Vers la structure interne du langage

Ing. J.-B. CABO
ECAM – Bruxelles

Cet article expose des notions de base de l'algèbre des chemins dans un graphe telle qu'elle est appliquée en Intelligence Artificielle. L'exposé aborde ensuite les notions de structures récursives des programmes qui peuvent aussi s'appliquer à l'analyse des données pour déboucher sur la notion très "automatique" de fonction de transfert. Quelques résultats de cette démarche sont mentionnés en guise de conclusion.

Mots-clefs : graphe, algorithme, prolog, récursivité, ordre, fonction de transfert

This paper exposes basic notions of path algebra in graphs as they are used in Artificial Intelligence. Then it approaches the concepts of recursive data structures vs. the analysis of recursive operations using them. After the introduction of transfer functions in the field of AI, some results of the application of theses ideas are mentioned as conclusion.

Keywords : graph, algorithms, prolog, recursivity, order, transfer function

1. Introduction

Cet article fait suite à un article précédent sur l'Intelligence Artificielle et qui est paru dans la même revue en février 1992 sous le titre « Introduction au calcul littéral sur ordinateur ». On y trouvait en guise d'introduction que l'absence de théorie sur les systèmes de représentation et de manipulation des connaissances était préjudiciable au développement de cette discipline passionnante liée à l'Automatique par son ambition de se passer de l'opérateur humain. L'objectif de ces quelques lignes est de présenter quelques liens formels qui sont apparus entre ces deux disciplines à l'aide d'exemples simples. Partant d'une complexité élémentaire, nous allons nous rapprocher de celle de la phrase simple vue comme un système de coordonnées dans un repère d'idées.

2. L'Intelligence Artificielle

Le traitement de l'information par les ordinateurs consiste à appliquer des algorithmes ou des séquences d'instructions sur des données. Classiquement, les données viennent d'une source d'information (capteur, fichier, base de données, etc.) et les instructions proviennent d'un programme. Cependant, ces deux éléments distincts tendent à se confondre en Intelligence Artificielle (notée IA par la suite). Les données peuvent devenir des programmes et inversement, les séquences d'instructions sont soupesées au vu du contexte de travail avant d'être éventuellement exécutées.

Certains éléments typiques de l'IA ont été exposés dans le précédent article et illustrés par un petit exemple. Ainsi, l'usage de la récursivité pour définir les programmes et les structures de données est très fréquent. Il en est de même pour la programmation déclarative (où on indique simplement ce que l'on sait), par opposition à la forme procédurale bien connue (où on doit préciser la séquence d'instructions en prévoyant tous les cas). Nous avons aussi abordé les systèmes à règles du type « Si … Alors … » qui sont à la base du langage Prolog. On peut aisément séparer ces règles en deux familles : les règles de réécriture qui construisent du formalisme et les règles de transformation qui rapprochent le problème de sa solution. Nous avions alors montré le cycle thermodynamique du moteur d'inférences dont le travail est proportionnel à la surface dans un diagramme utilité - désordre.

3. L'espace d'état

La notion d'espace d'état est identique en IA et en Automatique. Il s'agit de l'ensemble des valeurs possibles pour un vecteur de variables qui décrivent en nombre minimum et de manière complète l'état du système étudié. Il est courant de représenter les états X d'un problème avec les nœuds d'un graphe G dont les arcs V indiquent les transformations nécessaires au passage d'un état à un autre.

La nature des nœuds et des liens dépend du problème à résoudre. A titre d'exemple, un nœud peut représenter un signal et les arcs des fonctions de transfert ; un nœud peut être une ville et les arcs les routes d'une carte de l'Europe. Il faut de toute manière inventorier les nœuds et les valeurs possibles pour les arcs. On pose n le nombre de nœuds du graphe et S l'ensemble des valeurs pour les arcs. On construit alors la matrice d'incidence (ou d'adjacence) A du graphe G qui est de taille (n,n). La valeur ε est l'élément nul du graphe et représente la situation où il n'y a pas de lien entre les nœuds i et j.

$$A = a_{ij} = \begin{cases} s_{ij} & \text{si } (i,j) \in V \\ \varepsilon & \text{autrement} \end{cases}$$

Prenons l'exemple de la figure 1.a où le robot A doit se déplacer vers la croix située derrière le robot B sous les regards de 4 caméras. L'exposé sera ici limité à l'obtention de la matrice A et à son utilité, mais notons à l'aide de la figure 1.b que la caméra 1 ne verra jamais qu'un mur (I) et que son information sera donc inutile pour orchestrer les déplacements successifs.

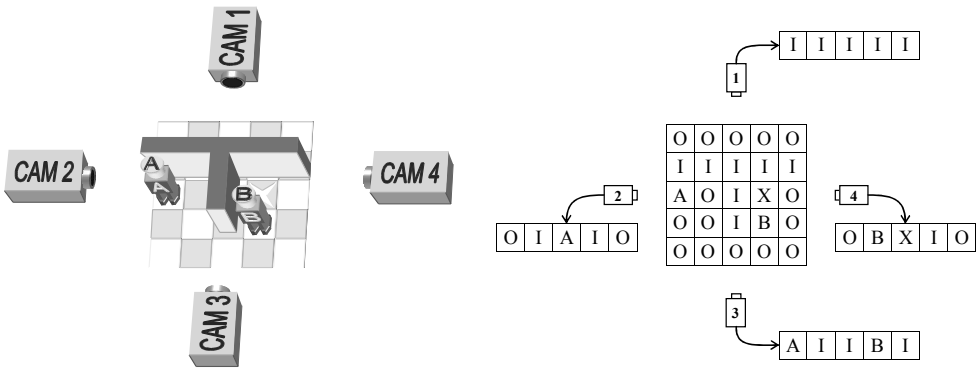

Figure 1.a. Le robot A doit aller sur la croix, b. regards des caméras

Il faut attaquer le problème en établissant le graphe de la figure 2 dont on déduira la matrice A. La zone accessible au robot A n'est constituée que de 13 cases qu'on pourra assimiler aux états du graphe. On a donc n=13. En une étape, le robot peut soit rester sur place (Φ), soit exécuter un déplacement d'une case vers le nord (N), le sud (S), l'est (E) ou l'ouest (W).

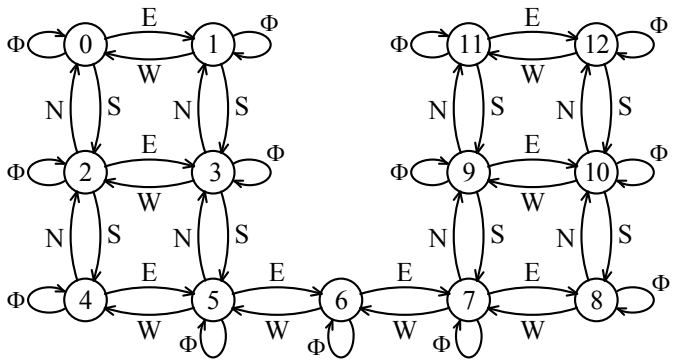

Figure 2 : graphe de déplacement du robot A

Chaque élément $a_{i,j}$ de la matrice A exprime le déplacement en une seule étape de l'état i-1 vers l'état j-1. On voit par exemple qu'on peut passer de l'état 3 à l'état 5 par un déplacement vers le sud puisque $a_{4,6} = S$.

$$A = \begin{pmatrix}
\Phi & E & S & \varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon \\
W & \Phi & \varepsilon & S & \varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon \\
N & \varepsilon & \Phi & E & S & \varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon \\
\varepsilon & N & W & \Phi & \varepsilon & S & \varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon \\
\varepsilon & \varepsilon & N & \varepsilon & \Phi & E & \varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon \\
\varepsilon & \varepsilon & \varepsilon & N & W & \Phi & E & \varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon \\
\varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon & W & \Phi & E & \varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon \\
\varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon & W & \Phi & E & N & \varepsilon & \varepsilon & \varepsilon \\
\varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon & W & \Phi & \varepsilon & N & \varepsilon & \varepsilon \\
\varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon & S & \varepsilon & \Phi & E & N & \varepsilon \\
\varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon & S & W & \Phi & \varepsilon & N \\
\varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon & S & \varepsilon & \Phi & E \\
\varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon & S & W & \Phi
\end{pmatrix}$$

Revenons maintenant à des considérations plus générales. En IA, on désigne le nœud de départ comme la prémisse du raisonnement qui aboutit au nœud de conclusion aussi appelé « goal » ou objectif à atteindre. Le raisonnement n'est rien d'autre que le chemin parcouru dans le graphe.

Il est naturel que plusieurs chemins relient deux états du problème. On définit la prise en considération de chemins parallèles par un opérateur d'addition généralisée dont la forme et les unités dépendent des points de départ et d'arrivée communs. L'élément neutre ε de l'addition est appelé « élément nul ».

$$c_{ij} = a_{ij} \oplus b_{ij} \quad \text{avec le zéro } \varepsilon \text{ tel que } a_{ij} \oplus \varepsilon = a_{ij}$$

La notion de graphe implique aussi la composition entre deux arcs successifs. Cette opération est la multiplication généralisée dont on se dote pour résoudre un problème particulier. Le neutre Φ de la multiplication est « l'élément unité » et représente l'opération de « rester sur place » dans le graphe.

$$c_{ij} = a_{ik} \otimes b_{kj} \quad \text{de i vers j via k} \quad \text{(avec k unique)}$$

$$c_{ij} = \overset{\oplus}{\sum_{1 \le k \le N}} a_{ik} \otimes b_{kj} \quad \text{idem via un quelconque noeud k du graphe}$$

$$\text{unité } \Phi \text{ telle que } a \otimes \Phi = a$$

La composition des transformations matricielles A et B devient une extension du produit matriciel ordinaire C=A*B avec les mêmes règles de compatibilité sur les lignes et les colonnes qui traduisent le fait que les états d'arrivée de A sont aussi les états de départ de B.

C'est ainsi que l'on peut donner une signification concrète aux différentes puissances de la matrice d'incidence. La valeur de l'exposant représente le nombre d'arcs que l'on veut emprunter dans le graphe même si les nœuds ne sont plus nécessairement distincts.

Dans notre exemple précédent, le calcul de A^2 nous fournit les déplacements possibles en deux étapes. Voici un fragment de la matrice A^2.

$$A^2 = \begin{pmatrix} \Phi\otimes\Phi \oplus E\otimes W \oplus S\otimes N & \Phi\otimes E \oplus E\otimes\Phi & \Phi\otimes S \oplus S\otimes\Phi & E\otimes S \oplus S\otimes E & \varepsilon & \varepsilon & \ldots & \varepsilon \\ W\otimes\Phi \oplus \Phi\otimes W & W\otimes E \oplus \Phi\otimes\Phi \oplus S\otimes N & W\otimes S \oplus S\otimes W & \Phi\otimes S \oplus S\otimes \Phi & 0 & 0 & \ldots & \varepsilon \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ \varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon & \ldots & S\otimes N \oplus W\otimes E \oplus \Phi\otimes\Phi \end{pmatrix}$$

Si l'on souhaite passer de l'état 0 à l'état 11, on constate que cela nécessite au moins 7 étapes car ce n'est qu'à partir de A^7 que le terme 1,12 devient différent de ε, en effet, un ordinateur peut calculer :

$$A^7_{1,12} = E\otimes S\otimes S\otimes E\otimes E\otimes N\otimes N \oplus S\otimes E\otimes S\otimes E\otimes E\otimes N\otimes N \oplus S\otimes S\otimes E\otimes E\otimes E\otimes N\otimes N$$

On peut retrouver facilement les trois chemins possibles proposés par ce terme en cheminant de l'état 0 à l'état 7 sur le graphe de la figure 2.

Remarquons ici l'aspect purement combinatoire du calcul matriciel qui ne sert qu'à développer de manière ordonnée l'ensemble des chemins possibles. De nombreux problèmes ne demandent pas une solution en k coups strictement. On préférera savoir si on peut les résoudre en k coups au plus. C'est pourquoi on introduit la matrice $A^{(k)}$ qui est l'ensemble des chemins de i à j formés de k+1 sommets pas forcément distincts.

$$A^{(k)} = E \oplus A \oplus A^2 \oplus A^3 \oplus \ldots \oplus A^k$$

La question de l'existence d'une limite de $A^{(k)}$ pour k tendant vers l'infini est d'importance car si elle aboutit pour une valeur raisonnable de k, nous avons alors un algorithme. Nous notons cette limite A^* (A étoile) qui est aussi le développement en série de $(E - A)^{-1}$.

4. La récursivité du 1er ordre

Un système du premier ordre est une structure contenant un appel récursif. Il concerne un système de représentation avec un seul emplacement pour accéder à un objet. La liste est un exemple très utile.

Définissons une liste d'entités A selon les habitudes du langage Prolog[1]. La liste peut être une liste vide [], ou composée d'un atome [A], ou deux [A,A], ou plus [A,A, … , A].

Toute liste non vide peut être divisée en une tête (le premier élément) et une queue (le reste, éventuellement vide), ce qui est indiqué par l'opérateur | en Prolog :

		exemple
liste_vide	= []	[ ]
liste_1	= [A] = [A | []]	[7]
liste_2	= [A, A] = [A | liste_1]	[18, 7]
liste_3	= [A, A, A] = [A | liste_2]	[13, 18, 7]
liste_n	= [A, A, …, A] = [A | liste_n-1]	[5, 19, 34, 13, 18, 7]

Nous pouvons donc modéliser la structure de la liste par la formule suivante :

A^* = [ ] *ou* [A] *ou* [A, A] *ou* [A, A, A] *ou* …
= toute liste du type A

Introduisons la notation $\oplus$ pour l'opérateur OU :

$$A^* = [\,] \oplus [A] \oplus [A, A] \oplus [A, A, A] \oplus \ldots \oplus [A, A, \ldots, A]$$

Pour accéder à un élément d'une liste ou pour modifier une liste, on dispose de l'opérateur de séparation « | » qui distingue la tête (d'une liste non vide) du reste (appelé queue). On peut définir l'opérateur « | » par

$[A] = [A \mid E]$ noté A^1 (ou A) où E est une liste vide
et $A^{n+1} = [A \mid A^n]$ où A^n est une liste non vide à n atomes

On a dès lors

$$A^* = \lim_{n\to\infty} E \oplus A \oplus A^2 \oplus A^3 \oplus \ldots \oplus A^n = 1 / (E\text{-}A)$$

La définition de l'opérateur « | » implique qu'il n'est possible de retirer un élément d'une liste qu'en parcourant la liste depuis le début.

---

[1] On considère ces entités comme des contenants plutôt que des contenus, on se permet donc d'attribuer ici la même lettre A à plusieurs entités successives *de même nature*, par exemple des entiers, même si leurs valeurs respectives sont différentes. ex: la liste [A, A] pourrait être [17, 9].

Ce type de développement en série permet de se passer des opérations de division et de soustraction, même pour calculer une expression telle que 1 / (E-*A*). C'est une propriété remarquable de ces structures de données que de permettre l'usage d'opérateurs qui ne sont pas autrement définis que par un développement en série. Pensons simplement à la définition des nombres négatifs en binaire qui transforment l'addition en soustraction.

C'est en raison de cette structure que les listes sont définies en Prolog par le nom de la structure répétée suivi d'une étoile * : une liste d'entiers est donc définie par integer*.

La récursivité appliquée aux structures de données (telles que les listes) a pour répondant une structure récursive identique au sein des opérateurs (tels que l'opérateur |). Voici l'analyse du prédicat[2] member destiné à parcourir une liste à la recherche d'un de ses éléments.

```
member(integer, integer*)
E ⇒   member(X,[X|_]):-!.                 /* explication syntaxe voir 3 */
A ⇒   member(X,[_|L]):-member(X,L).       /* explication syntaxe voir 4 */
```

Par exemple, en supposant `liste1 = [5,19,34,13].`
Si l'on appelle le prédicat `member(34, liste1).`
Prolog travaille de la façon suivante :

- ligne E: échoue car 34 ne commence pas la liste, tentons la suivante
- ligne A: member(34, [?,19,34,13]) *est vrai si* member(34, [19,34,13])
- ligne E: échoue car 34 ne commence pas la liste, tentons la suivante
- ligne A: member(34, [?,34,13]) *est vrai si* member(34, [34,13])
- ligne E: member(34, [34, ?]) *est vrai si* VRAI et terminer.

Conclusion : réussi, 34 appartient bien à la liste

[2] Un prédicat en Prolog est comparable à une fonction qui devrait donner une réponse vrai ou faux sur base des entrées qui lui sont fournies (ici un entier et une liste d'entiers).

[3] Cette ligne n'est considérée que si `member(X,[X|_])` a un sens, autrement dit, si on a appelé le prédicat `member` en fournissant un entier et une liste commençant par ce même entier. Le symbole "_" signifie *"une liste quelconque, peu importe"*. Le symbole "`:-`" signifie *"est vrai si"*, le symbole "!" signifie ici *" vrai, inutile de tester la ligne suivante"* .

[4] Cette ligne signifie : *"quel que soit le premier élément de la liste fournie,* member *est vrai si member(entier fourni, queue de liste) est vrai"*. Notons qu'on a désigné par la variable L la queue de la liste tandis que sa tête, désignée par _ nous importe peu.

E est la réponse immédiate, celle des clauses qui termine la récursion.
A est l'opération qui diminue d'une longueur la difficulté du process.

Ainsi, pour l'exemple de l'appartenance de X à une liste : soit X est la tête de la liste, et le programme est terminé; soit, il faut enlever la tête de la liste et recommencer la recherche dans la queue de celle-ci par un appel récursif. Les exécutions possibles du programme A^* appartiennent au développement en série du quasi inverse de A. *On retrouve donc pour les exécutions possibles du programme A^* le même genre de développement en série que pour les listes possibles A^* ou pour la matrice de tous les chemins possibles A^* dont il était question ci-dessus.*

5. La notion de fonction de transfert

Les structures de données et les programmes ont entre eux les mêmes relations que les transformées de signaux et les fonctions de transfert. La démarche générale est conservée : on recherche les structures récursives des formalismes d'entrée et de sortie IN^* et OUT^*, dont le rapport OUT^* / IN^* est la fonction de transfert.

En clair, le numérateur de la fonction de transfert contient la description récursive de ce qui doit être ajouté par le programme, tandis que le dénominateur décrit ce qui doit être enlevé par le programme.

On peut utiliser des fonctions de transfert d'ordre supérieur à 1 et constater que le formalisme du système de représentation reste beaucoup plus simple que les manipulations qu'il autorise.

6. Un modèle du second ordre

Un système du $2^{ème}$ ordre permet de déplacer des objets entre 2 emplacements représentés ici chacun par une liste. Voici un exemple de programme qui calcule les répartitions possibles des éléments entre 2 positions. Il y a deux listes d'entrées qui sont le point de départ du problème et deux listes de sorties qui sont une répartition possible. Il est doublement récursif.

```
domains
    symbollist = symbol*.
Predicates
    subset_gen_2 : (symbollist Place1_init,
           symbollist Place1,
           symbollist Place2_init,
           symbollist Place2) nondeterm (i,o,i,o).
Clauses
    subset_gen_2([],[],A,A):-!.
    subset_gen_2([A|B],[A|C],D,E):-subset_gen_2(B,C,D,E).
    subset_gen_2([A|B],C,D,[A|E]):-subset_gen_2(B,C,D,E).
Goal
    subset_gen_2([fermier, loup, chèvre, chou],C,[],D),
    write(C,"   ",D),nl,fail.
```

Le fonctionnement de ce programme purement académique n'est pas détaillé ici mais voici ce qu'il produit comme résultat :

Place 1	***Place 2***	***Notation binaire***
[fermier, loup, chèvre, chou]	[]	0000
[fermier, loup, chèvre]	[chou]	0001
[fermier, loup, chou]	[chèvre]	0010
[fermier, loup]	[chèvre, chou]	0011
[fermier, chèvre, chou]	[loup]	0100
[fermier, chèvre]	[loup, chou]	0101
[fermier, chou]	[loup, chèvre]	0110
[fermier]	[loup, chèvre, chou]	0111
[loup, chèvre, chou]	[fermier]	1000
[loup, chèvre]	[fermier, chou]	1001
[loup, chou]	[fermier, chèvre]	1010
[loup]	[fermier, chèvre, chou]	1011
[chèvre, chou]	[fermier, loup]	1100
[chèvre]	[fermier, loup, chou]	1101
[chou]	[fermier, loup, chèvre]	1110
[]	[fermier, loup, chèvre, chou]	1111

Il existe cependant une façon plus simple de représenter la situation d'une liste d'objets en attribuant à chacun le numéro de son emplacement. Pour 2 emplacements, nous avons le choix entre 0 et 1. Un nombre binaire du

même format que la liste de départ permet de représenter tous les états du système. On constate aussi la similarité entre les clauses du programme et la définition de la matrice d'adjacence de l'hypercube H(n+1) :

$$H_0 = 0$$

$$H_{n+1} = \begin{bmatrix} H_n & I \\ I & H_n \end{bmatrix}$$

$$n = \textit{la longueur des listes}$$

où l'on trouve une fin de récursivité, deux copies et deux appels récursifs.
L'analyse des valeurs propres de la matrice décrit platement les mouvements possibles dans une telle structure. Une valeur positive représente une progression vers l'avant et, sinon, une valeur propre négative représente un chemin de longueur λ en marche arrière.

$$\lambda\,(n,p) = n - 2(p-1)$$

$$f(n,p) = \frac{n!}{(n-p+1)!(p-1)!} = \textit{multiplicité de } \lambda$$

$$p = \textit{rang de la valeur} \in [1 \;\; n+1]$$

7. Et au-delà...

Dans le cas quelconque de N emplacements de capacité infinie, il y a 2N listes dans le programme et N appels récursifs. Le système de représentation devient un nombre de base N avec autant de chiffres qu'il y a d'objets dans la liste à répartir. La matrice d'adjacence montre bien les N appels récursifs et les N(N-1) copies. La tâche pour l'ordinateur devient rapidement énorme et chacun de préférer un système de coordonnées pour représenter un état.

$$C_{n+1} = \begin{bmatrix} C_n & I & \ldots & I \\ I & C_n & \ldots & I \\ \ldots & \ldots & \ldots & \ldots \\ I & I & \ldots & C_n \end{bmatrix} \; (N \times N) \;\; \textit{sousmatrices} \;\; (N^{n+1} \times N^{n+1})$$

$$C_0 = 0$$

L'analyse des valeurs propres montre bien l'intérêt de ces structures qui permettent des bonds plus importants dans le graphe.

$$\lambda\ (n,p) = (n-p+1)N - n \qquad p \in [1, n+1]$$

$$f(n,p) = \frac{(N-1)^{p-1} n!}{(n-p+1)!(p-1)!}$$

8. Conclusions

Nous avons pu montrer, au travers de quelques exemples, qu'un formalisme simple peut déjà masquer une structure complexe et un important travail de calcul. Cependant, il reflète toujours les transformations du vecteur d'état qui l'ont construit. Il en va de même lorsque nous nous exprimons en langage naturel. Nous utilisons davantage de mots que d'idées à exprimer dans la phrase. Au vu du nombre de mots dans le lexique de la langue, et de la longueur moyenne des phrases, on peut considérer que le contenu récursif d'une phrase est considérable. Et de fait, il y a généralement un million d'expressions possibles pour une phrase de complexité moyenne, quelle que soit la langue.

Le langage permet ainsi d'accumuler de manière simple un immense bagage culturel, le fruit de l'expérience historique acquise par toute une communauté. On y retrouve les points de vue intéressants à la résolution d'une multitude de problèmes dont nous avons même oublié l'existence ! Le langage permet, en peu de symboles, d'accroître notre puissance de calcul beaucoup plus rapidement que n'augmentent les coûts de la communication.

Une grande puissance de calcul par rapport au bagage à transmettre permet de s'affranchir du langage, mais la méthode est fondamentalement inefficace. Ceci revient à deviner le contenu d'un message dont nous n'aurions pas la clef.

Nous devons donc utiliser les liens entre les mots pour comprendre une phrase. Ce qui nous amène à traiter un graphe dont chaque nœud est un mot.

Nous avons dès lors appliqué cette analyse à des structures plus vastes mais guidées par un calcul de vraisemblance, ce qui nous a conduit à manipuler un neurone à sigmoïde pour chaque mot. Et c'est ainsi que nous nous som-

mes approchés des structures naturelles de l'intelligence dont nous comprenons mieux aujourd'hui la capacité étonnante de représentation complexe, qui mêlée à sa capacité de calculer des produits scalaires de grande taille, mérite le respect.

C'est ainsi qu'il nous est encore difficile d'exprimer ce qu'est l'intelligence d'un système. Mais il n'est pas interdit d'affirmer qu'un système intelligent recherche l'intelligence en dehors de lui-même et qu'un système n'est intelligent que s'il est reconnu comme tel par ses pairs.

9. Références bibliographiques

[1] GONDRAN Michel et MINOUX Michel, *Graphes et algorithmes*, 3e édition, Paris, Eyrolles, 1995.
[2] PRINS Christian, *Algorithmes de graphes*, Paris, Eyrolles, 1994
[3] THOMSON LEIGHTON F., *Introduction aux algorithmes et architectures parallèles*, International Thomson Publishing France et Morgan and Kaufmann Publishers Inc.
[4] Site du visual prolog sur internet : http://www.visual-prolog.com

10. Remerciements

L'auteur tient à remercier son collègue Francis Gueuning pour son intervention dans la réalisation de cet article. D'autres personnes ont aussi contribué à cette publication, en particulier celles qui ont aidé à la préparation de l'exposé présenté à l'Université Polytechnique de Varsovie, en octobre 2003, et dont cet article recouvre la première partie.