Laboratoire LaMI

Laboratoire de Méthodes Informatiques

Équipe SPÉCIF

CNRS UMR 8042

Typage et compilation de filtrage de chemins dans des collections topologiques

Antoine Spicher

Rapport de stage de DEA AMIB

Encadrants : M. Jean-Louis Giavitto et M. Olivier Michel

26 juin 2003

LaMI - Université d'Évry-Val d'Essonne
Tour Évry 2 / 4ème étage
523 place des Terrasses
91000 Évry - France

Remerciements

Je tiens tout d'abord à remercier mes maîtres de stage, Jean-Louis Giavitto et Olivier Michel, de même que Julien Cohen pour leur grande disponibilité, leurs idées et suggestions et de manière générale, pour toute l'aide qu'ils ont pu m'apporter durant le stage.

Je remercie également toute l'équipe de stagiaires, et en particulier Céline, Christophe, Clément et Fanny pour avoir répondu à mes questions ainsi que Matthieu pour m'avoir apporté des idées nouvelles et un point de vue extérieur intéressant.

Merci enfin à toute l'équipe SPÉCIF du LaMI pour m'avoir permis d'effectuer ce stage dans une ambiance agréable.

Résumé

Le projet MGS a pour objectif de concevoir un langage déclaratif de haut niveau dédié à la simulation de processus biologiques à structure dynamique. L'approche adoptée consiste à définir des règles d'évolution dans des structures de données complexes représentant l'état du système biologique. Ces règles d'évolution sont spécifiées à l'aide de règles de réécriture, appelées *transformations*, et les structures de données, appelées *collections topologiques*, sont abordées sous un angle topologique en définissant un voisinage pour chaque élément de la structure.

MGS propose une vision peu habituelle du filtrage qui permet de faire de la réécriture sur n'importe quelle organisation d'éléments, dès que celle-ci est définie au travers d'une unique relation de voisinage permettant de lier les éléments entre eux. Le filtrage est d'ailleurs implémenté dans MGS de façon générique et totalement indépendante des topologies des collections.

Cependant, cette généricité du filtrage perd toutes les propriétés inhérentes à chaque type de collection.

Ce rapport présente mon travail au sein du projet MGS qui consistait à étudier le filtrage. Après avoir participé à l'élaboration de l'algorithme de filtrage, je propose une classification des filtres fondée sur un calcul de type, et une optimisation du filtrage dont j'ai intégré l'application aux séquences et aux arbres modulo AC.

Table des matières

Table des figures

Contexte de l'étude

Position du problème

Modèlisation informatique et modèlisation mathématique.

La simulation par ordinateur d'un processus biologique passe par la définition d'un modèle formel. Ce modèle formel se distingue généralement des modèles mathématiques classiques par le grand nombre de composants hétérogènes impliqués dans la description du système, par la complexité des comportements de chaque entité, l'impossibilité de « compresser » l'évolution du système dans une forme analytique ou close, la présence de différents modes de fonctionnement, de non-linéarités et de discontinuités, etc.

Pour citer un exemple significatif, certains modèles de simulation informatique du cœur humain impliquent de l'ordre de 10^5 cellules d'une dizaine de types différents, chaque type correspondant à un ensemble d'équations non-linéaires capturant le comportement d'une cinquantaine de canaux ioniques, le tout organisé dans une géométrie réaliste [PH97]. La complexité de ce modèle interdit toute étude qualitative et les techniques mathématiques classiques échouent complètement à étudier les propriétés de ce genre de système.

On trouvera dans [HHLM99, Wol02, Cha02] une discussion permettant d'apprécier l'impact pour la biologie de cette différence entre modèle mathématique et modèle informatique.

La simulation informatique.

Cependant, dès qu'une description suffisamment rigoureuse est possible, l'informatique est capable de proposer la simulation du système biologique et donc l'exploration systématique de son comportement. Cela peut amener à valider ou invalider le modèle (en le comparant aux résultats expérimentaux disponibles), d'accéder à des paramètres impossibles à mesurer et, dans le meilleur des cas, à faire de la prédiction. La simulation est donc un outil précieux, indispensable et incontournable quand l'expérimentation *in-vivo* ou *in-vitro* est rendue impossible pour des raisons techniques, économiques ou éthiques.

La conception d'un environnement de simulation dédié à une classe de phénomènes biologiques est donc un problème très important et qui retient actuellement l'attention des informaticiens. On peut citer comme exemple de modèles de calcul inspirés par la biologie ou bien dédiés à la simulation d'une classe de phénomènes biologiques : les automates cellulaires [VN66], les systèmes de Lindenmayer [PL90a], les systèmes de Paun [Pau98b], le calcul par molécules (DNA computing, aqueous computing, ...) [Pau98a], la chimie artificielle [DZB00], les algorithmes évolutionnistes, les réseaux de neuronnes formels, diverses algèbres de processus (Danos, Cardelli, Shapiro), plusieurs adaptations des formalismes de réseaux de Petri ou d'automates booléens, etc.

Notons qu'un langage de simulation informatique joue de facto le rôle de langage de modélisation. On comprend donc l'intérêt de disposer de langages informatiques les plus proches possibles des notions biologiques manipulées et permettant de décrire adéquatement les processus en action.

Le problème de l'expressivité du langage de modélisation

Traditionnellement, les langages de simulation sont évalués suivant deux critères :
- l'expressivité du langage
- son efficacité

Un langage expressif permet de définir simplement et de manière concise, les entités d'une simulation et leurs interactions. Si on essaye de formaliser un système biologique à l'aide d'un langage impératif classique, il est évident que la majeure partie des efforts de programmation sera consacrée à la résolution de problèmes d'intendance tels que : la représentation des entités de la simulation, la gestion de la mémoire, la représentation du temps logique, la gestion du séquençage des activités des entités de la simulation, etc. Evidemment, le temps passé à ces tâche d'intendance est du temps perdu pour le biologiste (les anglophones parlent de "semantic gap").

Ainsi, il apparaît que l'*expressivité* d'un langage de simulation est au moins aussi importante que son *efficacité*. Ce point a été souligné par un rapport célèbre [NSF91] : les objets modélisés par les simulations actuelles sont de plus en plus complexes et leurs interactions compliquées ; les algorithmes de simulation deviennent de plus en plus sophistiqués ; il s'ensuit que le peu d'expressivité d'un langage constitue alors un obstacle à la mise en œuvre de ces nouvelles simulations. Avec la montée en puissance des microprocesseurs, et la standardisation des logiciels de base permettant de travailler sur des réseaux hétérogènes de stations de travail, *le nouveau challenge n'est plus seulement celui de la puissance de calcul mais celui de la programmation des modèles*.

C'est à ce problème que s'attaque le projet `MGS` en s'orientant vers des processus biologiques pouvant se modéliser par un *système dynamique*.

La modélisation des systèmes dynamiques biologiques

Les *systèmes dynamiques* (SD) sont un cadre très général pour modéliser des phénomènes décrits dans l'espace et qui évoluent dans le temps. Les entités constituant le système sont caractérisées par des grandeurs appelées *variables* dont la valeur évolue au cours du temps. Une variable peut prendre une valeur scalaire, ou bien représenter la variation d'une grandeur (scalaire ou vectorielle) sur une région de l'espace (comme par exemple la concentration d'une molécule dans un compartiment ou bien la vitesse en chaque point d'un fluide dans un tuyau). L'ensemble des variables qui décrit le système forme l'*état* du système. Les variations de cet état dans le temps composent la *trajectoire* du système et constituent le phénomène que l'on veut modéliser. L'espace dans lequel se déroule le phénomène correspond à l'ensemble des variables. La spécification d'un SD revient à décrire comment on passe d'un état à un autre dans une trajectoire.

L'approche des SD est de plus en plus importante en biologie et le schéma pointé par [TBCN00] :

$$\text{expression des gènes} \longrightarrow \textit{système dynamiques} \longrightarrow \text{physiologie de la cellule}$$

devient de plus en plus fondamental dans la tentative d'intégrer le volume exponentiels des données fournies par les méthodes de la biologie à grande échelle.

Il existe de nombreux formalismes différents pour décrire un SD : équation aux dérivées partielles, équation différentielle, réseau d'itérations couplées, automate, automate cellulaire, ..., pour n'en citer que quelques uns. Cependant, les systèmes dynamiques biologiques ont souvent une particularité qui empêche leur modélisation mathématique et qui oblige à recourir extensivement à la simulation : la structure de ces systèmes est elle-même dynamique [GGMP02, GM03]. Autrement dit, lors de la simulation, il est nécessaire de calculer la structure de l'espace des états (i.e. les variables) en même temps que la valeur des observables.

Un exemple simple permettra de faire la différence avec un SD habituellement rencontré en physique et un SD typique issue de la biologie. La description mécanique d'une pierre qui

tombe dans le champ de gravité correspond à un système dynamique dont les observables sont la position et la vitesse de la pierre. Ce système est toujours décrit par une position et une vitesse, même si la valeur de position et la valeur de la vitesse changent constamment au cours du temps. Nous dirons alors que le SD exhibe une structure *statique*.

Ce n'est pas le cas pour un système biologique. L'organisation dynamique des SD rencontrés en biologie a depuis longtemps été signalée sous des noms divers. On peut citer par exemple Rosen, Kaufman (hypercycle), Varela (système autopoïétique), Fontana (constructive dynamical system), etc.

Considérons par exemple le développement d'un embryon. A l'instant initial, nous devons décrire l'état de l'œuf (aussi compliqué que cela puisse être). Mais au cours du temps, il faudra décrire l'état d'une cellule, puis de 2, ..., de n, ..., et il nous faudra aussi décrire l'organisation spatiale changeante de ces cellules. Cette organisation spatiale est d'une importance fondamentale puisqu'elle influe sur la diffusion des gradients morphogénétiques qui à leur tour influent sur l'état de la cellule. Et bien sur, l'état des cellules influe sur l'organisations spatiale (par mobilité cellulaire, apoptose, différenciation, signalisation, etc.). Il est donc tout à fait impossible de décrire *a priori* la structure de l'état du système et nous dirons que la structure du système est *dynamique*.

L'exemple que nous venons d'esquisser est pris dans le domaine de la biologie du développement, mais les systèmes dynamiques rencontrés en biologie sont généralement des systèmes dynamiques à structure dynamique dès que l'on veut prendre en compte d'une manière un tant soit peu réaliste les phénomènes biologiques sous-jacents. Par exemple, dans le domaine de la régulation génétique, la prise en compte de la compartimentation et des réorganisations de la machinerie cellulaire conduisent directement aux SD à structure dynamique.

Cependant, si la description directe de l'espace des états (et donc de la structure du système) est impossible, il est toujours possible de décrire l'évolution du système comme résultant des interactions élémentaires set parallèles entre un ensemble organisé de composants.

Le langage MGS

Le projet `MGS` a pour objectif de concevoir un langage déclaratif de haut niveau dédié à la simulation de processus biologiques à structure dynamique. L'approche adoptée consiste à définir des règles d'évolution dans des structures de données complexes représentant l'état du système biologique. Ces règles d'évolution sont spécifiées à l'aide de règles de réécriture, appelées *transformations*, sur des structures de données, appelées *collections topologiques*, abordées sous un angle topologique en définissant un voisinage pour chaque élément de la structure.

Les travaux rapportés dans ce rapport reposent sur le filtrage et son implantation dans `MGS`. Un point remarquable est l'existence d'un langage de filtre, utilisé pour écrire les règles d'une transformation, qui est commun à tous les types de collection. Ce langage de filtres s'appuie sur les relations de voisinage et la notion de chemin induites par la topologie des collections.

Une règle de transformation `MGS` prend la forme $\alpha \Rightarrow \beta$ où α est un filtre permettant de spécifier un chemin dans une collection topologique. Le terme α est construit à partir d'opérateurs qui étendent la notion de langage régulier (concaténation, alternative, guarde, itération). Certaines classes de filtres correspondent à des chemins faciles à trouver dans une structure de données (par exemple en temps linéaire) et d'autres correspondent à des problèmes combinatoires non-polynomiaux.

Plan du rapport

Pour bien comprendre les mécanismes utilisés dans MGS, une première partie présente MGS de façon détaillée et technique. Cette présentation sera suivie de quelques exemples d'utilisation. Il s'agit principalement de phénomènes biologiques décrits brièvement et de leur programmation en MGS. Ces exemples montrent la facilité avec laquelle des phénomènes complexes peuvent être simulés avec MGS, en particulier dans le domaine de la biologie. Les différents exemples sont :

- L'expérience de BUGRIM : elle modélise l'activation d'une protéine à l'intérieur d'une cellule par la présence de deux signaux extérieurs.
- Le modèle de réaction-diffusion de TURING : il permet de modéliser l'apparition de motifs et une auto-organisation dans des systèmes biochimiques initialement homogènes.
- Le développement filamentaire d'*Anabaena catenula* : il s'agit d'une algue constituée d'un filament de cellules. On cherche à modéliser l'apparition de cellules différenciées à intervalle régulier le long du filament.
- Le développement du méristème : le méristème est un amas de cellules souches placé aux extrémités de certains organes des plantes (tiges et racines) et qui, en se développant, fait pousser la plante.
- TOPS : il s'agit de la programmation en MGS des diagrammes TOPS représentant des protéines et d'une règle de filtrage permettant d'y trouver un motif correspondant à une structure particulière de protéine.

La deuxième partie présente le travail que j'ai accompli durant ce stage et les résultats obtenus.

Il y a à la base du mécanisme de filtrage un algorithme générique qui permet de trouver des chemins dans n'importe quel type de collection. Cependant, il s'avère que sa complexité le rend rapidement inutilisable sur des simulations concrètes.

L'objectif du stage est d'apporter des éléments d'optimisation du filtrage. Cet objectif va être atteint en plusieurs étapes :

1. Dans un premier temps, il est intéressant d'observer les inconvénients de l'algorithme générique de base et d'apporter des solutions pour pouvoir en faire une implémentation la plus optimisée possible. Bien qu'il conserve la même complexité que l'algorithme d'origine, celui-ci doit être générique par rapport aux collections pour être utilisé lorsqu'aucun autre algorithme « spécialisé à la collection » n'a été intégré à MGS.

2. Ensuite, il faut proposer une classification des filtres à travers un système de typage (non standard) permettant de déterminer la complexité du filtrage. L'approche est similaire aux travaux de Vincent DORNIC ([Dor92]). On étudiera l'impact du problème de recherche en terme de coût du filtrage au pire. Cette fonction de coût permet de détecter si la recherche d'un motif particulier peut se faire avec les algorithmes d'une certaine classe.

3. On s'intéressera enfin à la compilation du filtrage. Pour certaines classes de filtres et, surtout, certains types de collections, des algorithmes très performants ont déjà été développés (par exemple pour le filtrage de séquence en utilisant des automates à états finis, ou bien le filtrage associatif-commutatif de termes, qui correspond à la recherche de chemins dans un multi-ensemble). De plus, certains filtres peuvent être réécrits dans une forme plus simple et plus efficace pour le filtrage suivant les propriétés de topologie de la collection. Par ailleurs, le rajout d'une contrainte supplémentaire peut simplifier la recherche d'un chemin (comme par exemple la recherche du chemin *minimal*).

Le chapitre 7 présente tout d'abord l'algorithme de filtrage générique utilisé avec n'importe quel type de collection topologique.

Une première optimisation sera ensuite apportée dans le chapitre 8 ; elle consiste en une séquencialisation des motifs qui permet d'éviter la manipulation des motifs avec notamment les transformations récursives effectuées par l'algorithme de base.

Le chapitre suivant (9) tentera alors de donner un coût aux motifs afin qu'on puisse les classer et déterminer la complexité de la recherche des chemins correspondants. Ce coût est présenté sous la forme d'un type, d'où l'emploi du terme *typage du filtrage*.

Finalement, deux optimisations plus profondes sont abordées dans le chapitre 10 ; elles concernent en effet une spécialisation de l'algorithme générique de filtrage suivant la topologie sur laquelle on travaille. Elles s'appliquent toutes les deux sur des collections monoïdales :

1. La première utilise un mécanisme de réécriture du motif pour accélérer la recherche dans les *arbres modulo AC*.
2. La seconde est une généralisation de l'algorithme de recherche de Boyer-Moore sur les séquences `MGS`.

Ma contribution au projet `MGS` est introductif à toutes les optimisations qu'on pourrait utiliser. Il reste énormément de travail à faire. Quelques propositions sont présentées à la fin du rapport.

Première partie

MGS : un langage dédié à la modélisation et à la simulation de phénomènes biologiques

Description de la partie

On trouve dans cette partie une présentation du langage `MGS` en trois parties. Tout d'abord, on présentera le domaine d'application de ce nouveau langage ; on présentera également les motivations d'un tel projet. Ensuite, les principes et les inspirations de `MGS` seront décrits. Enfin, une description détaillée du langage sera fournie pour permettre au lecteur de comprendre les exemples de simulation présentés dans la seconde partie du rapport, et la terminologie utilisée dans la présentation du travail effectué.

Cette présentation sera suivie d'une série de modélisations de phénomènes traduites ensuite en `MGS`. Elle permet d'observer l'intérêt réel qu'il y a à utiliser `MGS`. On pourra alors se rendre compte que `MGS` propose un outil facile d'accès et qui permet en quelques lignes de programmer un modèle.

Chapitre 1

Présentation de MGS

1.1 Modélisation et Biologie

La recherche en biologie est l'une des plus actives actuellement. Les progrès technologiques permettent en effet de mesurer de plus en plus précisément toutes sortes de paramètres. C'est une science travaillant à plusieurs échelles (depuis l'écosystème macroscopique jusqu'à l'organisation tridimensionnelle microscopique de molécules) ; elle tente de faire le lien entre ces différentes échelles et de comprendre les liens de causalité qui peuvent exister.

Les processus biologiques sont la plupart du temps des systèmes dynamiques structurés et hiérarchiquement organisés, dont la structure varie au cours du temps. Ce type de systèmes est courant dans les modèles de croissance de plantes, en biologie du développement, dans les modèles cellulaires intégrant plusieurs échelles, dans les mécanismes de transport de protéines et de compartimentation, etc.

La complexité des processus croît avec l'avancé technologique, et il est quasiment impossible pour un biologiste de gérer et comprendre l'ensemble des paramètres d'une expérience. L'informatique peut les aider dans ce travail en leur proposant des outils adaptés. Nous allons nous intéresser aux outils de modélisation et de simulation.

La modélisation permet une représentation du processus étudié. Le système est décrit par un certain nombre de paramètres. Une donnée de ces paramètres correspond à un état du système. Une fonction d'évolution permet, à partir de l'état courant du système et de paramètres extérieurs, de fournir le nouvel état du système. Ceci définit un modèle.

C'est à partir de cette description qu'il va être possible de simuler le processus biologique. Ainsi, on va, dans un premier temps, pouvoir vérifier que le modèle est valide en simulant le processus dans un cadre précis et observer si le résultat correspond à la réalité ; dans un second temps, il va être également possible de prédire l'évolution d'un système à partir d'un modèle valide.

Cependant, la complexité des systèmes biologiques rend difficile la spécification de leur structure et de leur dynamique. Les lois d'évolutions sont souvent informellement décrites comme un ensemble de transformations locales agissant sur un ensemble organisé d'entités.

Afin de passer de ce type de description informelle à un modèle calculatoire, plusieurs auteurs [Man01, FMP00] ont proposé de représenter l'état du système biologique par un terme t de la forme

$$t_1 + t_2 + \cdots + t_n$$

où un sous-terme t_i représente une entité biologique ou bien un message (signal, commande, information, action, etc.) adressé à une entité. La simulation de l'évolution du système biologique

se fait par réécriture du terme. On utilise des règles de réécriture de la forme

$$e + m \rightarrow e' + m'$$

La partie gauche correspond à un message m et à l'entité e auquel m est destiné ; la partie droite spécifie le nouvel état e' de l'entité et (éventuellement) de nouveaux messages m'. Notons que l'interaction directe entre entités peut être spécifiée par des règles de la forme

$$e_1 + e_2 \rightarrow e'_1 + e'_2$$

et la création d'une nouvelle entité par une règle de la forme

$$e \rightarrow e' + e''$$

etc.

L'opérateur + utilisé pour combiner les entités et les messages en un terme t peut être commutatif et associatif. Dans ce cas, le terme t réalise une « solution chimique » où les entités peuvent rencontrer librement les messages qui leurs sont adressés. Les entités et les messages interagissent donc de manière non organisée, simplement parce qu'ils sont présents dans la solution chimique.

Ce mécanisme est à la base du langage Gamma [BLM91] et du formalisme CHAM [BB89], où le terme t est interprété comme un multi-ensemble.

Le problème principal de cette approche, fondée sur la réécriture de termes, est l'impossibilité de représenter des *interactions organisées dans l'espace*. Ce besoin d'organisations plus complexes qu'une solution chimique motive par exemple plusieurs extensions [BH00].

Un des buts du projet MGS est de remédier à ce problème en permettant la représentation d'organisations complexes entre des entités variables et hétérogènes, ainsi que leur transformation par des règles locales.

MGS est l'acronyme de *un **M**odèle **G**énéral de **S**imulation (de système dynamique)*. Il s'agit d'un langage dédié à la modélisation et à la simulation de phénomènes biologiques.

Les langages dédiés fournissent au programmeur des abstractions et des notations adaptées à un domaine d'application particulier. Organisé autour d'un noyau généralement fonctionnel, leur spécialisation les rend a priori plus attractifs qu'un langage généraliste et permet de faciliter la programmation et la réutilisation de code.

1.2 Les principes du langage MGS

Le développement du langage MGS trouve ses inspirations et ses principes dans les travaux de A. Lindenmayer sur les L systèmes [RS92], G. Paun sur les P systèmes [Pau98b], G. Berry et G. Boudol sur la CHAM [BB89] ainsi que J.-P. Banâtre et D. Le Métayer sur Gamma [BLM86].

Ces langages utilisent une organisation structurée d'entités pour modéliser les systèmes. Les structures de données manipulées par les langages de programmation rendent compte de différentes sortes d'organisations simples : tableaux, arbres, ensembles, multi-ensembles (ou *bags*), séquences (ou listes), termes, *etc.* Par exemple, le type de structure de données sous-jacent dans Gamma, la CHAM et les P systèmes est le multi-ensemble, la séquence pour les L systèmes et le tableau pour les automates cellulaires (AC).

Nous appellerons *collection* tout ensemble « organisé » d'éléments (ceci englobe évidemment les structures qui viennent d'être citées).

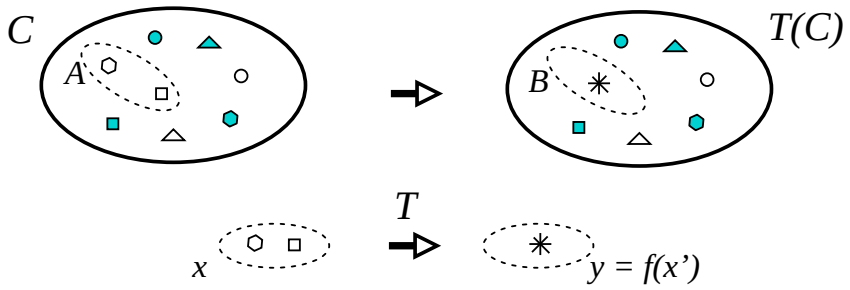


FIG. 1.1 – Transformation d'une collection. C est une collection. Une règle T spécifie qu'une sous-collection A dans C doit être remplacée par la collection B calculée à partir de A. La collection calculée dans la partie droite de la règle T dépend de la sous-collection filtrée dans sa partie gauche et éventuellement de son voisinage.

C'est par l'utilisation de toutes ces collections que MGS va être capable de simuler les systèmes dynamiques partageant les caractéristiques suivantes :

Espaces discrets. La structure de l'état est une collection discrète de valeurs : un multi-ensemble pour Gamma, une hiérarchie de membranes pour un P système, un mot pour un L système et un tableau pour un AC.

Temps discrets. L'état du système évolue selon une séquence discrète d'étapes temporelles.

Localité temporelle des transformations. Le calcul d'une nouvelle valeur dans un nouvel état dépend d'un nombre fini d'étapes précédentes (généralement une seule).

Localité spatiale des transformations. Le calcul d'une nouvelle collection se fait par combinaison structurelle du résultat de calculs plus élémentaires. Chacun de ces calculs élémentaires n'implique qu'un sous-ensemble de la collection initiale.

« Combinaison structurelle » signifie que les résultats élémentaires sont combinés dans une nouvelle collection, indépendamment de leur valeur. « Sous-ensemble » rend explicite le fait que seul un sous-ensemble déterminé et généralement petit des éléments initiaux est utilisé pour calculer une nouvelle valeur élémentaire (ceci est mesuré par exemple par le diamètre de la règle d'évolution d'un P système, le voisinage local d'un AC, le nombre de variables dans la partie droite d'une réaction Gamma ou par le contexte d'une règle dans un L système).

On peut également remarquer que, bien que la principale différence entre ces langages soit l'organisation des collections traitées, cela n'empêche pas le mécanisme de calcul (simulant l'évolution du système) d'être identique dans tous les cas : étant donné une collection C à traiter,

1. une sous-collection A est sélectionnée dans C,
2. une nouvelle collection B est calculée à partir de A (et éventuellement de son voisinage),
3. B vient se substituer à A.

voir Fig. 1.1. Ces trois étapes représentent une *transformation élémentaire*. Pour spécifier une transformation il faut préciser : comment sélectionner A, comment est calculé B et parfois les contraintes de la substitution de A par B dans C.

Ce point de vue abstrait permet d'unifier dans le même cadre formel les différents modèles de calculs cités. Pour chacun des langages il suffit de choisir la bonne organisation dans la collection utilisée.

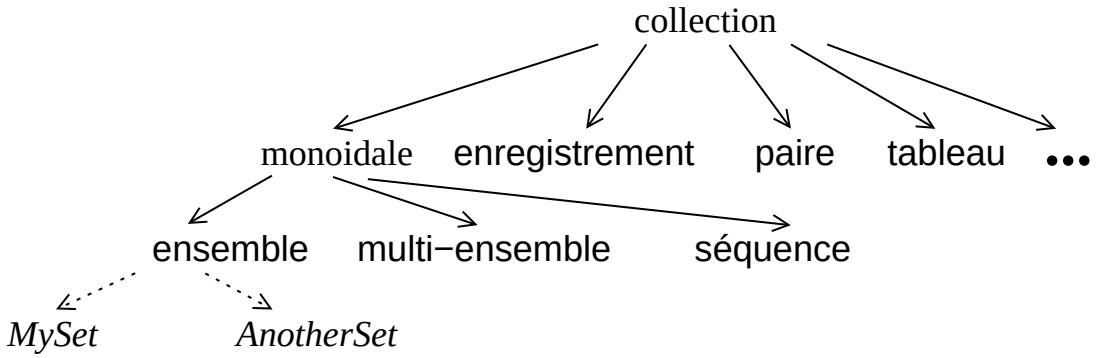


FIG. 1.2 – Relation de sous-typage des types de collection.

1.3 Description de MGS

MGS est un langage fonctionnel dynamiquement typé qui permet la manipulation de collections topologiques et leurs transformations. Les collections sont justes de nouvelles sortes de valeurs et les transformations sont des fonctions définies par des *règles*. MGS est un langage applicatif : les opérateurs utilisent des valeurs pour en calculer de nouvelles mais n'agissent pas par effet de bord.

Le typage est dit dynamique car il n'est pas vérifié statiquement et les erreurs de type sont détectées à l'exécution. L'ensemble des types possibles dans MGS est très riche, adapté au filtrage (*pattern matching*) et exploitable dans les règles de transformation.

1.3.1 Un nouveau type de valeur : les collections

En plus des types de base (entiers, flottants, chaînes de caractères, λ-expressions, etc.), MGS manipule également des collections.

Une collection peut être vue comme un ensemble de positions relatives les unes par rapport aux autres suivant une topologie. Celle-ci est définie à travers une relation de voisinage entre les éléments. Afin de souligner l'importance de l'organisation des collections, nous les appellerons *collections topologiques*.

Dans une séquence, par exemple, on peut définir le voisinage d'un élément comme l'élément précédent et le suivant. Nous noterons « , » la relation de voisinage, x, y signifiant que y est un voisin de x.

Cette approche topologique formalisant la notion de collection fait partie d'un long travail de recherche ([GMS96]) détaillé notamment dans [Gia00] qui étudie la notion de sous-collection et dans [GM01a] qui décrit un outil générique pour la définition de relations de voisinage. Les notions topologiques nécessaires à la définition de voisinages dans les séquences ou dans d'autres structures simples sont modestes. Néanmoins des notions plus complexes sont nécessaires pour la modélisation de certains processus biologiques [GM01c]. Par ailleurs, l'approche topologique permet de traiter différents problèmes de façon uniforme.

Les éléments d'une collection sont des valeurs qui peuvent être de base ou des collections quelconques et ne sont pas forcément du même type (collections hétérogènes) ; une collection peut contenir par exemple à la fois des entiers et des flottants. Les collections sont typées et il existe une relation de sous-typage entre leurs types telle que le montre la figure 1.2.

Les graphes

Pour ce qui concerne le travail développé ici, on peut considérer une collection topologique comme un graphe orienté quelconque. La relation de voisinage devient alors la condition d'existence d'un arc entre deux sommets du graphe. Les valeurs qui sont les éléments de la collection sont « positionnées » sur les sommets du graphe.
Les sous-types, présentés précédemment, correspondent en fait à des propriétés supplémentaires que possède le graphe :

- Les ensembles sont des graphes complets où chaque sommet est valué différemment de tous les autres ;
- Les listes correspondent à un graphe linéaire où les voisins d'un élément sont son prédécesseur et son successeur ;
- Les arbres sont des graphes sans cycle et qui possèdent une racine ;
- Les GBF sont des graphes de Cayley ;
- etc.

Cette vision est importante puisque si l'on veut écrire un algorithme général utilisable pour n'importe quelle collection, il suffit de l'écrire pour des graphes orientés quelconques. Néanmoins, chaque type de collection possède des propriétés qui lui sont propres et dont on peut se servir pour optimiser le filtrage (voire en utiliser d'autres plus adaptés).

MGS manipule donc plusieurs types de collections topologiques. Deux de ces types nous intéressent pour la suite du rapport, à savoir les *collections monoïdales* et les *GBF*.

Les collections monoïdales

Dans ce paragraphe, nous allons nous intéresser plus particulièrement aux types de collections ensembles, multi-ensembles et séquences. Ils sont dits *monoïdaux* car leurs valeurs peuvent êtres construites comme les éléments d'un monoïde avec l'opérateur d'ajout « , ». Suivant les types, l'opérateur « , » n'a pas les mêmes propriétés :

- pour une séquence, il est sans propriété particulière mise à part l'associativité ;
- les multi-ensembles sont obtenus avec un opérateur d'ajout associatif et commutatif ;
- les ensembles nécessitent un opérateur à la fois associatif, commutatif et idempotent.

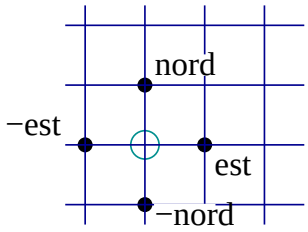
L'opérateur d'ajout muni de ses propriétés induit une topologie sur la collection et la relation de voisinage ; on dira que :

y est voisin de x dans une collection monoïdale C si C peut s'écrire U, x, y, V, où U et V sont des sous-collections de C.

Par exemple considérons l'ensemble $\{a, b, c\}$ (notation mathématique). Pour le définir en utilisant le constructeur « , » associé aux ensembles (qui, rappelons le, est associatif et commutatif), on peut écrire en MGS « $a, b, c,$ `set:()` » où `set:()` dénote l'ensemble vide. Les propriétés de l'opérateur « , » permettent d'obtenir le même ensemble avec l'expression « $a, c, b,$ `set:()` » et donc a et c sont voisins.

Les GBF

L'autre type de collection topologique de MGS auquel nous allons nous intéresser pendant le stage, est le GBF (pour *Group Based Field* en anglais ou champ de données fondé sur une structure de groupe en français).
Une telle structure de données se rencontre quand on discrétise de manière régulière un domaine de l'espace physique. Par exemple, considérons une discrétisation du plan par un maillage rectangulaire, le voisinage d'un point de discrétisation du plan peut être le suivant :



Chaque élément possède un voisin suivant chaque direction. On peut générer ce maillage à partir d'une structure de groupe dont les générateurs sont les directions.

De façon plus formelle, soient $a, b, c \ldots$ les directions permettant de se déplacer vers les voisins d'un point et $\mathcal{V}oisin(a, P)$ le voisin suivant la direction a d'un point P. Une direction a peut s'identifier avec l'opération de déplacement élémentaire $\mathcal{V}oisin(a, .)$. Ces déplacements peuvent être composés et cette composition possède une structure de groupe :
- elle est associative,
- pour chaque déplacement a on considère un déplacement inverse noté $-a$,
- et il existe un déplacement nul qui consiste à « ne pas se déplacer ».

L'application des déplacements à un point est l'action du groupe sur l'espace des points. Pour définir un espace homogène arbitraire, il faut donc spécifier deux choses :

1. définir le groupe des déplacements à partir des déplacements élémentaires $a, b, c \ldots$;
2. définir l'ensemble des points sur lequel le groupe va agir.

Pour résoudre dans MGS le second point, on considère que le groupe des déplacements agit sur lui-même : un élément du groupe correspond alors à un point de l'espace homogène et l'action du groupe correspond simplement à la loi du groupe : $\mathcal{V}oisin(a, P) = P.a$.

Pour spécifier un GBF (et plus généralement un groupe), on utilise une *présentation* : il s'agit d'une liste de générateurs et d'une liste d'équations entre générateurs. La syntaxe d'une présentation est généralement la suivante :

$$< g_1, g_2, \ldots, g_n; e_1, e_2, \ldots, e_m >$$

où $g_1, g_2, \ldots, g_n$ sont les générateurs et $e_1, e_2, \ldots, e_m$ les équations entre générateurs.

Tout élément du groupe s'obtient alors comme une somme des générateurs. Ici, ils correspondent aux déplacements élémentaires qui définissent le voisinage homogène.

Dans notre exemple de grille, la présentation du GBF est :

$$< nord, est; nord + est = est + nord >$$

Cette définition est très puissante et permet de faire du filtrage sur des structures non-algébriques, comme par exemple les tableaux.

1.3.2 Les enregistrements

Un *enregistrement* MGS est un type particulier de collection. Cependant, l'aspect collection des enregistrements restent frustre et on peut tout aussi bien les considérer comme des valeurs atomiques.

Il s'agit d'une table associant une valeur MGS de n'importe quel type à un nom symbolique appelé *champ*. Cette structure est très proche d'un enregistrement Pascal, d'une *struct C* ou encore d'un *record CAML*.

– Construction : un enregistrement se construit à l'aide des accolades :

$$\{a = 1, b = \text{"}red\text{"}\}$$

est un expression qui construit un enregistrement avec deux champs : un champ de nom a et de valeur 1, et un champ de nom b et de valeur la chaîne "red".

– Accès : il se fait grâce à la notation pointée :

$$\{a = (1 + 2), b = \text{"}red\text{"}\}.a$$

retourne la valeur de champ a, *i.e.* 3. L'accès à un champ inexistant renvoie la valeur <undef> sans déclencher d'erreur.

– Concaténation : une surcharge de l'opérateur + permet de fusionner deux enregistrements pour en donner un nouveau. Cette concaténation est asymétrique avec priorité à droite :

$$\{a = 1, b = \text{"}red\text{"}\} + \{a = 3, c = true\} = \{a = 3, b = \text{"}red\text{"}, c = true\}$$

1.3.3 Un nouveau type de fonction : les transformations

Une transformation est une fonction particulière, qui prend comme unique argument une collection, et qui en calcule une nouvelle. On peut donc itérer une transformation. Si on considère qu'une collection représente l'état d'un système dynamique (regroupant d'une certaine manière l'état des sous-systèmes), et qu'une transformation est une fonction d'évolution (définie par l'évolution des sous-systèmes et leur couplage), alors l'itération de la transformation sur la collection permet de faire évoluer le système au cours du temps.

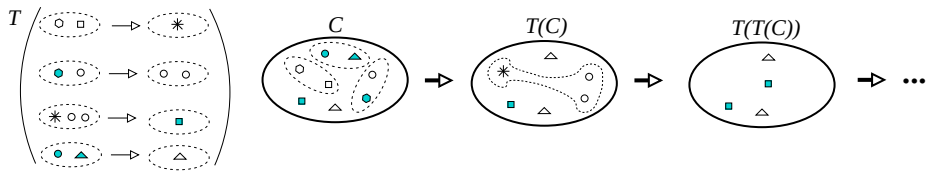


FIG. 1.3 – Transformation et itération d'une transformation. Une transformation T est un ensemble de règles de réécriture élémentaires appliquées en parallèle et indépendamment. Une transformation peut être itérée un nombre quelconque de fois, jusqu'à atteindre un point fixe par exemple.

Une transformation est un ensemble de règles noté :

$$\texttt{trans}\ T = \{\ \ldots \quad regle; \quad \ldots\ \}$$

et une règle a la forme suivante :

$$motif \ \Rightarrow \ expression$$

où *motif* définit un chemin qui permet de filtrer les sous-collections A de la collection C à laquelle la transformation est appliquée. A sera remplacée dans C par une collection B calculée par *expression*.

Lorsqu'une transformation est appliquée à une collection, la stratégie est d'appliquer en parallèle le plus grand nombre de règles possibles. Une règle peut être appliquée quand son motif filtre une sous-collection.

Le filtrage

Le filtrage d'un motif dans une collection est la notion la plus importante de MGS par rapport aux transformations. On peut noter que les motifs possèdent dans MGS leur propre sous-langage quasiment indépendant de la topologie sur laquelle ils sont appliqués (certains filtres sont spécifiquement réservés aux GBF dans lesquels un déplacement peut être défini grâce aux générateurs, et sont inapplicables dans les autres collections).

Les expressions de motifs ont donc le plus souvent un sens générique, puisqu'ils servent à filtrer n'importe quelle sorte de collection. La grammaire des expressions de motif est :

$$Pat ::= x \mid p, p' \mid p\,|dir{>}\,p' \mid p|p' \mid p^+ \mid p^* \mid p/exp \mid p \texttt{ as } x \mid (p)$$

où p, p' sont des motifs, x décrit des variables de motifs, exp est un prédicat et dir est une direction dans un GBF. Les explications qui suivent donnent une sémantique informelle de ces motifs :

variable : une variable de motif x filtre exactement un élément de la collection. La variable x peut être utilisée dans les gardes ou dans la partie gauche de la règle. MGS interdit la redéfinition de x.

voisin : p, p' est un motif qui filtre deux collections connectées p et p'. Par exemple, x, y filtre deux éléments connectés (*i.e*, y doit être voisin de x). La relation de connectivité dépend de la sorte de collection. Le motif $p\,|dir{>}\,p'$ est propre au GBF et filtre deux collections connectées p et p' suivant la direction dir. Ce motif ne peut évidemment pas s'appliquer sur autre chose qu'un GBF où dir est défini.

choix : $p|p'$ filtre une sous-collection qui est soit filtrée par p, soit par p'.

répétition : le motif $p+$ (resp. $p*$) filtre une sous-collection non vide (resp. une sous-collection pouvant être vide) d'éléments filtrés par p.

liaison : une liaison p `as` x donne le nom x à la collection filtrée par p. Ce nom peut être utilisé partout ailleurs dans la règle. Par exemple le motif (x, x) `as` y filtre deux éléments connectés de même valeur et y fait référence à cette sous-collection dans le reste de la règle.

garde : p/exp filtre la collection filtrée par p vérifiant exp. Par exemple, $y\,/\,y > 3$ filtre un élément y pourvu que y soit supérieur 3.

Voici un exemple complet. Le motif

$$((x/x > 3)|(y/y < 9))+ \texttt{ as } S \quad / \quad (\texttt{card}(S) < 5) \quad \& \quad (\texttt{fold}[+](S) > 10)$$

sélectionne une sous-collection S d'entiers strictement compris entre 3 et 9, tel que le cardinal de S soit inférieur à 5 et que la somme des éléments de la collection soit supérieure à 10. Si ce motif est utilisé avec une séquence (resp. un ensemble, un multi-ensemble), S dénote une sous-séquence (resp. un sous-ensemble, un sous-multi-ensemble).

Néanmoins, bien que toute transformation puisse être appliquée sur n'importe quelle collection, des erreurs sont levées lorsque le motif possède des constructions qui n'existent pas pour la collection. Par exemple, la construction « $\hat{}, x$ » sert à sélectionner dans une séquence un élément ne possédant pas de voisin gauche. Elle ne peut pas être appliquée sur autre chose qu'une séquence. Il en est de même pour les directions dans les GBF.

1.4 Conclusion

Nous n'avons présenté de MGS que la partie strictement nécessaire à la compréhension des exemples qui vont suivre. Une présentation détaillée est disponible dans [GM01c] et [GM01b].

Chapitre 2

Bugrin : modélisation d'un réseau biochimique

La chimie artificielle a pour but de réaliser virtuellement des réactions chimiques grâce à l'utilisation de programmes réalisant une expérience numériquement. Elle propose quelques avantages par rapport à la chimie « classique », dont celui de pouvoir recommencer l'expérience autant de fois qu'on le souhaite sans consommation de matière première. Cependant, pour qu'une expérience artificielle soit satisfaisante, il faut qu'elle colle à la réalité ; le modèle utilisé pour représenter l'expérience chimique réelle doit correspondre aux systèmes de réactions qui régissent le phénomène. Nous allons étudier un de ces modèles au travers des travaux de A. E. Bugrim sur la modélisation de signalisation biochimique.

2.1 Description

A. E. Bugrim (*cf.* [Bug]) a proposé un modèle informatique d'un processus de signalisation, c'est à dire un ensemble de réactions qui consistent à activer depuis l'extérieur d'une cellule, une protéine se trouvant à l'intérieur, par une cascade de réactions.

Le réseau biochimique sur lequel l'expérience de Bugrim s'appuie, est celui de l'activation de la phosphorylase kinase (PhK). Pour être totalement active, PhK doit être phosphorylée par une protéine kinase cAMP-dépendante (PKA). On dénote par C cette protéine. De plus, cette phosphorylation ne peut se faire que si PhK est préalablement liée au calcium.

Les deux signaux Agonist et Stimulus sont à l'origine de l'activation ; en effet, ils entraînent la création de C et l'importation depuis l'extérieur de la cellule de calcium. La figure 2.1 schématise le réseau d'interactions de l'expérience de Bugrim. Elle présente l'aspect temporel de la réaction avec les taux de diffusion τ_{diff}. L'organisation spatiale du réseau est décrite par la figure 2.2.

La création de cAMP débute lorsque le signal Agonist active le récepteur Rec situé sur la membrane plasmique de la cellule. Une cascade de 4 réactions se produit près de la membrane plasmique ; les protéines alors présentes (G-protein, AC) vont être activées pour, à leur tour, entraîner la transformation d'ATP (adénosine tri-phosphate) en cAMP. L'activité d'une protéine est représentée sur la figure 2.1 par une astérisque placée à côté de son nom : sa présence indique que la protéine est activée, et son absence qu'elle ne l'est pas. Cette protéine réagit alors avec PKA représentée sur la figure 2.2 par la structure R_2C_2. Ceci permet alors de libérer la sous-unité C qui constitue en partie PKA. Cette production de C subit un phénomène d'inhibition représenté par la réaction 7 de la figure 2.1. L'inhibiteur est dénoté par I et la forme inhibée de la sous-unité catalytique C de PKA par CI.

Le calcium est importé dans la cellule grâce à un canal qui s'ouvre sous l'effet du signal Stimulus. Des ions Ca^{2+} entre alors, attiré par le canal. Il se lie alors avec PhK pour donner

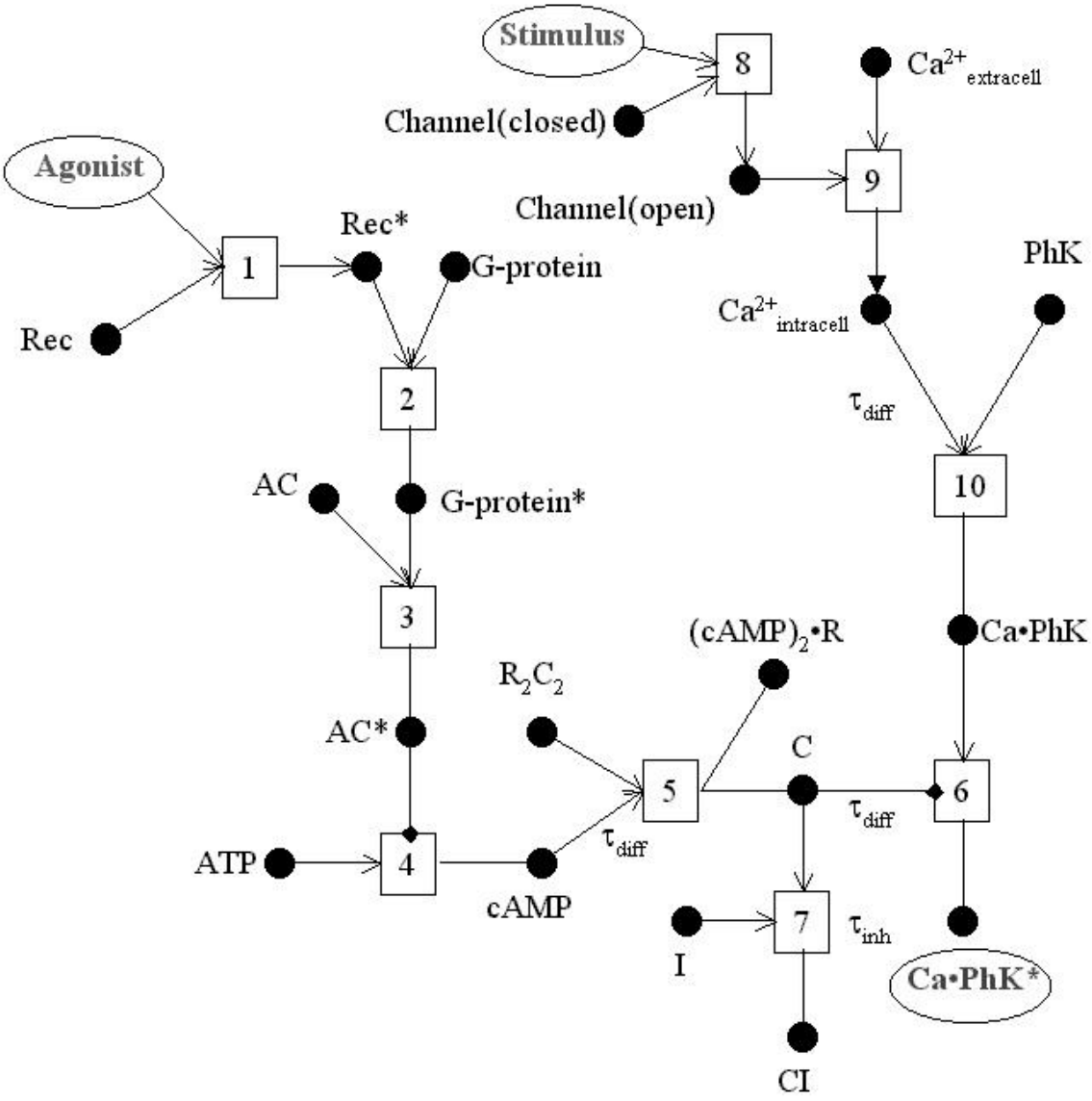


FIG. 2.1 – Réseau d'interaction de l'expérience de BUGRIM

Ca.PhK. *C* agit alors et active PhK.

Le point important de cette réaction n'est pas vraiment comment elle fonctionne, mais plutôt où et quand les différentes réactions ont lieu. En fait, la gestion du temps et de l'espace sont les plus difficiles à gérer dans un tel système. Les travaux de BUGRIM ont abouti sur une méthode simple de modélisation.

Certaines réactions ne s'effectuent pas immédiatement, car des facteurs biologiques complexes ralentissent le processus. BUGRIM décrit alors son expérience en simplifiant ces facteurs à l'aide d'un taux de diffusion. Il correspond à la durée du trajet de la molécule pour atteindre sa cible. Dans la modélisation, chaque élément possède donc un compteur calculant le temps écoulé. Dès qu'un certain temps (dépendant de la molécule et du volume ou de la surface de diffusion) s'est écoulé, la molécule devient réactive. Il s'agit d'une simplification ; il n'est plus utile de gérer les

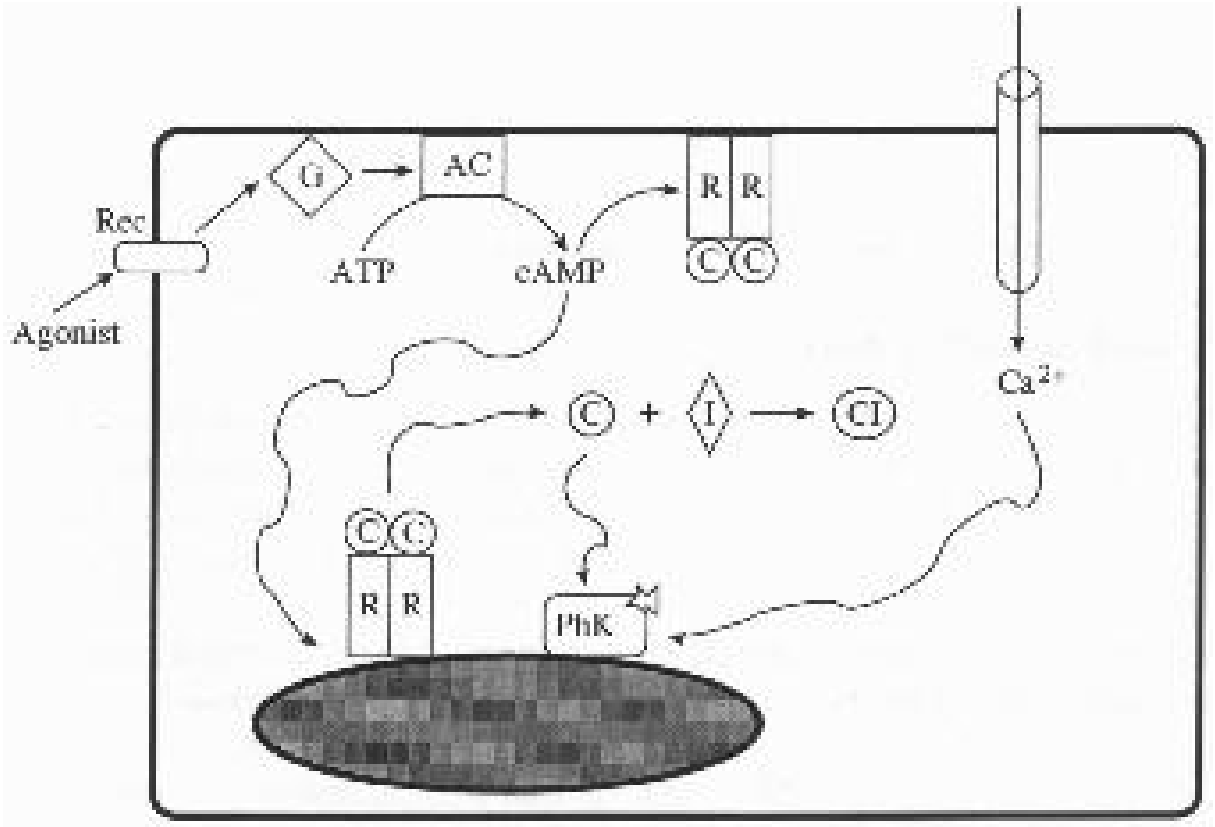


FIG. 2.2 – Organisation dans l'espace de l'expérience de BUGRIM

déplacements tri-dimensionnels des molécules. Un déplacement est un taux de diffusion qui doit atteindre un seuil pour que la molécule devienne réactive.

Nous allons maintenant observer comment il est possible de programmer ce modèle en MGS.

2.2 Le programme MGS

Structure du système

Le système est une cellule contenant des protéines et autres molécules en des quantités particulières.

Les acteurs des réactions sont les différentes molécules mises en jeu. Pour les modéliser, on utilise un enregistrement caractérisé par un champ `nom` permettant d'identifier le type de molécule. Chaque enregistrement créé représente une molécule ; on les insérera par la suite dans la collection qui permettra de modéliser la cellule :

```
state Molecule = {nom};;
```

En fonction du type de molécule, on lui ajoute un des deux champs suivant :

Activite : Au cours de la réaction, les molécules sont activées ou désactivées suivant leur environnement. Pour modéliser ce comportement, on utilise un entier qui prendra les valeurs 1 ou 0 suivant leur état, respectivement activé ou désactivé.

```
state Activite = {actif};;
```

Par exemple, pour le récepteur Rec, on a :

```
state Recepteur = Molecule + Acitivite + {nom = "recepteur"} ; ;
```

Diffuse : Il s'agit du compteur permettant d'observer un temps de diffusion avant que la molécule soit réactive (modélisation des déplacements). C'est un entier initialisé à 0 et incrémenté de 1 à chaque pas de temps. Chaque type molécule possède également un taux de diffusion particulier relié à la distance qu'elle doit parcourir pour atteindre le lieu de la réaction.

```
state Diffuse = {diffuse} ; ;
```

Par exemple, pour l'ion Ca^{2+}, on a :

```
state Ca2p = Molecule + Diffuse + {nom = "ca++"} ; ;
```

On construit ensuite la cellule pour qu'elle accueille les molécules. La cellule est modélisée très simplement ; elle est divisée en 3 compartiments imbriqués les uns dans les autres, à savoir la membrane plasmique, le cytoplasme et la membrane interne, comme le montre la figure 2.3. La cellule est entourée d'un environnement modélisant l'extérieur. Les phases de transport des

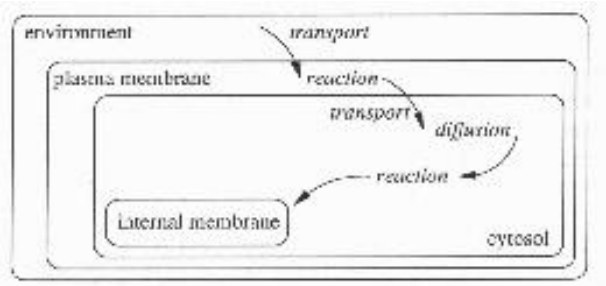


FIG. 2.3 – Modélisation d'une cellule dans l'expérience de BUGRIM

protéines ayant été simplifiées en temps de diffusion, il n'est pas nécessaire de situer les éléments les uns par rapport aux autres dans l'espace. Un multi-ensemble suffit donc à modéliser un comportement. Il suffit d'imbriquer les sous-ensembles pour modéliser la cellule. On crée pour cela deux types de multi-ensemble, l'un pour représenter les volumes et l'autre, les membranes :

```
collection Volume   = bag ; ;
collection Membrane = bag ; ;
```

On définit alors les 4 éléments modélisant la structure du système :

```
collection Univ       = Volume ; ;
collection Plasma     = Membrane ; ;
collection Cytosol    = Volume ; ;
collection Retic_Endo = Membrane ; ;
```

On crée enfin une cellule en emboîtant les collections comme suit :

```
RETICULUM := ():Retic_Endo ; ;
CYTOSOL   := {nom="i"}::RETICULUM::():Cytosol ; ;
PLASMA    := CYTOSOL::():Plasma ; ;
U := PLASMA::():Univ ; ;
```

Il ne reste plus qu'à placer les protéines dans la collection `U`. Ici, on a introduit l'inhibiteur I de production en C.

Dynamique du système

Il y a deux lois qui permettent de faire évoluer le système :

1. Il y a tout d'abord l'évolution spatiale des molécules présentes dans le cytoplasme. On va modéliser par la transformation `IncrCyt` la diffusion des molécules. Pour cela, on incrémente simplement les champs `diffuse` des molécules Ca^{2+}, cAMP et de l'enzyme C.

```
trans IncrCyt = {
  I1 = ca :Ca2p    => ca+{diffuse=ca.diffuse+1};
  I2 =  a :cAMP    =>  a+{diffuse=a.diffuse+1};
  I3 =  c :Cenzime =>  c+{diffuse=c.diffuse+1}
};;
```

2. On modélise ensuite les 10 réactions décrites dans la figure 2.1 dans une transformation comprenant 11 règles. On va juste décrire 2 de ces règles :

```
trans Biochimie = {
  Incr      = c :Cytosol => IncrCyt(c)::():Cytosol;
  ...
  Reaction8 = s :Stimulus1, p :Plasma =>
              (s+{actif=0}::ActiveChannel(p)::():Univ;
...
};;
```

   La règle `Incr` permet tout simplement de faire diffuser les molécules dans le cytoplasme. La règle `Reaction8` correspond à l'ouverture du canal laissant entrer les ions Ca^{2+} dans la cellule. `s` est un Stimulus activé, et `p` est la membrane plasmique. Cette réaction va faire perdre son activité à `s` et va ouvrir un canal dans `p` au moyen de la transformation `ActiveChannel` (elle est constituée d'une règle filtrant un canal désactivé et qui l'active).

   Toutes les autres réactions sont décrites de la même façon en `MGS`.

Une des réactions est particulière. Il s'agit de l'inhibition de C par I :

$$C + I \rightarrow CI$$

Le problème est de modéliser le taux d'inhibition décrit par Bugrim, qui représente le pourcentage de C transformé en CI. Étant donné que nous travaillons avec des molécules considérées individuellement (et non avec des concentrations), on ne peut déclarer que $x\%$ de C va être transformé. En effet, par exemple, il n'y a pas de sens à transformer 50% d'une seule molécule. On préférera agir de façon aléatoire, en considérant la probabilité qu'une molécule soit transformée. L'ajout de cet aléa permet d'avoir des état finaux différents.

On itère la transformation `Biochimie` jusqu'à ce qu'aucune règle ne s'applique plus.

2.3 Illustration

La figure 2.4 montre une représentation qu'on peut faire de ce modèle. Chaque boîte correspond à un multi-ensemble. La boîte externe représente l'univers et contient trois éléments : la molécule d'Agonist, dessinée sous la forme d'un cône mince, le calcium, représenté par l'autre cône, et la membrane plasmique correspondant à la boîte transparente. Les différentes molécules présentes dans la membrane sont dessinées sous forme de volume solide. L'ellipsoïde représente le cytoplasme, et la sphère qu'elle contient est le noyau de la cellule.

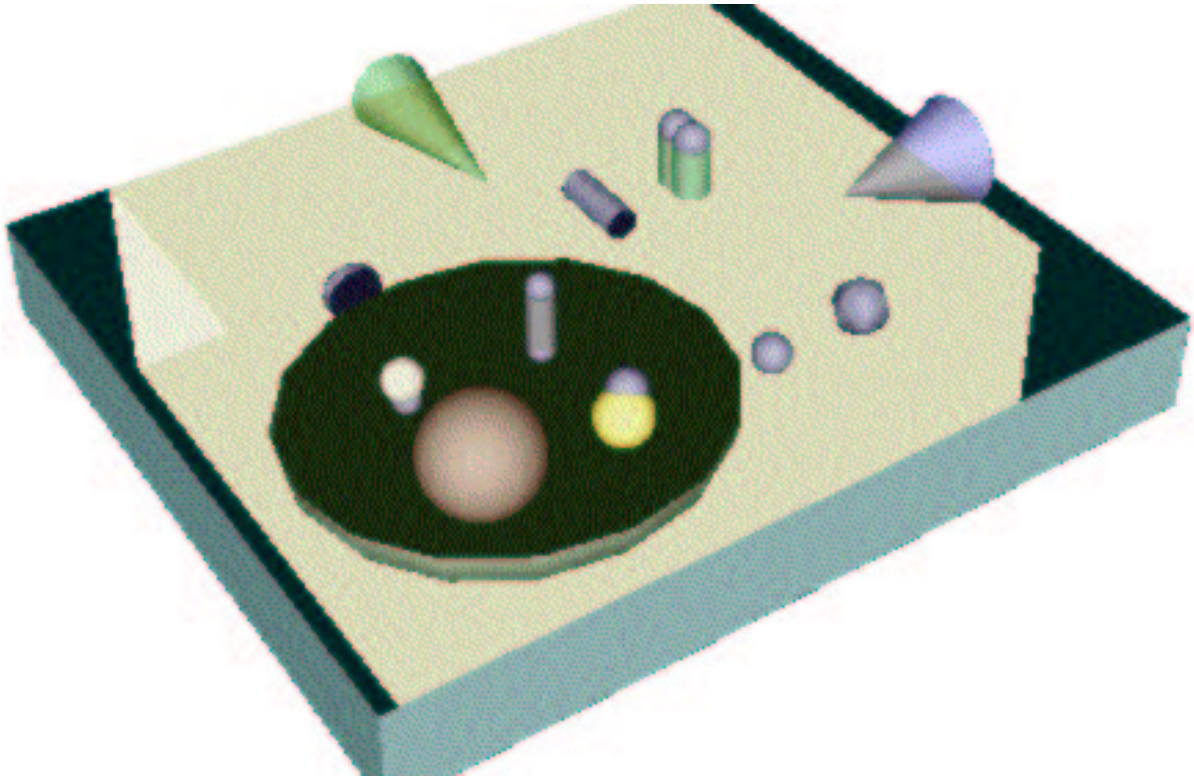


FIG. 2.4 – Modélisation de l'expérience de BUGRIM

Chapitre 3

TURING : un modèle de réaction-diffusion

Les systèmes d'équations de réaction-diffusion TURING sont considérés par les biologistes théoriques comme un des moyens par lequel une solution chimique ou bio-chimique initialement homogène, peut spontanément s'auto-organiser et laisser apparaître un motif macroscopique, par un mécanisme de réaction et de diffusion de substances chimiques, appelées *morphogène*. Nous allons mettre en place un exemple de système de TURING traduit en `MGS`.

3.1 Description

Dans les années 1950, le mathématicien Alan TURING (*cf.* [Tur52] & [BL74]) a proposé un système simple de réaction-diffusion décrivant des réactions chimiques et la diffusion pour expliquer complétement le phénomène de morphogénèse, *i.e.* le développement de formes dans les systèmes biologiques. Malheureusement, à cause la complexité des systèmes étudiés, les recherches n'ont pas encore abouti à la réalisation d'un modèle fondé sur les systèmes de TURING qui permettrait d'expliquer le phénomène de morphogénèse ; des études montrent tout de même que des solutions de microtubules *in vitro* peuvent s'auto-organiser en suivant le comportement du système, et que des mécanismes similaires pouvaient apparaître *in vivo* pendant l'embryogénèse (*cf.* [TVP99]).

Même si la présence de mécanismes suivant les lois des systèmes de Turing, n'a pas encore été démontrée, on utilise ces systèmes pour modéliser des exemples spécifiques tels que la coloration de la peau d'animaux (comme les tâches chez le léopard, ou encore les rayures chez le zèbre).

Les équations du modèles de TURING constituent un système d'équations différentielles partielles non-linéaires de la forme :

$$\frac{\partial \overrightarrow{u}}{\partial t} = \overrightarrow{k}.\nabla^2\overrightarrow{u} + \overrightarrow{F}\left(\overrightarrow{u}\right)$$

où le vecteur $\overrightarrow{u}$ décrit à chaque instant t les concentrations spatiales de substances chimiques appelées *morphogènes*. $\overrightarrow{F}$ dénote les spécifications de la réaction.

On propose dans cet exemple une programmation d'un système de TURING. On souhaite faire un apparaître un motif sur un tore. Ce tore est discrétisé en cellules, chacune ayant une concentration particulière en 2 morphogènes dénotés par u et v. Le système d'équations suivi par le modèle est :

$$\begin{aligned}
\frac{\partial u(x)}{\partial t} &= k * (\alpha - u(x)v(x)) + \mathrm{Diff}_u\left(\textstyle\sum_{y/x,y}(u(y) - u(x))\right) \\
\frac{\partial v(x)}{\partial t} &= k * (u(x)v(x) - v(x) - c(x)) + \mathrm{Diff}_v\left(\textstyle\sum_{y/x,y}(v(y) - v(x))\right)
\end{aligned}$$

où x dénote une cellule du tore ; $u(x)$ et $v(x)$ sont les concentrations en morphogènes u et v dans x. La fonction $c(x)$ correspond à une constante uniforme à tout le tore à laquelle on a ajouté un bruit spécifique à chaque cellule de façon aléatoire. Les variables α, k, Diff_u et Diff_v sont les paramètres de la réaction-diffusion. Enfin, $y/x, y$ désigne l'ensemble des voisins y de x, sur lesquels on somme.

Nous allons tout d'abord programmer ce système en `MGS`, puis observer les résultats.

3.2 Le programme `MGS`

Structure du système

Le système à modéliser est un tore. On va le discrétiser suivant 2 dimensions. Dans un premier temps, nous allons nous contenter de le découper dans une seule dimension. Cette division se fait naturellement régulièrement le long du tore. Il peut alors être assimilé à une séquence cyclique. Pour pouvoir étendre ensuite à une dimension supplémentaire, nous allons le représenter au moyen d'un GBF :

```
gbf torus = < a ; 180 a >; ;
```

Nous sommes en présence d'un GBF à une dimension `a`. L'équation `180 a` est en fait une écriture simplifiée et équivalente de `180*a = 0`. Elle signifie donc que si on se déplace dans le GBF 180 fois dans la direction `a`, on se retrouve à la position initiale. Le tore est donc divisé en 180 sous-cellules suivant cette dimension.

On souhaite maintenant passer à 2 dimensions. Il suffit donc d'ajouter une nouvelle dimension au GBF :

```
gbf torus = < a, b ; 180 a, 60 b >; ;
```

Le tore est divisé en 60 cellules dans cette dimension. En fin de compte, il est discrétisé en 10800 cellules.

Chaque cellule a pour valeur un enregistrement comportant les 3 champs `u`, `v` et `c` correspondant respectivement à la concentration en morphogène u, à celle en morphogène v et la constante de réaction c de la cellule. Pour initialiser ce tore, on utilise la transformation suivante :

```
trans init = {
  x => {
    u = u_steady,
    v = v_steady,
    c = beta_init + random(2*beta_rand) - beta_rand }
}; ;
```

A l'état initial, les concentrations en u et v sont uniformes sur les cellules du tore. La constante de réaction est calculée suivant la même base β_{init} pour toutes les cellules, à laquelle on ajoute un bruit compris entre $-\beta_{rand}$ et β_{rand}.

Il ne reste plus qu'à programmer la réaction-diffusion.

Dynamique du système

Le système de Turing constitue la loi d'évolution du système. Les lois d'évolution sont décrites en `MGS` par des transformations. On va donc programmer le système d'équation vu plus haut de la façon suivante :

```
trans Turing = {
  x =>
    let laplacian =
      neighborsfold(add, {u= 0.0 - 4.0*x.u, v= 0.0 - 4.0*x.v}, x)
    in
    let du = k * (alpha - x.u*x.v) + diffu*laplacian.u
    and dv = k * (x.u * x.v - x.v - x.c) + diffv*laplacian.v
      in {
        u = x.u + dt*du,
        v = x.v + dt*dv,
        c = x.c }
};;
```

La fonction `neighborsfold` applique la fonction `add` à tous les voisins de x. Elle utilise un accumulateur initialisé à `{u= 0.0 - 4.0*x.u, v= 0.0 - 4.0*x.v}`. Ceci permet de calculer un enregistrement contenant la somme des différences de concentration entre x et tous ces voisins (il en a 1 suivant chaque direction a, b, $-a$ et $-b$). Ceci correspond effectivement aux formules explicitées plus haut. La fonction `add` sert juste à faire la somme entre les champs `u` et `v` de deux enregistrements.

L'application de la transformation `Turing` sur le GBF correspond à une évolution du système d'un pas dans le temps ; en l'itérant, on fait évoluer les concentrations au cours du temps.

3.3 Illustration

On peut générer à partir de cette simulation un film montrant l'évolution du tore. Au cours du temps, chaque point de sa surface correspond à la concentration du morphogène v. Les deux figures suivantes présentent le tore à l'initialisation (3.1) et après 1200 itérations de la transformation (3.2). En chaque point du tore, l'épaisseur de l'anneau représente la concentration en v. Au début, la concentration est uniforme sur tout le tore ; il apparaît donc lisse. A l'état final, l'aspect ondulé correspond au motif et à l'auto-organisation de la concentration en v le long du tore sous l'effet de la diffusion des morphogènes.

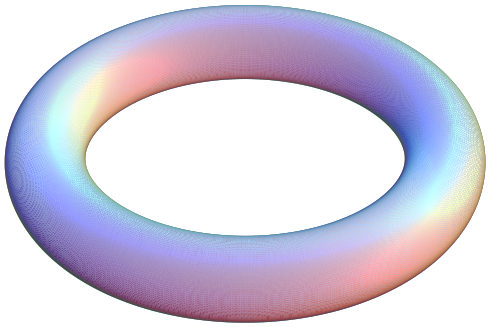


FIG. 3.1 – Modèle de TURING : état initial

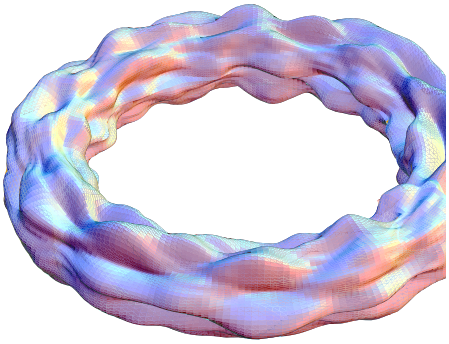


FIG. 3.2 – Modèle de TURING : état final

Chapitre 4

Exemple de développement filamentaire : *Anabaena Catenula*

L'anabaena est une cyanobactérie qui peut se développer en filament long de plus d'une centaine de cellules.

En carence d'azote, certaines cellules se différencient en cellules hétérocystes capables de fixer l'azote. Elles apparaissent de manière isolée et à intervalle régulier le long du filament . Ce motif s'explique par la réaction-diffusion de deux morphogènes dans un médium en croissance.

Le programme `MGS` correspond fidèlement aux règles de diffusion et de réaction de l'espace des cellules et aux règles de croissance de cet espace.

4.1 Description

Anabaena est une algue unicellulaire. Cependant, elle se présente dans certaines conditions sous la forme d'un filament de cellules. Ce filament a une particularité car, régulièrement le long de la chaîne, une cellule est différente ; on dit qu'elle s'est différenciée. On l'appelle cellule hétérocyste. Elle est plus grosse et vide. Contrairement aux autres cellules qui sont dans un état végétatif, elle ne peut plus se diviser et est condamnée à mourir.

Quel mécanisme est responsable de la différenciation de ces cellules, et surtout, du maintient de la distance à peu près constante qui les sépare ? Baker et Herman ont proposé le modèle de simulation suivant : les hétérocystes produisent une substance diffusée le long du filament et qui est utilisée par les cellules non-différenciées, créant ainsi un gradient à travers l'algue. Cette substance inhibe la différenciation des cellules végétatives en hétérocystes. Quand la concentration de cette molécule chute en-dessous d'une valeur seuil, la cellule n'est plus inhibée et se différencie en un hétérocyste.

Bien que le modèle de Baker et Herman soit capable de reproduire le motif observé suivi par les hétérocystes, il reste très sensible aux valeurs des paramètres. De petits changements de ces valeurs entraînent l'apparition simultanée de paires d'hétérocystes l'une à côté de l'autre. Ce n'est pas surprenant connaissant le fonctionnement du modèle. Le gradient de concentration de l'inhibiteur peut être trop faible au milieu d'une chaîne de cellules végétatives ; on ne peut alors pas définir précisément le point où le nouvel hétérocyste doit apparaître. En fait, la valeur seuil peut être atteinte par plusieurs cellules voisines simultanément, entraînant l'apparition de deux ou plusieurs hétérocystes voisins.

Le modèle de Baker et Herman peut être amélioré en ajoutant une compétition entre les cellules voisines sur le point de se différencier jusqu'à ce qu'une surpasse ses voisines. Wilcox est à l'origine de ce modèle « interactif ». Il peut être formalisé par l'utilisation d'un modèle de type

activateur/inhibiteur. A la substance qui inhibe la différenciation, les cellules diffusent également une seconde substance appelée activateur. La concentration de l'activateur est le critère qui distingue les cellules végétatives (faible concentration) et les hétérocystes (concentration élevée). Les deux substances agissent de façon contraire : la production de l'activateur est limitée par l'inhibiteur à moins que la concentration de ce dernier soit faible. Dans ce cas la production d'activateur croît radicalement grâce à une mécanisme autocatalytique (une forte concentration d'activateur entraîne une forte production d'activateur). Cependant, une concentration très élevée de l'activateur entraîne également la production de l'inhibiteur qui est alors diffusé aux cellules voisines. Les cellules produisant l'activateur inhibent donc sa production dans les cellules voisines. C'est ainsi qu'on peut modéliser la compétition entre cellules. Avec les bonnes valeurs des paramètres qui contrôlent ce mécanisme, on peut observer des cellules éloignées qui peuvent maintenir un état de production élevée d'activateur. Les travaux réalisés ici sont en étroite relation avec ceux de Przemyslaw PRUSINKIEWICZ (*cf.* [PL90b] & [Pru93]).

4.2 Le programme `MGS`

Structure du système

Le système est le filament de cellules qui compose l'algue. On utilise une séquence `MGS` pour représenter ce filament. Les éléments de cette séquence représentent les cellules. On les modélisera par des enregistrements comportant leurs concentrations d'activateur et d'inhibiteur, leur taille, leur statut (différencié ou non-différencié) et la polarité des cellules (information faisant partie du modèle).

L'enregistrement

```
init = {type = "C", a = 0.1, h = 10.0, x = 0.5, p = RightMark}
```

représente une cellule ; le champ `type` correspond au statut de la cellule. Ici, elle est dans l'état végétatif noté par la chaîne `"C"` (en opposition à `"D"` pour différencié). Les champs a et h représentent respectivement les concentrations en activateur et en inhibiteur de la cellule ; ici, la cellule présente une forte concentration en inhibiteur et peu d'activateur. Le champ x est la taille de la cellule qui grossit en suivant les règles de croissances. Enfin le champ p correspond à la polarité de la cellule, ici `RightMark` en opposition à `LeftMark`.

Cet exemple représente l'état de départ du système : une cellule unique non-différencié possédant une forte concentration d'inhibiteur et très peu d'activateur. Sa polarité est mise arbitrairement à `RightMark`. Sa taille est de 0.5 sachant que la taille maximale lm est de valeur 1.0.

Dynamique du système

Le modèle prévoit des règles d'évolution du système pour diffuser l'inhibiteur et faire croître et se diviser les cellules. Six règles (dont plusieurs équivalentes à la polarité près) suffisent pour décrire la dynamique d'Anabaena :

1. Les seules cellules pouvant se diviser sont celles de type `"C"`. Un prédicat C (respectivement D) permet de savoir si une cellule est végétative (respect. différenciée). Deux premières règles servent à la division cellulaire. Le principe est simple, si une cellule de type C a une taille supérieure à lm elle se divise créant ainsi deux cellules possédant les mêmes caractéristiques, à part la longueur et la polarité. L'une est plus petite que l'autre ; les variables `Longer` et `Shorter` permettent cette différence proportionnelle à la taille de départ. Voici ces deux règles.

   ```
   p1 = e / (C(e) & (e.x >= lm) & (e.p == RightMark))
   ```

```
  => { type ="C", a = e.a, h = e.h, x = e.x * longer, p = LeftMark},
     { type ="C", a = e.a, h = e.h, x = e.x * shorter, p = RightMark};

p2 = e / (C(e) & (e.x >= lm) & (e.p == LeftMark))
  => { type ="C", a = e.a, h = e.h, x = e.x * shorter, p = LeftMark},
     { type ="C", a = e.a, h = e.h, x = e.x * longer, p = RightMark};
```

2. La règle suivante représente la diffusion des substances entre cellules voisines. On cherche trois cellules voisines végétatives. On modifie alors les paramètres de la cellule centrale suivant la loi de diffusion et les concentrations des voisines. Soit e la cellule centrale, de paramètres a, h, x et p. Les concentrations des cellules voisines sont notées par a_l et h_l pour les concentrations en activateur et en inhibiteur de la cellule de gauche ; on a de la même façon a_r et h_r pour la cellule de droite. Les nouvelles valeurs de a et h, notées a' et h', sont données par :

$$a' = a + \left(\frac{\rho}{h}\left(\frac{a^2}{1+\kappa a^2} + a_0\right) - \mu a + D_a \frac{a_l + a_r - 2a}{xw}\right)\Delta t$$

et

$$h' = h + \left(\rho\left(\frac{a^2}{1+\kappa a^2} + h_0\right) - \nu h + D_h \frac{h_l + h_r - 2h}{xw}\right)\Delta t$$

avec ρ, κ, μ, ν, D_h, D_a et w les paramètres de la loi de diffusion. La règle est la suivante :

```
p3 = e / (C(e) & (e.x >= lm) & (e.p == RightMark))
  => let al = (left e).a in let ar = (right e).a in
     let hl = (left e).h in let hr = (right e).h in
     let h = e.h in let a = e.a in let x = e.x in let p = e.p
       in
       {  type = "D",
          a = a',
          h = h',
          x = x,
          p = p };
```

On peut remarquer que la cellule passe dans une phase de différenciation.

3. Les règles suivantes concernent les cellules en phase de différenciation (`type="D"`). On observe la valeur de l'activateur et la taille de la cellule. Si la cellule est plus grosse que `Shorter*gr` ou que la concentration de l'activateur est inférieure à un seuil `thr`, on fait grossir la cellule. La taille est multipliée par le facteur de croissance `gr`, et les concentrations sont réduites en conséquence. Ceci permet de contrôler la croisssance des cellules.

   Dans tous les cas, le type est modifié pour revenir à `"C"` afin que la cellule puisse subir la diffusion de la règle `p3`. Le code des deux règles est le suivant :

```
p4 = e / (D(e) & (e.a >= thr) & (e.x < shorter*gr))
  => { type ="C", a = e.a, h = e.h, x = e.x, p = e.p};

p5 = e / (D(e) & (e.a < thr) | (e.x >= shorter*gr))
  => { type ="C", a = e.a/gr, h = e.h/gr, x = e.x*gr, p = e.p};
```

4. La dernière règle sert à faire évoluer le système dans le cas où aucune des règles précédentes ne peut être appliquée :

```
p6 = e / C(e)
  => { type ="D", a = e.a, h = e.h, x = e.x, p = e.p};
```

4.3 Illustration

L'itération de la transformation définie par les règles p_1 à p_6 permet de simuler le développement d'un filament de Anabaena. Les figures 4.2 et 4.1 représentent cette évolution.

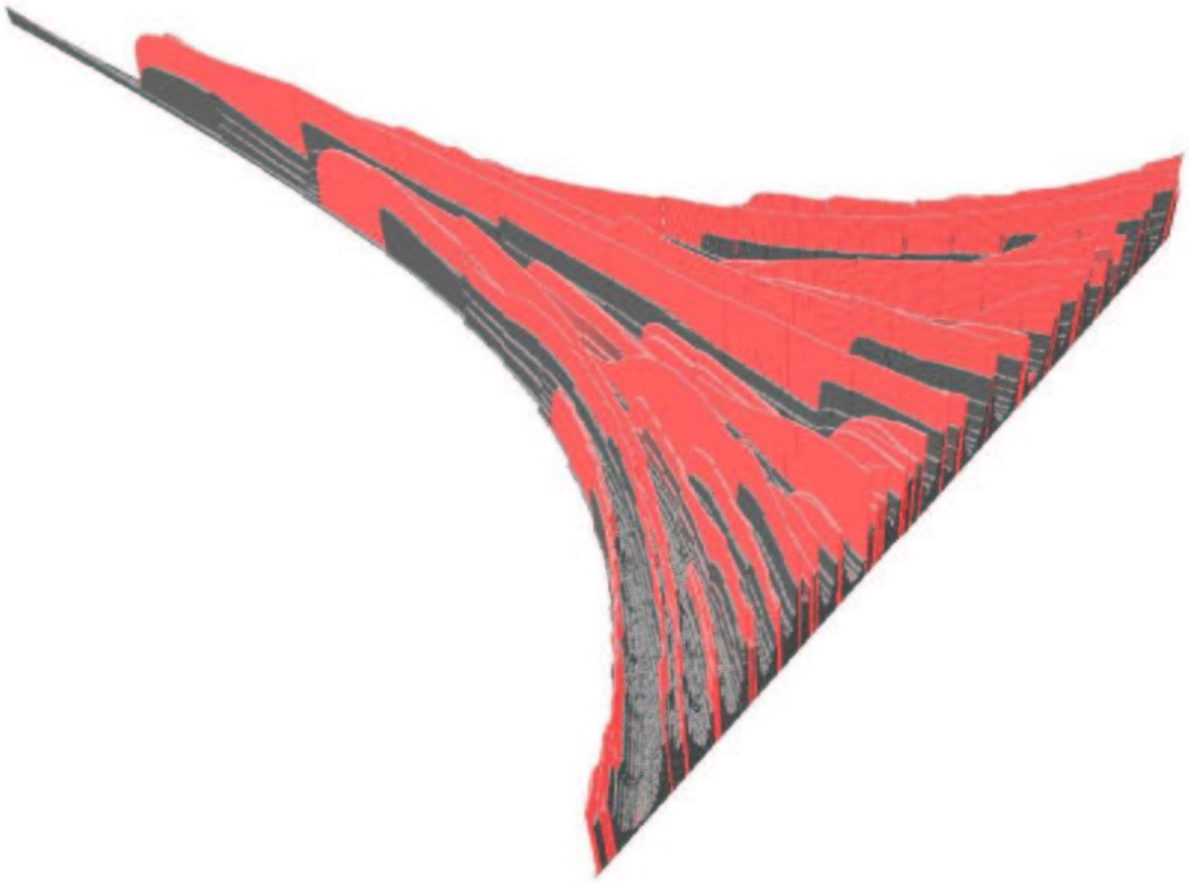


FIG. 4.1 – Simulation de l'évolution du filament de l'algue Anabaena au cours du temps

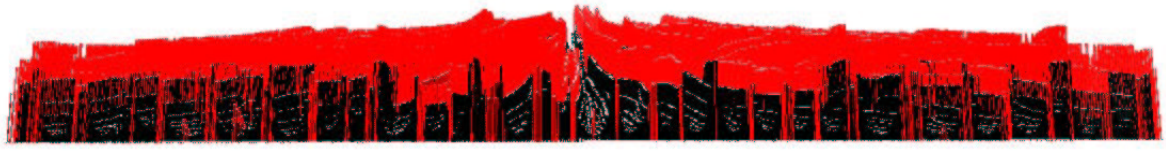


FIG. 4.2 – Représentation simulée d'un filament de l'algue Anabaena

Dans la figure 4.1, le temps s'écoule du coin supérieur gauche au coin inférieur droit. Chaque coupe transversale de cette figure permet d'observer l'état du filament à un moment précis. C'est ce qu'on observe sur la figure 4.2. La hauteur des cellules représente la concentration en activateur. Les cellules sont dessinées en rouge quand l'activateur atteint un seuil de concentration élevée entraînant la différenciation. Les cellules noires sont dans un état végétatif.

Chapitre 5

Développement du méristème

Le méristème est une zone cellulaire hautement active des plantes. En effet, il est constitué d'un paquet de cellules souches qui se divisent et permettent à la plante de pousser. Le projet GNOMON piloté par le CIRAD (Montpellier) et l'INRA (Versaille) a pour objectif de développer des méthodes algorithmiques et logicielles permettant d'aborder les problèmes posés par la modélisation de l'architecture des plantes et de leur croissance. L'étude de la modélisation du méristème entre dans le cadre de ce projet. Nous allons voir comment `MGS` peut apporter une solution facile à la programmation d'un tel modèle.

5.1 Description

Le projet GNOMON s'intéresse entre autre aux modèles algorithmiques de croissances de plantes. Il cherche en particulier à mettre en place les fondements d'un de croissance d'une population de cellules souches chez les plantes, appelée *méristème apical caulinaire*, notamment chez *Arabidopsis thaliana*. La figure 5.1 présente l'organisation du méristème, tansique la figure 5.2 est une photographie d'un méritème vu au microscope électronique. Le méristème caulinaire

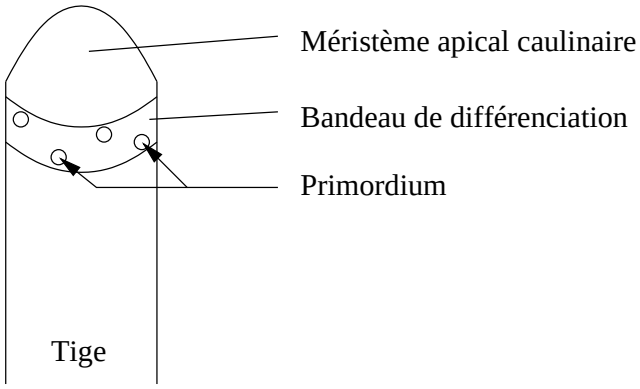


FIG. 5.1 – Représentation schématique du méristème apical caulinaire

constitue le haut de la tige des plantes. Il est entièrement constitué de cellules souches (capables de se différencier en n'importe quelle type de cellules spécialisées sous l'effet d'un message). Endessous du méristème se trouve un bandeau où les cellules souches se différencient en cellules construisant ainsi la tige de la plante. Cependant, toutes les cellules ne se différencient pas de la même façon et certaines, regroupées en un point appelé *primordium*, sont à l'origine d'un organe différent, comme par exemple une branche. Les primordia sont placés autour du bandeau assez

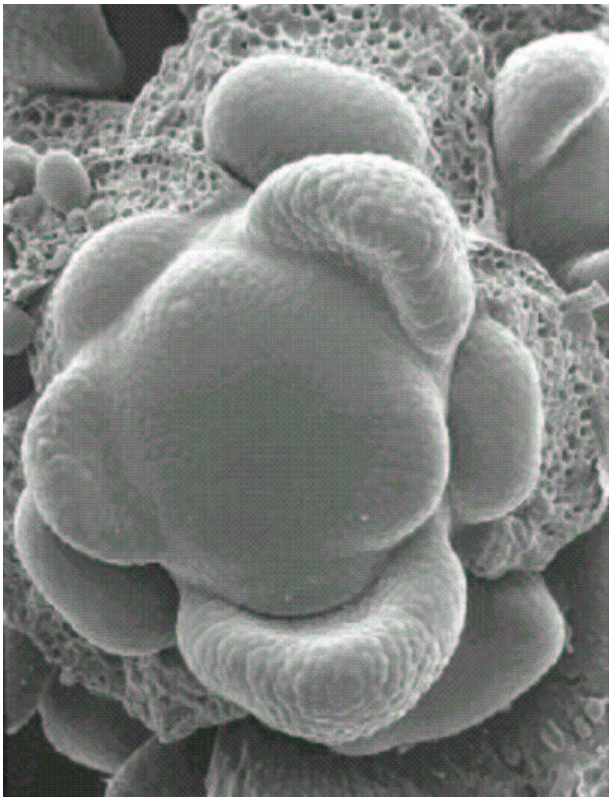


FIG. 5.2 – Méristème vu au microscope (photographie de Jan Traas, INRA-Versaille)

régulièrement suivant une spirale (relié à la suite de Fibonacci).

On souhaite tester l'hypothèse des « champs d'inhibition ». On suppose qu'un primordium inhibe la formation d'un autre primordium dans son environnement direct. La substance importante étant impliquée dans ces mécanismes d'inhibition serait l'auxine, une hormone végétale (*cf.* 5.3). Une forte concentration d'auxine stimulerait la formation de primordium alors que son absence l'inhiberait. On suppose qu'il existe un gradient d'auxine dans le méristème, créé par la présence de transporteurs d'auxine entre les cellules (pompes membranaires) localisés de façon polarisée. Le choix de représentation du tissu méristématique constitue un élément déterminant dans la modélisation des flux d'auxine, et par conséquent sur le processus organo-génétique lui-même.

On envisage de prendre en compte dans ce modèle des données génétiques ; les gènes dirigeant la formation du méristème ont été identifiés et on connaît également une partie des mécanismes régissant l'interaction entre ces gènes.

Nous proposons ici un exemple de modélisation de l'organisation spatiale des cellules. On ne se préoccupe pas de la diffusion d'auxine ou encore de la régulation génétique. On cherche juste à donner une base solide à la modélisation du méristème avec `MGS`. Cet exemple a été développé en collaboration avec Pierre BARBIER de REUILLE (Inra, UMR AMAP) qui fait une thèse au sein du projet GNOMON. Nous insistons ici sur le fait que notre objectif n'est pas de développer un modèle réaliste complet de la croissance du meristème, mais uniquement d'esquisser rapidement et de valider l'architecture et les notions logicielles nécessaires à un tel programme. Cet exemple est particulièrement révélateur des besoins de la modélisation biologique en structures spatiales complexes et de la nécessité de coupler ces structures spatiales avec les processus biochimiques.

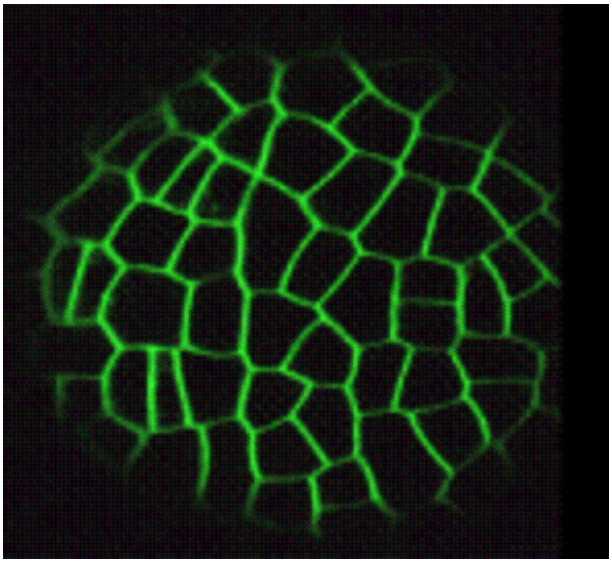


FIG. 5.3 – Flux d'auxine coloré dans un méristème (photographie de Jan Traas, INRA-Versaille)

5.2 Le programme MGS

Un méristème est composé d'une centaine de cellules voisines les unes des autres avec une connexité de l'ordre de 6 dans le plan. On souhaite juste travailler sur des cellules molles agglutinées les unes aux autres. On ajoutera également un développement de ce paquet de cellules en suivant des règles d'évolution créées pour donner vie à ce modèle.

Structure du système

Une première hypothèse qui a été faite par les biologistes du CIRAD et de l'INRA, est que la géométrie complexes des cellules pouvait se résumer par un graphe de Delaunay. Dans ce type de graphe, chaque cellule est représenté par un point P (son barycentre par exemple) et la connexité des cellules est calculé à partir de la position de ces points par une procédure dite "triangulation de Delaunay".

La triangulation de Delaunay associe à chaque point P la région dont tous les points sont plus proches de P que d'un autre point P'. Dans cette représentation, on a donc remplacer la diversité des situations géométriques rencontrés dans la réalité par une procédure automatique qui "compresse" cette information. En effet, toute la géométrie du système est décrite uniquement par un nuage de point.

Cette représentation approche la réalité car, bien sûr, la frontière de chaque cellule ne correspond pas exactement au voisinage calculé par la procédure de Delaunay. Cependant, cette approximation est utile dans une première phase et plusieurs éléments laissent penser qu'elle est peut être suffisante. Ce type de représentation de la géométrie d'un tissu de cellules, ou même d'un organe, a été proposé en 1981 par Honda, et est utilisé classiquement depuis.

Pour pouvoir conserver les cellules collées les une aux autres, on a inventé une force équivalente à la présence d'un ressort entre chaque cellule. L'utilisation de ressort permet d'attirer les cellules entre elles mais sans qu'elles s'écrasent les unes contre les autres. C'est ainsi que la mollesse de la structure cellulaire peut être modélisée. Les cellules positionnées dans l'espace bougent ensemble pour tenter de se rapprocher le plus les unes des autres. Il suffit de faire interagir les

cellules jusqu'à obtenir un équilibre pour créer un amas cellulaire réaliste.

Pour représenter les cellules, on utilise des enregistrements. Les différents champs sont les coordonnées tri-dimensionnelles (x, y, z) de la cellule pour pouvoir la situer dans l'espace, une distance l pour calculer l'interaction entre les cellules, et un paramètre `grow` permettant de savoir si la cellule croît.

```
state Position = {x, y, z };;
state Cell     = Position + {l};;
state Germ0    = Cell + {grow = 0};;
state Germ1    = Cell + {grow = 1};;
state Germ2    = Cell + {grow = 2};;
```

Techniquement, la relation de voisinage liant les cellules les unes aux autres dans la collection est calculée, comme indiqué précédemment, au moyen d'un graphe de DELAUNAY en dimension 3 par rapport à la position dans l'espace des cellules. `MGS` propose en effet des collections de type graphe de DELAUNAY. La triangulation de DELAUNAY est l'ensemble des tétraèdres en 3D dont les sommets sont les cellules et dont la boule circonscrite ne contient aucune autre cellule de l'espace dans son intérieur. Le graphe de DELAUNAY est le sous-complexe cellulaire de la triangulation de DELAUNAY. Elle permet ici d'harmoniser les interactions entre les cellules.

Dans la situation initiale, il y a 12 cellules de type `grow` 2, et une de type 0. La figure 5.4 montre l'organisation des cellules dans l'espace.

Dynamique du système

L'évolution des cellules dépend de leur type `grow` :
- `grow` = 0 : la cellule donne naissance à deux petites cellules de type 1 ; on peut la comparer à une cellule souche ;
- `grow` = 1 : la cellule se transforme en trois cellules de type 2 ;
- `grow` = 2 : la cellule grossit et ne se divise plus.

Cette loi d'évolution est codée en `MGS` à travers 3 règles de réécriture, pour chaque type `grow` de cellule :

```
trans Grow =
{
  r / r.grow < 0
    => r + {grow = r.grow+1 };

  r/ r.grow == 0
    =>
      r+{z = r.z + alpha, grow = -5},
      r+{x = alpha*cos(p1+theta), y = alpha*sin(p1+theta), grow=1, l = r.l/2},
      r+{x = alpha*cos(p4+theta), y = alpha*sin(p4+theta), grow=1, l = r.l/2};

  r/r.grow == 1
    =>
      r+{x = r.x + 0.2, y = r.y + 0.2, l = r.l*2, grow=2},
      r+{x = r.x*cos(theta2) - r.y*sin(theta2),
         y = r.x*sin(theta2) + r.y*cos(theta2),
         l = 2*r.l, grow=2},
      r+{x = r.x*cos(theta2) + r.y*sin(theta2),
```

```
            y = -r.x*sin(theta2) + r.y*cos(theta2),
            l = 2*r.l, grow=2} ;

    r/r.grow == 2 => r + {l = r.l + 0.15} ;
  } ; ;
```

On peut remarquer que la cellule souche voit son champ `grow` passer à -5 lorsqu'elle donne naissance à deux nouvelles cellules. Cela permet de temporiser le phénomène ; il faut en effet qu'il repasse à 0 sous l'effet de la première règle pour que la seconde s'applique. Ainsi, les cellules ont le temps de grossir entre deux nouvelles arrivantes.

Le code peut paraître compliqué ; il ne s'agit en fait que des calculs de rotation qui permettent de placer les cellules nouvellement crées ; ainsi, l'amas de cellules se développe de façon régulière.

`MGS` ne permet actuellement pas de modifier la structure du graphe de DELAUNAY ; or, celui-ci doit être modifié puisque les cellules se divisent. On peut néanmoins contourner le problème en mettant dans une séquence l'ensemble des cellules, en calculant sur cette séquence les divisions cellulaires, et en recalculant ensuite le graphe de DELAUNAY. La fonction itérée qui fait évoluer le système est la suivante :

```
fun H(d) =
  let dd = Meca[20](d) in
  let s  = sequify(dd) in
  let ns = Grow(s)     in
    delaunayfy(D3 :(), ns)
;;
```

Le paramètre `d` est le graphe de DELAUNAY représentant le système. On lance plusieurs fois la transformation `Meca` qui simule l'interaction entre les cellules qui a été présentée précédemment. Elle permet d'atteindre l'état d'équilibre. On transforme alors le graphe en une séquence `s` pour ensuite appliquer les règles d'évolution des cellules. Enfin, on recalcule le graphe de DELAUNAY.

5.3 Illustration

Le programme fournit un résultat assez réaliste bien qu'il ne se fonde pas sur aucune hypothèse biologique. Cependant, on constate qu'il est facile d'utiliser le langage `MGS` pour modéliser toute sorte de système. La collaboration avec le projet GNOMON est donc en bonne voie comme le montre les figures suivantes. On voit en effet comment les cellules bourgeonnent pour former un cône de plus en plus grand.

Sur ces figures, le code des couleurs est le suivant : rouge quand `grow` $= 2$, vert pour 1, et bleu de -5 à 0.

La figure 5.4 représente l'état initial de la simulation. Les figures 5.5 et 5.6 correspondent à la division de la cellule souche (en bleu) en deux cellules vertes. La figure 5.6 propose une vue d'en haut de l'amas cellulaire.

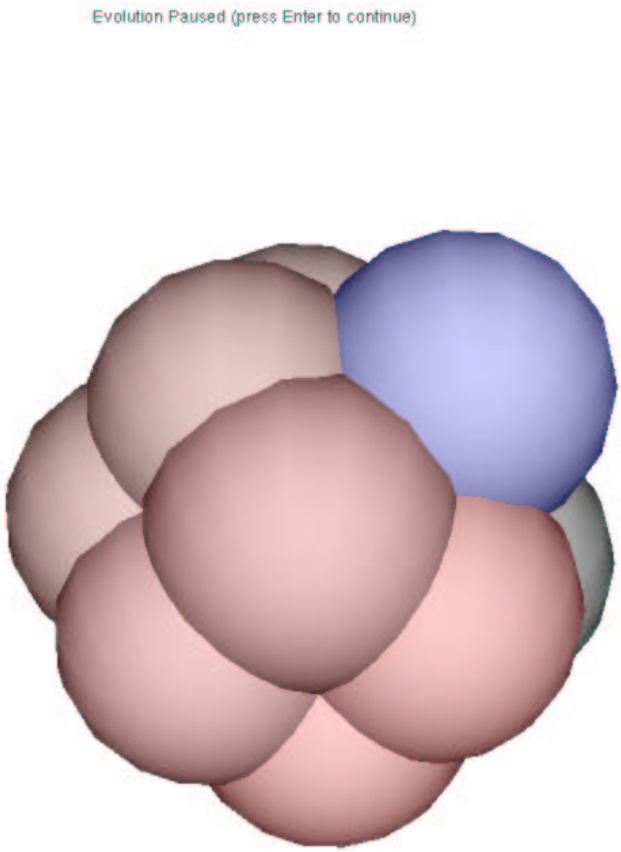


FIG. 5.4 – Simulation de méristème : état initial

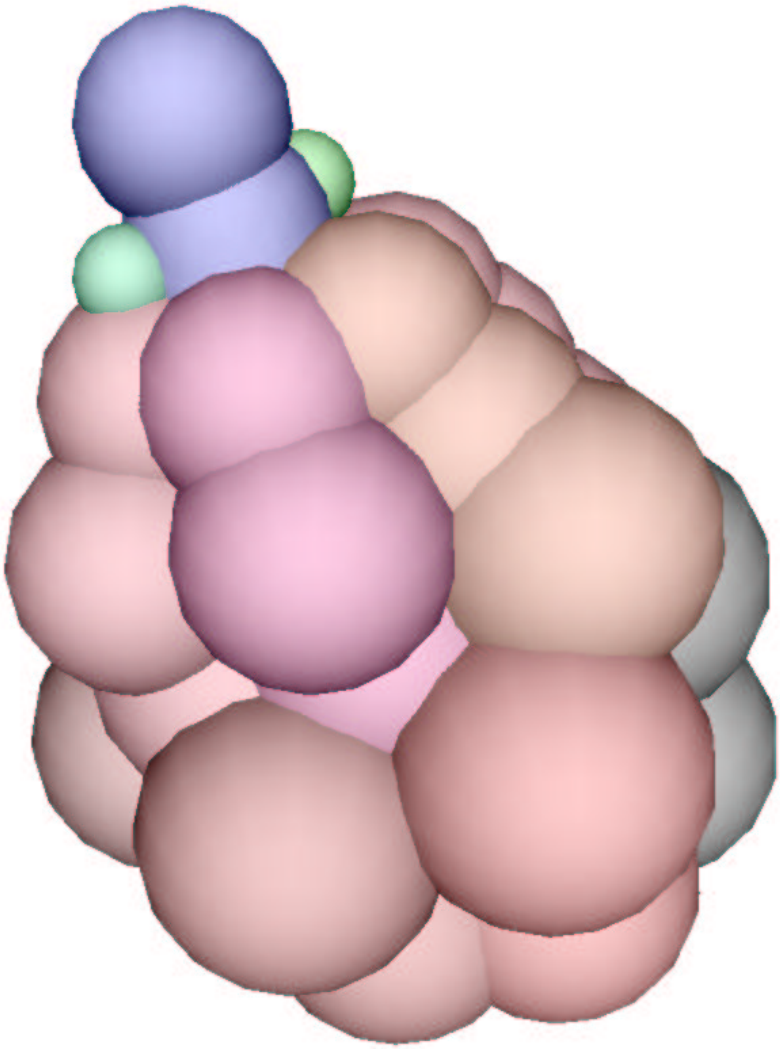


FIG. 5.5 – Simulation de méristème : division de la cellule souche

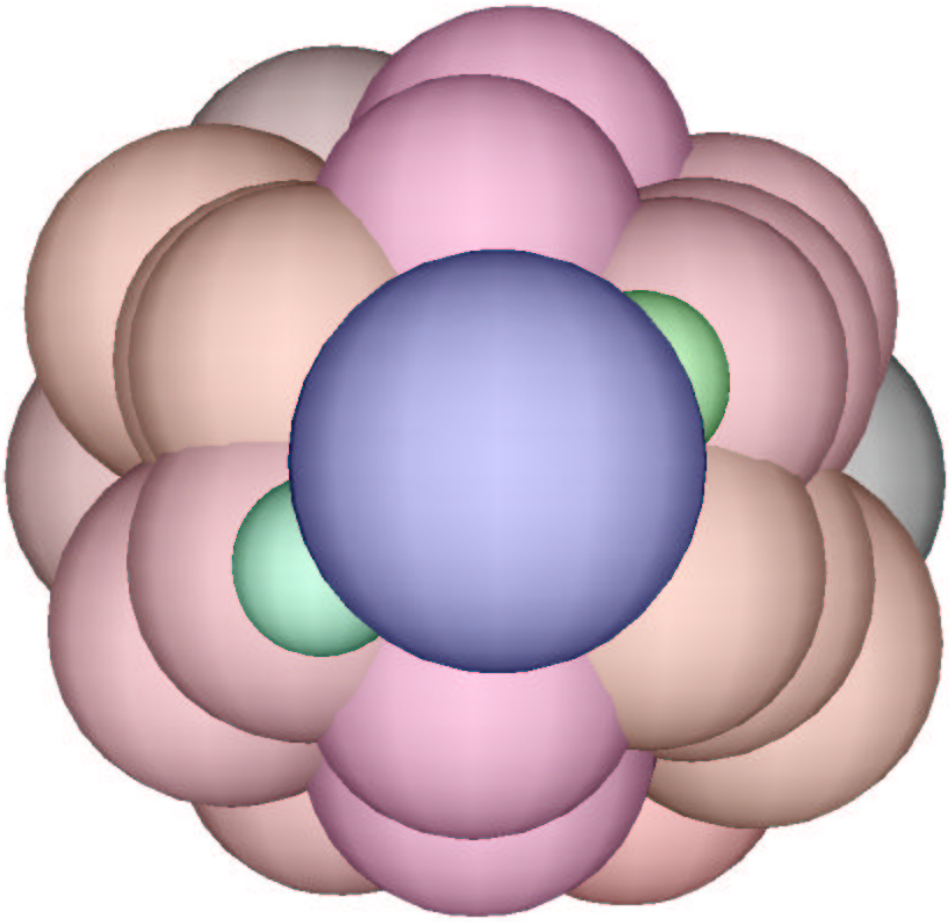


FIG. 5.6 – Simulation de méristème : vue du dessus du méristème

Chapitre 6

TOPS : des motifs topologiques pour les protéines

Ce dernier exemple propose la création d'un programme `MGS` permettant la recherche de motifs particuliers dans les digrammes TOPS. Ces digrammes permettent de décrire la topologie des protéines à partir des structures secondaires qui les constituent.

6.1 Description

Après avoir déterminé la forme tridimensionnelle d'une protéine, les biologistes doivent en déduire des hypothèses sur ses fonctionnalités. Une méthode utilisée actuellement est la comparaison deux à deux de la structure de la protéine inconnue avec celles de protéines déjà connues. Des outils tels que DALI ou CATH sont dédiés à ce genre de recherche. Le problème est que les bases de protéines connues croissent rapidement. La Protein Data Bank compte environ 15000 descriptions de structures de protéines, nombre qui ne fera qu'augmenter dans les années à venir. De plus, si des similitudes ont été trouvées avec d'autres protéines, on ne peut pas assurer que cela implique une similitude entre leurs fonctionnalités.

Une autre approche consiste à comparer les protéines, non pas au niveau de la séquence (avec un alignement), mais par rapport à leur structure. Si on se place à un niveau simple de comparaison, on peut décrire la structure des protéines par un « cartoon » TOPS. TOPS est utilisé pour ***Topology Of Protein Structure***. La figure 6.1 présente la description par un tel dessin de la protéine 2bopA0, et une comparaison avec son diagramme généré par RasMol. Le cartoon TOPS

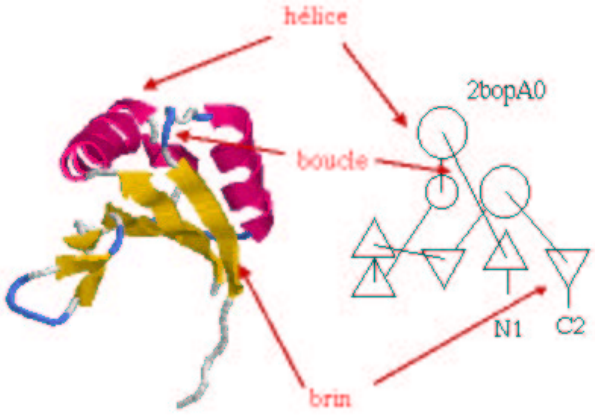


FIG. 6.1 – Cartoon TOPS et digramme de RasMol de 2bop20

montre les éléments de structure secondaire, *i.e.* les brins β (représentés par des triangles) et les hélices α (représentées par des cercles) ; il montre aussi comment ils sont connectés dans la séquence depuis le site initial amine au site terminal carboxyle, et leurs positions et orientations spatiales relatives. Cependant, le cartoon TOPS ne suit pas des règles strictes en ce qui concerne l'apparence.

Les digrammes TOPS ont développé sur les bases des cartoons par GILBERT (*cf.* [GWNT99]). Ils présentent les structures secondaires des protéines sous un aspect formel. Ils donnent des informations sur les groupements de brins β. (ils sont connectés par des liaisons hydrogène qui peuvent être parallèles ou anti-parallèles), et des informations sur les orientations relatives des éléments (connectés suivant leur chiralité gauche ou droite). Ce diagramme comporte plusieurs simplifications. Notamment, toutes les liaisons hydrogènes connectant une paire de brins β sont représentées par une seule sur le schema. De même, bien qu'on puisse définir des chiralités entre n'importe quels éléments secondaires, le diagramme TOPS ne représente qu'un sous-ensemble des plus importantes chiralités ; il correspond aux informations implicites sur les positions dans le cartoon TOPS. La figure 6.2 présente le diagramme de 2bop20.

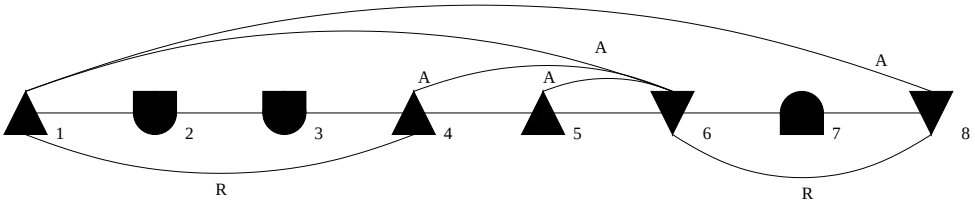


FIG. 6.2 – Diagramme TOPS de 2bop20

Le diagramme suit la définition suivante :

$$\begin{aligned}
\text{Diagramme} &= (S, H, C) \\
\Sigma &= \{\alpha^+, \alpha^-, \beta^+, \beta^-\} \\
S &= (S_1, ..., S_k), 1 \leq i \leq k, S_i \in \Sigma \\
H &= \{(S_i, \delta, S_j) | S_1, S_j \in \{\beta^+, \beta^-\} \delta = P \leftarrow (S_i = S_j) \delta = A \leftarrow (S_i \neq S_j)\} \\
C &= \{(S_i, \varphi, S_j) | S_1, S_j \in \Sigma, \varphi \in \{L, R\}\}
\end{aligned}$$

Les sommets du graphe représentent les structures secondaires composant la protéine ; ils sont de quatre types définis dans Σ. Ils correspondent à l'orientation vers le haut ou vers le bas des brins β et des hélices α. Ces sommets sont chaînés les uns aux autres dans la séquence S. A cela, on ajoute quatre sortes d'arcs : valués par A ou P, suivant le parallélisme entre deux brins β, et par L ou R pour la chiralité entre les éléments (droite ou gauche).

L'exemple de 2bop20 est formalisé de la façon suivante :

$$\begin{aligned}
2bop20 &= (S, H, C) \\
S &= (\beta_1^+, \alpha_2^-, \alpha_3^-, \beta_4^+, \beta_5^+, \beta_6^-, \alpha_7^+, \beta_8^-) \\
H &= \{(\beta_1^+, A, \beta_6^-), (\beta_1^+, A, \beta_8^-), (\beta_4^+, A, \beta_6^-), (\beta_5^+, A, \beta_6^-)\} \\
C &= \{(\beta_1^+, R, \beta_6^-), (\beta_6^-, R, \beta_8^-)\}
\end{aligned}$$

Les travaux présentés dans [VG] concernent la mise au point d'un algorithme de filtrage dans les diagrammes TOPS. Les motifs représentés également par des graphes, suivent la même définition que les diagramme TOPS. La figure 6.3 décrit un motif. En comparant les deux diagrammes 6.2 et 6.3, on s'aperçoit qu'on peut faire correspondre respectivement les sommets 1, 2, 3, 4, 5 et 6 du motif avec les sommets 1, 2, 4, 6, 7 et 8 du diagramme de 2bop20. La figure 6.4 représente ce filtrage : les sommets dessinés en gris correspondent au motif. Le problème correspondant à ce

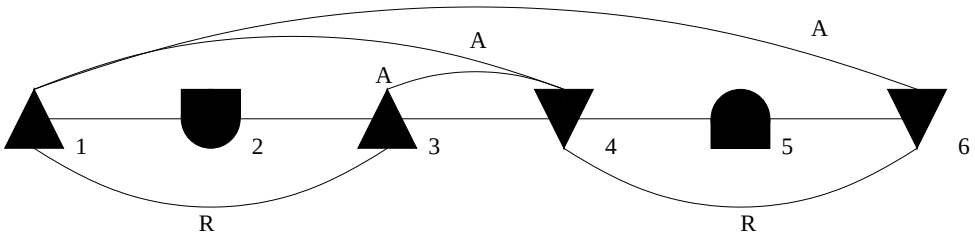


FIG. 6.3 – Exemple de motif TOPS

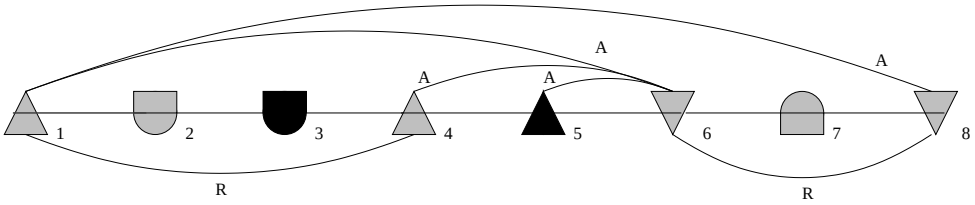


FIG. 6.4 – Filtrage de diagramme TOPS

filtrage de diagramme TOPS est en fait un problème de comparaison de graphe (en anglais *graph matching*). Il peut en effet être réduit facilement aux problèmes de sous-graphe isomorphique et de sous-graphe commun maximal.

6.2 Le programme MGS

Nous allons programmer s'il est possible d'écrire en MGS une structure de données et un filtre adapté à ce problème en MGS.

Représentation du diagramme

Il faut mettre au point une collection qui permet de gérer les diagrammes TOPS. Les graphes associés aux diagrammes TOPS sont particuliers : il s'agit de graphes aux sommets ordonnés et aux arcs étiquetés.

Nous allons les représenter par une séquence de sommet qui permet de conserver leur ordre. Chaque sommet possède un identifiant et un type. Un enregistrement convient à les représenter.

```
state Sommet = {nom,type};;
```

Les arcs sont étiquetés et joignent deux sommets du graphe. On va faire porter l'information concernant l'arc sur le sommet le plus en tête de la liste S dans la définition du diagramme. On rajoute aux enregistrements possédant 4 champs, un par type d'arc, qui seront des ensembles de sommets.

```
state LinkL = Sommet + {linkL};;
state LinkR = Sommet + {linkR};;
state LinkP = Sommet + {linkP};;
state LinkA = Sommet + {linkA};;
```

par exemple l'arc $(\beta_1^+, A, \beta_8^-)$ du diagramme de 2bop20 sera défini par l'enregistrement :

```
{nom="S1",type="beta+",linkA="S8",():set}
```

Le diagramme complet de 2bop20 représenté par :

```
diag2bop20 =
  {nom="S1",type="beta+",linkA=("S6","S8",():set),linkR=("S4",():set)},
  {nom="S2",type="alpha-"},
  {nom="S3",type="alpha-"},
  {nom="S4",type="beta+",linkA="S6",():set},
  {nom="S5",type="beta+",linkA="S6",():set},
  {nom="S6",type="beta-",linkR="S8",():set},
  {nom="S7",type="alpha+"},
  {nom="S8",type="beta-"}
```

Représentation du motif

La difficulté dans l'écriture du motif est la gestion des arcs. On utilise pleinement le système de gestion de variables proposé dans `MGS` pour conserver en mémoire les liens qui existent entre les sommets. Voici le programme `MGS` du filtre pour le motif présenté plus haut en exemple :

```
motif =
  {nom=n1,type="beta+",linkA=la1,linkR=lr1}, _*,
  {nom=n2,type="alpha-"}, _*,
  {nom=n3,type="beta+",linkA=la3}
    / member(n3,lr1), _*,
  {nom=n4,type="beta-",linkR=lr4}
    / member(n4,la1) & member(n4,la3), _*,
  {nom=n5,type="alpha+"}, _*,
  {nom=n6,type="beta-"}
    / member(n6,la1) & member(n6,lr4)
;
```

Conclusion

Comme nous venons de le voir à travers plusieurs exemples, `MGS` est un langage de programmation réellement adapté à la modélisation de systèmes dynamiques à structure dynamique très utilisés en biologie.

Cependant, `MGS` peut présenter quelques inconvénients, avec notamment une lenteur lors des calculs de certaines simulations. En effet, dès que les éléments sont en grande quantité dans des collections complexes (comme pour l'expérience de BUGRIM), le temps de calcul augmente de façon conséquente.

Dans la suite de ce rapport, nous allons voir pourquoi `MGS` met autant de temps à filtrer des collections. Nous étudierons pour cela l'algorithme de filtrage et apporterons quelques optimisations.

Deuxième partie

Etude du filtrage dans MGS

Description de la partie

Nous venons de voir comment MGS pouvait être utilisé pour simuler quelques processus biologiques. On peut remarquer la facilité avec laquelle l'organisation des éléments et la traduction des règles d'évolution du système en transformations, peuvent être programmées.

C'est l'utilisation que MGS fait du filtrage qui permet d'obtenir une telle facilité de programmation recherchée par les biologistes. On avait l'habitude de voir dans les langages fonctionnels une utilisation des filtres fondés sur la construction algébrique des éléments. *OCaml* représente très bien ce type d'utilisation.

Dans MGS, le filtrage repose sur un autre point de vue. Il vise à déterminer des sous-parties de collections sur lesquelles les règles de transformation sont appliquées. Là où les langages fonctionnels permettent habituellement d'avoir une vision **globale** de la structure filtrée (par exemple, une liste est soit vide, soit composée d'un élément en tête et de la queue de la liste), MGS permet une vision **locale** (on peut par exemple chercher dans une liste deux éléments consécutifs vérifiant une certaine propriété). C'est notamment grâce à ce principe de localité que les modèles biologiques, souvent décrits par des règles d'évolution locales, sont facilement traduits en MGS.

Cette utilisation différente des filtres permet entre autre de faire de la réécriture sur des constructions non-algébriques, ce qui ne pouvait pas être envisagé avant. On peut en effet programmer des transformations de tableaux (et de beaucoup d'autres organisations régulières de l'espace) grâce aux GBF. Plus généralement, MGS doit être capable de faire de la réécriture sur n'importe quelle organisation d'éléments, dès que celle-ci est définie au travers une unique relation de voisinage permettant de lier les éléments entre eux. Aussi, le filtrage est programmé dans MGS de façon générique et totalement indépendante des topologies que suivent les collections.

Cependant, la généricité du filtrage a un coût : on n'utilise pas les propriétés inhérentes à chaque collection. Par exemple, si on filtre dans un ensemble, on sait que tous les éléments sont voisins les uns des autres. Cette information est perdue alors qu'elle permet, comme nous le verrons plus loin, de filtrer plus efficacement. De fait, le prix de la généricité est donc très élevé : les problèmes de filtrage, ou plus généralement de recherche de chemins dans des graphes, sont déjà difficiles, et MGS accroît cette difficulté en tirant pas partie des propriétés des graphes.

Mon travail consistait à évaluer le prix de la généricité et à optimiser le filtrage ponctuellement. En effet, il n'est pas possible de l'optimiser de façon conséquente sans prendre en compte les propriétés des collections. On est donc obligé de spécialiser l'algorithme générique, voire même d'en utiliser un autre, et cela pour **chaque** collection topologique.

Nous décrirons tout d'abord la technique de « dérivation » de motif qui est à la base du filtrage dans MGS. A partir de ça, nous verrons que l'algorithme générique tel qu'il est implanté dans MGS actuellement et qui est équivalent à la définition de la dérivation des motifs, est fondé sur une séquencialisation des motifs.

Pour bien comprendre les éléments devant être mis en place durant le filtrage et son coût selon le motif, le chapitre 9 présente une classification des motifs selon leurs complexités temporelles et spatiales, que j'ai inventée. Le coût d'un motif est alors défini par un type qui correspond à la

classe de complexité du motif dans cette classification. Cette classification est indépendante de la collection. Elle n'amène donc pas à une optimisation mais se révèle être un indicateur efficace pour comparer le temps mis par deux motifs différents pour être filtrés par un même type de collection.

Enfin, je présente dans le chapitre 10 les optimisations que j'ai élaborée. Elles sont fondées sur la recherche en premier des éléments les plus contraignant du motif, principalement les constantes. L'algorithme générique cherche les chemins filtrés à partir de **toutes** les positions de la collection. La recherche des constantes en premier réduit considérablement le nombre de points d'entrée de l'algorithme. J'ai spécialisé cette optimisation sur deux types de collections :

1. les structures d'arbre modulo AC : une réécriture du motif fondée sur les propriétés de commutativité et d'associativité de la relation de voisinage permet d'obtenir des résultats très satisfaisants. Il suffit alors de placer en têtes du motif les éléments les plus contraignants.
2. les séquences : le filtrage d'un mot dans un texte est très proche de la recherche d'un chemin dans une séquence `MGS`. Une généralisation de l'algorithme de Boyer-Moore a permis d'obtenir de bons résultats. Cet algorithme cherche en fait à déplacer le plus possible la fenêtre de lecture lors du filtrage. Le calcul de ces déplacements dépend fortement de la présence d'éléments contraignant le long du motif.

Il suivra la conclusion du stage rappelant les différents résultats et travaux effectués, et décrivant d'autres sources de recherche que j'ai trouvées pendant le stage sans avoir eu le temps de les exploiter.

Chapitre 7

Un algorithme de filtrage générique inefficace

Pour comprendre le principe du filtrage dans `MGS`, il est important d'en connaître la source d'inspiration. Ce chapitre développe donc le mécanisme de « dérivation » de motif à l'origine du filtrage dans `MGS`.

7.1 Contexte

La première étape d'une transformation est la recherche de *chemins* à l'intérieur d'une collection filtrés par l'une des règles. Un chemin dans une collection est une suite ordonnée d'éléments de cette collection tels que deux éléments consécutifs du chemin sont voisins.

Deux points de vue peuvent en effet être notés : on peut filtrer des sous-collections ou des chemins. Le chemin permet de déterminer une sous-collection, mais il ordonne également les positions les unes par aux autres. Or, il est beaucoup plus facile de substituer (dernière étape de l'application d'une règles) un chemin par un autre dans une collection qu'une sous-collection par une autre. Le chemin est donc plus expressif que la sous-collection et a été choisi comme point de vue pour le filtrage.

Les concepteurs de `MGS` ont fait une analogie pour trouver l'ensemble des chemins d'une collection filtrés par un motif. L'algorithme de recherche est en effet inspiré de la technique de dérivation d'expressions régulières de J.A. Brzozowski [Brz64].

7.2 Dérivation d'expressions régulières

Avant tout, rappelons la grammaire d'une expression régulière

$$ExpReg ::= \ a \ \mid \ r_1.r_2 \ \mid \ r_1 + r_2 \ \mid \ r^* \ \mid \ r^+ \ \mid \ r?$$

où r, r_1 et r_2 sont des expressions régulières et a est un élément de A. L'opérateur . correspond à la concaténation, $+$ à l'alternative (r_1 *ou* r_2), * et $^+$ à une répétition et ? à un élément optionnel.

L'idée de dérivation d'une expression régulière est fondée sur la définition suivante : soit r une expression régulière sur un alphabet A et L_r le langage reconnu par r ; la dérivée de r par rapport à un symbole a de A, notée $\frac{\partial r}{\partial a}$, est définie par

$$\frac{\partial r}{\partial a} = \{m \in A^* / am \in L_r\}$$

La dérivée d'une expression régulière peut aussi s'exprimer sous la forme d'une expression régulière. La figure 7.1 en donne la définition par induction sur la construction d'une expression régulière. Par exemple

$$\frac{\partial a.(a+b)*}{\partial a} = (a+b)* \ + \emptyset$$

Ceci signifie que am est reconnu par $a(a+b)*$ si et seulement si m est reconnu par $(a+b)*$.

On trouve dans la figure 7.1 la notation $[r]$; il s'agit de l'annulateur de l'expression régulière r. Il est définit par :

$$[r] = \begin{cases} \emptyset & \text{si } \varepsilon \notin L_r \\ \{\varepsilon\} & \text{si } \varepsilon \in L_r \end{cases}$$

Autrement dit, un mot $m = a_1 a_2 ... a_n$ est reconnu par une expression régulière si et seulement si $a_2 ... a_n$ est reconnu par la dérivée de r par rapport à a_1. Ainsi, en utilisant les règles inductives de la structure d'une expression régulière r dans le calcul de la dérivée, on ramène le problème d'appartenance d'un mot à un langage à celui d'appartenance du mot vide ε à un autre langage défini par une expression régulière r'. Ce deuxième problème revient au calcul de l'annulateur de r'. Or il existe une définition inductive sur la grammaire des expressions régulières, de l'annulateur (*cf.* Figure 7.2).

Le problème de reconnaissance d'un mot par une expression régulière peut être donc récursivement sur la construction de l'expression régulière.

7.2.1 Dérivation de motifs

L'idée du filtrage dans `MGS` est d'étendre cette dérivation des expressions régulières : les motifs jouent alors un rôle similaire à celui des expressions régulières et la collection celui de l'alphabet. Il existe néanmoins quelques différences :

- Dériver une expression régulière sert à vérifier l'appartenance d'un mot à un langage. Dans notre cas, il s'agit de se préoccuper de l'existence de certains chemins dans une collection.
- Parcourir un mot se fait simplement de gauche à droite alors qu'il n'existe pas de parcours canonique général des collections. En revanche un chemin dénote une partie d'une collection ainsi qu'un parcours de cette partie.
- Les motifs de chemins peuvent contenir des expressions booléennes (liées aux gardes) dont la valeur n'est calculée que lors du filtrage.

Pour faciliter la définition de la dérivation de motif, on simplifie leur grammaire :

$$\begin{array}{lll} \mathit{Motif} & ::= & \mathit{Atome} \mid \mathit{Atome\ dir\ Motif} \\ \mathit{Atome} & ::= & \mathit{id/exp} \mid \mathit{dir}* \end{array}$$

dir correspond à « , » ou à un déplacement plus précis comme pour les GBF (par exemple |*est+nord*>). *id* correspond aux identifiants. Il existe une traduction de la grammaire des filres précédemment citée (1.3.3) dans cette nouvelle grammaire.

La dérivée d'un motif est définie de la façon suivante. Étant donné une collection C, un environnement E, et un ensemble Π des éléments de C non encore visités, la dérivée d'un motif m par rapport à une position p s'écrit $\frac{\partial m}{\partial p}(C, E, \Pi)$. Celle-ci dénote l'ensemble des chemins dans C $vois(\Pi, dir, p)$commençant par l'élément p de C, filtrés par le motif m et ne passant que par des positions de Π. E est nécessaire à l'évaluation des gardes dans m et Π permet de savoir quels éléments peuvent être filtrés dans la collection. Le résultat d'une dérivation est un ensemble de chemins. L'ensemble des chemins filtrés par un motif m dans une collection C est calculé par

$$\frac{\partial \varepsilon}{\partial \alpha} = \emptyset \tag{7.1}$$
$$\frac{\partial \emptyset}{\partial \alpha} = \emptyset \tag{7.2}$$
$$\frac{\partial \alpha}{\partial \alpha} = \varepsilon \tag{7.3}$$
$$\frac{\partial \beta}{\partial \alpha} = \emptyset \quad \text{si } \alpha \neq \beta \tag{7.4}$$
$$\frac{\partial r.r'}{\partial \alpha} = \frac{\partial r}{\partial \alpha}.r' + [r].\frac{\partial r'}{\partial \alpha} \tag{7.5}$$
$$\frac{\partial r + r'}{\partial \alpha} = \frac{\partial r}{\partial \alpha} + \frac{\partial r'}{\partial \alpha} \tag{7.6}$$
$$\frac{\partial r^*}{\partial \alpha} = \frac{\partial r}{\partial \alpha}.r^* \tag{7.7}$$
$$\frac{\partial r^+}{\partial \alpha} = \frac{\partial r}{\partial \alpha}.r^* \tag{7.8}$$
$$\frac{\partial r?}{\partial \alpha} = \frac{\partial r}{\partial \alpha} \tag{7.9}$$

FIG. 7.1 – Définition inductive de la dérivée d'une expression régulière.

$$[\varepsilon] = \varepsilon \tag{7.10}$$
$$[\emptyset] = \emptyset \tag{7.11}$$
$$[\alpha] = \emptyset \tag{7.12}$$
$$[r.r'] = [r].[r'] \tag{7.13}$$
$$[r + r'] = [r] + [r'] \tag{7.14}$$
$$[r^*] = \varepsilon \tag{7.15}$$
$$[r^+] = [r] \tag{7.16}$$
$$[r?] = \varepsilon \tag{7.17}$$

FIG. 7.2 – Définition inductive de l'annulateur d'une expression régulière

$$\bigcup_{p \in dom(C)} \frac{\partial m}{\partial p}(C, \epsilon, dom(C))$$

où ϵ est l'environnement vide et où $dom(C)$ est l'ensemble fini des éléments de C qui ont une valeur définie.

Le calcul de la dérivée est défini par une induction sur la structure du motif présentée dans la figure 7.3. La fonction $eval(E, C, e)$ donne la valeur booléenne de l'expression e dans l'environnement E et dans la collection C. La fonction $vois(\Pi, dir, p)$ donne l'ensemble des voisins de p dans C selon la direction dir et qui sont dans Π.

On représente un chemin par une liste de *positions*. Une position permet de se repérer dans une collection topologique. L'indice des éléments d'un tableau ou d'une matrice est une position pour ces topologies. La notion de position est très dépendante de la topologie ; voici une liste de correspondance entre position et topologie :

- Pour les GBF, il s'agit simplement d'un élément du groupe. On peut se repérer dans un GBF en observant le déplacement à partir d'un point d'origine choisi arbitrairement. Un élément du groupe est représenté par la forme normale de SMITH d'un mot représentant les éléments.
- Pour les séquences, un entier suffit pour repérer la position des éléments dans la liste.
- Pour les ensembles, la position est l'élément lui-même.
- Dans les multi-ensembles, on associe à chaque occurrence d'un élément un entier. Ainsi, on considérera comme position le couple (élément, indice d'occurrence).
- Pour les graphes en général, on numérote de façon aléatoire chaque sommet. On utilise cette numérotation pour les positions.

Voici la définition des quelques notations introduites dans la figure 7.3 :

- [] est la liste vide,
- {...} dénote un ensemble,
- $\emptyset$ est l'ensemble vide,
- $E + [id \mapsto p]$ est une modification de la définition de l'environnement E qui associe maintenant la position p à l'identifiant id.

Les équations de cette figure s'interprètent de la façon suivante :

18-19 : Seul le chemin vide peut être filtré dans une collection vide.

20 : `x/e` filtre le chemin $[p]$ à partir de p si et seulement si `e` est vérifié dans E augmenté de la liaison `x`$\mapsto p$.

21 : Les chemins filtrés par `d*` depuis p sont le chemin vide et les chemins filtrés par `x/true d d*` depuis p (`x` est une variable fraîche).

22 : Les chemins filtrés par `(x/e) d M` en p si `e` est vérifiée sont les chemins filtrés par `M` à partir des voisins de p selon la direction `d` auxquels on a ajouté p en tête. p étant maintenant utilisé et n'étant donc plus disponible, il est retiré de l'ensemble Π.

23 : Les chemins filtrés par `d* d' M` sont les chemins filtrés par `M` à partir des voisins de p selon `d` (répétition nulle) et les chemins filtrés par `(x/true) d d* d' M` (répétition non nulle).

7.2.2 Un algorithme peu efficace

Les définitions de la figure 7.3 constituent un algorithme qui spécifie tous les chemins d'une collection filtrés par un motif m. Cepdendant, cet algorithme ne peut pas être utilisé sous cette forme pour une implantation efficace des transformations. En effet, il permet de trouver tous les chemins à partir d'un position donnée ; or, `MGS` n'en requiert qu'un seul. En effet, si deux

$$\frac{\partial dir*}{\partial p}(C,E,\emptyset) = \{[]\} \tag{7.18}$$

$$\frac{\partial M}{\partial p}(C,E,\emptyset) = \emptyset \quad \text{avec } M \neq dir* \tag{7.19}$$

$$\frac{\partial id/expr}{\partial p}(C,E,\Pi) = \text{if } eval(E+[id\mapsto p],C,expr) \text{ then } \{[p]\} \text{ else } \emptyset \tag{7.20}$$

$$\frac{\partial dir*}{\partial p}(C,E,\Pi) = \{[]\} \cup \frac{\partial (id/true\ dir\ dir*)}{\partial p}(C,E,\Pi) \quad \text{où } id \text{ est une variable fraîche} \tag{7.21}$$

$$\begin{aligned}\frac{\partial id/expr\ dir\ M}{\partial p}(C,E,\Pi) = {} & \text{let } E' = E+[id\mapsto p] \text{ and } \Pi' = \Pi - p \text{ in} \\ & \text{if } eval(E',C,expr) \\ & \text{then } p \otimes \left(\bigcup_{p'\in vois(\Pi',dir,p)} \frac{\partial M}{\partial p'}(C,E',\Pi') \right) \\ & \text{else } \emptyset\end{aligned} \tag{7.22}$$

$$\begin{aligned}\frac{\partial dir*dir'M}{\partial p}(C,E,\Pi) = {} & \left(\bigcup_{p'\in vois(\Pi,dir',p)} \frac{\partial M}{\partial p'}(C,E,\Pi) \right) \\ & \cup \frac{\partial id/true\ dir\ dir*\ dir'\ M}{\partial p}(C,E,\Pi) \\ & \text{où } id \text{ est une variable fraîche}\end{aligned} \tag{7.23}$$

FIG. 7.3 – Calcul de la dérivée d'un motif. (On suppose dans les équations 3 à 6 que $\Pi \neq \emptyset$.)

chemins sont trouvés et qu'ils partagent le premier élément, il faut donc choisir celui qui va être transformé. Il faut donc mettre au point un algorithme qui ne fasse pas une recherche exhaustive.

Cependant, l'une des propriétés de cet algorithme est que si lors de l'application de la transformation on ne trouve plus de chemin qui puisse être filtré par le motif, c'est qu'il n'en existe pas. Pour cela, l'algorithme doit échouer à partir de toutes les positions de la collection. Ainsi, pour qu'un pas de transformation se termine, il faut faire cette vérification qui dans le pire des cas parcourt la collection entière.

Soient n le nombre d'élément dans la collection, ν le nombre maximum de voisins pour chaque position, et m la taille du chemin devant être filtré. Le nombre de comparaison atomique va être au pire de l'ordre de $n\nu^m$.

Ainsi, bien que le principe de dérivation soit très puissant puisqu'il autorise une généricité du filtrage, il ne peut pas être utilisé tel quel dans `MGS`.

Un des objectifs du projet `MGS` est de développer un traitement uniforme des collections, de la manière la plus indépendante possible de la topologie. De ce point de vue, la conception de cet algorithme général est un succès qui valide le postulat de départ. Mais dans la pratique, on perd les propriétés inhérentes à chaque type de collection. Par exemple, il serait plus efficace d'utiliser des algorithmes de recherche d'expressions régulières dans des textes pour les séquences. De même, l'utilisation des algorithmes de résolutions de requêtes dans les bases de données XML, est plus adapté à la recherche de motifs dans des arbres (les arbres peuvent être construits comme

des GBF libres).

Notre idée est donc de classer les motifs suivant les algorithmes de recherche connus et, en fonction de la topologie et du motif, choisir l'algorithme de complexité la plus faible.

Nous allons commencer par étudier les motifs de façon indépendante des topologies. En effet, certains motifs sont, *a priori*, moins coûteux que d'autres. Par exemple, « $x, y/y > 0$ » est moins coûteux que « $x, y/y > x$ » ; en effet, pour le premier, on n'a pas besoin d'observer la valeur de x pour choisir y tandis que dans le second, y dépend de la valeur de x.

On peut remarquer qu'une des constructions des motifs `MGS` est une nouveauté par rapport aux sources d'inspiration : il s'agit de l'itération. La répétition d'un motif un nombre arbitraire de fois est, par exemple, absente du formalisme CHAM et du langage Gamma. Ce nouvel élément apporte des difficultés supplémentaires puisque, intuitivement, un élément itéré (et donc filtrant des chemins de longueur non-bornée) est plus coûteux à filtrer qu'un motif de longueur bornée. J'ai décidé de traiter d'abord les motifs non-étoilés, puis d'essayer de rajouter après la notion de répétition de motif, pour séparer les difficultés.

Avant cela, je vais décrire la traduction d'un terme représentant un filtre à une forme linéaire. Cette forme est celle utilisée par une variante "on-line" de l'algorithme précédent. L'avantage de cette variante est qu'elle offre un algorithme de filtrage exhaustif qui n'énumère pas exhaustivement l'ensemble des chemins à filtrer.

Chapitre 8

Séquencialisation des filtres

L'algorithme de dérivation de motif souffre de deux défauts importants :

1. il modifie la structure du motif. On peut prendre par exemple le cas de l'itération d'un motif p : l'algorithme transforme le filtrage de p^* en $(p, p^*)|\epsilon$. C'est à travers ces modifications successives que l'algorithme progresse. Le problème vient du fait que cette transformation est potentiellement lourde à l'implémentation.
2. il recherche de façon exhaustive tous les chemins qui sont filtrés par le motif. Or, deux chemins ne peuvent avoir de position en commun. En effet, la transformation du premier chemin pourrait modifier le second. Il est donc inutile de chercher tous les chemins à partir d'une même position, un seul suffit.

Une autre vision apparaît comme plus adaptée : cette nouvelle approche consiste à se déplacer dans une collection, à partir d'une position et en énumérant les éléments du motif. On peut comparer cela à une *tortue*[1] qui se déplace de position en position dans la collection, considérée comme un graphe orienté quelconque, et dont le parcours est imposé par le motif.

Le problème fondamental est de pouvoir mémoriser le chemin emprunté pour pouvoir revenir en arrière en cas d'échec. En effet, lors d'un déplacement, il se peut que plusieurs directions soient possibles (d'un point de vue local). La tortue doit alors choisir ; si elle suit un chemin qui n'aboutit pas, elle va devoir revenir à cette intersection pour choisir une autre voie. Il lui faut donc marquer l'intersection pour pouvoir y revenir en cas d'échec.

L'algorithme se présente naturellement sous une forme itérative. Après en avoir fait une présentation, nous observerons un aspect essentiel de son fonctionnement, la séquencialisation des motifs, et son implémentation dans `MGS`.

8.1 Constructeur des motifs et notations

La grammaire de filtre du langage `MGS` décrit précedemment correspond à la syntaxe dans laquelle l'utilisateur écrira les motifs. Un filtre est composé de plusieurs éléments unitaires. Pour plus de clarté, j'ai désigné par un certain nombre de notations ces différents éléments :

- *CtePat* : élément filtrant une position dont on impose la valeur
- *RecordPat* : élément filtrant des enregistrements `MGS` ayant certaines propriétés
- *BlankPat* : élément filtrant n'importe quelle position
- *CommaPat* : déplacement selon une direction suivant la relation de voisinage dans la collection

[1] Le terme de tortue vient du langage *Logo* où un programme pilote le déplacement d'un "tortue" dans un plan euclidien.

```
Algo Filtre ( motif : pattern_table, coll : mgs_collection, depart : 'position)
  Début
    Soit Placer le motif en depart dans coll ;
    Soit marques = [] ;
    TantQue motif n'est pas entièrement parcouru Faire
      Rattrape
        Lire motif et déplacer la tortue dans coll
        Si il y a plusieurs choix possibles Alors
          empiler une marque dans marques
        FinSi
      Avec
        | On ne peut pas déplacer la tortue →
          Si marque est vide Alors
            /* aucun chemin trouvé */
          Sinon
            On revient à la dernière marque empilée dans marques
          FinSi
      FinRattrape
    FinTantQue
  Fin
```

FIG. 8.1 – Algorithme itératif de recherche d'un chemin

- *GuardedPat* : garde devant être évaluée à `vrai` pour accepter le motif
- *AsPat* : enregistrement de l'environnement une association entre identifiant et valeur
- *UpdatePat* : mise à jour de l'environnement qui associe aux identifiants définit dans le motif (par les *AsPat*) la valeur correspondante (*cf.* 8.4)
- *OrPat* : alternative entre deux motifs
- *IterPat* : itération d'un motif

8.2 Un algorithme de recherche itératif

L'idée générale de l'algorithme est de parcourir la collection suivant les directions indiquées par le motif tout en empilant des marques dans une pile, lors de choix dans les déplacements. Aussitôt qu'il n'y a plus de déplacement possible, on dépile une marque pour choisir une autre voie. S'il n'y a plus de marque, c'est qu'aucun chemin à partir de la position initiale n'est filtré par le motif. Une marque permet de savoir quelles sont les directions qui n'ont pas encore été visitées. Si la position courante a ν voisins, on en choisit un et on inscrit dans la marque qu'il reste les $\nu - 1$ autres voisins à visiter. L'algorithme est présenté dans la figure 8.1.

Les paramètres de cet algorithme sont au nombre de 3 :

1. *motif* : il s'agit du motif à filtrer.
2. *coll* est la collection dans laquelle la recherche est faite. Son type `mgs_collection`, correspond au type générique des collections de `MGS`.
3. *depart* : il s'agit de la position à partir de laquelle on va commencer la recherche. Pour obtenir tous les chemins de *coll* qui peuvent être filtrés pat *motif*, il faut relancer l'algorithme pour chaque position de départ différente.

Remarque Il est tout à fait intéressant d'observer l'absence totale d'implication du motif au moment du marquage. En fait celui-ci ne dépend que de la présence de plusieurs voisins et

donc d'un choix dans le déplacement. Il apparaît alors que la topologie de la collection est plus importante que la forme du motif par rapport à la complexité de l'algorithme.

Aussi, une optimisation n'est possible que par la spécialisation de l'algorithme par rapport au type de collection et indépendamment du motif. Le motif reste important puisqu'il dirige la tortue tout en validant si un élément est filtré ou non.

8.3 L'applatissement des motifs

La lecture du motif est le point essentiel de cet algorithme de recherche. Le moyen le plus simple de le lire est de le transformer en une séquence d'atomes ; on parle alors d'applatissment puisqu'on perd l'arbre de dérivation issu de l'analyse syntaxique du motif.

On peut en fait transformer un motif en une séquence :
- d'atomes : il s'agit de motif unitaire qui filtre un seul élément (*CtePat*, *BlankPat* et *RecordPat*)
- de directions : il s'agit des déplacements élémentaires qui agissent sur la tortue (*CommaPat*).

Lorsqu'un filtre est applati, il suit le schéma suivant qu'il faut respecter :

Atome	Direction	Atome	...	Direction	Atome

On peut également rajouter dans cette séquence les gardes qui sont des éléments qui ne filtrent pas de position et qui ne déplacent pas la tortue ; leur rôle est de conditionner la continuation de l'algorithme.

Pour éviter tous les problèmes inhérents à leur présence (notamment dans le cas des répétitions), nous avons éliminé les *AsPat* qui permettent de nommer des sous-chemins filtrés. Une étape de la construction de cette séquence consiste donc à supprimer les *AsPat* en faisant porter l'information de l'identifiant sur chaque constructeur.

La répétition est un élément particulier de cette séquence en ce qu'il porte lui-même une sous-séquence. Sans entrer dans les détails, l'algorithme de filtrage est relancé (appel récursif) en prenant comme paramètre le sous-motif, la même collection et comme position initiale, la position courante. La gestion de l'itération subit actuellement des changements assez conséquents ; on essaiera de se détacher le plus possible de ce point de vue algorithmique dans la suite du rapport. Il va sans dire que les apports concernants l'itération n'ont pas pu être intégrés à la version actuellement en développement de `MGS`. Cependant, ils ont pu être vérifiés sur une version antérieure.

8.4 Un algorithme de recherche récursif

Du point de vue de l'implémentation, l'algorithme itératif est compliqué et trop peu évolutif pour être inséré dans un projet dont le code doit être très souple puisqu'il est souvent modifié.

Le langage `MGS` a été développé suivant une organisation de classe (dans le sens d'une programmation orientée objet) qui suit les principes mis en avant dans le projet.

On souhaite associer le filtrage au motif ; en effet, on veut avoir un algorithme générique. Le filtrage est conceptuellement indépendant des topologies et suit l'algorithme présent dès l'écriture du motif. Par exemple, le motif $1, 2, x^*$ correspond à l'algorithme de filtrage *« je filtre un motif dont le premier élément a pour valeur* 1*, le deuxième* 2*, suivit de n'importe quel chemin »* qui s'applique sur n'importe quelle collection topologique.

L'association du filtrage et du motif se fait par la représentation des motifs par un objet ayant une méthode `filtre`.

Ce qui permet d'obtenir un code plus souple, peut être décrit par l'exemple suivant : on reprend le même motif que précédemment $m = 1, 2, x^*$. Filtrer m à partir d'une position courante p revient à vérifier que 1 filtre l'élément à la position p puis à filtrer $m' = , 2, x^*$ à partir de p. De la même façon, filtrer m' à partir de p revient à se déplacer dans n'importe quelle position p' voisine de p, puis à filtrer le motif $m'' = 2, x^*$ à partir de p'. On continue ce développement ainsi de suite jusqu'à atteindre le motif vide ϵ qui annonce la fin du filtrage.

En utilisant ce développement récursif adapté à la séquencialisation du motif, on sépare le code servant à chaque type d'élément unitaire du motif applati. De cette façon, on peut faire évoluer par exemple le filtrage de l'étoile sans modifier les autres constructions.

Description de la hierarchie de classe. La classe mère de tous les motifs est `pattern` ; voici un extrait de sa déclaration en *OCaml* :

```
class virtual ['position] pattern (patvar : int list) = object(self)
  ...
  method virtual filtre : 'position generic_topo
                       -> 'position return_path
  ...
end ; ;
```

Plusieurs points sont à noter :

- La classe `pattern` est paramétrée par le type `'position` ; on utilise des types différents pour représenter une position suivant la topologie. Ce paramètre permet de statisfaire la généricité du motif qui peut être utilisé avec n'importe quel type de position.
- La variable `patvar` correspond à l'ensemble des identifiants qui permettent de nommer des sous-chemins filtrés par le pattern. Elle a été créée lors de la suppression des *AsPat*. Soit le motif 1 *as* x *as* y qui filtre un élément de valeur 1 auquel on va associer les noms x et y. La variable `patvar` aura dans ce cas la valeur $[x; y]$.
- la méthode filtre est virtuelle. En effet, son implémentation dépend du type de filtre (s'il s'agit d'un *CtePat* ou d'un *BlankPat*). Elle prend en argument un objet de la classe `'position generic_topo` qui représente une collection topologique *générique*. Il s'agit en fait de la classe mère des topologies. Elle retourne un chemin de position dont le type est `'position return_path`. On remarque bien la dépendance au type des positions et la généricité de la classe `pattern`.

A partir de cette classe mère, des sous-classes vont être implémentées pour chacun des éléments de la séquence du motif. Lors de l'intanciation de ces classes, les objets vont être chaînés les uns aux autres. Voici un exemple de déclaration d'une de ces sous-classes ; il s'agit de la classe `blank_pattern` qui filtre un élément de valeur quelconque :

```
class ['position] blank_pattern patvar next = object(self)
  inherit ['position] linked_pattern patvar true next
  ...
  method filtre topo =
    begin
      (* implémentation du filtre d'un élément quelconque *)
    end
end ; ;
```

Quelques remarques :

- On retrouve à nouveau le paramètre `'position` qui permet la généricité des motifs.

– Le paramètre `next` est un objet de la classe `pattern` ; il correspond à la continuation du motif.
C'est grâce à ce champ que le développement est fait : si on reprend l'exemple $1, 2, x^*$, on construit un objet de classe `cte_pattern` filtrant les éléments de valeur 1, et dont le champ `next` est un objet de classe `pattern` qui filtre le motif $, 2, x^*$. Ce champ `next` n'est autre qu'un objet de la classe `comma_pattern` qui déplace la tortue de la position courante à l'un des voisins ; Son champ `next` est un objet de classe `pattern` qui filtre $2, x^*$.
Ainsi, `next` permet de faire le chaînage des éléments du motif séquencialisé. La gestion du chaînage est assurée par la classe `linked_pattern` qui hérite directement de la classe `pattern`.
– Voici la liste des classes élémentaires constituant les motifs :

`cte_pattern` : filtre les éléments d'une valeur donnée,

`blank_pattern` : filtre les éléments de valeur quelconque,

`record_pattern` : filtre les éléments étant des enregistrements `MGS` vérifiant certaines propriétés,

`comma_pattern` : permet les déplacements de la tortue,

`guarded_pattern` : vérifie qu'une garde est évaluée à *vrai*,

`update_pattern` : permet de mettre à jour l'environnement (qui associe les identifiants aux sous-chemins auxquels ils correspondent). Ils apparaissent dès qu'on souhaite nommer deux sous-motifs liés par une virgule ; prenons comme exemple $(p\ as\ P,\ p'\ as\ P')\ as\ Q$, où p et p' sont des sous-motifs et P et P' leurs identifiants respectifs. Un objet de la classe `update_pattern` met à jour l'environnement en associant à Q la concaténation des sous-chemins identifiés par P et P'. Si on note E l'environnement, cela revient à effectuer $E.(Q) \leftarrow Union\ E.(P)\ E.(P')$.

`or_pattern` : permet de faire une disjonction dans le motif

`iter_pattern` : correspond à l'itération d'un sous-motif

Toutes ces classes héritent de `linked_pattern` et donc de `pattern`.

Bien entendu, il existe une classe terminale qui représente en fait le motif ϵ ; il s'agit de la classe `end_pattern` qui hérite de `pattern`. La figure 8.2 présente la hiérarchie des classes.

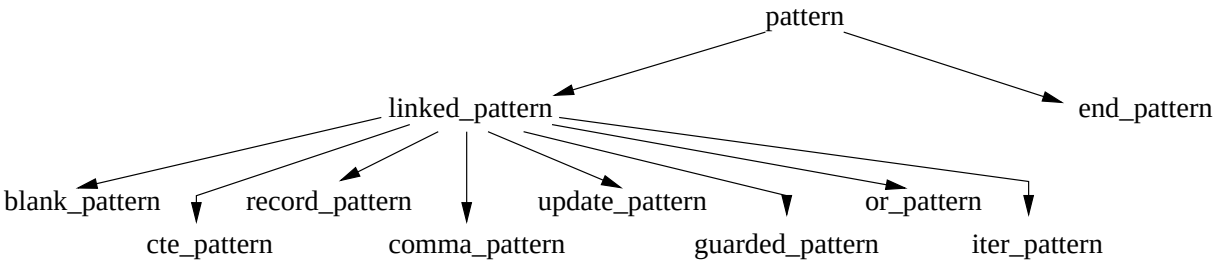


FIG. 8.2 – Hiérarchie des classes définissant les motifs

Transformations successives des motifs. C'est dans ces transformations successives qu'on va pouvoir insérer des optimisations si cela est possible. La figure 8.3 présente les transformations successives subies par le motif.

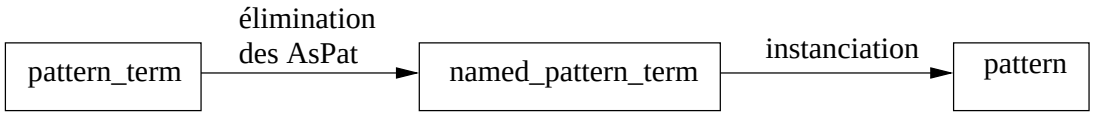


FIG. 8.3 – Transformations successives d'un motif

8.5 Conclusion

Les traitements successifs d'un motif permettent de le transformer d'un arbre de dérivation à un objet de classe `pattern` possédant une méthode `filtre` implémentant le filtrage. Cet objet cache en fait une séquence d'objets qui filtrent un élément unitaire du chemin et qui passe ensuite le contrôle au voisin qui fait de même, et ainsi de suite jusqu'au bout de cette liste. Cette séquencialisation du motif permet notamment d'avoir un algorithme qui ne travaille plus par manipulation du motif lui même, mais qui s'en sert comme d'un plan de route pour se déplacer dans la collection. Elle permet également de séparer le code du filtrage en petites sous-parties indépendantes, ce qui est très avantageux dans un projet tel que `MGS`.

Les objets de type `pattern` ne possèdent aucun champ mutable (*i.e.* modifiable après la construction). Par conséquent, les paramètres qui leur sont passés dès l'instanciation ne peuvent être modifiés. Ceci permet d'être sûr que rien ne peut être modifié par des effets de bord non désirés lors du fitrage. Aussi, si une optimisation demande la réécriture d'un motif, celle-ci devra se faire avant l'instanciation. La figure 8.3 révélera son importance dans le chapitre 10, traitant de l'optimisation.

Chapitre 9

Coût et classification des filtres

Maintenant que nous possédons un algorithme générique complet répondant exactement aux exigences posées par les principes de MGS, il faut étudier sa complexité. Comme nous l'avons déjà dit, celle-ci est très dépendante de la topologie (au niveau de la pile des marques). Cependant, en pratique, une transformation sera souvent liée à la collection sur laquelle elle est appliquée ; il n'y a pas de raison de comparer les performances de l'algorithme sur un même motif mais appliqué à deux collections différentes.

Par contre, il est possible de comparer deux motifs. Ceci peut permettre par exemple de choisir entre deux motifs équivalents celui qu'il faut garder, ou encore de privilégier parmi plusieurs motifs le moins coûteux.

Ce chapitre propose une classification des motifs suivant un algorithme fondé sur la *reconnaissance* (et non la recherche) des chemins ; elle cherche en fait à déterminer la machine minimale (automate fini déterministe, non-déterministe, automate à pile, etc.) qui permet de résoudre le problème de reconnaissance. Elle associe aux motifs une classe correspondant au type de machine qu'il faut utiliser. Ces classes ou types de motifs peuvent être comparés.

9.1 Inspiration

Cette première classification utilise une conséquence de la définition des transformations et de la notion de filtrage dans MGS. En effet, *la seule propriété assurée* pour l'application des règles d'une transformation, en terme de stratégie d'application, est que si aucune règle ne s'applique sur une collection C, alors il n'existe aucun chemin dans C pouvant être filtré par la partie gauche des règles de la transformation.

Un algorithme naïf assurant cette propriété serait de générer tous les chemins et d'exécuter un algorithme de reconnaissance utilisant des automates équivalents aux motifs contenus dans la partie gauche des règles.

On sait générer tous les chemins d'une collection : pour cela, on commence par les chemins de longueurs 1 ; on construit ensuite les chemins de longueurs n à partir des chemins de longueurs $n-1$ en ajoutant les voisins des éléments des extrémités de ces derniers. Le nombre d'éléments dans une collection étant finis, le nombre de chemins l'est aussi.

Il va falloir maintenant construire les automates qui vont permettre de déterminer si un chemin est filtré par un motif. C'est grâce à cet automate que nous classifierons les motifs ; en fait, on va chercher à construire la machine *de complexité minimale* qui va permettre cette reconnaissance.

Il va de soi que cet algorithme est le pire que l'on puisse imaginer : il génère exhaustivement tous les chemins alors que seul un petit sous-ensemble est pertinent. Néanmoins, il n'a pas pour vocation d'être implémenté mais juste utilisé pour déterminer les classes de complexité des motifs. L'idée de cet algorithme est inspirée de l'utilisation des automates pour savoir si un mot appartient au langage associé à une expression régulière. Naturellement, cette façon de procéder va correspondre à un coût "au pire". Ceci n'a cependant pas trop d'importance puisqu'il vaut mieux surévalué plutôt que le contraire.

9.2 Automate de reconnaissance de chemin

La machine la plus simple à construire est l'automate fini déterministe. On va donc réduire la structure des motifs pour que ces automates soient suffisants pour les représenter.

Les motifs construits avec uniquement des $CtePat$ et des $CommaPat$ peuvent être représentés par un AFD (automate fini déterministe). Les AFD ont la propriété de n'utiliser aucune mémoire pour fonctionner et de reconnaître des chemins en un temps linéaire avec la taille du chemin.

La construction est la suivante :

- $CtePat(v)$ est représenté par un automate à deux états (un initial et un final) et une transition valuée par la valeur v. Si le chemin est constitué d'un seul élément de valeur v alors il est reconnu (*cf.* Figure 9.1).

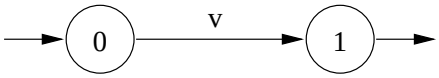


FIG. 9.1 – Automate représentant $CtePat(v)$

- $CommaPat(p_1, p_2)$: on construit l'automate $\mathcal{A}_1$ reconnaissant p_1 et l'automate $\mathcal{A}_2$ reconnaissant p_2, et on les « concatène » : pour cela, on fusionne l'état final de $\mathcal{A}_1$ avec l'état initial de $\mathcal{A}_2$ (*cf.* Figure 9.2).

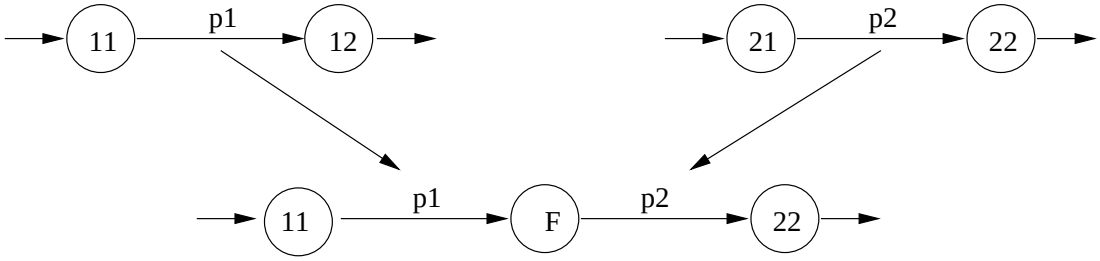


FIG. 9.2 – Automate représentant $CommaPat(p_1, p_2)$

On peut prouver par induction, que tous les automates formés par ces deux règles possèdent un seul état initial et un seul état final :

- la propriété est vrai pour le cas de base $CtePat$
- si on suppose que $\mathcal{A}_1$ et $\mathcal{A}_2$ reconnaissant respectivement p_1 et p_2, vérifient cette propriété, alors on peut fusionner l'état final de $\mathcal{A}_1$ avec l'état initial de $\mathcal{A}_2$ sans ambiguïté. L'automate résultant possède alors un unique état initial, celui de $\mathcal{A}_1$, et un unique état final, celui de $\mathcal{A}_2$. Cet automate reconnaît $CommaPat(p_1, p_2)$.

Nous allons essayer de compléter le plus possible la structure en conservant les propriété de complexité des AFD.

Introduction de *BlankPat*

On peut ajouter la construction *BlankPat* : ce motif permet de reconnaître n'importe quelle valeur de type *mgs_value*. L'automate associé est identique à celui de *CtePat* à ceci près que l'arc n'est pas valué ; le passage d'un état à l'autre se fait par la reconnaissance de n'importe quelle valeur. La complexité de l'automate ne change pas.

Introduction de *GuardedPat*

On peut distinguer deux sortes de garde qui vont avoir une incidence sur la complexité de l'automate.

- Certaines gardes qui s'appliquent sur des sous-chemins de longueur 1, n'impliquent pas de dépendance avec le reste du chemin. C'est le cas par exemple pour $x, y/y > 0$; peu importe la valeur de x tant que y est positif.
- Les autres gardes impliquent des dépendances en arrière. Dans $x, y/y > x$, la valeur de y dépend de celle de x.

Le premier cas est identique à la construction *CtePat*. L'automate associé à ce type de garde est simple : on reprend le même que celui utilisé pour $CtePat(v)$ dans lequel la transition n'est plus calculée par v mais par la garde. La transition ne sera autorisée que si la valeur du chemin vérifie le prédicat de la garde. On conserve toutes les propriétés déjà signalées des AFD.

Les autres gardes nous font sortir légèrement du cadre des AFD tels qu'on les manipule habituellement. En effet, l'automate représentant $GuardedPat(p, g)$ est exactement le même que celui de p, à une différence près : l'état final doit évaluer à *vrai* la garde g, pour que le chemin soit reconnu par le motif. Cette différence de comportement s'insère parfaitement à l'intérieur des AFD déjà vus. La fusion d'un état initial d'un automate $\mathcal{A}_2$ avec l'état final gardé d'un automate $\mathcal{A}_1$ se fait aussi naturellement : dans l'état résultant, on n'acceptera, pour un chemin donné, de poursuivre la reconnaissance si et seulement si la garde g est vérifiée.

Le problème vient du fait que les gardes font référence à des éléments déjà filtrés par le motif mais auxquels on impose de vérifier la garde g. Aussi, il est important d'associer, au fur et à mesure de la reconnaissance, les noms des identifiants aux sous-chemins qu'ils désignent. Pour cela, on a besoin d'espace mémoire et de rajouter quelques instructions impératives au cours du filtrage :

- on crée pour cela un environnement E qui n'est autre qu'une fonction associant des identifiants à des sous-chemins ;
- à chaque fois qu'on rencontre lors de la reconnaissance du chemin un *AsPat*, on insère une correspondance dans l'environnement E ;
- pour évaluer une garde g, on substitue chaque identifiant par sa valeur dans E.

La création de ce nouvel environnement modifie la complexité en espace de l'algorithme. Néanmoins, on connaît après analyse du filtre la place que prendra E en mémoire. Une classe de motif apparaît à travers ce changement où la reconnaissance se fait en temps linéaire sur un espace borné.

Introduction de *OrPat*

Pour terminer, il nous reste à regarder le cas du *OrPat*. Pour construire l'automate associé, on construit les automates reconnaissant chacun des motifs de la disjonction pour ensuite en faire l'union. Le problème est que cette méthode amène à construire un automate pouvant être non-déterministe.

Néanmoins, dans une implantation possible de l'algorithme, la marche à suivre sera la suivante : pour un chemin donné, on exécute l'algorithme de reconnaissance avec le premier motif et s'il échoue, on essaiera le second (retour-arrière). Aussi, de l'espace mémoire sera utilisé (en plus de l'éventuel environnement E) : la pile sera utilisée pour gérer les retours-arrière. L'espace pris est borné en fonction du nombre de $OrPat$ présent dans le motif.

Le temps mis pour reconnaître un chemin ne sera plus linéaire avec la taille du chemin. En effet, des retours en arrière dus au possible échec sur la branche du $OrPat$ appelée en premier vont nous obliger à revenir au choix pour repartir vers la branche restante et recommencer le filtrage. La présence d'un $OrPat$ dans un motif détermine alors une nouvelle classe de complexité.

Une autre possibilité serait de réécrire les motifs pour éliminer les $OrPat$ au prix d'une augmentation du nombre de règles. Par exemple :

$$x, (a|b), y \Rightarrow expr \text{ devient } \begin{cases} x, a, y \Rightarrow expr \\ x, b, y \Rightarrow expr \end{cases}$$

Cette solution convient plus à la construction directe d'automates déterministes. En effet, une transformation est composée de plusieurs règles ; aussi, il faudra de toute façon filtrer le chemin à travers les automates des filtres de chaque règle pour savoir laquelle convient. Là où l'automate de $x, (a|b), y$ n'est pas déterministe, ceux de x, a, y et x, b, y le sont. Il suffit donc de les construire séparément et de les utiliser comme s'il s'agissait de deux règles différentes.

Il n'y a plus de pile d'exécution ici mais les deux méthodes sont de complexité identique ; pour les deux, une analyse *a priori* du motif (*i.e.* sans se préoccuper des chemins ou des sortes de collections) suffit à déterminer la complexité. On préférera néanmoins utiliser la seconde solution plus proche de travaux déjà en cours, comme par exemple dans le projet ELAN.

Introduction de *IterPat*

Comme nous l'avons déjà dit, ce genre de répétition est une nouveauté apportée par `MGS`. Un $IterPat$ est considéré de la même façon que l'étoile des expressions régulières ; en fait, il s'agit d'un $OrPat$ infini :

$$p^* \ = \ \epsilon \mid p.p^* \ = \ \epsilon \mid p \mid p^2 \mid p^3 \mid ...$$

où p^n signifie que le motif p est répété exactement n fois. Le rajout d'un $IterPat$ remet en cause entièrement les différentes classes de complexité définies jusqu'à présent :

- Si une nouvelle variable est introduite par un $AsPat$ pour identifier un sous-motif contenant une répétition, on ne peut plus déterminer la taille de E juste en observant le motif. La taille de E n'est plus bornée.
- On considère $IterPat$ comme une infinité de $OrPat$. On ne peut pas réécrire la règle comme nous le proposions pour les $OrPat$ simples :

$$p* \Rightarrow expr \text{ deviendrait } \begin{cases} \epsilon & \Rightarrow expr \\ p & \Rightarrow expr \\ p^2 & \Rightarrow expr \\ p^3 & \Rightarrow expr \\ ... & \end{cases}$$

  ce qui produirait une infinité de règles. Il faut donc considérer une pile d'exécution de taille non-bornée qui enregistrerait les boucles successives. La figure 9.3 montre l'utilisation d'une telle pile sur un exemple. On passe alors d'un automate fini déterministe à un automate non-déterministe à pile. On ne peut pas déterminer la complexité de la reconnaissance par la seule observation du motif.

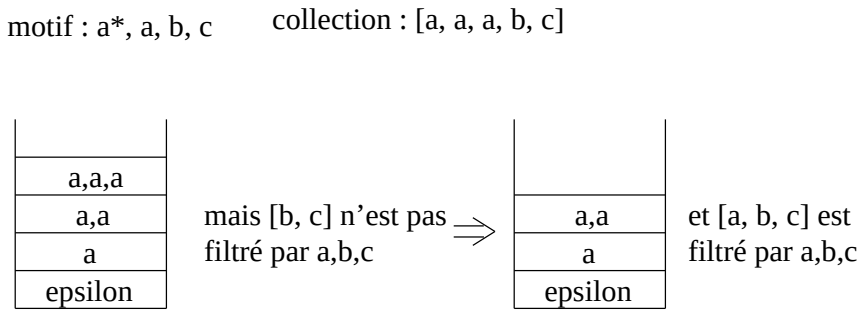


FIG. 9.3 – Exemple d'utilisation d'une pile pour gérer une itération

On peut alors créer une nouvelle classe correspondant à la présence d'une étoile dans un motif.

9.3 Classification

Les automates de reconnaissance ont été définis de façon informelle dans le paragraphe précédent. Nous allons élaborer à partir de cette description, un algorithme utilisant la construction inductive des motifs pour définir formellement les classes de complexité.

Pour commencer, il est important de distinguer les différents éléments qui sont mis en jeu et qui modifient la complexité de l'automate de reconnaissance. Il y en a deux :

1. L'environnement E sert à associer les identifiants aux sous-chemins filtrés. Sa taille varie selon l'automate. Soit il est vide si aucun $AsPat$ n'apparaît dans le filtre, soit il est de taille non-bornée en présnce d'itération, et bornée dans touts les autres cas.
2. La pile d'exécution est utilisée lorsqu'une répétition est présente dans le filtre. On ne peut pas déterminer sa taille à l'avance.

La complexité en temps est linéaire en la taille du chemin filtré lorsqu'il n'y a ni $OrPat$, ni $IterPat$ dans le motif.

On va pouvoir décrire les classes de complexité sous la forme d'un triplet donnant la complexité en temps, la complexité en espace due à E et la complexité en espace due à la pile d'exécution.

$$\begin{array}{rcl} tps\_cpl & ::= & \text{Lineaire} \mid \text{NonLineaire} \\ env\_cpl & ::= & \text{Rien} \mid \text{Borne} \mid \text{NonBorne} \\ pile\_cpl & ::= & \text{Absente} \mid \text{Presente} \end{array}$$

Une relation d'ordre existe pour chacun de ces types :

- Lineaire $<$ NonLineaire
- Rien $<$ Borne $<$ NonBorne
- Presente $<$ Absente

On définit alors les fonctions maxTpsCpl, maxEnvCpl et maxPileCpl qui retournent le maximum entre deux complexités. On définit ensuite le type des classes et la fonction maxClasse de la façon suivante :

$$\begin{array}{l} classe\_cpl ::= (tps\_cpl, env\_cpl, pile\_cpl) \\ \text{maxClasse } (t_1, e_1, p_1)\ (t_2, e_2, p_2) = \\ \quad (\text{maxTpsCpl } t_1\ t_2,\ \text{maxEnvCpl } e_1\ e2,\ \text{maxPileCpl } p_1\ p_2) \end{array}$$

Il ne reste plus qu'à écrire l'algorithme de classification qui va renvoyer une classe de type $classe\_cpl$ à partir d'un motif. Voici la définition inductive sur la construction d'un motif de la classification :

p	$classeDe(p)$
$BlankPat$	(Lineaire, Rien, Absente)
$CtePat$	(Lineaire, Rien, Absente)
$RecordPat$	(Lineaire, Rien, Absente)
$AsPat(id, p')$	$classeDe(p')$
$CommaPat(p_1, p_2)$	$maxClasse\ (classeDe(p_1), classeDe(p_2))$
$OrPat(p_1, p_2)$	Soit $m = maxClasse\ (classeDe(p_1), classeDe(p_2))$ $maxClasse\ m$ (NonLineaire, Rien, Absente)
$IterPat(p')$	$maxClasse\ (classeDe(p'))$ (NonLineaire, Rien, Presente)
$GuardedPat(p', g)$	Soit $(t, e, s) = m = classeDe(p')$ Si p' est de longueur 1 et $g$ne dépend que de p' alors m Si g dépend d'une itération alors $(t$, NonBorne, $s)$ Sinon $(t$, Borne, $s)$

Dans cette définition, p', p_1 et p_2 dénotent des sous-motifs, id un identifiant et g une garde.

Le tableau suivant présente quelques exemples de motifs et leurs complexités. La complexité dépendant de la pile n'apparaît pas :

		Temps	
		`Lineaire`	`NonLineaire`
E	`Rien`	$1, \_, 2$	$1, (2\|3), 4$
	`Borne`	$(1, \_, 2)\ as\ x/P(x)$	$(\_, 1)\ as\ x, y/P(x, y)$
	`NonBorne`	impossible	$\_^*\ as\ x/P(x)$

9.4 Inconvénients

Rappelons l'utilisation potentielle d'une classification de des filtres suivant leur complexité. Notre classification permet de déterminer si un chemin spécifié par un filtre peut être reconnu (accepté, vérifié) par un AFD, ou bien par un AFD gérant en plus un environnement, ou encore un AFD gérant un environnement et une pile.

L'idée est que si un chemin est accepté par un AFD, alors on peut utiliser un algorithme plus efficace pour le filtrage (par exemple, on n'a pas besoin de gérer de pile).

Cependant, cette classification présente trois inconvénients :

1. Est-ce qu'une classification des motifs fondée sur la reconnaissance de chemin est adaptée pour un algorithme de recherche de chemins ? Bien que correct, l'algorithme de recherche fondé sur la reconnaissance est éloigné de la réalité ; on ne peut pas se baser sur lui pour trouver une optimisation de la recherche. En effet, dans certains cas, on souhaiterait ne pas à avoir à vérifier tous les chemins si le filtre n'est pas adapté à la topologie : par exemple, si le motif filtre des chemins suivant une certaine direction alors que le GBF n'utilise pas cette direction, la première étape qui génère tous les chemins à tester est inutile.

2. Avant l'introduction des étoiles, nous avons toujours pu nous arranger pour construire un automate déterministe dont le nombre d'états est fini (l'indéterminisme du $OrPat$ disparaissant avec la réécriture de règle). Seule la présence de l'environnement E n'est pas

habituelle ; néanmoins, sa taille est connue dès l'analyse du filtre. De plus, sa présence est indispensable pour le calcul de la partie droite des règles.

La présence d'un itération impose de considérer que E n'est plus de taille bornée (ce qui fait apparaître une nouvelle classe de complexité), mais aussi l'ajout d'une structure de pile qui nous fait sortir du cadre des AFD pour entrer dans celui des automates à pile. On pourrait tout de même l'accepter comme nouvelle classe de complexité. Cependant, elle ne convient pas à certains cas : soit le motif suivant $1, 2, \_^*$ qui filtre les chemins d'une topologie commençant par $1, 2$. Tous les chemins commençant par $1, 2$ sont donc filtrés et il n'est pas utile d'utiliser une pile pour évaluer $\_^*$ (ce motif filtre en effet n'importe quel chemin) ; l'algorithme renvoie `(NonLineaire, Rien, Presente)` alors qu'on souhaiterait avoir `(Lineaire, Rien, Absente)`.

3. Pour élaborer cette classification, on construit l'automate en ajoutant les éléments manquant pour permettre la gestion de motifs plus complexes. Cette méthode correspond à un mécanisme d'enrichissements successifs des automates.

   Il serait peut-être plus adapté de reprendre l'algorithme général possible, qu'on spécialisera suivant la classe de motif. De plus, cette méthode permet une optimisation qui dépend aussi de la collection topologique ; il sera plus facile de faire le lien entre la complexité des filtres et les optimisations possibles associées aux topologies.

Chapitre 10

Optimisation

Nous possédons maintenant un algorithme de filtrage générique et une classification des filtres tous deux indépendants des collections. Nous avons vu également qu'une optimisation du filtrage ne peut être faite qu'en spécialisant l'algorithme actuel en fonction de la topologie sur laquelle il est appliqué. On pourra ainsi profiter des propriétés des collections "oubliées" par la généricité. Une telle optimisation peut demander une classification des filtres afin de pouvoir les comparer.

Dans ce chapitre, je présente une idée d'optimisation que j'applique sur deux types de topologies, les séquences et les arbres modulo AC. L'idée générale part d'une tentative d'amélioration de l'algorithme présenté dans le chapitre 8. On souhaite réduire le nombre d'entrées possibles de l'algorithme du filtrage qui jusqu'ici tenter de trouver un chemin à partir de chaque position. Si le motif contient par exemple une constante v, on n'essaie de filtrer qu'à partir des éléments de valeur v.

10.1 Idée générale de l'optimisation

Nous avons vu plus haut que l'algorithme de filtrage était en $\mathcal{O}(n\nu^m)$, avec n le nombre d'éléments dans la collection, m la longueur du motif, et ν le nombre maximum de voisins d'un élément suivant la topologie.

On peut tenter de réduire cette complexité en divisant le motif en deux parties autour d'un pivot qu'il est facile de localiser dans une collection. Par exemple, une constante est un bon pivot. Une fois ce pivot désigné, on inverse la partie gauche du filtre. On obtient 2 sous-filtres que l'on peut rechercher dans la collection. Soit le motif $p = p_1, e, p_2$, avec p_1 et p_2 deux sous-motifs et e l'élément pivot de p. Le principe est de rechercher tous les éléments de la collection filtrés par e, puis, à partir de chaque position trouvée, on exécute le filtrage de p_2 puis celui de p_1^{-1} (inverse de p_1).

Deux difficultés se présentent alors :

1. Il faut inverser le sous-motif p_1. Le problème majeur se situe essentiellement au niveau des gardes (et des mises à jour d'environnement ; la solution est la même). En effet, il est possible qu'une garde présente dans p_2 prenne comme arguments des éléments définis dans p_1. Soit $fv(p)$ les variables libres de p, et $bv(p)$ ses variables liées. On peut avoir un problème si $\exists x \in fv(p_2)/x \in bv(p_1)$. C'est la cas du motif $x, \_^*, 1, \_^*, y/(x == y)$, où si on prend comme pivot 1, la garde $g(x, y) = (x == y)$ est séparée de x ; on a :

$$\begin{cases} p_1^{-1} = & \_^*, x \\ p_2 = & \_^*, y/(x == y) \end{cases}$$

Or pour pouvoir évaluer g, il faut avoir filtré x. En supposant que le filtrage de p_2 soit effectué avant celui de p_1^{-1}, on va déplacer g de p_2 vers p_1^{-1} pour obtenir :

$$\begin{cases} p_1^{-1} = & \_^*, x/(x == y) \\ p_2 = & \_^*, y \end{cases}$$

2. Le choix du pivot est primordial. L'algorithme actuel filtre à partir de chaque position de la la collection, pour tenir compte de tous les chemins possibles. L'apparition du pivot e permet d'avoir des informations sur l'élément : il doit être filtré par e. L'idée est de réduire le nombre de positions de départ. Pour cela, il suffit de choisir e pour qu'il soit le plus contraignant possible, *i.e.* qu'il filtre peu d'éléments dans la collection. On préférera par exemple filtrer une constante à un élément quelconque.

   On remarque également qu'en cas d'échec du filtrage de p_2, il est inutile de filtrer p_1^{-1}. Aussi, on essaie de placer e le plus près du centre du motif pour diviser par 2 la longueur de celui-ci.

   Pour résumer, on cherche un pivot, le plus contraignant possible et placé le plus près du milieu du motif.

Cette optimisation est en fait fondée sur le fait que rechercher en premier les éléments les plus contraignants permet de déterminer les cas d'échec au plus tôt. Cela n'a cependant pas été mis en pratique de façon générale. J'ai plutôt essayer de l'adapter à deux types particuliers de collections, pour pouvoir aller plus loin. Il s'agit des séquences et des structures modulo AC, *i.e.* les ensembles et les multi-ensembles.

10.2 Optimisation des filtres dans les arbres modulo AC

10.2.1 La structure d'arbre modulo AC

Ces constructions permettent de créer des ensembles et des multi-ensembles. Elles sont dites *modulo AC* pour associatif-commutatif; il s'agit des propriétés dont est munie la virgule qui permet de construire ses collections :

- **Commutatif** signifie que si a est voisin de b (soit a, b), alors b est voisin de a (soit b, a). La relation de voisinage n'est pas forcément commutative comme dans les séquences par exemple. Par rapport à la structure de graphe générale que nous avons introduite précédemment, cela signifie que si b est un successeur de a alors b est également prédécesseur de a. Le graphe obtenu peut donc être considéré comme un graph non-orienté.
- **Associatif** signifie que $a, (b, c) = (a, b), c = a, b, c$. En l'associant à la commutativité, on montre que l'opérateur "," est également transitif : si a est voisin de b et b voisin de c, alors a est voisin de c. Il s'agit de la fermeture transitive de la relation de succession dans le graphe.

Comme les ensembles (respectivement les multi-ensembles) sont générés par l'ajout successif de ses éléments dans l'ensemble vide (respec. le multi-ensemble vide) en suivant ces propriétés, tous les éléments sont voisins les uns des autres. Le graphe représentant ces topologies est donc un graphe complet non-orienté.

La différence entre ensembles et multi-ensembles réside dans le fait que l'opérateur d'ajout possède une propriété supplémentaire pour les ensembles ; il est *idempotent*. Cela signifie qu'il n y a pas de répétition dans les ensembles : $set(a, a) = set(a)$.

10.2.2 Propriétés d'un motif

Cette particularité du constructeur a des conséquences sur le motif lorsqu'on souhaite l'appliquer sur ce type de collection. Dans le reste du rapport, nous appelerons AC, les structures d'arbre modulo AC.

On peut trouver deux propriétés intéressantes des motifs :

1. Les propriétés des ACs assurent que depuis n'importe quelle position, on a accès à tous les autres éléments de l'AC (le graphe représentant la relation de voisinage est en effet complet). De plus, les directions que les motifs peuvent suivre sont obligatoirement des "," dénotant l'ensemble des voisins possibles.

   Les directions deviennent donc inutiles pour les ACs ; il n'est pas utile de surcharger le motif avec un opérateur demandant de filtrer le reste du motif depuis toutes les positions restantes.

2. Le connecteur "," qui permet de construire les motifs a les mêmes propriétés que l'opérateur d'ajout d'un élément dans un AC. L'associativité et la commutativité autorisent les permutations à l'intérieur du motif. La notion d'ordre dans lequel il faut filtrer un AC n'a absolument aucune importance. Par exemple, il est claire que filtrer $1, 2$ dans un AC est totalement équivalent à filtrer $2, 1$. L'associativité permet de faire beaucoup mieux : le motif $(\_^*, 1), x$ peut être réécrit en $1, x, \_^*$.

   On peut donc déplacer des sous-parties du motif pour le réordonner de sorte qu'il soit plus efficace lors du filtrage.

10.2.3 Optimisation

Nous avons vu qu'il et préférable de chercher les éléments les plus contaignants en premier. Nous allons associer cette idée avec les propriétés des motifs.

Par réécriture, un nouveau motif équivalent et optimisé va pouvoir être obtenu. On peut distinguer deux étapes dans cette transformation :

1. Dans un premier temps, on supprime l'ensemble des "," pour éviter de faire des vérifications inutiles pendant le filtrage ;
2. On réordonne ensuite le filtre pour placer les éléments les plus contraignants au début. Les **règles principales** à suivre pour réordonner sont les suivantes :
   - On ne réordonne pas un motif s'il contient une ou plusieurs disjonctions ; en fait, il ne constitue plus une séquence, mais un arbre. Il devient alors trop délicat de toucher à un motif pour effectuer les commutations.
     L'optimisation qu'on est en train de présenter reste valable en présence de disjonctions. Il faut néanmoins utiliser la division de la règle en deux (*cf.* 9.2). On applique ensuite l'optimisation sur chacun des motifs issus de la séparation.
   - Les gardes et les mises à jour de l'environnement doivent être placées au plus tôt. Évidemment, il faut imposer qu'elles soient placées après la définition des identifiants qu'elles utilisent. Si on a par exemple la mise à jour $P = Union(x, y)$, on a comme contraintes que x et y se trouvent avant.
   - Pour les autres éléments du filtre, l'ordre à suivre est :

     $$CtePat \prec RecordPat \prec BlankPat \prec IterPat$$

     avec $x \prec y$ signifiant *«x est avant y»*.

   Cela correspond en fait à l'ordonnancement des éléments du filtre suivant leur niveau de contrainte. Autrement dit, $CtePat$ est plus contraignant que $RecordPat$, lui même plus contraignant que $BlankPat$. Les itérations sont mises le plus loin possible dans le motif car il s'agit des éléments les plus coûteux à filtrer ; elles filtrent en effet des chemins de

longueur variable, à la différence des autres qui filtrent des chemins de taille 1.

Il reste à déterminer l'ordre pour chaque sous-classe pour pouvoir comparer 2 $CtePat$, 2 $RecordPat$, etc. Pour cela, on essaie une fois de plus de suivre une politique qui permet de faire échouer le filtre le plus rapidement possible (*cf.* 10.1). Nous appelerons cet ensemble de règles, **contraintes secondaires** :

- $CtePat$: En supposant que le filtrage d'une constante dure le temps unitaire, on ne peut pas les ordonner.
  Cependant, en pratique, le temps de calcul pour faire la comparaison de deux valeurs `MGS` peut différer de beaucoup. La comparaison de deux entiers est par exemple beaucoup plus rapide que celle de deux chaîne de caractères. Il faut donc réordonner les $CtePat$ de telle sorte que les types les plus simples soient au début.
- $RecordPat$: Filtrer un enregistrement correspond en fait à vérifier une liste de propriétés que doivent respecter les champs de l'enregistrement.
  Toujours dans la même optique, on essaiera de mettre en premier les filtres d'enregistrement possédant le moins de propriétés à vérifier.
- $BlankPat$: Leur présence est souvent associée à celle de gardes. En fait, c'est en ordonnant les gardes qu'on va pouvoir organiser les $BlankPat$. Les gardes sont en fait des prédicats. Les identifiants correspondent souvent à un $BlankPat$ comme dans le filtre $x, y/f(x, y)$ où on souhaite trouver un couples d'éléments quelconques vérifiant la propriété f.
  On remarque qu'il est moins coûteux de trouver un n-uplet qu'un p-uplet parmi N éléments si $p > n$; en effet,

$$n < p \Rightarrow Card(Nuplet(n, N)) < Card(Nuplet(p, N))$$

  avec $Nuplet(n, N)$ signifiant l'ensemble des n-uplets d'un ensemble de cardinal N, et $Card$, le cardinal d'un ensemble. Aussi, il est judicieux de placer les gardes d'arité plus faible en premier. En effet, plus l'arité est faible, plus le nombre de uplets à vérifier est petit. De plus, dans le cas où le n-uplet est trouvé, il reste $N - n$ élément pour construire le p-uplet. Voici, par exemple ce qu'on cherche à obtenir :

$$(x, a, y, b, z/g(x, y, z))/f(a, b)$$

  est réécrit en

$$a, b/f(a, b), x, y, z/g(x, y, z)$$

- $IterPat$: A la différence des éléments de filtre précédant, $IterPat$ peut filtrer des chemins de longueur supérieure à 1. Une première remarque qu'on peut faire est qu'il est possible d'appliquer l'algorithme d'optimisation de motif sur les sous-motifs qui doivent être itérés. On obtient dès lors une première optimisation.
  Pour comparer deux itérations, il faut déterminer laquelle des deux prend le plus de temps. Pour cela, on peut utiliser la classification mis au point précedemment (*cf.* 9). Il suffit donc de calculer le type du motif itéré de chaque $IterPat$ pour déterminer lequel il faut mettre en premier.

Ces règles fournissent un ensemble de contraintes qui permettent le réordonnancement. On peut remarquer que l'application des règles principales permettent d'obtenir un ordre partiel sur les éléments du motif. En revanche, il n'est pas assuré que les contraintes secondaires n'entrent pas en contradiction avec les premières. En effet, elles peuvent conduire à un cylce tel que $(a \prec b) \wedge (b \prec c) \wedge (c \prec a)$ qui entraîne, par transitivité de la relation d'ordre $\prec$, que $(a \prec b) \wedge (b \prec a)$.

10.2.4 Algorithme de tri topologique

Nous venons de voir qu'à partir d'un motif, on est en moyen de produire une liste de constraintes par application d'une suite de règle. Il ne reste qu'à trouver à partir de toutes ces contraintes l'ordre qu'il va falloir suivre.

On cherche donc à trier les différents éléments du filtre afin d'en déduire un ordre total qui formera le motif optimisé. Or, les règles principales permettent d'obtenir une relation d'ordre partiel. On est assuré que si on représente cet ordre sous la forme d'un graphe, aucun cycle n'apparaît (ce qui revient à ne pas avoir de contradiction suivant l'ordre $\prec$).

Un algorithme permet de créer un ordre total à partir d'un ordre partiel ; il s'agit du *tri topologique* présenté en pseudo-langage par la figure 10.1 :

Algo TriTopologique (g : `graphe`)
  Début
    Soit $zeros$ = ensemble des sommets sans prédécesseurs de g ;
    Soit res = `[]` ;
    TantQue $zeros \neq$ `[]` **Faire**
      Soit z, un élément de $zeros$;
      $g := g/\{z\}$;
      $res := z :: res$;
      $zeros :=$ ensemble des sommets sans prédécesseurs de g
    FinTantQue ;
    Retourner $inverser(res)$
  Fin

FIG. 10.1 – Algorithme de tri topologique

Description

Cet algorithme utilise 3 variables :

1. g : il s'agit du graphe des contraintes ;
2. $zeros$: c'est un ensemble qui contient les sommets de g qui n'ont pas de prédécesseur. Ces sommets correspondent aux éléments les plus petits par rapport à l'ordre $\prec$;
3. res : il s'agit de la liste des éléments ordonnés suivant un ordre total respectant l'ordre partiel $\prec$.

L'algorithme construit res en lui mettant en tête un sommet z n'ayant aucun prédécesseur. Il enlève du graphe le sommet choisi puisqu'il est pris en compte dans l'ordre total. Il met enfin à jour l'ensemble $zeros$ qui est modifié si au moins l'un des successeurs de z ne possède plus d'autre prédécesseur.

Amélioration possible

L'algorithme de tri topologique n'impose aucune obligation en ce qui concerne le choix du sommet z de $zeros$. En fait, tous les sommets de la liste $zeros$ sont suceptibles d'être pris tout en conservant l'ordre partiel $\prec$.

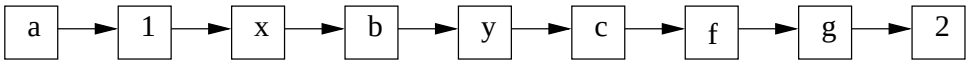
On peut néanmoins privilégier l'un d'eux. Il s'avère en fait qu'il suffit de respecter les règles principales pour construire le graphe des contraintes **sans cycle**, et de choisir z en suivant les contraintes secondaires de comparaisons de deux éléments de même classes de filtrage, pour obtenir un ordre total satisfaisant. Le choix pivilégié de z nous permet donc de respecter toutes les règles d'optimisation précitées.

exemple

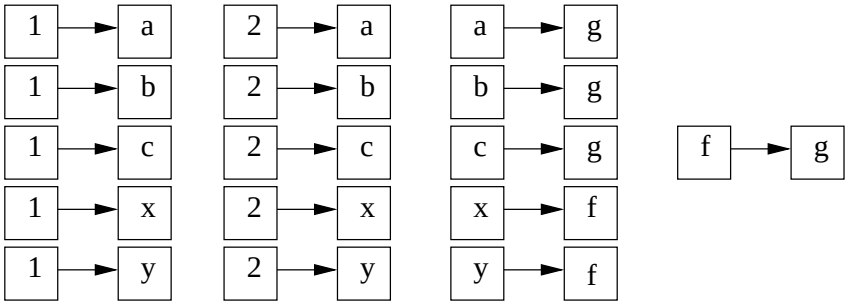
La figure 10.2 présente un exemple d'application de l'algorithme.

motif : a, 1, x, b, y, (c / f(x,y)) / g(a,b,c), 2

séquence d'origine :



Contraintes :



Séquence après tri topologique :

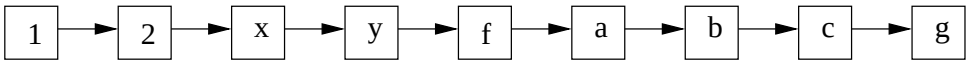


FIG. 10.2 – Exemple d'application de l'algorithme de tri topologique

10.2.5 Implémentation

On peut déjà commencer par une première remarque concernant la recherche des constantes ; les ACs présentent des avantages certains. En effet, au lieu d'attendre que la position courante soit de la bonne valeur, il suffit de rechercher directement la constante dans l'ensemble et de modifier la position courante en conséquence. Cette manipulation n'a pas de répercution sur le filtrage puisqu'on a accès à toute les éléments depuis n'importe quelle position. Cependant, cette recherche se fait en un temps logarithmique ; en effet, `MGS` représente les ensembles et les multi-ensembles par des arbres binaires.

Notre optimisation consiste à modifier le motif pour en créer un autre équivalent. Bien sûr, le motif optimisé ne peut être utilisé qu'avec les ACs. Aussi, il faut prévoir dans `MGS` un aiguillage permettant d'optimiser le motif si une règle est appliquée sur un AC.

Prétraitement du motif

Pour intégrer l'optimisation du filtrage dans les ACs, on a besoin de créer un nouveau motif. Après des essais non-fructueux d'implémentation au niveau de la classe d'objet `pattern`, j'ai décidé de me placer juste avant l'instanciation des motifs en objet de classe `pattern`. De ce point de vue, on profite de tous les aspects du motif, aussi bien sa forme d'arbre de dérivation que sa forme séquencielle, avec les prétraitements concernant les mises à jour de l'environnement (suppression des *AsPat* et ajout des *UpdatePat*). La figure 10.3 montre où l'instanciation des motifs a été modifiée pour intégrer l'optimisation.

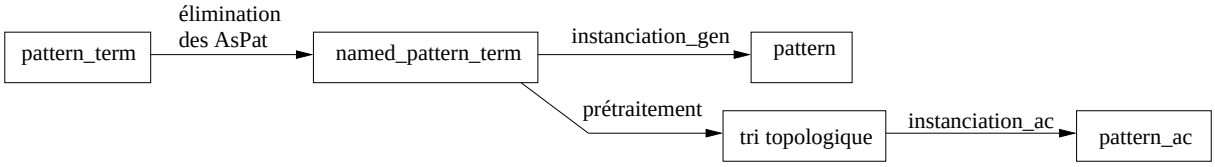


FIG. 10.3 – Instanciation des motifs pour l'optimisations des ACs

Un nouvel algorithme de filtrage

L'algorithme de filtrage pour les ACs est légèrement différent de l'algorithme général puisqu'on souhaite avoir une gestion particulière du filtrage de constantes, et gérer différemment les déplacements dans l'ACs (les *CommaPat* ayant été supprimées).

Pour cela, j'ai codé une nouvelle sorte de `pattern` qui hérite directement des classes existantes comme le montre la figure 10.4. En fait, seule la méthode `filtre` diffère et elle a donc été réimplémentée dans le cadre des ACs.

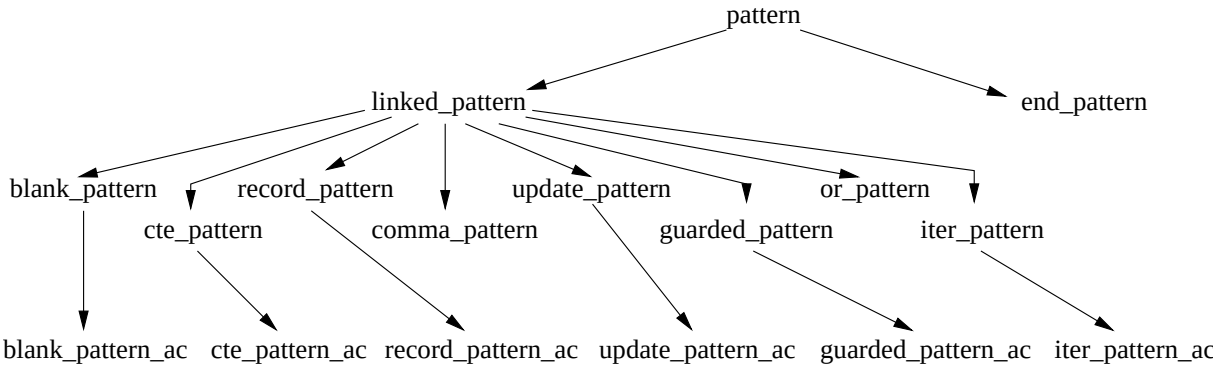


FIG. 10.4 – Hiérarchie des classes pour l'optimisation des ACs

10.2.6 Analyses des performances

Les bons résultats de l'optimisation des ACs peut être observés à travers deux exemples. La façon de procéder pour déterminer les performances consiste à appliquer une transformation sur un AC dans un premier temps sans l'optimisation, et donc en utilisant l'algorithme générique, et dans un second temps avec la réécriture du motif et le nouveau filtrage des ACs.

Premier cas

On souhaite montrer dans cet exemple qu'il est plus efficace de mettre les *CtePat* avant les *BlankPat*. Nous allons évaluer l'impact de l'optimisation sur la transformation suivante :

```
trans T = {
  _, 2 => 2
};;
```

Rappelons que l'optimisation des ACs transforme `T` en

```
trans T' = {
  2, _ => 2
};;
```

On applique `T` sur l'ensemble des entiers compris entre 1 et une variable N jusqu'à atteindre un point fixe. Le tableau suivant montre les performances dues à l'optimisation suivant le cardinal de l'ensemble :

N	filtrage générique (en seconde)	filtrage optimisé (en seconde)
220	1.59	1.09
300	4.25	2.60
400	10.58	6.05
450	15.24	8.54
500	21.07	11.67
550	28.09	15.51
600	36.48	20.03
700	58.12	31.06

On remarque que l'optimisation permet dans ce cas de diviser par 2 le temps de la recherche.

Second cas

Ce deuxième cas montre que l'optimisation permet d'atteindre des performances considérables. La transformation utilisée est :

```
trans T = {
  100, x, y, 101, 300 => 100,101,300
};;
```

La collection est l'ensemble des entiers compris entre 1 et 2000 ; la transformation est appliquée jusqu'à trouver un point fixe. On obtient le résultat :

filtrage générique (en seconde)	filtrage optimisé (en seconde)
1395.69	6.19

Autres tests

L'intégration de l'optimisation n'étant pas complète (il manque notamment la gestion de l'itération), on ne peut pas fournir d'autres exemples.

Cependant, des tests préliminaires ont été réalisés avec une version précédente de `MGS` pour savoir si cette optimisation était pertinente. Le tableau présente l'ensemble des résultats obtenus :

Description : permutation entre	cardinal N de l'ensemble	motif m	motif réécrit m'	temps pour m (en seconde)	temps pour m' (en seconde)
CtePat et *BlankPat*	2000	$\_, 100$	$100, \_$	7	$< 1sec$
CtePat et *IterPat*	11	$\_^+, 10$	$10, \_^+$	31	3
BlankPat et *IterPat*	11	$\_^+, \_$	$\_, \_^+$	30	3

10.2.7 Conclusion

Les propriétés de commutativité et d'associativité des ACs permet une bonne manipulation des filtres. On peut alors réécrire les motifs pour faire remonter en début de filtre les éléments les plus contraignant et les moins coûteux à filtrer, tels que les constantes.

Les résultats de cette optimisation sont satisfaisants. Bien que la complexité du filtrage reste exponentiel, on peut espérer avoir au moins un facteur 2 pour l'optimisation du temps d'exécution.

Cependant, pour que la réécriture soit réellement efficace, il faut que le motif soit précis, ce qui signifie, en d'autres termes, qu'il doit présenter un maximum des éléments contraignants.

Durant le stage, j'ai tenté plusieurs fois d'intégrer l'optimisation des motifs pour les ACs à plusieurs endroit différents. Ceci m'a permis de connaître en détail le mécanisme de filtrage (au niveau de l'implémentation) du projet `MGS` ; mais surtout, j'ai pu proposer une méthode générique qui permet d'intégrer n'importe quelle optimisation du filtrage, que ce soit en terme de traitement de motif avant l'instanciation, ou en terme de modification de l'algorithme de filtrage. J'utilise cette méthode pour intégrer l'optimisation des séquences qui suit.

10.3 Optimisation des filtres dans les séquences

10.3.1 Parallèle avec la recherche de mot dans un texte

Là encore, nous nous intéressons à un type de collection monoïdale. A la différence des ACs, l'opérateur d'ajout "," des séquences est uniquement associatif (*i.e.* $(a, b), c = a, (b, c) = a, b, c$). Une séquence se présente comme une forme linéaire. On peut en avoir une vision récursive très proche des listes de *OCaml* ; elle est composée d'un élément de tête h et d'une liste qui constitue sa queue t ; la relation de voisinage constituant la topologie des séquences fait que la tête de t est l'unique voisin de h.

Cependant, mis à part cette linéarité, cette collection ne présente pas de propriété qui pourrait amener à une optimisation des motifs. On peut tout de même améliorer la recherche en cherchant une source d'optimisation dans d'autres domaines. Il s'avère que le filtrage d'un motif dans une séquence semble un travail proche de la recherche d'un mot ou d'une expression régulière dans un texte ; or dans ce domaine, il existe de nombreux algorithmes. Il doit donc être possible d'utiliser ces algorithmes pour s'inspirer et améliorer le filtrage dans les séquences `MGS`.

Observons quelques exemples d'algorithmes utilisés sur les textes :

- *Egrep* est une application *Unix* qui permet la recherche de chaîne de caractères dans des fichiers textes.
  Elle utilise des expressions régulières pour spécifier le motif à chercher et construit un algorithme fini déterministe (AFD) pour implanter la recherche elle-même. La première version de l'application, *Grep*, a été écrite par Ken Thompson, puis elle a été améliorée en 1976 par Al Aho qui implémenta l'utilisation d'AFD pour la recherche (*cf.* [Hum98]). *Egrep* était néanmoins beaucoup plus lent que *Grep* pour les motifs complexes car l'automate était entièrement construit *avant* la recherche. Aho augmenta l'efficacité par une technique appelée «cached lazy evaluation».
- Les *ESMA* (pour ***E**xact **S**tring **M**atching **A**lgorithms*) sont également très étudiés et utilisés pour la recherche d'occurence exact de mot dans un texte. Le cours d'algorithmique du DEA AMIB a présenté cette année certains de ces algorithmes. Le lecteur intéressé par plus de détails sur ces algorithmes, peut se référer à [CL]. Ils sont notamment utilisés en bio-informatique pour, par exemple, la recherche de structures particulières dans un génome.

Ils sont parmi les algorithmes les plus rapides (plus rapides que *Egrep*), mais ils présentent tout de même des désavantages par rapport à leur utilisation avec les séquences `MGS` :
- ils utilisent des mots exacts et non des motifs ou des expressions régulières comme `MGS` ;
- l'alphabet qu'ils utilisent doit être fini pour pouvoir appliquer les propriétés d'optimisation.

Ces deux exemples représentent les deux types d'algorithme que nous pouvons utiliser. Ils semblent très différents et difficilement conciliables.

Du côté de *Egrep*, les mainteneurs du projet essaie d'implémenter une version utilisant les expressions régulières de l'algorithme de Boyer-Moore. Leur problème est de réussir à déterminer la taille d'une chaîne de caractères spécifiée par une expression régulière. Or la présence de l'itération dans la grammaire empêche de déterminer cette taille. Nous serons également confrontés à ce problème dans `MGS`. Cependant, des travaux tentent de généraliser des *ESMA* pour qu'ils gèrent les expressions régulières (*cf.* [Wat01a] et [Wat01b]). Ils ont l'air de dépasser le problème de l'itération.

Il reste cependant que tous ces algorithmes et leurs développements travaillent avec des alphabets finis, ce qui ne convient à la recherche de motif dans `MGS`. J'ai donc tenter de résoudre le problème en généralisant l'algorithme de Boyer-Moore (*cf.* [BM77]) aux alphabets infinis ; j'ai mis de côté les itérations et les disjonctions pour éviter de rencontrer le problème de la longueur indéterminée des mots filtrés.

10.3.2 Algorithme de Boyer-Moore

Choix de l'algorithme de Boyer-Moore

Il s'agit d'un des *ESMA*. Il est réputé pour être le plus rapide et le plus utilisé parmi eux avec l'algorithme de Knuth-Morris-Pratt. Pour simplifier, nous appellerons BM l'algorithme de Boyer-Moore, et KMP celui de Knuth-Morris-Pratt.

Ces algorithmes sont fondés sur l'algorithme naïf de recherche qui consiste à vérifier l'égalité à partir de chaque lettre dans le texte. Cet algorithme rappelle l'algorithme général de `MGS` qui fonctionne suivant le même principe mais sur n'importe quelle collection. Par contre, BM et KMP tentent de déplacer la fenêtre de recherche de plus d'une position à la fois. Ils effectuent pour cela une phase de prétraitement sur les motifs pour précalculer ces déplacements.

Avant de comparer BM et KMP, nous allons rappeler quelques caractéristiques de chacun :

	BM	KMP
sens de lecture dans la fenêtre	de droite à gauche	de gauche à droite
phase de prétraitement	$\mathcal{O}(m+\sigma)$	$\mathcal{O}(m)$
phase de recherche	$\mathcal{O}(mn)$	$\mathcal{O}(m+n)$

avec m la longueur du motif, n la taille du texte, et σ la taille de l'alphabet. Ce tableau nous porterait à choisir KMP plutôt que BM comme algorithme de départ. Cependant BM à d'autres avantages :
- il est considéré comme l'algorithme de recherche le plus efficace et il est utilisé dans la plupart les logiciels pour implémenter les fonctions "rechercher" ou "remplacer" ;
- il est en pratique le plus utilisé. En effet, si on compare ses résultats avec ceux de KMP, il est la plupart du temps plus rapide ;
- il ne fait que $3n$ comparaisons dans le pire des cas, s'il est utilisé avec un mot non-périodique ;
- il peut atteindre la performance maximale de $\mathcal{O}(n/m)$
- les travaux de Bruce W. Watson montre qu'il peut être étendu aux expressions régulières et que ses performances sont meilleures que celles de l'extension de KMP.

C'est pour ces raisons que j'ai choisi BM pour cette optimisation.

présentation de BM [1]

L'algorithme lit les caractère du motif de droite à gauche en commençant par celui le plus à droite. Dans le cas où une erreur de comparaison se produit (ou si le mot est entièrement trouvé), il utilise deux fonctions précaculées pour déplacer la fenêtre vers la droite. Les deux fonctions s'appellent *good-suffix shift* (déplacement dû au bon suffixe) et *bad-character shift* (déplacement dû au mauvais caractère).

Supposons qu'une erreur apparaît au cour de la comparaison entre le caractère $x[i] = a$ du motif x, et le caractère $y[i+j] = b$ du texte y. La fenêtre est placée en position j du texte. On a alors $x[i+1..m-1] = y[i+j+1..j+m-1] = u$ et $x[i] \neq y[i+j]$.

- Le déplacement *good-suffix shift* consiste à aligner le segment $y[i+j+1..j+m-1] = x[i+1..m-1]$ avec son occurence la plus à droite dans x qui est précédée d'un caractère différent de $x[i]$ (*cf.* 10.5). S'il n'existe pas de tel segment, le déplacement consiste à aligner

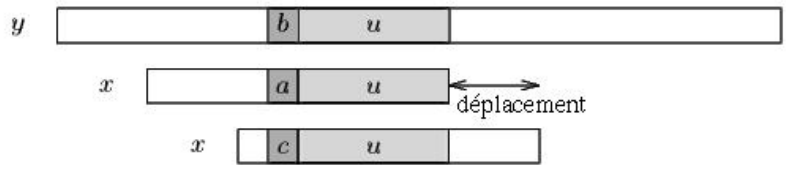


FIG. 10.5 – Good-suffix shift : u apparaît à nouveau, précédé du caratère c différent de a

le plus long suffixe v de $y[i+j+1..j+m-1]$ avec un préfixe de x (*cf.* 10.6).

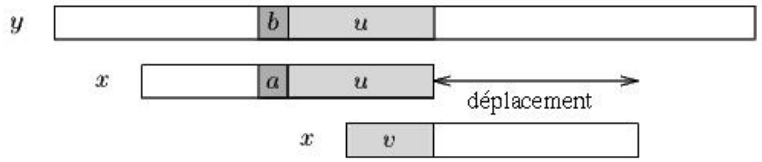


FIG. 10.6 – Good-suffix shift : seul un suffixe de u est présent dans x

- Le déplacement *bad-character shift* consiste à aligner le caractère $y[i+j]$ avec son occurence la plus à droite dans $x[0..m-2]$ (*cf.* 10.7).

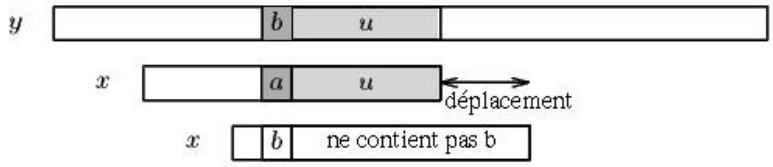


FIG. 10.7 – Bad-character shift : a est présent dans x

Si $y[i+j]$ n'apparaît pas dans le motif x, aucune occurence de x dans y ne peut contenir $y[i+j]$ et la fenêtre est déplacer de telle sorte que son premier caractère soit le voisin

[1] insiprée de celle de Charras et Lecroq sur
http ://www-igm.univ-mlv.fr/Ïecroq/string/node14.html#SECTION00140

immédiat de $y[i+j]$ qui ne se trouve plus dans la fenêtre. En fait, la fenêtre se retrouve en $y[i+j+1]$ (*cf.* 10.8).

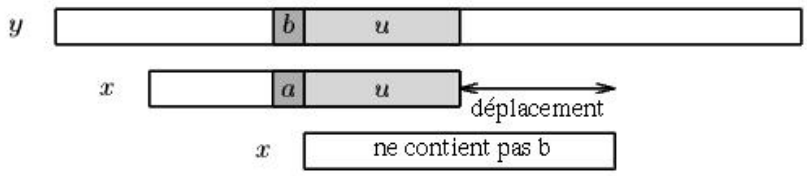


FIG. 10.8 – Bad-character shift : a n'apparaît pas dans x

Pour effectuer le meilleur déplacement possible, l'algorithme déplace la fenêtre du maximum rendu par ces deux fonctions.

Nous ne rentrerons pas dans les détails pour expliquer les phases de prétraitement. Les algorithmes de ces fonctions sont présentés directement dans la figure 10.9. Ils sont importants vu que je dois les modifier pour qu'ils supportent les alphabets infinis. La notation $bmBc\ (x.(i)) \leftarrow (m-i-1)$ signifie qu'on définit la fonction $bmBc$ en $x.(i)$, et qu'elle vaut $m-i-1$.

Introduction de la gestion du *BlankPat*

C'est à cause de la présence d'élément de type *BlankPat* dans le motif, que celui-ci devient sensible à la finitude de l'alphabet. S'il n'y avait pas ce type de filtre, on pourrait construire l'alphabet à partir des constantes présentes dans le filtre et en rajoutant un caractère spécial signifiant « tous autres caractères absents du filtre ». La présence des *BlankPat* empêche de connaître la valeur de l'élément filtré ; celui-ci peut aussi se trouver ailleur dans le motif tout comme il peut être absent. Or c'est sur ce genre de remarque que sont construits les fonctions de déplacement.

Les changements à apporter pour que l'algorithme gère les *BlankPat* sont en nombre réduit. Bien sûr, l'efficacité de l'algorithme se voit diminuée, mais on ne fera jamais pire qu'un déplacement de taille 1. Les modifications apparaissent lors des comparaisons entre caractères du motif.

preBadChar redéfinit la fonction $bmBc$ à chaque tour de boucle. Cette fonction qui prend en argument un caractère de l'alphabet, retourne la taille du déplacement à faire. Or, dans `MGS`, $x.(i)$ ne dénote pas forcément une constante ; si c'est un *BlankPat*, toute la fonction doit être redéfinie. On modifie l'intérieur de la boucle de la façon suivante :

Si $x.(i) \neq BlankPat$ **Alors**
  $bmBc\ (x.(i)) \leftarrow (m-i-1)$
Sinon
  $bmBc \leftarrow (fun\ \_ \rightarrow (m-i-1))$
FinSi

preGoodSuff construit la fonction $bmGs$ en deux étapes. Elle constitue d'abord le tableau des suffixes du motif x avec la fonction `suffixes`. En fait, la variable `suff` suit la définition suivante :

$$\forall i \in [|1; m-1|],\ \text{suff}[i] = \max\{k : x[i-k+1..i] = x[m-k..m-1]\}$$

Il s'agit de la longueur du sous-mot de x le plus long se terminant en i tel qu'il soit au suffixe de x de même taille.

```
Algo preBadChar (x : motif, m : longueur de x)
    Début
        Soit bmBc = (fun _ → m)
        Pour i = 0 à (m − 2) Faire
            bmBc (x.(i)) ← (m − i − 1)
        FinPour
        Retourner bmBc
    Fin

Algo suffixes (x : motif, m : longueur de x)
    Début
        Soit suff, un tableau de taille m
        Soit g = m − 1
        Pour i = (m − 2) à 0 Faire
            Si (i > g&& suff[i + m − 1 − f] < i − g) Alors
                suff[i] = suff[i + m − 1 − f]
            Sinon
                Si i < g Alors g = i FinSi
                f = i
                TantQue (g >= 0&&x[g] == x[g + m − 1 − f]) Faire
                    g = g − 1
                FinTantQue
                suff[i] = f − g
            FinSi
        FinPour
        Retourner suff
    Fin

Algo preGoodSuff (x : motif, m : longueur de x)
    Début
        Soit suff = suffixes(x, m)
        Soit bmGs = (fun _ → m)
        Soit j = 0
        Pour i = (m − 1) à (−1) Faire
            Si (i == −1|| suff[i] == i + 1) Alors
                TantQue j < m − 1 − i Faire
                    Si (bmGs[j] == m) Alors bmGs (j) ← (m − 1 − i) FinSi
                FinTantQue
            FinSi
        FinPour
        Pour i = (m − 1) à (−1) Faire
            bmGs (m − 1 − suff[i]) ← (m − 1 − i)
        FinPour
        Retourner bmGs
    Fin
```

FIG. 10.9 – Pseudo-langage des définitions de *bad-character shift* et de *good-suffix shift*

Dans la boucle **TantQue**, il y a une comparaison entre 2 caractères du motif. Cependant, le terme de *comparaison* est ici employé parce qu'il s'agit de deux caractères. Il faut en fait comprendre cette égalité de la façon suivante :

$$x[g_1] == x[g_2] \text{ ssi soit } m, \text{ un élément filtré par } x[g_1] \text{ ; } m \text{ est filtré par } x[g_2]$$

Le tableau suivant présente la valeur de cette égalité lors de la comparaison de $CtePat$ et de $BlankPat$:

		$x[g_2]$	
		$CtePat(v_2)$	$BlankPat$
$x[g_1]$	$CtePat(v_1)$	$v_1 == v_2$	`vrai`
	$BlankPat$	*Je ne sais pas*	`vrai`

C'est évidemment la case « *Je ne sais pas* » qui pose problème. En fait, on ne peut répondre qu'au moment du filtrage. Cependant, pour prendre ce cas en compte, il suffit de modifier la définition de `suff` ; maintenant, suff[i] ne correspond pas à la taille la plus grande mais aux tailles les plus grandes. On doit prendre en compte à la fois l'échec et à la fois le succès du filtrage. La boucle **TantQue** est transformée ainsi :

Soit *cont* = `vrai`
TantQue ($g >= 0$&& *cont*) **Faire**
Filtrer ($x[g], x[g+m-1-f]$) **Avec**
| $CtePatv_1, CtePatv_2 \rightarrow cont = (v_1 == v_2)$
| $BlankPat, CtePat \rightarrow$ suff[i] $= (f-g) ::$ suff[i]
| $CtePat, BlankPat \rightarrow$ **Skip**
| $BlankPat, BlankPat \rightarrow$ **Skip**
FinFiltrer
Si *cont* **Alors** $g = g - 1$ **finSi**
FinTantQue

Le filtrage de $BlankPat, CtePat$ montre qu'on prend en compte le cas où $BlankPat$ a filtré un élément différent de $CtePat$, mais on ne modifie pas la variable *cont* pour que l'algorithme tourne toujours comme si l'élément filtré était égal à la constante.

On obtient alors une liste de tailles des suffixes pour chaque position ; on applique ensuite $preGoodSuff$ comme avant mais en itérant sur chaque élément de la liste `suff[i]` quand celle-ci est utilisée.

10.3.3 Algorithme de filtrage optimisé pour les séquences

L'algorithme de Boyer-Moore a été amélioré pour gérer les $BlankPat$ et les $CtePat$. On peut également faire en sorte de prendre en compte les $RecordPat$ en créant de la même façon que pour $BlankPat$ un tableau de comparaison qui permet de répondre à la question « *si m est filtré par $x[g_1]$, est-il filtré par $x[g_2]$* ? ». Ceci n'est pas détaillé dans ce rapport mais peut être fait facilement.

Tous les éléments unitaires des filtres (éléments filtrant des chemins de longueur 1) sont gérés par BM. Il faut maintenant introduire les éléments restant et préciser les limites qu'on s'est fixé. Nous avons déjà vu que l'optimisation ne concerne pas (pour l'instant) les disjonctions et les itérations. Il reste donc à intégrer les déplacements ($CommaPat$), les gardes ($GuardedPat$) et les mises à jour d'environnement ($UpdatePat$).

En ce qui concerne les directions, on retrouve la même inutilité que pour les ACs. Les $CommaPat$ permettent les déplacements vers l'unique voisin de chaque élément. Aussi, lors de la séquencialisation des motifs elles seront supprimées.

Les *GuardedPat* et les *UpdatePat* posent un petit problème dû au sens de lecture dans fenêtre (de droite à gauche). Ils apparaissent avant que leurs paramètres n'aient été définis. Par exemple, pour le motif $1, x/f(x)$, la séquencialisation donne :

$CtePat(1)$	$BlankPat(x)$	$GuardedPat(f)$

Si on lit se tableau de droite à gauche ce tableau, f est évaluée avant la définition de x. On pourrait réécrire le motif pour déplacer la garde f et la placer avant x. Cependant, lors de l'évaluation, on évaluerait f avant de savoir si $CtePat(1)$ filtre bien un élément de valeur `MGS` 1. On préfère tester les constantes en premier.

Description

Pour ne pas avoir à faire les inversions, j'ai séparé le filtrage en deux étapes :

1. on construit un motif allégé qui ne contient plus que les éléments unitaires du motif d'origine. On l'appelera squelette du motif puisqu'il permet de trouver un chemin dans la séquence qui correspond en gros au filtre de départ mais où les gardes et les mises à jour de l'environnement ne sont pas prises en compte.
   On utilise le squelette du motif pour définir les fonctions *good-suffix shift* et *bad-charater shift*. L'application de BM va renvoyer un site où les éléments unitaires sont filtrés correctement et où il reste à vérifier les *GuardedPat* et les *UpdatePat*.

   Pour implémenter la recherche d'un site valable pour le squelette, j'ai créé une nouvelle classe d'objets utilisant BM.

   ```
   class ['position] boyerMoore_pattern_seq skel m bmGs bmBc next =
     object(self)
     inherit ['position] linked_pattern [] true next
     ...
     method filtre topo =
       begin
         (* implémantation de l'algorithme de Boyer-Moore pour skel *)
       end
   end;;
   ```

   Plusieurs paramètres sont à fournir lors de la création d'une instance :
   - `skel` est le squelette du motif d'origine ;
   - `m` est la longueur du motif ;
   - `bmGs` et `bmBc` sont respectivement les fonctions de déplacement *good-suffix shift* et *bad-charater shoft* ;
   - `next` est la suite de la séquence qui permet de faire la deuxième étape du filtrage

2. la seconde étape consiste à lancer le filtrage normalement en utilisant les objets propres aux séquences. Je les ai ajoutés à la hierarchie de classe de `pattern` comme le montre la figure 10.10. Ce sont des éléments de motif normaux dont la méthode `filtre` sert juste à mettre à jour l'environnement et à vérifier les gardes. Si un échec se produit, il est rapporté en tête de séquence à l'objet de classe *boyerMoore_pattern_seq* ; celui-ci se charge alors de déplacer la fenêtre toujours en suivant l'algorithme BM. Par contre, si le chemin vérifie toutes les conditions imposées par les gardes, la règle est appliquée.

Pour l'exemple précédent, on obtient après l'instanciation en objet la séquence suivante :

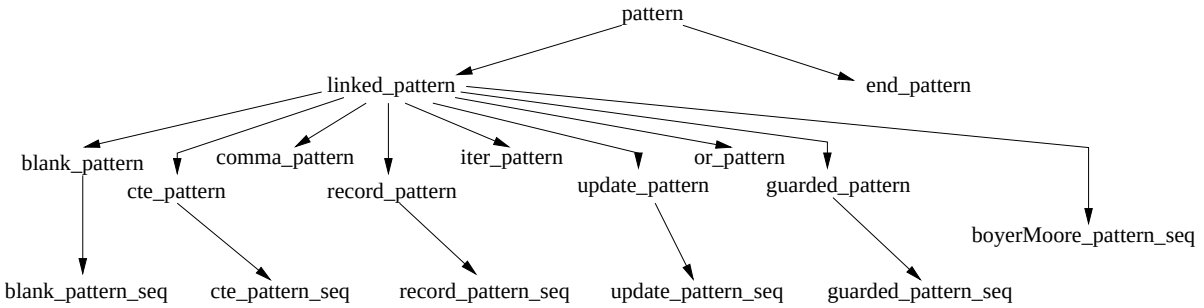


FIG. 10.10 – Hiérarchie des classes pour l'optimisation des séquences

$$\boxed{\begin{array}{|c|}\hline boyerMoore\_pattern\_seq \\ \hline skel = 1, x \\ \hline \end{array}} \rightarrow \boxed{cte\_pattern\_seq(1)} \rightarrow \boxed{blank\_pattern\_seq(x)} \rightarrow \boxed{guarded\_pattern\_seq(f)} \rightarrow \boxed{end\_pattern}$$

En ce qui concerne l'implémentation de la définition du motif squelette et de ses fonctions de déplacement associées, et l'instanciation du motif en objet, on procédera de la même façon que pour les ACs comme le montre la figure 10.11.

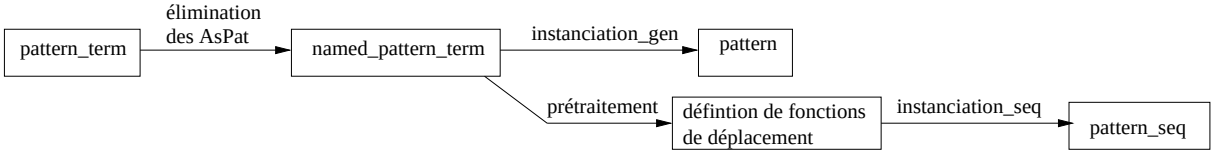


FIG. 10.11 – Instanciation des motifs pour l'optimisations des séquences

10.3.4 Analyses des performances

Les performances de BM sont moins spectaculaires que celles de l'optimisation du filtrage des ACs. Elles restent tout de même perceptibles.

L'optimisation des séquences possède un défaut : si les déplacements de la fenêtre sur la séquence sont toujours égaux à 1, on retrouve l'algorithme naïf du début. Cependant, le fait d'avoir intégré BM ajoute des instructions qui peuvent ralentir la recherche au point de faire moins bien que l'algorithme naïf. On commencera par évaluer cette perte.

On observera ensuite les performances de l'algorithme sur un exemple simple.

Enfin, dans un cadre plus pratique nous verrons comment réagit la généralisation de BM à une recherche de mot dans un texte.

Cas où les déplacements sont petits

Pour chercher à savoir combien BM perd par rapport à l'algorithme naïf dans des cas dégénérés où la taille déplacements est 1, on utilisera deux exemples. Les deux transformations sont utilisées :

```
trans T1 = {
  _,_,_ => 1
```

```
};;

trans T2 = {
  _,1,_ => 1
};;
```

On appliquera ces règles sur des séquences d'entiers assez longues (entre 50000 et 250000 éléments). Elles sont générées aléatoirement. Voici les tableaus récapitulatifs des résultats :

Taille de la séquence avec T1	filtrage générique (en seconde)	filtrage optimisé (en seconde)
50000	< 0.5	< 0.5
100000	0.58	0.60
150000	1.04	1.08
200000	1.38	1.47
250000	1.80	1.94

Taille de la séquence avec T2	filtrage générique (en seconde)	filtrage optimisé (en seconde)
50000	< 0.5	< 0.5
100000	1.015	1.025
150000	1.580	1.630
200000	2.240	2.280
250000	2.860	2.950

Les différences entre les deux algorithmes ne sont pas significatives bien qu'elles penchent en faveur de l'algorithme naïf. Sachant que les performances sont à peu près équivalentes, on peut intégrer l'optimisation de BM à MGS sans risquer d'ajouter des ralentissements non désirés dans ce genre de cas.

Cas simples

Nous allons observer maintenant le comportement de BM avec des exemples simples rendant compte du gain apporté par l'algorithme. On construit pour cela plusieurs transformations :

```
trans T1 = {
  _,1,_,2 => 1,2
};;

trans T2 = {
  _,1,_,2,_ => 1,2
};;

trans T3 = {
  _,1,_,2,_,3 => 1,2,3
};;

trans T4 = {
  _,1,_,2,_,3,_ => 1,2,3
};;
```

Ces 4 transformations vont nous permettre d'étudier, en plus de l'impact de l'optimisation, comment réagit BM à l'ajout d'un *BlankPat* à la fin d'un motif (comparaison entre T1 et T2, et entre T3 et T4).

Les tableaux suivant présentent le résultat.

Taille de la séquence	T1		T2	
	générique	optimisé	générique	optimisé
100000	3.35	2.49	3.43	2.44
150000	5.27	4.36	5.85	4.56
200000	7.65	5.88	7.88	6.50
250000	9.85	7.24	9.31	8.80

Taille de la séquence	T3		T4	
	générique	optimisé	générique	optimisé
100000	1.38	0.98	1.37	1.03
150000	2.59	1.77	2.56	1.92
200000	3.45	2.34	3.43	2.62
250000	4.29	3.00	4.26	3.24

Dans chaque cas, il y a effectivement une optimisation du temps de recherche des motifs qui se situe entre 18 et 30%. L'utilité de BM est donc justifié. Cependant, on peut constater également une perte d'efficacité avec l'ajout d'un *BlankPat* au bout du motif. Les séries de tests de `T3` et `T4` sont intéressante de ce point de vue. En effet, on observe que les temps de calcul avant et après ajout du *BlankPat* sont les mêmes pour l'algorithme générique. En revanche, BM est sensible à cette modification. Alors que l'optimisation était de 30% pour `T3`, elle n'atteint que 25% pour `T4`. Cette sensibilité est celle des fonctions de déplacement par rapport à la présence d'élément contraignant et d'éléments plus souples dans le filtre. L'optimisation apporté par la généralisation de l'algorithme de BM est tout de même flagrante.

Un exemple concret

On souhaite représenter l'action d'un ensemble d'enzymes de restriction sur l'ADN. La structure de l'ADN est simplifiée à une séquence des lettres `A`, `T`, `G` et `C`. Les chaînes d'ADN peuvent être représentée par une séquence `MGS` de caractères dont la déclaration est la suivante :

```
collection ADN = seq ; ;
```

ADN est un sous-type de `seq`.

Une enzyme de restriction est représentée par une règle qui divise le brin d'ADN en deux ; on peut par exemple avoir la règle :

```
EcoRI = x+ as X, ("G","A","A","T","T","C"), y+ as Y
      ⇒ (X,"G") ::("A","A","T","T","C",Y) ::():bag ;
```

Malheureusement nous ne gérons pas l'itération dans l'optimisation. Pour pâlier à ce manque, on procède différemment pour marquer la point de clivage. On utilisera donc la transformation suivante :

```
trans EcoRI = {
  "G","A","A","T","T","C" => "G","CL","A","A","T","T","C" } ; ;
```

Le caractère `"CL"` désigne le point de clivage où agit l'enzyme.

Cet exemple est typique de l'utilisation de BM. On devrait constater que l'algorithme générique, identique à l'algorithme naïf, n'est vraiment pas adapté à ce genre de recherche. Pour cela, on crée un génome de 2000000 paires de bases choisies aléatoirement avec une équiprobabilité pour chaque type base, et on applique la transformation `T`.

filtrage générique (en seconde)	filtrage optimisé (en seconde)
5.48	4.08

L'optimisation est d'environ 26%. L'optimisation est flagrante sachant qu'on est pas assuré de voir apparître le motif "GAATTC" dans la séquence, celle-ci étant générée aléatoirement. Une augmentation moyenne des performances de l'ordre de 30% est satisfaisant.

Conclusion

L'optimisation des séquences est inspirée des algorithmes de recherche de mots ou d'expressions régulières dans des chaînes de caractères. Cependant, les algorithmes connus travaillent principalement sur des motifs construits sur des alphabets de taille finie. Or dans `MGS`, le constructeur *BlankPat* permet de filtrer n'importe quelle valeur, et il y en a une infinité possible.

J'ai pu adapter l'algorithme de Boyer-Moore pour qu'il puisse gérer le constructeur *BlankPat*. Bien sûr, les fonctions de déplacement ne sont pas aussi perfomantes quand il y a dans le motif un tel constructeur. Néanmoins, l'analyse des performances montre qu'on peut atteindre des optimisations de l'ordre de 30%.

Pour l'instant l'algorithme ne prend pas en compte des motifs présentant des disjonctions ou encore des itérations. Il est possible d'introduire ces constructeurs dans l'optimisation :

- Dans le chapitre 9, nous avons vu qu'on pouvait diviser un motif contenant des disjonctions en plusieurs motifs. L'algorithme de Boyer-Moore supporte la recherche d'un seul mot. Il existe une généralisation de cet algorithme pour la recherche de plusieurs mots : c'est l'algorithme de Commentz-Walter. Il fonctionne suivant le même principe, c'est à dire en utilisant des fonctions de déplacement précalculées. Il est sûrement possible de compléter cet algorithme de la même façon que celui de Boyer-Moore pour gérer les *BlankPat*.
- Les itérations peuvent aussi faire l'objet d'une généralisation des algorithmes de Boyer-Moore et de Commentz-Walter. En effet, Bruce W. Watson propose dans [Wat01a] et [Wat01b] une généralisation pour les grammaires régulières. Or l'itération est le résultat d'une production de la forme :

  $$X \rightarrow AX$$

  Toujours de la même façon, cet algorithme utilise des fonctions de déplacement calculées en fonction des productions. On peut donc intégrer l'ajout du *BlankPat* de façon équivalente à celle que j'ai présentée plus haut. Le cas de la disjonction est en fait inclu dans les grammaire régulière.

L'optimisation des séquences n'est pas définitive ; mais le fait d'avoir réussi à intégrer le constructeur *BlankPat* semble prometteur pour une intégration des algorithmes plus généraux gérant des grammaires régulières.

Troisième partie

Conclusions et perspectives

Conclusion

MGS tente de fournir aux biologistes un outil de programmation dédié à la modélisation et à la simulation. Il s'agit en effet d'un langage de programmation déclaratif fondé sur deux nouvelles notions :

1. les collections : ce sont des ensembles dans lesquels les éléments sont organisés les uns par rapport aux autres par une relation de voisinage. Elles permettent de décrire les systèmes biologiques à modéliser.
2. les transformations : ce sont des fonctions particulières agissant sur les collections par filtrage et réécriture de sous-collections. Elles permettent de décrire les fonctions d'évolution du système.

MGS porpose de représenter l'état d'un système (biologique) par une collection topologique. L'évolution du système est spécifiée par une transformation qui définit les interactions entre les divers éléments du système.

Une des originalités de MGS réside dans le point de vue adopté sur le filtrage : il permet d'agir de façon locale sur les collections ; les langages déclaratifs font d'habitude du filtrage sur la construction des valeurs. On ne pouvait alors travailler que sur des constructions algébriques. Pour MGS, le filtrage se fait sur n'importe quel type de valeur (algébrique ou non, comme les tableaux) tant qu'elle est organisée suivant une relation de voisinage.

C'est pourquoi le filtrage programmé en MGS est un algorithme générique, ne tenant compte d'aucune propriété particulière de la relation de voisinage (ou topologie) des collections. De ce fait, on perd toutes les propriétés inhérentes aux topologies, et l'algorithme de filtrage n'en profite pas. Aussi, la complexité du filtrage est maximale.

Le principe du filtrage de MGS est fondé sur la dérivation de motif, une technique qui permet notamment de se déplacer dans la collection et de trouver un chemin par induction sur la construction du motif. Il présente plusieurs défauts, avec une recherche exhaustive des chemins et des transformations récursives du motif.

Un nouvel algorithme a tout d'abord été mis au point ; il est fondé sur une séquencialisation des motifs et une présentation par classe qui permet une bonne évolutivité du code. J'ai participé à la conception et à la mise au point de la séquencialisation des termes de filtrages.

Pour pouvoir optimiser le filtrage, il a fallu ensuite analysé sa complexité. Cette analyse s'est traduite par la création d'une classification des motifs construite sur la machine de complexité minimale qui permet de résoudre le filtrage d'un motif en particulier. J'ai construit un algorithme fondé sur la construction inductive des motifs qui renvoie le type du motif. Ce type correspond au coût du filtrage d'un tel motif, indépendamment de la collection, et par conséquent à une classe de complexité.

Cependant, l'algorithme de filtrage est très sensible à la topologie de la collection. Si on veut faire une optimisation, on ne peut le faire qu'en spécialisant l'algorithme suivant les propriétés

de la relation de voisinage. On pourra ainsi récupérer ce qui a été perdu à cause de la généricité. Bien entendu, une telle optimisation doit respecter les principes imposés lors de la conception de MGS. Le comportement du langage vis-à-vis de l'utilisateur ne doit pas être modifié.

Enfin, j'ai mis au point une optimisation en suivant ces consignes. Elle est fondée sur une réduction du nombre des positions d'entrée du filtrage. En effet, l'algorithme de base teste s'il existe un chemin à partir de chaque position. L'optimisation consiste à trouver un sous-ensemble de positions par lesquelles les chemins doivent passer. Il faut pour cela que le motif présente des contraintes fortes, comme la présence de constantes dans le chemin.

Pour pouvoir profiter des propriétés des topologies, j'ai appliqué ce principe d'optimisation sur deux types de collections :

1. Les arbres modulo AC : les propriétés de commutativité et d'associativité de la relation de voisinage permettent de manipuler facilement le motif. Aussi, on déduit d'une analyse du motif une liste de contraintes (optimisant le filtrage) et on le réécrit en un nouveau motif équivalent mais optimisé. Les résultats sont impressionnants, notamment un exemple où la recherche est 225 fois plus rapide après la réécriture.
2. Les séquences : il existe de nombreux algorithmes de filtrage de mots ou d'expressions régulières dans des textes qui peuvent inspirer une optimisation. J'ai adapté l'algorithme de Boyer-Moore pour les séquences MGS. Cette optimisation fournit un gain en temps de calcul de l'ordre 25%.

Discussion

Il est intéressant de récapituler les défauts et avantages de ce que j'ai apporté à MGS durant ce stage. Nous allons donc discuter sur deux travaux originaux : la classification et l'optimisation du filtrage.

Classification des motifs

L'intérêt de la classification des motifs est de pouvoir étudier la comportement du filtrage suivant le motif et indépendamment de la collection. Le but est de prévoir si la durée d'une recherche est élevée ; autrement dit, de répondre à la question *quel est le coût du filtrage*.

La classification que je propose est fondée sur un algorithme de reconnaissance des chemins qui construit une machine (automate fini déterministe, automate à pile, etc.) qui permet cette reconnaissance. L'étude de la complexité de la machine permet d'obtenir des classes de complexité des motifs.

La construction de la machine, et par conséquent la recherche de la classe de complexité, se fait de façon inductive sur la construction du motif. L'algorithme fournit non pas une valeur numérique évaluant le coût du filtre, mais un type correspondant à la classe de complexité à laquelle appartient le motif.

Les résultats fournis par l'algorithme de classification sont satisfaisants à quelques exceptions près. En effet, le motif $1, x^*$ reconnaît tous les chemins commençant par 1. Or l'algorithme le place dans la classe des filtres les plus coûteux soutenant qu'il est nécessaire d'utiliser une pile pour gérer l'itération. La classification reste tout de même utilisable.

Il existe néanmoins un défaut plus profond situé à la base de la classification. L'utilisation de la complexité du problème de reconnaissance n'est peut être pas adaptée l'évaluation de la complexité du problème de recherche. De plus, on ne se fonde pas sur l'algorithme de filtrage tel qu'il est implémenté dans MGS. Tout ceci nous amènerait à faire une autre classification fondée

sur cet algorithme.

Le stage ne s'est pas orienté dans cette voie. L'étude de l'algorithme de filtrage montre en effet que sa complexité dépend de façon plus importante de la topologie de la collection que du motif. On s'est donc tourné vers l'optimisation du filtrage en fonction de la topologie.

Optimisation du filtrage

La généricité du filtrage dans `MGS` empêche de pouvoir profiter pleinement des propriétés de la topologie de certaines collections. Aussi, pour pouvoir optimiser le filtrage, il faut spécialiser l'algorithme générique.

Je propose une optimisation qui tente de rechercher dans le motif dans des éléments contraignants (*i.e.* qui impose un propriété forte sur l'élément filtré), comme par exemple les constantes afin de réduire le nombre d'itérations du filtrage. En effet, celui-ci balaye toutes les positions, alors que l'optimisation permet de n'avoir q'un sous-ensemble de position comme point de départ.

Cette optimisation est appliquée aux séquences par une généralisation de l'algorithme de Boyer-Moore qui prend appuie sur de tels éléments, et aux arbres modulo AC par la réécriture de motif qui permet de faire remonter les éléments contraignant pour les tester en premier. L'optimisation donne de bons résultats dans les deux cas.

On peut regretter qu'il n'y ait pas de définition précise de la contrainte imposée par un élément du filtre. Jusqu'ici, on n'a considéré que les constantes étaient des bons pivots pour diviser le motif en deux. Malheureusement, il est assez rare de trouver de tels éléments dans un filtre ; les éléments des systèmes modélisés sont généralement représentés par des enregistrements, et les règles utilisent plutôt des gardes. Le problème est qu'il est difficile de savoir d'une garde si elle est contraignante : comment vérifier de façon automatique, quelle garde est la plus contraignante entre $x/(x > 5)$ et $x/(x == 5)$?

Je n'ai pas été plus loin de l'étude de la contrainte imposée par un filtre par manque de temps. Cependant, l'idée de comparer des filtres suivant leur contrainte peut amener à une nouvelle classification des motifs.

Perspectives

Je vais conclure ce rapport par les ouvertures possibles qui apparaissent autour de `MGS`, et en particulier du filtrage. Dans un premier temps, je vais présenter différentes voies qui pourraient apporter un complément à mon travail. Enfin, je présenterai le véritable but sous-jacent à l'étude des motifs dans `MGS`.

Poursuite du travail entrepris

Nous avons vu pour les séquences qu'il pouvait être intéressant d'utiliser des algorithmes de filtrage connus étudiés dans différents domaines. L'objectif serait d'intégrer des algorithmes connus et optimaux à `MGS` dans les cas où on peut s'y ramener. Le problème n'est alors pas d'intégrer de nouveaux algorithmes, ni de les trouver. Il s'agit plutôt de déterminer à quel problème connu on peut se ramener juste en observant les propriétés du motif et de la topologie. Pour cela deux étapes sont nécessaires. La première consiste à trouver les algorithmes spécialisés. Le deuxième est de trouver le lien entre ces problèmes et `MGS`. La plupart du temps, on espère que ces algorithmes correspondent à des cas particuliers de filtrage dans `MGS` qu'on pourrait détecter.

Parmi les différents travaux tournant autour du filtrage, quelques uns sont particulièrement intéressants :

1. En considérant les graphes comme des structures logiques, on peut exprimer formellement leurs propriétés par des formules logiques. On peut également décrire des classes de graphes par une formule écrite dans une logique appropriée permettant d'exprimer leurs propriétés caractéristiques. C'est sur ces problèmes qu'à travailler Bruno COURCELLE (LaBRI de Bordeaux , *cf.* [Cou97]). Ses motivations sont de donner une caractérisation logique aux classes de complexités et d'utiliser ces formules logiques comme d'un support, comparable aux grammaires et aux automates, pour spécifier des classes de complexité des graphes et d'établir des propriétés sur ces classes à partir de leur description logique.

   Ces travaux amènent à des résultats intéressants tels que l'étroite relation qu'il existe entre la logique monadique du second ordre et les grammaires de graphe *context-free*. Quant à la logique du premier ordre, elle permet de caractériser une sous-classe des langages réguliers qui sont des langages testables localement.

   Deux résultats peuvent nous intéresser :
   - Les classes de complexité mises en évidence par la méthode de COURCELLE sont-elles liées à notre classification ?
   - Pour les éléments étoilés et gardés, il se peut qu'on puisse classer les gardes suivant leurs propriétés logiques, et donc classer de tel motif. Par exemple, on a envie de dire que

     $$(\_^* \ as \ X)/(\text{ la longueur de } X \text{ est un nombre premier})$$

     est plus complexe que

     $$(\_^* \ as \ X)/(\text{ la longueur de } X \text{ est paire}),$$

     lui-même plus complexe que

     $$(\_^* \ as \ X)/(\text{ la longueur de } X \text{ est supérieure à } 5).$$

2. Les données semi-structurées ([AQM+96] et [AV97]) ont connu une poussée de popularité à la lumière de leurs diverses applications telles que la gestion de plusieurs formes de données (comme par exemple les données XML), intégrant des sources de données hétérogènes. La structure des données semi-structurées est irrégulière, partiellement connue, ou sujette à des changements fréquents. C'est à cause de ces propriétés particulières que pour formuler des requêtes sans spécifier la structure exacte des données, les expressions régulières sont essentielles. L'évaluation de ces expressions régulières traverse plus de sommets qu'il n'est nécessaire. Supprimer les parcours inutiles serait une importante optimisation. On peut trouver dans plusieurs travaux des algorithmes développant des techniques plus ou moins optimisées pour répondre à ces problèmes, notamment en ce qui concerne XML ([CK02]).

   Or, il s'avère en fait que les données de `MGS` sont semi-structurées et qu'on cherche à résoudre des requêtes écrites sous forme d'expressions (plus ou moins) régulières. Il serait donc judicieux d'essayer de rapprocher les travaux fait sur XML et sur les données semi-structurées en générale à la problématique de `MGS`.

3. Dans les langages fonctionnels, on trouve une définition par cas du filtrage (comme dans *OCaml* par exemple). Beaucoup de projets de recherche visent à trouver les algorithmes les plus optimisés possibles pour tester ces cas. On peut reprendre le travail de Manuel SERRANO sur `Bigloo` (*Scheme*, cf [M.]).

   En ce qui concerne plus particulièrement le compilateur d'*Objective-Caml*, plusieurs optimisations ont été introduites ; elles sont fondées sur des permutations de règles, l'utilisation d'informations sur le caractère exhaustif des filtres, et l'utilisation d'exceptions statiques

et d'informations sur le contexte. Elles ont permis d'accélérer les exécutables sans pour autant grossir la taille du code. Ces techniques sont détaillées dans [LFM02]. Il est possible que les mêmes améliorations puissent être apportées à la compilation des transformations de MGS.

4. La réécriture de termes est aussi une source d'algorithmes optimisés ; on peut notamment signaler dans le projet ELAN, les travaux de Pierre-Etienne MOREAU ([Mor]) sur la compilation du filtrage de terme modulo AC.
   L'une des idées d'optimisation étudiée par l'équipe d'ELAN est la construction d'automates pour effectuer les recherches. Ils tentent de fusionner les automates de chaque règle par des moyens plus ou moins naturels (l'idée est de regrouper ce qui est commun aux différentes règles). Ils obtiennent alors un automate optimisé qui représente plusieurs règles en même temps.

Vers les collections topologiques et le filtrage de dimension supérieure

Le filtrage non-conventionnel que nous avons commencé à étudier dans MGS, permet de modéliser des systèmes dynamiques dont la structure est dynamique. Ce type de systèmes, très difficile à modéliser et à simuler, se rencontre en biologie du développement comme nous l'avons vu dans les exemples, mais aussi à un niveau moléculaire pour la représentation de la conformation des protéines (exemple des POT graphes) ou bien dans la modélisations des cascades biochimiques. En dehors de la biologie, ce type de systèmes se rencontre aussi dans la modélisation des flots sur un réseau dynamique (trafic routier, internet, ...), dans la modélisation des systèmes mécaniques déformables (modélisation des couches géologiques, simulation des corps mous, animations), etc.

L'approche topologique du filtrage développé par le projet MGS offre donc un outil puissant permettant de résoudre de manière très concise toutes sortes de problèmes. Les algorithmes de filtrage permettent de déterminer les configurations d'entités biologiques en interaction (les motifs). Sur les structures linéaires (comme par exemple la séquence), ces algorithmes sont bien connus et maîtrisés. On dispose en particulier d'une classification de la sophistication des motifs qui permet de s'orienter sur le choix de tel ou tel autre algorithme de recherche. Ce n'est absolument pas le cas pour des structures plus complexes comme les GBF ou les graphes de Delaunay. Le travail présenté dans cette thèse est un premier pas pour étendre les résultats bien connus en bio-informatique sur les structures linéaires et très utilisés dans le traitement des séquences, à des structures fondamentalement plus riches. Ce premier pas doit bien sur être poursuivis.

Cependant, les collections topologiques disponibles actuellement en MGS limitent toujours les applications envisageables. Il faudrait en effet pouvoir représenter des relations spatiales plus complexes. Un des enjeux du projet MGS est donc à présent de développer des collections topologiques de dimension supérieure, capable de représenter des relations spatiales tridimensionelles. On a déjà vu la nécessité des organisations spatiales tri-dimensionnelles avec la modélisation de la croissance du méristème. De ce point de vue, la topologie induite par le calcul de la partition de DELAUNAYE est une première solution, qui reste toutefois bien en deçà des besoins géométriques. Par exemple, la représentation d'un appareil de GOLGIE, qui présente des "trous", ne peux passer par ce type d'outil.

Le passage aux dimensions supérieures, qui serait si utile pour les applications, n'est bien sur possible que si l'on peut concevoir un langage de filtres permettant de spécifier des règles de transformations dans ces structures topologiques, d'étudier les propriétés de ces filtres (en particulier les typer) et de concevoir des algorithmes de filtrage efficaces. Ce domaine de recherche est encore complètement vierge.

Bibliographie

[AQM+96] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The lorel query language for semistructured data. *International Journal on Digital Libraities*, 1996.

[AV97] S. Abiteboul and V. Vianu. Regular path queries with constraints. *Proceedings of ACM Symposium on Principles of Database Systems*, 1997.

[BB89] G. Berry and G. Boudol. *The chemical abstract machine*. 1989.

[BH00] R. Brown and A. Heyworth. Using rewriting systems to compute left kan extensions and induced actions of categories. *Journal of Symbolic Computation*, 2000.

[BL74] J. Bard and I. Lauder. How well does turing's theory of morphogenesis work ? *Journal of Theoretical Biology*, (45) :507–531, 1974.

[BLM86] J. P. Banâtre and D. Le Métayer. A new computational model and its discipline of programming. 1986.

[BLM91] J. P. Banâtre and D. Le Métayer. Introduction to gamma. *Computer Science*, (574) :197–202, 1991.

[BM77] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Comm. ACM*, pages 62–72, 1977.

[Brz64] J. A. Brzozowski. Derivatives of regular expression. *JACM*, 11 :481–494, 1964.

[Bug] A. E. Bugrim. Logic-based approach to the analysis of spatially distributed biochemical networks. *Center for computationnal Biology*.

[Cha02] Gregory J. Chaitin. Meta-mathematics and the foundations of mathematics. *Bulletin of the European Association for Theoretical Computer Science*, 77 :167–179, June 2002.

[CK02] T.-S. Chung and H.-J. Kim. A two phase optimization technique for xml queries with multiple regular path expressions. *The Journal of Systems and Software*, pages 183–193, 2002.

[CL] C. Charras and T. Lecroq. http ://www-igm.univ-mlv.fr/lecroq/string/index.html.

[Cou97] B. Courcelle. *Handbook of graph grammars and computing by graph transformations*, volume 1, chapter The Expression Of Graph Properties and Graph Transformations in Monadic Second-order Logic, pages 313–400. 1997.

[Dit01] P. Dittrich. Artificial chemistry webpage. `ls11-www.cs.uni-dortmund.de/achem`, 2001.

[Dor92] V. Dornic. *Analyse de Complexité des programmes : Vérification et Inférence*. PhD thesis, Université Paris VI, 1992.

[DZB00] P. Dittrich, Jens Ziegler, and Wolfgang Banzhaf. Artificial chemistries - a review. *Artificial Life*, 2000. (to be submitted, available from the authors, cf [Dit01]).

[FMP00] M. Fisher, G. Malcolm, and R. Paton. patio-logical processes in intracellular signalling. *BioSystems*, pages 83–92, 2000.

[GGMP02] J.-L. Giavitto, C. Godin, O. Michel, and P. Prusinkiewicz. *Modelling and Simulation of biological processes in the context of genomics*, chapter "Computational Models for Integrative and Developmental Biology". Genopole Evry, July 2002. (final proceedings and tutorials).

[Gia00] Jean-Louis Giavitto. A framework for the recursive definition of data structures. In *Proceedings of the 2nd Imternational ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP-00)*, pages 45–55. ACM Press, 2000.

[GM01a] Jean-Louis Giavitto and Olivier Michel. Declarative definition of group indexed data structures and approximation of their domains. In *Proceedings of the 3nd Imternational ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP-01)*. ACM Press, 2001.

[GM01b] Jean-Louis Giavitto and Olivier Michel. Mgs : Tutorial du langage mgs. Technical report, LaMI – Université d'Évry Val d'Essonne, 2001.

[GM01c] Jean-Louis Giavitto and Olivier Michel. `MGS` : a programming language for the transformations of topological collections. Technical Report 61-2001, LaMI – Université d'Évry Val d'Essonne, May 2001.

[GM03] J.-L. Giavitto and O. Michel. Modeling the topological organization of cellular processes. *BioSystems*, 2003. (accepted for publication).

[GMS96] Jean-Louis Giavitto, Olivier Michel, and Jean-Paul Sansonnet. Group-based fields. In *Parallel Symbolic Languages and Systems (Int. Workshop PSLS'95)*, volume LNCS 1068, pages 209–215. Springer, 1996.

[GWNT99] D. Gilbert, D.R. Westhead, N. Nagano, and J.M. Thornton. Motif-based searching in tops protein topology databases. *bioinformatics*, (15) :317–326, 1999.

[HHLM99] L. H. Hartwell, J. J. Hopfield, S. Leibler, and A. W. Murray. From molecular to molecular cell biology. *Nature*, 402 :47–52, 1999.

[Hum98] A. Hume. Grep wars. In *Spring(London) EUUG Conference Proceedings*, pages 237–245, 1998.

[LFM02] F. Le Fessant and L. Maranget. Optimizingpattern matching. 2002.

[M.] Serrano M. http ://www-sop.inria.fr/mimosa/personnel/Manuel.Serrano/diffusion/publi.html.

[Man01] Vincenzo Manca. Logical string rewriting. *Theoretical Computer Science*, 264 :25–51, 2001.

[Mor] P.-E. Moreau. http ://www.loria.fr/equipes/protheo/SOFTWARES/ELAN/.

[NSF91] Grand challenges : High performance computing and communications. A Report by the Committee on Physical, Mathematical and Engineering Sciences, NSF/CISE, 1800 G Street NW, Washington, DC 20550, 1991.

[Pau98a] Gheorge Paun, editor. *Computing with Bio-Molecules : Theory and Experiments.* Springer, 1998.

[Pau98b] Gheorghe Paun. Computing with membranes. Technical Report TUCS-TR-208, TUCS - Turku Centre for Computer Science, November 11 1998.

[PH97] A. V. Paniflov and A. V. Holden, editors. *Computational biology of the heart.* John Wiley and Sons, 1997.

[PL90a] P. Prusinkiewicz and A. Lindenmayer. *The algorithmic beauty of plants.* Springer-Verlag, New York, 1990. With J. S. Hanan, F. D. Fracchia, D. R. Fowler, M. J. M. de Boer, and L. Mercer.

[PL90b] P. Prusinkiewicz and A. Lindenmayer. The algorithmic beauty of plants. Springer-Verlag, New York, 1990.

[Pru93] P. Prusinkiewicz. Modeling and visualization of biological structures. In *Graphics Interface '93*, pages 128–137, 1993.

[RS92] Grzegorz Rozenberg and Arto Salomaa. *Lindenmayer Systems.* Springer, Berlin, 1992.

[TBCN00] J.J. Tyson, M.T. Borisuk, K. Chen, and B. Novak. *Computational Modeling of Genetic and Biochemical Networks*, chapter Analysis of Complex Dynamics in Cell Cycle Regulation, pages 287–306. MIT Press, 2000.

[Tur52] A.M. Turing. The chemical basis of morphogenesis. *hil. Trans. Royal Society*, B237 :37–72, 1952.

[TVP99] J. Tabony, L. Vuillard, and C. Papaseit. Topological self-organisation and pattern formation by way of microtubule reaction-diffusion processes. *Advances in Complex Systems*, 2(3) :221–276, 1999.

[VG] J. Viksna and D. Gilbert. Pattern matching and pattern discovery algorithms for protein topologies.

[VN66] J. Von Neumann. *Theory of Self-Reproducing Automata.* Univ. of Illinois Press, 1966.

[Wat01a] B. W. Watson. A collection of new regular grammar pattern matching algorithm. 2001.

[Wat01b] B. W. Watson. A new algorithm for regular grammar pattern matching. the Fourth European Symposium on Algorithms, 2001.

[Wol02] Stephen Wolfram. *A new kind of science.* Wolfram Media, 2002.