 2 3]
 [4 5 6]]
```

Attention, ne toujours pas oublier de mettre tout les chiffres entre [ ] ou entre ( ) pour créer le tableau.

Cependant, il est possible que dans certains cas, les éléments dans le tableau ne soient pas connus ou qu'on ait besoin de créer un tableau avec x éléments mais sans forcément le remplir de chiffres précis.

Dans ce cas, certaines fonctions comme zeros ou ones permettent de créer un tableau rempli de 0 (ou de 1 avec la fonction ones) avec les dimensions et le nombre d'éléments souhaité.

La fonction suivie d'un tuple contenant le nombre d'éléments pour chaque dimension permet de créer le tableau (le shape du tableau). Le dernier chiffre du tuple représente le nombre d'éléments dans le premier axe (axe = dimension) ce qui équivaut au nombre d'éléments sur une ligne. Le chiffre précédent représente le nombre d'éléments dans le deuxième axe ce qui équivaut au nombre d'éléments dans un colonne. On continue ce résonnement jusqu'au premier chiffre qui représente, pour une dimension n, le nombre d'éléments dans l'axe n.

Par exemple, si le tableau a 3 dimensions, le tuple contient 3 chiffres avec comme premier chiffre le nombre d'éléments dans le troisième axe.

```
>>> import numpy as np
>>> x = np.zeros((2, 3))
>>> print(x)
[[ 0.  0.  0.]
 [ 0.  0.  0.]]
```

Dans ce code, le tableau voulu a 2 dimensions. Le chiffre 2 représente ici le nombre de 0 qu'il doit y avoir dans une colonne et le chiffre 3 représente le nombre de 0 qu'il doit y avoir dans une ligne.

La fonction zeros permet de remplir de 0 tous les éléments du tableau, la fonction ones pour remplir le tableau de 1 mais on peut utiliser la fonction full pour préciser le chiffre de son choix qui remplira le tableau et ainsi utiliser d'autres chiffres. Le fonctionnement reste le même, il suffit de rajouter le chiffre choisi pour remplir le tableau après le tuple.

```
>>> import numpy as np
>>> x = np.full((3, 2), 4)
>>> print(x)
[[ 4.  4.]
 [ 4.  4.]
 [ 4.  4.]]
```

Mais le plus souvent, l'intérêt sera de définir les dimensions et le nombre d'éléments du tableau, sans que les chiffres dans le tableau importe et alors, la fonction empty peut être utilisée.

Elle permet de remplir le tableau de nombre aléatoire tout en délimitant comme les fonctions précédentes le tableau. Le fonctionnement reste le même mais il n'y a plus besoin de s'occuper des chiffres dans le tableau.

```
>>> import numpy as np
>>> x = np.empty((2, 3))
>>> print(x)
[[ 1.76791957e-309 1.49538150e+036 6.96959571e-312]
 [ 1.33360453e+241 1.80594574e-309 2.34189024e-305]]
```

On peut aussi utiliser avec le module random donné par NumPy la fonction randn pour remplir son tableau de chiffre aléatoire. Cependant, les chiffres seront orientés vers 0 et il ne faut pas utiliser de tuple pour cette fonction.

```
>>> import numpy as np
>>> x = np.random.randn(2, 3)
>>> print(x)
[[ 0.02589327  0.45107346  0.75404118]
```

```
 [-0.50015038 -0.14491038 -0.23552643]]
```

On a vu comment créer un tableau si on connaît tous les éléments, si on souhaite seulement définir ses dimensions et son nombre d'éléments, mais il reste un dernier cas important, si on souhaite définir un tableau à une dimension avec un élément de départ défini, une fin définie et un écart égal entre chaque élément.

Dans ce cas-là, on utilisera deux autres fonctions qui permettront de ne pas devoir recopier tout les éléments du tableau ce qui sera pratique surtout s'il s'agit d'un tableau avec un nombre d'éléments élevé.

La première fonction est la fonction arange qui permet de définir le premier élément, le dernier élément et l'écart qu'il y a entre chaque élément. L'écart par défaut est de 1 avec comme premier élément 0. A ce moment là, il suffit d'utiliser la fonction suivie d'un nombre défini comme la fin du tableau, attention que ce chiffre n'est pas compris dans le tableau, le dernier chiffre du tableau est le chiffre précédent. Si le chiffre choisi est 3, le dernier chiffre du tableau sera donc 2.

```
>>> import numpy as np
>>> x = np.arange(3)
>>> print(x)
[0 1 2]
```

Pour changer la fonction par défaut, il faut écrire après la fonction le premier élément, l'élément de fin et l'écart entre chaque élément.

```
>>> import numpy as np
>>> x = np.arange(2, 5, 0.5)
>>> print(x)
[2. 2.5 3. 3.5 4. 4.5]
```

La fonction ne permet par contre pas de définir le nombre d'éléments dans le tableau. Si l'on préfère connaître le nombre d'éléments au lieu de la valeur de l'écart entre les chiffres, on peut utiliser la fonction linspace.

Tout comme arange, il faut écrire après la fonction le premier élément ainsi que l'élément de fin mais, au lieu de mettre ensuite l'écart entre les nombres, il faut écrire le nombre de chiffres que l'on souhaite avoir dans le tableau. Attention, avec linspace, il faut considérer le chiffre de fin comme le dernier élément du tableau, contrairement à arange où il ne fallait pas.

```
>>> import numpy as np
>>> x = np.linspace(2, 5, 3)
>>> print(x)
[2, 3.5, 5]
```

5.6.3 Les attributs et les manipulations de tableau

Il existe dans NumPy beaucoup d'attributs mais on va évoquer seulement les trois plus intéressants et utilisés :

ndarray.ndim : il permet de connaître le nombre d'axes du tableau. Pour rappel, les dimensions dans NumPy sont appelées axes.

```
>>> import numpy as np
>>> x = np.array([(3,2,3), (4,1,6)])
>>> print(x.ndim)
2
```

ndarray.shape : il donne la taille de chaque axe. Si le tableau a 2 axes, il donnera comme réponse (x,y) avec x qui correspond aux nombres de colonnes du tableau et y pour le nombre de lignes. Il s'agit du même principe que pour les fonctions zeros, ones, full et empty puisque il s'agit en fait du shape que l'on inscrit après la fonction. La taille du tuple varie en fonction du nombre d'axes du tableau.

```
>>> import numpy as np
>>> x = np.array([(3,2,3), (4,1,6)])
>>> print(x.shape)
(2,3)
```

ndarray.size : Il donne le nombre total d'éléments dans le tableau. Cela représente tout simplement le produit de x et y dans le shape (et des autres termes dans le shape si il y plus que 2 axes au tableau).

```
>>> import numpy as np
>>> x = np.array([(3,2,3), (4,1,6)])
>>> print(x.size)
6
```

Il y a aussi dans Numpy différentes méthodes permettant de manipuler le tableau créé.

np.hstack et np.vstack : Ces deux méthodes permettent d'assembler deux tableaux ensemble horizontalement (np.hstcak) ou verticalement (np.vstack) en utilisant la fonction suivie d'un tuple constitué des deux tableaux.

```
>>> import numpy as np
>>> x = np.zeros((2,3))
>>> y = np.ones((2,6))
>>> z = np.hstack((x, y))
>>> print(z)
[[0. 0. 0. 1. 1. 1. 1. 1. 1.]
 [0. 0. 0. 1. 1. 1. 1. 1. 1.]]
```

Attention, pour utiliser la méthode np.hstack, il faut que les deux tableaux aient le même nombre d'éléments dans une colonne (le premier terme de leur shape doit être identique) et que les deux tableaux aient le même nombre d'axe. Par contre, pour utiliser la méthode np.vstack, il faut que les deux tableaux aient le même nombre d'éléments dans une ligne (le second terme dans leur shape doit être identique) et que les deux tableaux aient aussi le même nombre d'axes. Pour les deux méthodes, il est possible de fusionner autant de tableaux qu'on le souhaite ensemble, il suffit de rajouter dans le tuple les tableaux supplémentaires.

reshape : cette méthode permet de changer la forme d'un tableau pour lui donner une nouvelle forme. Cependant, on ne peut pas changer les dimensions et les nombres d'éléments qu'il y a dans le tableau. Le reshape doit donc avoir le même produit que le shape du tableau initial.

```
>>> import numpy as np
>>> x = np.arange(3, 9, 1)
```

```
>>> print(x)
[3 4 5 6 7 8]
>>> print(x.shape)
(6,1)
>>> y = x.reshape((2,3))
>>> print(y)
[[3 4 5]
 [6 7 8]]
>>> print(y.shape)
(2,3)
```

Dans ce cas-ci, le produit 6*1 = 2*3 est vrai, le reshape est donc fonctionnel.

ravel : la méthode ravel permet de transformer rapidement un tableau de deux axes ou plus en un tableau à une dimension.

```
>>> import numpy as np
>>> x = np.ones((2,6))
>>> print(x)
[[1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1.]]
>>> y = x.ravel( )
>>> print(y)
[1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
```

5.6.4 Opérations de base

Pour faire une opération classique comme une addition ou une multiplication entre deux tableaux, il suffit de mettre les deux tableaux avec un +/* entre pour le faire. L'opération se fait élément par élément. L'élément 0 du tableau 1 s'additionne/se multiplie avec l'élément 0 du tableau 2, l'élément 1 du tableau 1 s'additionne/se multiplie avec l'élément 1 du tableau 2, etc. Le résultat donnera un nouveau tableau.

```
>>> import numpy as np
>>> x = np.ones((2,3))
>>> y = np.array([(4,1,2), (2,0,6)])
>>> z = x+y
>>> print(z)
```

```
[[5. 2. 3.]
 [3. 1. 7.]]
```

Il y a moyen aussi de additionner/multiplier le tableau à une constante, ce qui fait comme résultat un nouveau tableau où chaque élément du tableau subit l'opération avec cette constante.

```
>>> import numpy as np
>>> y = np.array([(4,1,2), (2,0,6)])
>>> z = y+3
>>> print(z)
[[7 4 5]
 [5 3 9]]
```

Il est de plus possible de faire un opération sans forcément obtenir un nouveau tableau mais simplement en modifiant un tableau par une opération. Par exemple, multiplier tout les éléments du tableau par la valeur 2. Pour cela, il faut utiliser le signe *= .

```
>>> import numpy as np
>>> y = np.array([(4,1,2), (2,0,6)])
>>> y *= 2
>>> print(y)
[[ 8  2  4]
 [ 4  0 12]]
```

Il existe aussi quelques méthodes utilisables dans les opérations comme sum qui permet de calculer la somme de tous les éléments du tableau ou max et min qui permettent de définir respectivement la valeur maximale et la valeur minimale dans le tableau.

```
>>> import numpy as np
>>> y = np.array([(4,1,2), (2,0,6)])
>>> print(y.sum())
15
>>> print(y.max())
6
>>> print(y.min())
0
```

Dans ces exemples, les méthodes sont utilisées sur tout le tableau mais il y a moyen aussi d'utiliser une méthode que sur un axe précis.

```
>>> import numpy as np
>>> y = np.array([(4,1,2), (2,0,6)])
>>> print(y.sum(axis 0))
[6 1 8]
>>> print(y.max(axis 1))
[4 6]
```

On notera ici que l'axis 0 = les colonnes et l'axis 1 = les lignes : shape = (0,1).

5.6.5 Récapitulatif

```
>>> import numpy as np
>>> a = np.array([(4,2,3), (7,0,8), (6,5,1)])
>>> print(a)
[[4 2 3]
 [7 0 8]
 [6 5 1]]
>>> a *= 2
>>> print(a)
[[ 8  4  6]
 [14  0 16]
 [12 10  2]]
>>> b = np.linspace(1,5,3)
>>> print(b)
[1. 3. 5.]
>>> x = np.vstack((a,b))
>>> print(x)
[[ 8.  4.  6.]
 [14.  0. 16.]
 [12. 10.  2.]
 [ 1.  3.  5.]]
>>> print(x.ndim)
2
>>> print(x.shape)
(4, 3)
>>> print(x.size)
12
>>> c = np.full((2,6),2)
>>> print(c)
```

```
[[2 2 2 2 2 2]
 [2 2 2 2 2 2]]
>>> d = c.reshape((4,3))
>>> print(d)
[[2 2 2]
 [2 2 2]
 [2 2 2]
 [2 2 2]]
>>> y = x+d
>>> print(y)
[[10.  6.  8.]
 [16.  2. 18.]
 [14. 12.  4.]
 [ 3.  5.  7.]]
>>> print(y.max())
18.0
>>> print(y.sum(axis = 0))
[43. 25. 37.]
>>> print(y.min(axis = 1))
[6. 2. 4. 3.]
>>> z = y.ravel()
>>> print(z)
[10.  6.  8. 16.  2. 18. 14. 12.  4.  3.  5.  7.]
>>> print(z.ndim)
1
>>> print(z.shape)
(12, )
>>> print(z.size)
12
```

6 Remerciements.

Nous remercions avant tout Sophie Gayes qui tout au long du projet nous a soutenu et a répondu à nos questions, nous remercions aussi Mr. Joachim sans qui rien n'aurait été possible et qui nous a soutenu plus que tout pendant cette période difficile et longue à passer.