Rapport Final
de
Projet de Programmation

Petru Valicov, Mathieu Nagle, Michaël Ramamonjisoa Christophe Delmotte

Clients :

M. Bruno Courcelle
M. Mamadou Kante

Chargé de TD.
M. Nicolas Bonichon

Remerciement

Nous tenons à remercier nos clients et plus particulièrement M Kanté pour ses explications et le temps qu'il a bien voulu nous a consacré. Nous remercions aussi M Bonichon pour ses conseils.

Résumé du sujet

Il existe des polynômes qui permettent de caractériser des graphes. MM. Kanté et Courcelle, nos clients, étudient en particulier les définitions récursives de certains des polynômes associés à des graphes. Le but du projet est de développer un logiciel pour automatiser les calculs fastidieux et facilement entachés d'erreurs de ces polynômes. Notre travail consiste à fournir un outil de calcul et d'analyse de ces polynômes, ainsi que son interface graphique. L'ensemble des fonctionnnalités demandées et apparues lors du déroulement du projet a été réalisé.

Table des matières

Chapitre 1

Analyse du sujet

1.1 Contexte

Les graphes constituent un des thèmes centraux de recherche en mathématiques-informatique. Ces objets combinatoires représentent un concept assez riche pour modéliser de nombreux problèmes. Ainsi, étudier ces problèmes revient à étudier certaines propriétés des graphes (les circuits eulériens, la coloration, la planarité etc.). Les polynômes associés aux graphes sont un des moyens pour étudier des propriétés sur les graphes, notamment le comptage de configurations (par exemple le nombre de colorations avec k couleurs ou le nombre d'arbres couvrants). A priori on ne se limite pas qu'à cette utilisation (le polynôme multivarié de Sokal [SOK05] permet aussi d'énumérer les configurations).

La notion de polynôme associé à un graphe a été introduite par G.D. Birkhoff et D.C. Lewis en 1946 [eLD46] avec les *polynômes chromatiques*. Un tel polynôme permet de déterminer le nombre de façons dont on peut colorer un graphe avec k couleurs (colorer les sommets du graphe de telle sorte que 2 sommets adjacents n'aient pas la même couleur). Plus exactement une *k-coloration* d'un graphe G=(V,E) c'est une fonction f :V→{1,...,k} tel que ∀ (u,v)∈E, f(u)≠f(v). On définit le polynôme chromatique : P(G,k)=# {k-coloration de G}. On peut le représenter sous une forme de définition récursive en opérant sur les arêtes du graphe en paramètre - une première approche récursive :

- $P(G,x) = x^n$, si le graphe G possède n sommets isolés
- $P(G,x) = P(G-e,x) - P(G/e,x)$, sinon

où $G-e$ supprimé l'arête e dans G et G/e réduit les deux sommets extrémités de l'arête e dans un seul sommet (contraction).

Ces études ont été notamment reprises par Tutte dans les années 1950-1970, qui a introduit le *polynôme de Tutte*. Ce polynôme représente une généralisation du polynôme chromatique sachant qu'il porte sur deux variables (contrairement au polynôme chromatique). Il est défini comme série génératrice des sous-graphes comptés selon le nombre d'arêtes et de composantes connexes [BER06] :

$$T(G;x,y) = \sum_{H \subseteq G} (x-1)^{c(H)-c}(y-1)^{|H|+c(H)-s}$$

où s est le nombre de sommets, $|H|$ est le nombre d'arêtes du sous-graphe H de G et $c(H)$ le nombre de composantes connexes du H.

On peut utiliser les relations de récurrences pour caractériser le polynôme de Tutte :

- $T(G;x,y) = x^i y^j$, si le graphe G ne contient que des isthmes (arêtes telles que leur suppression augmente le nombre de composantes connexes) et des boucles, i et j représentent le nombre d'isthmes et de boucles respectivement.
- $T(G;x,y) = T(G-e;x,y) + T(G/e;x,y)$, e étant une arête quelconque qui n'est pas ni un isthme, ni une boucle ;.

On remarque que le résultat du calcul du polynôme de Tutte reste le même quelque soit l'arête a satisfaisant la condition considérée. Un résultat important est que toute fonction portant sur un graphe et satisfaisant les deux propriétés énoncées ci-dessus, représente l'évaluation du polynôme de Tutte. Ce polynôme permet ainsi

de compter plusieurs configurations. Par exemple, pour $x = 1$ et $y = 1$ l'évaluation du polynôme représente le nombre d'arbres couvrants.

Récemment dans les années 2000, Richard Arratia, Bela Bollobas et Gregory B. Sorkin [SOR04] ont défini le *polynôme entrelacé* (à une et à deux variables). Leur but était le calcul du nombre de circuits eulériens, ceci pouvant aider à la résolution des problèmes de séquençage d'ADN. Le polynôme entrelacé à deux variables a une structure identique à celle du polynôme de Tutte sachant que les conditions portent sur les sommets et non pas sur les arêtes. Les opérations sur les sommets seront abordées plus en détails lors de la description du sujet.

Dernièrement B. Courcelle [COU07] a introduit *les polynômes entrelacés à plusieurs variables* généralisant ainsi certains polynômes entrelacés. Les indéterminées du polynôme multivarié dépendent des sommets ou des arêtes du graphe (le polynôme multivarié de Tutte défini par Sokal [SOK05] porte sur les arêtes), mais ils peuvent être associés à aucun élément du graphe. Pour tout graphe G , avec V son ensemble de sommets, on a :

$$P(G) = \sum_{A \subseteq V} x_A \cdot u^{rk(G[A])}$$

où $rk(G[A])$ est le rang de la matrice d'adjacence associée au sous-graphe de G induit par l'ensemble de sommets A.
La définition d'un polynôme entrelacé peut être représenté d'une manière récursive. Voici un exemple :

- $P(G) = 1$ si G est vide,
- $P(G) = (1 + x_a) \cdot P(G - a)$ si a est isolé sans boucle,
- $P(G) = x_a u \cdot P(bc(G, a) - a) + P(G - a)$ si a est incident à une boucle, isolé ou non,
- $P(G) = x_a x_b u^2 \cdot P(p(G, a, b) - a - b) + P(G - a) + P(p(G, a, b) - b)$ si a est adjacent à b, a et b sans boucles.

Ici x_a représente une variable associée au sommet a et u - une variable ordinaire (pour tous les sommets).
Les indéterminées peuvent être substituées par d'autres polynômes pour obtenir certains polynômes connus, cela pour avoir un lien entre ces objets mathématiques.

1.2 Présentation du sujet

Le logiciel représente un utilitaire, qui dans des phases exploratoires permet de tester plus rapidement des hypothèses sur les familles de polynômes trouvées, pour certains types de graphes. L'exemple fourni en annexe, nous éclaire bien en effet sur la quantité de calculs manuels induite pour un simple graphe de 5 sommets.

Le logiciel est principalement destiné à être utilisé par des chercheurs du LaBRI. B. Courcelle et M. Kante font partie du groupe *Méthodes Formelles* et traitent en particulier de la thématique *Logique Graphes et Langages (LGL)*. La problématique des graphes est à des degrés divers commune à plusieurs domaines de recherche, il n'est donc pas exclu que le cadre des utilisateurs dépasse celui des clients initiaux. De plus la mise en commun des moyens et des ressources étant de mise dans le monde de la recherche, il est souhaité que le logiciel soit accessible sur le site du LaBRI sous-forme d'une applet.

Pour entamer la description en détail de l'application nous remarquons qu'un scénario type d'exécution du logiciel se décompose en quatre phases :

- Une phase de définition d'un *graphe*
- Une phase de *définition récursive de polynôme*, (analyse syntaxique de la définition saisie par l'utilisateur).
- Une phase d'interprétation de *définition récursive de polynôme* sur un graphe (calcul)
- Une phase d'analyse et de transformation de polynôme obtenu.

Dans tout ce qui va suivre, nous considérons des graphes simples non-orientés avec potentiellement des bo[width=15]ucles. Nous admettrons tout d'abord que les graphes dont il s'agit, sont représentés sans ambiguïté par une *matrice d'adjacence* de booléens (valeurs dans $\{0, 1\}$).

1.2.1 Opérations élémentaires sur les graphes

Toutes les opérations décrites ci-dessous représentent les opérations élémentaires qui sont utilisées, pour former une *définition récursive de polynôme*. Il existe trois types d'opérations sur un graphe G.

- des opérations de *suppression de sommets du graphes*
- des opérations de *sélection de sommets du graphe*

- des opérations de *complémentation* du graphe, après sélection ou suppression de sommets portant sur sa matrice d'adjacence restreinte.

Dans tout ce qui suit, par *complément d'arêtes entre deux sommets a et b*, on entend : [width=15]

- Suppression de l'arête entre a et b si elle existe : $A_G(a,b) = A_G(b,a) = 0$
- ou création de l'arête entre a et b si elle n'existe pas : $A_G(a,b) = A_G(b,a) = 1$

Sélection par Voisinage du sommet a :$N(G,a)$

L'opérateur voisinage permet de sélectionner les sommets voisins de a (a exclu). Cet opérateur permet de localiser l'effet des opérateurs de complémentation à une certaine partie du graphe.

Suppression d'un sommet : $(G-a)$

Le graphe $H = (G-a)$ est un sous-graphe induit par l'ensemble $V_G - a$ (V_G : ensemble des sommets de G, auquel on a supprimé le sommet a et toutes les arêtes incidentes). La matrice d'adjacence A_H est égale à A_G à laquelle on a supprimé la ligne et la colonne du sommet a.

Suppression d'un ensemble de sommets X : $(G-X)$

Soit X un ensemble de sommets de G. Le graphe $H = (G-X)$ est un sous-graphe induit par l'ensemble $V_G - X$ (V_G auquel on a supprimé l'ensemble de sommets X et toutes les arêtes incidentes à ces sommets). La matrice d'adjacence A_H est égale à A_G à laquelle on a supprimé les lignes et les colonnes des sommet de X.

Complément des boucles des sommets X de G : $inv(G,X)$

Crée ou supprime des boucles sur les sommets de X. Sur la matrice d'adjacence cela se traduit par un complément des valeurs de la diagonale correspondant aux sommets de X.

Complément des sommets sans toucher aux boucles $c(G)$

Supprime (resp. crée) des arêtes si elles existent (resp. n'existent pas). Sur la matrice d'adjacence cela se traduit par un complément de toutes les valeurs de la matrice d'adjacence exceptées celles de la diagonale.

Complément local : $lc(G,a)$ **ou** $G * a$

Supprime (resp. crée) des arêtes existantes (resp. absentes) entre les sommets incidents au sommet a. Cela veut dire que l'on complémente les arêtes entre les voisins de a. Remarque : Cela ne concerne pas les boucles.

Complément local étendu aux boucles : $bc(G)$ **ou** G^a

Idem que pour *le Complément local* en l'étendant aux boucles.

Pivotage : $p(G,a,b)$ **ou** G^{ab}

Les voisins des sommets a et b définissent une partition de l'ensemble des sommets en 4 ensembles disjoints.

- Les sommets voisins de a exclusivement
- Les sommets voisins de b exclusivement
- Les sommets voisins de a et de b
- Les sommets voisins ni de a ni de b

Le pivotage complémente les arêtes reliant des sommets figurant dans 2 catégories distinctes parmi les trois premières, sans toucher ni aux boucles, ni aux arêtes incidentes à a ou à b.

Rang du graphe *rk(G)*

Le rang du graphe c'est le rang de la matrice d'adjacence.

Nombre de composantes connexes cc(G)

Le nombre des composantes connexes du graphe.

1.2.2 Définition récursive d'un polynôme

La définition récursive fait intervenir un ensemble fini d'*opérations prédéfinies (ou de fonctions élémentaires)* ainsi que la règle d'application propre à chaque opération. Le couple *(fonction, règle)* est dénommé *clause*. Compte tenu des remarques faites en début de chapitre, il est nécessaire de préciser que la syntaxe de ces clauses, figée pour l'instant à un certain nombres d'opérations et à un alphabet, détermine complètement l'expressivité du langage associé, c'est à dire son aptitude en un minimum d'opérations à définir une famille de polynômes.

1.2.3 Les variables du polynôme

Les polynômes traités comportent deux types de variables. Les *indéterminées ordinaires* et les *indéterminées associées aux sommets* :

- les indéterminées ordinaires, variables structurelles du polynôme (ex : $x, y, z...$)
- les indéterminées associées aux sommets, ce sont des variables propres à chaque sommet, pour les distinguer, ces variables seront indicées par le nom du sommet auquel elles se réfèrent (x_a, y_a sont des variables associées au sommet a). Elles peuvent représenter par exemple un "poids" attaché au sommet ou un "potentiel" électrique.

En résumé, ce qui les différencie est uniquement le fait que les indéterminées associées aux sommets sont indicées par le nom du sommet qui les porte (ils ont une portée locale), alors que les *indéterminées ordinaires* ont une signification globale. Voici un exemple d'un polynôme qui fait intervenir les deux types de variables :

- $P(G) = 5 + x_0 + x_1 + x^2 * x_0 * x_1 + 2 * x_2$

Chacune des variables pourra être utilisée lors de l'évaluation (sous condition) d'une clause dans laquelle le sommet considéré est sélectionné (la question de l'ordonnancement des clauses et de l'éligibilité des sommets, sera abordée dans le chapitre des besoins fonctionnels).

1.2.4 Syntaxe d'une clause de définition récursive

Une définition récursive est constituée d'un ensemble fini de clauses ; c'est une définition par cas. par exemple : Soit G un graphe, une définition récursive d'un polynôme $P(G)$ peut s'écrire.

$$P(G) = \begin{cases} 1 & si \quad G = \varnothing \\ 10.P(G-a).x & si \quad \text{a est un sommet isolé} \\ y.P(G-a).P(G-b) & si \quad \text{a et b sont adjacents} \end{cases}$$

où x et y sont des variables ordinaires. On distingue *l'expression* (la valeur prise par la fonction) et la *condition* pour laquelle cette expression s'applique. Donc chaque clause a la forme suivante :

$$P(G) = E(G, a, b, ..., c) \text{ si } C(G, a, b, ..., c)$$

La condition

Une condition porte sur le graphe et les propriétés des sommets $a, b, ...c$. Les conditions possibles sont : $G = \varnothing$ (G n'a aucun sommet), ou toute combinaison booléenne formée par les *conditions élémentaires* suivantes :

- $N(G, a)$ est vide (a est un sommet isolé avec ou sans boucle).
- a est incident à une boucle (a possède une boucle).
- a et b sont adjacents

Les opérations booléennes élémentaires prises en compte sont :

- La négation
- La conjonction
- la disjonction

Exemple $C(a, b, c) = c$ incident à une boucle $\wedge$ c et a adjacents $\wedge$ c et b adjacents, est une condition portant sur les sommets a, b et c.

ordre	**arité**.	**Condition**	**Expression**
`0`	0	C_0	E_0
`1`	1	C_1	E_1
`2`	1	C_2	E_2
...	...	...	...
`n`	1	C_n	E_n
`n+1`	2	C_{n+1}	E_{n+1}
`n+2`	2	C_{n+2}	E_{n+2}
...	...	...	...
`m`	2	C_m	E_m
`m+1`	3	C_{m+1}	E_{m+1}
...	...	...	...

FIG. 1.1 – Tableau de clauses ordonnées en fonction de l'arité

L'expression

Une expression porte généralement sur les sommets a,b ..., c qui interviennent dans la condition de la clause correspondante. E(G,a,b,...,c) est de la forme :

- 0
- Une somme finie *d'expressions élémentaires.*

L'expression permet de calculer le polynôme associé au graphe auquel on applique une opération, par exemple l'inversion de la matrice d'adjacence. Les *expressions élémentaires* sont des produits constitués :

- de polynômes d'indéterminées (ordinaires ou associées aux sommets) : $P_i(G) = (1 + x_a)y_a y_b y x^2$
- de polynômes résultant du calcul récursif portant sur les sous-graphes induits après application de la clause traitée : $P_p(G) = P(H(G, a, b, ..., c))$ où $H(G, a, b, ..., c)$ est un sous-graphe induit résultant de G.

Remarque C'est cette dernière forme d'expression élémentaire, qui justifie la nature récursive de la définition. La syntaxe de $H(G, a, b, ..., c)$ fait intervenir les opérations spécifiques sur les graphes, qui ont été définies précédemment ainsi que le langage associé.

1.2.5 Calcul

Le principe général de la récursion étant établi, l'objet de la récursion étant identifié : *l'expression élémentaire.* Intéressons nous à présent au déroulement d'une étape de ce calcul. Quatre entités interviennent dans ce calcul :

- des nombres entiers relatifs
- des opérations arithmétiques :**addition, soustraction et multiplication et exposant**
- des polynômes d'indéterminées, qui sont formalisés dans l'expression d'une clause.
- des polynômes résultats des récursions sur les sous-graphes qui au moment où se déroule le calcul sont également des polynômes d'indéterminées.

Pour effectuer le calcul, les clauses doivent être ordonnées en fonction du nombre de sommets différents intervenant dans la condition (par la suite appelé **arité**) :

Le calcul de base est donc tout simplement une addition ou un produit de polynômes. Son algorithme général se décrit ainsi pour un graphe donné :

```
Recherche de la première condition satisfaite
Analyse syntaxique en évaluant l'expression correspondante
Parcours du sous-arbre créé par chaque occurence d'un appel récursif dans l'expression
.. et on répète les trois opérations précédentes jusqu'à la condition d'arrêt.
```

Arrêt de la récursion

L'arrêt de la récursion, induit l'existence d'une condition de base dans la définition récursive. Dans les exemples fournis, il s'agit d'un test de vacuité de l'ensemble de sommets du graphe considéré. En fait toute ex-

pression non récursive, constituée par un polynôme d'indéterminées ordinaires, conduit à l'arrêt de la récursion. Mais si la condition correspondant à cette expression n'est jamais remplie le calcul récursif boucle.
Une autre condition nécessaire à l'arrêt est bien entendue que le calcul puisse conduire à un moment où un autre à des sous-graphes de taille inférieure. Cela se traduit au niveau des clauses, par au moins l'existence d'une suppression de sommet, dans la définition récursive.
Le fait qu'une condition donnée puisse être remplie lors du déroulement d'un calcul étant indécidable, le test de l'arrêt d'un calcul récursif l'est également. Conformément aux avis exprimés par les futurs utilisateurs, on considère que la définition récursive saisie est *correcte* du moins de ce point de vue.

Présentation du polynôme calculé

Le polynôme résultat sera affiché d'après l'ordre lexicographique des indéterminées des monômes. Pour les monômes composés des mêmes indéterminées, c'est alors le degré des indéterminées qui prévaudra.
Par exemples :
- $P(x) = x^2 * y * z + x * z + y$
- $Q(x) = x^2 * z + y^4 + z^2 + x_1 * y_1 + z_1$

Sauvegarde des résultats

Le résultat des calculs ainsi décrits, qui fait l'objet même des recherches que nous avons citées en introduction, est donc un polynôme qui devra tout d'abord être représenté, puis enregistré. Cette affirmation anodine, sous-entend naturellement en premier lieu qu'un alphabet soit formalisé, puis qu'une grammaire soit définie, et permette de concevoir des *polynômes bien formés*. Il convient donc naturellement de préciser cette grammaire. L'enregistrement du polynôme pourra alors être réalisé, a priori dans un fichier texte afin qu'il puisse être décodé sans ambiguïté. Ni la grammaire, ni le codage utilisé pour les fichiers ne sont imposés, il conviendra naturellement que ces formats soient en cohérence avec ceux définis pour les clauses.

Analyse et transformation du polynôme

Les besoins exprimés impliquent certaines manipulations particulières des polynômes obtenus. Cette partie est donc d'une certaine manière indépendante des traitements que nous venons de décrire. Ces opérations, dont l'importance est moindre, au regard des autres fonctionnalités, doivent néanmoins être prises en compte dès la conception, ne serait ce que pour permettre l'extensibilité du logiciel. Ces opérations, sont :
- la substitution d'indéterminées par un polynôme
- composition de polynômes par les opérations arithmétiques (addition, soustraction, multiplication et exposant)

Ces fonctionnalités, accessoires sont détaillées dans le chapitre relatif aux besoins fonctionnels.

1.3 Exemple de calcul

Soit le graphe G défini par la matrice d'adjacence A_G :

	1	2	3	4	5
1	0	1	1	1	1
2	1	0	0	1	1
3	1	0	0	1	0
4	1	1	1	0	0
5	1	1	0	0	0

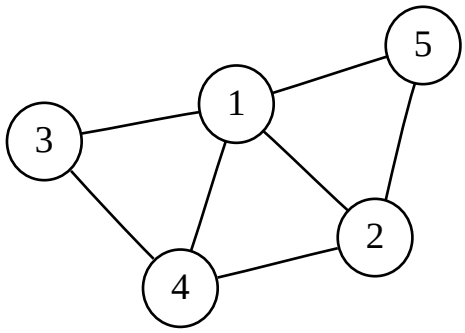


FIG. 1.2 – Une représentation de ce graphe

Soit $P(G)$, la définition récursive :

$$Q(G) = \begin{cases} 1 & si \quad G = \varnothing \\ x.Q(G-a) & si \quad \text{a est un sommet isolé} \\ Q(G-a) + Q(p(G,a,b)-b) & si \quad \text{a et b sont adjacents} \end{cases}$$

Le polynôme obtenu à partir de la définition récursive ci-dessus appliquée à G est : $Q(G) = 5x^2 + 6x$
Les opérations sur les graphes sont explicitées au début de chapitre et le détail.
Voici les premières étapes du calcul :

Polynôme $Q(G)$

Les sommets 1 et 2 sont adjacents. Par définition, on a donc :
$Q(G) = Q(G-1) + Q(p(G,1,2)-2)$

Appel : $Q(G-1)$

A_{G-1}

	2	3	4	5
2	0	0	1	1
3	0	0	1	0
4	1	1	0	0
5	1	0	0	0

Les sommets 2 et 5 sont adjacents. On a donc :
$Q(G-1) = Q((G-1)-2) + Q(p((G-1),2,5)-5)$
Pour simplifier l'expression, on note $G-1$ par B :
$Q(B) = Q(B-2) + Q(p(B,2,5)-5)$

Appel : $Q(B-2)$

A_{B-2}

	3	4	5
3	0	1	0
4	1	0	0
5	0	0	0

Le sommet 5 est isolé, donc :
$Q(B-2) = x * Q((B-2)-5)$
$B-2 = C$
$Q(C) = x * Q(C-5)$
le reste du calcul est présenté en annexes.

Chapitre 2

Analyse de l'existant

Notre projet, fait suite à une première tentative, l'année dernière sur le même sujet. Nous nous sommes appuyés sur cette précédente réalisation, afin d'éviter de reproduire les erreurs de conception qui ont fait que ce projet ne donne pas satisfaction.
Aussi, nous avons privilégié, la recherche d'outils directement exploitables, ciblée sur les points que nous avons jugés fondamentaux, à savoir, le calcul sur des polynômes et l'analyse syntaxique d'expressions.

2.1 Projet Pdp de l'année dernière

Une analyse neutre et objective du logiciel a été effectuée sur ce projet initial. Cet exercice nous a conduit naturellement à prendre un point de vue critique.
Il est d'ailleurs essentiel de noter, que l'existence d'une expérience précédente a certainement conduit le client à modifier ces mêmes exigences (même si l'énoncé du sujet est identique).

Cette analyse portera donc ensuite sur les choix faits par nos prédécesseurs ainsi que sur ce qu'ils induisent en termes de non-fonctionnalités au regard du cahier des charges qui est le nôtre aujourd'hui.

2.1.1 Non-conformités identifiées par le client

Non-conformités fonctionnelles La première des non-conformités citées est avant tout l'inexactitude des calculs générés. Cette inexactitude a d'ailleurs rapidement et naturellement conduit le client à délaisser l'usage de ce logiciel. Ce qu'il faut au moins retenir de cet état de fait, c'est qu'a priori le processus de conception n'a apparemment pas permis d'identifier ces erreurs de calcul. De plus l'absence des tests unitaires, rend très difficile l'identification des causes possibles des erreurs.

- Erreur de traitement ?
- Cas non prévu ?
- Défaut de conception ?

Les limites du langage défini, sont également pénalisantes aux dires des utilisateurs (nombre et noms des variables). C'est pour cela que la syntaxe du langage a été redéfinie.

Nous avons fini par réussir à faire fonctionner le programme sur Windows uniquement, sur un scénario simple (le premier exemple du sujet), cet essai nous a d'ailleurs permis de valider l'exemple de calcul fourni en annexe. Pour le deuxième exemple fourni, le programme a généré des erreurs différentes suivants les graphes en entrée (mêmes graphes que pour le jeu d'essai précédent). Ces erreurs, sous forme d'exceptions non traitées par le logiciel, font état de problèmes d'allocation (OutOfBoundException) dans des modules différents. On peut donc dire que le traitement des erreurs était insuffisant.

Interface existante

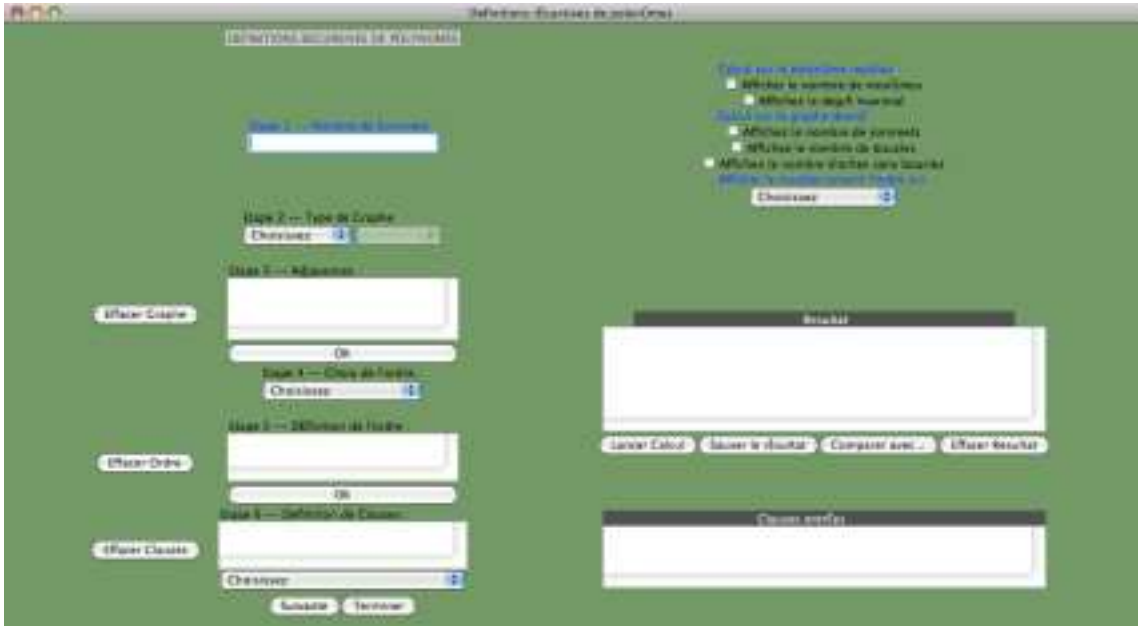


FIG. 2.1 – Aperçu de l'interface du projet existant.

Non-conformités non-fonctionnelles Une liste non-exhaustive des non-conformités est :
- Affichage incomplet de l'interface (sur PC linux et MAC), ne permettant pas d'accéder à certains boutons essentiels
- Saisie de graphe non conviviale (liste d'adjacence, double saisie obligatoire des adjacences : a-b et b-a).
- Syntaxe des clauses (émission permanente d'exceptions)
- A priori impossibilité de poursuivre une saisie après erreur ?
- enregistrement des polynômes/graphes/clauses non-fonctionnels.

Non-conformités non-structurelles La disproportion des trois paquetages proposés, conduit à penser que le paquetage *"Datas"* doté de 55 classes et interfaces, regroupant les classes liées aux graphes, aux clauses et aux calculs auraient mérité d'être allégé. Les clauses et tout ce qui à trait à la syntaxe notamment pour des raisons de maintenabilité et de cohérence auraient pues être associées à l'analyseur.

2.1.2 Analyseur

Les principales non-fonctionnalités structurelles relevées concernent l'analyseur. L'analyseur de clauses développé est basé sur un analyseur de formules mathématiques pré-existant. Outre le fait que l'origine de cet analyseur ait conduit à polluer le code avec des fonctions mathématiques qui ne sont d'aucune utilité (sin, cos ,exp et log) du fait que les nombres manipulés sont uniquement des entiers relatifs, il ne s'avère a priori pas adapté au besoin d'expressivité nécessité par le langage de définition des polynômes récursifs. Comme nous l'avons vu dans le chapitre d'introduction, ce langage de définition de clauses, est par nature le module susceptible d'évoluer le plus, afin d'enrichir son expressivité et par la même les capacités futures de l'outil. C'est de toute évidence la pierre angulaire de l'édifice qu'il convient d'adapter et de maîtriser. A ce titre, l'importance de ce module mérite la définition d'une *grammaire* au sens formel. En effet, la grammaire définie est plutôt rigide car apparemment de type *top-down*. Or la syntaxe requise n'est certainement pas LL(1) et justifie donc pleinement l'usage d'un analyseur *bottom-up*, ne serait-ce que pour les problèmes de parenthésage. Une analyse rapide du code suffit à se rendre compte que l'usage intensif de *if-else* lors de l'analyse syntaxique d'une clause, est difficilement *debuggable* mais surtout il s'avère que le produit résultant risque fort d'être peu évolutif.

2.1.3 Graphes

L'importance des graphes, à notre avis, a été surestimée, dans ce projet, au détriment de l'analyseur. Si le code de certaines opérations sur les graphes pourra certainement être repris, tout du moins en partie, les structures de données associées semblent bien lourdes au regard de leur réelle utilité.
Il en découle que la structure de données retenue de *HashMap* en plus d'être complexe, s'avère aussi pénalisante pour certaines opérations comme le *pivotage*.
Malgré cette complexité inutile, la saisie des graphes comme nous l'avons vu ne satisfait pas le client du point de vue de l'interface, aussi M. Kante suggère une saisie de matrice d'adjacence.

2.1.4 Calcul du polynôme

Le calcul du polynôme s'effectue à l'aide des données représentées en mémoire, le graphe et la définition récursive du polynôme. Cette définition est constituée de clauses se décomposant en expressions et conditions.

Chaque expression se représente ensuite par un arbre dont les noeuds se différencient par leurs types :
- constante,
- variable liée,
- variable non liée,
- opérateur unaire,
- opérateur binaire,
- expression partielle (appel récursif),
- polynôme à plusieurs variables liées ou non.

Une expression s'applique à la chaîne de caractères représentant le graphe *graphe expression*. Ces représentations textuelles peuvent résulter d'opérations unaires, binaires ou ternaires sur des graphes.

L'application d'une clause initiale dont la condition est vérifiée donne une première expression. La représentation sous forme d'un arbre de celle-ci introduit de nouvelles expressions partielles s'appliquant à l'expression textuelle de graphes.

L'arbre résultant de l'évaluation de l'expression initiale, ainsi que des expressions partielles détermine le polynôme.

2.1.5 Illustration

L'illustration suivante indique comment l'exemple du précédent projet de programmation était traité par ce même projet.

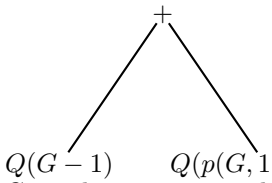
Matrice d'adjacence du graphe :

	1	2	3	4
1	0	1	1	1
2	1	0	1	0
3	1	1	0	0
4	1	0	0	0

(1)$Q(G) = 1$ si G est vide,
(2)$Q(G) = x * Q(G - a)$ si a est un sommet isolé,
(3)$Q(G) = Q(G - a) + Q(p(G, a, b) - b)$ si a et b sont adjacents.

La dernière clause s'applique aux sommets 1 et 2. Nous avons donc l'expression initiale suivante :
$Q(G) = Q(G - 1) + Q(p(G, 1, 2) - 2)$

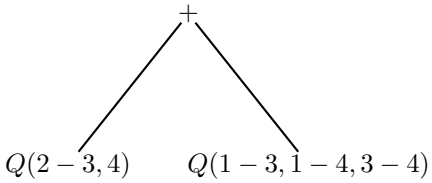
Il suit l'arbre ci-dessous après application de la clause (3) sur les sommets 1 et 2 :



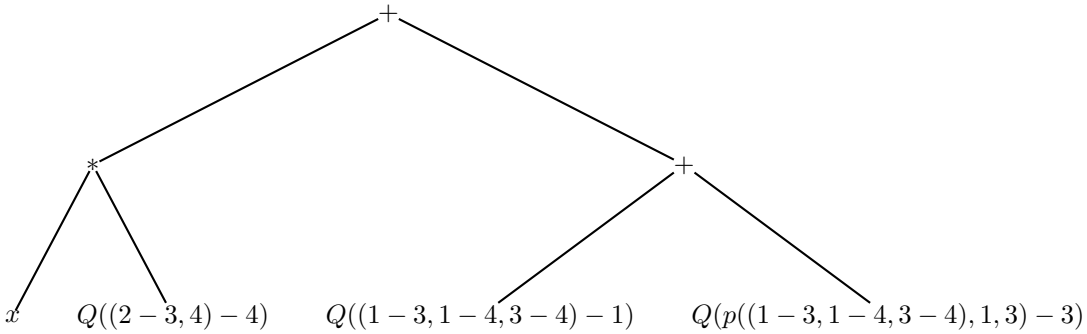
Cet arbre, en particulier les expressions partielles (ou appels récursifs) prend en paramètre un graphe. Les "graphes expressions" sont ici représentés par $G - 1$ et $p(G, 1, 2) - 2)$ qui valent respectivement :

- $2 - 3, 4$: graphe composé des sommets 2, 3 et 4 et d'une arête entre les sommets 2 et 3 ;
- $1 - 3, 1 - 4, 3 - 4$: graphe complet composé des sommets 1, 3 et 4.

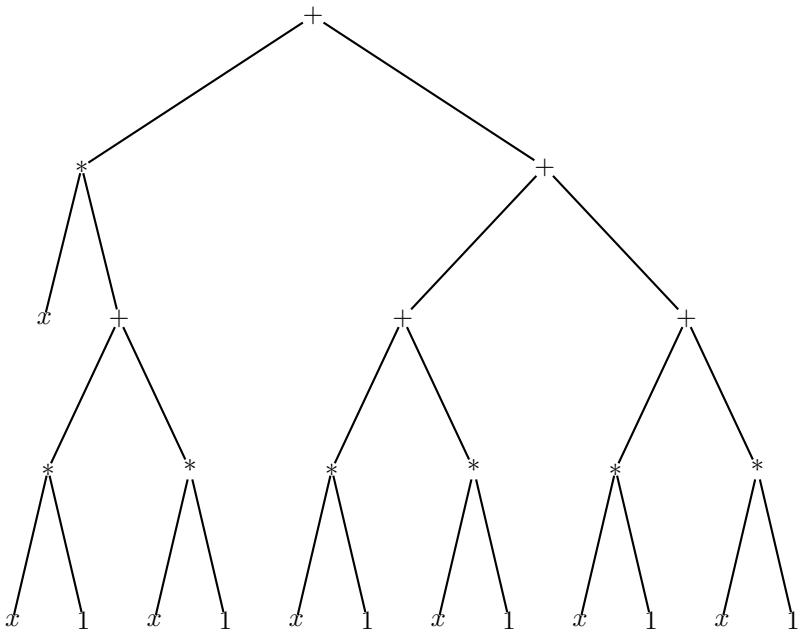
L'arbre précédent équivaut donc à :



Cela donne ensuite l'arbre suivant après application de la clause (2) sur le sommet 4, et de la clause (3) sur les sommets 1 et 3.



Après valuation de toutes les expressions partielles, on obtient l'arbre suivant :



Le parcours infixe de cet arbre en tenant compte de la priorité des opérateurs donne alors l'expression du polynôme associé au graphe. Il s'agit ici de :
$x * (x * (1) + x * (1)) + x * (1) + x * (1) + x * (1) = 2 * x^2 + 4 * x$,
c'est–dire le résultat calcul en exemple dans le précédent rapport.

2.1.6 Analyse

A travers l'explication du fonctionnement du précédent projet de programmation ainsi que de l'exemple, on constate une interaction permanente entre l'analyse d'une expression et la génération de l'arbre permettant de déterminer le polynôme. Le graphe lui évolue au sein de cet arbre sous une forme textuelle, dite *graphe expression*.

2.2 Les outils et codes disponibles sur les graphes

Il existe bon nombre d'outils et de librairies sur les graphes, dont la plupart sont consacrés à la représentation graphique. Parmi ceux-ci, deux ont particulièrement retenu notre attention, pour des raisons différentes, qu'il convient d'expliciter.

2.2.1 Graphviz

Graphviz est un ensemble d'outils en lignes de commandes, développé par ATT, et très utilisé pour leur capacité à représenter tous les types de graphes. Pour ce qui nous concerne, nous avons utilisé l'outil **neato** qui correspond aux types de graphes visés. Si la représentation graphique d'un graphe est en dehors du périmètre de notre projet, il nous permet simplement de matérialiser les graphes sur lesquels nous travaillons. Mais surtout, il définit un format de fichier .txt, très répandu et notamment pour lesquels il existe des librairies toutes faites, dans la plupart des langages de programmation.
Le langage **dot** est en particulier utilisé par la plupart des outils manipulant des graphes et notamment ceux développés au Labri (AltaRica, Tulip ...).

2.2.2 GraphThing

C'est un petit utilitaire éducatif disponible en particulier sur linux, si cet outil ne remplit pas les fonctionnalités requises, il s'avère être un prototype intéressant de saisie de graphes-types, en offrant notamment des

exemples d'interfaces adaptés à de nombreux types de graphes. A ce titre il constitue un prototype intéressant. Il est en plus doté de fonctionnalités puissantes comme le calcul du polynôme chromatique, du nombre chromatique, du rayon, du diamètre ...

2.2.3 Codes open-source utilisés

Il existe une multitude de produit et de librairies java traitant des graphes en licence GPL (Jgrapht, sjgraph ...). Les besoins du projet étant limités en ce qui concerne les graphes, et les opérations majeures concernant les graphes étant spécifique à notre application *(pivotage, complémentation locale...)* et intervenant au coeur du calcul récursif, il a été choisi de redévelopper ces fonctionnalités pour avoir une meilleure maîtrise de ces opérations.

2.2.4 Les analyseurs lexicaux et grammaticaux

Une étape clé du traitement consiste en l'interprétation des expressions saisies par l'utilisateur. L'utilisateur ayant toute liberté de combiner ces expressions, le *parsing* de telles commandes revient en fait à faire de l'analyse syntaxique. Nous nous sommes donc intéressés aux outils adaptés à ce type de travaux de compilation. Cette recherche nous a naturellement conduit au binôme **Lex/Yacc** pour lequel nous avons trouvé les équivalents en Java **JFlex/Cup**. Cette solution envisagée, permet d'anticiper le développement et la mise au point de la grammaire de manière autonome (Lex/Yacc), puis d'intégrer les codes dans le projet en java.
Le couplage des fonctionnalités de l'analyseur syntaxique *Cup* avec un langage orienté objet tel que Java, offre de plus la persective d'intégrer le calcul récursif sur des objets élaborés directement dans l'automate d'analyse syntaxique. Il existe d'autres produits équivallents tels que JavaCC, ou sableCC, une fois encore le besoin d'efficacité à primer et nous avons choisi JFlex/Cup pour leur similitude avec Lex/Yacc que nous avons eu l'occasion de pratiquer au premier semestre. Nous avons utilisé la version 1.4.1 de JFlex et la version v0.11 de Cup. Ces produits sont maintenant stables, ils sont disponibles sur : `http://www2.cs.tum.edu/projects/cup/`.

2.2.5 Les polynômes

Une autre partie essentielle du projet est consituée naturellement par le calcul d'expressions mathématiques. Ces calculs interviennent à la fois pendant le déroulement de la récursion, pour ce qui est du calcul de polynômes à partir d'une définition récursive et d'un graphe mais aussi lors des opérations d'analyse et de transformation du polynôme ainsi obtenu.

Le calcul symbolique sur les polynômes

Les spécificités majeures des calculs nécessaires sont :
- des calculs sur des entiers relatifs
- calculs de polynômes multivariés (comportant plusieurs variables)

Nous avons utilisé un outil dénommé **meditor**, assorti d'une bibliothèque en Java qui satisfaisait ces critères.

Présentation de JSCL-meditor

La bibliothèque JSCL meditor est issue de sourceforge `http://sourceforge.net/projects/jscl-meditor/`. Comme le décrivent ces auteurs, *meditor est un éditeur de texte avec des capacités de calcul symbolique. En premier lieu, il est concu pour résoudre des systèmes d'équations algébriques de degré quelconque à plusieurs inconnues en calculant les bases standards (ou de Groebner) d'idéaux de polynômes.*
En plus de ces fonctionnalités, meditor manipule également des vecteurs, des matrices , des expressions logiques et outre ces capacités de résolution de systèmes d'équations, il est capable de calculer les dérivées et intégrales, de simplifier des expressions mathématiques, de faire de la substitution de variables, de l'export en XML, Latex et pdf.

La bibliothèque utilisée est dans sa version 2.3_03, elle est sous licence GPL (fournie en annexe).
Les critères qui ont conduit au choix de cette bibliothèque sont :

- les fonctionalités de calcul formel et plus particulièrement le calcul sur des polynomes multivariés.
- la dynamique du projet, qui existant depuis 2004 a été remis à jour en 2008 (version utilisée), ce qui atteste d'une certaine continuité et d'un suivi des développements.
- La possibilité de simuler les calculs par le biais de l'interface fournie (logiciel **meditor**).
- le projet fourni comprend non seulement les codes, mais aussi une documentation sous la forme d'une API, qui permet au moins de naviguer dans l'arborescence des sources, et bien entendu les JUnits utilisés.
- potentiel intéressant d'évolution pour notre application

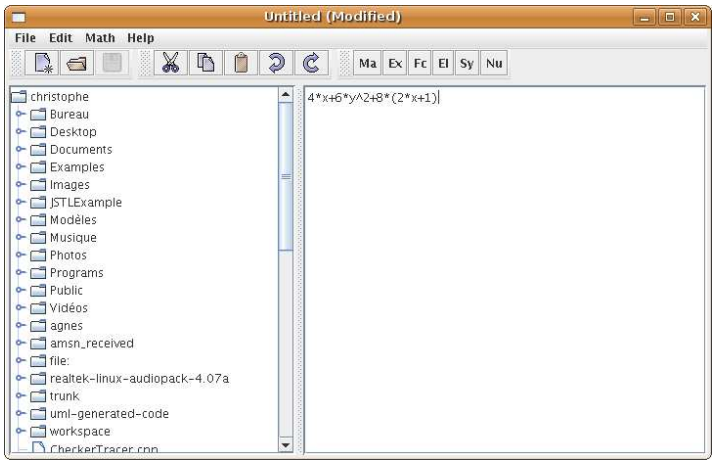


FIG. 2.2 – Interface de meditor

Nous sommes conscients que cette bibliothèque va au-delà des besoins strictement nécessaires à notre projet. Nonobstant, l'intégration de ces codes dans une librairie séparée rend le logiciel potentiellement plus riche sans pour autant alourdir le code inutilement.

2.2.6 Divers

Calcul matriciel

Une des fonctionnalités demandées consistait à calculer le rang de la matrice d'adjacence. Ce calcul ne figurant pas parmi les fonctionnalités de la bibliothèque, et n'étant pas non plus une priorité, il a été décidé d'importer une bibliothèque supplémentaire qui réaliserait ce traitement. La bibliothèque choisie JAMA (Java Matrix), remplit cette fonction ainsi que quelques autres calculs élémentaires d'algèbre linéaire (inversion, décomposition LU ...). Cette bibliothèque utilisée dans sa version 1.0.2 est accessible à l'adresse
`http://math.nist.gov/javanumerics/jama/` est libre de droits, conformément aux indications sur le site. *Copyright Notice This software is a cooperative product of The MathWorks and the National Institute of Standards and Technology (NIST) which has been released to the public domain. Neither The MathWorks nor NIST assumes any responsibility whatsoever for its use by other parties, and makes no guarantees, expressed or implied, about its quality, reliability, or any other characteristic.*

Interfaçage avec le système d'exploitation

L'appel dans l'applet à la fonction **aide**, développée en HTML pour une meilleure convivialité nécessite une interface avec le système d'exploitation de la machine sur laquelle elle est executée. Cette interface est réalisée par la bibliothèque *jdic* (JDesktop Integrated Components) qui permet aux applications java d'interagir avec l'environnement de travail d'accueil.

Chapitre 3

Besoins-non fonctionnels

3.1 Fiabilité

Le niveau de fiabilité requis par le client est élevé, dans des conditions de fonctionnement "normales" à savoir :

- Entrée correcte (syntaxiquement et fonctionnellement)
- volume de données non-critique (moins de 20 sommets)

Qualification La traduction de cette exigence est 100% de résultats cohérents avec les jeux d'essais fournis (entrées et résultats attendus).

Contraintes Les tests qui sont mis en place sont ceux fournis par le client. Ces jeux d'essais ainsi que les résultats attendus figurent en annexe.

Risques induits En cas d'erreurs avérées et non résolues les limites de fonctionnement devront être établies.

3.2 Convivialité

Il s'agit de la priorité numéro 2 évaluée par le client, après la fiabilité des calculs.

Qualification La convivialité concerne à la fois la syntaxe des langages qui seront définis et la facilité d'utilisation des interfaces proposées. Ce besoin justifie aussi le choix d'une *applet* exécutée via un navigateur Web et ne nécessitant donc pas de procédure d'installation spécifique.

Contraintes Les syntaxes seront définies et validées par le client. Les interfaces seront développées après prototypage préalable.

Risques induits Cette exigence est au moins aussi lourde de conséquences que la précédente. Le développement d'interface, très chronophage représentera donc une part importante du temps de développement.

3.3 Extensibilité-maintenabilité

Qualification Le besoin d'évolutivité et d'extensibilité est apparu lors des discussions avec M. Kanté. Il convient donc d'expliciter et de prévoir les orientations possibles de telles évolutions, afin de prévoir dès la conception leur intégration. Les secteurs d'évolutions envisagés cités ci-dessous sont qualifiés en terme de priorité relative, les uns par rapport aux autres et dans la mesure du possible dans l'échelle de priorités générales. Les sources d'extensibilité recensées sont

Enrichissement du langage des *clauses* Evolutivité prioritaire

Ajout de fonctionnalités d'analyse et de manipulation des polynômes résultats Evolutivité prioritaire.

Proposition de nouveaux *graphes types* Ce besoin d'évolutivité n'est pas essentiel, car relève d'un confort supplémentaire (il est possible de s'en passer).

Extension à de nouveaux types de graphes (orientés-multiples ...) Ce besoin d'évolutivité n'est pas essentiel pour le client, il ne sera pas pris en compte lors de la conception.

Contraintes Outre le surplus de conception, les diverses possibilités d'extension seront classifiées, un chapitre de la documentation finale sera dédié à cet aspect du logiciel.

Risques induits Temps

3.4 Robustesse

Qualification Le besoin de robustesse est faible. Il n'est pas demandé au logiciel d'être tolérant aux pannes, que ce soient les erreurs induites par des mauvaises saisies de l'opérateur ou par des limitations de la machine dans des cas extrêmes de fonctionnement (nombre important de sommets). Le seuil de fonctionnement normal est fixé à 20 sommets. Toute amélioration de la robustesse est un plus au cahier des charges.

3.5 Performances

Aucune exigence de performance n'est requise pour les calculs.

Chapitre 4

Besoins fonctionnels

4.1 Les graphes

4.1.1 Définition d'un graphe

L'utilisateur doit pouvoir **définir des graphes** :
- non orientés,
- sans arêtes multiples,
- éventuellement avec boucles.

La définition doit être possible :
- soit par le biais d'une **"saisie interactive"**,
- soit via le **chargement d'un fichier texte** résultant d'une utilisation précédente.

4.1.2 Saisie interactive d'un graphe

Au moyen du clavier et/ou de la souris, l'utilisateur doit pouvoir saisir son graphe tout en ayant à l'écran un contrôle sur ses saisies. L'existence de graphes particuliers doit être prise en compte durant cette phase. Il est par exemple relativement pertinent de ne devoir entrer que le nombre de sommets pour un graphe complet. Liste des graphes types à prendre en compte est :
- graphe complet,
- graphe étoile
- graphe bipartis
- cycle
- chemin

Le graphe dont la matrice d'adjacence est la suivante pourrait donc se définir dans une boîte de dialogue textuelle comme suit :

Matrice d'adjacence :

	1	2	3	4	5	6
1	**0**	1	1	0	1	0
2	1	**0**	0	0	0	0
3	1	0	**1**	0	1	1
4	0	0	0	**0**	0	0
5	1	0	1	0	**0**	0
6	0	0	1	0	0	**0**

Texte saisi par l'utilisateur (de façon optimale) :

```
#6 1:2 3 5 3:3 5 6
```

Le graphe résultant de la suppression dans le précédent des sommets 1 et 5 lui, se saisirait ainsi :

Matrice d'adjacence :

	2	3	4	6
2	**0**	0	0	0
3	0	**1**	0	1
4	0	0	**0**	0
6	0	1	0	**0**

Texte saisi par l'utilisateur :

```
2 3 4 6 3:3 6
```

Il serait faux ici de remplacer la première ligne saisie par `#4`.

4.1.3 Sauvegarde et chargement d'un graphe

Il s'agit ici de pouvoir sauvegarder le graphe dans un fichier texte. Il doit aussi être possible d'effectuer l'opération inverse, c'est-à-dire lire un fichier texte définissant un graphe et traduire son contenu en une représentation mémoire du graphe défini. Sauvegarder le graphe dans un format déjà utilisé par les utilisateurs auxquels le programme est destiné serait avantageux.

4.1.4 Opérations sur les graphes

Les opérations sont celles explicitées au Chapitre 1. Pour mémoire la liste est :
- Sélection par voisinage.
- Suppression d'un sommet.
- Suppression d'un ensemble de sommets.
- Complément des boucles des sommets.
- Complément des sommets sans toucher aux boucles.
- Complément local.
- Complément local étendu aux boucles.
- Pivotage.

4.1.5 Fonctions élémentaires sur les graphes supplémentaires

En plus des opérations citées ci-dessus, une liste de fonctions supplémentaires a été envisagée. Elles permettront d'augmenter l'expressivité du langage de *définition des polynômes récursifs*. Il s'agit d'une liste de calculs classiques portant sur les graphes :
- Nombre de composantes connexes du graphe
- Rang de la matrice d'adjacence associée à un graphe
- Nombres de sommets
- Nombres de sommets avec boucle

4.2 Définitions récursives

Comme nous l'avons vu précédemment, la définition récursive d'un polynôme, correspond à une suite finie de clauses, dont l'application à un graphe G donné permet de générer un polynôme associé à G.

4.2.1 Saisie des clauses

La définition doit être possible :
- soit par le biais d'une **"saisie interactive"**,
- soit via le **chargement d'un fichier texte** résultant d'une utilisation précédente.

4.2.2 Enregistrement des clauses

Les syntaxes de saisie et d'enregistrement des clauses doivent être cohérentes, et *humainement compréhensibles* conformément aux spécifications données dans le sujet.

4.2.3 Application des clauses

A priori, le polynôme résultant d'un calcul récursif appliqué à un graphe donné est dépendant de l'ordre dans lequel les clauses sont appliquées. Par conséquent, un ordonnancement de l'application des clauses est imposé, il est défini d'abord au moyen d'un ordre total sur les sommets puis dans l'ordre d'écriture des clauses. On note *i-clause*, une clause portant sur i sommets (union du nombre de sommets distincts mentionnés dans la condition et dans l'expression). En résumé :

- L'ordre d'application des *i-clauses* est dans un premier temps donné par i
- Les clauses portant sur un même nombre de sommets, sont appliquées dans l'ordre lexicographique des sommets la constituant.
- Les i-clauses portant strictement sur les mêmes sommets sont appliquées dans l'ordre d'énoncé des clauses.

4.3 Exploitation des polynômes

Les polynômes obtenus à partir d'un graphe et d'une définition récursive, permettent comme nous l'avons vu de déterminer certaines propriétés du graphe. L'étude de ces polynômes résultant, au travers d'opérations diverses permet d'étudier et de mettre en évidence, ces propriétés. Ces opérations sont décrites ci-dessous.

4.3.1 Substitution d'indéterminées

Comme nous l'avons vu, un polynôme est constitué d'une somme de monômes, formés chacun par un produit d'indéterminées.
Dans le cas *d'indéterminées ordinaires*, la substitution d'indéterminées consiste à remplacer une de ces indéterminées par un polynôme formé de tout ou partie des indéterminées ordinaires.
Soit $P(G) = 5x^2 + 6x$, le polynôme calculé dans notre exemple [1] :
Une opération de substitution peut être de substituer le polynôme $(x + 1)$ à x. Le résultat de la substitution serait alors : $Q'(G) = 5(x + 1)^2 + 6(x + 1)$ soit $Q'(G) = 5x^2 + 16x + 11$.
Dans le cas *d'indéterminée associé à un sommet*, la règle de substitution est identique, si ce n'est que le polynôme se substituant à une indéterminée doit être une expression des indéterminées du sommet en question. par exemple à x_1 ne peut être substitué qu'un polynôme d'indéterminées du sommet 1 :$(x_1y_1^2 + x_1 + 1)$. Il s'agit dans ce cas, en réalité de générer un polynôme modifié localement (pour un sommet donné).

4.3.2 Opérations sur les polynômes

Il s'agit d'opérations arithmétiques (+,-,*,^) sur des polynômes qui auront été précédemment calculés.

[1] cf 1.3

Chapitre 5

Réalisation et fournitures

Le développement a été réalisé en Java. Ce choix a été motivé par les contraintes d'utilisation, exprimées par M Kanté, à savoir :
- le fonctionnement en environnement multi-plateformes,
- le choix restrictif de l'applet et l'accessiblité via Internet,
- la nécessité d'une interface graphique assez évoluée.

Le ramasse-miettes Java nous a facilité la gestion de mémoire et la réutilisation de bibliothéques.

5.1 Description des fournitures

L'application est fournie sous la forme d'un fichier compressé **jar**. Ce fichier contient l'arborescence de l'ensembles des codes sources, de la documentation et des bibliothèques et executables nécessaires à sa génération.

5.1.1 Codes sources

Le répertoire **src** contient paquetages du projet, ainsi que les sources des JUnits (paquetages testGraphe, testPoly et testPolyRec). Les fichiers .lex et .cup des analyseurs syntaxiques sont dans le paquetage analyseur.

5.1.2 Documentation

Le répertoire *doc* contient la javadoc. La notice utilisateur, est dans un répertoire **notice** dans le répertoire **bin**.

5.1.3 Bibliothèques

Le répertoire *lib* contient les *jar* nécessaires à l'execution de l'application ainsi que les jar permettant d'installer JFlex et cup.
- Jama-1.0.2
- java-cup-11a
- jdic
- JFlex-1.4.1
- junit
- meditor

Il contient également les répertoires nécessaires au fonctionnement de **jdic** sur les plateformes linux, mac, sunos et windows.

5.1.4 Jeux d'essais

Le répertoire *JeuxEssais* contient les fichiers de tests utilisés pour les JUnits. Il sont répartis en trois répertoires , *graphes*, *définitions* et *résultats*.

Le répertoire *graphes* contient des graphes types et des graphes saisis aléatoirement. Leur tailles varient entre 0 et 20, avec des différents ordre de parcours sur les sommets Le répertorie *définitions* comprends toutes les définitions fournies par les clients Le répertorie *résultats* contient les polynômes - résultats des calcul de certains définitions récursives sur certains graphes.

5.2 Synthèse des fonctionnalités remplies

5.2.1 Saisie des graphes

Les fonctionnalités réalisées sont conformes aux besoins fonctionnels (chapitre 4) exprimés :
- Saisie interactive des graphes représentés par leur matrice d'adjacence
- Chargement à partir de fichiers texte.
- Possibilité de saisie automatique de graphes types.
- Enregistrement dans un fichier texte.
- Gestion d'une liste de graphes.
- Saisie de l'ordre de parcours des sommets du graphe (ordre de recherche des sommets qui satisfont une condition).
- Identification d'un graphe.
- Inversion des valeurs dans la matrice d'adjacence.

5.2.2 Saisie des définitions

- Saisie de la table de clauses (la syntaxe proposée est très proche de la syntaxe présentée dans le sujet)
- Chargement à partir d'un fichier texte.
- Enregistrement dans un fichier texte.
- Identification d'une définition.
- Gestion d'une définition.

5.2.3 Calculs

Calcul Récursif

- Calcul d'un polynôme à partir d'un graphe et d'une définition récursive en prenant en compte l'ordre des sommets définis
- Toutes les opérations élémentaires prioritaires ont été implémentées ainsi que les opérations complémentaires : le rang du graphe et le nombre de composantes connexes du graphe.
- Enregistrement du polynôme, du graphe et de la définition utilisée, dans un fichier texte
- Identification du polynôme résultat
- Chargement à partir d'un fichier

Analyse des résultats

- Evaluation d'un polynôme en substituant des indéterminées.
- Filtrage d'un polynôme (affichage de certains monômes).
- Opérations arithmétiques sur les polynômes calculés.
- Affichage des caractéristiques d'un polynôme.

5.2.4 Interface graphique

L'interface graphique a fait l'objet d'un prototypage et d'une validation par M Kanté.

5.3 Maintenance

Le besoin exprimé de maintenabilité du code nous a conduit à produire d'une part la documentation des classes générées. Cette documentation a été produite automatiquement par **Eclipse**, au format *javadoc*. Nous

avons dans la mesure du possible anoté le code de manière à ce que les commentaires essentielles transparaissent dans la *javadoc* (ces opérations ont été systématisées tout du moins pour les paquetages **polynome** et **graphe**. De plus les tests unitaires sont fournis et permettent ainsi de disposer d'éléments de référence pour des évolutions ou des corrections futures.

5.4 Evolutions

5.4.1 Calcul itératif

La méthode de calcul itérative implémentée s'avère fonctionnelle. Néanmoins, cela se restreint à un cadre précis étant donné qu'il ne s'agit pas là de la méthode de calcul retenue et développée.

Ainsi, en tout premier lieu, il n'a pas été fait de liaisons avec l'analyse syntaxique. Les définitions récursives (ensemble de clauses composées d'expressions et de conditions) ne peuvent donc être reconnues et chargées à partir de fichiers. Seules sont utilisables celles implémentées ("en dur").
Ensuite, l'ensemble des conditions élémentaires composant la condition d'une clause doivent être vérifiées. En effet, on ne tient pas compte ici du connecteur logique `ou` ni d'éventuelles priorités.
Enfin, des fonctionnalités telles que l'ordre de parcours des sommets ne sont pas implémentées. La liaison avec une interface graphique demeure elle aussi à faire.

La méthode de calcul itérative décrite présente un intérêt certain notamment du point de vue algorithmique. En effet, elle illustre une méthode permettant de dérécursiver un algorithme de calcul typiquement récursif. Ainsi, elle permet de s'abstraire d'éventuels problèmes liés à la gestion d'une pile d'appels (*StackOverFlow*).

Comparativement à la méthode récursive adoptée, elle présente surtout un intérêt en terme de performance. En effet, l'analyse syntaxique des expressions (toujours les mêmes) n'est pas réïtérée pour chaque appel dans le seul but d'exécuter l'action associée, mais est seulement effectuée une fois, lors de l'initialisation qui créée des instances d'objets *Expression* directement calculables.

Ainsi, pour une série donnée de calculs portant sur les mêmes graphes et définitions, on a constaté une division par un facteur 6 du temps de calcul par rapport à la méthode adoptée. Rappelons néanmoins que la performance ne fait pas partie des besoins non-fonctionnels, et que si tel avait été le cas, nous aurions probablement porté plus d'attention à la méthode itérative.

5.4.2 Evolution des grammaires

Restant toujours dans l'optique d'une possible extension des fonctionnalités de notre logiciel, nous avons essayé de rendre le plus simple possible l'ajout des nouvelles règles dans la grammaire des fichiers .cup ainsi que le changement du lexique du langage utilisé dans les définitions récursives. Ceci étant, il faut avoir un minimum de notions de compilation, pour introduire correctement les changements. JCup et JLex étant très proches de Yacc et Lex, une connaissance des principes des ces derniers est souhaitable. Voici les demarches à suivre pour introduire de nouvelles règles de grammaire :

Dans le fichier .cup :

1. Ajouter A : :=B C : / $*$ *actionjava* $*$ / : pour une règle de type $A- > BC$. L'action java représente les instructions qui seront exécutées au moment de la reconnaissance de cette règle.
2. Ajouter dans le champ "terminal" (respectivement "non-terminal"), au debut du fichier, les nouveaux terminaux (respectivement non-terminaux) introduits.

Pour definir un attribut associé à un terminal ou à un non-terminal du membre droit d'une règle, il faut écrire la règle sous la forme $A ::= B$: **nom_attribut**C (on associe l'attribut à B). L'attribut associé au membre gauche de la règle est défini par défaut par le mot clé **RESULT**. Lorsqu'on utilise les attributs associés aux éléments du vocabulaire de la grammaire, il faut définir un type *java* pour ces éléments (Object, String, Integer etc.).

Etant donné que le fichier .cup et accompagné par un fichier .lex, souvent en modifiant la grammaire on modifie aussi le lexique. Notamment pour introduire de nouveaux terminaux dans le fichier .cup. Il faut les définir dans le fichier .lex. Voici la ligne à ajouter dans le fichier **LexemesExpression.lex** pour introduire le terminal *DIV* representant le symbol :

```
"/" {return new Symbol (symExpression.DIV);}
```

Une fois les modifications effectuées, il faudra regénérer à nouveaux l'analyseur lexical et l'analyseur syntaxique puis recompiler tout le projet en utilisant le makefile fourni. Sa description sera donnée dans la section suivante.

5.4.3 Evolution des calculs sur les polynômes

C'est la classe **polynomImpl**, qui contient la fabrique d'opérateurs sur les polynômes. Ces opérateurs sont des méthodes de classes renvoyant un **Polynome**. Ces calculs sont basés sur la bibliothèque JSCL. La démarche à suivre pour implémenter un nouveau calcul :

- Si le calcul est disponible dans la JSCL (`http://sourceforge.net/projects/jscl-meditor/`) procéder de la même manière que les opérateurs ADD, MOINS de la fabrique de **polynomImpl**
- Sinon le calcul doit être codé dans une autre classe, afin de ne pas dépendre de la bibliothèque JSCL, en prenant garde de bien renvoyer un **nouvel** objet qui soit le plus générique possible, fournissant les services de **Polynome**.

Evolution du paquetage graphe

Le paquetage `Graphe` a été créé de manière à être aisément extensible.

En premier lieu, nous avons tout d'abord les différentes fabriques. Elles permettent d'ajouter de nouvelle opérations ou conditions portant sur les graphes. L'unique contrainte pour qu'elles soient exploitables est de les ajouter à la grammaire.

Concernant l'implémentation des graphes, en lieu et place de booléens, nous avons choisi d'utiliser des entiers comme éléments de la matrice d'adjacences. Ainsi, le paquetage peut être étendu afin de traiter les graphes orientés (et éventuellement avec des arcs pondérés).

5.4.4 Evolution de l'interface

Une interfaca graphique n'est jamais totalement terminée et peut toujours évoluer. Voici une liste d'évolution possibles au sein de notre interface.

Onglet Graphe

- Les graphes types disponibles sont les graphes les plus utilisés. Cependant il est possible d'en rajouter. Pour cela il faut modifier la classe **SaisieGraphe** et ajouter dans *private String graphes[]* le nom du graphe à rajouter. Ensuite pour que la matrice correspondante soit construite, il faut ajouter dans **Modele.java** et plus précisemment dans *switch (type)* (type est l'indice de la JComboBox) la méthode de construction,
- Permettre une numérotation des sommets autres que 1,2,3,4,... comme 10,20,30,40...,

Onglet Définition

Afficher une liste de conditions prédéfinies et pouvoir les choisir au lieu de les saisir manuellement.

Onglet résultats

- Ajouter des boutons pour choisir d'ajouter ou du supprimer les polynômes résultats dans la liste. Ceci se fait dans la classe **Resultat**,

– Le chargement multiple de fichiers résultats pour ensuite effectuer des opérations dessus. Pour cela il faut modifier lors du chargement un ajout multiple dans la HashMap et dans la liste.

Une sélection multiple pour supprimer plus rapidement les graphes, les définitions récursives ou les résultats plus rapidement.

5.5 Recompilation du projet

Makefile et ANT

Nous avons tenu ici à décrire l'une des procédures permettant de recompiler le projet. Nous avons opté pour celle se basant sur Makefile et ANT. En effet, la méthode consistant à recompiler le projet à l'aide d'un environnement de développement tel que Eclipse, se révèle peu appropriée pour un utilisateur. A l'inverse, une recompilation à l'aide de Makefile et ANT se fait par une simple ligne de commande.

Principe ANT s'appuie sur des fichiers contenant les directives nécessaires à une compilation (`build.xml`). Il s'agit donc de créer ce fichier en fonction des nécessités. Dans notre cas, le fichier build.xml a été généré automatiquement à l'aide de Eclipse (Export puis Ant Buildfiles).

Nous devons donc en tout premier lieu regénérer l'analyseur lexical à partir des fichiers .lex, puis le parseur à partir de l'analyseur lexicale. Cela se fait à l'aide d'un `Makefile`. Nous pouvons alors recompiler le projet à l'aide de ANT en tenant compte de nouvelles règles de grammaire par exemple.

Compilation Afin de recompiler le projet, il s'agit dans un premier temps d'exécuter le Makefile dans le répertoire `src/analyseur/`. Pour cela nous exécutons dans ce même répertoire la commmande `make`.
Dans un second temps, il faut exécuter ANT en lui passant en argument la cible de construction indiquée dans le fichier de compilation. Typiquement, sous un environnement Linux, cela se fait par la commande suivante :
`ant build` (à la racine du projet contenant le fichier `build.xml`).

Ces deux étapes ont été regroupées dans un script shell afin de faciliter la recompilation. Après modification, il s'agit donc pour l'utilisateur de taper à la racine du projet la commande suivante :
`sh recompiler.sh`

De là l'utilisateur peut réutiliser le projet avec ses modifications personnelles.

Chapitre 6

Architecture

6.1 Architecture globales

L'architecture retenue et les choix réalisés l'ont été afin de privilégier la justesse des calculs mais aussi afin de pouvoir consacrer le temps nécessaire à l'Interface Homme Machine qui est essentielle pour MM. Kanté et Courcelle. L'option de la réutilisation a donc été retenue pour les calculs sur les polynômes, ainsi que pour les fonctions d'analyse syntaxique requises. L'objectif de ces choix était de disposer le plus rapidement possible des briques de bases nécessaires à l'avancement du projet. La deuxième option majeure a concerné la structure du calcul récursif. Lors de la phase de réflexion sur le projet, il a semblé naturel de calculer récursivement le polynôme du fait de la nature récursive de sa définition.
Sur les conseils de M. Bonichon notre chargé de TD, nous avons néanmoins fait l'étude d'un calcul itératif, conscient que le traitement récursif était plus exigeant en termes de ressources, comme nous le verrons, alors que le calcul itératif était pour sa part plus risqué. En effet, la gestion du contexte, pris en charge par le système lors d'un calcul récursif, doit alors être réalisée manuellement. Les travaux concernant cette méthode ont aboutis, à une maquette, qui était fonctionnelle pour une définition codée en dur. Le fonctionnement de cette méthode est présenté en annexe.

L'architecture, assez proche de celle qui avait été envisagée lors du rapport intermédiaire, est centrée sur quatre paquetages pour les calculs auquel vient s'ajouter un paquetage indépendant pour l'IHM.

- **analyseur** : analyses syntaxiques et calcul de polynôme associé à un graphe,
- **definition** : représentation d'une définition récursive,
- **graphe** : représentation et définition d'un graphe,
- **polynome** : calcul symbolique sur les polynômes,
- **graphique** : Interface Homme Machine.

Les paquetages **polynome** et **graphe** sont fonctionnellement décrits par des interfaces, qui assurent une complète indépendance des modules les uns par rapport aux autres.

6.2 Organisation des processus-élémentaires

Les besoins exprimés par les clients ont conduit à un développement *multi-threads*. Ceci afin de permettre l'interruption du traitement en cours et une bonne gestion de l'interface graphique.

6.2.1 Thread principal

Il contient l'applet mère et le gestionnaire d'événement qui lui est associé. C'est le père des autres processus élémentaires. Il est lancé lors de l'initialisation de l'applet.

6.2.2 Thread calcul

Ce thread réalise le calcul du polynôme à partir d'une définition récursive. Il peut être interrompu à tout moment par l'utilisateur. Il est instancié par chaque objet de type **Calcul**.

6.2.3 Thread Affichage

Ce thread réalise l'affichage du polynôme résultat à l'issu du calcul récursif. Il est synchronisé avec la fin du thread du calcul à l'aide de la méthode *join()*. Ce thread était nécessaire pour que l'utilisateur puisse agir au niveau de l'interface pendant le déroulement du calcul notamment pour l'interrompre. Sans ce thread, l'interface reste figée car il faut attendre la fin du calcul pour afficher le résultat.

6.2.4 Thread chrono

Ce thread est lancé en parallèle au calcul. Il sert à estimer le temps écoulé du traitement et à l'afficher. Il permet donc à l'utilisateur de savoir le temps écoulé depuis le lancement du calcul et s'il est nécessaire de continuer le calcul ou s'il faut mieux l'interrompre.

C'est threads sont tous dépendants les uns des autres mais la synchronisation principale est entre l'affichage et le calcul. La méthode join() permet l'attente de la fin du thread de calcul pour afficher le résultat.

6.3 Structure des modules

Avant de présenter le fonctionnement de l'application, une description des paquetages est nécessaire.

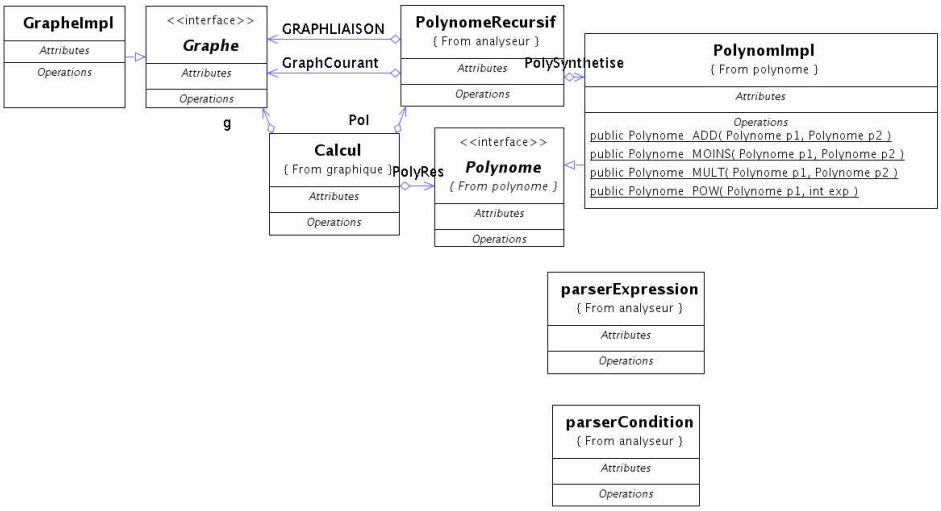


Fig. 6.1 – Architecture globale

6.3.1 Paquetage analyseur

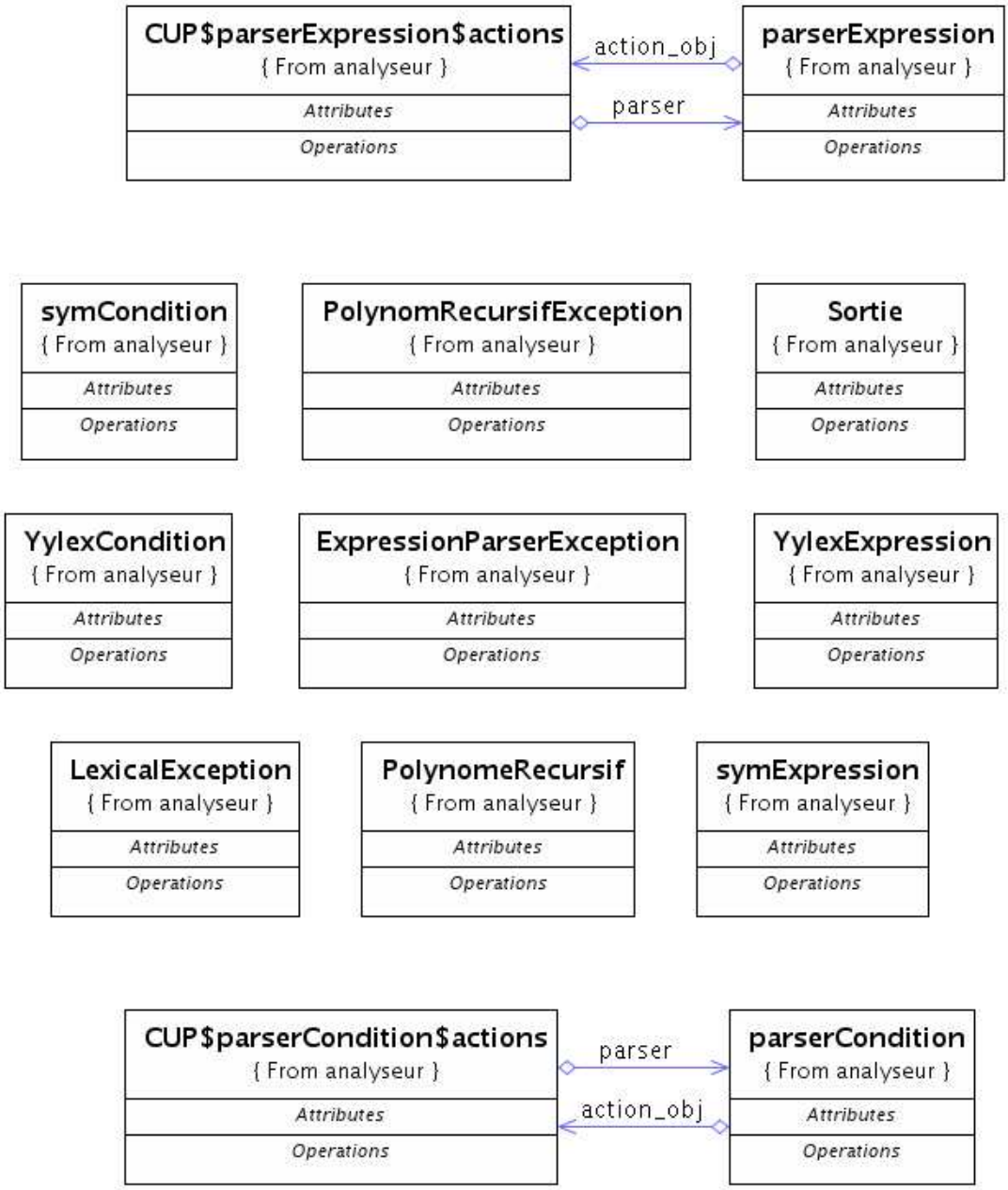


FIG. 6.2 – Paquetage analyseur

Ce paquetage contient les classes et les fichiers nécessaires pour l'analyse syntaxique des expressions et des conditions saisies par l'utilisateur ainsi que les classes qui correspondent à leur évaluation. Donc c'est à la fois le paquetage d'analyse et de calcul.
Les fichiers .cup permettent de définir une grammaire formelle pour pouvoir construire un parseur LALR à partir de celle-ci avec le logiciel Cup (équivalent du Yacc pour Java). Il est important de noter qu'on peut définir une action par défaut à effectuer avant de commencer le parsing de la chaîne de caractères. Dans notre cas l'action

par défaut est l'initialisation du graphe. En effet, à chaque appel récursif un nouveau graphe est construit à partir du graphe précédent, par suppression de sommets. Notre principe est de parser la clause sélectionnée (la condition et l'expression associée) à chaque appel récursif. C'est pour cette raison que nous avons choisi de construire un nouveau graphe, plutôt que d'utiliser une référence.
Comme pour Yacc un fichier .cup doit être accompagné par un fichier .lex (l'équivalent en Java est jFlex) dans lequel on définit le lexique de la grammaire. Une fois le *parseur* généré, ces fichiers n'interviennent plus dans l'application. Ils sont destinés à être utilisés au moment du changement éventuel de la grammaire et des règles lexicales lors de l'extension des fonctionnalités du logiciel. Les définitions du lexique, de la syntaxe ainsi que d'une grammaire en Cup seront expliquées dans le manuel d'utilisation.

Fichier Grammaire_Condition.cup

Ce fichier contient une grammaire formelle pour reconnaître et évaluer une condition ou une combinaison booléenne de conditions élémentaires sur un graphe. La grammaire est construite de manière à respecter la priorité de l'opérateur "et" par rapport à l'opérateur "ou" ainsi que les parenthèsages. A chaque étape de réduction d'une règle, l'attribut synthétisé du non-terminal du membre gauche récupère la valeur booléenne calculé à partir des attributs synthétisés des non-terminaux et des terminaux du membre droit. Ainsi, à la racine de l'arbre syntaxique, se trouve la valeur booléenne de la condition de la clause. Voici la grammaire décrite formellement où les terminaux sont le symboles **" ;"**, les **parenthèses** et les mots écrits uniquement avec des **majuscules** :
S ⟶ Expression **;**
Expression ⟶ Expression **OU** TermeCondition
Expression ⟶ TermeCondition
Expression ⟶ **GPHE VIDE**
TermeCondition ⟶ TermeCondition **ET** FacteurCondition
TermeCondition ⟶ FacteurCondition
FacteurCondition ⟶ Condition
FacteurCondition ⟶ (Expression)
Condition ⟶ **NON** (Condition)
Condition ⟶ **SOMMET INCD BOUCLE**
Condition ⟶ **SOMMET ET SOMMET ADJACENTS**

Fichier Grammaire_Expression.cup

Ce fichier contient la grammaire permettant de reconnaître et d'évaluer l'expression d'une clause. Comme pour la grammaire de la condition, à chaque étape de réduction, l'attribut synthétisé du membre gauche est calculé à partir des attributs synthétisés des membre droits. A la racine de l'arbre syntaxique se trouve le polynôme correspondant à l'expression. Voici la grammaire décrite formellement où les terminaux sont le symboles **" ;"**, les **parenthèses**, les **crochets**, les **accolades**, les opérations mathématiques (*****,+,**-**, ^) et les mots écrits uniquement avec des **majuscules** :
S ⟶ Expression **;**
Expression ⟶ Polynome + Expression
Expression ⟶ Polynome - Expression
Expression ⟶ Polynome
Polynome ⟶ Monome * Polynome
Polynome ⟶ Monome
Polynome ⟶ PolyRec * Polynome
Polynome ⟶ PolyRec
Polynome ⟶ (SommeMonomes) * Polynome
Polynome ⟶ (SommeMonomes) ^ Nombre * Polynome
Polynome ⟶ (SommeMonomes) ^ Nombre
Polynome ⟶ (SommeMonomes)
Polynome ⟶ (PolyRec) ^ Nombre * Polynome
Polynome ⟶ (PolyRec) ^ Nombre
PolyRec ⟶ P (Graphe)

SommeMonomes ⟶ SommeMonomes + TermeSommeMonomes
SommeMonomes ⟶ SommeMonomes - TermeSommeMonomes
SommeMonomes ⟶ TermeSommeMonomes
TermeSommeMonomes ⟶ TermeSommeMonomes * FacteurSommeMonomes
TermeSommeMonomes ⟶ FacteurSommeMonomes
FacteurSommeMonomes ⟶ Monome
FacteurSommeMonomes ⟶ (SommeMonomes) ^ Nombre
FacteurSommeMonomes ⟶ (SommeMonomes)
Monome ⟶ Nombre
Monome ⟶ **VARIABLE** _ **CHIFFRE** ^ Nombre
Monome ⟶ **VARIABLE** _ CHIFFRE
Monome ⟶ **VARIABLE** Nombre
Monome ⟶ **VARIABLE**
Nombre ⟶ **RK** (Graphe)
Nombre ⟶ **CC** (Graphe)
Graphe ⟶ Graphe - CHIFFRE
Graphe ⟶ **PIV** (Graphe , **CHIFFRE** , **CHIFFRE**)
Graphe ⟶ INV (Graphe , Ensemble)
Graphe ⟶ Graphe - Ensemble
Graphe ⟶ **C** (Graphe)
Graphe ⟶ **LC** (Graphe , CHIFFRE)
Graphe ⟶ **B C** (Graphe , **CHIFFRE**)
Graphe ⟶ Graphe [Ensemble]
Graphe ⟶ **G**
Ensemble ⟶ { Ens }
Ensemble ⟶ **N** (Graphe , **CHIFFRE**)
Ens ⟶ Ens , **CHIFFRE**
Ens ⟶ **CHIFFRE**

Fichiers LexemesCondition.lex et LexemesExpression.lex

Dans ces fichiers on définit le lexique des conditions, respectivement des expressions. A partir de ces fichiers nous avons généré les deux automates d'analyse lexicale *YylexCondition* et *YylexExpression* avec **JFlex**, dont les tokens lexicaux sont récupérés dans le fichiers .cup.

Les classes parserCondition, parserExpression, symCondition et symExpression

Ce sont les classes générées par Cup à partir des fichiers des grammaires. Les deux premiers représentent les parseurs pour la condition et pour l'expression. Les classes *sym* représentent les tables de symboles générées en exploitant l'automate de l'analyse lexicale. Nous les avons utilisées tels qu'elles ont été générées.

Classe PolynomeRecursif

C'est la classe centrale du paquetage. C'est elle qui interagit avec les autres paquetages et c'est elle qui prend en charge le calcul. D'abord nous présentons le déroulement d'un calcul : On cherche l'ensemble de sommets de taille minimale qui satisfont la condition de plus petite arité (nombre de sommets différents qui interviennent dans la condition). On récupére l'expression associée à cette condition et on l'évalue pour l'ensemble de sommets trouvés. Le traitement reste identique à chaque appel récursif de type P(G) jusqu'à ce que le cas de base de la récursivité soit atteint.

La classe possède deux constructeurs :

– `PolynomeRecursif(DefinitionRecursive, Graphe)` est appelé au lancement du calcul pour instancier la valeur initiale du graphe et la variable statique de type DéfinitionRecursive *definition*. La variable est **static** pour pouvoir la réutiliser dans toutes les instances de la classe car la définition récursive reste la même pour tous les appels recursifs.

- `PolynomeRecursif(Graphe)` est utilisé pour construire une nouvelle instance de PolynomeRecursif et donc implicitement un nouveau calcul, à chaque appel récursif de type $P(G)$. Le graphe pris en paramètre est celui obtenu par les opérations décrites .
- La méthode `void rechercheExpression()` trouve l'ensemble de sommets le plus petit lexicographiquement (ou d'après l'ordre spécifié), satisfaisant une condition de plus petite arité. Pour cela nous avons utilisé la méthode `int[] k\_upletSuivant(int,int,int)` qui renvoie la position de l'ensemble suivant dans la liste des ensembles de sommets. Ainsi, on parcourt la liste ordonnée des conditions de la même arité que l'ensemble de sommets du k-uplet courant, et on retient la première qui soit satisfaite. Le k-uplet trouvé est stocké dans le membre privé de la classe `int sommetsCondition[]`.
- La méthode `boolean evalueCondition(String)` prend en paramètre une chaîne de caractères qui représente la condition dans laquelle on a remplacé les termes a, b,c par les sommets correspondants dans SommetsCondition. Cette chaîne est passée en paramètre à la méthode `parse()` de la classe **parserCondition** pour évaluer la condition.
- La méthode *void calculPolynome(Expression)* prend en paramètre une expression avec les mêmes sommets que ceux utilisés pour évaluer la condition et retourne la valeur du polynôme calculé avec la méthode `parse()` de la classe **parserExpression**.

6.3.2 Paquetage définition

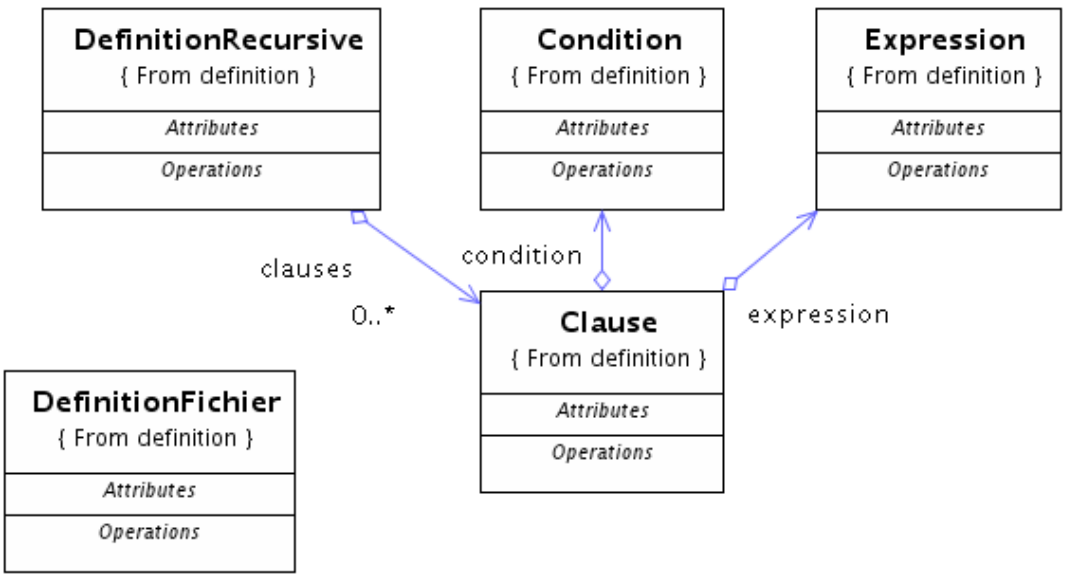


FIG. 6.3 – Paquetage definition

Ce paquetage comprend des classes qui manipulent les clauses, les conditions et les expressions des définitions récursives.

Classe Condition

Le constructeur de la classe prend en paramètre une chaîne de caractères qui représente la condition saisie par l'utilisateur et en extrait le nombre de sommets différents pour récupérer l'arité. La méthode `remplaceSommetsCondition(Integer[])` remplace les sommets donnés sous la forme de lettres a, b, c, ... par les sommets du graphe contenus dans le tableau d'entiers fourni en paramètre.

Classe Expression

Cette classe correspond à l'expression. De la même manière que la classe **Condition**, le constructeur prend en paramètre la chaîne de caractères saisie par l'utilisateur.

Classe Clause

La classe **Clause** contient deux champs privés *expression* et *condition*.

Classe DefinitionRecursive

Le membre important de cette classe est la liste *(type ArrayList)* de clauses. La fonction `boolean ajouterClause(Clause)` ajoute une clause à la liste en conservant l'ordre sur les arités et d'insertion.

Classe DefinitionFichier

Cette classe comprend des méthodes de chargement, de lecture.

6.3.3 Paquetage polynome

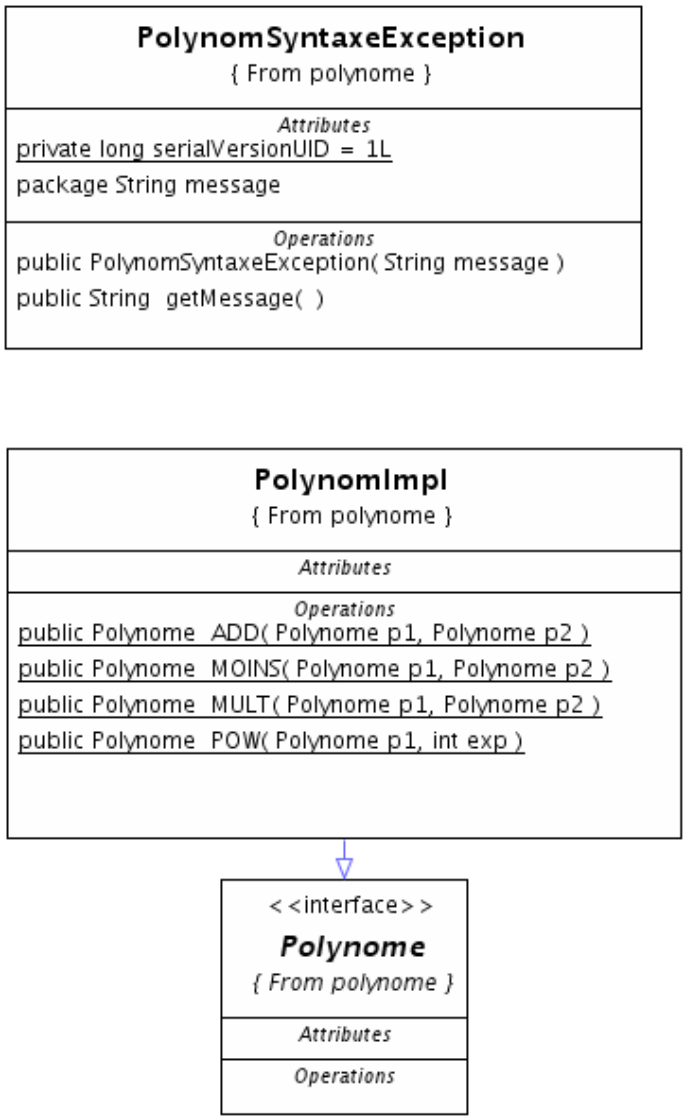


FIG. 6.4 – Paquetage polynome

Rôle du paquetage

Ce paquetage fourni le service de calcul symbolique nécessaire à l'application. Il est utilisé principalement par le paquetage **analyseur** pour le calcul récursif de polynomes, et par le paquetage **graphique** pour l'implémentation des calculs sur les polynomes résultats. Ce paquetage sert en fait de *façade* à la bibliothèque JSCL, citée précédemment, qui réalise effectivement les calculs formels.

Composition du paquetage

Le paquetage est composé d'une interface : **Polynome** qui décrit l'ensemble des fonctions utilisées par l'application. **PolynomImpl** est la classe qui implémente ces fonctions ainsi qu'une *Fabrique* d'opérations

arithmétiques et une classe **PolynomSyntaxeException** qui permet de répercuter les exceptions provenant de la bibliothèque vers le reste de l'application.

Présentation de l'Interface

L'interface **Polynome** décrit les services attendus pour un objet polynome. Il s'agit en résumé :
- des *getters* sur les éléments constitutifs d'un polynôme, à savoir le degré, les variables et coefficients de chaque monome.
- d'une méthode `developpe()` qui fournit le Polynome résultant d'une opération.
- des fonctions particulières requises pour l'application (substitution, synthèse de calcul ..)

Implémentation des calculs

Les concepts mathématiques utilisés par cette bibliothèque étant plutôt complexes, le choix a été fait d'en utiliser les fonctionnalités de hauts niveaux, en simulant des appels à la bibliothèque depuis l'interface **meditor**. Ainsi les objets fondamentaux manipulés sont des **Expression** (il s'agit de la classe **Expression** fournie par la bibliothèque JSCL), et les calculs effectués consistent à générer des objets **Expression** représentant les calculs souhaités. Puis à les développer (méthode *developpe()* décrite par l'interface) qui convertit une *Expression* en Polynome.

Les opérations arithmétiques sont codées sous la forme d'une *fabrique*, il s'agit donc de méthodes de classe, qui prennent en paramètre un ou plusieurs objets de type *Polynome* et qui renvoient un autre objet *Polynome*. En interne, la fabrique réalise ces opérations sur des objets **Expression**.

Par exemple l'opération addition de Polynome s'écrit :

```
public static final Polynome ADD(final Polynome p1, final Polynome p2) {
return new PolynomImpl(((PolynomImpl) p1).ExprPoly.add(((PolynomImpl) p2).ExprPoly)).developpe();
}
```

L'opération **ADD** est en fait réalisée à partir de l'opération *add* sur des **Expression**. Toutes les opérations sont codées de cette façon.

Le modèle de conception de la **Fabrique** a été préféré à une implémentation plus classique des opérations, du type `p1.add(p2)` pour des raisons d'harmonie avec le paquetage *graphe* et de sécurité des traitements lors du parsing. En effet, lors de l'analyse d'une expression du type :$P(G) * (x + 1) + P(G)$, il était essentiel pour nous d'être certain que pour un même niveau de récursion, le graphe G ainsi que le polynôme $P(G)$ à chaque occurence dans l'expression. Ce procédé nous oblige certes à recréer de nouveaux objets. Il est cependant beaucoup plus sûr et évite de faire des clonages d'objets, rendant ainsi le code beaucoup plus clair.

L'inconvénient majeur lié à l'utilisation de la bibliothèque est que les polynômes ne sont pas représentés sous une forme canonique, l'accès aux éléments nécessite donc l'évaluation de la variable de type **Expression** représentant le *Polynome*. Pour éviter de reproduire cette évaluation, elle est systématiquement effectuée lors de la construction des objets, par une méthode privée : `valueOf()`, ensuite elle est conservée dans une variable membre *ExprPoly*.

L'implémentation des calculs a été considérablement facilité par l'usage de cette bibliothèque. Cette brique de base a été rapidement disponible. Dans une première version, les opérations étaient directement implémentées avec des objets de type **String**, dans une deuxième version les opérations portaient sur des **Polynome**. La dernière version, qui n'a pas eu d'impact sur le reste de l'application a consisté à rationaliser l'utilisation de la bibliothèque JSCL, en introduisant notamment une variable membre de type *Expression* qui évite de reproduire les mêmes calculs.

Traitements requis pour l'analyse des polynômes

Les opérations requises concernent le polynôme résultant d'un calcul récursif. Ce sont des opérations de base qui consistent à calculer :
- le polynôme en valuant tout ou partie des indéterminées,

– à en afficher une représentation synthétique (en gardant les indéterminées),
– à réaliser des opérations arithmétiques sur plusieurs polynômes.

Substitution de variables La substitution de variables est réalisée à partir des fonctionnalités de la bibliothèque JSCL, (méthode`substitute`). Pour plus de souplesse à l'utilisation, une version vectorielle de cette opération est proposée *MultiSubstitute*, elle permet de subsituer en une commande plusieurs variables.

Filtrage du polynôme Le filtrage du polynôme consiste à parcourir et afficher les monômes selon certains critères. Il fait appel aux *getters* du polynôme, pour parcourir successivement tous les monômes. Cette fonctionnalité, à l'origine assez peu performante en temps, a été fortement accélérée par le stockage de l'expression évaluée lors de la construction d'un polynôme. Les monômes sont triés d'abord par ordre lexicographique il est alors nécessaire de tous les parcourir pour pouvoir filtrer le degré. A titre indicatif, un polynôme de 1024 monômes et de 11 variables est filtré environ en 10 secondes sur une machine DualCore récente. D'une manière générale, il a été choisi de se baser complètement sur la bibliothèque, et de ne pas sur-dimensionner la taille des objets polynômes plus que nécessaire.

Descriptif du polynôme Cette opération fournit des informations générales sur le polynôme (nombre de variables, de monômes et degré maximal) et du fait du stockage de l'expression, le temps de réponse est quasi-instantané (pour le polynôme cité en exemple ci-dessus).

Gestion des exceptions

La gestion des exceptions dans le paquetage *Polynome* correspond à deux types d'erreurs :

1. les exceptions **ParseException** levées par la bibliothèque,
2. les exceptions levées par la classe **PolynomImpl**.

Il faut noter qu'en principe les expressions évaluées lors du processus de calcul d'un polynôme récursif, ont préalablement subies une analyse syntaxique. De plus notre grammaire est incluse dans la grammaire reconnue par la bibliothèque (cf. annexe) donc à priori ces exceptions ne devraient pas être interceptées lors des calculs récursifs.
Pour les opérations élémentaires, toutes les exceptions provenant de la bibliothèque sont interceptées, lors de l'évaluation (méthode privée *valueOf()*) puis systématiquement converties en exception de type *PolynomSyntaxeException* et transmises aux appelants. Ces exceptions donc peu probables lors du calcul récursif ont été regroupées avec les exceptions générées lors des post-traitements (substitution , filtrage) qui sont déclenchées notamment pour des raisons de mauvais paramétrage, et traitées dans le paquetage **graphique**.

6.3.4 Paquetage graphique

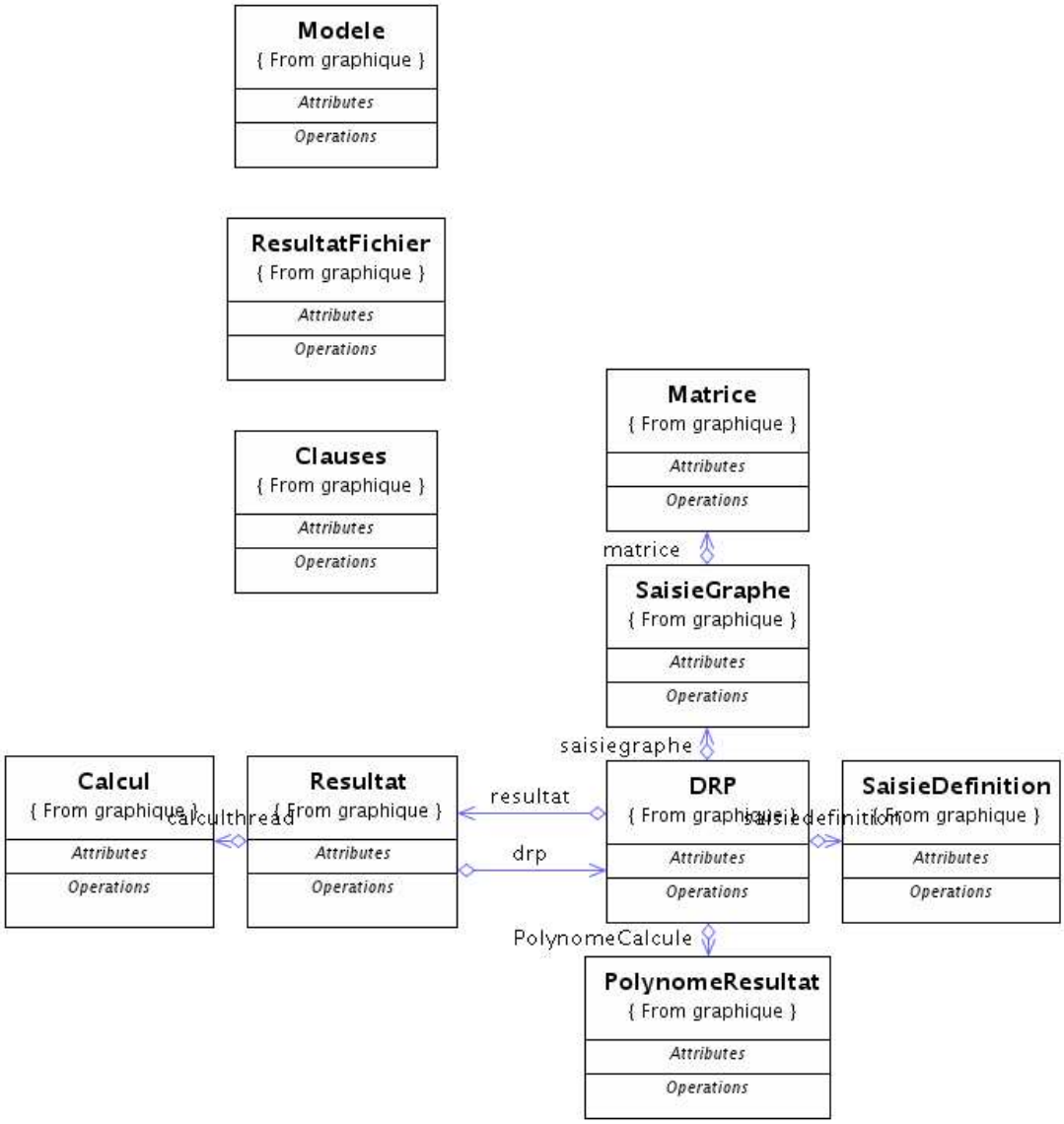


FIG. 6.5 – Paquetage graphique

Le paquetage graphique contient les classes construisant l'interface graphique. Il se compose de onze classes dont une applet.

DRP

Cette classe permet l'initialisation de l'applet. Elle contient également tous les *ActionsListeners* pour le menu de l'applet (sauvegarder, charger) ainsi que ceux pour le calcul (lancer, arrêter...) et les opérations sur les polynômes. **DRP** crée la base de l'applet qui sera visible (création des onglets et de leur contenu ainsi que du menu). Lors du chargement d'un graphe, un appel à la classe **GrapheFichier** et plus précisement *chargerGraphe()* est fait. Cette méthode prend en paramètre une chaine de caractères et renvoie un graphe qui

est ensuite transformé pour l'affichage. Le même système est présent pour le chargement et la sauvegarde des définitions et des résultats. C'est à cet endroit que les vérifications, pour effectuer les actions, sont faites.

SaisieGraphe

Ce panneau permet la saisie des différentes données sur les graphes comme le nombre de sommets et le nom donné pour chaque graphe. Ce panneau se décompose en plusieurs parties.

- Une partie pour la saisie des données souhaitées par l'utilisateur (*panneausaisie*).
- Une autre qui permet à l'utilisateur de lister ses graphes pour éviter de devoir les resaisir(*panneauliste*). Ceci se fait à l'aide d'une JList pour l'affichage. Pour la correspondance entre le nom du graphe et le graphe lui même, une HashMap est utilisée (*HashMap <Object, Graphe>*). Le choix de la HashMap a été fait pour éviter les doublons ce qui facilite le traitement. Ces deux parties sont codées au sein même de ce fichier.
- La dernière partie de ce panneau contient la matrice d'adjacence créée avec la classe **Matrice** (voir suite). Avant l'affichage, certaines validations sont effectuées afin de s'assurer que les données saisies sont cohérentes.

Le champ *ordre des sommets*, permet de spécifier l'ordre dans lequel seront générés les k-uplets de sommets (par défaut c'est l'ordre lexicographique 1,2,3,...) qui sera pris en compte pour les calculs.

FIG. 6.6 – Onglet SaisieGraphe

Matrice

Cette classe est un Jpanel. Elle construit la matrice d'adjacences en récupérant les données saisies par l'utilisateur pour construire la matrice correspondante. La matrice est représentée par un JTable. Dans cette classe il existe également les méthodes de conversions de l'interface vers un graphe (*conversionInterfaceGraphe(String, EnsembleSommets)*) et inversement (*conversionGrapheInterface(Graphe)*) qui sont notamment utilisées lors des sauvegardes et des chargements de fichiers graphes.

FIG. 6.7 – JTable repésentant la matrice

Modele

La classe permet de créer son propre modèle à utiliser ensuite dans une JTable. Dans cette classe, les fonctions principales propres aux JTables sont redéfinies. Nous pouvons par exemple choisir si les cellules d'un JTable sont éditables ou non.

SaisieDefinition

Ce panneau permet la saisie des définitions récursives. Il se décompose de la même façon la classe Saisie-Graphe. Comme pour **SaisieGraphe**, une JList et une HashMap sont présentes pour garder les définitions déjà saisies(*HashMap<Object, DefinitionRecursive>*).

FIG. 6.8 – Onglet SaisieDefinition

Clauses

Cette classe crée la table permettant le saisie des différentes clauses. Elle prend en paramètre le nombre de clauses saisies par l'utilisateur et construit la Jtable. Pour cette table le modèle utilisé est le modéle par défaut. Une méthode de conversion (*conversionDefinitionInterface(DefinitionRecursive)*)est utilisée pour récupérer les clauses lors d'un chargement.

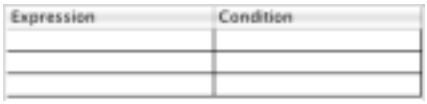


FIG. 6.9 – JTable représentant les clauses

Calcul

La classe **Calcul** hérite d'un Thread. Ceci permet de garder la main au niveau de l'interface pendant que s'effectue le calcul. Elle récupère une définition récursive *d* et un graphe *g* et constuit le polynôme récursif *Pol* à partir de ces données. Puis dans la méthode *run()* du thread, le calcul est lancé puis affiché.

Résultat

Cette classe construit le panneau contenu dans le troisième onglet. Une liste est aussi présente pour garder disponibles les résultats des précédents calculs. Ceci est toujours géré par une HasMap (*HashMap <Object, PolynomImpl>*). Les méthodes *ajouterItem(String...)* et *supprimerItem(String...)* sont nécessaires pour ajouter des entrées dans la liste des résultats.
Le panneau est composé de 4 parties :

- La liste des polynomes résultats précédemment calculés ou chargés ;
- Une *JTextArea* pour les résultats des différents calculs ;
- Un panneau de paramétrage des calculs (à droite) ;
- Un panneau pour les *JButton* appelant les calculs.

Le choix de l'applet, de taille fixe dans une page HTML, nous a conduit à rendre les différentes parties de cet onglet redimensionnables. En effet, les polynômes résultats peuvent comporter de nombreux monômes (potentiellement plusieurs milliers). Nous avons donc estimé qu'il était pratique de pouvoir maximiser la taille du composant dédié à l'affichage des résultats.

FIG. 6.10 – Onglet Resultat

PolynomeResultat

Cette classe fait le lien entre l'interface et les fichiers de sauvegarde des résultats (voir **ResultatFichier**). Le polynôme résultat est construit avec *PolynomeResulat(Graphe, DefinitionRecursive, String)* où String correspond au résultat du calcul avec un graphe et une définition récursive. Lors de la sauvegarde d'un résultat, les méthodes *getGraphe()*, *getDefinition()* et *getResultat()* sont appelées pour sauvegarder chaque paramètre du calcul.

6.3.5 Paquetage graphes

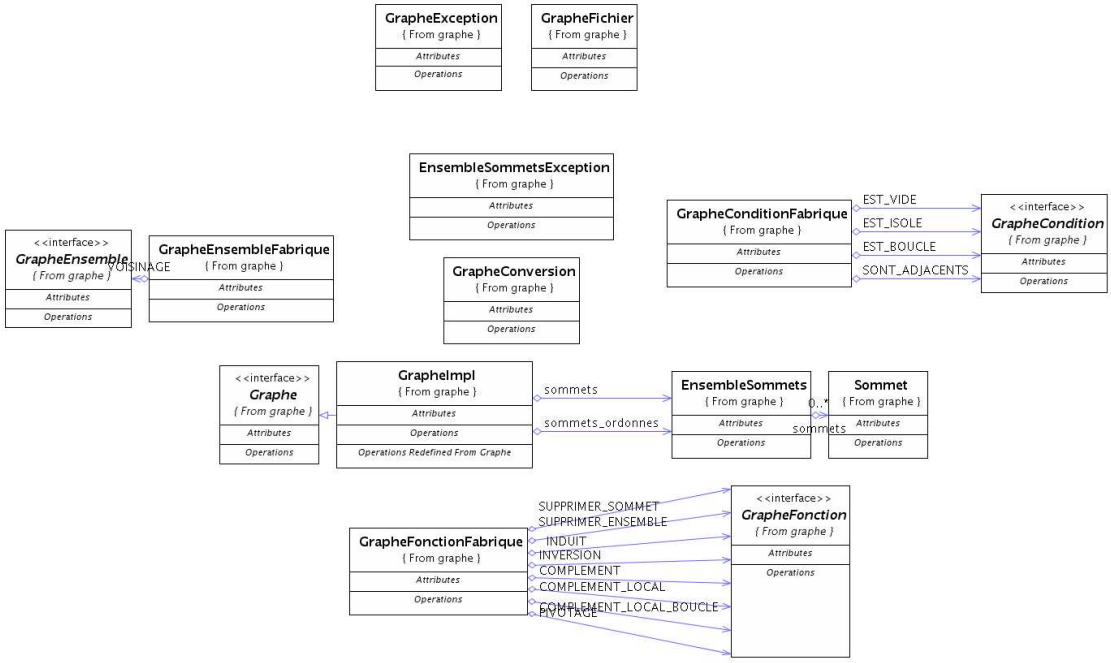


FIG. 6.11 – Paquetage graphe

Un graphe G se représente par l'ensemble de ses sommets V_G et sa matrice d'adjacence A_G (ou ses listes l'adjacences). Les éléments fondamentaux ont donc été implémentés afin de représenter un graphe. En l'occurrence, un sommet, un ensemble de sommets et le graphe lui-même.

Les sommets

Étant identifié par un entier, un sommet est descriptible par une variable de type entier. Il a néanmoins semblé préférable d'en créer un objet afin de ne pas avoir d'erreurs liées à un typage inapproprié. Qui plus est, l'extensibilité du projet s'en trouve améliorée.

L'objet `Sommet` a donc été créée conformément au type abstrait correspondant. Il en résulte deux méthodes permettant respectivement de créer un sommet à partir de son identifiant (un entier), et de récupérer cet identifiant pour un sommet donné. Les méthodes conventionnelles `clone`, `equals` et `toString` ont par ailleurs été redéfinies.

L'ensemble des sommets

L'ensemble des sommets d'un graphe est un intervalle ordonné d'entiers. Cet intervalle est éventuellement discontinu du fait de la suppression de sommets, mais demeure tout de même ordonné suite à cette opération.

Une liste a donc semblé être la structure la plus appropriée pour représenter un ensemble de sommets. En effet, la suppression d'un élément de celle-ci ne modifie pas l'ordre de ceux restants.

L'objet `EnsembleSommets` a donc été créé avec une variable d'objet privée de type `ArrayList <Sommet>`. Cet objet comporte les méthodes d'objet effectuant les opérations classiques sur une liste. Les méthodes `clone`, `equals` et `toString` ont ici aussi été redéfinies.

L'adjacence des sommets

L'adjacence des sommets d'un graphe peut se décrire par une matrice d'adjacence ou des listes d'adjacences. La performance ne faisant pas partie des besoins non fonctionnels, la structure choisie a été celle supposée présentant le moins d'inconvénients d'implémentation.

L'adjacence des sommets du graphe est donc représentée par une matrice d'adjacence. Il s'agit ici d'une matrice carrée d'entiers. L'utilisation d'entiers en lieu et place de booléens a été retenue. Bien que suffisants pour décrire la matrice d'adjacence d'un graphe non orienté, les booléens présentent l'inconvénient de restreindre l'extensibilité ou la réutilisabilité. A l'inverse, les entiers permettent aisément d'étendre le paquetage afin de traiter des graphes orientés.

Le graphe

L'interface Le calcul d'un polynôme à partir de sa définition récursive étant paramétré par un graphe, il a été créé une interface permettant de définir les opérations "garanties" associées au type abstrait graphe.

L'implémentation L'objet représentant le graphe se compose donc de deux variables privées d'objet. D'une part, une variable de type `EnsembleSommets` représentant l'ensemble des sommets du graphe. De l'autre, une variable de type `int[][]` représentant la matrice d'adjacence.
Par ailleurs, l'objet comporte une autre variable de type `EnsembleSommets`. Celle-ci permet de décrire l'ordre de parcours des sommets défini par l'utilisateur.
Enfin, deux chaînes de caractères permettent d'associer un nom et un commentaire au graphe.

Cette objet comporte les méthodes effectuant les opérations décrites par le type abstrait `Graphe non orienté` (l'opération déterminant le degré d'un sommet, ici absente, s'obtient par le cardinal du voisinage du dit sommet (`sommet.voisinsSommet().nombreSommets()`)).
Deux méthodes supplémentaires ont été définies afin de calculer le rang de la matrice d'adjacence du graphe, ainsi que son nombre de composantes connexes.
Comme auparavant, `clone`, `equals` et `toString` sont ici redéfinies.

Par ailleurs, des méthodes liées aux fonctions et conditions portant sur un graphe ont été définies en supplément. Leur utilité est justifiée dans la partie suivante traitant les fonctions et conditions.

Les fonctions et conditions

Aspect fonctionnel Les fonctions et conditions portant sur des graphes ont été implémentées à l'aide de fabriques de fonctions. En effet, les expressions et conditions composant les clauses d'une définition récursive nécessitent d'être définies avant un calcul. Néanmoins les graphes et sommets paramétrant ces fonctions et ces conditions sont inconnus lors de la création de la définition récursive.
Il s'agit là typiquement d'une émulation du style de programmation fonctionnel dans le cadre d'une programmation orientée objet.

Les interfaces En préalable à l'implémentation des fonctions et conditions, trois interfaces ont été créées afin de préciser les méthodes assurées par les fonctions définies dans les différentes fabriques.

En premier lieu, nous avons donc la fabrique de conditions. Celles-ci sont paramétrées par un graphe et un ensemble de sommets, et renvoient une valeur de vérité.
Ensuite, nous avons les fabriques de fonctions. On en distingue ici deux. Bien que paramétrées par un graphe et un ensemble de sommets, les fonctions des deux fabriques diffèrent par l'élément retourné. D'un côté nous avons une fabrique dont les fonctions doivent renvoyer un graphe. De l'autre, nous avons une fabrique dont les fonctions doivent renvoyer un ensemble de sommets (l'opération déterminant le voisinage d'un sommet est l'unique opération définie par cette fabrique).

L'implémentation L'implémentation s'effectue donc au travers des trois classes suivantes :
- `GrapheConditionFabrique` (implémente l'interface `GrapheCondition`) ;
- `GrapheEnsembleFabrique` (implémente l'interface `GrapheEnsemble`) ;
- `GrapheFonctionFabrique` (implémente l'interface `GrapheFonction`).

Les fonctions définies exploitent ici essentiellement les méthodes définies à l'aide de l'interface `Graphe`. Elle utilisent par ailleurs les méthodes de la classe `EnsembleSommets`. En effet les ensembles de sommets et les sommets paramètrent la majeure partie des fonctions et conditions.
L'utilisation d'un ensemble de sommets en tant que paramètre présente ici l'intérêt de ne devoir implémenter qu'une unique méthode, et cela quel que soit le nombre de sommets paramétrant une opération.
Ainsi, une fonction non paramétrée par un sommet ou un ensemble de sommet se voit passer en paramètre un ensemble de sommets vide. A l'inverse, une opération paramétrée par un nombre déterminé de sommets se voit passer en paramètre un ensemble de sommets dont le cardinal est ce même nombre déterminé.

Les exceptions

Les exceptions concernant les méthodes d'un ensemble de sommets et d'un graphe se traduisent ici par les classes `EnsembleSommetsException` et `GrapheException`. Étendant la classe `Exception`, celles-ci sont susceptibles d'être levées, de manière générale, dès lors qu'une opération est utilisée sans que ses pré-conditions soient vérifiées.

6.4 Scenario type d'exécution

Décrivons à présent le processus d'exécution lors des différentes phases d'utilisation.

6.4.1 Initialisation d'un graphe et d'une définition récursive

Une fois chargées dans l'application, les définitions peuvent être stockées dans une **HashMap** avec comme identifiant leur nom. La définition et le graphe pris en compte pour le calcul sont les derniers affichés. Ils sont stockés dans les variables membre correspondantes de classe **DRP**.

6.4.2 Calcul récursif

Le lancement du calcul est réalisé par l'appel de la méthode *start()* d'un objet de type **Calcul**. Le calcul est dans un premier temps initialisé, par la construction d'un **PolynomeRecursif**. La trace de cette initialisation est fournie ci-dessous par une impression d'écran issue du *profiler*. La lecture des appels doit être effectuée de manière ascendante.

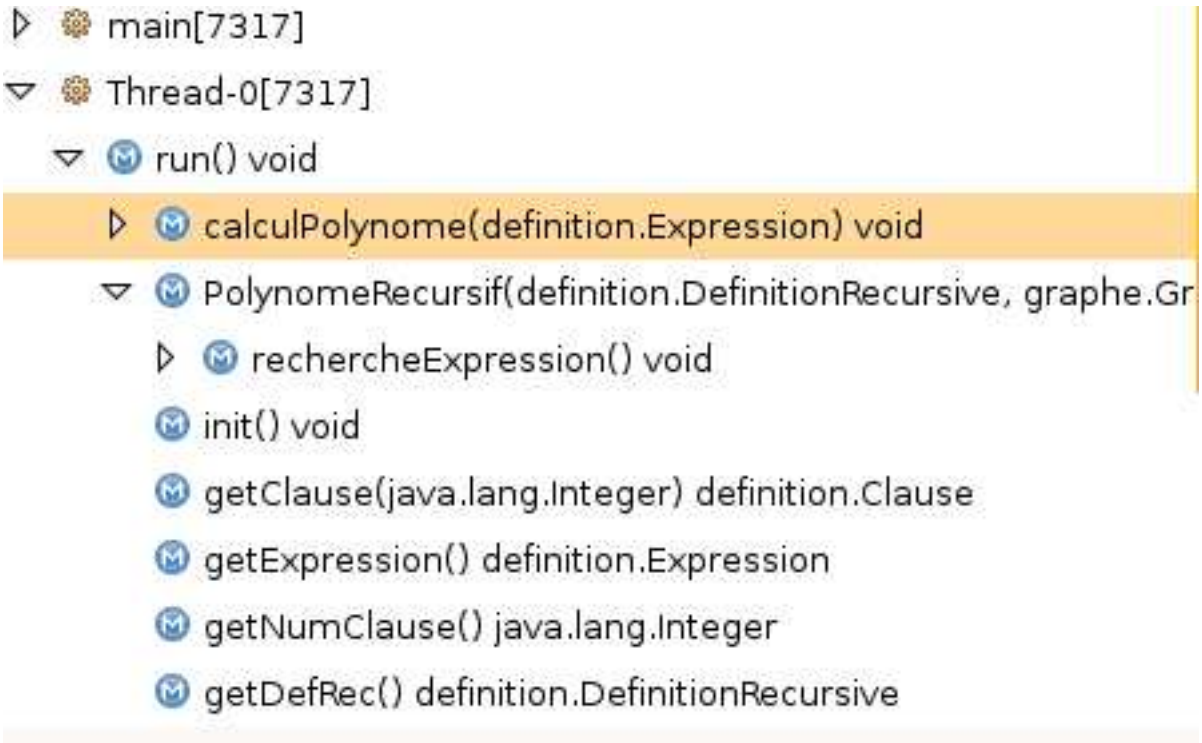


FIG. 6.12 – Initialisation du calcul

Cette initialisation nécessite un appel à la méthode `rechercheExpression()` qui trouve la première condition et le premier k-uplet de sommets d'après l'ordre spécifié (par défaut l'ordre lexicographique). La trace de cette méthode est représentée ci-dessous.

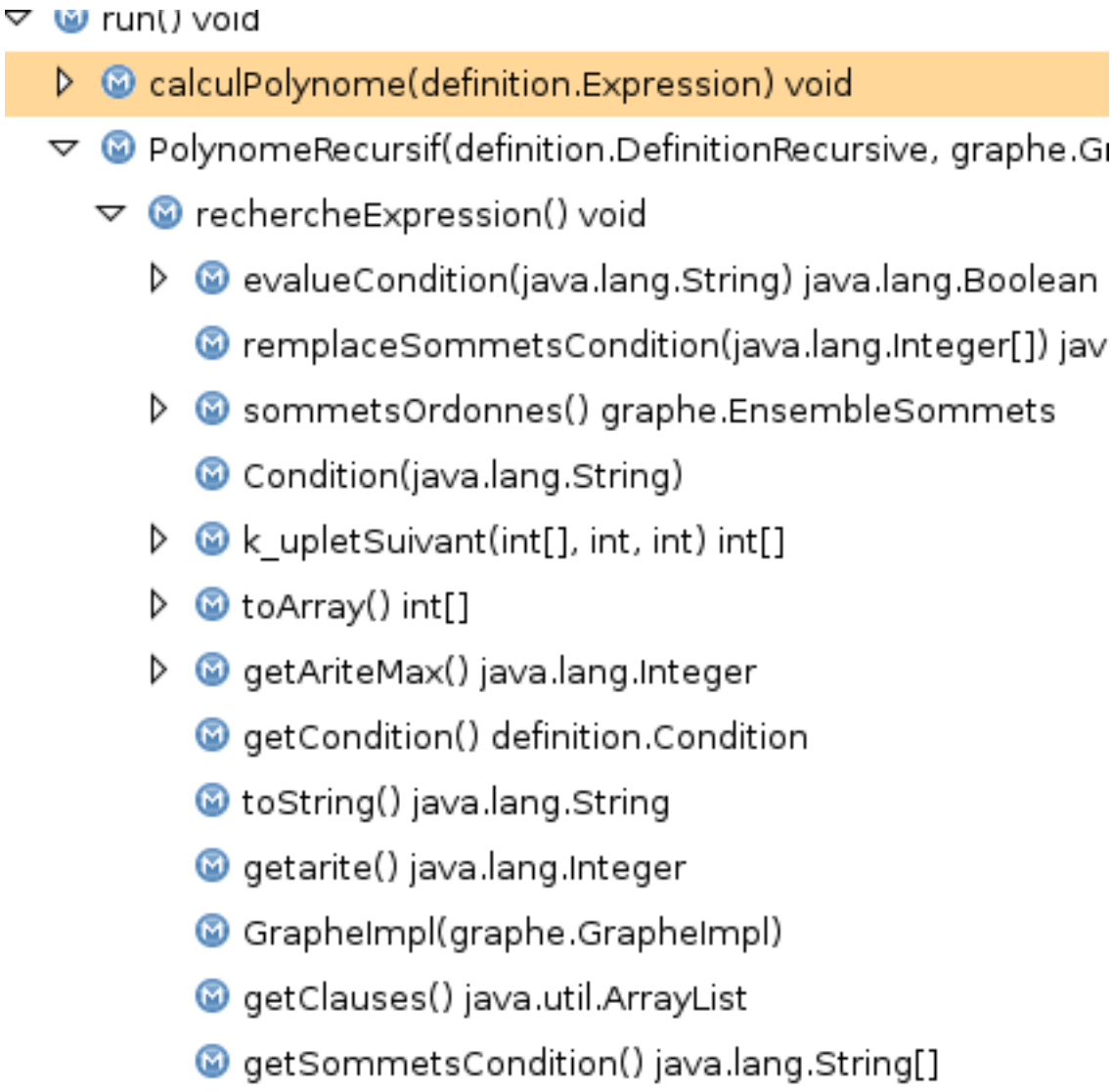


FIG. 6.13 – Recherche de l'expression à appliquer

1. pour cela on génère dans cet ordre les k-uplets,
2. ensuite pour chacun on remplace les sommets génériques de la condition par ceux du k-uplet,
3. on *parse* la condition ainsi obtenue en récupérant le booléen à la racine de l'arbre syntaxique (vrai ⇒ le k-uplet satisfait la condition).
   (a) on récupère l'expression correspondant à la condition satisfaite,
   (b) on remplace le k-uplet dans l'expression,
   (c) on la *parse*,
   (d) lors de l'analyse syntaxique de l'expression, chaque fois que l'on réduit une règle de type PolyRec ⟶ P(G), dans l'action associée à la grammaire, on instancie un objet de type **PolynomeRecursif** avec le deuxième constructeur (
   `new PolynomeRecursif(Graphe))`
      i. on recommence à l'étape 2 pour l'appel récursif suivant (le *Graphe* est celui obtenu par des opérations sur le graphe initial).

(e) on récupère le polynôme

6.4.3 Analyse du résultat

Sur tout polynôme calculé, il est possible d'effectuer trois types d'évaluation :
- substitution de variable(s) sur un polynôme,
- filtrage d'un polynôme,
- opérations arithmétiques sur plusieurs polynômes.

6.5 Complexité

Dans ce chapitre nous nous intéresserons à la complexité du calcul du polynôme à partir d'une définition récursive et d'un graphe. Dans un premier temps on évaluera la complexité seulement en fonction du nombre des appels récursifs.

La complexité du programme au pire des cas est exponentielle. Plus exactement elle est de $O(r^n)$, r étant le nombre maximal d'appels récursifs et n - la profondeur de l'arbre des appels récursifs (c'est souvent le nombre de sommets du graphe). Un exemple de comportement exponentiel est le graphe complet K_n pour la définition suivante :
$Q(G) = x.Q(G - a)$, si le sommet a est isolé
$Q(G) = Q(G - a) + Q(G^a - a) + Q(G^{ab} - b)$, si les sommets a et b sont adjacents
$Q(G) = 1$, si G est un graphe vide
Ici, la condition "les sommets a et b sont adjacents" est toujours satisfaite (un graphe complet est avec tous les sommets reliés entre eux), ce que implique que le nombre des appels récursifs est de 3^n. Un tel ordre de grandeur est justifié par la nature exponentielle de la plupart des définitions récursives.

Regardons maintenant la complexité au niveau du nombre d'opérations nécéssaires pour trouver un ensemble de sommets satisfaisant une condition. Pour chaque appel récursif la fonction de recherche de l'expression génère des k-uplets de sommets en fonction des conditions de la définition. Seuls les k-uplets où k est l'arité d'une des conditions sont traités. On parse les conditions d'arité k jusqu'à ce qu'une soit vérifiée. Nous remarquons que la fonction ne génère pas tous les k-uplets possibles (mots de longueur k) car un sommet ne peut apparaître qu'une seule fois dans chaque ensemble de sommets satisfaisants la clause. Comme dans le cas des appels récursifs, nous nous plaçons dans le cas le plus défavorable : pour un graphe à n sommets, toutes les clauses ont la même arité k ($k \leq n$) la plus proche possible de n. Si uniquement la dernière condition est satisfaite, au pire des cas, l'application va effectuer $m \cdot A_n^k$ *parsings* de conditions pour les évaluer avec chaque k-uplet, m étant le nombre de clauses.

Ainsi la complexité du calcul, au pire des cas est de $O((rt)^n)$, avec $t = m \cdot A_n^k$ - le nombre d'opérations nécéssaires pour trouver un k-uplet satisfaisant une condition d'arité k.

Un tel ordre de grandeur rend inexploitable l'application pour un graphe avec un nombre de sommets supérieur à 20. Le calcul des polynômes ne peut être effectué que pour des graphes de "petite" taille. Ceci correspond aux besoins du clients, car souvent pour tester certaines hypothèses il suffit de les verifier pour des graphes à un nombre de sommets inférieur à 20.

Calcul itératif du polynôme

Lors de la phase de réflexion sur le projet, il a semblé évident de calculer récursivement le polynôme du fait de sa définition récursive. Nous avons néanmoins aussi envisagé d'effectuer le calcul de manière itérative, ayant à l'esprit les inconvénients liés à une pile d'appels.

Après discussion avec notre chargé de travaux dirigés de projet de programmation, nous avons entamé la méthode récursive et la méthode itérative, dans la perspective de n'en retenir qu'une. Ainsi, devant les risques et difficultés potentiels de la version itérative, nous avons concentré l'intégralité de nos efforts sur la méthode récursive.

Nous tenons néanmoins à présenter notre sur travail sur la méthode itérative, ainsi que l'intérêt qu'elle présente au regard des résultats obtenus.

Chapitre 7

Formats de données et grammaires

7.1 Syntaxe utilisateur

Nous avons uniformisé la saisie des clauses. Ainsi les syntaxes proposées pour l'interface et pour les fichiers de sauvegarde, sont proches de celle proposée dans le sujet. Si ce n'est que chaque condition et chaque expression doit se terminer par un " ;", pour indiquer au parseur la fin de la chaîne de caractères à traiter.

7.1.1 Condition

- a **et** b **adjacents ;** - pour la condition a et b sont adjacents
- **G vide ;** - pour la condition que le graphe est vide
- $N(G, a)$ **vide ;** - pour la condition que a est un sommet isolé
- a **incident boucle ;** - pour la condition que a est incident à une boucle
- **non**(condition) **;** - la négation de la condition élémentaire

Pour faire une combinaison booléenne entre les conditions il faut ajouter l'opérateur booléen ("et", "ou") voulu entre deux conditions.

7.1.2 Expression

Sommet

Les sommets sont des lettres de l'aphabet latin de a à s, sauf c et p.

Variable

Les variables sont des lettres de l'alphabet latin de t à z.
- **x** - une variable ordinaire
- **x_a** - la variable x associé au sommet a (x_a)

Ensemble de sommets

Un ensemble X peut être :
- **N(G,a)** - le voisinage du sommet a
- **{a,b,c,...}**

Graphe

- **G-a** - l'opération de suppression du sommet a
- **G - X** - l'opération de suppression de l'ensemble de sommets X
- **G[X]** - graphe induit par l'ensemble de sommets X
- **c(G)** - l'opération de complémentation des arêtes sans toucher aux boucles
- **lc(G,a)** - l'opération de complémentation locale

- **inv(G,X)** - l'opération de complémentation des boucles des sommets de X. X est un ensemble
- **bc(G,a)** - l'opération de complémentation locale étendue aux boucles
- **p(G,a,b)** - l'opération de pivotage sur (a, b)

Autres opérations

- **rk(G)** - l'opération de calcul du rang du graphe
- **cc(G)** - l'opération de calcul du nombre de composantes connexes dans le graphe
- ^ - l'exposant

7.2 Gestion des fichiers

La gestion des fichiers autorise la sauvegarde et le chargement de différents éléments à partir de fichiers. Ces éléments peuvent être un graphe, une définition récursive ou un résultat.

Un résultat présente la particularité d'être généré à partir d'une définition récursive et d'un graphe. Ces éléments doivent donc pouvoir être sauvegardés et chargés en même temps que le résultat.
L'approche consistant à référencer les fichiers contenant le graphe et la définition récursive à partir desquels le résultat a été calculé a été envisagée. Elle n'a néanmoins pas été retenue puisque qu'un résultat ne résulte pas nécessairement d'un graphe et d'une définition ayant été sauvegardés. Contraindre une sauvegarde du graphe et de la définition récursive étant un comportement inapproprié, ceux-ci sont ont donc été intégrés au fichier contenant le résultat.

Cette particularité a donc orienté l'implémentation des classes liées à la gestion des fichiers afin d'éviter toute duplication de code autant que faire se peut.

L'implémentation de la gestion des fichiers se traduit donc par :
- deux méthodes permettant respectivement de lire un graphe et une définition dans un fichier ouvert déterminé ;
- deux méthodes permettant respectivement d'écrire un graphe et une définition à partir d'un fichier ouvert déterminé.

Ainsi, le chargement et la sauvegarde d'un graphe ou d'une définition se traduit par l'ouverture de ce fichier puis l'écriture ou la lecture dans ce fichier à l'aide des méthodes décrites précédemment.

Concernant le résultat, on exploite ces mêmes méthodes en leur passant successivement en paramètre le même fichier de résultat déjà ouvert.

Format de lecture et d'écriture

Les formats des fichiers doivent répondre à une contrainte de lisibilité afin qu'ils soient modifiables aisément. Par ailleurs, ils doivent contenir diverses informations devant être reconnaissables (nom, commentaire, graphe, définition, résultat, ...).
On utilise donc un balisage minimaliste afin de distinguer, sans trop être intrusif, les différentes informations contenues dans les fichiers.
Une reconnaissance à l'aide de balises présente par ailleurs un avantage en terme de souplesses des formats des fichiers sauvegardes. Ainsi, bien qu'un format type soit défini pour chaque type de fichier de sauvegarde, les éléments contenus par ces fichiers peuvent être ordonnés différemment par les utilisateurs créant ou modifiant ces fichiers.

Format et syntaxe des fichiers de sauvegardes

Le format et la syntaxe des différents fichiers de sauvegarde sont définis ci-après. Afin que les utilisateurs puissent modifier les fichiers de sauvegarde, ils sont rappelés dans le manuel d'utilisation.

Les graphes Les informations stockées concernant un graphe ainsi que les balises associées sont les suivantes :

- `#` : commentaire sur le graphe ;
- `/` : nom du graphe.

L'ensemble de sommets doit être représenté sur une unique ligne :
- `{` : début de l'ensemble des sommets du graphe ;
- `,` : séparateur des sommets de l'ensemble ;
- `}` : fin de l'ensemble des sommets du graphe ;
- `{}` : ensemble de sommets vide.

La matrice d'adjacence se présente sur plusieurs lignes :
- `[` : début de la matrice d'adjacence du graphe (sur une ligne) ;
- `[` et `]` : début et fin d'une ligne de la matrice d'adjacence (sur la même ligne) ;
- ` ` : séparateur d'une ligne de la matrice d'adjacence ;
- `];` : fin de la matrice d'adjacence du graphe (sur une ligne).

Typiquement, nous avons donc un fichier tel que le suivant :

```
#commentaire
/nom du graphe
{ 1, 2, 3, 4, 5}
[
[0 1 0 0 1]
[1 0 1 0 0]
[0 1 0 1 0]
[0 0 1 0 1]
[1 0 0 1 0]
];
```

Le graphe vide se décrit quant à lui ainsi :

```
#graphe vide
/G
{}
[
];
```

Les définitions récursives de polynômes De même que pour les graphes un fichier contenant une définition récursive d'un polynôme se présente ainsi :
- `/` : nom de la définition récursive.

Une clause d'une définition récursive doit se décrire sur deux lignes (l'expression et la condition) :
- `=` : expression associée à une clause ;
- `si` : condition associée à une clause.

En illustration, nous avons donc un fichier dont le contenu est le suivant :

```
/Q(G)
= 1;
si G vide;
= x*Q(G - a );
si N(G, a ) vide;
= Q(G-a)+Q(p(G,a,b)-b);
si a et b adjacents;
```

Les résultats Un fichier de résultat comprend trois éléments de natures différentes comme expliqué auparavant. Un fichier de résultat est donc balisé de la sorte (les balises sont sur une ligne) :
- `<graphe>` : début de la définition d'un graphe ;
- `</graphe>` : fin de la définition d'un graphe.
- `<definition>` : début d'une définition récursive ;
- `</definition>` : fin d'une définition récursive.
- `<resultat>` : début d'un résultat.
- `</resultat>` : fin d'un résultat.

Pour le graphe et la définition récursive à partir desquels le résultat à été calculé, la syntaxe du texte entourée par les balises de début et de fin est similaire à celle décrite précédemment. Concernant le résultat, il est présenté sur une unique ligne.

L'ordonnancement des éléments des fichiers Pour les trois types de fichiers, les éléments d'information peuvent être décrits sans ordonnancement particulier.
Ainsi, par exemple pour un graphe, on peut parfaitement décrire la matrice d'adjacence, le nom, l'ensemble des sommets puis un commentaire. De même, pour un fichier de résultat, on peut aussi décrire le résultat, le graphe puis la définition (le graphe ou la définition récursive peuvent eux-mêmes être "désordonnés").
Ainsi, à la lecture, un fichier ne respectant pas le format "standard" préconisé ne sera pas un inconvénient. Néanmoins l'ordre sera restitué à l'écriture.

Chapitre 8

Tests

8.1 Les Tests Unitaires

L'exactitude des résultats des calculs étant très importante pour les clients, nous avons prédéfinis des tests unitaires pour chaque module intervenant lors des calculs. Ainsi, à chaque étape de la conception du logiciel nous avons pu vérifier que le calcul n'était pas altéré.

8.1.1 Tests unitaires des opérations sur les polynômes

Le paquetage **testPoly** est dédié aux tests du paquetage*polynome*. Il comprend une suite JUnit de 51 tests destinée à montrer :

1. l'exactitude des calculs arithmétiques implémentés par la **fabrique** (addition, soustraction, produit de polynomes et élévation d'un polynôme à une puissance entière) et respect des règles de base (commutativité, associativité).
2. l'exactitude des opérations sur les polynômes (*getters* sur les composants d'un polynôme, par exemple le degré du i-ième monome ...)
3. le comportement aux limites

Remarque : Les JUnits de la bibliothèque JSCL n'ont pas été ré-utilisés car pour notre application, le domaine de définition est limité aux entiers, alors que pour les test de la JSCL il s'agit des complexes.
En ce qui concerne les tests aux limites, hormis les tests avec des valeurs nulles, les tests produits démontrent que les calculs entiers sont réalisés avec des BigInteger (vont au delà des limites des types de bases int de 32 bits).

8.1.2 Tests unitaires sur les graphes

Afin de vérifier la justesse des différentes implémentations, nous avons défini des tests unitaires pour chaque élément fondamental. L'ensemble de ces tests figure dans le paquetage **testGraphe**

Les sommets

Les tests unitaires liés à cette classe relativement réduite portent principalement sur les méthodes de classe `Object` ayant été redéfinies.

Les ensembles de sommets

Il s'agit ici de s'assurer que l'ensemble résultant d'une opération est bien celui attendu. Par ailleurs, on effectue ici des tests aux limites, notamment dans le cas d'un ensemble vide ou d'un ensemble à un sommet.

Les graphes

Le principe des tests ici est sensiblement similaire à ceux des ensembles de sommets.

Par ailleurs, l'implémentation est basée sur le type abstrait `Graphe`, dont il s'agit de vérifier les axiomes.

Enfin, on exploite ici autant que faire se peut la notion de composition de fonctions. En effet, certaines fonctions sont l'inverse d'autres. On s'assure donc que la composition d'une fonction et de son inverse correspond à la fonction identité. Par exemple, un graphe auquel on ajoute le sommet 42 puis auquel on retire ce même sommet doit être inchangé.

La gestion des fichiers

Les tests unitaires concernant les fichiers consistent essentiellement à sauvegarder un objet (un graphe, un ensemble de sommet et/ou un résultat) dans un fichier. Il s'agit ensuite de charger l'objet à partir de ce même fichier, puis de s'assurer de l'égalité entre l'objet sauvegardé et celui chargé.

8.2 Tests en boîte noire

Les tests en *boite-noire* ont été réalisés à partir des définitions récursives existantes pour lesquelles nous disposions de résultats connus. Certaines fonctionnalités, et en particulier des opérations élémentaires sur les graphes n'apparaissant pas dans ces définitions, n'ont pas pu être éprouvées globalement.

8.2.1 Tests réalisés

Les tests unitaires sur les calculs récursifs, ont fait office de tests en **boite-noire**. Nous avons donc profité des facilités offertes par les *JUnits* pour automatiser ces tests. La saisie des graphes et des définitions et le lancement du calcul se résumant à :

```
Graphe g = GrapheFichier.chargerGraphe( REPTEST+"graphes/S3");
DefinitionRecursive DefRec = new DefinitionRecursive("Q(G)") ;
DefRec = DefinitionFichier.chargerDefinition(REPTEST+"definitions/Defstandard");
Calcul c1=new Calcul(DefRec,g);
c1.start() ;
while ( c1.isAlive() );
```

Une suite JUnits est équivalente à un traitement en *batch*.

Ce sont les tests finaux qui permettent de vérifier les résultats des calculs des polynômes à partir des définitions récursives et de types de graphes pour lesquels le résultat est déjà connu.

Le paquetage **testPolyRec** comprend une suite de 23 tests pour des graphes de taille entre 3 et 20 sommets. Les définitions récursives sont celles fournies par les clients dans la présentation du sujet, plus celles utilisées pour tester les calculs de l'application de nos prédécesseurs. Nous nous sommes particulièrement intéressés aux dernières car ce sont elles qui ont permis aux clients de s'apercevoir de l'inexactitude des calculs de nos pédécesseurs. Nous citerons donc en particulier, les tests 12 qui vérifie la propriété citée dans le sujet, suivant laquelle, pour la définition $P(G)$ ci-dessous, on constate que le produit des polynômes de deux graphes est égal au polynôme associé au graphe-union des deux graphes disjoints.
Nous avons de plus vérifié qu'en changeant l'ordre de suppression de sommet, le résultat des calcul ne changeait pas (tests 15 à 23).

Pour ces tests, l'ensemble des opérations arithmétiques sur les polynômes ont été implémenté.

A part la vérification des polynômes obtenus nous avons pu constater que le temps de calcul pour des graphes à un grand nombre de sommets (K_{12} par exemple) et pour des définitions avec trois appels récursifs

par expression, est non négligeable.

8.2.2 Les fonctionnalités non-testées globalement

Certaines opérations élémentaires sur les graphes ne faisaient pas parties des définitions récursives utilisées. Elles ont cependant fait l'objet de tests unitaires. En voici la liste :

- lc - Complémentation locale,
- rk - rang du graphe,
- cc - nombre de composantes connexes du graphe,
- G-X - suppression d'un ensemble de sommets (c'est une boucle de suppression de sommets qui elle a été testée).

8.2.3 Tests de l'interface graphique

Les tests effectués sur l'interface ont permis de traiter la majorité des exceptions qui pouvaient être levées. Cela comprend aussi, les exceptions issues de manipulations non-conformes à la notice utilisateur. Il se peut cependant que certaines exceptions persistent. L'organisation des traitements n'est en théorie pas altéré (à chaque étape de calcul, les données nécessaires sont réiniitalisées).

8.3 Tests de profil

Les tests de profil ont été réalisés avec le plugin d'Eclipse TPTP. La performance n'étant pas une contrainte, ces tests avaient principalement pour objectif d'illustrer l'étude de complexité. Cependant, les temps de traitements étant importants, lors du *profiling* nous avons dû nous limiter à des configurations minimales (Graphes S2, S3 et K3) avec la définition

- $P(G) = 1$ si G est vide,
- $P(G) = (1 + x_a) \cdot P(G - a)$ si a est isolé sans boucle,
- $P(G) = x_a u \cdot P(bc(G, a) - a) + P(G - a)$ si a est incident à une boucle, isolé ou non,
- $P(G) = x_a x_b u^2 \cdot P(p(G, a, b) - a - b) + P(G - a) + P(p(G, a, b) - b)$ si a est adjacent à b, a et b sans boucles.

Ces tests ont permis d'estimer les paquetages et les classes les plus sollicités ainsi que les méthodes les plus appelées. Les deux illustrations ci-dessous, mettent en évidence d'une part que les paquetages les plus sollicités sont ceux de la bibliothèque JSCL. On notera en particulier et comme on pouvait le prévoir, que les méthodes de calcul de la classe **Expression** sont les plus invoquées. Ces tests témoignent donc que le comportement de l'application est conforme à celui que nous avons envisagé lors de l'interfaçage de la bibliothèque JSCL, à savoir de simuler les calculs arithmétiques au niveau de cette classe.

analyseur	31,327572	0,030593	61,717902	1024
definition	0,042764	0,000269	0,042764	159
graphe	0,285999	0,000290	0,285999	985
graphique	0,230380	0,076793	30,874813	3
jscl.math	19,719781	0,000233	29,893679	84609
jscl.math.function	0,708740	0,000239	7,683009	2962
jscl.math.polynomial	5,225571	0,000286	12,323567	18249
jscl.math.polynomial.groebner	0,225290	0,000313	5,997286	719
jscl.text	3,986428	0,000247	5,770300	16164
polynome	0,167761	0,001234	30,061440	136

FIG. 8.1 – Synthèse des appels des paquetages

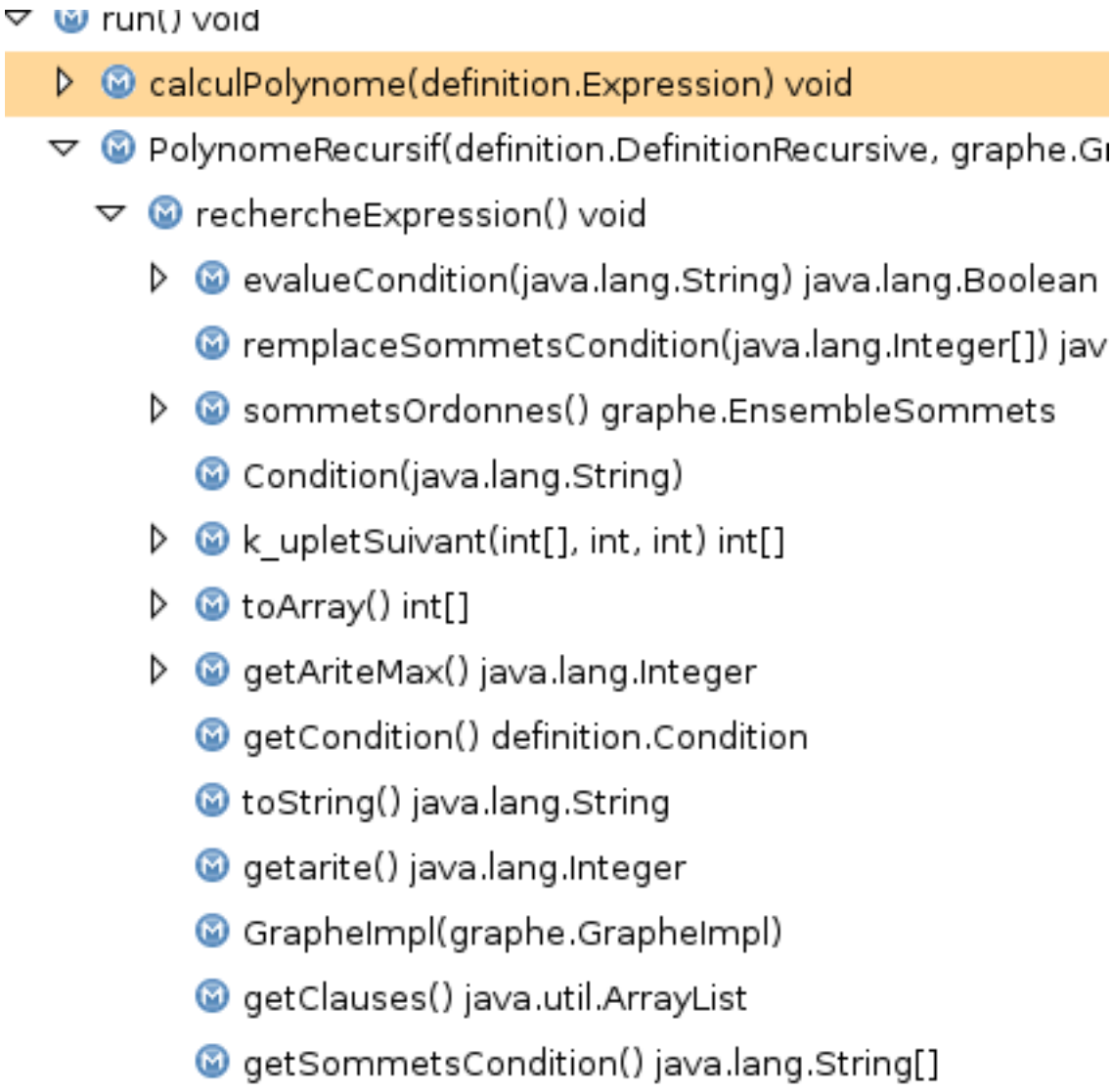


FIG. 8.2 – Synthèse des appels des méthodes de la classe **Expression**

8.4 Tests système

L'application a été testée sur plateforme linux, mac et windows.

Chapitre 9

Bilan du projet

9.1 Planning de développement

Le développement s'est déroulé en quatres phases, pour lesquelles nous avons suivi une méthode plutôt itérative.

9.1.1 Phase 1 : Briques de bases

Pour la première phase, qui a duré deux semaines, deux équipes ont été constituées. M. Nagle et M. Ramamonjisoa ont été chargés de concevoir et de coder la partie Graphe, P. Valicov et C. Delmotte pour définition des grammaires, mise en oeuvre de l'analyse syntaxique des clauses et calcul sur les polynômes. A l'issue de cette phase, les opérations élémentaires (fabriques) sur les graphes et les polynômes étaient disponibles dans une première version, la grammaire de base était fonctionnelle pour l'analyse d'une expression, les actions de calcul sur les polynômes correspondant aux étapes d'une réduction étaient effectuées.

9.1.2 Phase 2 : Mise en oeuvre de la récursion et intégration

La deuxième phase, d'une durée également de deux semaines avait pour but d'effectuer un calcul complet pour une définition *codée en dur*. Sur la base des briques réalisées, et testées, les deux mêmes équipes ont travaillé en parallèle chacune sur une des pistes évoquées (méthode itérative et récursive).

9.1.3 Phase 3 : IHM et fonctionnalités supplémentaires

Cette phase a constitué l'étape critique du projet, avec d'une part la décision de coder l'interface graphique et d'autre part l'intégration de tous les modules. Lors de cette phase, les modules *analyseur*, *graphe* et *polynome* ont connu leurs dernières évolutions.

9.1.4 Phase 4 :

Cette dernière phase a été l'intégration de l'interface puis des fonctionnalités évoluées (de priorité moindre pour le client). Elle a conduit en particulier au multi-threading de l'application.

9.2 Critique du projet

L'une des critique principale et que nous n'avons pas utilisé le serveur Savane mis à notre disposition. Dans l'ensemble les fonctionnalités demandées sont intégrées au programme. Cependant nous sommes conscients que le déroulement des calculs n'est pas optimal. La méthode de calcul choisie est d'une complexité importante comparée à la méthode présentée en annexe (Voir annexe : Présentation de la méthode itérative). Celle mise en place étant plus facile à implémenter, c'est pour cela qu'elle a été retenu afin de finir le projet dans les délais. Le temps de calcul n'étant pas une contrainte, notre méthode est encore une fois justifiée.

Au niveau interface utilisateur, les messages d'erreurs ne sont pas toujours cohérents. Cela est dû au fait que le même message d'erreur est utilisé pour différents cas.
Les tests aux limites, concernant les graphes de grande taille, n'ont pu être effectués à l'échelle que nous avions prévu, et ce par manque de temps. Pour ces tests en particulier, il aurait été judicieux de développer un outil de calcul en mode *batch*. L'applet n'a pas été inséré dans une page Web. Cela est dû au fait que les fonctions de lecture et d'enregistrement de fichiers nécessitent un certificat de sécurité, qu'il conviendra d'acquérir.

Bien que le groupe était non homogéne et a été formé par M Narbel, nous avons su travailler ensemble dans une bonne ambiance. Nous nous sommes organisés selon nos domaines de compétences afin de travailler en autonomie.

Chapitre 10

Annexes

10.1 Exemple de référence

Soit le graphe G défini par la matrice d'adjacence A_G suivante :

	1	2	3	4	5
1	0	1	1	1	1
2	1	0	0	1	1
3	1	0	0	1	0
4	1	1	1	0	0
5	1	1	0	0	0

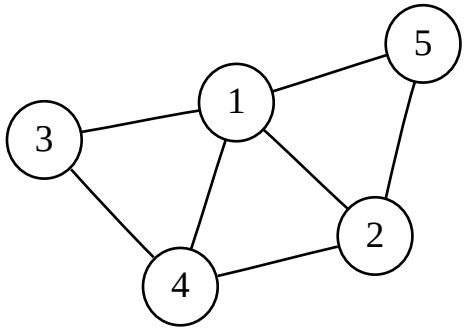


FIG. 10.1 – Graphe de l'exemple

$V_G = \{1, 2, 3, 4, 5\}$

A ce graphe sans boucle, on associe le polynôme $Q(G)$ dans $Z[x]$ défini récursivement ainsi :
$Q(G) = 1$ si $G = \emptyset$ (est vide),
$Q(G) = x * Q(G - a)$ si a est un sommet isolé,
$Q(G) = Q(G - a) + Q(p(G, a, b) - b)$ si a et b sont adjacents.

Polynôme $Q(G)$

Les sommets 1 et 2 sont adjacents. Par définition, on a donc :
$Q(G) = Q(G - 1) + Q(p(G, 1, 2) - 2)$

Appel : $Q(G-1)$

A_{G-1}

	2	3	4	5
2	0	0	1	1
3	0	0	1	0
4	1	1	0	0
5	1	0	0	0

Les sommets 2 et 5 sont adjacents. On a donc :
$Q(G-1) = Q((G-1)-2) + Q(p((G-1),2,5)-5)$
Pour simplifier l'expression, on note $G-1$ par B :
$Q(B) = Q(B-2) + Q(p(B,2,5)-5)$

Appel : $Q(B-2)$

A_{B-2}

	3	4	5
3	0	1	0
4	1	0	0
5	0	0	0

Le sommet 5 est isolé, donc :
$Q(B-2) = x * Q((B-2)-5)$
$B-2 = C$
$Q(C) = x * Q(C-5)$

Appel : $Q(C-5)$

A_{C-5}

	3	4
3	0	1
4	1	0

Les sommets 3 et 4 sont adjacents, donc :
$Q(C-5) = Q((C-5)-3) + Q(p((C-5),3,4)-4)$
Pour simplifier l'expression, on note $C-5$ par D :
$Q(D) = Q(D-3) + Q(p(D,3,4)-4)$

Appel : $Q(D-3)$

A_{D-3}

	4
4	0

Le sommet 4 est isolé, donc :
$Q(D-3) = x * Q((D-3)-4)$
$D-3 = E$
$Q(E) = x * Q(E-4)$

Appel : $Q(E-4)$

$E-4 = \emptyset$
On a donc :
$Q(E-4) = 1$

Retour : $Q(D-3)$

Nous avons donc :
$Q(E) = x * 1$
C'est-à-dire :
$Q(D-3) = x$

Appel : $Q(p(D,3,4)-4)$

$p(D,3,4)$

	3	4
3	0	1
4	1	0

$A_{p(D,3,4)-4}$

	3
3	0

Le sommet 3 est isolé, donc :
$Q(p(D,3,4)-4) = x * Q((p(D,3,4)-4)-3)$

Appel : $Q((p(D,3,4)-4)-3)$

$(p(D,3,4)-4)-3 = \emptyset$
On a donc :
$Q((p(D,3,4)-4)-3) = 1$

Retour : $Q(p(D,3,4)-4)$

Nous avons donc :
$Q(p(D,3,4)-4) = x * 1 = x$

Retour : $Q(C-5)$

$Q(C-5) = Q(D)$
$Q(D) = Q(D-3) + Q(p(D,3,4)-4)$
$Q(D) = x + x = 2 * x$

Retour : $Q(B-2)$

$Q(B-2) = Q(C)$
$Q(C) = x * Q(C-5)$
$Q(C) = x * (2 * x) = 2 * x^2$

Retour : $Q(G-1)$

$Q(G-1) = Q(B)$
$Q(B) = Q(B-2) + Q(p(B,2,5)-5)$
$Q(B) = 2 * x^2 + Q(p(B,2,5)-5)$

Appel : $Q(p(B,2,5)-5)$

$p(B,2,5)$

	2	3	4	5
2	0	0	1	1
3	0	0	1	0
4	1	1	0	0
5	1	0	0	0

$A_{p(B,2,5)-5}$

	2	3	4
2	0	0	1
3	0	0	1
4	1	1	0

Les sommets 2 et 4 sont adjacents, donc :
$Q(p(B,2,5)-5) = Q((p(B,2,5)-5)-2) + Q(p((p(B,2,5)-5),2,4)-4)$
$p(B,2,5)-5 = F$
$Q(F) = Q(F-2) + Q(p(F,2,4)-4)$

Appel : $Q(F-2)$

$Q(F-2) = Q(C-5)$
$Q(F-2) = 2 * x$

Appel : $Q(p(F,2,4)-4)$

$p(F,2,4) = F$
$Q(p(F,2,4)-4) = Q(F-4)$

$F-4$		2	3
	2	0	0
	3	0	0

Le sommet 2 est isolé, donc :
$Q(F-4) = x * Q((F-4)-2)$
Le sommet 3 est isolé.
$Q((F-4)-2) = x * Q(((F-4)-2)-3)$
Le sommet graphe est vide, donc :
$Q(F-4) = x^2$

Retour : $Q(p(B,2,5)-5)$

$Q(p(B,2,5)-5) = Q(F)$
$Q(F) = Q(F-2) + Q(p(F,2,4)-4)$
$Q(p(B,2,5)-5) = 2 * x + x^2$

Retour : $Q(G-1)$

$Q(G-1) = 2 * x^2 + 2 * x + x^2$
$Q(G-1) = 2 * x + 3 * x^2$

Appel : $Q(p(G,1,2)-2)$

$p(G,1,2)$	1	2	3	4	5
1	0	1	1	1	1
2	1	0	0	1	1
3	1	0	0	**0**	**1**
4	1	1	**0**	0	0
5	1	1	**1**	0	0

$N(G,1) = \{2,3,4,5\}$
$N(G,2) = \{1,4,5\}$
$\{1,2\} \cap \{3,4\} = \emptyset$
$3 \in N(G,1) - N(G,2)$
$4 \in N(G,2)$
$\{1,2\} \cap \{3,5\} = \emptyset$
$3 \in N(G,1) - N(G,2)$
$5 \in N(G,2)$

$p(G,1,2) - 2$

	1	3	4	5
1	0	1	1	1
3	1	0	**0**	**1**
4	1	**0**	0	0
5	1	**1**	0	0

Notons $Q(p(G,1,2)-2)$ par $Q(H)$.

Les sommets 1 et 3 sont adjacents, donc :
$Q(H) = Q(H-1) + Q(p(H,1,3)-3)$

Appel : $Q(H-1)$

Nous avons un graphe identique à C, donc :
$Q(H-1) = 2x^2$

Appel : $Q(p(H,1,3)-3)$

Nous avons un graphe complet, donc :
$Q(p(H,1,3)-3) = 4x$

Retour : $Q(H)$

$Q(H) = Q(H-1) + Q(p(H,1,3)-3)$
$Q(H) = (2x^2 + 4x)$

retour : $Q(G)$

$Q(G) = Q(G-1) + Q(p(G,1,2)-2)$
$Q(G) = (2 * x + 3 * x^2) + (2x^2 + 4x)$
Nous avons donc finalement :
$Q(G) = 5 * x^2 + 6x$

10.2 Présentation de la méthode de calcul itérative

Au travers d'un exemple trivial, nous présentons le déroulement du calcul itératif du polynôme à partir de sa définition récursif et d'un graphe.

Nous considérons donc le graphe suivant ainsi que la définition récursive suivante :

$G : A_G$

	1	2	3
1	0	1	1
2	1	0	0
3	1	0	0

$V_G = \{1,2,3,4,5\}$

$Q(G) = 1$ si $G = \emptyset$ (est vide),
$Q(G) = x * Q(G-a)$ si a est un sommet isolé,
$Q(G) = Q(G-a) + Q(p(G,a,b)-b)$ si a et b sont adjacents.

Initialisation du calcul On détermine en premier lieu la clause dont la condition est vérifiée. En l'occurrence, il s'agit ici de la troisième dont la condition est vérifiée par les sommets 1 et 2 adjacents. On associe donc respectivement aux indéterminées a et b les sommets 1 et 2 :

a	b
1	2

L'expression de cette troisième clause décomposée sous forme d'une liste d'expressions (expression terminale ou opération sur un graphe), nous donne après évaluation des expressions partielles dont les graphes sont systématiquement parenthésés :

$G-a$	$+$	$p(G,a,b)$	$G-b$

$G-1$	$\Rightarrow$	(	$G1$	)
$p(G,1,2)$	$G-2$	$\Rightarrow$	(	$G2$	)
---	---	---	---	---	---

Nous obtenons donc l'expression initiale suivante à l'aide des expressions partielles :

(	$G1$	)	$+$	(	$G2$	)

Itération L'itération consiste à partir de l'expression initiale, à substituer tout graphe par une expression partielle si une clause s'applique pour ce graphe.

La deuxième clause s'applique pour le graphe $G1$ puisque le sommet 2 est isolé. Nous avons donc :

x	$*$	$G-a$

a
2

$G1-2$	$\Rightarrow$	(	$G3$	)

Ainsi, dans notre exemple, nous avons l'expression initiale suivante :

(	$G1$	)	$+$	(	$G2$	)

Nous obtenons donc après substitution l'expression suivante :

(	x	$*$	(	$G3$	)	)	$+$	(	$G2$	)

De même, pour $G2$, la troisième clause s'applique puisque les sommets 1 et 3 sont adjacents, d'où :

a	b
1	3

$G-a$	$+$	$p(G,a,b)$	$G-b$

$G2-1$	$\Rightarrow$	(	$G4$	)
$p(G2,1,3)$	$G2-3$	$\Rightarrow$	(	$G5$	)
---	---	---	---	---	---

Nous obtenons donc après substitution une nouvelle expression :

(	x	$*$	(	$G3$	)	)	$+$	(	(	$G4$	)	$+$	(	$G5$	)	)

De là, pour le graphe $G3$ s'applique la deuxième clause puisque le sommet 2 est isolé. D'où après substitution de $G3$ l'expression suivante :

(	x	$*$	(	x	$*$	(	$G6$	)	)	)	$+$	(	(	$G4$	)	$+$	(	$G5$	)	)

De même, la deuxième clause s'applique pour les graphes $G4$ et $G5$, d'où l'expression :

(	x	$*$	(	x	$*$	(	$G6$	)	)	)	$+$	(	(	x	$*$	(	$G7$	)	)	$+$	(	x	$*$	(	$G8$	)	)	)

Fin de l'itération Dans la précédente expression, les graphes $G6$, $G7$ et $G8$ sont vides. La première clause de la définition récursive ayant 1 pour expression s'applique donc. Nous obtenons ainsi l'expression suivante :

(	x	$*$	(	x	$*$	(	1	)	)	)	$+$	(	(	x	$*$	(	1	)	)	$+$	(	x	$*$	(	1	)	)	)

L'expression, n'ayant plus de graphes en son sein, nous avons le polynôme résultat :

$$(x*(x*(1)))+((x*(1))+(x*(1)))$$

Après factorisation, nous obtenons $2*x+x^2$, soit le résultat attendu pour un graphe en étoile à trois sommets et la définition récursive donnée.

10.3 Manuel utilisateur

Manuel d'utilisation

Générer un graphe
Générer une définition récursive
Calcul
Opérations sur les résultats
Chargement de fichiers
Enregistrement de fichiers
Format des fichiers graphes
Format des fichiers définitions
Format des fichiers résultats

Générer un graphe

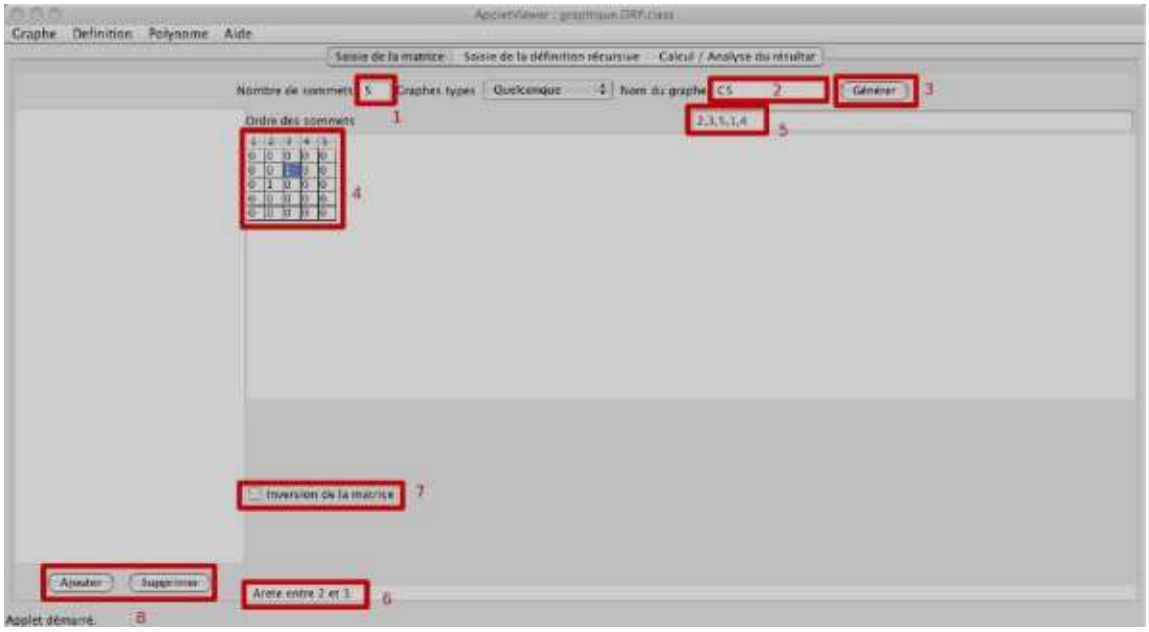


FIG. 10.2 – Saisie du graphe

Démarche à suivre :

1. Saisir le nombre de sommets souhaité,
2. Saisir un nom pour le graphe,
3. Appuyez sur générer,
4. La matrice d'adjacence s'affiche dans la panneau,
5. Saisir un ordre de sommets particulier. Si rien n'est saisie l'ordre par défaut est l'ordre lexicographique.

Il est aussi possible de choisir un type de graphe prédéfini :

1. Saisir le nombre de sommets souhaité,
2. Choisir dans le menu déroulant. Il faut choisir entre :
   - Complet,
   - Etoile : le centre sera par défaut le sommet 1. Pour changer il faudra changer l'ordre des sommets,
   - Bipartie-Complet : la taille d'une des parties du graphes vous sera demandée. L'autre sera automatiquement calculée à partir du nombre de sommets,
   - Cycle.
3. Saisir un nom pour le graphe,
4. Appuyer sur générer,
5. La matrice d'adjacence s'affiche dans la panneau,
6. Saisir un ordre de sommets particulier. Si rien n'est saisie l'ordre par défaut est l'ordre lexicographique.

Pour changer les valeurs dans la matrice il suffit de double cliquer sur la case souhaitée. Les coordonnées de la case sont affichées en dessous (6). Vous avez aussi la possibilité de saisir la matrice inverse puis de l'inverser si c'est plus pratique (7). Si vous souhaitez garder le graphe pour de futurs calculs, vous pouvez l'ajouter à la

liste sur la gauche en cliquant sur Ajouter. Cliquer sur Supprimer pour l'enlever (8).

Haut de page

Générer une définition récursive

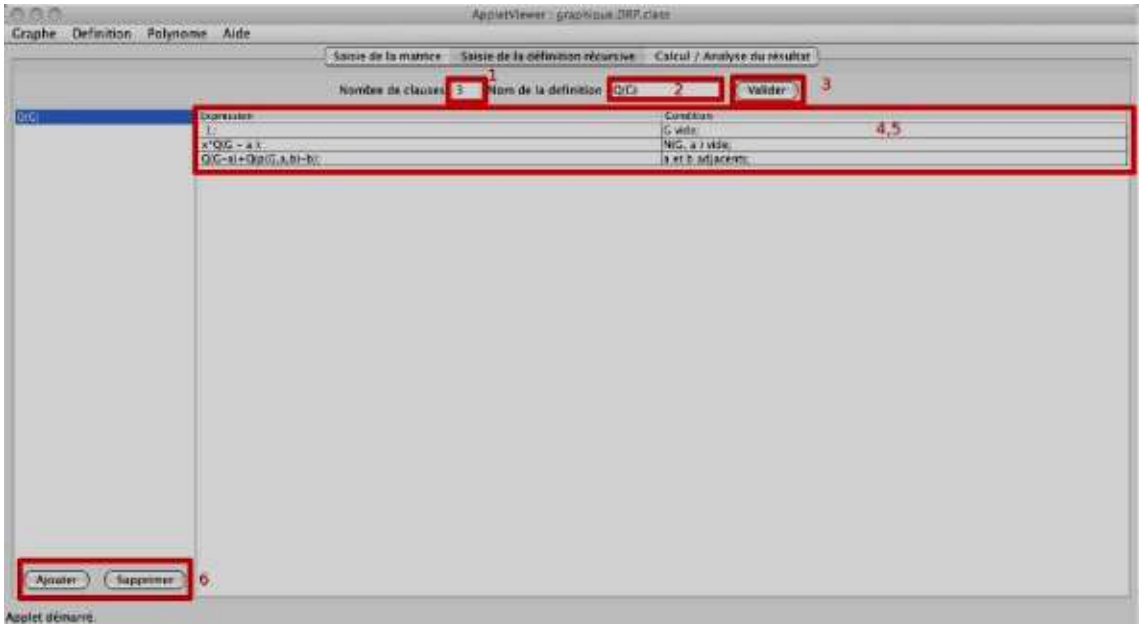


FIG. 10.3 – Saisie de la définition récursive

Démarche à suivre :

1. Saisir le nombre de clauses de la définition récursives,
2. Saisir le nom de la définition récursive,
3. Appuyez sur générer,
4. Le tableau des définitions récursives s'affiche,
5. Saisir les clauses.

Si vous souhaitez garder la définition récursive pour de futurs calculs, vous pouvez l'ajouter à la liste sur la gauche en cliquant sur Ajouter. Cliquer sur Supprimer pour l'enlever(6).

Syntaxe des clauses :

Les expressions et les conditions se terminent toujours par un ;.

- Les expressions
  Les sommets et les variables sont représentés par des lettres minuscules. Les majuscules sont réservées pour les noms de polynômes sauf G pour le graphe et N pour le voisinage.

  - Sommet
    Les sommets sont des lettres de l'aphabet latin de a à s, sauf c et p, uniquement en minuscule.

- Variable
  - Les variables sont des lettres de l'alphabet latin de t à z, uniquement en minuscule,
  - x - une variable ordinaire,
  - x_a - la variable x associée au sommet a (xa).
- Ensemble de sommets Un ensemble X peut être :
  - N(G,a) - le voisinage du sommet a,
  - a,b,c,....

- Opération élémentaire sur les graphes
  - G-a - l'opération de suppression du sommet a,
  - G-X - l'opération de suppression de l'ensemble de sommets X,
  - G[X] - graphe induit par l'ensemble de sommets X,
  - c(G) - l'opération de complémentation des arêtes sans toucher aux boucles,
  - lc(G,a) - l'opération de complémentation locale,
  - inv(G,X) - l'opération de complémentation des boucles des sommets de X . X est un ensemble,
  - bc(G,a) - l'opération de complémentation locale étendue aux boucles,
  - p(G,a,b) - l'opération de pivotage sur (a, b).

- Autres opérations
  - rk(G) - l'opération de calcul du rang du graphe,
  - cc(G) - l'opération de calcul du nombre de composantes connexes dans le graphe,
  - ^ - l'exposant.

Une expression élémentaire peut être une combinaison d'opération. Une expression est donc un composition d'expression élémentaire.
Exemple d'expression : Q(G-a)+Q(p(G,a,b)-b) ;

- Les conditions élémentaires
  - a et b adjacents ; - pour la condition a et b sont adjacents,
  - G vide ; - pour la condition que le graphe est vide,
  - N (G, a) vide ; - pour la condition que a est un sommet isolé,
  - a incident boucle ; - pour la condition que a est incident à une boucle,
  - non(condition) ; - la négation de la condition élémentaire.

Une condition est une composition de conditions élémentaires reliées par une opérateur booléen.
- (condition élémentaire) et (condition élémentaire)
- (condition élémentaire) ou (condition élémentaire)
- non (condition élémentaire)

Exemple de condition : a et b adjacents ;

Haut de page

Calcul

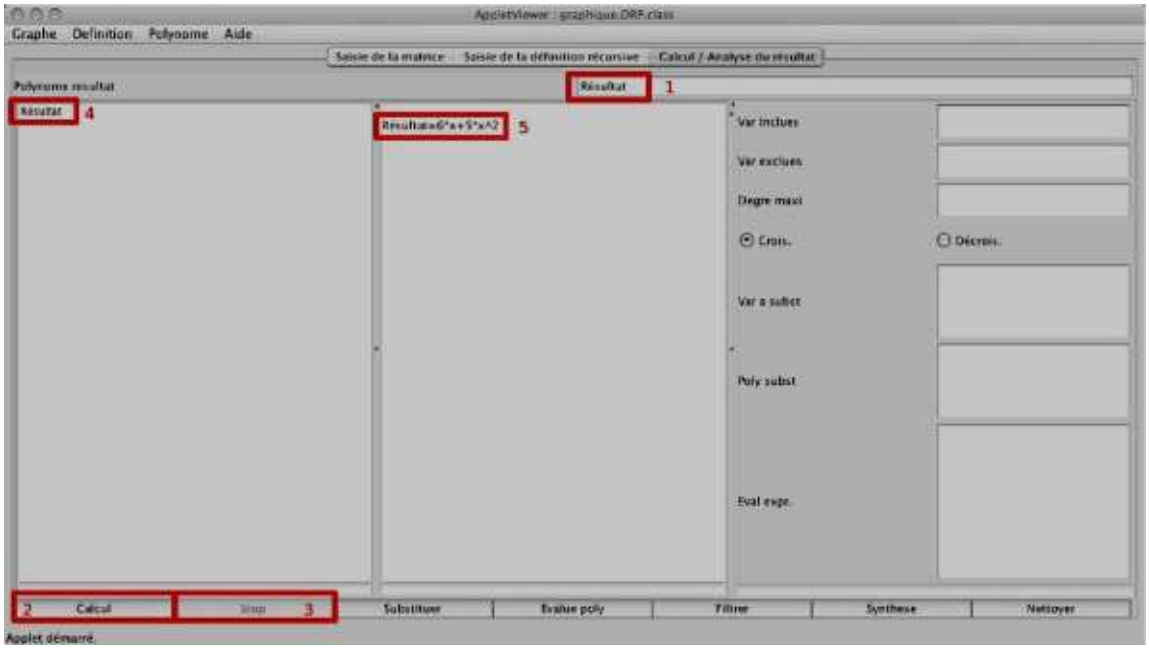


FIG. 10.4 – Démarche pour le calcul

Une fois le graphe, la définition et un nom pour le polynôme résultat saisis (1) vous pouvez lancer le calcul à l'aide du bouton lancer le calcul (2). Si vous trouvez que le calcul dure trop longtemps vous pouvez l'arrêter à l'aide du bouton Arreter le calcul (3). Le résulat est automatiquement ajouté à la liste (4). Tous les résultats restent aussi affichés au centre (5).
Si le nom que vous avez choisi est déjà dans la lsite, une fenêtre s'ouvira pour vous demandez si vous souhaitez ou non effacer l'ancien résultat. Pour effacer cliquez sur OK sinon sur Annuler.

Haut de page

Opérations sur les résultats

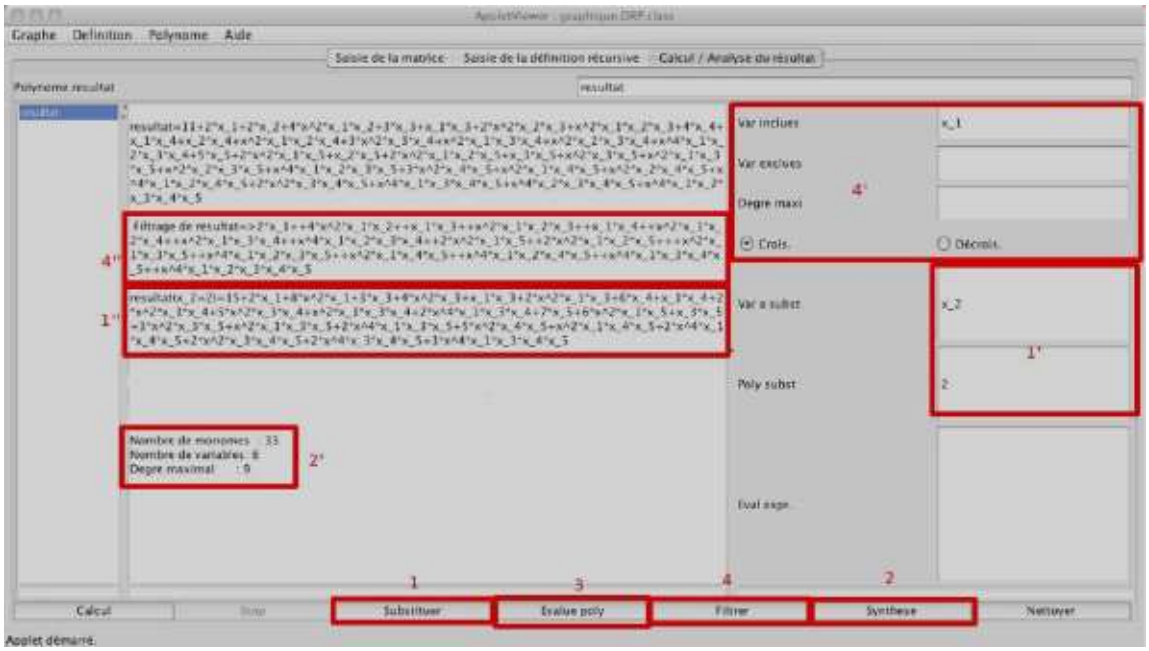


FIG. 10.5 – Opérations sur les résultats

Différentes opérations :

- Substitution : Il faut saisir les variables à substituer dans la case Var a subst et donner les nouvelles variables dans Poly subst (1'). Celles ci peuvent être aussi des chiffres ou des polynômes. Il est possible de substituer plusieurs variables en les séparant avec des virgules. Le résultat s'affiche dans le panneau des résultats (1"),
- Synthése : Elle permet de connaître le nombre de monômes, le nombre de variables et le degré maximum (2'). Ceci pour le polynôme selectionné dans la liste,
- Evalue poly : permet d'effectuer des opérations (addition, multiplication, soustraction) sur des polynômes arithmétiques et sur des polynômes résultats Exemple : resultat - expression avec expression un polynôme résultat ou un polynôme arithmétique (3'). Le résultat s'affiche dans le panneau principale.
- Filtrer : filtrer l'affichage des polynômes résultats (4') :
  - Choisir les variables à afficher. A saisir dans Var inclues,
  - Choisir les variables à exclure. A saisir dans Var exlues,
  - Choisir le degré maximum des variables à afficher ou à exclure,
  - Choisir l'ordre d'affichage, croissant ou décroissant,
  - Le résultat du filtre s'affiche dans le panneau des résultats (4").

Si le nom que vous avez choisi est déjà dans la lsite, une fenêtre s'ouvira pour vous demandez si vous souhaitez ou non effacer l'ancien résultat. Pour effacer cliquez sur OK sinon sur Annuler.

Haut de page

Chargement de fichiers

Il est possible de charger des graphes, des définitions récursives et des polynômes.

- Chargement d'un graphe
  1. Aller dans le menu Graphe,

2. Choisir lire graphe,
3. Récupération et affichage du nombre de sommets,
4. Récupération et affichage de l'ordre,
5. Récupération et affichage du nom,
6. Récupération et affichage de la matrice correspondant au graphe enregistré.

– Chargement d'une définition récursive
1. Aller dans le menu Definition,
2. Choisir lire définition,
3. Récupération et affichage du nombre de clauses,
4. Récupération et affichage du nom,
5. Récupération et affichage des clauses enregistrées.

– Chargement d'un résultat
1. Aller dans le menu Polynome,
2. Choisir lire polynome resultat,
3. Récupération et affichage du résultat,
4. Récupération et affichage du nom.

Haut de page

Enregistrement de fichiers

Il est possible de sauvegarder des graphes, des définitions récursives et des polynômes.

– Sauvegarde d'un graphe
1. Aller dans le menu Graphe,
2. Choisir Enregistrer graphe,
3. Choisir le nom du fichier et écrire l'extension .txt.

– Sauvegarde d'une définition récursive
1. Aller dans le menu Definition,
2. Choisir Enregistrer definition,
3. Choisir le nom du fichier et écrire l'extension .txt.

– Sauvegarde d'un résultat
1. Aller dans le menu Polynome,
2. Choisir Enregistrer polynome resultat,
3. Choisir le nom du fichier et écrire l'extension .txt.

Haut de page

Format des fichiers graphes

Il est possible de modifier directement les fichiers à la main pour cela voici le format utilisé. Voici un exemple :
commentaire

/nom du graphe
$\{1, 2, 3, 4, 5\}$ Numéros des sommets ici graphe de 5 sommets.
[
[0 1 0 0 1]
[1 0 1 0 0]
[0 1 0 1 0] Matrice d'adjacence saisie dans l'interface.
[0 0 1 0 1]
[1 0 0 1 0]
] ;

Haut de page

Format des fichiers définitions

Pour modifier le fichier des définitions récursives voici un exemple de fichier :

/Q(G)	Nom de la définition
= 1 ;	= Expression ;
si G vide ;	si Condition ;
= x*Q(G - a) ;	
si N(G, a) vide ;	
= Q(G-a)+Q(p(G,a,b)-b) ;	
si a et b adjacents ;	

Haut de page

Format des fichiers résultats

<graphe>	Début de la partie graphe
/C	Voir Format des fichiers graphes
$\{1, 2, 3, 4, 5, 6, 7, 8\}$	
[	
[0 1 1 1 1 1 1 1]	
[1 0 1 1 1 1 1 1]	
[1 1 0 1 1 1 1 1]	
[1 1 1 0 1 1 1 1]	
[1 1 1 1 0 1 1 1]	
[1 1 1 1 1 0 1 1]	
[1 1 1 1 1 1 0 0]	
[1 1 1 1 1 1 0 0]	
] ;	
</graphe>	Fin de la partie graphe
<definition>	Début de la partie définition récursive
/Standard	Voir Format des fichiers définitions
= 1 ;	
si G vide ;	
= x*Q(G - a) ;	
si N(G, a) vide ;	
= Q(G-a)+Q(p(G,a,b)-b) ;	
si a et b adjacents ;	
</definition>	Fin de la partie définition
<resultat>	Début du résultat
$P(C) = 64 * x + 32 * x^2$	
</resultat>	Fin du résultat

Haut de page

10.4 Codes sources

Bibliographie

[BER06] Olivier BERNARDI. *Combinatoire des cartes et polynôme de Tutte*. PhD thesis, Université de Bordeaux 1, 2006.

[COU07] Bruno COURCELLE. A multivariate interlace polynomial. *Preprint arXiv cs/0702016*, Janvier 2007. Available at http ://front.math.ucdavis.edu/0702.6016.

[eLD46] BIRKHOFF G.D. et LEWIS D. Chromatic polynomials. *Transactions of the American Mathematical Society*, 60 :355–451, November 1946.

[SOK05] A. SOKAL. The multivariate tutte polynomial (alias potts model) for graphs and matroids. *London Math. Soc. Lec.*, 327 :173–226, Mars 2005. Surveys in Combinatorics.

[SOR04] Richard ARRATIA Bela BOLLOBAS Gregory B. SORKIN. A two-variable interlace polynomial. *Journal of Combinatorial Theory*, Series B 92 :199–233, Septembre 2004.