Logiciel d'investigation de codes

KARP Nicolas, HARIVEL Christophe, MIGNOT Ludovic

Licence Informatique $3^{ème}$ année

Responsable: Jean NERAUD

18 mai 2005

Table des matières

1 Partie Théorie **4**
1.1 Définitions . . . 4
1.1.1 Combinatoire des mots . . . 4
1.1.2 Graphe . . . 5
1.1.3 Graphe de Spehner . . . 6

2 Description des méthodes contenues dans les packages utilisés :Package Langage **6**
2.1 Alphabet . . . 6
2.1.1 Introduction . . . 6
2.1.2 Méthode de construction . . . 7
2.1.3 Requêtes . . . 7
2.1.4 Commandes . . . 7
2.1.5 Exemple d'utilisation . . . 8
2.2 Mot . . . 8
2.2.1 Introduction . . . 8
2.2.2 Méthodes de construction . . . 9
2.2.3 Requêtes . . . 9
2.2.4 Commandes . . . 10
2.2.5 Exemple d'utilisation . . . 10
2.3 Langage . . . 10
2.3.1 Introduction . . . 10
2.3.2 Méthode de construction . . . 11
2.3.3 Requêtes . . . 11
2.3.4 Commandes . . . 12
2.3.5 Exemple d'utilisation . . . 12

3 Description des méthodes contenues dans les packages utilisés :Package Graphe **14**
3.1 Introduction . . . 14
3.2 Sommet . . . 14
3.2.1 Construction . . . 14
3.2.2 Requêtes . . . 14
3.3 Arc . . . 14
3.3.1 Construction . . . 14
3.3.2 Requêtes . . . 15
3.4 Etiquette-Couple . . . 15
3.4.1 Construction . . . 15
3.4.2 Requêtes . . . 15
3.5 Graphe . . . 15
3.5.1 Construction . . . 15
3.5.2 Requêtes . . . 15
3.5.3 Commandes . . . 16
3.6 Graphe de Spehner . . . 16
3.6.1 Construction . . . 16
3.7 Matrice d'adjacence . . . 16
3.7.1 Construction . . . 16
3.7.2 Requêtes . . . 16
3.8 Exemple de construction d'un graphe . . . 17

4 Description des algorithmes utilisés **20**
4.1 Graphe de Spehner . . . 20
4.1.1 Construction du graphe de Spehner . . . 20
4.2 Matrice d'adjacence . . . 21
4.2.1 Existence d'un arc . . . 21
4.2.2 Présence d'un sommet sur un circuit quelconque . . . 21
4.2.3 Recherche d'arcs particuliers :arcs sortant d'un sommet . . . 21
4.2.4 Recherche d'arcs particuliers :arcs rentrant vers un sommet . . . 22
4.2.5 Recherche d'arcs particuliers :arc reliant deux sommets . . . 22
4.2.6 Recherche d'arcs particuliers :ensemble d'arcs reliant deux sommets . . . 22
4.2.7 Existence d'un chemin entre deux sommets . . . 22
4.2.8 Recherche et Obtention d'un circuit . . . 22
4.2.9 Recherche de circuit contenant un sommet . . . 23
4.2.10 Recherche de sommets sur un même circuit qu'un sommet et accessibles par un arc depuis ce sommet . . . 24
4.2.11 Test d'accessibilité de sommets . . . 24
4.2.12 Test de coAccessibilité de sommets . . . 24
4.2.13 Test : sommet isolé . . . 24
4.3 Langage . . . 24
4.3.1 Génération de mots . . . 24
4.3.2 Découpage en facteurs particuliers : préfixes et suffixes . . . 25
4.3.3 Test et obtention d'un système générateur libre . . . 26

5 Manuel Utilisateur **26**
5.1 Création de l'alphabet . . . 27
5.2 Création du langage . . . 28
5.3 Affichage du graphe de Spehner et utilisation des différents outils proposés. . . . 30

Conclusion **33**

Références bibliographiques **34**

1 Partie Théorie

Dans cette partie nous allons rappeler les définitions de bases, et expliquer comment fonctionne le graphe de Spehner.

1.1 Définitions

1.1.1 Combinatoire des mots

- Alphabet :

  L'alphabet est l'ensemble de base sur lequel nous pouvons créer des mots. Il est constitué de lettres, c'est à dire de mots de longueur 1. En combinant ces lettres, nous pouvons donc former des mots. Nous verrons les mots et les langages (ensemble de mots) par la suite.

- Langage :

  Un langage est un ensemble de mots construit sur un alphabet particulier. On ne peut ajouter un mot à un langage que s'il peut être construit comme une combinaison des lettres de cet alphabet.

- Monoïde :

  On dit que (E,T) est un monoïde si T est une loi de composition interne pour l'ensemble E qui est associative :

  Pour tous a,b,c de E : (aTb)Tc=aT(bTc)

  La structure de monoïde est particulièrement utilisée en informatique, lorsqu'il s'agit de théorie des langages. L'ensemble des mots sur un alphabet, muni de l'opération de concaténation de deux mots, constitue en effet un monoïde.

- Morphisme :

  Soit f une application de G dans G' ; on dit que f est un (homo)morphisme de groupes si, pour tous x et y de G, on a :

  f(x.y)=f(x).f(y)

- Préfixe (resp. suffixe) propre :

  Un préfixe (resp. suffixe) est dit propre s'il est différent du mot lui-même.

- Code :

  Un langage est un code si chaque mot qu'il engendre peut se décomposer de manière unique sur ce langage.

- Code préfixe :

  Un code est dit préfixe s'il respecte la "condition du préfixe" :
  Aucun mot de code préfixe d'un autre.

- Code suffixe :

  Un code est dit suffixe s'il respecte la "condition du suffixe" :
  Aucun mot de code n'est suffixe d'un autre.

- Code biffixe :

  C'est un code qui est à la fois préfixe et suffixe.

1.1.2 Graphe

- Graphe :

  Un graphe G est un couple (V,E) où :
  - V est un ensemble (fini) d'objets. Les éléments de V sont appelés les sommets du graphe.
  - E est sous-ensemble de V×V. Les éléments de E sont appelés les arêtes du graphe.

  Une arête "e" du graphe est une paire e=(x,y) de sommets. Les sommets x et y sont les extrémités de l'arête.

- Graphe connexe :

  Un graphe non orienté est connexe, si et seulement si pour toute paire de sommets [a,b] il existe une chaîne entre les sommets a et b. Si on parle de connexité pour un graphe orienté, c'est que l'on considère non pas ce graphe, mais le graphe non-orienté correspondant.

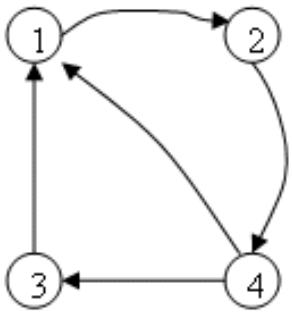
- Graphe fortement connexe :

  Un graphe orienté est dit fortement connexe, si pour tout couple de sommets (u,v) du graphe il existe un chemin de u à v et de v à u.

- Matrice d'adjacence :

  Soient G un graphe, A une matrice, et N l'ensemble des noeuds du graphe G.
  Quelques-soient "i" et "j" appartenant à N, si pour toutes arêtes de "i" vers "j" du graphe G, A(i, j) = 1 . Et dans tous les autres cas A(i,j)=0 alors A est la matrice d'adjacence du graphe G.

  Exemple d'un graphe et de sa matrice d'adjacence :



Graphe G

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

Matrice d'adjacence de G

1.1.3 Graphe de Spehner

Comment ça marche ?

Le graphe de *Spehner* est établi à partir d'un langage X donné qui lui même est établi à partir d'un alphabet "A". De ce langage, on extrait de chaque mot tous les préfixes étants suffixes d'un autre mot. Ainsi nous obtenons un nouvel ensemble "K" auquel nous ajoutons le mot *vide* ϵ.
Chaque élément de cet ensemble constitura un noeud de notre graphe. Il nous faut à présent trouver l'ensemble "C" des arcs correspondant à ce graphe.

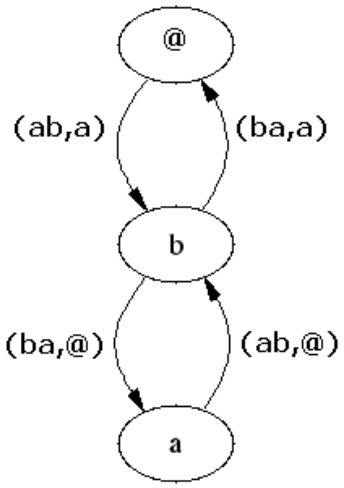
Chaque arc (u,v) partant du noeud "u" vers le noeud "v", sera étiqueté par un couple (x,y) avec x∈X et y∈X*. Pour déterminer l'ensemble "C" ainsi que les étiquettes correspondantes à chacun des arcs, on applique la règle suivante :

$$\text{si} \quad x=u.y.v \quad \text{avec} : \begin{cases} - \ x \in X \text{ et } y \in X^* \\ - \ u.y \neq \epsilon \text{ et } y.v \neq \epsilon \\ - \ u,v \in K \end{cases}$$

alors l'arc (u,v) sera ajouté au graphe avec l'étiquette (x,y).

Exemple :

Voici un exemple de construction du graphe de Spehner à partir du langage L={a,ab,ba}



On constate qu'il existe un chemin de ϵ(symbolisé par @ sur le graphe) vers ϵ, on en déduit que le langage n'est pas un code.
En effet, il existe le mot "aba" qui possède une double décomposition sur L :

$$a \ . \ ba = ab \ . \ a$$

2 Description des méthodes contenues dans les packages utilisés :Package Langage

2.1 Alphabet

2.1.1 Introduction

L'alphabet est l'ensemble de base sur lequel nous pouvons créer des mots. Il est constitué de lettres, c'est à dire de mots de longueur 1. En combinant ces lettres, nous pouvons donc former des mots. Nous

verrons les mots et les langages [1] par la suite.

Les méthodes existantes pour cette classe, AlphabetImpl, implantant la notion d'alphabet, permettent d'implanter les opérations de base des alphabets.

2.1.2 Méthode de construction

public AlphabetImpl()
permet de créer un alphabet vide, c'est à dire ne contenant aucune lettre. Pour l'utiliser, il suffit d'ajouter des lettres.

2.1.3 Requêtes

public Set getAlphabet()
permet d'obtenir l'ensemble des lettres constituant l'alphabet, retourné sous la forme d'un ensemble.

public boolean contains(String lettre)
permet de tester si l'alphabet possède une lettre particulière, la chaîne lettre donnée en argument.

public boolean engendre(Mot mot)
permet de tester si l'alphabet, en combinant les lettres qu'il contient, peut engendrer le mot passé en argument. Il suffit de découper le mot en facteur de longueur 1, puis de tester pour chacun s'il est contenu dans l'alphabet.

public String toString()
permet d'obtenir la chaîne de caractères décrivant l'alphabet, sous la forme { a, b} pour un ensemble contenant a et b.

public boolean equals (Alphabet a)
permet de tester l'égalité de deux alphabets, c'est à dire si les deux alphabets possèdent les mêmes lettres.

public Alphabet union(Alphabet a)
renvoie l'alphabet formé par les lettres de l'objet et celles du paramètre, c'est à dire l'union des deux ensembles de lettres.

public Alphabet cloner()
renvoie un clone de l'alphabet, c'est-à-dire un ensemble contenant les mêmes lettres.

2.1.4 Commandes

public void addLettre(String lettre)
permet d'ajouter une lettre à l'alphabet, la lettre lettre passée en paramètre.

public void removeLettre(String lettre)
permet de retirer une lettre de l'alphabet s'il la contient.

[1] ensemble de mots

2.1.5 Exemple d'utilisation

1. Création de deux alphabets vides
   Alphabet a1=new AlphabetImpl() ; \\a1=∅
   Alphabet a2=new AlphabetImpl() ; \\a2=∅

2. Ajout de lettres
   a1.addLettre(a) ; \\a1={a}
   a1.addLettre(b) ; \\a1={a,b}
   a2.addLettre(a) ; \\a2={a}
   a2.addLettre(b) ; \\a2={a,b}
   a2.addLettre(c) ; \\a2={a,b,c}
   a2.addLettre(d) ; \\a2={a,b,c,d}

3. Élimination de lettres
   a2.removeLettre(c) ; \\a2={a,b,d}

4. Clonage d'un alphabet
   Alphabet a3=new AlphabetImpl() ; \\a3=∅
   a3=a2.cloner() ; \\a3={a,b,d}

5. Union de 2 alphabets
   Alphabet a4=new AlphabetImpl() ; \\a4=∅
   a4=a1.union(a2) ; \\a4={a,b,d}

6. Test de contenance
   a4.contains(a) ; \\true car a4 contient a
   a4.contains(c) ; \\false car a4 ne contient pas c

7. Test d'engendrement
   Mot mot=new MotImpl("abda") ;[2]
   a1.engendre(mot) ; \\false car a1 ne contient pas d
   a2.engendre(mot) ; \\true car a2 contient toutes les lettres pour former abda

8. Test d'égalité
   a1.equals(a2) ; \\false car a1 ne contient pas d
   a3.equals(a2) ; \\true car a3 est un clone de a2

2.2 Mot

2.2.1 Introduction

Un mot est une suite de lettre. On peut ensuite les regrouper dans un langage, dont les propriétés seront évoquées plus loin.

Des opérations de bases sont nécessaires pour pouvoir manipuler ces objets.

[2] voir création des mots ci-après

2.2.2 Méthodes de construction

public MotImpl()
permet de construire le mot vide ϵ [3].

public MotImpl(String s)
permet de construire le mot représenté par la chaîne passée en argument.

2.2.3 Requêtes

public Set getPréfixe()
permet d'obtenir l'ensemble des préfixes de ce mot. Un préfixe correspond à une sous-chaine de la chaîne formée par les lettres du mot, commençant par la première lettre du mot, formée par les lettres consécutives[4]. Sa taille est donc inférieure ou égale a celle du mot originel.

public Set getSuffixe()
permet d'obtenir l'ensemble des suffixes de ce mot. Un suffixe correspond à une sous-chaine de la chaîne formée par les lettres du mot, se finissant par la dernière lettre du mot, formée par les lettres consécutives[5]. Sa taille est donc inférieure ou égale a celle du mot originel.

public Set getPréfixeProp()
retourne le même ensemble que getPrefixe() mais ne contenant pas le mot propre.

public Set getSuffixeProp()
retourne le même ensemble que getSuffixe() mais ne contenant pas le mot propre.

public String getMot()
permet d'obtenir la chaîne de caractère décrivant le mot.

public String toString()
permet d'obtenir la chaîne de caractère décrivant le mot.

public int compareTo(Object o)
permet de comparer deux mots selon l'ordre lexicographique. Cette fonction renvoie -1 si o est plus grand que le mot, 1 si o est plus petit et 0 si les deux mots sont égaux.

public boolean equals(Object o)
teste si deux mots sont identiques, c'est à dire s'ils sont caractérisés par la même suite de lettre.

public boolean commencePar(Mot m)
permet de tester si le mot argument est un préfixe du mot.

public boolean terminePar(Mot m)
permet de tester si le mot argument est un suffixe du mot.

[3] dont la représentation dans le programme est @
[4] exemple : pour abba, l'ensemble de préfixe est {a,ab,abb,abba }
[5] exemple : pour abba, l'ensemble de suffixe est {a,ba,bba,abba }

public Mot concat(Mot m)
retourne la concaténation du mot avec l'argument m. La concaténation d'un mot x quelconque avec le mot vide est égale à x. [6]

2.2.4 Commandes

public void retirerPref(Mot m)
découpe le mot en lui ôtant le préfixe passé en argument.

public void retirerSuf(Mot m)
découpe le mot en lui ôtant le suffixe passé en argument.

2.2.5 Exemple d'utilisation

1. création de mots
   Mot m1=new MotImpl() ; \\création du mot vide représenté par @
   Mot m2=new MotImpl(" abba ") ; \\création du mot abba
2. Obtention des préfixes et suffixes
   m2.getPréfixe() ; \\retourne l'ensemble de mots {@, a, ab, abb, abba}
   m2.getPréfixeProp() ; \\retourne l'ensemble de mots {@, a, ab, abb}
   m2.getSuffixe() ; \\retourne l'ensemble de mots {@, a, ba, bba, abba}
   m2.getSuffixeProp() ; \\retourne l'ensemble de mots {@, a, ba, bba}
3. Comparaison de mots
   Mot m3=new MotImpl("abba") ;
   Mot m4=new MotImpl("abc") ;
   m3.compareTo(m4) ; \\renvoie -1 car m3<m4
   m4.compareTo(m3) ; \\renvoie 1, car m4>m3
   m3.compareTo(m2) ; \\renvoie 0 car égalité
   m3.equals(m2) ;\\true
   m3.equals(m4) ;\\false
4. Test des facteurs Mot m5=new MotImpl("ab") ;
   Mot m6=new MotImpl("ba") ;
   m3.commencePar(m5) ; \\true
   m3.commencePar(m4) ; \\false
   m3.terminePar(m4) ; \\false
   m3.terminePar(m6) ; \\true
5. Concaténation
   Mot m7=M5.concat(m6) ; \\m7 devient le mot abba
6. Découpage Mot m8=new MotImpl("abbaab") ; m8.retirerPref(new MotImpl("abba")) ; \\m8 devient ab m8.retirerSuf("new MotImpl("ab")) ; \\m8 devient le mot vide

2.3 Langage

2.3.1 Introduction

Un langage est un ensemble de mots construit sur un alphabet particulier. On ne peut ajouter un mot à un langage que s'il peut être construit comme une combinaison des lettres de cet alphabet.

Les opérations supportées dans notre implantation sont les suivantes.

[6] x.ϵ=ϵ.x=x

2.3.2 Méthode de construction

public LangageImpl(Alphabet a)
permet de construire un langage vide, ne contenant aucun mot, avec l'argument comme alphabet.

2.3.3 Requêtes

public Alphabet getAlphabet()
permet d'obtenir l'alphabet engendrant ce langage.

public Set getLangage()
retourne l'ensemble des mots formant le langage.

public boolean contains(Mot mot)
teste si le langage contient le mot passé en argument.

public String toString()
renvoie la chaîne de caractères décrivant ce langage.

public Langage suffixe()
crée et retourne le langage formé par les suffixes des mots de ce langage. [7]

public Langage préfixe()
crée et retourne le langage formé par les préfixes des mots de ce langage.

public Langage intersection(Langage l2)
réalise l'intersection du langage avec l'argument, c'est-à-dire que le langage ne contient plus que les mots appartenant aux deux langages. Si un mot est présent dans cet ensemble, c'est qu'il est présent dans le langage objet utilisé et dans le langage l2.

public Langage prefEtSuff()
renvoie le langage qui ne contient alors que les mots à la fois préfixes et suffixes des mots de ce langage (mais non préfixe propre et suffixe propre à la fois).Un mot est dans cet ensemble s'il est soit préfixe d'un mot et suffixe d'un autre, ou alors préfixe propre et suffixe propre du même mot.

public boolean engendre(Mot m)
teste si le mot argument peut être obtenu par combinaison des mots de ce langage.[8]

public boolean isEmpty()
teste si le langage est vide, c'est-à-dire s'il ne contient aucun mot.

public boolean estGénérateurMinimum()
permet de tester si un langage est un générateur minimum, c'est à dire si chacun de ces mots n'est pas décomposable à partir des autres mots de ce langage.

public Langage getGénérateurMinimum()
permet, si le langage n'est pas un système générateur minimum, de retourner le système générateur minimum de ce langage.

[7] exemple : pour {ab,ba,bb } on obtiendrait { a,ab,b,ba,bb }
[8] pour l'algorithme détaillé, voir section algorithme.

2.3.4 Commandes

public void addEns(Set s)
permet d'ajouter un ensemble de mots à ce langage.

public void addMot(Mot mot)
permet d'ajouter le mot argument au langage.

public void addMot(String mot)
permet d'ajouter le mot argument au langage, mot qui sera créé par cette méthode.

public void removeMot(Mot mot)
permet de retirer un mot du langage.

public void removeMot(String mot)
permet de retirer le mot représenter par la chaîne de caractères passée en argument.

2.3.5 Exemple d'utilisation

1. Construction : nécessite un alphabet
   Alphabet α=new AlphabetImpl() ;
   \\on initialise α à {a,b} (voir plus haut)
   Langage l1=new LangageImpl(α) ; \\création d'un langage vide sur l'alphabet {a,b}
   \\l1=$\emptyset$
2. Ajout de mot
   Mot m1=new MotImpl("abba") ;
   Mot m2=new MotImpl("bb") ;
   l1.addMot(m1) ; \\l1={abba}
   l1.addMot(m2) ; \\l1={abba,bb}
   l1.addMot("cc") ; \\Impossible car "cc" n'est pas engendré par {a,b} alphabet du langage
3. Elimination de mot
   l1.removeMot(m1) ; \\l1={bb}
4. Test de contenance
   l1.isEmpty() ; \\false car contient le mot bb
   Mot m3=new MotImpl("bbbb") ;
   l1.engendre(m2) ; \\true
   l1.engendre(m3) ; \\true car m3=m2.m2
   l1.contains(m1) ; \\false
5. Recherche de facteurs
   Langage l2=new LangageImpl(α) ;// l2=$\emptyset$
   l2.addMot("a") ;\\l2={a}
   l2.addMot("ab") ; \\l2={a,ab}
   l2.addMot("bb") ; \\l2={a,ab,bb}
   Langage l3= l2.prefixe() ; \\l3={@,a,ab,b,bb}
   Langage l4= l2.suffixe() ; \\l4={@,a,b,ab,bb}
   Langage l5=l1.prefixe() ; \\l5={@,b,bb}
   Langage l6=prefEtSuff(l2) ; \\l6={@,a,b}

6. Intersection
   Langage l7=l5.intersection(l4) ; \\7={@,b,bb}

3 Description des méthodes contenues dans les packages utilisés :Package Graphe

3.1 Introduction

Notre implantation de graphe est découpée en plusieurs classes correspondant aux différents éléments constituants un graphe. Nous détaillerons donc dans cette partie chaque élément, c'est-à-dire donner une définition de l'élément puis une brève explication de ses méthodes.

Nous verrons ensuite un exemple d'utilisation de notre graphe, puis nous expliquerons les algorithmes utilisés pour les méthodes de notre graphe dans la partie suivante.

3.2 Sommet

Un sommet est l'élément principal d'un graphe. C'est le symbole d'un état particulier, qui est lui-même relié à d'autres états. Les transitions sont assurées par des arcs, possédant une valeur particulière [9]. Nous reviendrons dessus par la suite.

Dans notre implantation, un sommet correspond à un mot, nous verrons l'utilité par la suite, quand nous définirons notre implantation du graphe de Spehner.

Les méthodes implantées sont les suivantes.

3.2.1 Construction

public SommetImpl(Mot m)
construit un sommet correspondant au mot m passé en paramètre.

3.2.2 Requêtes

public int compareTo(Object o)
permet de comparer deux sommets selon leur mot correspondant, c'est-à-dire selon l'ordre lexicographique.

public boolean equals(Object o)
permet de tester l'égalité de deux sommets, c'est-à-dire s'il possède le même mot correspondant.

3.3 Arc

Un arc est une liaison du graphe, reliant deux sommets [10], et possédant une valeur spécifique [11].Il possède aussi un sens, c'est-à-dire qu'il relie un sommet origine à un sommet extrémité.

3.3.1 Construction

public ArcImpl(Sommet x, Sommet y, EtiquetteCouple e)
permet de construire un arc entre le sommet origine x et le sommet extrémité y, de valeur e.

[9] (étiquette)
[10] ou un sommet à lui-même
[11] étiquette

3.3.2 Requêtes

public boolean equals(Object o)
teste si deux arcs sont égaux, c'est à dire s'il possède la même origine, la même extrémité, ainsi que la même étiquette.

public int compareTo(Object o)
permet de comparer deux arcs en comparant tout d'abord les origines, puis les extrémités si les origines sont égales, puis les étiquettes si les extrémités sont égales.

3.4 Etiquette-Couple

Nous avons choisi une implantation directe de l'étiquette sous la forme d'un couple de deux mots [12]. Une étiquette est une valeur pondérant un arc reliant deux sommets. Nous verrons la signification particulière des étiquettes dans le graphe de Spehner par la suite.

3.4.1 Construction

public EtiquetteCoupleImpl(Mot x, Mot y)
permet de construire une étiquette (x,y) où x et y sont des mots.

3.4.2 Requêtes

public boolean equals(Object o)
permet de tester si deux étiquettes sont identiques, c'est à dire si leurs couples de mots sont identiques.

public int compareTo(Object o)
compare les étiquettes en comparant les premières composantes, puis les secondes en cas d'égalité des premières. Cette fonction renvoie -1 si o est plus grand que l'objet, 1 si o est plus petit, et 0 en cas d'égalité.

3.5 Graphe

Le graphe est défini par un ensemble de sommets, reliés les uns aux autres par des arcs valués par des étiquettes. C'est un outil permettant de représenter et de résoudre des problèmes variés. Ici, nous verrons notre implantation générale d'un graphe, puis nous verrons une spécialisation, le graphe de Spehner, résolvant le problème de savoir si un langage est un code ou non.

3.5.1 Construction

public GrapheImpl()
construit un graphe ne possédant aucun sommet, ni d'arcs valués.

3.5.2 Requêtes

public boolean contains(Sommet s)
permet de tester si le graphe possède le sommet s.

[12] Il serait facile dŚabstraire le type d'étiquette en changeant les valeurs de cette classe.

3.5.3 Commandes

public void addSommet(Sommet s)
permet d'ajouter le sommet s à l'ensemble des sommets du graphe.

public void addArc(Arc a)
permet d'ajouter l'arc a à l'ensemble des arcs du graphe, si l'origine et l'extrémité de l'arc sont bien dans le graphe. L'étiquette de l'arc est alors ajoutée à l'ensemble des étiquettes du graphe.

3.6 Graphe de Spehner

Cet outil est un graphe particulier, construit selon des règles précises que nous verrons dans le chapitre algorithmes, permettant de tester si un langage [13] est un code ou non.

Il étend la notion de graphe en construisant ses sommets, arcs et étiquettes à partir d'un langage.

3.6.1 Construction

public GrapheSpehnerImpl(Langage l)
construit un graphe de Spehner à partir du langage l.

3.7 Matrice d'adjacence

Cette matrice est un outil de représentation de graphe, permettant de voir les liens entre les sommets du graphe. Matrice binaire, elle est indexée sur les sommets et peut prendre les valeurs 1 ou 0 selon la présence ou non d'un arc entre les sommets.

Dans notre implantation, c'est cette matrice qui teste s'il existe des arcs entre des sommets, si des sommets appartiennent à des circuits, etc... Nous expliquerons les algorithmes utilisés dans la section algorithme de ce dossier.

3.7.1 Construction

public MatriceAdjacenceImpl(Graphe g)
permet de construire la matrice d'adjacence du graphe g.

3.7.2 Requêtes

public boolean contains(Sommet s)
teste si le sommet s est dans le graphe représenté par la matrice.

public boolean arcExisteEntre(Sommet s1,Sommet s2)
teste s'il existe un arc entre les sommets s1 et s2.

public boolean aCircuitDe(Sommet s)
teste si le sommet s est sur un circuit dans le graphe.

public Set getArcsDe(Sommet s)
permet d'obtenir un ensemble d'arcs constitués des arcs dont l'origine est le sommet s.

[13] ensemble de mots

public Set getArcsVers(Sommet s)
permet d'obtenir un ensemble d'arcs constitués des arcs dont l'extrémité est le sommet s.

public Arc getArcEntre(Sommet s1, Sommet s2)
permet d'obtenir un arc dont l'origine est s1, et l'extrémité est s2.

public Set getArcsEntre(Sommet s1, Sommet s2)
permet d'obtenir l'ensemble des arcs dont l'origine est s1 et l'extrémité s2.

public boolean aCheminDeVers(Sommet s1,Sommet s2)
teste s'il existe un chemin de s1 à s2 [14].

public LinkedList getCircuitDe(Sommet s1)
permet d'obtenir une suite de sommets sur le circuit de s1, en utilisant un algorithme de *backtracking*[15] que nous décrirons dans la section algorithme.

public Set getSomSurCircuitAvec(Sommet s1)
permet d'obtenir les sommets sur le même circuit que le sommet s1, c'est à dire les sommets présents sur le même circuit que s1.

public Set getSomSurCircuitEtAccAvec(Sommet s1)
permet d'obtenir l'ensemble des sommets sur un même circuit que s1. De plus, chaque sommet est extrémité d'au moins un arc d'origine s1. En fait, cette fonction retourne les sommets accessibles depuis s1 par un chemin de longueur 1, sur le même circuit que s1.

public boolean isAccessible(Sommet s)
teste l'existence d'un chemin de longueur quelconque arrivant au sommet s, c'est-à-dire teste l'existence d'un arc reliant un sommet (différent de s) à s.

public boolean isCoaccessible(Sommet s)
teste l'existence d'un chemin de longueur quelconque partant du sommet s, c'est-à-dire teste l'existence d'un arc reliant s à un sommet (différent de s).

public boolean isIsolé(Sommet s)
teste si le sommet est isolé, c'est à dire teste si le graphe ne possède pas d'arcs reliant le sommet s à un autre sommet [16].

3.8 Exemple de construction d'un graphe

Dans notre cas, nous avons besoin de l'utilisation du package Langage[17].

Nous allons construire un graphe possédant 4 sommets, a, ab, b, bb, les 3 premiers étant sur un même circuit, le 4eme (bb) étant isolé.

[14] c'est à dire une suite d'arcs menant de s1 à s2
[15] rebroussement
[16] si un sommet est isolé, c'est qu'il n'est ni accessible, ni coaccessible
[17] décrit dans les parties précédentes

Ici les étiquettes des arcs n'ont pas de significations précises, nous expliquons la construction d'un graphe où les arcs sont valués par des couples de mots. Nous expliquerons la signification des éléments particuliers du graphe de Spehner dans la partie algorithmes.

1. Création des mots utilisés
   Mot m1=new MotImpl("a") ;
   Mot m2= new MotImpl("ab") ;
   Mot m3= new MotImpl("b") ;
   Mot m4= new MotImpl("bb") ;
2. Création des sommets
   Sommet s1=new SommetImpl(m1) ;
   Sommet s2=new SommetImpl(m2) ;
   Sommet s3=new SommetImpl(m3) ;
   Sommet s4=new SommetImpl(m4) ;
3. Création des étiquettes des arcs
   EtiquetteCouple e1=newEtiquetteCoupleImpl(m1,m1) ;
   \\nous n'utiliserons qu'une étiquette. Pour en construire d'autres,
   \\construire deux mots suivant la construction ci-dessus
   \\puis les placer dans le constructeur d'une étiquette.
4. Création des arcs
   Arc a1=new ArcImpl(s1,s2,e1) ;
   Arc a2=new ArcImpl(s2,s3,e1) ;
   Arc a3=new ArcImpl(s3,s1,e1) ;
5. Création du graphe
   Graphe g=new GrapheImpl() ;
   g.addSommet(s1) ;
   g.addSommet(s2) ;
   g.addSommet(s3) ;
   g.addSommet(s4) ;
   g.addArc(a1) ;
   g.addArc(a2) ;
   g.addArc(a3) ;
6. Utilisation de la matrice d'adjacence
   MatriceAdjacence matr=new MatriceAdjacenceImpl(g) ;
   mat.contains(s1) ; \\true
   mat.arcExisteEntre(s1,s2) ; \\true
   mat.arcExisteEntre(s1,s4) ; \\false
   mat.aCircuitDe(s1) ; \\true
   mat.getArcsDe(s1) ;\\{a1}
   mat.getArcsVers(s2) ;\\{a1}
   mat.getArcEntre(s1,s2) ;\\a1
   mat.getArcsEntre(s1,s2) ;\\{a1}
   mat.aCheminDeVers(s1, s2) ;\\true
   mat.aCheminDeVers(s1, s4) ;\\false
   mat.getCircuitDe(s1) ;\\{s1,s2,s3}
   mat.getSomSurCircuitAvec(s1) ;\\{s1,s2,s3}
   mat.getSomSurCircuitEtAccAvec(s1) ;\\{s2}
   mat.isAccessible(s1) ;\\true
   mat.isAccessible(s4) ;\\false
   mat.isCoaccessible(s1) ;\\true

mat.isCoaccessible(s4) ;\\false
mat.isIsolé(s4) ;\\true

4 Description des algorithmes utilisés

4.1 Graphe de Spehner

4.1.1 Construction du graphe de Spehner

public GrapheSpehnerImpl(Langage l)

Tout d'abord, nous devons construire un graphe simple. Trois ensembles sont nécessaires : les sommets, les arcs, et les étiquettes.

Les sommets du graphe sont définis à partir du langage l d'alphabet a sur lequel est construit le graphe. Un sommet est présent dans le graphe si le mot qui lui correspond (voir chapitres mot et sommet de la description des packages graphe et langage) est à la fois préfixe et suffixe de mots du langage. Prenons un exemple. Pour le langage {a, ab, ba} engendré par l'alphabet {a, b}, en utilisant la méthode prefEtSuff(), nous obtenons l'ensemble de mots {@, a, b}. En effet, a est suffixe de a et préfixe de ab, b est préfixe de ba et suffixe de ab, et @ est préfixe et suffixe de tout mot.

Expliquons maintenant l'ajout des arcs du graphe. Un arc (u,v) d'étiquette (x,y) [18] et y un mot de l* est ajouté si et seulement si x=u.y.v avec u.y !=@ et y.v !=@. C'est-à-dire que pour chaque mot du langage, nous devons vérifier s'il peut se décomposer comme une concaténation de mots. Ainsi nous utilisons l'algorithme suivant :

```
Pour tout x dans l
    Pour tout u de a* préfixe et suffixe de l (donc sommet du graphe)
        Pour tout v de a* préfixe et suffixe de l (donc sommet du graphe)
            tmp=x
            si u est préfixe de tmp alors
                retirer préfixe u de tmp
                si v est suffixe de tmp alors
                    retirer suffixe v de tmp
                    si l engendre tmp alors
                        si concaténation(u,tmp) est différente de @ alors
                            si concaténation(tmp,v) est différente de @ alors
                                si u et v sont différents de @
                                    ajouter l'arc (u,v) d'étiquette (x,tmp) au graphe
                                finsi
                            finsi
                        finsi
                    finsi
                finsi
            finsi
        fin Pour tout
    fin Pour tout
fin Pour tout
```

Ainsi nous construisons un graphe respectant les règles de construction du graphe de Spehner.

[18] où u et v sont des sommets du graphe et x est un mot de l

4.2 Matrice d'adjacence

4.2.1 Existence d'un arc

public boolean arcExisteEntre(Sommet s1,Sommet s2)
Pour cette fonction, il suffit de tester pour chaque arc du graphe si l'origine est s1 et l'extrémité s2. Si un arc vérifie cette condition, alors la méthode retourne vrai.

4.2.2 Présence d'un sommet sur un circuit quelconque

public boolean aCircuitDe(Sommet s)
Pour tester si le graphe possède un circuit contenant le sommet s, nous avons choisi d'utiliser une itération sur les arcs. Chaque sommet accessible est ajouté à une liste de traitement, et si tous les sommets sont traités sans avoir trouvé d'arc revenant en s, alors il n'y a pas de circuit. L'algorithme est le suivant :

```
Soit F une file vide (outil FIFO) (éléments à traiter)
Soit E un ensemble (élément déjà traités)
Si s ne possède aucun arc sortant alors
    Retourner faux
finsi
Ajouter s à F.
Tant que E != sommets du graphe faire
    f=défiler(F) où défiler renvoie le premier élément de F en l'éliminant de F
    ajouter f à E
    Soit U l'ensemble des arcs sortant de f
    Pour tout u de U faire
        Soit t le sommet extrémité de u
        Si t=s et que f!=s alors
            Retourner vrai
        Finsi
        Si E ne contient pas t alors
            Ajouter t à F
        Finsi
    Fin pour tout
    Si F est vide alors
        Retourner faux
    Fin si
Fin Tant que
Retourner faux
```

Dès qu'un circuit est trouvé, l'algorithme retourne vrai. Si tous les sommets accessibles ont été explorés sans pouvoir trouver un arc retournant vers le sommet origine s, alors il n'y a aucun circuit contenant s dans le graphe.

4.2.3 Recherche d'arcs particuliers :arcs sortant d'un sommet

public Set getArcsDe(Sommet s)
Pour renvoyer l'ensemble des arcs sortant de s, nous créons un ensemble solution, vide au départ que nous remplissons, en itérant sur chaque arc du graphe, par les arcs dont l'origine est s.

4.2.4 Recherche d'arcs particuliers :arcs rentrant vers un sommet

public Set getArcsVers(Sommet s)
Pour renvoyer l'ensemble des arcs entrant vers s, nous créons un ensemble solution, vide au départ que nous remplissons, en itérant sur chaque arc du graphe, par les arcs dont l'extrémité est s.

4.2.5 Recherche d'arcs particuliers :arc reliant deux sommets

public Arc getArcEntre(Sommet s1, Sommet s2)
Pour obtenir un arc qui part de s1 vers s2, nous testons chaque arc si son origine est s1 et si son extrémité est s2. Le premier arc trouvé est alors renvoyé.

4.2.6 Recherche d'arcs particuliers :ensemble d'arcs reliant deux sommets

public Set getArcsEntre(Sommet s1, Sommet s2)
Pour obtenir l'ensemble des arcs existants, nous testons chaque arc du graphe et si son origine est s1 et son extrémité est s2, nous le plaçons dans l'ensemble retourné en solution de l'algorithme.

4.2.7 Existence d'un chemin entre deux sommets

public boolean aCheminDeVers(Sommet s1,Sommet s2)
Pour tester si le graphe possède une suite d'arc reliant le sommet s1 au sommet s2, nous utilisons le même algorithme que pour trouver un circuit, à la différence que l'algorithme s'arrête quand il trouve un arc menant à s2.

4.2.8 Recherche et Obtention d'un circuit

public LinkedList getCircuitDe(Sommet s1)
Pour obtenir un circuit dans un graphe (en ayant déjà testé s'il y en avait un), nous avons mis au point un algorithme de backtracking décomposé en deux parties. Tout d'abord une partie d'initialisation, contenue dans cette fonction nommée getCircuitDe, puis une fonction récursive nommée getCheminDe(Sommet s1,Sommet s2, Set somBloqués) où s1 est l'origine du chemin recherché, s2 l'extrémité et somBloqués un ensemble de sommets par lesquels le chemin ne doit pas passer. En effet, il est nécessaire de bloquer des sommets afin de ne pas tomber dans une recherche infinie en cas de circuit.

La première partie initialise les ensembles utiles, lance la fonction de backtracking puis renvoie la solution. La fonction de backtracking permet d'avancer sommet par sommet vers un circuit, et de faire marche arrière si elle est bloquée.
Nous allons donc étudier ces deux parties.

public LinkedList getCircuitDe(Sommet s1)

Soit A l'ensemble des sommets sur un même circuit que s1.
Soit résultat une liste (contenant les sommets consécutifs du circuit)
ajouter s1 à res
Soit F les sommets sur un même circuit que s1 et accessibles directement(il existe un arc entre les deux)
Pour tout f de F faire
s'il existe un arc entre f et s1 alors
ajouter f à résultat
ajouter s1 à résultat
renvoyer résultat

finsi
fin pour tout
Choisir un sommet x dans F
Soit sommetBloqués un ensemble vide (ensemble des sommets bloqués utilisés pour le rebroussement)
Ajouter s1 à sommetBloqués
Ajouter à résultat un chemin de x vers s1 qui ne traverse pas sommetBloqués
retourner résultat

List getCheminDe(Sommet s1,Sommet s2, Set somBloqués)
Cette fonction retourne un chemin de s1 vers s2 qui ne passe pas par les sommets bloqués. Ainsi, on peut gérer un rebroussement afin d'éviter de tomber dans un circuit infini.

Soit res une liste.
Ajouter s1 aux sommets bloqués.
Ajouter s1 à res.
S'il existe un arc reliant s1 à s2 alors
ajouter s2 à res
retourner res
finsi
Soit E un ensemble(ensemble des sommets à traiter)
Pour tout x sommet sur le même circuit que s1 et accessible directement depuis x faire
si x n'est pas bloqué alors
ajouter x à E
finsi
finpourtout
si E est vide alors
retirer s1 de res
renvoyer un chemin vide
finsi
pour tout sommet x de E faire
Soit chemin une liste
chemin=getCheminDe(x,s2,somBloqués)
si chemin n'est pas vide alors
ajouter chemin à res
retourner res
finsi
finpourtout

4.2.9 Recherche de circuit contenant un sommet

public Set getSomSurCircuitAvec(Sommet s1)
Pour trouver les sommets sur le même circuit que s1, il suffit d'étudier, pour chaque sommet s2 différent de s1 s'il existe un chemin de s1 vers s2, et de s2 vers s1. Si ces deux chemins existent alors s2 est ajouté à l'ensemble solution.

4.2.10 Recherche de sommets sur un même circuit qu'un sommet et accessibles par un arc depuis ce sommet

public Set getSomSurCircuitEtAccAvec(Sommet s1)
Cette fonction renvoie les sommets sur un même circuit et connexes avec s1 par un arc vers s1. Il suffit de déterminer les sommets sur le circuit, puis de regarder quels sont ceux où un arc existe entre s1 et ces sommets. Si un sommet vérifie ces conditions, alors il est ajouté à l'ensemble retourné.

4.2.11 Test d'accessibilité de sommets

public boolean isAccessible(Sommet s)
Pour tester si un sommet est accessible depuis un autre sommet, il suffit de tester s'il existe un arc rentrant en s, avec pour origine un sommet différent de s.

4.2.12 Test de coAccessibilité de sommets

public boolean isCoaccessible(Sommet s)
Pour tester si un sommet est coaccessible, il suffit de tester s'il existe un arc sortant de s dont l'extrémité est un sommet différent de s

4.2.13 Test : sommet isolé

public boolean isIsolé(Sommet s) Un sommet est isolé s'il est ni accessible ni coaccessible.

4.3 Langage

4.3.1 Génération de mots

public boolean engendre(Mot m)
Pour tester si un mot peut être engendré, par l'opération *, par un langage, nous avons choisi de décomposer le mot selon des préfixes correspondants aux mots de ce langage, puis pour chacun d'entre eux, de retirer ce préfixe et de tester si le mot obtenu est engendré par le langage. L'algorithme est le suivant :

```
Si m est le mot vide alors
    Retourner vrai
FinSi
Si m n'est pas un mot engendré par l'alphabet du langage alors
    Retourner faux
FinSi
Si le langage contient m alors
    Retourner vrai
FinSi
Resultat<-faux
Soit EnsPref l'ensemble des préfixes de m
Si aucun des préfixes contenus dans EnsPref n'est dans le langage alors
    Retourner faux
FinSi
Pour tout pref de EnsPref faire
    Tmp<-m
    Si pref est un mot du langage alors
        Retirer le préfixe pref de tmp
        Retourner (résultat ou engendre(tmp))
```

```
    FinSi
FinPourTout
Retourner résultat
```

Ainsi nous testons si un mot peut se décomposer comme concaténation de facteurs appartenant à un ensemble, le langage.

4.3.2 Découpage en facteurs particuliers : préfixes et suffixes

public Langage prefEtSuff()
Cette fonction renvoie un ensemble de mots spécifiques. Un mot qui est présent dans cet ensemble vérifie une de ces deux propriétés :soit c'est un préfixe d'un mot du langage et suffixe d'un second mot différent du premier, soit c'est un préfixe et un suffixe du même mot, mais différent de ce mot. Prenons un exemple : le langage " prefEtSuff " du langage abaaba serait alors {@, a, aba}. L'algorithme utilisé est le suivant :

```
Soit a une copie de l'alphabet sur lequel est construit le langage
Ajouter @ à a
Soit Résultat un nouveau langage sur a
Ajouter @ à Résultat
Soit l un nouveau langage sur a
Copier les mots du langage dans l
Pour tout mot x de l faire
    Retirer x à l
    Soit l2 l'ensemble des suffixes de l
    Soit s l'ensemble des préfixes de l
    Pour tout mot y de s faire
        Si y appartient à l2 alors
            Ajouter y a Résultat
        finSi
    FinPourTout
    Ajouter x à l
FinPourTout
Soit PrefPropr un ensemble (stockage des préfixes propres du mot traité)
Soit SuffPropr un ensemble (stockage des suffixes propres du mot traité)
Pour tout mot x du langage faire
    PrefPropr<-préfixes propres de x
    SuffPropr<-suffixes propres de x
    Pour tout pref de PrefPropr faire
        Pour tout suff de SuffPropr faire
            Si pref=suff alors
                Ajouter pref à Résultat
            FinSi
        FinPourTout
    FinPourTout
FinPourTout
Retourner Résultat
```

Ainsi, chaque mot présent dans le langage solution de l'algorithme est un mot vérifiant les critères de départ.

4.3.3 Test et obtention d'un système générateur libre

public boolean estGénérateurMinimum()
Pour tester si un langage est un système générateur libre, il suffit de tester si chacun des mots qui le composent ne peut être engendré par la concaténation de mots de ce langage. Il suffit donc de tester pour tout mot m du langage l si l-m ne l'engendre pas. Si on trouve un mot qui est la concaténation d'autres mots de l, on renvoie faux. Si aucun des mots ne vérifie cette propriété, alors le langage est un générateur libre.

public Langage getGénérateurMinimum()
Pour obtenir le système générateur minimum d'un langage l, il suffit de lui retirer chaque mot engendré par l-m. Si le langage est déjà un système générateur minimum, alors il est inchangé et la fonction le retourne.

5 Manuel Utilisateur

Cette partie correspond à la description de la partie graphique de notre application. La figure 1 représente notre interface graphique.

L'utilisation se fait en trois étapes :

Tout d'abord, il faut créer l'alphabet en entrant les lettres le constituant, puis construire le langage en entrant les mots écrits sur l'alphabet précédemment créé. Enfin la troisième étape consiste en l'affichage du graphe de Spehner (sa matrice d'adjacence) et de tous les outils qui vont avec (Recherche de mots ayant une double décomposition, exemple de double décomposition, remise à zéro et réponse à la question : "Le langage est-il un code ?").

Ce manuel se décomposera donc en trois principales sous parties qui seront :

1. Création de l'alphabet.
2. Ajout des mots au langage.
3. Affichage du graphe de Spehner et utilisation des différents outils proposés.

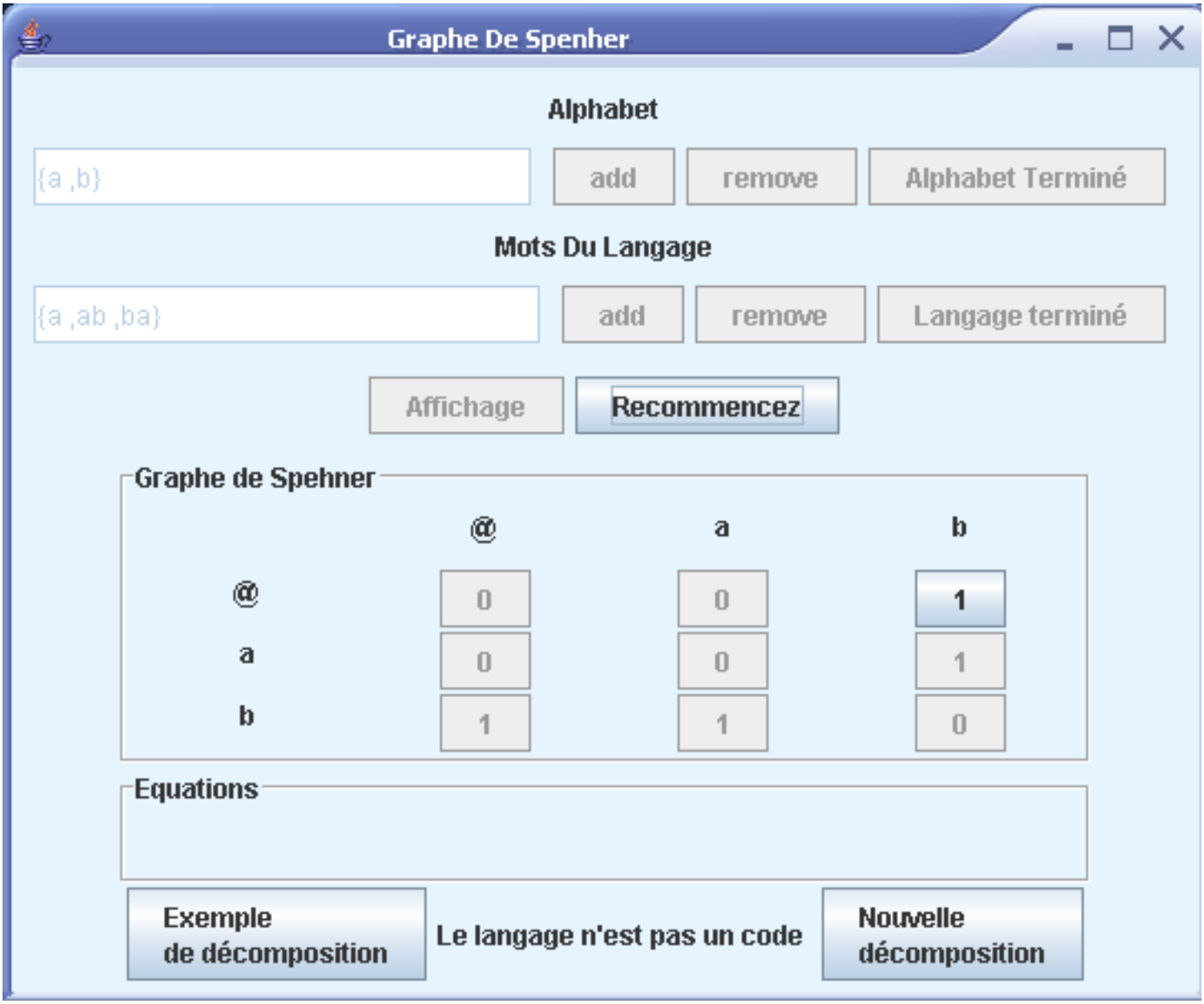


FIG. 1 – Partie graphique de notre application.

5.1 Création de l'alphabet

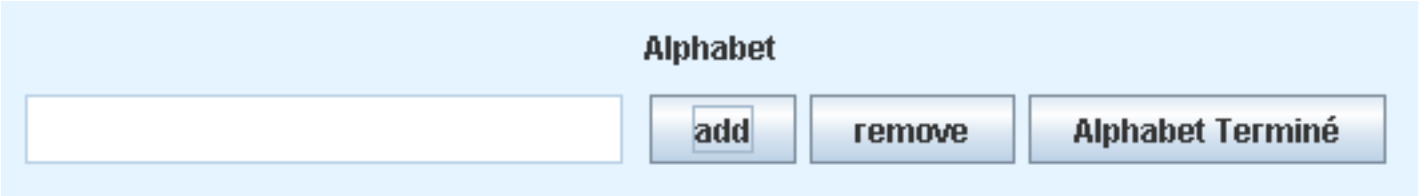


FIG. 2 – Création de l'alphabet.

La figure 2 représente la partie de notre interface graphique permettant la création de l'alphabet. Sur cette image, nous pouvons voir une zone de texte permettant l'affichage de l'alphabet (représentée par un JLabel en Java) suivie de 3 boutons représentant chacun une action bien précise.

L'appui sur le bouton "Add" ouvre une fenêtre pop-up avec un champ texte éditable (un JTextField en Java) permettant à l'utilisateur de rentrer une lettre et ainsi pouvoir l'ajouter à l'alphabet. De même,

l'appui sur le bouton "Remove" ouvre une fenêtre pop-up avec un champ texte éditable permettant à l'utilisateur de rentrer une lettre et ainsi pouvoir retirer cette lettre de l'alphabet. L'appui sur l'un de ces deux boutons provoque l'actualisation du champ texte permettant la visualisation de l'alphabet.

L'utilisateur est automatiquement prévenu de la réussite ou de l'échec des différentes actions par l'ouverture d'une fenêtre pop-up (JOptionPane en Java) comme le montre l'exemple sur la figure 4 indiquant une erreur lors de l'ajout d'une lettre à l'alphabet ou la figure 3 représentant un message affiché lorsque l'ajout d'une lettre à l'alphabet s'est déroulé avec succès.

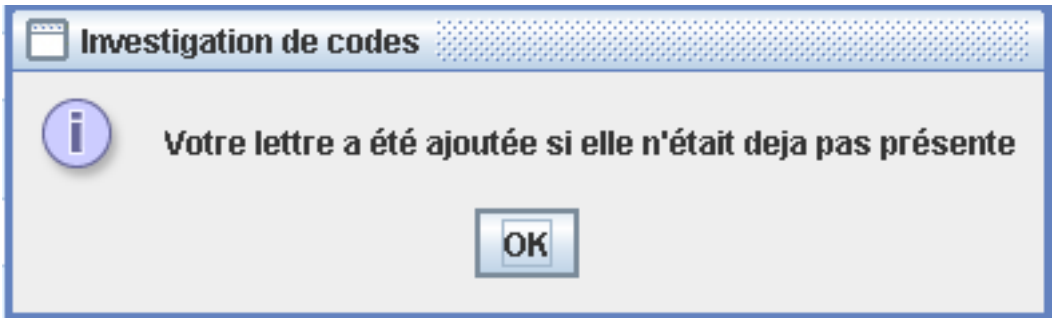


FIG. 3 – Exemple de message lors d'une action réussie.

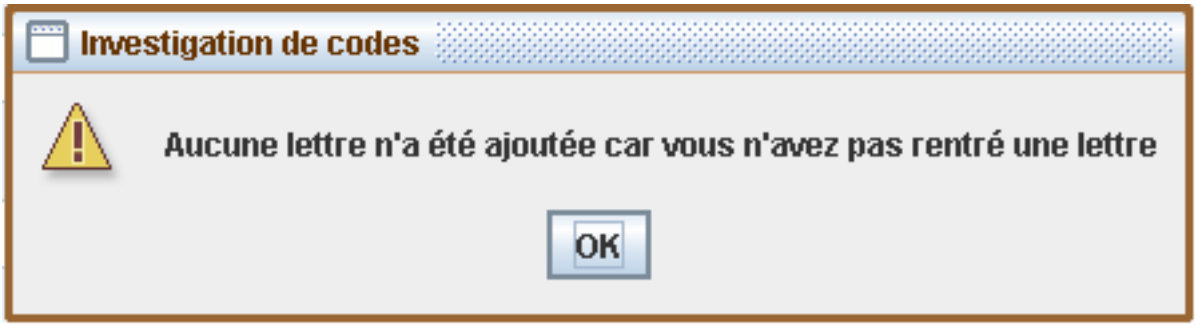


FIG. 4 – Exemple de message d'erreur.

Le troisième bouton "Alphabet Terminé" permet de valider la création de l'alphabet et de passer à l'étape suivante. Après l'appui sur ce bouton, l'alphabet créé ne sera plus éditable.

5.2 Création du langage

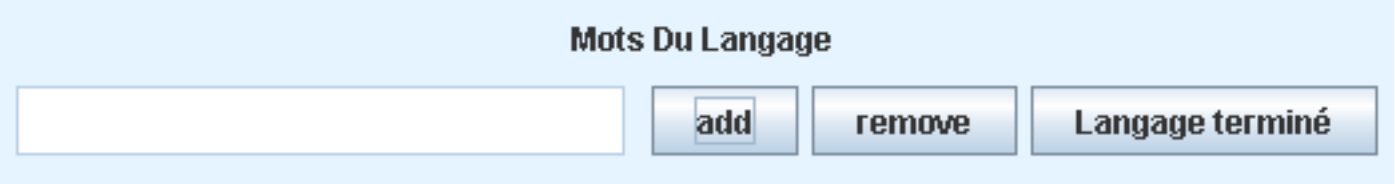


FIG. 5 – Création du langage.

La figure 5 représente la partie graphique de notre interface permettant la création du langage basé sur l'alphabet précédemment créé. Sur cette image, nous pouvons voir une zone de texte permettant l'affichage des différents mots constituant le langage et trois boutons permettant d'effectuer diverses actions sur celui-ci.

L'appui sur le bouton "Add" ouvre une fenêtre pop-up avec un champ texte éditable (un JTextField en Java) permettant à l'utilisateur de rentrer un mot et ainsi pouvoir ajouter ce mot au langage. L'appui sur le bouton "Remove" provoque l'ouverture lui aussi d'une fenêtre pop-up avec un champ texte éditable (un JTextField en Java) permettant à l'utilisateur de rentrer un mot et ainsi pouvoir retirer ce mot du langage.
L'appui sur l'un de ces deux boutons provoque l'actualisation du champ texte permettant la visualisation des différents mots constituant le langage. L'utilisateur est automatiquement prévenu de la réussite ou de l'échec des différentes actions par l'ouverture d'une fenêtre pop-up (JOptionPane en Java) comme le montre l'exemple sur la figure 7 indiquant une erreur lors de l'ajout d'un mot ou la figure 6 indiquant la réussite de l'action demandée.
Le troisième bouton appelé "Langage terminé" sert à signaler à notre logiciel que tous les mots ont été ajoutés au langage par l'utilisateur. Le logiciel teste alors si le langage entré est un générateur minimal. Si c'est le cas, le langage rentré par l'utilisateur reste le même. Sinon on extrait le générateur minimal de ce langage et c'est sur celui-ci que le logiciel travaillera par la suite. La figure 8 représente le message affiché lorsque le langage proposé par l'utilisateur n'est pas un générateur minimal.

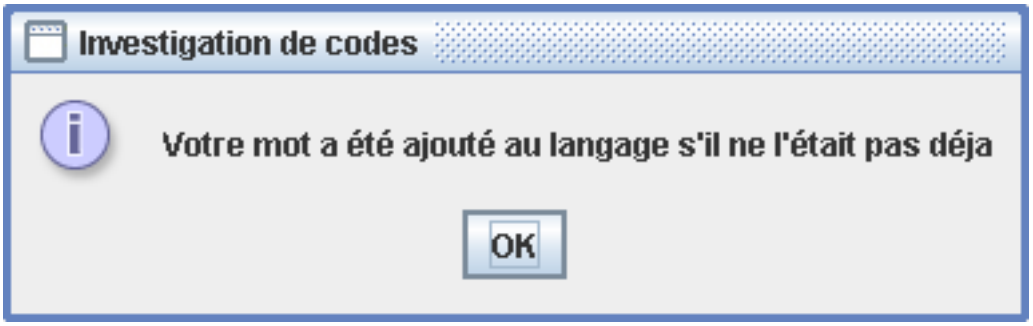


FIG. 6 – Exemple de message lors d'une action réussie.

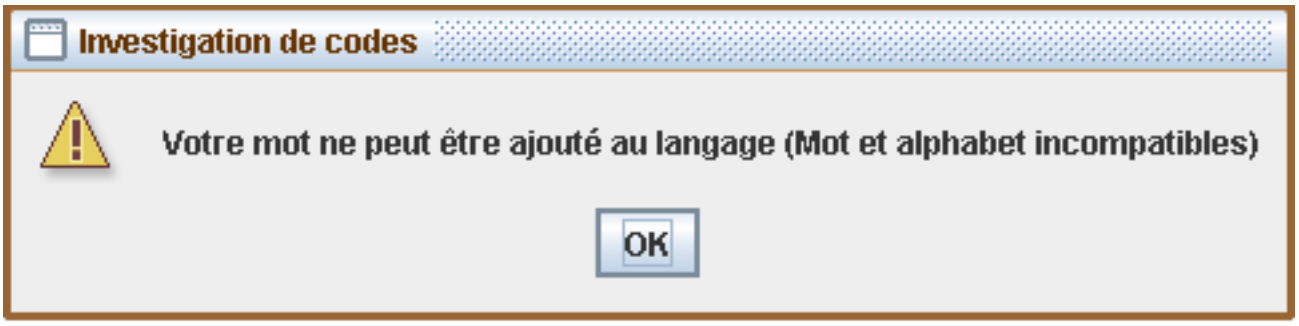


FIG. 7 – Exemple de message d'erreur.

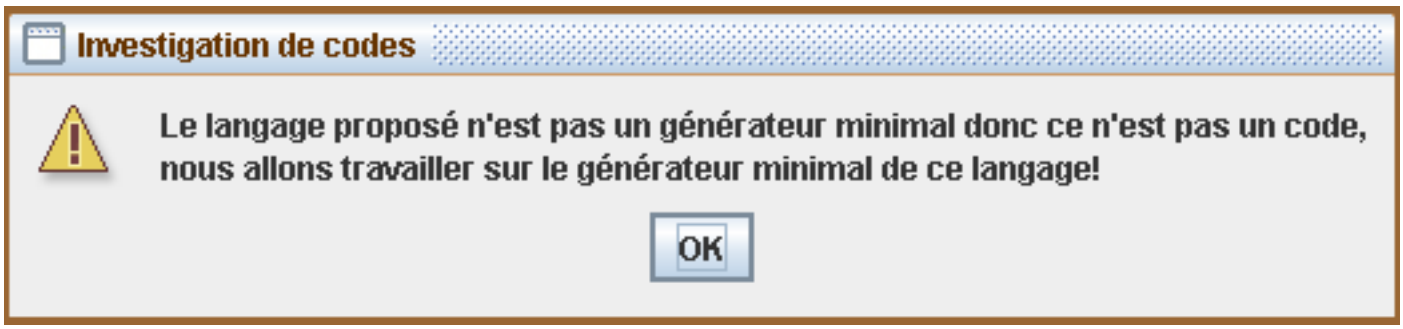


FIG. 8 – Message affiché lorsque le langage entré n'est pas un générateur minimal.

5.3 Affichage du graphe de Spehner et utilisation des différents outils proposés.

Après avoir créé l'alphabet et le langage, l'utilisateur peut alors créer le graphe de Spehner[19], plus précisément la matrice d'adjacence du graphe de Spehner, et ainsi vérifier que le langage construit précédemment est un code.

FIG. 9 – Affichage du graphe et réinitialisation de notre interface.

La figure 9 représente la partie graphique concernant l'affichage de la matrice d'adjacence du graphe de Spehner associée au langage précédemment construit et la réactualisation de notre interface pour effectuer un nouveau test sur un autre langage. L'appui sur le bouton "Affichage" crée la matrice d'adjacence et l'affiche au sud de notre application graphique. L'appui sur le bouton "Recommencez" crée un nouvel alphabet ainsi qu'un nouveau langage et efface la matrice d'adjacence créée au sud (si une été déjà présente auparavant) pour permettre à l'utilisateur de tester un nouveau langage.

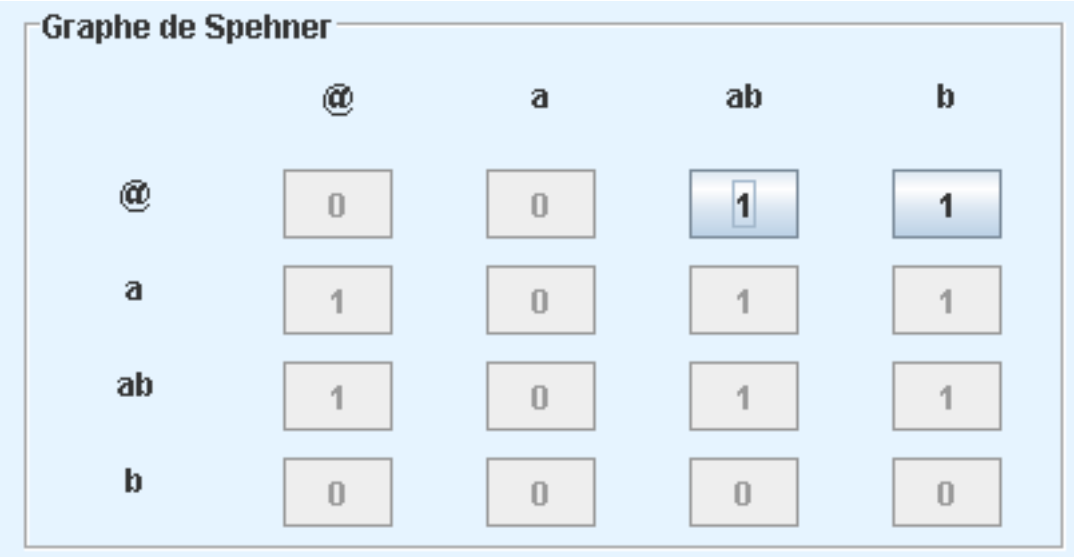


FIG. 10 – Matrice d'adjacence représentant le graphe de Spehner.

La figure 10 représente un exemple d'une matrice d'adjacence représentant le graphe de Spehner pour le langage {ab,a,b,abab}.Tous les boutons représentés avec un "1" représentent un arc entre deux sommets du graphe, tous les boutons avec un "0" indiquent qu'il n'existe pas d'arc entre les deux sommets de la matrice.

Une des fonctionnalités de ce graphe est de permettre à l'utilisateur, si le langage n'est pas un code, de trouver un exemple de mots ayant une double décomposition. Tous les arcs étant sur un cycle de ϵ à ϵ et partant de ϵ sont allumés au départ. Reprenons la matrice d'adjacence de la figure 10. Nous pouvons voir que les sommets ab et b sont sur un cycle de ϵ vers ϵ et accessibles à partir de ϵ car les arcs (ϵ,b) et (ϵ,ab) ont un label égal à "1".
Ensuite un clic sur le bouton représentant l'arc (ϵ,ab) provoque l'allumage des boutons représentant les

[19]J.C. SPEHNER *Université de Mulhouse*

arcs partant de ab et dont les sommets d'arrivée sont sur un cycle de ϵ vers ϵ, c'est à dire les arcs (ab,ϵ) et (ab,ab) (cf figure 11). Et ainsi de suite...

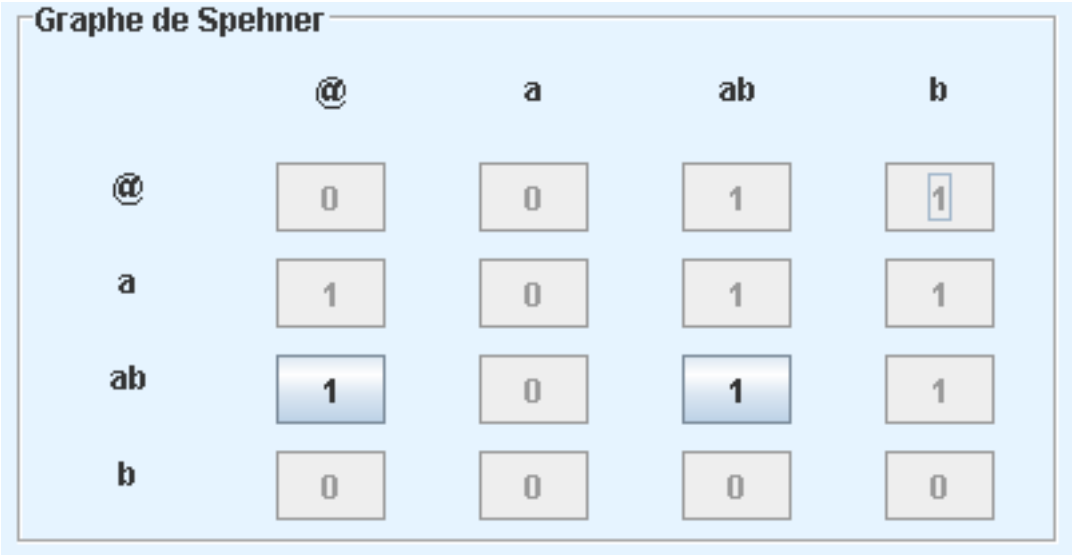


FIG. 11 – Exemple parcours du graphe.

L'appui sur l'un des arcs, pendant l'exploration du graphe à la recherche de mots ayant une double décomposition, provoque l'affichage dans la partie "Équations" des équations des mots en double décomposition, c'est à dire de la valeur des étiquettes mises dans un sens particulier selon que le nombre d'arcs visités précédemment est pair ou impair. Si le signe de l'équation est un égal, cela implique que l'utilisateur a parcouru un cycle de ϵ vers ϵ et qu'il a trouvé un mot en double décomposition (cf figure 12).

FIG. 12 – Mot en double décomposition.

La figure 13 montre les différents outils associés au graphe de Spehner lorsque le langage proposé n'est pas un code. Vous pouvez demander au logiciel de générer un mot en double décomposition avec le bouton "Exemple de décomposition".Un exemple vous est donné à la figure 15. De plus, vous avez une indication sur la nature du langage, s'il est un code préfixe, suffixe ou aucun des deux. Vous pouvez alors re-parcourir la matrice en effaçant les parcours déjà effectués grâce au bouton "Nouvelle Décomposition". Les deux boutons situés sur cette figure ne s'affichent que dans le cas où le langage n'est pas un code. A contrario, seul le message précisant la nature du langage est représenté comme le montre la figure 14.

FIG. 13 – Différents outils associés à la matrice d'adjacence (si le langage n'est pas un code).

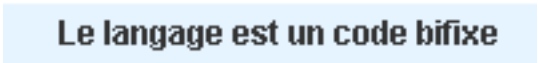


FIG. 14 – Exemple de message lorsque le langage proposé est un code.

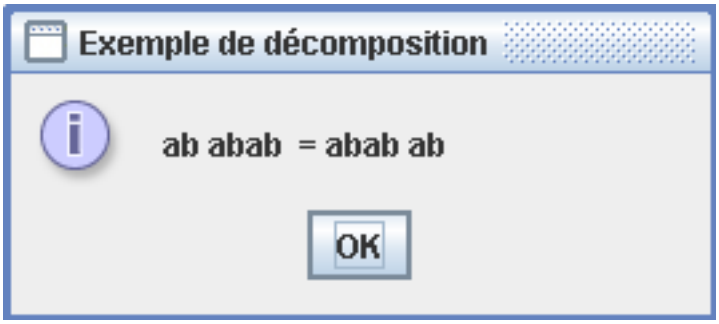


FIG. 15 – Ouverture d'une fenêtre avec un exemple de mot en double décomposition.

Apres avoir fait des essais sur certains langages voici quelques exemples :
$L = \{aaa, bbb, aab, aaab, abb, abaa, bba\}$ n'est pas un code car aaa.bbb.aaab = aaab.bba.aab ;
$L = \{aa, aba, abb, ba, bb\}$ est un code préfixe ;
$L = \{ab, a^3b, bcb, c, bac, ac\}$ est un code préfixe ;

Conclusion

En conclusion, les problèmes sont d'ordre algorithmique pour la plupart. En effet nous avons dû rechercher différents algorithmes pour obtenir différentes opérations implantées dans notre logiciel. De plus les algorithmes que nous avons créés ne sont pas optimum en terme de complexité. Cependant la taille des graphes couramment utilisés n'est pas suffisament importante pour ralentir le système.

Grâce au langage Java, notre logiciel et ses différents algorithmes sont évolutifs. Nous avons découpé notre logiciel en différentes classes permettant à un nouveau programmeur de les redéfinir comme il le souhaite dans un souci d'optimisation. Notre interface graphique est assez simple d'utilisation pour plus de convivialité.

D'autres algorithmes existent pour répondre à la problématique de départ[20] mais cependant la méthode de J.C. Spehner nous fournit plus de détails sur le langage et sur les mots en double décomposition.

[20] Le langage est-il un code ?

Références

[1] Jean BERSTEL and Dominique PERRIN. *Theory of codes.* Academic press, 1985.

[2] Thomas H. CORMEN, Charles E. LEISERSON, Ronald L. RIVEST, and Clifford STEIN. *Introduction à l'algorithmique.* DUNOD, 2002.

[3] M. Lothaire. *Combinatorics on words.* Cambridge University Press, 1997.

[4] Jean-Claude SPEHNER. *Quelques problèmes d'extension, de conjugaison, et de présentation des sous monoïdes d'un monoïde libre.* PhD thesis, Université Paris VII, 1976.