Résolution du problème du labyrinthe par apprentissage par renforcement avec un réseau neuronal - Extraction d'un réseau de neurones des équations de Navier-Stokes

July 18, 2019

Implémentation : `https://github.com/Daetheys/StageL3` (NOTE : Un collab pourrait être une bonne idée pour plus facilement tester le code et l'expliquer en même temps)

1 Introduction

Mon stage s'est déroulé dans l'équipe SequeL à l'INRIA Lille sous la direction de Philippe Preux. L'équipe SequeL s'interesse à l'apprentissage séquentiel c'est à dire à une forme d'apprentissage où l'information arrive au fur et à mesure que l'agent explore son environnement. Elle travaille nottement sur l'apprentissage par renforcement et sur les problèmes de Bandits.
Mon stage est de 6 semaines et je dois travailler sur la résolution du problème du labyrinthe c'est à dire proposer un agent capable de résoudre n'importe quel labyrinthe donné sous la forme de tableau de 0 et de 1. Je dois le faire par apprentissage par renforcement ce qui est un exercice assez classique en utilisant du Q-Learning tabulaire par exemple. Or pour ce stage je vais devoir trouver un moyen de m'affranchir de cette table et utiliser un réseau neuronal à la place. Cette dernière contrainte rend le problème beaucoup plus difficile car les informations sensées être stockées dans la table ne sont pas très continues (à cause des murs qui font une rupture) ce qui rend la table très difficile à remplacer par un réseau de neurones classique tout en conservant de bonnes performances. L'enjeu de ce remplacement est double : tout

d'abord toutes les tables ne peuvent pas être apprisent par Q-Learning ce qui fait que l'utilisation d'un réseau de neurone pourrait être bénéfique car il pourrait être capable de trouver un lien entre ces données. Ceci permettrait alors de pouvoir, avec peu de données, prédire d'autres cases de la table ce qui permettrait d'accélérer fortement la vitesse de résolution d'un labyrinthe par Q-Learning. De plus, ce pouvoir prédictif permettrait une factorisation des informations de la table et donc de réduire l'espace mémoire necessaire au stockage des informations.
Je me suis dans un premier temps interessé à l'amélioration en temps car de nos jours l'espace mémoire n'est pas vraiment un problème et devoir résoudre un labyrinthe tellement grand que les approches classiques ne fonctionnent plus me semble être un problème qui n'arrive que très peu en pratique. J'ai alors eu une idée assez originale en observant les champs de vecteurs que devait apprendre le réseau neuronal, en effet, ils ressemblaient à un écoulement d'eau dans le labyrinthe. J'ai donc travaillé sur les équations différentielles de mécanique de fluides (Equations de Navier-Stokes) et j'ai cherché à faire un réseau de neurones qui satisferait ces contraintes.
Ainsi nous allons tout d'abord voir le contexte scientifique et quelques définition afin de pouvoir suivre cette aventure des équations de Navier Stokes avec des réseaux neuronaux particuliers et enfin voir une façon de résoudre des labyrinthes très grands (NOTE : Je vais peut être enlever cette dernière partie car mon rapport est déjà très long).

2 Contexte Scientifique

2.1 Apprentissage par renforcement

L'apprentissage par renforcement est une forme d'apprentissage qui pourrait être vu comme une simulation psychologique du fonctionnement comportemental des êtres vivants. L'agent va devoir maximiser une récompense qu'il va obtenir en effectuant, dans une certaine configuration, une certaine action qu'il aura choisi. Il peut avoir ainsi différentes récompenses, une pour chaque action possible dans la configuration dans laquelle il se trouve et devra maximiser la récompense obtenue (la plupart du temps, sur le long terme). Or l'agent ne connait pas à l'avance les récompense associées aux différentes actions qui lui sont proposées. Il devra donc les découvrir au fil du temps. Il y a alors un choix à faire : explorer de nouvelles possibilités ou exploiter celles

qui sont déjà connues. L'objectif de cet apprentissage est de trouver une bonne politique (fonction de choix, qui a une configuration donnée, donne l'action la plus interessante) permettant de maximiser la récompense.

2.2 Q-Learning

Une des façons de faire de l'apprentissage par renforcement est d'utiliser la méthode de Q-Learning qui consiste en le fait, à partir d'une configuration donnée, de regarder pour chaque action possible les récompenses connues associées. En fonction des données connues et inconnues, on décide d'explorer une nouvelle possibilité ou d'en exploiter une autre. Suite à ce choix, l'agent reçoit de l'environnement la récompense réelle liée à son action qui est alors apprise par l'agent et serra desormais connue. On stocke usuellement ces résultats dans un tableau/dictionnaire et on parle alors de Q-Learning tabulaire.

2.3 Stratégies simples

Il existe plusieures stratégies classiques pour réguler l'exploration et l'exploitation: le epsilon-greedy et le epsilon-first ont l'avantage d'être assez simples. Le concept du epsilon-greedy est de choisir l'exploration avec une probabilité epsilon et d'exploiter le reste du temps (c'est à dire qu'on va beaucoup plus exploiter que explorer). Enfin le epsilon-first consiste en le fait d'explorer pendant les N premiers coups puis d'exploiter le reste du temps. Il existe bien entendu beaucoup de variantes de ces 2 méthodes nottement en faisant varier epsilon en fonction du temps. L'idée étant, la plupart du temps, d'exploiter de plus en plus et donc de faire un mélange bien dosé des 2 concepts.

2.4 Réseau de neurones

Un réseau de neurones artificiel est fondamentalement un graphe orienté sans cycle sur lequel va s'effectuer un calcul par propagation de l'information. Certains noeuds sont dits d'entrée et d'autres de sortie. Une fois les noeuds d'entrée remplis par une valeur (souvent un réel), le calcul peut commencer : chaque noeud dont toutes les connections entrantes sont déjà calculées peut être calculé en effectuant l'opération du noeud sur les données entrantes (par exemple pour un noeud + dont les valeurs les noeuds précédents sont 4,5 et 7, la valeur de ce noeud une fois calculé est 4+5+7=16). On peut donc effectuer

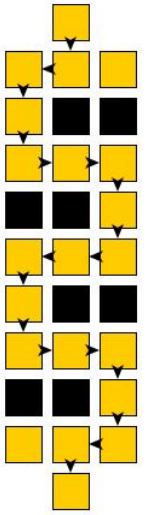
un calcul par propagation des information des noeuds d'entrée jusqu'à ce que les noeuds de sortie soient calculés. On peut alors calculer la fonction représentée par le graphe à partir d'un vecteur d'entrée. Ce qu'on appelle "réseau neuronal" est un tel graphe mais constitué de motifs élémentaires appelés "neurones". Ces neurones sont des petits calculs qui sont, pour les réseaux neuronaux classiques, pour les différentes entrées i : inputi * wi — + — sigma. C'est à dire que l'on somme les entrées pondérées par les wi puis on compose la fonction sigma qui est appellée "fonction d'activation" du neurone. Cette structure imite le fonctionnement des neurones biologiques qui, à partir de potentiels d'action arrivants, et qui sont plus ou moins bien transmit par les synapses, sont sommés puis si le potentiel résultant est suffisant, il se créé un nouveau potentiel d'action au niveau de la "zone gachette" qui serra transmit aux neurones suivants. Ceci me permet donc d'introduire la fonction d'activation Heavyside qui n'est autre que la fonction indicatrice de R+ (si il y a un signal positif, un potentiel d'action est envoyé). Il existe beaucoup d'autres fonctions d'activation comme relu(x) = max(0,x) qui est très utilisée.

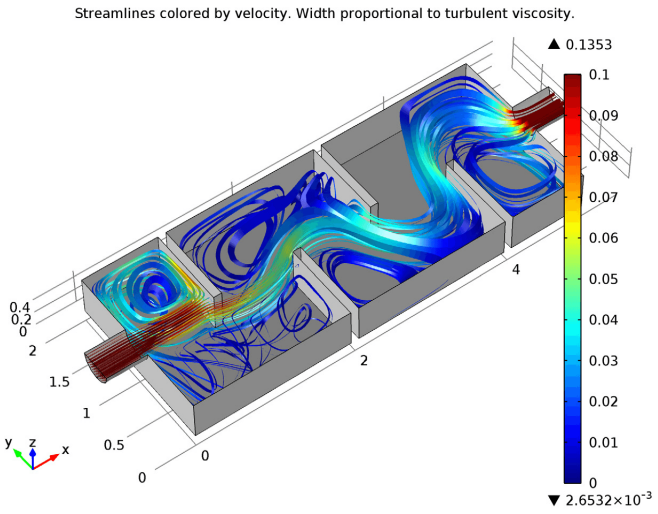
3 Les équations de Navier Stokes

3.1 Motivation

Les approches classiques de resolution du problème du labyrinthe par renforcement avec un réseau de neurones consomment beaucoup plus de place mémoire qu'une approche tabulaire et prennent bien plus de temps pour résoudre le labyrinthe et apprendre une solution. C'est donc le but de mon stage que de trouver une nouvelle approche afin de réussir à mieux utiliser les réseaux de neurones et à mieux les comprendre pour être capable de résoudre ce problème classique de manière plus efficace. Après avoir programmé une résolution du labyrinthe par Q-Learning tabulaire, il restait à trouver une forme de réseau de neurones (ce qu'on appelle "topologie" du réseau de neurones) qui serrait capable de remplacer la table. D'après mon maître de stage, il serrait plus simple de trouver une topologie capable d'apprendre les actions à effectuer directement plutôt que les récompenses associées à chaque action. Il s'agirait donc de partir sur une variante du Q-Learning dont le concept serrait, au lieu de stocker une récompense dans la table, de stocker directement les actions les plus rentables pour l'agent. Ceci serrait plus facile à

apprendre pour un réseau de neurones. Or si on sait que les réseaux de neurones classiques ne sont pas bons pour résoudre ce problème, je pense qu'il est plus interessant de partir sur autre chose. En effet, le fait que les réseaux de neurones classiques soient si mauvais à cette tâche n'est pas étonnant, ces approximateurs de fonctions ne peuvent approximer facilement que des fonctions très continues et dès qu'il faut traiter d'un problème assez complexe, le nombre de neurones necessaires explose. Or ce qui fait la puissance des réseaux de neurones est le fait qu'ils soient composés de petits éléments que sont les neurones et que ces neurones vont interagir entre eux dans une structure pour résoudre des problèmes complexes. On peut alors penser que cette difficulté à résoudre ce problème de labyrinthe pourrait venir des composants élémentaires des réseaux de neurones classiques qui sont optimisés pour apprendre des fonctions continues et que en les remplaçant par d'autres mieux optimisés pour les environnements qui ont la forme d'un labyrinthe, on pourrait avoir de meilleurs résultats.
D'un autre coté, en étudiant les tables que le réseau de neurone devait apprendre, je me suis rendu compte que si on considérait cette table comme un champs de vecteurs, cela ressemblait à l'écoulement d'un fluide de l'entrée vers la sortie.





(NOTE : Il faudra probablement refaire les images car je n'ai pas les droits sur l'image de droite et que ce serra surement plus interessant de donner une resolution par Navier Stokes sur un labyrinthe 25x25)
Je me suis alors interessé aux équations de Navier-Stokes qui régissent la

mécanique des fluides et je me suis demandé comment il était possible de designer un neurone capable d'apprendre efficacement des solutions de ces équations. Puis en me souvenant du cours d'apprentissage d'algorithmique 2 de L3, je me suis dit : "Pourquoi seulement pouvoir approximer facilement des solutions et ne pas plutôt pouvoir apprendre **uniquement** des solutions ?". Les solutions des équations de Navier-Stokes serraient alors notre ensemble dit d'hypothèses (ensemble des fonctions dans lequel on va chercher notre fonction solution).

3.2 Navier-Stokes, les contraintes

Les équations de Navier Stokes sur le champs de vitesse sont les suivantes :

$$div\overrightarrow{v} = \overrightarrow{0}$$
$$\rho(\frac{\partial\overrightarrow{v}}{\partial t} + (\overrightarrow{v}.\overrightarrow{\nabla}).\overrightarrow{v}) = -\overrightarrow{\nabla}p + \mu\nabla^2\overrightarrow{v} + \overrightarrow{f}$$

Pour le moment, personne n'a trouvé de solution générale de ces équations différentielles. Le problème de savoir s'il existe une solution pour toute condition initiale pour les équations à 3 dimensions est même un problème considéré comme aussi difficile que P=NP. Heureusement les labyrinthes considérés ici sont en 2 dimensions donc nous n'aurons pas ce problème. Je compte résoudre ces équations avec un maillage simple : mon labyrinthe est constitué de cases, il s'agirait de prendre un point au centre de chaque case. De plus, ces équations ont une inconnue de plus : le champs de pression p (on n'utilisera pas le terme $\overrightarrow{f}$). Il va être utile pour coder les murs et la sortie, il s'agira de mettre une plus forte pression dans les murs (environ 100 fois plus que dans les couloirs) ainsi que une pression très très faible à la sortie (environ 10**5 fois moins que dans les couloirs). Maintenant que le labyrinthe est modélisé, on va supprimer les termes inutiles des équations : je n'ai pas besoin d'un facteur temporel car les fluctuations ne m'interessent pas, je veux juste une solution qui va de l'entrée vers la sortie et qui m'indique vers où aller sur chaque case. On va donc considérer les équations de Navier-Stokes stationnaires et on va aussi oublier le terme de viscosité (plutôt utile près des bords mais je ne m'interesse que au centre des cases) ainsi que le terme de force car on a codé les murs dans le champs de pression. Voici les nouvelles équations de Navier-Stokes que je vais utiliser :

$$div\,\overrightarrow{v} = \overrightarrow{0}$$
$$\rho(\overrightarrow{v}.\overrightarrow{\nabla}).\overrightarrow{v} = -\overrightarrow{\nabla}p$$

A partir de ces équations, il va être possible de récupérer quelques résultats théoriques assez interessants.
Tout d'abord, on ne sait pas actuellement résoudre ces équations explicitement. Il n'est donc pas question d'essayer même de manière détournée de les résoudre pendant mon stage, il va donc être important de baliser le terrain avant d'essayer de faire quoi que ce soit afin d'esperer trouver quelque chose dans mon court séjour au laboratoire. Je vais donc aller beaucoup plus loin dans ces hypothèses et même supposer qu'elles n'ont pas de solution trouvables avec les outils mathématiques actuels. Ainsi comme la plupart des réseaux de neurones classiques ont une sortie calculable explicitement à partir d'un graphe, si j'arrivais à résoudre mon problème général, alors je pourrais résoudre Navier-Stokes ce qui est exclus. Avec cette simple réduction je sais déjà que je ne pourrais pas résoudre mon problème avec un réseau de neurones 100% explicite. Il va donc devoir y avoir des noeuds qui vont devoir executer des opérations non triviales comme par exemple résoudre des équations dont on ne connait pas de solution exacte. On pourrait alors se dire "On va soigner le mal par le mal" et essayer de faire un réseau de neurones dont tous les noeuds serraient des résolutions approchées de sous-équations différentielles et que la sortie finale soit une solution approchée des équations de Navier-Stokes. Cette méthode m'a paru assez originale mais n'a rien donné de mon coté (c'est une piste qui peut être très intéressante à suivre mais qui dépasse complétement mes connaissances en équations différentielles actuellement).
De plus, si je fais un réseau de neurones, j'ai besoin de le faire apprendre (sinon je ne pourrais jamais le faire remplacer ma table). La méthode d'apprentissage la plus utilisée est la descente de gradient, je dois donc être capable de dériver partiellement mes fonctions dans mon réseau de manière explicite si possible (il faut être précis quand on fait des descentes de gradient si on veut avoir de bons résultats surtout si je veux avoir des solutions de ces équations non linéaires après apprentissage). Ces fonctions sur mes noeuds doivent donc être de dérivée explicite mais qu'on ne puisse pas les intégrer explicitement sinon on résoudrait Navier-Stokes. Ce jeu de contraintes était suffisament complexe pour me faire penser à une autre façon de faire.

3.3 Navier-Stokes par un réseau de neurones, contourner les contraintes

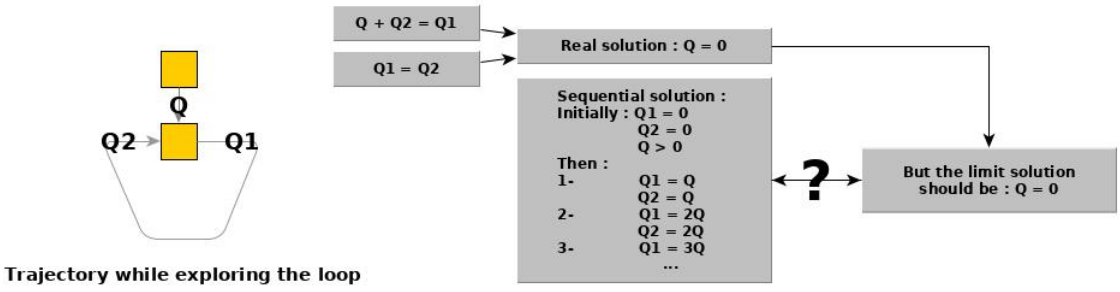
Si je ne peux pas apprendre par descente de gradient, il va falloir trouver une autre façon d'apprendre. De plus, les équations différentielles sont des équations de continuité locale et mes dérivées sont selon X et Y. L'idée la plus logique serrait alors de faire un plan de neurones, il serrait alors plus simple de faire ces dérivées partielles. De plus, pourquoi ne pas faire dans un premier temps une "table de neurones", on n'y gagnerait pas en place par rapport au Q-Learning tabulaire mais on pourrait utiliser la puissance d'induction des réseaux de neurones pour apprendre les liens qui lient les elements de la table et donc pour avoir un pouvoir prédictif bien supérieur à la méthode tabulaire. On aurait alors un gros gain en temps. L'interet de cette table de neurones serrait aussi d'avoir une méthode d'apprentissage efficace car on n'aurait que a donner au neurone la valeur qu'on veut qu'il apprenne (ce ne serraient plus les connections entre neurones qui auraient des poids mais les neurones eux-mêmes) puis il se chargera de reconstruire une solution de Navier-Stokes en modifiant légèrement les neurones aux alentours en profitant de cette structure de plan pour faire ses calculs efficacement. On n'aurait donc pas de solution explicite de Navier-Stokes car on n'aurait pas de fonction à qui on communiquerait les coordonées x et y et qui renverrait vx et vy et qui aurait des poids internes au réseau modifiables donnant toute une famille de solutions. Ici on n'a qu'un êtat, qu'une solution statique qui serrait alors modifiée en une autre quand on ferrait apprendre un autre neurone. De plus, on n'a pas besoin de descendre le gradient donc les 2 conditions de la partie précédente sont satisfaites. Ceci semble donc une bonne approche.

3.4 Les equations de Navier-Stokes séquentielles

Maintenant que l'on a une piste de topologie et presque un modèle pour les neurones, il m'a semblé important de vérifier que cette solution était bien alignée avec le sujet posé et qu'elle y répondrait bien. Vu que l'on a un agent qui va se déplacer dans le labyrinthe, il est important que cet apprentissage puisse se faire de manière progressive. On va devoir par conséquent trouver une manière de résoudre Navier-Stokes dans ces conditions. Le soucis est que quand on arrive dans un cul de sac, les équations de Navier-Stokes (nottement la divergence), dit que toute l'aile doit avoir une vitesse de 0 (eau stagnante car elle n'est pas dans le courant qui relie l'entrée à la sortie). Or

si on se permet de faire une propagation dans les neurones (pour remonter l'information et modifier la valeur des neurones de l'aile), ce ne serrait pas un cout énorme par rapport à un réseau de neurones classique qui necessite à chaque apprentissage de rétropropager de l'information à travers tous ses neurones (ici on ne passerait que par quelques uns qui sont dans une même composante qui s'est revelée être une impasse). Cependant, en terme de complexité dans le pire des cas, ce serrait de l'ordre d'un algorithme de Dijkstra (et si on s'autorisait une telle complexité, autant utiliser un Dijkstra dès le début et se souvenir du chemin, ca bien plus efficace). Il faut donc trouver un moyen de corriger le courant dans ces composantes sans avoir à retropropager et uniquement à partir des mouvements de l'agent (on ne s'autorise que à modifier les neurones connexes au neurone sur lequel on est).

Il se trouve que si l'agent se trouve au fond d'un cul de sac, il va devoir faire le chemin en sens inverse, arriver à un croisement puis repartir une autre direction. Or la divergence dit, à chaque pas qu'il ferra pour revenir, que le courant sur sa case vaudra 0. On pourrait alors résoudre la divergence sur chaque case lors de son retour progressif et ainsi résoudre au moins la divergence de manière séquentielle. De plus, une fois que plusieurs culs de sac auront été trouvés et que l'agent serra remonté à un croisement dont toutes les branches descendants le courant ont un courant nul, le courant du croisement serra alors lui aussi nul. On pourra alors encore une fois remonter et donc résoudre la divergence sur tout le labyrinthe de manière progressive si on n'a pas de cycle. Il y a cependant un véritable problème à résoudre la divergence en cas de cycle de manière séquentielle.



En effet, en fonction de notre façon de résoudre les équations, les valeurs de vitesse pourraient s'emballer et l'agent pourrait se retrouver bloqué dans la boucle. Il n'y a pas vraiment de solution facile à pour résoudre ce problème séquentiellement sans ajouter de complexité temporelle ou spaciale, on le gardera à l'esprit pour essayer de trouver une solution plus tard quand on con-

naitra mieux tout ça (on peut noter que tant qu'à utiliser la divergence dans les équations, pouquoi ne pas bloquer le rotationel et ajouter une équation comme par exemple :

$$\overrightarrow{rot}\overrightarrow{v} = \overrightarrow{0}$$

mais elle n'est pas facile à résoudre séquentiellement et le comportement des fluides n'exclu pas les tourbillons donc elle entrera probablement en conflit avec les équations de Navier-Stokes dans certains cas). Enfin, la dernière équation n'est clairement pas triviale donc je vais me contenter de la discrétiser et de voir ce que cela va donner séquentiellement.

3.5 Vers des équations plus discrétes

Après avoir fait une première implémentation d'une résolution de Navier-Stokes, je me suis rendu compte que ces équations ne sont vraiment pas facile à résoudre, même numériquement. En effet, une simple imprécision peut avoir des effets énormes sur la suite de la résolution. Ce n'est pas pas étonnant de ne pas trouver d'implémentation "triviale" d'une résolution de ce type d'équation sur internet. La plupart utilisent des bibliothéques comme Dolfin (mais j'ai besoin de savoir tous les calculs faits afin d'en faire un graphe pour faire ma structure de réseau de neurones) ou bien cette résolution est un sujet de thèse qui résulte en 300 pages de résultats techniques et d'implémentations complexes (souvent avec la méthode des éléments finis qui est bien pour essayer de simuler un environnement continu, ce qui ne m'interesse pas vraiment). J'ai alors pensé justement à essayer d'augmenter la résolution de mon environnement afin d'avoir des solutions moins approchées et plus continues, puis de faire une moyenne sur chaque case mais cela restait assez compliqué nottement vu le nombre d'inconnues dans les équations et la complexité de la méthode des éléments finis. (Je rapelle qu'on veut en faire un graphe après et qu'il doit rester le plus simple possible, il n'est donc pas question d'introduire les polynômes utilisés dans cette méthode). L'idée que j'ai eu à ce moment a été la suivante : "Vu que j'ai travaillé pendant déjà quelques semaines sur la problématique des équations de Navier-Stokes, pourquoi je n'essairais pas de faire moi-même mes équations de "Navier-Stokes" adaptées à mon environnement". Voici la liste des nouvelles équations :

$$div(\overrightarrow{v}) = \begin{cases} \alpha & \text{à l'entrée} \\ -\alpha & \text{à la sortie} \\ 0 & \text{sur le reste des cases} \end{cases}$$

$$\overrightarrow{v} = \overrightarrow{ave}\,\overrightarrow{v} + \lambda\overrightarrow{v_{learn}} + \overrightarrow{dev}$$

avec

$$\overrightarrow{v} = \begin{pmatrix} vx \\ vy \end{pmatrix}$$

$$\overrightarrow{ave}\,\overrightarrow{v} = \begin{pmatrix} \frac{vx(x+1,y)+vx(x-1,y)}{2} \\ \frac{vy(x,y+1)+vy(x,y-1)}{2} \end{pmatrix}$$

$$\overrightarrow{dev} = sv(vx).sv(vy).(cy.\frac{\|vy\|}{2} - cx.\frac{\|vx\|}{2}).\begin{pmatrix} 1 \\ -1 \end{pmatrix}$$

$$sv(x) = \begin{cases} sg(x) & sg(x) \neq 0 \\ random(\{-1,1\}) & sinon \end{cases}$$

$$sg(y) = \begin{cases} 1 & y > 0 \\ 0 & y = 0 \\ -1 & y < 0 \end{cases}$$

$$cy = (-vy(x,y-1)vy(x,y) \geq 0) + (-vy(x,y+1)vy(x,y) \geq 0)$$

$$cx = (-vx(x-1,y)vx(x,y) \geq 0) + (-vx(x+1,y)vx(x,y) \geq 0)$$

$$(x >= 0) = \begin{cases} 1 & x \geq 0 \\ 0 & \text{sinon} \end{cases}$$

J'introduit aussi la fonction P qui servira plus tard.

$$P(x,y) = \begin{cases} 1 & \text{si il y a un obstacle en x y} \\ 0 & \text{sinon} \end{cases}$$

Ces équations ont l'air un peu complexes mais en réalité elles sont très simples. Leures expressions sont assez compliquées car elles reposent souvent sur des disjonctions de cas, choses faciles à faire de manière discrète mais beaucoup moins de manière continue ce qui explique ces nombreuses notations.
Le terme de divergence indique simplement une conservation de la matière

(on n'est pas sensé perdre de fluide dans le labyrinthe), on a du fluide à déplacer depuis l'entrée qui serra absorbé à la sortie. Enfin la deuxieme équation se compose en 3 termes : un terme de continuité:

$$\overrightarrow{avec}\overrightarrow{v}$$

un terme d'apprentissage forcé:

$$\lambda\overrightarrow{v_{learn}}$$

et un terme d'exploration :

$$\overrightarrow{dev}$$

. Le terme de continuité sert à essayer de conserver une continuité de vitesse entre les cases. Cependant ce terme n'est pas parfait et n'assure pas la conservation de la matière ce qui posera des problèmes pratiques plus tard (on pourrait ajouter un terme de correction assez facilement mais je voulais rester sur quelque chose de relativement simple). Le terme d'apprentissage forcé permet d'augmenter la valeur de vitesse dans un sens ou dans l'autre en fonction de ce que l'on veut faire. Il faut savoir que manipuler ce terme peut être assez dangereux si on veut avoir de bons résultats car il force l'algorithme à apprendre une valeur spécifique qui pourrait nuire au pouvoir anticipatif de l'algorithme.En effet, le système d'équations est assez complexe et chaotique ce qui fait qu'il est hautement improbable en donnant à la main plusieures vitesses, qu'elles concordent avec les équations. Il est donc plutôt là pour inciter le réseau de neurones à considérer une solution particulière allant plutôt dans une direction à partir d'un point. (De futurs travaux pourraient conduire à de meilleurs modèles qui integreraient mieux ce facteur d'apprentissage) Enfin le dernier terme permet de faire en sorte que le réseau neuronal explore par lui même (voir chapitre suivant), ce qui rend justement ce facteur d'apprentissage anecdotique. Le plus gros du travail serra fait pas le pouvoir de prédiction du réseau neuronal.

3.6 Resolution des équations de "pseudo Navier Stokes"

Ces équations ont malheureusement de nombreuses solutions en fonction de l'endroit où on veut les résoudre. Le principe de l'apprentissage séquentiel est que nous n'allons regarder que les cases adjacentes à notre agent et résoudre ces équations sur ces cases. Pour qu'elles soient résoluent de manière plus

pertinente, vu que l'agent est en mouvement, on considère qu'il arrive d'une case adjacente et que l'on veut résoudre les équations sur la nouvelle case sur laquelle il arrive. On suppose donc que la valeur de la case depuis laquelle l'agent vient est correcte, elle serra donc considérée comme étant une constante, ce qui permettra de véritablement propager de l'information dans la labyrinthe. De plus on suppose que la vitesse dans les murs est de 0 (donc constante), ce que l'on peut écrire en modifiant la première équation de la forme (mais elle est un peu moins digeste) :

$$divx + divy = \begin{cases} \alpha & \text{à l'entrée} \\ -\alpha & \text{à la sortie} \\ 0 & \text{sur le reste des cases} \end{cases}$$

avec

$$divx = vx(x+1,y).(P(x+1,y)=0) - vx(x-1,y).(P(x-1,y)=0)$$

$$divy = vy(x,y+1).(P(x,y+1)=0) - vy(x,y-1).(P(x,y-1)=0)$$

On pourrait aussi garder l'ancienne équation et ajouter celle ci, beaucoup plus digeste mais moins explicite à résoudre :

$$P\overrightarrow{v} = \overrightarrow{0} \qquad \textit{On tue la vitesse dans les murs}$$

Ainsi pour résoudre la divergence, si on est dans un couloir, on a 2 murs, et la case de derriere donc il ne reste qu'une inconnue : on peut résoudre. On remarque que si on est dans un cul de sac, on n'a pas d'inconnue et seulement dans ce cas on se permet de modifier la case de derriere (qui vaudra 0 par conséquent). Dans les cas de croisements, il existe plusieures solutions aux équations de Navier-Stokes et il en est de même pour nos équations de "pseudo Navier Stokes". Il va donc falloir faire un choix pour les résoudre. Pour ce faire, on va faire le choix d'augmenter les courants déjà existants proportionnellement à leur force (un courant faible serra peu affecté tandis qu'un courant fort variera beaucoup plus). On va donc utiliser ces formules :

$$v\alpha(x+a,y+b) += \begin{cases} \frac{s.\Delta div * vx(x+a,y+b)}{somme} & \text{si somme} \neq 0 \\ \frac{s.\Delta div}{nb\_couloirs} & \text{si somme} = 0 \end{cases}$$

avec a et b dans [-1,0,1] tels que a.b = 0 et α pouvant signifier x ou y

$$s = a + b$$

$$\Delta div = -div(\overrightarrow{v})(x,y) + \begin{cases} \alpha & \text{(x,y) est l'entrée} \\ -\alpha & \text{(x,y) est la sortie} \\ 0 & \text{sinon} \end{cases}$$

$$somme = sumx + sumy$$

$$sumx = \|v(x+1,y)\|.(1-P(x+1,y)) + \|v(x-1,y)\|.(1-P(x-1,y)$$

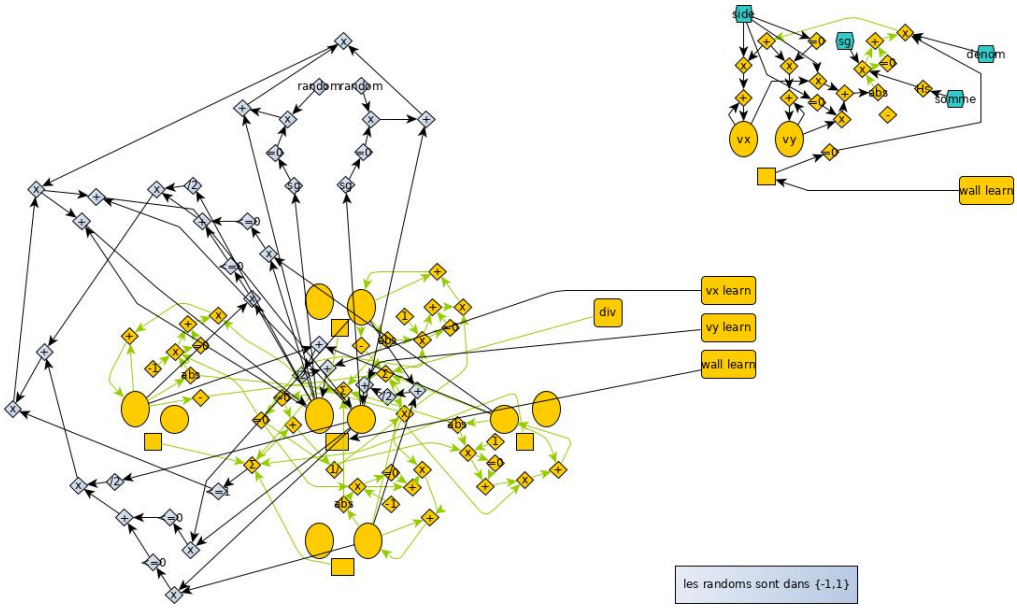
$$sumy = \|v(x,y+1)\|.(1-P(x,y+1)) + \|v(x,y-1)\|.(1-P(x,y-1))$$

$$nb\_couloirs = \sum_{\substack{(i,j)\in[-1,0,1] \\ i.j=0}} P(x+i,y+j)$$

Avec ces formules on trouve les nouvelles valeures des neurones proches en faisant varier a et b (on ne doit pas essayer de résoudre la divergence dans un mur (si (x+a,y+b) est un mur), c'est écrit dans les équations).
Maintenant que l'on connait les valeurs des neurones proches, on va pouvoir trouver la valeur du neurone en (x,y), pour cela il suffit simplement d'utiliser la formule donnée précédement. Attention cependant, les valeurs de vitesse pour les cases adjacentes qui apparaissent dans le terme de déviation ($\overrightarrow{dev}$) sont celles avant la correction de la divergence (sinon ça marche beaucoup moins bien). Je vous laisse tester vous même le code sur le github si vous voulez voir comment cela fonctionne.

3.7 D'un algorithme séquentiel vers un réseau neuronal

Le sujet du stage reste de résoudre le problème avec un réseau de neurones. Il va donc falloir s'y ramener mais tout le travail effectué jusqu'à maintenant a été fait pour rendre cette tâche plus facile. En effet, on a une résolution locale des équations (c'est pourquoi les equations différentielles peuvent être interessantes pour faire des réseaux de neurones en apprentissage séquentiel), ce qui permet de résoudre l'équation sur chaque point indépendament des autres (il faut tout de même que ces points soient accédés tours à tours selon une trajectoire car les équations ont étées faites pour un apprentissage séquentiel). On peut donc imaginer un plan avec des motifs élémentaires qui organiseraient leur voisinage immédiat en communiquant entre eux tout en respectant le procédé decrit ci-dessus qui ne donne que des solutions des équations de "pseudo Navier Stokes". J'ai donc construit le graphe de ces calculs afin de dessiner des machines élémentaires qui seront nos neurones.



(NOTE : une fois que je l'aurai rendu un peu plus joli, je le mettrais surement à la fin en annexes en format paysage - histoire de l'avoir en plus gros)

Ainsi le réseau de neurones trouvé est simulé par l'algorithme actuellement sur le github (NOTE : je ferai surement aussi une implémentation avec directement le réseau de neurones afin de montrer que tout marche bien). Nous avons donc bien réussi à faire un réseau de neurones qui ne peut donc apprendre que des solutions des équations de "pseudo Navier Stokes".

3.8 Retour sur l'apprentissage par renforcement

En réalité on n'a pas besoin de beaucoup ajouter de choses à ce réseau de neurones. En effet, il a déjà la capacité d'explorer par lui-même. Ainsi d'un point de vue de l'apprentissage par renforcement, il s'agit juste de se laisser emporter par le courant sur la case actuelle (donc exploiter ce que propose le réseau) et on résoud le labyrinthe de manière optimale (aucun mouvement inutile, le réseau mémorise tout). La façon dont on va choisir la prochaine case sur laquelle aller à partir du réseau de neurones va dépendre non seulement de la case actuelle mais aussi des cases connexes : on détermine un score pour x et un pour y :

$$scorex = \begin{cases} 0 & sg(x) = 0 \\ vx(x,y).vx(x + sg(vx), y) & sinon \end{cases}$$

$$scorey = \begin{cases} 0 & sg(y) = 0 \\ vx(x,y).vx(x,y+sg(vy)) & sinon \end{cases}$$

Une fois ces grandeurs calculées, on prend la plus grande et on effectue le mouvement associé à sg(vx) ou sg(vy) en fonction de si c'est scorex ou scorey le plus grand. En cas d'égalité on prend aléatoirement l'un des 2 sauf si les 2 scores valent 0 (dans ce cas, on ne peut pas utiliser sg(vx) ni sg(vy) car cela ne donne aucune information). On va donc choisir un mouvement possible (c'est le seul moment où on va faire attention à ne pas marcher sur un mur, le reste du temps les équations le font toutes seules) en faisant attention, si possible, de ne pas retourner sur la dernière case sur laquelle on était. Ce cas où scorex = scorey = 0 arrive quand l'agent à trouvé une impasse et essaie donc de remonter le cours d'eau pour retrouver du courant. A chaque pas il va mettre à 0 la valeur de la vitesse de la case suivante et va ainsi progressivement remonter car il ne retourne pas sur la case sur laquelle il était avant. Un problème se pose si il arrive sur un croisement dont toutes les branches sont de vitesse nulle (il a déjà éliminé les autres couloirs). Dans ce cas, il pourrait repartir dans une branche déjà visitée et perdre du temps. C'est ici que l'on va pouvoir aider le réseau de neurones car il a raison de dire que les vitesses sont nulles mais il ne faut pas que l'agent se perde. Il suffit donc, avant de résoudre la divergence, de chercher une case de vitesse non nulle à proximité et de s'y déplacer. Cette opération se fait en temps et espace constant donc je me suis permis de l'implémenter sur le github. La convergence est ainsi plus rapide.

3.9 Limites techniques de l'implémentation

L'implémentation d'un tel algorithme pose plusieurs problèmes technique dont un majeur qui n'affecte pas le fait de trouver une solution mais qui gène en cas d'exploration de structures très arborescentes dans le labyrinthe. En effet, il y a une faille dans les équations que j'ai trouvées (j'en avait déjà parlé dans le chapitre consacré à leure introduction). La conservation de la matière n'est pas respectée parfaitement par la seconde équation. (NOTE : Il faut que je comprenne pourquoi ca ne marche pas bien) Si rien n'entrait en conflit avec ça, tout fonctionnerait toujours bien (j'avais prévu l'éventualité en designant les équations, ce n'était pas grave pour la résolution). Or regulièrement, quand l'agent arrive dans un cul de sac, il doit mettre les valeurs de ce couloir

à 0. Ce qui fait qu'il calcule le terme vu précédement pour résoudre l'équation de divergence. Cependant ce terme qui doit être l'opposé de la valeur actuelle sur la case a été calculé différement de la valeur actuellement sur la case. Il en résulte un problème bien connu des flottants en informatique : on n'obtient pas parfaitement 0. Or l'algorithme fait une très grande différence entre les valeurs valant 0 (pas de courant) et celles ne valant pas 0 ce qui fait qu'il va falloir "cut" ces valeurs trop faibles pour les mettre à 0. Or comme je l'ai expliqué juste avant, ces valeurs faibles sont mélangées avec celles qui sont liées à un problème de conservation de la matière. Je ne sais donc pas quelles valeurs je dois cut ! Ceci résulte en le fait que parfois l'agent va se perdre dans des zones à 0 de courant et effectuer une marche aléatoire pour trouver son chemin. Une fois qu'il l'aura retrouvé, tout rentrera dans l'ordre jusqu'à la prochaine fois qu'il se perdra (ceci arrive donc près des impasses et est particulièrement embettant dans les zones très arborescentes avec beaucoup d'impasses).

3.10 Résultats et comparaison au Q-Learning classique

[FAIRE LES GRAPHIQUES ET LES INSERER ICI - Encore en cours de travail]

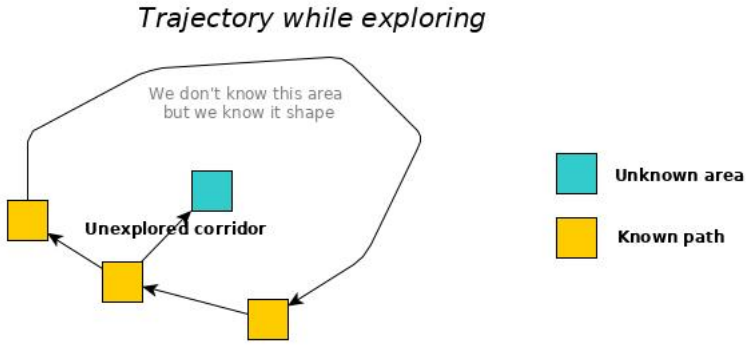
3.11 Conséquences de ces travaux

Ces travaux ont permis de créer un réseau de neurones fait spécifiquement pour les besoins du problème à partir de contraintes sous formes d'équations. Cette méthode diffère de beaucoup des autres méthodes similaires dans le sens où, on n'utilise pas la fonction de loss des réseau de neurones classiques pour approcher des solutions mais on est certain à 100% que les contraintes sont respectées. Ces contraintes offrent au réseau de neurones un pouvoir prédictif très puissant lui permettant de résoudre le labyrinthe de manière bien plus efficace que un Q-Learning tabulaire ou un Deep Q-Learning (avec un réseau de neurones). On pourrait alors imaginer appliquer ce procédé à d'autres problèmes sur lesquels on pourrait faire de plus grandes avancées (voir les chapitre Reflections sur le problème du labyrinthe pour les limites des avancées sur ce problème), comme par exemple les echecs. En effet, le plus difficile ces temps ci en apprentissage par renforcement, c'est de traiter des problèmes avec beaucoup d'actions possibles. Il est alors difficile pour l'agent de les tester toutes rapidement ce qui fait que la phase d'exploration

est très longue. En ajoutant des contraintes pour guider l'agent, on pourrait drastiquement réduire le nombre d'actions necessaires pour résoudre un problème. Par exemple, actuellement, AlphaZero, l'IA championne du monde aux echecs, shogi et Go utilise 2 fois plus d'output que le nombre de cases de l'echiquier (soit 128 pour les echecs). Or aux echecs, on peut coder un movement en 4 coordonées entre 1 et 8. Il s'agirait donc de faire un réseau sous contraintes avec 4 output et avec pour contraintes : les outputs sont dans [1,8], les 2 premieres sorties codent la case de départ du mouvement donc une pièce du joueur IA doit s'y trouver, les 2 deuxiemes sorties codent la case d'arrivée du mouvement donc il faut que ce mouvement soit valide (pas de piece alliée à l'arrivée et il faut que le déplacement soit possible avec la pièce choisie). Bien entendu écrire formellement ces contraintes et trouver un réseau qui les satisfait ne serra pas une chose triviale mais c'est une poursuite possible de ce travail sur les labyrinthes et Navier-Stokes en généralisant la méthode pour trouver des réseaux de neurones. Une autre possibilité extrêmement utile vient du fait que les roboticiens rejettent en bloc l'utilisation du deep learning car ils ont peur que leur robot puisse faire un mal à des humains. En effet le deep learning n'est jamais fiable à 100%. Cependant avec cette méthode, on pourrait dévelloper des réseaux de neurones fiables cette fois ci à 100% et faits spécifiquement pour une tâche (donc a apprentissage très rapide ce qui résoud un autre problème de l'utilisation du deep learning en robotique : il risque d'y avoir de la casse rapidement si l'apprentissage est trop long ou pas assez efficace). Bien entendu, il reste toujours une partie très difficile : écrire formellement les contraintes permettant d'être certain que le robot ne fasse pas de mal à un humain (et il faudra en extraire un réseau neuronal).

3.12 Reflections sur le problème du labyrinthe

Au fond, on ne peut pas faire de magie sur le problème du labyrinthe, on devra forcément explorer les ailes une par une jusqu'à trouver la sortie, la seule amélioration que l'on peut apporter se situe au niveau de la mémoire. C'est à dire se souvenir du chemin utilisé pour aller de l'entrée vers la sortie et être capable éventuellement de changer ce chemin si le labyrinthe change en temps réel. La seule chose que l'agent pourrait savoir sur une partie du labyrinthe sans l'explorer est que si il explore une aile qui entoure une autre.



On peut alors savoir que cette aile est probablement sans issue (il faut pour cela tout de même supposer que le labyrinthe est "physique"). Cette déduction permettrait, dans le cas où la sortie est sur un bord, par exemple, de savoir qu'il ne vaut pas la peine d'explorer l'aile. On peut aussi en fonction de la taille maximale que peut avoir cette aile, si la sortie est dans une salle d'une certaine taille, déduire si oui ou non elle a la place de s'y trouver.

4 Autre algorithme

Je n'aurai probablement pas la place de parler du deuxieme algorithme.

5 Conclusion

A voir en fonction des résultats comparés au Q-Learning tabulaire et au Deep Q-Learning. Je pense que le plus important dans ce travail n'est pas le fait de pouvoir mieux résoudre le problème du labyrinthe (cf Reflections sur le problème du labyrinthe). Le véritable interet et ce qui est vraiment innovant dedans c'est surtout d'avoir été capable d'extraire un réseau de neurones d'un lot de contraintes. Ce concept inédit permettrait de vraiment améliorer les possibilités du machine learning par la suite en permettant de travailler sur des modèles avec une convergence infiniment plus rapide, avec moins de données et avec une meilleure robustesse (on a éliminé plein de fonctions suite à la prise en compte des contraintes). On pourrait alors par exemple, battre Google au niveau de la reconnaissance d'images car la quantité de données ne serrait plus un problème. L'idéal serrait d'avoir une fonction qui étant donné un lot de contraintes, soit capable de produire un réseau de neu-

rones particulier, spécifiquement fait pour résoudre le problème lié au lot de contraintes donné et qui renvoie NONE s'il n'en existe pas. Or ce problème est indécidable (le problème de décision sous-jacent est indécidable - cf annexes). Il va donc falloir travailler sur les ensembles d'équations particuliers ou alors proposer une nouvelle formulation du problème qui, cette fois, soit décidable. Dans tous les cas, il y a une chance pour que ces méthodes soient l'avenir du machine learning car elles permettraient de résoudrent extrêment efficacement des problèmes que l'on n'arrive pas à résoudre aujourd'hui nottement au niveau de l'exploration de l'environnement par apprentissage par renforcement par exemple.

6 Annexes

6.1 Preuve (assez succinte) de l'indécidabilité

Données : Un lot de contraintes (un ensemble d'équations supposé fini de la forme ($\{F_i(X_i) = 0\}_{i\in\mathbb{N}}$ avec les F_i des fonctions et X_i des vecteurs de variables)). Ces équations portent sur une fonction f (celle que l'on cherche) mais peuvent aussi porter sur d'autre objets externes donc l'ensemble est noté E et est fini(comme d'autres fonctions, ce qui laisse beaucoup de possibilités).
Expected Output : Une fonction $f_{W,E}$ (avec W un vecteur de taille n$\in \mathbb{N}$) qui pour tout W de taille n satisfait les contraintes.

On va considérer le problème de décision sous-jacent :
Données : Un lot de contraintes (un ensemble d'équations supposé fini de la forme ($\{F_i(X_i) = 0\}_{i\in\mathbb{N}}$ avec les F_i des fonctions et X_i des vecteurs de variables)). Ces équations portent sur une fonction f (celle que l'on cherche) mais peuvent aussi porter sur d'autre objets externes dont l'ensemble est noté E et est fini.
Problème : Est-ce qu'il existe une fonction $f_{W,E}$ (avec W un vecteur de taille n$\in \mathbb{N}$) qui pour tout W de taille n satisfait ces contraintes ? satisfait les contraintes.

Résumé:
On va réduire le problème de l'arret : On va montrer que l'on peut coder toutes les formules logiques que l'on veut dans les contraintes puis on va coder une machine de Turing avec des portes logiques (je ne ferai pas le détail ici),

enfin, pour une entrée donnée, si cette machine s'arrete sur cette entrée, on ajoute un symbole spécial (disons #) au début de la bande. Enfin on ajoute un contrainte qui dit que ce symbole apparaîtra au bout d'un moment. On aura donc, une fonction f existe ssi la machine s'arrete sur l'entrée donnée. Donc le problème serra indécidable.

Détail: Prenons une machine de Turing M ainsi que un mot d'entrée w. On construit M' qui, quand elle s'arrete écrit # au début du ruban. On code M' en un nombre fini de formules logiques utilisant un nombre infini dénombrable de variables logiques. Maintenant on va réécrire les formules logiques de la machine de Turing de la façon suivante, pour une formule f, on construit une formule $\widetilde{f}$ à travers le processus p (qui tranforme f en $\widetilde{f}$):

$$f = a \wedge b \rightarrow \widetilde{a}.\widetilde{b} = \widetilde{f}$$

$$f = \neg a \rightarrow 1 - \widetilde{a} = \widetilde{f}$$

$$f = \forall x.g \rightarrow p(g) = \widetilde{f}$$

$$f = (a = b) \rightarrow (p(a) = p(b)) = \widetilde{(}f)$$

Les autres règles s'en déduisent. La dernière règle utilise une notation curryfiée, on considère que

$$(h : x \rightarrow 0) = 0$$

Ainsi il faut comprendre : si $\forall x.g(x)$ (est vraie), alors g (est vraie) et on transforme g récursivement. Ensuite, pour le stockage des variables, vu que l'on a un ensemble dénombrable de variables, on peut utiliser une fonction Memory :

$$Memory : \mathbb{N} \rightarrow \{0, 1\} \cup MemEns$$

avec MemEns étant l'ensemble des fonctions de $\mathbb{N}$ dans $\{0, 1\} \cup$ MemEns (on peut un peut voir Memory comme un arbre : soit un entier donne une feuille et donc une valeur entre 0 et 1, soit elle donne un noeud : une autre fonction Memory). Ainsi quand on voit une variable logique pendant la transformation par p a_X avec X un vecteur ayant potentiellement plusieures composantes, on va la remplacer par Memory(name(a))[X], avec name un fonction qui étiquette chaque symbole de variable (ne peut apparaitre que en nombre fini) par un entier différent (par exemple, a_X devient 1, b_Y devient

2, ... sans distinction de leurs indices). Ainsi, la fonction Memory principale n'etiquette que les symboles, puis on applique récursivement le procédé suivant :

$$X = (x1, x2, ...xn)$$

$$X2 = (x2, ..., xn)$$

$$Memory(name(a))[X, Y, ...] = \begin{cases} Memory(name(a))[x1, X2, Y, ...] & \text{x1 vecteur} \\ \begin{cases} Memory(name(a))(x1)[X2, Y, ...] & X2 \neq [] \\ Memory(name(a))(x1)[Y, ...] & X2 = [] \end{cases} & \text{sinon} \end{cases}$$

et

$$Memory(name(a))(x1)...(zn)[] = Memory(name(a)(x1)...(zn)$$

Quand la stack d'arbres est vide , on a la valeur de notre variable qui serra dans la contrainte. Ainsi pour chaque variables logique qui apparaît dans les formules logiques, on a un chemin dans Memory qui code cette variable dans [0,1]. De plus, chaque contrainte finira par = 1 (vraie). On va encore rajouter une contrainte :

$$p(\exists n.Bande(n)(0) = \#)$$

avec Bande(t)(x) la fonction qui à un instant t, associe le caractère codé en position x sur la bande. On peut donc bien transformer ces formules booléennes du 1er ordre qui sont en nombre fini en contraintes puis donner ces contraintes à notre fonction dont on veut démontrer la non calculabilité. On a alors : la fonction f existe ssi la machine termine. Donc indécidable.