Algorithmes efficaces de recherche pour une machine chimique pair-à-pair

Matthieu Perrin

Rapport de stage
master informatique spécialité Recherche en informatique
Université de Rennes I

Algorithmes efficaces de recherche pour une machine chimique pair-à-pair

Matthieu Perrin
École Normale Supérieure de Cachan, antenne de Bretagne
matthieupierre.perrin@gmail.com

Stage réalisé
de Février 2012 à Juin 2012

Supervisé par

Marin Bertier
Maître de conférences
INSA de Rennes
Équipe ASAP, IRISA,
INRIA Rennes Bretagne Atlantique
Marin.Bertier@irisa.fr

Cédric Tedeschi
Maître de conférences
Université de Rennes I
Équipe Myriads, IRISA,
INRIA Rennes Bretagne Atlantique
cedric.tedeschi@inria.fr

Résumé

La difficulté de la programmation des systèmes répartis à large échelle est encore aujourd'hui un frein à leur développement. Le paradigme chimique fournit une abstraction de haut niveau adaptée à la programmation de tels systèmes. Cependant, l'exécution de programmes chimiques nécessite la recherche de tuples d'éléments d'un multi-ensemble vérifiant une certaine propriété. Le but de ce rapport est d'améliorer la complexité de cette opération.

Mots clés

Algorithmique répartie, calcul chimique, système pair-à-pair

Internship report
master of Computer Sciences speciality research in computer sciences
University of Rennes I

Efficient research algorithms for a peer-to-peer chemical machine

Matthieu Perrin
École Normale Supérieure de Cachan, Brittany extension
matthieupierre.perrin@gmail.com

Internship conducted
from February 2012 to June 2012

Under the supervision of

Marin Bertier
Assistant professor
INSA Rennes
ASAP teem, IRISA,
INRIA Rennes Bretagne Atlantique
Marin.Bertier@irisa.fr

Cédric Tedeschi
Assistant professor
University of Rennes I
Myriads teem, IRISA,
INRIA Rennes Bretagne Atlantique
cedric.tedeschi@inria.fr

Abstract

The difficulty of programming large scale distributed systems is still a limit for their developement. The chemical paradigm provides a high level abstraction appropriate to the programming of such systems. However, executing chemical programs requires being able to find a tuple of elements satisfying a given property in a multiset. This report aims at presenting a new algorithm that improves the complexity of this operation.

Keywords

distributed algorithms, chemical computing, peer-to-peer systems

"Indeed, I believe that virtually every important aspect of programming arises somewhere in the context of sorting and searching!"

Donald E. Knuth,
The Art of Computer Programming

Table des matières

Introduction

La construction de grosses applications réparties capables de s'adapter de façon autonome à l'évolution de leur environnement est encore aujourd'hui un travail de titans. L'hétérogénéité, la distribution géographique et les faibles garanties de fiabilité offertes par les ressources disponibles imposent une gestion fine des données, des messages et des ressources de calcul afin de garantir des propriétées d'*auto-organisation* en une structure adaptée au problème, d'*auto-adaptation* en fonction de la variation de performance de constituants de la plate-forme au cours du temps, et d'*auto-réparation* pour contrer le manque de fiabilité des machines.

Pour répondre à ces difficultés, de nouveaux paradigmes de programmation doivent être pensés pour apporter des abstractions de haut niveau susceptibles de cacher la complexité inhérente aux réseaux hétérogènes, et donc simplifier la conception d'applications réparties.

Le paradigme de programmation chimique [4] permet de se représenter des calculs comme des réactions chimiques apparaissant de façon autonome et parallèle parmi les molécules de données afin de produire de nouvelles molécules de résultat. Exécuter des programmes chimiques sur une plate-forme répartie revient à effectuer des réécritures sur un multi-ensemble contenant les molécules du calcul et distribué sur la plate-forme.

La construction d'un tel multi-ensemble dans une plate-forme dynamique pair-à-pair à large échelle a pour le moment seulement été initialisée, en se basant soit sur des tables de hachage distribuées [21, 19], soit sur des technologies de type *gossip* [11]. De nombreux problèmes restent cependant ouverts dans ce domaine. En particulier, la recherche d'un sous-ensemble de molécules satisfaisant la condition de réaction d'une règle est toujours basée sur une exploraation exhaustive de toutes les combinaisons possibles de molécules, ce qui est particulièrement peu efficace, surtout pour les règles prenant beaucoup d'arguments.

La question que l'on s'est posé pendant ce stage concernait la possibilité d'améliorer le temps de recherche de réactifs pour des règles compliquées. La première partie de ce rapport expose plus en détail le principe de la programmation chimique. Nous étudierons en deuxième partie comment une étude des règles sous un aspect plus théorique permet d'optimiser la recherche de réactifs. En troisième partie, nous verrons comment les leçons de l'étude théoriques peuvent servir à créer une machine chimique répartie plus efficace. Enfin, nous terminerons par un exposé du travail qu'il reste à accomplir et nous conclurons.

Chapitre 1

La programmation chimique

Le but de ce chapitre est d'expliquer les principes de la programmation chimique. Nous verrons tout d'abord les principaux formalismes et langages basés sur ce paradigme, puis nous étudierons des exemples de programmes chimiques. Dans la dernière partie, nous entrerons plus dans les détails des problèmes posés par la mise en œuvre de la programmation chimique.

1.1 Les modèles de la programmation chimique

Dans un véritable système chimique, des molécules de toutes sortes entrent en contact et réagissent pour en former de nouvelles. Les règles qui régissent ces réactions sont à l'origine de la vie et de tout ce que l'on peut trouver sur Terre.

Le paradigme de la programmation chimique est basé sur une métaphore de cet aspect particulièrement productif de la nature. Les données à traiter y sont comparées à des molécules et les algorithmes à exécuter sont représentés comme des règles de réaction sur ces molécules.

La chimie a inspiré de nombreux formalismes [9] utiles dans de nombreux domaines très différents. Nous allons maintenant brièvement présenter quelques-uns des plus importants dans le domaine de la programmation chimique.

Le modèle Γ. Un programme écrit en Γ [4, 15] est représenté par un couple (R_Γ, A_Γ) où R_Γ est la fonction booléenne des conditions de réactions, qui définit si n molécules peuvent réagir ou non, et A_Γ est la fonction d'action, qui définit les produits des réactions. Un programme peut être exécuté sur un multi-ensemble M, contenant les molécules à faire réagir.

L'exécution d'un programme s'apparente à une suite de réécritures du multi-ensemble. Tant qu'il existe des molécules dans M qui vérifient la condition de réaction, ils sont remplacés par le résultat de la fonction d'action, selon l'algorithme 1.

Le programme chimique est terminé lorsque plus aucune réaction n'est possible. On parle alors d'état d'*inertie* du système. La programmation chimique est non-déterministe, car l'ordre dans lequel deux réactions concurentes sont exécutées ne peut pas être prédit. Cela implique que l'unicité de l'état d'inertie, et donc du résultat du programme, ne peut être assurée que par le programmeur lui-même s'il souhaite que cette propriété soit vérifiée.

Algorithme 1 : L'opérateur Γ applique un programme (R_Γ, A_Γ) à un multi-ensemble M

```
1 Γ(R_Γ, A_Γ)(M) :
2 |  if ∃x_1, ··· , x_n ∈ M|R_Γ(x_1, ··· , x_n) then
3 |  |  return Γ(R_Γ, A_Γ)(M − {x_1, ··· , x_n}) ∪ A_Γ(x_1, ··· , x_n)
4 |  else
5 |  |  return M
6 end
```

Le γ-calcul. À l'instar du λ-calcul au regard de la programmation fonctionnelle, le γ-calcul [6] est un système de calcul minimal d'ordre supérieur permettant de formaliser la programmation chimique.

Un γ-terme est défini par la grammaire :

$$\begin{array}{rcll} M & ::= & x & \text{variable} \\ & | & (\gamma\,\langle x\rangle\,.M) & \gamma-\text{abstraction} \\ & | & (M_1, M_2) & \text{multi}-\text{ensemble} \\ & | & \langle M\rangle & \text{solution.} \end{array}$$

L'opérateur $\ll, \gg$, associatif et commutatif permet d'énumérer les *molécules*, d'un multi-ensemble. Certaines molécules particulières, les sous-soulutions, ont pour rôle d'isoler des molécules du reste du multi-ensemble en formant des sous-solutions. Un γ-terme peut donc être vu comme une hiérarchie de solutions contenant des molécules.

En γ-calcul, les règles de réaction sont directement intégrées dans la solution sous la forme de molécules particulières, les γ-abstractions. Il devient ainsi possible d'utiliser l'ordre supérieur, qui permet de modifier le comportement du programme au cours du temps. Par exemple, une règle qui n'est plus utilisée peut être supprimée, ou à l'inverse de nouvelles règles peuvent être introduites.

Comme dans Γ, les réactions ont lieu jusqu'à ce que l'inertie soit atteinte. Une sous-solution inerte peut alors être utilisée par une règle comme n'importe quelle molécule.

Ce formalisme est Turing-complet, puisqu'il permet d'encoder le λ-calcul. Il est même plus expressif que ce dernier car il permet d'exprimer du non-déterminisme.

Le langage HOCL. HOCL [3] (pour *Higher Order Chemical Language*), est un langage utilisable basé sur le γ-calcul, mais étendu avec des types, des valeurs de base telles que les entiers, les booléens et les chaînes de caractères, ainsi que des expressions, des couples, la solution vide et la possibilité de nommer les molécules.

Une règle de HOCL est définie par la syntaxe

$$\textbf{replace } x_1 :: T_1, \cdots x_n :: T_n \textbf{ by } P(x_1, \cdots x_n) \textbf{ if } C(x_1, \cdots x_n)$$

qui signifie que, si elles vérifient la conditon $C(x_1, \cdots x_n)$, les molécules $x_1, \cdots x_n$ de types respectifs $T_1, \cdots T_n$ doivent être supprimées du multi-ensemble et être remplacées par les valeurs renvoyées par $P(x_1, \cdots x_n)$.

L'implémentation séquentielle du langage requiert une compilation les programmes HOCL en Java avant de les exécuter. Pour cette raison, toute fonction Java est acceptée pour P et les molécules peuvent être des instances de classes Java.

1.2 Programmer en chimique

Après avoir vu les principaux formalismes et langages de programmation du paradigme chimique, nous allons illustrer l'expressivité du paradigme grâce à quelques exemples.

Un exemple simple. L'algorithme 2, qui calcule le plus grand nombre pair d'un multi-ensemble, utilise tous les principaux aspects de HOCL.

Algorithme 2 : Calcul du plus grand élément pair en HOCL

1 **let** $getEvens =$ **replace** $x :: int$ **by** $\_$ **if** $x\%2! = 0$ **in**
2 **let** $getMax =$ **replace** $x :: int, y :: int$ **by** x **if** $x \geq y$ **in**
3 **let** $replaceRule =$ **replace-one** $\langle getEvens, ?\omega \rangle$ **by** $\langle getMax, \omega \rangle$ **in**
4 **let** $clean =$ **replace-one** $\langle getMax, ?\omega \rangle$ **by** ω **in**
5 $\langle replaceRule, clean, \langle getEvens, 2, 3, 5, 6, 8, 9 \rangle \rangle$

Initialement, la sous-solution n'est pas inerte. Seule la règle $getEvens$, qui supprime tous les nombres impairs, peut réagir pour arriver à l'état $\langle replaceRule, clean, \langle getEvens, 2, 6, 8 \rangle \rangle$, dans lequel la sous-solution est inerte. La règle $replaceRule$ peut alors réagir, donnant l'état $\langle clean, \langle getMax, 2, 6, 8 \rangle \rangle$, qui réagit pour donner le résultat $\langle clean, \langle getMax, 8 \rangle \rangle$, puis $\langle 8 \rangle$, qui est bien le plus grand nombre pair parmi ceux initialement dans le multi-ensemble.

Autres exemples. L'article [5] ouvre de nombreuses pistes pour implémenter des algorithmes de traitement de chaînes de caractères ou d'images ou de recherche du plus court chemin dans un graphe dans un contexte chimique.

Par ailleurs, le paradigme chimique se montre bien adapté à la spécification des applications décentralisées. Nous avons choisi deux exemples qui montrent des domaines où la programmation chimique a permis d'apporter un nouveau regard ces dernières années.

Un système informatique est dit autonome [17] s'il est capable de s'auto-organiser en respectant des propriétés définies à l'avance. Ces systèmes s'occupent de leur propre comportement sans intervention extérieure. La principale difficulté de conception d'un tel système vient de la transformation des propriétés que l'on cherche à garantir en algorithmes réels. La programmation chimique peut être utilisée pour spécifier [5], et même implémenter [2] de tels systèmes.

L'Internet des services a pendant longtemps été considéré comme le futur d'Internet, mais l'absence de moyens simples et décentralisés pour gérer la composition de services en a empêché le grand développement prévu. Dans ce domaine, on a besoin de répondre à certains challenges injectant des propriétés d'auto-organisation, d'auto-adaptation et d'auto-réparation, voire d'auto-optimisation. Pour cela, le modèle chimique, possédant intrinsèquement ces qualités, est prometteur. En particulier, il a récement été montré [12] que le

paradigme chimique pouvait être utilisé pour coordonner les services de manière totalement décentralisée.

1.3 Les défis des machines chimiques

Le paradigme chimique offre de nouveaux horizons à l'inventivité des développeurs grâce à des concepts simples et puissants. En principe, la recherche de réactions est largement parallélisable, en raison de la propriété de localité : aucune propriété d'ensemble ne peut être demandée sur le multi-ensemble en entier dans une condition de réaction. Il reste pourtant des problèmes ouverts à résoudre pour avoir une machine chimique complète.

Exécution d'un programme chimique. L'algorithme général (algorithme 3), adapté des articles [5, 9] qui présentent une solution générale à la chimie artificielle, est composé d'une boucle de recherche de molécules du multi-ensemble P pouvant réagir ensemble. Lorsque de telles molécules sont découvertes, elles sont remplacées par le produit de leur réaction.

Algorithme 3 : Algorithme général d'exécution d'un programme chimique

```
1 while ¬Inert(P) do
2     if ∃ (r = replace x_1, ··· x_n by y_1, ··· y_m if f(x_1, ··· x_n)), x_1, ··· x_n ∈ Ps.t.f(x_1, ··· , x_n)
      then
3         P :=remove(P, x_1, ··· x_n);
4         P :=insert(P, y_1, ··· y_m);
```

Plusieurs processus qui partagent le même multi-ensemble de molécules peuvent exécuter cet algorithme en parallèle. Cela pose trois sous-problèmes.

- L'algorithme s'arrête lorsque la solution est inerte. En HOCL, cette situation d'inertie est atteinte quand toutes les conditions de réaction ne sont vérifiées par aucun tuple de molécules. Comme nous le verrons dans la partie 2.1, décider si une solution est inerte est un problème algorithmiquement difficile.
- Trouver les réactions possibles dans le multi-ensemble est le problème dual du précédent, puisque la terminaison est assurée si et seulement si aucune réaction n'est possible. Cependant, séparer les deux sous-problèmes peut être pratique dans le cas d'un calcul parallèle car il est possible qu'à un instant donné, aucune réaction ne soit possible dans un processus en raison de verrouillages de certaines molécules par d'autres processus, sans que la réaction soit terminée pour autant.
- Le dernier problème est la modification du multi-ensemble. À chaque itération de l'algorithme, des molécules sont retirées de la solution et d'autres y sont ajoutées de manière atomique.

Un algorithme de capture atomique. La capture atomique de molécules peut être vu comme un problème de synchronisation entre des processus dans lequel la propriété de vivacité est que si plusieurs processus cherchent à effectuer une réaction, au moins l'un d'eux pourra la faire. L'article [7] propose un algorithme basé sur deux protocoles. L'un, dit *optimiste*, est conçu pour minimiser l'utilisation du réseau en abandonnant une réaction dès que l'un des réactifs qu'il envisageait pour sa réaction est également prisé par un autre pair. Dans un système dans

lequel beaucoup de réactions sont possibles, la probabilité qu'un réactif soit prisé par plusieurs pairs est faible, donc un pair optimiste garde l'espoir de pouvoir exécuter des réactions. Un système de commutation permet de toujours choisir l'algorithme le plus avantageux des deux.

Recherche de molécules dans un multi-ensemble. La recherche de molécules pouvant réagir dans le multi-ensemble est très proche du problème DisCSP [8] qui est presque toujours traité par une exploration exhaustive des nœuds du réseau, avec éventuellement une optimisation grâce au back-tracking [10, 22].

Naturellement, la recherche de molécules est donc également toujours traitée par une recherche exhaustive. On peut citer par exemple un algorithme utilisant une mémoire partagée entre les cœurs de calcul [14], ou un système regroupant les molécules en *sacs* dans lesquels tous les tests ont été effectués [16].

Nous pouvons enfin présenter les travaux réalisés dans [18], qui se basent sur un réseau pair-à-pair non structuré et des protocoles épidémiques pour modéliser le mouvement brownien naturellement présent dans une solution chimique réelle. Cependant, ce travail ne cherche pas à détecter l'inertie, ce qui rend l'approche incomplète.

Ainsi, pour savoir si une solution contenant n molécules et une règle d'arité k est inerte, tous ces algorithmes testeront toutes les combinaisons de k molécules, parmi les n possibles.

Ce qu'il ressort de cette analyse est la difficulté de trouver des algorithmes efficaces pour la recherche de réactifs dans une solution chimique. [16] fait même le postulat que la complexité en $\left(\begin{array}{c} n \\ k \end{array} \right)$ qu'ils atteignent est une limite optimale. Nous nous proposons dans la suite du rapport d'étudier le bien-fondé de cette assertion.

Chapitre 2

Étude théorique

Nous venons de voir combien le pardigme chimique était prometteur en terme d'expressivité et d'adaptation aux problèmes habituels de l'algorithmique répartie. Des systèmes autonomes à la programmation de l'internet des services, la programmation chimique simplifie de nombreux problèmes apparamment très complexes. Derrière la simplicité du paradigme se cache pourtant un grand problème : comment trouver des molécules capables de réagir entre elles, ou, à l'inverse, comment montrer que plus aucune réaction n'est possible dans la solution ? À cette question, toutes les implémentations actuelles du langage se résignent à tester toutes les combinaisons imaginables de molécules, impliquant une complexité $\begin{pmatrix} n \\ k \end{pmatrix} \underset{n \gg k}{\sim} n^k$ pour n molécules et une règle d'arité k. Dans ce chapitre, nous allons chercher à comprendre ce qui rend la recherche de réactions aussi compliqué, et en déduire un nouvel algorithme plus efficace.

2.1 Formalisation de la recherche de réactions

Le problème que l'on cherche à résoudre dans cette partie est la recherche de réactifs dans une réaction chimique. Plus précisément, on se place dans un contexte plus proche de Γ pour concevoir un algorithme prenant comme entrée :

- une règle chimique $R =$ **replace** $x_1 :: T_1, \cdots, x_k :: T_k$ **by** $F(x_1, \cdots, x_k)$ **if** $C(x_1, \cdots, x_k)$ et
- un multi-ensemble M de molécules ;

et en sortie :

- $\perp$ si $M \cup \langle R \rangle$ est inerte,
- un k-uplet $(m_1, \cdots m_k)$ où $\{m_1, \cdots m_k\} \subset M$, le type de m_i est T_i pour tout i et $C(m_1, \cdots, m_k)$ est vérifiée, sinon.

On voit ici que la définition du résultat que l'on cherche à obtenir ne fait pas référence aux produits de la règle, qui n'entrent effectivement pas en compte dans la recherche de réactifs. Aussi, à partir de maintenant, nous désignerons la règle précédente par

$$R = \textbf{find } x_1 :: T_1, \cdots, x_k :: T_k \textbf{ such that } C(x_1, \cdots, x_k).$$

S'il n'y avait pas de condition de réaction, le problème serait simple à résoudre, puisqu'il suffirait de comparer le nombre de molécules disponibles pour chaque type au nombre de

molécules demandées. C'est donc sur l'étude de cette condition que va porter l'essentiel de la suite de ce rapport.

Réduction du problème aux conjonctions de littéraux. Cette condition de réaction est une formule de la logique propositionnelle, dont les littéraux sont l'application d'une fonction booléenne aux variables x_1 à x_k. La définition de ces littéraux est laissée très libre, puisque certaines implémentations du langage autorisent toutes les expressions Java, ce qui rend le problème de la recherche de réactifs en théorie indécidable. L'évaluation de code Java étant très éloigné du sujet de ce rapport, nous faisons l'hypothèse que l'évaluation des conditions de réaction termine, et nous évaluerons les complexités en fonction du nombre d'évaluations d'une partie de celle ci.

Comme toute formule propositionnelle, une condition de réaction peut être mise sous forme normale disjonctive. On peut encore simplifier la formule pour ne garder qu'un seul littéral par sous-ensemble de variables dans chaque terme en remplaçant :

- $\neg f(x_1, \cdots, x_p)$ par $(\neg f)(x_1, \cdots, x_p)$,
- $f_1(x_1, \cdots, x_p) \wedge f_2(x_1, \cdots, x_p)$ par $(f_1 \wedge f_2)(x_1, \cdots, x_p)$, ou plus généralement si l'un des deux ensembles de variables concernés est inclus dans l'autre.

En outre, le type d'une molécule peut être vu comme une condition sur cette molécule. Notre formule est alors de la forme :

$$R = \textbf{find } x_1, \cdots, x_k \textbf{ such that } \bigvee_{i=1}^{M} \bigwedge_{X \in \prod_{j=1}^{k} \{\varepsilon, x_j\}} f_{i,X}(X).$$

Dans cette écriture, les X doivent être vus comme des sous-ensemble deux à deux disjoints des variables.

Les différents termes de la disjonction peuvent faire l'objet d'une recherche séparée, puisque la règle **find** $x_1, \cdots, x_k$ **such that** $C_1(x_1, \cdots, x_k) \vee C_2(x_1, \cdots, x_k)$ peut s'appliquer à des molécules $x_1, \cdots, x_k$ si et seulement si $C_1(x_1, \cdots, x_k) \vee C_2(x_1, \cdots, x_k)$ c'est à dire si et seulement si $C_1(x_1, \cdots, x_k)$ ou $C_2(x_1, \cdots, x_k)$, soit si et seulement si l'une des règles **find** $x_1, \cdots, x_k$ **such that** $C_1(x_1, \cdots, x_k)$ ou **find** $x_1, \cdots, x_k$ **such that** $C_2(x_1, \cdots, x_k)$ peut s'appliquer. La règle R est donc équivalente à l'ensemble de règles :

$$\bigcup_{i=1}^{M} \left\{ R_i = \textbf{find } x_1, \cdots, x_k \textbf{ such that } \bigwedge_{X \in \prod_{j=1}^{k} \{\varepsilon, x_j\}} f_{i,X}(X) \right\}$$

Le nombre M de formules générées est potentiellement exponentiel par rapport à k. Cependant, en pratique :

- k est souvent de taille raisonnable, et est indépendant du multi-ensemble ;
- le passage à l'exponentielle ne concerne pas toutes les formules, mais seulement celles pour lesquelles la recherche est théoriquement difficile, comme les formules SAT ;
- la recherche peut être faite de manière complètement parallèle.

À partir de maintenant, nous pouvons restreindre l'analyse à des règles de cette forme, que nous résumerons en $R =$ **find** $x_1, \cdots, x_k$ **such that** $f_1(X_1) \wedge \cdots \wedge f_l(X_l)$.

Notations et définitions. Sur ces considérations, nous allons maintenant nous abstraire du problème. Considérons les ensembles suivants :

- $\mathcal{V}$ un ensemble dénombrable de symboles, appelés *variables* ;
- $\mathcal{M}$ un ensemble dénombrable de *molécules* ;
- $\mathcal{S}$ l'ensemble des multi-ensembles finis de molécules, appelés *solutions* ;
- $\mathcal{F} = \bigcap_{n\in\mathbb{N}}(\mathcal{M}^n \mapsto \mathbb{B})$ l'ensemble des *fonctions* booléennes sur des tuples de molécules ;
- $\mathcal{P} = \mathcal{F} \times \mathscr{P}(\mathcal{V})$ (où $\mathscr{P}$ représente l'ensemble des parties) l'ensemble des *prédicats*. Un littéral $f(x_1, \cdots, x_p)$ dans une règle peut être représenté comme un prédicat $(f, \{x_1, \cdots, x_p\})$. Étant donné un prédicat $p = (f, E) \in \mathcal{P}$, on note $\mathcal{F}(p) = f$ et $\arg(p) = E$;
- $\mathcal{A} = \mathcal{F} \times \mathscr{P}(\mathcal{V} \times \mathcal{M})$ l'ensemble des *propositions*, qui sont l'équivalent des prédicats pour les molécules. Une proposition $a = (f, \{(x_1, m_1), ..., (x_k, m_k)\}) \in \mathcal{A}$ est appelée *axiome* si $f(m_1, ..., m_k)$ est vraie. On notera $\mathcal{F}(a) = f$, $\arg(a) = \{x_1, ..., x_k\}$, $a(x_i) = m_i$ et $\mathcal{M}(a) = \{m_1, ..., m_k\}$. Par extension, si $A \subset \mathcal{A}$, on notera $\mathcal{M}(A) = \bigcup_{a\in A} \mathcal{M}(A)$;
- $\mathcal{R} = \mathscr{P}(\mathcal{V}) \times \mathscr{P}(\mathcal{P})$ l'ensemble des *règles*. Une règle

  $$\textbf{replace } x_1, \cdots, x_k \textbf{ by } Y \textbf{ if } f_1(x_{1,1} \cdots, x_{1,a_1}) \wedge ... \wedge f_l(x_{l,1} \cdots, x_{l,a_l})$$

  sera représentée dans ce formalisme par le couple

  $$\left(\{x_1, \cdots, x_k\}, \{(f_1, \{x_{1,1}, \cdots, x_{1,a_1}\}), \cdots, (f_l, \{x_{l,1}, \cdots, x_{l,a_l}\})\}\right).$$

  Pour $R = (V, P) \in \mathcal{R}$, on définit les accesseurs $\mathcal{V}(R) = V$ et $\mathcal{P}(R) = P$.

Le lecteur attentif aura remarqué l'utilisation de la lettre $\mathcal{A}$, initiale du mot *axiome*, pour désigner l'ensemble des propositions. Comme nous cherchons à vérifier les conditions de réactions, seuls les axiomes que l'on peut tirer d'une solution nous intéressent réellement.

Définition (ensemble d'axiomes induits)**.** Pour une solution $S \in \mathcal{S}$ et une règle $R \in \mathcal{R}$, on définit l'ensemble des axiomes *induits* par R dans S comme l'ensemble des axiomes dont les molécules sont dans S et satisfont un prédicat de R :

$$\mathcal{A}(R, S) = \left\{(\mathcal{F}(p), \{(x_i, m_i) : x_i \in \arg(p) \wedge m_i \in S\}) : p \in \mathcal{P}(R) \wedge \mathcal{F}(p)\left(m_1, ..., m_{|\arg(p)|}\right)\right\} \tag{2.1}$$

On peut remarquer une certaine correspondance entre le monde des règles et le monde des solutions. Les éléments de base sont les variables pour les règles et les molécules pour les solutions, et les éléments de la logique sont les prédicats dans le monde des règles et les propositions dans le monde des solutions. Pour cette raison, on donne ci-dessous deux définitions de la satisfaction d'un élément du monde des règles par un élément de celui des molécules.

Définition (satisfaction d'un prédicat)**.** Soient $p \in \mathcal{P}$ et $a \in \mathcal{A}$, on dit que a *satisfait* p, et on note $a \models p$, si les trois conditions suivantes sont réunies :
- a est un axiome,
- $\mathcal{F}(a) = \mathcal{F}(p)$,
- $\arg(a) = \arg(p)$.

Définition (satisfaction d'une variable)**.** Soient $P \subset \mathcal{P}$, $x \in \mathcal{V}$, $A \subset \mathcal{A}$ et $m \in \mathcal{M}$. On dit que m satisfait x dans A, et on écrit $(A, m) \models (P, x)$, ou simplement $m \models x$ s'il n'y a pas d'ambiguïté si

$$\forall p \in P, x \in \arg(p) \Rightarrow \exists a \in A, (x \in \arg(a) \wedge a(x) = m \wedge a \models p). \tag{2.2}$$

Moins formellement, une molécule ne satisfait pas une variable si l'on sait qu'il est impossible qu'elle soit affectée à cette variable dans une réaction, par exemple si elle n'est pas du bon type.

Étude de la complexité intrinsèque du problème. Sous l'hypothèse de terminaison de l'évaluation des conditions de réaction — en un temps alors nécessairement indépendant de n et k — le problème de la recherche de réactifs est de classe NP, puisque faire tous les tests entre des molécules de manière non-déterministe permet de le résoudre en temps polynomial. Nous allons donc étudier la NP-difficulté du problème, qui n'avait jamais été formellement établie.

Définition (rang d'une règle)**.** On appelle *rang* d'une règle R la plus grande arité des prédicats de R :

$$\mathrm{rg}(R) = \max_{p \in \mathcal{P}(R)} |\arg(p)|. \tag{2.3}$$

Pour tout n, on appelle $\mathcal{R}_n$ l'ensemble des règles de rang inférieur ou égal à n.

En programmation chimique, la grande majorité des règles que l'on peut trouver sont de rang 1 ou 2. Il est pourtant possible de créer des règles de rang supérieur, comme par exemple la règle de rang k :

$$NulSum_k = \textbf{find } x_1 \cdots x_k \textbf{ such that } \sum_{i=1}^{k} x_i = 0$$

Une autre façon de créer de telles règles est d'utiliser l'interface entre Java et HOCL pour utiliser des méthodes de Java à plusieurs arguments dans un programme chimique.

Propriété 1 (NP-difficulté par rapport au rang)**.** *Le problème de la recherche de réactifs est NP-difficile en fonction du rang de la règle.*

Démonstration. Remarquons que la recherche de n molécules dont la somme s'annule est un problème difficile. En effet, il rappelle le fameux problème NP-complet de la somme de sous-ensembles : étant donné un ensemble d'entiers E de taille n, existe-t-il un sous-ensemble de E dont la somme des éléments est nulle ? Supposons que l'on soit capable de savoir s'il existe k molécules de somme 0 dans une solution de taille n en un temps inférieur à $a.n^p$, où a est un réel positif et p est indépendant de k. Alors si l'on effectue le test pour tous les $1 \leq k \leq n$ sur l'ensemble E, on peut résoudre le problème de la somme de sous-ensembles en un temps inférieur à $a.n^{p+1}$, c'est à dire polynomial. On a donc bien réussi à ramener la recherche de réactifs pour la règle $NulSum_k$ à celui de la somme de sous-ensembles, ce qui prouve bien notre propriété. □

Il risque donc d'être difficile de trouver un algorithme plus efficace que $|S|^{\mathrm{rg}(R)}$ dans le pire cas. Nous pouvons cependant chercher à améliorer la recherche dans les cas où le rang d'une règle est inférieur à son arité.

Comme nous l'avons signalé précédement, la plupart des règles de programmes réels en HOCL appartiennent à $\mathcal{R}_2$. Cet ensemble peut être vu comme celui des graphes dont les sommets sont les variables et les arêtes sont les prédicats. Nous réutiliserons donc naturellement des termes et des notations sur les règles héritées de la théorie des graphes, que nous étendrons aux règles de rang quelconque.

Définition (sous-règle)**.** Soient $R_1, R_2 \in \mathcal{R}$. On dit que la règle R_1 est une *sous-règle* de R_2, noté $R_1 \subset R_2$, si $\mathcal{V}(R_1) \subset \mathcal{V}(R_2)$ et $\mathcal{P}(R_1) \subset \mathcal{P}(R_2)$.

Définition (connexité)**.** On dit qu'une règle R est *connexe* si

$$\forall x, y \in \mathcal{V}(R), \exists x_1, \cdots, x_n \in \mathcal{V}(R), x_1 = x \wedge x_n = y \wedge (\forall i, \exists p \in \mathcal{P}(R), \{x_i, x_{i+1}\} \subset \arg(p)). \tag{2.4}$$

Cela permet de définir les *parties connexes* d'une règle.

Définition (clique)**.** On dit qu'une règle R est une *clique* si

$$\forall x, y \in \mathcal{V}(R), \exists p \in \mathcal{P}(R), \{x, y\} \subset \arg(p). \tag{2.5}$$

Dans le cas de règles de rang 2, l'ensemble des axiomes induits peut également être vu comme un graphe dans lequel les nœuds sont les molécules, et les arêtes sont les axiomes. Trouver des réactifs revient à rechercher un sous-graphe isomorphe à celui de la règle dans le graphe des axiomes. Or ce problème est lui aussi connu comme NP-complet, parce qu'il contient le problème de la recherche de cliques.

Propriété 2 (NP-difficulté pour le rang 2)**.** *Le problème de la recherche de réactifs est NP-difficile, en fonction de son arité, pour une règle de rang 2.*

Démonstration. Supposons que l'on connaisse un algorithme polynomial en k pour la recherche de réactifs pour toute règle de rang 2.

Soit un graphe $G = (V, E)$. On veut savoir s'il existe un sous graphe de G de k nœuds qui est une clique. On crée le programme chimique dans lequel les molécules sont les nœuds de G et l'unique règle cherche k molécules toutes reliées les unes aux autres dans G :

$$\left\langle \textbf{find } x_1, \cdots, x_k \textbf{ such that } \bigwedge_{i=1}^{k-1} \bigwedge_{j=i+1}^{k} (x_i, x_j) \in E, n \in V \right\rangle$$

Selon notre hypothèse, ce programme chimique trouve une clique dans G en temps polynomial. On a bien réduit notre problème à celui de la recherche de cliques dans un graphe, ce qui montre la propriété. □

La règle utilisée pour montrer la propriété 2 est une clique, c'est à dire une règle compliquée puisque sa condition de réaction possède autant de littéraux qu'il y a de couples de variables. Il faudrait donc trouver une façon de caractériser la complexité d'une règle. C'est ce que nous faisons dans la partie suivante, qui définit la *carrure* d'une règle.

2.2 Carrure d'une règle

Nous venons d'être confrontés à deux résultats négatifs qui peuvent donner une vision pessimiste de la difficulté du problème. Pourtant, la plupart des règles chimiques ne sont ni des cliques, ni de rang très élevé. Il nous manque une notion de complexité d'une règle pour identifier ce qui peut rendre la recherche compliquée. Dans cette partie, nous introduisons la carrure d'une règle comme une mesure de cette complexité.

Définition (jointure d'une règle)**.** On considère que les variables d'une règle sont totalement triés selon une relation d'ordre $\lessdot$. On définit la *jointure* $\mathcal{J}_{\lessdot}(R)$ de R selon l'ordre $\lessdot$ comme l'ensemble des variables qui ont plusieurs voisines plus petites qu'elles :

$$\mathcal{J}_{\lessdot}(R) = \{x \in \mathcal{V}(R) : |\{p \in \mathcal{P}(R) : \exists y \lessdot x, \{x, y\} \subset \arg(p)\}| \geq 2\}. \tag{2.6}$$

Définition (carrure d'une règle)**.** La *carrure* d'une règle est la taille de sa plus petite jointure par rapport à tous les ordres possibles sur les règles.

$$\mathcal{C}(R) = \min_{\lessdot} |\mathcal{J}_{\lessdot}(R)|.$$

Une jointure $\mathcal{J}_{\lessdot}(R)$ telle que $|\mathcal{J}_{\lessdot}(R)| = \mathcal{C}(R)$ est dite *minimale*.

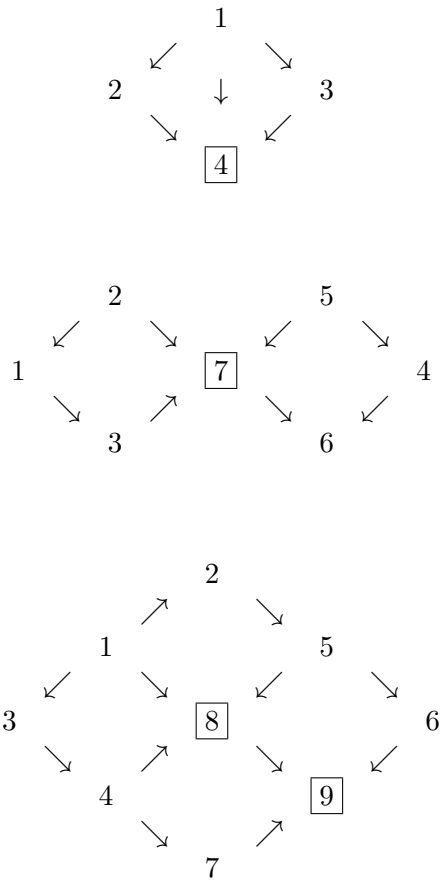

FIGURE 2.1 – le pont, le huit et le treillis avec leur jointure minimale

Remarque. Trouver une jointure minimale de façon efficace est un problème toujours ouvert. Cependant, celui-ci ne concerne que la règle elle-même, indépendamment du reste du multi-ensemble. Ce calcul peut donc être réalisé à la compilation, et sa complexité n'influence pas le temps de recherche des réactifs.

Exemples. La carrure d'un arbre est nulle. En effet, par définition, chaque élément d'un arbre possède un unique père, sauf la racine qui n'en a pas du tout. Par conséquent, en suivant l'ordre topologique, on ne trouve aucun nœud dans la jointure.

La carrure d'un cycle vaut 1 car, quel que soit l'ordre que l'on choisit, le plus grand élément a comme pères son prédécesseur et son successeur.

Comme autres exemples de graphes, on peut citer le pont et le huit qui ont une carrure de 1, et le treillis qui a une carrure de 2 (voir figure 2.1).

Propriétés de la carrure. La carrure a été introduite comme une mesure de la complexité d'une règle. Nous allons maintenant montrer quelques bornes sur la carrure.

Propriété 3 (croissance de $\mathcal{C}$)**.** *Pour toutes règles R_1 et R_2, si $R_1 \subset R_2$, alors $\mathcal{C}(R_1) \leq \mathcal{C}(R_2)$.*

Démonstration. Le principe de cette démonstration est de commencer par enlever les éléments

de $\mathcal{P}(R_2) \setminus \mathcal{P}(R_1)$ un par un, puis ceux de $\mathcal{V}(R_2) \setminus \mathcal{V}(R_1)$ pour arriver à R_1 depuis R_2. On commencera donc par une récurrence sur $|\mathcal{P}(R_2)| - |\mathcal{P}(R_1)|$, puis on montrera que l'on peut enlever simplement les variables non-contraintes.

Supposons $\mathcal{V}(R_1) = \mathcal{V}(R_2)$. Si $|\mathcal{P}(R_2)| - |\mathcal{P}(R_1)| = 0$, on a $R_1 = R_2$ donc $\mathcal{C}(R_1) = \mathcal{C}(R_2)$.

On suppose maintenant qu'il existe un $n \geq 0$ tel que pour toutes règles R_1' et R_2' vérifiant $\mathcal{V}(R_1') = \mathcal{V}(R_2') = V$, $\mathcal{P}(R_2') \subset \mathcal{P}(R_1')$ et $|\mathcal{P}(R_2')| - |\mathcal{P}(R_1')| = n$, on a $\mathcal{C}(R_1') \leq \mathcal{C}(R_2')$. Soient R_1 et R_2 vérifiant $\mathcal{P}(R_1) \subset \mathcal{P}(R_2)$ et $|\mathcal{P}(R_2)| - |\mathcal{P}(R_1)| = n + 1$. Soit $p \in \mathcal{P}(R_2) \setminus \mathcal{P}(R_1)$. On considère la règle R telle que $\mathcal{V}(R) = V$ et $\mathcal{P}(R) = \mathcal{P}(R_1) \cup \{(f, A)\}$. On a $|\mathcal{P}(R_2)| - |\mathcal{P}(R)| = n$ donc par hypothèse de récurrence, $\mathcal{C}(R) \leq \mathcal{C}(R_2)$.

Soit un ordre $\lessdot$ sur les variables tel que $\mathcal{J}_{\lessdot}(R)$ soit minimale. Pour tout $x \in V$,

$$\{p \in V : \exists y \lessdot x, \{x, y\} \subset \arg(p)\} \subset \{p \in \mathcal{P}(R) : \exists y \lessdot x, \{x, y\} \subset \arg(p)\},$$

donc $\mathcal{J}_{\lessdot}(R_1) \subset \mathcal{J}_{\lessdot}(R)$, et donc $\mathcal{C}(R_1) \leq |\mathcal{J}_{\lessdot}(R_1)| \leq |\mathcal{J}_{\lessdot}(R)| = \mathcal{C}(R) \leq \mathcal{C}(R_2)$.

D'après le principe d'induction, on peut en déduire que pour toutes règles $R_1 \subset R_2$ possédant les mêmes variables, $\mathcal{C}(R_1) \leq \mathcal{C}(R_2)$.

On considère maintenant la suppression des variables. Soient R_1 et R_2 tels que $\mathcal{P}(R_1) = \mathcal{P}(R_2) = F$ et $\mathcal{V}(R_1) \subset \mathcal{V}(R_2)$. Quel que soit l'ordre $\lessdot$ considéré, pour toute variable $x \in \mathcal{V}(R_2) \setminus \mathcal{V}(R_1)$, $|\{p \in V : \exists y \lessdot x, \{x, y\} \subset \arg(p)\}| = 0$, donc $\mathcal{J}_{\lessdot}(R_1) = \mathcal{J}_{\lessdot}(R_2)$, donc $\mathcal{C}(R_1) = \mathcal{C}(R_2)$.

Pour conclure avec des règles quelconques $R_1 \subset R_2$, on a bien

$$\mathcal{C}(R_1) = \mathcal{C}(\mathcal{V}(R_2), \mathcal{P}(R_1)) \leq \mathcal{C}(R_2).$$

□

Essayons de comparer la carrure d'une règle à son arité. Ici encore, le cas des règles de rang 2 est intéressant pour commencer.

Théorème 1 (carrure d'une règle de $\mathcal{R}_2$)**.** *Soit une R une règle de rang 2. Alors*

$$\mathcal{C}(R) + \mathrm{rg}(R) \leq |\mathcal{V}(R)|$$

et l'égalité est atteinte si et seulement si R est une clique.

Démonstration. Soient R une règle de rang 2, et $\lessdot$ un ordre sur $\mathcal{V}(R)$. Les deux plus petites variables x et y ont au plus 1 voisin supérieur. Par conséquent, ils ne font pas partie de $\mathcal{J}_{\lessdot}(R)$, donc $\mathcal{C}(R) \leq |\mathcal{V}(R)| - 2 = |\mathcal{V}(R)| - \mathrm{rg}(R)$.

Si R est une clique, toute variable z différente de x et y a comme voisin supérieur au moins x et y, donc $z \in \mathcal{J}_{\lessdot}(R)$, et $|\mathcal{J}_{\lessdot}(R)| = |\mathcal{V}(R)| - \mathrm{rg}(R)$, donc $\mathcal{C}(R) = |\mathcal{V}(R)| - \mathrm{rg}(R)$.

Si au contraire R n'est pas une clique, alors il existe x et y qui ne sont pas reliés. Considérons R' une clique dans laquelle on a supprimé le lien entre x et y. Alors R est une sous-règle de R'. Soit z une variable différente de x et y, et un ordre selon lequel les trois plus grands éléments sont $z \lessdot x \lessdot y$ dans cet ordre. Alors $\mathcal{J}_{\lessdot}(R') = k - 3$. On en déduit $\mathcal{C}(R) \leq \mathcal{C}(R') \leq k - 3$. □

Pour une règle de rang supérieur, cette propriété est fausse, et on n'a même pas $\mathcal{C}(R) \leq |\mathcal{V}(R)| - 2$. Cela est dû au fait que deux variables peuvent être reliés par plusieurs prédicats.

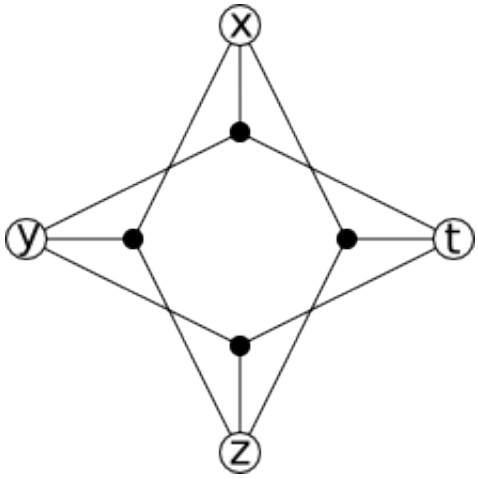

FIGURE 2.2 – Graphe d'une règle de rang 3 et de carrure 3

On peut prendre l'exemple de la règle suivante, dont le graphe est représenté sur la figure 2.2 :

$$\textbf{find } x, y, z, t \textbf{ such that } f(x, y, z) \wedge g(x, y, t) \wedge h(x, z, t) \wedge i(y, z, t).$$

On remarque que cette règle à 4 arguments est de carrure 3, ce qui entre en contradiction avec le théorème 1. Cela montre que dans le cas de règles de rang supérieur, il est nécessaire d'assouplir ce théorème en utilisant la notion d'équivalence de règles.

Définition (Équivalence de règles)**.** On définit $\equiv$ comme la plus petite relation d'équivalence sur les règles vérifiant :

$$\forall R \in \mathcal{R}, \mathcal{P}(R) = \{(f_1, E_1), (f_2, E_2)\} \cup P \Rightarrow R \equiv (\mathcal{V}(R), \{(f_1 \wedge f_2, E_1 \cup E_2)\} \cup P).$$

Autrement dit, deux règles sont équivalents si elles correspondent à la même règle du programme chimique, mais que le découpage de la condition de réaction en littéraux a été faite différement.

Théorème 2 (borne sur $\mathcal{C}$)**.** *Pour toute règle R, il existe une règle $R' \equiv R$ telle que*

$$\mathcal{C}(R') + \mathrm{rg}(R') \leq |\mathcal{V}(R')|.$$

Démonstration. Soient une règle R et $\lessdot$ un ordre sur $\mathcal{V}(R)$. Considérons $R' = \left(\mathcal{V}(R), \left\{\bigwedge_{f \in \mathcal{P}(R)} f\right\}\right) \equiv R$. Une seule fonction est présente, donc la carrure est nulle. Or $\mathrm{rg}(R') \leq |\mathcal{V}(R')|$, donc $\mathcal{C}(R') + \mathrm{rg}(R') \leq |\mathcal{V}(R')|$.

Remarque. La règle R' ici définie n'est généralement pas celle qui minimise $\mathcal{C}(R') + \mathrm{rg}(R')$. Elle est simplement donnée à titre d'illustration.

□

2.3 Affectations

Dans la partie précédente, nous avons défini une notion de complexité des règles. Pour les règles de rang 2, nous avons pu voir le lien qui unissait la carrure, les cliques et l'arité d'une règle. Il reste à présent à voir comment on peut tirer parti de cette nouvelle notion en pratique.

Définition (ensemble pur d'axiomes)**.** Soient $P \subset \mathcal{P}$ et $A \subset \mathcal{A}$. On dit que A est *pur* au regard de P si

$$\forall m \in \mathcal{M}(A), \exists x \in \mathcal{V}, (A, m) \models (P, x) \tag{2.7}$$

Définition (affectation)**.** Soient $A \subset \mathcal{A}$, $P \subset \mathcal{P}$, $x_1, \cdots, x_p \in \mathcal{V}$ et $m_1, \cdots, m_p \in \mathcal{M}$. On appelle *affectation* de $m_1, \cdots, m_p$ à $x_1, \cdots, x_p$, noté $A[x_1 := m_1, \cdots, x_p := m_p]$, la réunion de tous les sous-ensembles purs au regard de P de $\{a \in A : \forall i, x_i \in \arg(a) \Rightarrow a(x_i) = m_i\}$.

Propriété 4 (caractérisation de l'affectation)**.** *$A' = A[x_1 := m_1, \cdots, x_p := m_p]$ est le plus grand sous-ensemble pur de A, au sens de l'inclusion, vérifiant $\forall 1 \leq i \leq p, (A', m) \models (P, x_i) \Rightarrow m = m_i$.*

Démonstration. Trois choses sont à démontrer dans cette propriété :
- la pureté de l'affectation,
- la propriété $\forall i, m \models x_i \Rightarrow m = m_i$,
- et le fait que ce soit le plus grand tel ensemble.

Notons $B = \{a \in A : \forall i, x_i \in \arg(a) \Rightarrow a(x_i) = m_i\}$ l'ensemble qui a servi à définir l'affectation.

Soient A_1 et A_2 deux sous-ensembles purs de B. Soit $m \in \mathcal{M}(A_1 \cup A_2) = \mathcal{M}(A_1) \cup \mathcal{M}(A_2)$. Si $m \in \mathcal{M}(A_1)$, $\exists x \in \mathcal{V}, (A_1, m) \models (P, x)$, donc $\exists x \in \mathcal{V}, (A_1 \cup A_1, m) \models (P, x)$. On a le même résultat si $m \in \mathcal{M}(A_2)$. On en déduit que $A_1 \cup A_2$ est pur. Comme le nombre de sous-ensembles de B est fini, on peut en déduire que A est pur.

De plus, $A' \subset B$. Par conséquent, $\forall i, \forall a, x_i \in \arg(a) \Rightarrow a(x_i) = m_i$, donc $\forall i, m \models x_i \Rightarrow m = m_i$.

Enfin, soit A'' un sous-ensemble pur de A vérifiant $\forall i, m \models x_i \Rightarrow m = m_i$. Alors A'' est un sous-ensemble pur de B. Donc par définition de l'affectation, $A'' \subset A'$. □

Propriété 5 (Calcul de l'affectation)**.** *L'algorithme 4 retourne $A[x_1 := m_1, \cdots, x_n := m_n]$.*

Algorithme 4 : Calcul de $A[x_1 := m_1, \cdots, x_n := m_n]$

```
1 A' ← {a ∈ A : ∀i, x_i ∈ arg(a) ⇒ a(x_i) = m_i};
2 repeat                                   /* Computes a biggest fix point */
3 |   removed ← {a ∈ A' : ∃x ∈ arg(a), a(x) ⊭ x};
4 |   A' ← A' \ removed ;  /* Remove the points that cannot be in A[···] */
5 until removed = ∅;
6 return A'
```

Démonstration. Montrons tout d'abord qu'à chaque itération de la boucle, $A'[x_1 := m_1, \cdots, x_p := m_p]$ demeure inchangé. Soient $A_1 = A'[x_1 := m_1, \cdots, x_p := m_p]$ et $A_2 = (A' \backslash removed)[x_1 := m_1, \cdots, x_p := m_p]$. On a $A_2 \subset A_1$ par construction. Réciproquement, soit B un sous-ensemble pur de A_1. Comme B est pur, $removed \cap B = \emptyset$, donc $B \subset A_2$. Par conséquent, la réunion de tels B est identique, donc $A_1 = A_2$.

Soit B la valeur retournée par l'algorithme 4. On peut déduire du paragraphe précédent que $B[x_1 := m_1, \cdots, x_p := m_p] = A[x_1 := m_1, \cdots, x_p := m_p]$. Or, si B n'était pas pur, par définition de la pureté, $\exists m \in \mathcal{M}(B), \forall x \in \mathcal{V}, m \not\models x$. Cette molécule apparait dans une certaine propriété a avec une certaine variable x, donc $\exists a \in B, \exists x \in \arg(a), a(x) \not\models x$. Mais alors lors

de la dernière itération, ce a doit être mis dans $\mathrm{removed}$. Cela est absurde puisque $\mathrm{removed} = \emptyset$ lors de la dernière itération. Par conséquent, B est pur, donc

$$B = B[x_1 := m_1, \cdots, x_n := m_n] = A[x_1 := m_1, \cdots, x_n := m_n]$$

et l'algorithme est correct. □

Les deux propriétés suivantes, qui manipulent l'arithmétique des affectations, montrent la souplesse du calcul sur les affectations. Elles introduisent également $A[\,]$, appelé l'*épuré* de A ou l'*affectation vide*, qui est le plus grand sous-ensemble pur de A.

Propriété 6 (Séparation des variables affectées)**.** *Si X désigne $\ll x_1 := m_1, \cdots, x_x := m_x \gg$ et Y désigne $\ll y_1 := n_1, \cdots, y_y := n_y \gg$, on a $A[X, Y] = A[X][Y] = A[Y][X] = (A[X] \cap A[Y])[\,]$.*

Démonstration. Soit $A' \subset A$ pur. Les conditions suivantes sont équivalentes :

$$\begin{array}{ccc}
(m \models x_i \Rightarrow m = m_i) \wedge (n \models y_i \Rightarrow n = n_i) & s.s.i. & A' \subset A[X, Y] \\
\underset{s.s.i.}{\Updownarrow} & & \\
(A' \subset A[X]) \wedge (n \models y_i \Rightarrow n = n_i) & s.s.i. & A' \subset A[X][Y] \\
\underset{s.s.i.}{\Updownarrow} & & \\
(A' \subset A[X]) \wedge (A' \subset A[Y]) & s.s.i. & A' \subset A[X] \cap A[Y]
\end{array}$$

On en déduit que $A[X, Y][\,] = A[X][Y][\,] = (A[X] \cap A[Y])[\,]$, puis le résultat énoncé par symétrie entre X et Y, et grâce à la propriété 4. □

Propriété 7. *A et $A[\,]$ ont les mêmes affectations.*

Démonstration. Soit $X \equiv \ll x_1 := m_1, \cdots, x_n := m_n \gg$. D'après la propriété 6, on a $A[\,][X] = A[X][\,]$, et d'après la propriété 4, $A[X][\,] = A[X]$. On en déduit $A[\,][X] = A[X]$. □

Définition (affectation maximale)**.** Une affectation $A' = A[x_1 := m_1, \cdots, x_n := m_n]$ est *maximale* si $\forall x \in \mathcal{V}(R), |\{m \in \mathcal{M} : (A', m) \models (P, x)\}| \leq 1$. En particulier A' est maximale lorsqu'une molécule a été choisie pour chaque variable.

Définition (affectation valide)**.** Une affectation $A' = A[x_1 := m_1, \cdots, x_n := m_n]$ est *valide* si $\forall x \in \mathcal{V}(R), |\{m \in \mathcal{M} : (A', m) \models (P, x)\}| \geq 1$.

Propriété 8. *Si $R \in \mathcal{R}$ est connexe non vide, une affectation $A' = A[x_1 := m_1, \cdots, x_n := m_n]$ au regard de $\mathcal{P}(R)$ est valide si et seulement si $\exists x \in \mathcal{V}(R), |\{m \in \mathcal{M} : (A', m) \models (P, x)\}| \geq 1$, si et seulement si A' est non vide.*

Démonstration. Supposons que R est connexe, et soit $A' = A[x_1 := m_1, \cdots, x_n := m_n]$ une affectation.

Si A' est valide, alors $\forall x \in \mathcal{V}(R), |\{m : m \models x\}| \geq 1$ donc puisque R n'est pas vide, $\exists x \in \mathcal{V}(R), |\{m : m \models x\}| \geq 1$.

Réciproquement, supposons $\exists x \in \mathcal{V}(R), |\{m : m \models x\}| \geq 1$. Supposons par l'absurde que A' n'est pas valide, alors $\exists y \in \mathcal{V}(R), |\{m : m \models y\}| = 0$. Or R est connexe, donc il existe un chemin $x_1, \cdots, x_p$ avec $x_1 = x$ et $x_p = y$. Il existe donc un indice i pour lequel $|\{m : m \models x_i\}| > 0$ et $|\{m : m \models x_{i+1}\}| = 0$, ainsi qu'un prédicat $p \in \mathcal{P}(R)$ tel que $\{x_i, x_{i+1}\} \subset \arg(p)$. Soit $m_i \models x_i$. $(f, \{(x_i, m_i), (x_j, m_j) \cdots\}) \in \mathcal{P}(R)$, et donc $(m_j \models x_j)$. Ceci est absurde puisque $|\{m : m \models x_j\}| > 0$. Donc A' est valide. □

D'après ces deux définitions, une affectation est valide et maximale si $\forall x \in \mathcal{V}(R), |\{m \in \mathcal{M} : (A', m) \models (P, x)\}| = 1$. Cette affectation définit alors de manière immédiate une réaction possible. Inversement, une réaction définit une affectation valide maximale. Le problème de la recherche de réactifs peut donc être redéfini comme la recherche d'une affectation valide maximale dans l'ensemble des axiomes induits par une règle sur un multi-ensemble.

2.4 Présentation de l'algorithme

Nous avons maintenant tous les outils théoriques pour exposer l'algorithme de recherche de réactifs.

Théorème 3 (maximisabilité de l'affectation de la jointure)**.** *Soient R une règle d'arité k dont les variables sont ordonnées par $\lessdot$, $\{x_1, \cdots, x_c\} = \mathcal{J}_{\lessdot}(R)$ et $m_1, \cdots, m_c \in \mathcal{M}$.*

$A[x_1 := m_1, \cdots, x_c := m_c]$ est valide si et seulement si il existe $m_{c+1}, \cdots, m_k \in \mathcal{M}$ telles que $A[x_1 := m_1, \cdots, x_n := m_k]$ est valide maximale.

Démonstration. Supposons qu'il existe $m_{c+1}, \cdots, m_k$ tels que $A[x_1 := m_1, \cdots, x_k := m_k]$ est valide, alors

$$\begin{array}{rcl} A[x_1 := m_1, \cdots, x_k := m_k] & = & A[x_1 := m_1, \cdots, x_c := m_c][x_{c+1} := m_{c+1}, \cdots, x_k := m_k] \\ & \subset & A[x_1 := m_1, \cdots, x_c := m_c]. \end{array}$$

Comme $A' = A[x_1 := m_1, \cdots, x_k := m_k]$ est valide, alors $\forall x \in \mathcal{J}_{\lessdot}(R), |\{m \in A' : m \models x\}| \geq 1$, donc $A[x_1 := m_1, \cdots, x_c := m_c]$ est également valide.

Algorithme 5 : Maximisation de $A' = A[x_1 := m_1, \cdots, x_c := m_c]$ selon $\lessdot$

```
for x_i = x_1 ⋖ ··· ⋖ x_k do
    switch {p ∈ P(R) : ∃y ⋖ x_i, {x_i, y} ⊂ F(p)} do
        case ∅ : m_i ∈ {m : m ⊨ x_i};
        case {p} : m_i ∈ {m : m ⊨ x_i ∧ ∃a ⊨ p, a(x_i) = m ∧ ∀x_j ⋖ x_i, a(x_j) = m_j};
        case _ : m_i ∈ {m : m ⊨ x_i};
return A'[x_1 := m_1, ··· , x_k := m_k];
```

Réciproquement, supposons que $A' = A[x_1 := m_1, \cdots, x_c := m_c]$ est valide, alors l'algorithme 5 retourne une affectation valide maximale. En effet, l'affectation est bien maximale, il reste donc à montrer qu'elle est valide, c'est à dire que les molécules choisis par l'algorithme existent bien et qu'elles forment un graphe pur.

Étudions les trois cas de l'algorithme.

- Si x_i n'a pas de voisine plus petite qu'elle, il existe une valeur modélisant x_i car A' est valide.
- Si un unique prédicat p relie x_i à des voisines plus petites, qui ont donc été déjà choisies, il existe un axiome $a \models p$ reliant ces voisines car A' est valide. On choisit la molécule $a(x_i)$ pour l'un de ces axiomes.
- Dans le dernier cas, $x_i \in \mathcal{J}_{\lessdot}(R)$, et est donc affectée dans A'. Comme cette affectation est valide, $\{m : m \models x_i\}$ contient une et une seule valeur m_i, qui est convient bien avec tous les choix des voisines déjà choisies.

Dans les trois cas, l'existence est assurée. De plus, la pureté du graphe est vérifiée par construction. □

Nous pouvons maintenant assembler tous les éléments pour construire un algorithme complet qui résout le problème de la recherche de réactifs. L'algorithme 6 résout ce problème pour une pour une règle R dans laquelle les variables sont triées selon $\lessdot$ et un ensemble d'axiomes A.

Algorithme 6 : Recherche d'une affectation maximale valide dans R.

```
1 forall m_1 ⊨ x_1, ··· , m_c ⊨ x_c do      /* Chose values for J_≪(R) = (x_1, ··· , x_c) */
2     if valid(A[x_1 := m_1, ··· , x_c := m_c]) then      /* Check the affectation */
3         return maximize(A[x_1 := m_1, ··· , x_c := m_c])   /* there is a solution */
4 return ⊥                                                  /* there is no solution */
```

Propriété 9 (Correction). *L'algorithme 6 renvoie une affectation maximale valide si il en existe une et $\perp$ sinon.*

Démonstration. Si l'algorithme ne retourne pas $\perp$, il termine à la ligne 3. Dans ce cas, $A[x_1 := m_1, \cdots, x_c := m_c]$ est valide, donc d'après le théorème 3, il existe une jointure maximale valide, qui est celle retournée par l'algorithme.

Réciproquement, si l'algorithme retourne $\perp$, montrons par l'absurde qu'il n'existe pas d'affectation maximale valide. Supposons donc qu'il existe une affectation maximale valide $A[x_1 := m_1, \cdots, x_k := m_k]$. Les valeurs $m_1, \cdots, m_c$ sont testées par l'algorithme, mais aucune valeur n'a été renvoyée, puisque la ligne 4 a été atteinte. On en déduit que $A[x_1 := m_1, \cdots, x_c := m_c]$ n'est pas valide. Cela est absurde puisque $A[x_1 := m_1, \cdots, x_k := m_k]$ est valide. □

Propriété 10 (Complexité). *L'algorithme 6 s'execute en $\mathcal{O}\left(|\mathcal{M}(A)|^{|\mathcal{J}_{\lessdot}(R)|+\mathrm{rg}(R)}\right)$ opérations dans le pire cas.*

Démonstration. Soit $n = |\mathcal{M}(A)|$ le nombre de molécules dans A. Si la solution est inerte, ce qui correspond au pire cas, il y a exactement $\prod_{x \in \mathcal{J}_{\lessdot}(R)} |\{m \models x\}| \leq n^{|\mathcal{J}_{\lessdot}(R)|}$ exécutions de la boucle principale. Or, chaque itération nécessite le calcul d'une affectation, dont la complexité est proportionnelle par rapport au nombre d'axiomes dans A, soit inférieur à $n^{\mathrm{rg}(R)}$, puis la vérification d'une validité et le calcul d'une maximisation qui peuvent être faits en temps proportionnel au nombre de variables dans R, c'est à dire négligeable par rapport à la taille de A. On en déduit que la complexité est $\mathcal{O}\left(n^{|\mathcal{J}_{\lessdot}(R)|+\mathrm{rg}(R)}\right)$. □

Remarque. Si l'ordre $\lessdot$ définit une jointure minimale, et si on a choisi R pour que le théorème 2 soit vérifié, la complexité dans le pire cas de l'algorithme est $\mathcal{O}\left(n^{\mathcal{C}(R)+\mathrm{rg}(R)}\right)$ pour n molécules dans le multi-ensemble. D'après le théorème 2, $\mathcal{C}(R) + \mathrm{rg}(R) \leq |\mathcal{V}(R)|$, et le gain peut même être évalué à la compilation.

L'algorithme 6 permet d'améliorer la complexité de la recherche de réactifs dans la plupart des cas. Cependant, il s'agit d'un algorithme séquentiel. Dans la partie suivant, nous allons voir comment on peut utiliser ces résultats dans le cadre d'un système décentralisé.

Chapitre 3

Algorithme décentralisé

Dans le chapitre précédent, nous avons vu qu'il était possible d'améliorer la complexité de la recherche de réactifs en étudiant les conditions de réaction des règles.

On se propose de résoudre le même problème dans le cas d'un système décentralisé de type pair-à-pair dans lequel les pairs communiquent en s'envoyant des messages délivrés de façon asynchrone. Nous faisons l'hypothèse que le réseau et les pairs sont fiables, c'est à dire que, d'une part, tout message envoyé sera délivré, et d'autre part, les pairs ne peuvent pas crasher et exécutent fidèlement les algorithmes fournis.

Dans cette partie, nous ne considèrerons que les règles de rang 2. Dans ce contexte, l'ensemble des axiomes induit est un graphe dans lequel les nœuds sont les molécules et les arêtes sont les axiomes. Les pairs participant au stockage du graphe possèdent en mémoire une liste de molécules ainsi que la liste des identifiants de leurs voisines, permettant de contacter les pairs qui les hébergent. Pour augmenter la localité, des migrations de molécules sur les pairs voisins sont organisés.

Le but de ce chapitre est de montrer que la recherche de réactifs et la détection de l'inertie peuvent être résolus dans ce contexte en utilisant l'algorithme introduit dans le chapitre précédent. Le principe de cet algorithme est de rechercher des affectations valides dans l'ensemble des axiomes induits. Il est donc nécessaire de commencer par construire cet ensemble. Trois sous-problèmes doivent être résolus pour obtenir un algorithme complet.

1. La construction du graphe des axiomes induits, présentée dans la première section, requiert l'évaluation de tous les prédicats nécessaires.
2. L'algorithme 6, présenté dans le chapitre précédent, doit ensuite être exécuté sur le graphe ainsi construit. La deuxième section expliquera comment on peut faire les choix et les affectations de façon décentralisée.
3. Il est enfin nécessaire de regrouper toutes les informations nécessaires à la détection de l'inertie de la sous-solution. Cela sera abordé dans la dernière section.

3.1 Construction de l'ensemble d'axiomes induits

Le temps d'exécution de l'évaluation des propositions est imprédictible. Pour cette raison, nous prendrons le nombre d'évaluations comme une mesure de la complexité. Dans ce contexte, il convient d'effectuer tous les tests nécessaires une et une seule fois.

Principe général de la construction du graphe. Imaginons tout d'abord que l'on veuille tester deux-à-deux toutes les molécules entre elles. Une façon de s'en sortir serait d'utiliser un sac de molécules entre lesquelles tous les tests ont déjà été effectués. Ce sac est initialement vide, et lors de l'ajout d'une molécule m_1, celle ci est confrontée à toutes les molécules du sac tout en y étant insérée.

Que se passe-t-il si une deuxième molécule m_2 est insérée avant que m_1 ait terminé son travail ? Dans ce cas, le nombre de tests entre m_1 et m_2 dépend de l'ordre de parcours du sac. Si la molécule est insérée avant d'effectuer les tests, celui entre m_1 et m_2 risque d'être exécuté deux fois. Or, cette situation où de nombreuses molécules doivent être insérées en même temps correspond bien à la phase d'initialisation, puis de l'insertion des produits d'une réaction produisant plusieurs nouvelles molécules. Plus grave encore, si les molécules sont insérées après les tests, ce dernier risque de ne pas être du tout effectué, ce qui rendrait des réactions introuvables et rendrait faux l'ensemble du dispositif.

L'enseignement que nous pouvons tirer de cette réflexion est qu'une synchronisation est nécessaire lors de l'insertion d'une molécule. On peut résoudre ce problème grâce à une liste de molécules. Une molécule est insérée en début de liste et est testée avec toutes les molécules placées derrière elle dans la liste. Ainsi, deux molécules seront toujours ajoutées l'une après l'autre à l'extrémité de la chaîne, et le test entre les deux sera bien effectué une unique fois.

Ici, nous ne voulons pas tester tous les couples de molécules, mais seulement les molécules pouvant être passées en arguments d'un même prédicat. Aussi, on considère l'utilisation d'une liste différente pour chaque variable. La synchronisation doit alors être faite entre les extrémités de toutes les listes.

Une structure en arbres pour la construction du graphe. C'est ce principe que nous proposons d'adapter pour la construction répartie du graphe des molécules. Les ensembles de molécules de chaque variable sont réparties sur plusieurs pairs. L'un de ceux-ci joue le rôle de l'extrémité de la liste dans le modèle précédent. Il doit donc être facilement accessible.

Nous proposons d'organiser les pairs en arbres, un par variable, dont les racines sont les points d'insertion. Pour faciliter le raisonnement, nous considèrerons que les arbres sont binaires, bien que le nombre de fils ne soit pas vraiment important.

Plus précisément, chaque nœud de l'arbre héberge trois ensembles de molécules. Lorsqu'une nouvelle molécule est ajoutée dans la solution, elle est placée dans l'ensemble **U**, pour *unregistered*, qui regroupe les molécules qui n'ont pas encore été insérées dans l'ensemble **R**, pour *registered*, des molécules *enregistrées*. Le troisième ensemble, appelé **T**, pour *to test*, regroupe les molécules enregistrées pour lesquelles il reste des tests à effectuer.

Chemin suivi par une molécule. Lorsqu'une nouvelle molécule est ajoutée dans la solution, elle est placée dans l'ensemble **U** de n'importe quel nœud de n'importe quel arbre. Au fil du temps, elle monte dans l'arbre jusqu'à atteindre la racine. Alors, elle est transférée à la racine de l'arbre gérant la variable à laquelle elle est rattachée pour y être enregistrée dans l'ensemble **R**, ainsi qu'à l'ensemble **T** de toutes les racines correspondant à des variables pour lesquelles le test est nécessaire. La molécule dans **R** descend ensuite dans l'arbre pour laisser la place aux nouvelles inscriptions, tandis que les copies de la molécule dans **T** sont diffusées à tous les nœuds de l'arbre pour faire les tests. Il est nécessaire d'empêcher les molécules de **R** de doubler celles de **T** pour garantir l'existence et l'unicité des tests.

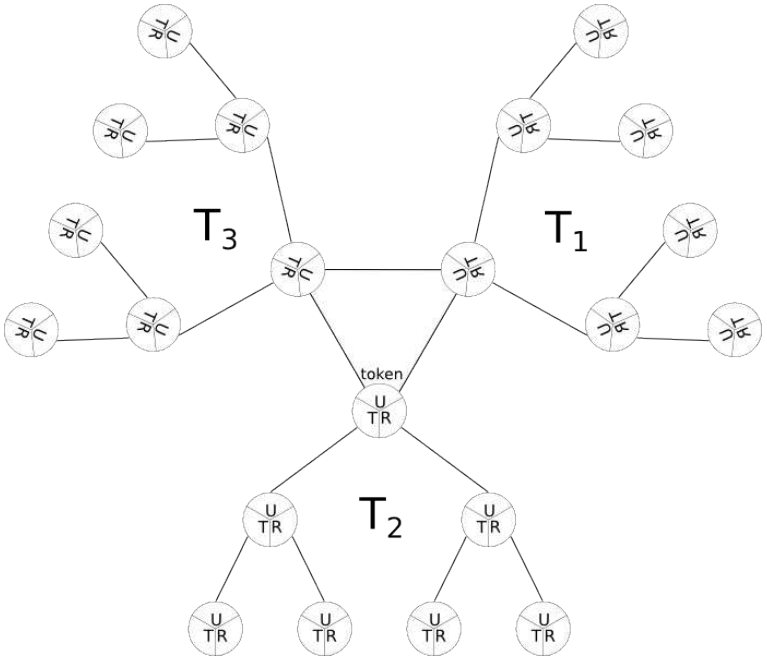

FIGURE 3.1 – L'organisation des pairs qui participent à la construction du graphe : chaque pair possède un ensemble de molécules à enregistrer, un ensemble de molécules déjà enregistrer et un ensemble de molécules à tester à transmettre aux fils. Les racines ont un rôle particulier puisqu'elles doivent initier les tests entre molécules.

L'algorithme 7 présente cette migration du point de vue d'un nœud de l'arbre. À l'instar des protocoles de *gossip,* chaque pair exécute le même algorithme périodiquement. Les racines des arbres et les autres nœuds exécutent des protocoles différents, dont nous allons à présent expliquer les grandes lignes.

Le but des racines est d'enregistrer le plus possible de molécules pour débuter les réactions. Nous venons de voir qu'il était nécessaire d'ajouter une phase de synchronisation entre les racines. Celle-ci est gérée par un jeton qui tourne entre les racines selon un ordre prédéfini. Périodiquement, donc, la racine qui possède le jeton enregistre les molécules de **U** dont le type est le même que celui de l'arbre dont elle est la racine et contacte les autres racines pour leur transmettre :

- l'ensemble des molécules qu'elles doivent elles-même enregistrer pour faire correspondre les types,
- l'ensemble des molécules qu'elle a enregistré au début de la période pour lesquelles des tests doivent être effectués
- et le jeton à la racine de l'arbre du type suivant.

De leur côté, les autres nœuds doivent permettre aux molécules pas enregistrées de rejoindre la racine, et aux autres de descendre dans l'arbre. Cela est effectué grâcé à trois messages. Premièrement, un fils contacte son père dans l'arbre, en lui envoyant des informations à propos de sa charge personnelle et d'autres utiles à la détection de l'inertie (voir section 3.3).

À la réception de ce message, le père calcule le nombre de molécules à échanger. Le choix de ce nombre peut permettre la mise en œuvre d'une politique, visant par exemple à maximiser la taille des messages où à équilibrer la charge entre les nœuds de l'arbre. Les molécules de **T** doivent être envoyées en priorité sur celles de **R** pour assurer l'unicité des tests. Dans

l'algorithme 7 à la ligne 29, on utilise la fonction $T.get\_once\_for\_each\_son(n, son)$ qui retourne $min(n, |T|)$ molécules de **T**, tout en assurant que chaque molécule soit retournée une et une seule fois pour chaque fils. Lorsqu'une molécule a été retournée pour tous les fils, elle est supprimée de l'ensemble. À la ligne 30, la fonction $R.get\_and\_remove(n, son)$ retourne un ensemble de même taille, mais supprime les molécules qui ont été retournées, indépendamment du fils qui a effectué la requête. Ainsi, la réponse du père à son fils comprend un ensemble de molécules à tester, un ensemble de molécules enregistrées et le nombre de molécules à envoyer en retour.

À son tour, le fils reçoit le message et fait les tests entre les nouvelles molécules à tester et les molécules enregistrées dont il avait déjà la charge. Les tests entre les molécules envoyées par le père ont déjà été effectués, et ils ne doivent donc pas être refaits. En retour, le fils envoie le nombre de molécules pas enregistrées demandé.

Lorsqu'un nouvel axiome est découvert, la version enregistrée des deux molécules doit être notifiée. Il est donc nécessaire de rechercher la molécule dans l'arbre. Comme nous allons le voir à présent, la structure d'arbres que nous venons de définir est particulièrement bien adaptée à cette recherche.

Recherche d'une molécule dans l'arbre. La molécule est toujours enregistrée à la racine, mais elle migre ensuite dans l'arbre. Il ne suffit donc pas d'une adresse pour la retrouver. Nous proposons donc de nous servir de la structure précédement décrite comme d'une table de hashage distribuée. Au moment de leur création, les molécules reçoivent un identifiant unique formé de l'identifiant de leur créateur et d'un numéro unique au créateur qui servent à identifier la molécule, ainsi qu'un nombre réel tiré aléatoirement selon une loi uniforme entre 0 et 1 qui permet de la retrouver dans l'arbre.

À chaque nœud d'un arbre est associé un intervalle de $[0, 1]$ tel que :
- l'intervalle de la racine est $[0, 1]$,
- les intervalles de deux fils d'un même nœud sont disjoints,
- la réunion des intervalles des fils d'un nœud est égale à l'intervalle du nœud.

Une valeur typique pour ces intervalles dans un arbre binaire est $[0, 1]$ pour la racine, $[0; 0, 5]$ et $]0, 5; 1]$ pour ses fils, les quatre intervalles de longueur $\frac{1}{4}$ pour ses petits fils, et ainsi de suite.

Nous voulons faire en sorte que toutes les molécules de **R** présentes sur un nœud aient la partie aléatoire de leur identifiant comprise dans l'intervalle du nœud. En conséquence, cela est également vrai pour toutes les molécules du sous arbre dont le nœud est la racine. Pour obtenir ce résultat, il suffit de n'envoyer à chaque fils que les molécules dont la partie aléatoire correspond.

Pour trouver une molécule dans l'arbre, il suffit donc d'envoyer un message à la racine, qui la recherche parmi les molécules qu'elle héberge, et transmet le message au fils dont l'intervalle correspond si elle ne la trouve pas. Celui-ci répète l'opération, et ainsi de suite jusqu'à ce que la molécule soit trouvée. La complexité dans le pire cas est donc égale à la hauteur de l'arbre en terme de messages échangés. Une amélioration consiste à répondre en communiquant l'adresse trouvée au pair qui a initié la requête. Ainsi, lors de la demande suivante, le haut de l'arbre n'aura pas besoin d'être exploré. Si h est la hauteur de l'arbre, la complexité pour n requêtes par le même pair est donc réduite à $1 + \frac{h}{n} \underset{n\to\infty}{\longrightarrow} 1$ message.

Cependant, l'algorithme tel qu'il vient d'être introduit n'est pas suffisant. En effet, il est possible que la recherche d'une molécule soit effectuée en parallèle de la migration de cette

Algorithme 7 : Algorithme simplifié de la construction du graphe

```
1  peer : father;
2  peer[ ] : sons;
3  set⟨molecule⟩ : R, U, T;
4  boolean : is_root, has_token;
5  map⟨type, peer⟩ : roots;
6  type : type;
7  periodically :
8      if is_root then
9          if has_token then
10             forall type t do
11                 set⟨molecule⟩ : good_type = {m ∈ U : m ⊨ t};
12                 set⟨molecule⟩ : need_test = {m ∈ U : m ⊨ type ∧ type ↔ t};
13                 send contact_roots(good_type, need_test, t = type+1) to roots(t);
14             R ← R ∪ {m ∈ U : m ⊨ type};
15             U ← ∅;
16     else
17         send contact_father(R.size(), U.size(), T.size()) to father;
18 end
19 receiving contact_roots(set⟨molecule⟩ good_t, set⟨molecule⟩ test, boolean token) :
20     has_token ← token;
21     U ← U ∪ good_t;
22     T ← T ∪ test;
23     forall Molecule m ∈ test do
24         forall Molecule mr ∈ R do
25             DoTest(m, mr);
26 end
27 receiving contact_father(int sizeU, int sizeR, int sizeT) from son :
28     int×int : (send, send_back) = f(sizeU, sizeR, sizeT, U.size, R.size, T.size);
29     set⟨molecule⟩ : sendT = T.get_once_for_each_son(send, son);
30     set⟨molecule⟩ : sendR = R.get_and_remove(send − sendT.size, son);
31     send answer_son(sendT, sendR, send_back) to son;
32 end
33 receiving answer_son(set⟨molecule⟩ sendT, set⟨molecule⟩ sendR, int send_back) :
34     T ← T ∪ sendT;
35     forall Molecule m ∈ sendT do
36         forall Molecule mr ∈ R do
37             DoTest(m, mr);
38     R ← R ∪ sendR;
39     send ack_father(U.get_and_remove(sendBack)) to father;
40 end
41 receiving ack_father(set⟨molecule⟩ sendU) :
42     U ← U ∪ sendT;
43 end
```

même molécule. Il est alors possible, à cause de l'asynchronie que le message de recherche arrive avant celui de migration. Dans ce cas, le fils ne trouvera pas la molécule, transmettra la requête à son propre fils, et la molécule ne sera jamais retrouvée. Pour pallier ce problème, on apporte une modification à l'algorithme d'échange de molécules. Dorénavant, les identifiants des molécules R sont conservés par le père jusqu'à la réception de la réponse du fils. Pendant ce temps là, les messages à destination de la molécule transmise sont conservés par le père. Les messages doivent être numérotés pour pouvoir retrouver de quelles molécules le message est l'accusé de réception.

La figure 3.2 montre un exemple d'exécution de l'algorithme qui récapitule plusieurs points énoncés ci-dessus. Sur cette figure, quatre pairs sont représentés. Un pair, son fils et son petit-fils font partie de l'arbre. Le quatrième pair cherche à retrouver des molécules dans l'arbre. Par exemple, on peut imaginer qu'il fait partie d'un autre arbre qui découvre de nouveaux axiomes dans A. Il cherche donc à notifier sa présence à la molécule rouge, puis à la molécule bleue.

Ces deux molécules sont initialement hébergées par le père, mais les fils initient des protocoles d'échange de molécules. Sur la figure, seules les molécules de **R** sont représentées. Lorsque le père reçoit la requête pour la molécule rouge (en pointillés), il vient d'envoyer cette dernière à son fils et est en attente de réponse. Un peu plus tard, le même scénario se produit pour la molécule bleue. Par un hasard du réseau, le père recevra l'accusé de réception de la molécule bleue avant celui de la molécule rouge. Il fait donc suivre les requêtes dans l'ordre inverse de l'ordre de réception. Cependant, lorsque le fils reçoit la requête pour la molécule bleue, il a déjà envoyé cette dernière à son propre fils et reçu l'accusé de réception. Il peut donc transmettre la requête sans attendre au petit-fils. Finalement, les deux molécules sont retrouvées, et l'auteur des deux demandes est averti du déplacement.

Les listes de voisins sont hébergées par des pairs différents que ceux qui participent à la création du graphe. Il est donc nécessaire de conserver des liens de la molécule de **R** vers sa liste de voisins. Pour cela, un système semblable de boîtes aux lettres est également utilisé.

Dans cette partie, nous avons montré que nous étions capables de construire le graphe de manière décentralisée en utilisant une structure à base d'arbres. Ces arbres nous offrent trois avantages. Premièrement, on peut les utiliser comme des arbres de diffusion pour paralléliser les tests de propositions atomiques. Deuxièmement, en les utilisant comme des arbres de recherche, on peut retrouver efficacement des molécules dans le réseau. Enfin, la recherche de la racine, nécessaire pour l'insertion de molécules, peut être faite efficacement également. De plus, la surcharge de cette dernière est maitrisée par le mécanisme de contacts réguliers, dans lequel le père choisit la taille des messages échangés. Maintenant que nous avons le graphe des molécules, nous pouvons nous pencher sur la répartition de l'algorithme de recherche de réactions introduit dans la partie 2.4.

3.2 Recherche de réactifs

La recherche de réactifs suit le schéma suggéré par l'algorithme 6. Il faut et il suffit de vérifier la vacuité de toutes les parties connexes de toutes les affectations d'une jointure. La première étape est donc de générer tous les tuples de molécules définissant une affectation, la deuxième est d'épurer le graphe selon chacun de ces tuples et la troisième est de vérifier la vacuité, conformément à la propriété 8, c'est à dire de vérifier que l'une des molécules du tuple n'a pas été éliminée.

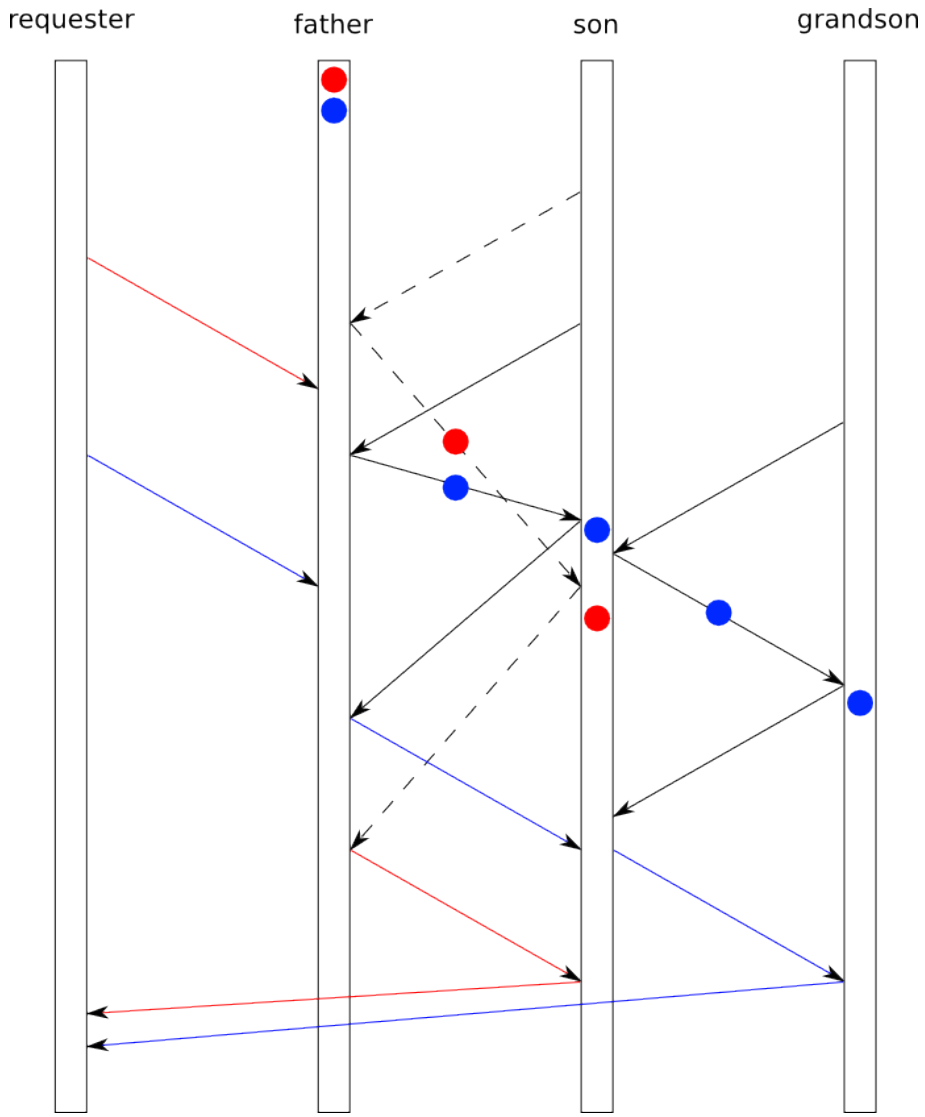

FIGURE 3.2 – Diagramme de séquence de la recherche de molécules dans l'arbre. Quatre pairs sont représentés sur cette figure. D'une part, trois nœuds de l'un des arbres, un père, son fils et son petit-fils sont à droite de la figure. D'autre part, un pair cherche à contacter les molécules rouge et bleue initialement hébergées par le père. Ces molécules migrent vers le fils, puis vers le petit-fils pour l'une d'elle, ce qui oblige aux parents de faire suivre les requêtes.

N'oublions pas que l'un des buts de la décentralisation de l'algorithme est de pouvoir tirer partie d'une plus grande capacité de calcul. Pour l'exploiter, il est nécessaire de paralléliser la recherche au maximum. Or, justement, les calculs effectués pour les différentes jointures sont indépendants. La parallélisation peut donc être réalisée sur les choix.

Algorithme 8 : Recherche de réactifs pour une règle de jointure $[x_1, \cdots, x_n]$

```
1  explore(A, [x_1, ··· , x_n]) :
2      if n=0 then
3          compute A[ ];
4          if A[ ] ≠ ∅ then
5              find_reaction(A[ ]);
6      else
7          forall m ⊨ x_1 do
8              if n = 1 then
9                  compute A[x_1 ← m];
10                 if A[x_1 ← m] ≠ ∅ then
11                     find_reaction(A[x_1 ← m]);
12             else
13                 compute_and_clone A[x_1 ← m];
14                 explore(A[x_1 ← m], [x_2, ··· , x_n]);
15 end
```

L'algorithme 8 montre les grandes lignes de l'algorithmes que nous proposons. Il traite différement le cas où la carrure est nulle et les autres cas.

Dans ce cas, tous les pairs doivent participer conjointement à l'opération selon l'algorithme 9. Le principe est le suivant : quand une molécule s'aperçoit qu'il lui manque un voisin, elle passe son état de *vivante* à *épurée* et notifie ses voisins. Un jour, toutes les molécules qui ne sont pas dans le graphe épuré seront marquées comme *épurée*. Pendant cette opération, on cherche des réactifs sans tenir compte du fait que le graphe n'est pas encore nécessairement épuré, c'est à dire en suivant l'algorithme 5. Si une réaction est repérée, elle est effectuée, si on tombe sur une impossibilité, on abandonne les molécules déjà trouvées pour tester une autre réaction. L'algorithme ne s'arrête que quand toutes les molécules sont marquées comme *épurées*, signe que plus aucune réaction n'est possible.

Si la carrure n'est pas nulle, l'algorithme est récursif sur le nombre de variables de la jointure qui n'ont pas encore été affectées. À chaque étape, la plus grande variable restante x dans la jointure est choisie, et pour toute molécule $m \models x$, l'affectation $A[x \leftarrow m]$ est calculée. Si x est la dernière variable dans la jointure, on sait que l'on peut trouver des réactifs si et seulement si l'affectation n'est pas vide. Sinon, on copie l'affectation sur de nouveaux pairs et on recommence avec le reste de la jointure.

Le calcul de l'affectation $A[x \leftarrow m]$ suit le même principe que l'épuration pour la carrure nulle, mais il est démarré et terminé sur le nœud qui héberge m. Toutes les molécules correspondant à une variable x initient des protocoles précédent, donc tous les messages et toutes les marques doivent être estampillés par l'identifiant de l'auteur de la requête. Pendant toute la procédure, chaque molécule possède un état qui varie en suivant l'automate présenté sur la figure 3.3. Elle est terminée quand l'initiateur de la requête a choisi son état entre *alive* et *refi-*

Algorithme 9 : Épuration décentralisée du graphe des molécules

```
1  void refine() :
2      forall molecule m : is_refined(m) do
3          m.mark ← "refined";
4          forall molecule n ∈ m.neighbors do
5              send refined(m, n) to n.owner;
6  end
7  boolean is_refined(m) :
8      return ∃x ∈ m.variable.neighbors : m.neighbors.getsize(x, "refined") = 0;
9  end
10 receiving refined(molecule m, molecule n) :
11     n.neighbors.get(m).mark ← "refined";
12     if n.mark ≠ "refined" ∧ is_refined(n) then
13         n.mark ← "refined";
14         forall molecule n' ∈ n.neighbors do
15             send refined(n, n') to n'.owner;
16 end
```

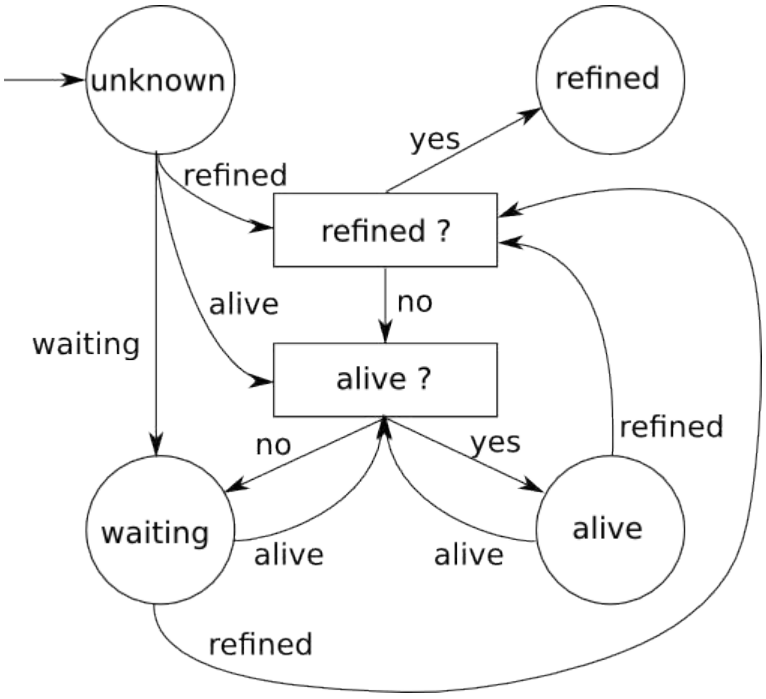

FIGURE 3.3 – Pendant le calcul d'une affectation, une molécule peut être dans l'un des quatre états *unknown*(l'état initial), *waiting*, *alive* ou *refined*. Quand une molécule change d'état, elle notifie ses voisins dans le graphe. La molécule notifiée réagit selon les règles présentées dans cet automate en tenant compte de son propre état, du message reçu, et de ce qu'elle sait déjà du statut de ses voisines. *refined ?* est faux si elle a au moins un voisin *waiting* ou *alive* pour chaque variable, et *alive ?* est vrai si tous ses voisins sont dans l'état *alive* ou *refined*, ou si elle a au moins un voisin *alive* pour chaque variable.

ned. À ce moment là, il transmet un message à tout le graphe épuré, qui est connexe, soit pour initier la copie, soit pour choisir les réactifs si tous les choix ont été effectués dans la jointure.

Il peut ne pas sembler évident de prime abord que cet algorithme termine, c'est à dire qu'en un temps fini, toutes les molécules aient choisi l'état *alive* ou *refined*. En effet, on pourrait penser qu'un inter-blocage est possible avec plusieurs molécules attendant des réponses mutuellement les unes des autre. Nous allons montrer que cette situation est impossible.

Imaginons que les messages aient été estampillées par des horloges de Lamport, alors l'estampille du premier message reçu par chaque molécule permet de définir un ordre partiel sur les molécules (atteinte par l'algorithme, donc dont l'état n'est pas *unknown*) dont le plus petit élément est l'initiateur de la requête. Complétons cet ordre partiel en un ordre total, et supposons qu'il existe une molécule bloquée dans l'état *waiting*. Considérons la plus grande selon notre ordre total. Ses voisines qui lui sont supérieures sont dans l'état *alive* ou *refined*, et lui ont donc envoyé un message en changeant d'état. Celles qui lui sont inférieures ont reçu un message avant elle, et lui ont donc envoyé un message. Toutes les voisines de notre molécule lui ont donc envoyé un message. Elle finira donc par recevoir ces messages, et donc par faire son choix, ce qui contredit notre hypothèse et montre bien que l'inter-blocage est impossible.

Étudions maintenant la complexité de la recherche de molécules. En terme de messages, l'épuration en utilise un nombre proportionnel au nombre d'arêtes dans le graphe, et est effectuée autant de fois qu'il y a d'affectations à tester. La complexité en messages est donc $\mathcal{O}(n^{2+\mathcal{C}(R)})$. La constante est en revanche d'autant plus petite que le nombre de molécules hébergées par nœud est élevé, puisque l'on cherche à regrouper des molécules voisines sur les mêmes pairs. Par ailleurs, le pire cas suggère que le graphe des molécules est composé d'une seule composante connexe.

La complexité en ressources de calcul utilisées est elle proportionnelle au nombre de molécules pour chaque copie du graphe. Or il est fait une copie pour chaque molécule pouvant être assignée à chaque variable de la jointure, sauf la dernière. La complexité en ressources de calcul est donc $\mathcal{O}(n + n^{\mathcal{C}(R)})$.

Enfin, calculons la complexité en temps. La complexité d'une épuration est toujours proportionnelle au nombre d'arêtes dans le graphe. Ici, ce n'est pas le nombre d'épurations qui est important, puisqu'elles sont faites en parallèle. Le point important est le nombre maximal d'épurations causalement dépendantes entre le début et la fin de l'algorithme, qui est égal à la carrure de la règle. Cela nous donne donc dans le pire cas une complexité en $\mathcal{O}(\mathcal{C}(R).n^2)$, c'est à dire polynomiale par rapport à la fois à la complexité de la règle et au nombre de molécules. Nous avons donc atteint l'objectif de paralléliser au maximum la recherche de molécules.

3.3 Détection de l'inertie

Le dernier point à traiter est naturellement la détection de l'inertie. Celle-ci est atteinte quand le graphe a été complètement construit et toutes les affectations d'une jointure on été testées. Le premier point se résume aux conditions suivantes :

- toutes les molécules ont été proposées, ou dit autrement, il n'y a aucune réaction en cours qui serait susceptible de produire de nouvelles molécules. Pour s'assurer de cela, on marque les réactifs quand on débute une réaction et on ne les supprime qu'après l'insertion des produits. Cette condition peut donc s'énoncer : $\forall m \in R, \neg m.reacting$;
- toutes les molécules ont été enregistrées : $U = \emptyset$;

– tous les tests ont été effectués : $T = \emptyset$.

Comme dans la recherche de réactifs, la détection de l'inertie dépends de la carrure de la règle.

Si la carrure est nulle, on a vu plus tôt que l'inertie est atteinte quand toutes les molécules ont été marquées *refined*.

Sinon, la plus grande variable dans la jointure selon $\lessdot$ joue une rôle particulier. Il faut en premier lieu que chacune des molécule m qui puisse y être affectée ait terminé son instance de l'algorithme sans avoir trouvé de réaction possible. Il est possible que des réactifs aient été trouvés mais que la réaction n'ait pas pu avoir lieu à cause d'un conflit entre deux réactions cherchant à utiliser la même molécule. Dans ce cas l'algorithme devra être redémarré avant de décider que la solution est inerte. Le protocole pessimiste introduit dans [7], qui assure qu'au moins une réaction aura toujours lieu si cela se produit, permet de certifier que cette situation ne se produira qu'un nombre fini de fois si le programme chimique termine.

L'autre propriété plus délicate à garantir est le fait que ces tests ont bien été débutés sur une version complète du graphe. Pour s'en assurer, on marque les molécules accessibles depuis m lors de la phase d'épuration et de copie, et une exploration du graphe suffit à vérifier que toutes les molécules ont bien été marquées.

Toutes ces propriétés se réduisent à un marquage des molécules, qui peut être fait dans l'arbre. Lors des échanges père-fils, toutes les informations sur la vacuité de **T** et **U** et sur les marques des molécules de **R** sont communiquées au père. Quand toutes les racines ont reçu suffisement d'information, l'inertie peut être décidée.

Travaux futurs et conclusion

Pour aller plus loin

Les algorithmes décrits dans ce rapport offrent une vue incomplète de la recherche de réactifs dans une solution, de l'insertion des molécules jusqu'à la détection de l'inertie. De nombreux points restent à préciser pour avoir une machine chimique pair-à-pair complète. Ceux-ci peuvent concerner les hypothèses que nous avons faites sur les variables, les travaux en amont et en aval du présent travail, ou encore des optimisations. Cette partie va maintenant présenter certaines pistes pouvant être creusées pour les continuer ces travaux.

Compilation. Jusqu'à présent, nous ne nous sommes intéressés qu'à la recherche effective des molécules à partir du graphe de la règle. Pour obtenir celui-ci, nous avons utilisé des arguments de calculabilité pour exprimer que la compilation était effectivement possible. Il serait intéressant de rechercher des algorithmes efficaces, par exemple, pour choisir l'ordre des variables dans la règle.

Ce choix peut même être laissé plus souple que ce qui est exprimé dans la partie 2.4. Le fait de choisir un ordre dont la jointure est minimale permet de faire le choix à la compilation tout en apportant des garanties sur la complexité de l'algorithme. Pourtant les algorithmes sont plus souples et fonctionne quelle que soit la jointure. La complexité peut donc être encore réduite en faisant un choix plus fin de l'ordre sur les variables.

Optimisations selon les règles. On pourrait aller encore plus loin dans l'analyse des conditions de réaction. Par exemple, beaucoup de littéraux sont de la forme $f(x) \circ g(y)$ où x et y sont des variables, f et g sont des fonctions avec le même ensemble d'arrivée E et $\circ$ est une relation d'ordre ou d'équivalence sur les éléments de E. Dans ce cas, la construction du graphe peut être améliorée, par exemple grâce à un tri sur les valeurs de $f(m)$ et $g(n)$ pour les molécules $m \models x$ et $n \models y$. Cela permettrait en outre une représentation condensée du graphe, et donc limiterait la complexité spatiale.

Pendant tout le rapport, nous avons considéré que chaque molécule ne pouvait être affectée qu'à une seule variable. Il est toujours possible de se ramener à ce cas en imaginant qu'il existe une copie de chaque molécule par variable à laquelle elle peut être affectée, et un lien entre chaque paire de ces variables, disont x et y, signifiant $\ll x \neq y \gg$. Cependant, ce lien est très particulier, puisqu'on sait que pour chaque molécule m pouvant être affectée à x, il existe une unique molécule n pouvant être affectée à y ne vérifiant pas $m \neq n$. En conséquence, de nouvelles améliorations à ce cas particulier devraient être recherchées, la première d'entre elles étant, comme dans le cas précédent, l'inutilité de la recherche des voisins dans le graphe.

Relâchement des hypothèses sur les règles. Les composantes connexes d'une règle R peuvent être traitées indépendamment. Nous avons pour l'instant uniquement traité les graphes connexes, souvent considérés comme les cas les plus délicats. Pourtant, la carrure d'un graphe non-connexe est la somme des carrures de ses composants, mais en parallélisant la recherche des différentes composantes connexes, on peut réduire la complexité à une fonction du maximum de ces carrures. Dans ce cas, l'inertie est atteinte quand il existe une composante connexe pour laquelle aucune réaction n'existe.

La théorie s'applique à toutes les règles quel que soit leur rang. Pourtant, dans la version décentralisée de l'algorithme, nous n'avons considéré que le cas de $\mathcal{R}_2$. Pour étendre l'algorithme, une piste serait d'explorer l'arbre plusieurs fois : lors des premiers parcours, on crée les couples, puis les triplets... qui sont remis dans l'espace **U** des molécules pas enregistrées. Le test n'a lieu que quand il est effectivement faisable. Le défaut de cette méthode est qu'il fait exploser la complexité spatiale de l'algorithme. L'exploration du graphe et la détection de l'inertie peuvent alors être faites exactement comme dans le cas du rang 2.

Structuration du réseau. Jusqu'à présent, nous avons considéré que la structure en arbre était donnée, et qu'il était toujours possible de trouver de nouveaux pairs dans le réseau pour paralléliser les calculs. L'obtention d'une telle topologie à partir d'un réseau complètement non structuré n'a pas encore été explorée.

Dans le même ordre d'idée, l'algorithme proposé ne tolère pas les fautes. Un système de réplication de données devrait être mis être mis en place pour assurer, d'une part, de ne pas perdre de molécules dans la solution, et d'autre part de conserver les propriétés sur la complexité de l'algorithme.

On peut encore mentionner l'équilibrage de la charge de travail entre les pairs. Notre technique d'échange entre les pères et les fils garantit, que tous les pères travaillent autant, et donc qu'il n'y a pas de surcharge, par exemple pour les racines des arbres. Cependant, une surcharge qui n'a pas été évaluée dans le cas moyen peut être créée par le système de recherche d'une molécule. Celle-ci démarre en effet toujours de la racine lors de la première recherche. Pour une molécules donnée, le cas le pire est donc qu'il y ait le maximum de recherches complètement indépendantes. Cela se produit si aucun lien n'est découvert par les nœuds internet des arbres (pour l'indépendance), mais un lien est découvert par toutes les feuilles de ces arbres (pour la maximisation). Dans ce cas, le nombre de messages qui transitent par la racine est proportionnelle à la taille de l'arbre, c'est à dire au nombre de molécules si l'on arrive à garder une quantité bornée et à peu près constante de molécules par nœud.

Aspects avancés du langage. Le langage HOCL possède des aspects plus avancés que ceux étudiés ici. Le premier d'entre eux est la possibilité d'avoir plusieurs règles dans la même solution. Dans ce cas, il est possible de faire les recherches complètement en parallèle. En revanche, dans ce cas, l'unicité des molécules dans **R** n'est plus garantie. En outre, une partie des calculs, par exemple l'évaluation des littéraux, peut parfois être mutualisée. Il serait donc intéressant d'étudier comment la structure proposée peut être adaptée à plusieurs règles.

En raison de l'ordre supérieur, il est également possible qu'une règle soit supprimée alors que la recherche de réactifs pour cette même règle est en cours. Si des réactifs sont trouvés, il faut alors vérifier si la règle est présente avant de faire la réaction.

Enfin, le dernier aspect du langage à traiter est la possibilité d'utiliser des sous-solutions,

bien qu'il ne semble pas impliquer de nouveaux problèmes algorithmiques vraiment difficiles à première vue.

Conclusion

L'exécution d'un programme chimique nécessite la recherche de molécules vérifiant une condition de réaction. Trouver des réactifs de manière efficace est un problème algorithmiquement difficile, à tel point que les précédentes études partaient du principe qu'il était impossible de le résoudre en une complexité inférieure à $\mathcal{O}(n^k)$, pour une solution de n molécules et une règle ayant k arguments.

Dans ce rapport, nous avons montré qu'en étudiant les conditions de réaction, ce problème pouvait être résolu en un temps dominé par $n^{\mathrm{rg}(R)+\mathcal{C}(R)}$, avec $\mathrm{rg}(R)+\mathcal{C}(R) \leq k$. Pour les règles de rang 2, nous avons réussi à caractériser le cas d'égalité. Nous avons ensuite montré qu'il était possible, dans un système pair-à-pair sûr, de paralléliser cet algorithme suffisament pour garantir un temps de recherche polynomial si le nombre de pairs est suffisant.

Finalement, si de nombreux points restent à améliorer avant d'avoir une machine chimique complètement efficace, ce travail a permis d'améliorer les techniques de recherche existantes, tout en montrant les potentialités de l'étude des règles de réaction pour améliorer l'implémentation du langage.

Remerciements

Il y a beaucoup de personnes que j'aimerais remercier, en commençant bien sûr par mes deux encadrants, Cédric Tedeschi et Marin Bertier, toujours très disponibles pour m'aider dans mon travail.

Je souhaite également remercier l'ensemble de l'équipe Myriads. Je veux porter un attention toute particulière pour Yann Radenac pour l'aide qu'il m'a apporté lors de la prise en main du langage HOCL, et pour Stefania Costache et Nikos Parlavantzas qui ont partagé leur bureau avec moi ces derniers mois et avec qui j'ai passé de bons moments.

Bibliographie

[1] Ozalp Babaoglu, Geoffrey Canright, Andreas Deutsch, Gianni A. Di Caro, Frederick Ducatelle, Luca M. Gambardella, Niloy Ganguly, Márk Jelasity, Roberto Montemanni, Alberto Montresor, and Tore Urnes. Design patterns from biology for distributed computing. *ACM Trans. Auton. Adapt. Syst.*, 1(1) :26–66, September 2006.

[2] J.-P. Banâtre, P. Fradet, and Y. Radenac. Chemical Specification of Autonomic Systems. In *13th International Conference on Intelligent and Adaptive Systems and Software Engineering*, Nice, July 2004.

[3] J.-P. Banâtre, P. Fradet, and Y. Radenac. Generalised multisets for chemical programming. *Mathematical. Structures in Comp. Sci.*, 16 :557–580, August 2006.

[4] J.-P. Banâtre, A. Coutant, and D. Le Metayer. A parallel machine for multiset transformation and its programming style. *Future Generation Computer Systems*, 4(2) :133 – 144, 1988.

[5] J.-P. Banâtre, P. Fradet, and D. Le Métayer. Gamma and the Chemical Reaction Model : Fifteen Years After. In Cristian Calude, Gheorghe PAun, Grzegorz Rozenberg, and Arto Salomaa, editors, *Multiset Processing*, volume 2235 of *Lecture Notes in Computer Science*, pages 17–44. Springer Berlin / Heidelberg, 2001.

[6] J.-P. Banâtre, P. Fradet, and Y. Radenac. Principles of Chemical Programming. *Electronic Notes in Theoretical Computer Science*, 124(1) :133 – 147, 2005. Proceedings of the 5th International Workshop on Rule-Based Programming (RULE 2004) Rule-Based Programming 2004.

[7] M. Bertier, M. Obrovac, and C. Tedeschi. A Protocol for the Atomic Capture of Multiple Molecules on Large Scale Platforms. In Luciano Bononi, Ajoy Datta, Stéphane Devismes, and Archan Misra, editors, *Distributed Computing and Networking*, volume 7129 of *Lecture Notes in Computer Science*, pages 1–15. Springer Berlin / Heidelberg, 2012.

[8] Ismel Brito and Pedro Meseguer. Synchronous, asynchronous and hybrid algorithms for DisCSP. In Mark Wallace, editor, *Principles and Practice of Constraint Programming*. 2004.

[9] P. Dittrich. Artificial chemistries-a review. *Artificial Life*, 7(3) :225–275, 2001.

[10] Arnaud Doniec, Sylvain Piechowiak, and René Mandiau. A discsp solving algorithm based on sessions. In Ingrid Russell and Zdravko Markov, editors, *FLAIRS Conference*, pages 666–670. AAAI Press, 2005.

[11] P. T. Eugster, R. Guerraoui, A.-M. Kermarrec, and L. Massoulié. Epidemic Information Dissemination in Distributed Systems. *Computer*, 37 :60–67, May 2004.

[12] H. Fernandez, T. Priol, and C. Tedeschi. Decentralized Approach for Execution of Composite Web Services Using the Chemical Paradigm. In *ICWS*, pages 139–146, 2010.

[13] Hector Fernandez, Cédric Tedeschi, and Thierry Priol. A Chemistry-Inspired Workflow Management System for Scientific Applications in Clouds. In *7th IEEE International Conference on e-Science*, Stockholm, Suède, December 2011.

[14] K. Gladitz and H. Kuchen. Shared Memory Implementation of the Gamma-Operation. *J. Symb. Comput.*, 21(4) :577–591, 1996.

[15] C. Hankin, D. Le Métayer, and D. Sands. A Parallel Programming Style and Its Algebra of Programs. In *Proceedings of the 5th International PARLE Conference on Parallel Architectures and Languages Europe*, PARLE '93, pages 367–378, London, UK, 1993. Springer-Verlag.

[16] Marko Obrovac and Tedeschi Cédric. On the Feasibility of a Distributed Runtime for the Chemical Programming Model. In *14th International Workshop on Advances in Parallel and Distributed Computational Models (APDCM 2012), in conjunction with IPDPS 2012*, Shanghai, China, May, 21 2012. IEEE. To appear.

[17] Manish Parashar and Salim Hariri. Autonomic computing : An overview. In *International Workshop on Unconventional Programming Paradigms (UPP)*, pages 257–269, 2004.

[18] R. Proust and C. Tedeschi. Peer-to-Peer Algorithms for a Distributed Chemical Machine. Technical report, ÉNS Cachan, Antenne de Bretagne and Irisa, Université Rennes 1/Inria, team MYRIADS, 2011.

[19] Antony Rowstron and Peter Druschel. Pastry : Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In Rachid Guerraoui, editor, *Middleware 2001*, volume 2218 of *Lecture Notes in Computer Science*, pages 329–350. Springer Berlin / Heidelberg, 2001.

[20] C. P. Schnorr and M. Euchner. Lattice basis reduction : Improved practical algorithms and solving subset sum problems. *Mathematical Programming*, 66 :181–199, 1994. 10.1007/BF01581144.

[21] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord : A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '01, pages 149–160, New York, NY, USA, 2001. ACM.

[22] R. Zivan and A. Meisels. Parallel backtrack search on discsp, 2002.

[23] Roie Zivan and Amnon Meisels. Message delay and discsp search algorithms. *Annals of Mathematics and Artificial Intelligence*, 46(4) :415–439, April 2006.