Projet d'algorithmique des graphes *[IF107/PG105]*

Rapport final 16/05/2007

Équipe de développement :

segment impair
- Thibaut Carlier
- Mohamed El Afrit
- Matthieu Lefebvre

segment pair
- Maxime Bochon
- Yannick Martin

Table des matières

1 Introduction **5**
- 1.1 Présentation du problème . . . 5
- 1.2 Objectifs techniques . . . 5
- 1.3 Démarche initiale . . . 5

2 Notions sur les graphes **6**
- 2.1 Graphes . . . 6
  - 2.1.1 Graphe orienté . . . 6
  - 2.1.2 Graphe simple . . . 6
  - 2.1.3 Graphe vide . . . 7
  - 2.1.4 Graphe discret . . . 7
- 2.2 Éléments constitutifs . . . 7
  - 2.2.1 Sommets . . . 7
  - 2.2.2 Arcs . . . 7
  - 2.2.3 Chemins . . . 8
- 2.3 Opérations sur les graphes simples orientés . . . 8
  - 2.3.1 Initialisation . . . 8
  - 2.3.2 Ajout d'un sommet . . . 8
  - 2.3.3 Ajout d'un arc . . . 8
  - 2.3.4 Transposition . . . 8
- 2.4 Forte connexité . . . 8
  - 2.4.1 Remarques préalables . . . 8
  - 2.4.2 Sommets fortement connexes . . . 8
  - 2.4.3 Graphe fortement connexe . . . 8
  - 2.4.4 Composante fortement connexe . . . 9

3 Analyse **10**
- 3.1 Modélisation et type de graphe . . . 10
  - 3.1.1 Pertinence de l'objet graphe . . . 10
  - 3.1.2 Choix d'un type de graphe . . . 11
  - 3.1.3 Conclusion . . . 12
- 3.2 Reformulation du problème . . . 12
  - 3.2.1 Forte connexité . . . 12
  - 3.2.2 Connexion universelle . . . 12

4 Algorithmes relatifs à la forte connexité **14**
- 4.1 Conventions . . . 14
- 4.2 Recherche des composantes fortement connexes . . . 14
  - 4.2.1 Parcours du graphe . . . 14
  - 4.2.2 Algorithme de parcours du graphe . . . 15
  - 4.2.3 Transposition du graphe . . . 16
  - 4.2.4 Parcours du graphe transposé . . . 16

5 Algorithmes relatifs à la conversion universelle **19**
5.1 Réduction des composantes fortement connexes 19
5.1.1 Représentant d'un sommet 19
5.2 Construction du graphe quotient 19
5.2.1 Description du problème 19
5.2.2 Successeurs aux composantes fortement connexes 20
5.2.3 Le graphe quotient 22
5.3 Algorithme de complétion 24
5.4 Fonctions auxiliaires 25
5.5 Manipulation de la liste d'arcs du graphe biparti 25
5.6 Sources et puits du graphe quotient 25

6 Types abstraits de données **29**
6.1 T.A.D. Liste de Sommets (LS) 29
6.2 T.A.D. Liste d'Arcs (LA) 29
6.3 T.A.D. Partition (P) 30
6.4 T.A.D. Graphe (G) 30
6.5 Conventions de notation 30

7 Implémentation en Lisp **32**
7.1 Représentation des graphes 32
7.1.1 Structures en Lisp 32
7.1.2 Point commun entre les représentations 32
7.1.3 Représentation concrète par arcs (RCA) 33
7.1.4 Représentation concrète par successeurs (RCS) 33
7.1.5 Représentation abstraite par adjacence (RAA) 33
7.1.6 Passage d'une représentation à une autre 33
7.2 Implémentations des TAD 33
7.2.1 Liste de sommets 34
7.2.2 Liste d'arcs 34
7.2.3 Partition 34
7.2.4 Graphe 35
7.2.5 Implémentation du TAD graphe 35
7.3 Implémentation de l'algorithme de forte connexité 35
7.3.1 Parcours du graphe 36
7.3.2 Détermination des CFC 37
7.3.3 Le graphe est-il connexe ? 37
7.4 Implémentation de l'algorithme de connexion universelle 37
7.4.1 Le graphe quotient 37
7.4.2 Completion du graphe 39
7.5 Les performances 41
7.6 Rapport de bogues 41
7.6.1 Bogue 1 : parcours manquant 41
7.6.2 Bogue 2 : tri futile 41

8 Implémentation en C **43**
8.1 Adaptation du TAD 43
8.1.1 Restriction au premier problème 43
8.1.2 Gestion mémoire 43
8.1.3 Copie ou partage 43
8.1.4 Effets de bord 45
8.1.5 Rôle des pointeurs 45
8.2 Approche modulaire 45
8.3 Une implémentation des TAD auxiliaires 46

8.3.1 Liste de Sommets 46
8.3.2 Liste d'arcs 48
8.3.3 Partition 49
8.4 Deux implémentations du TAD Graphe 49
8.4.1 Représentation par liste d'arcs 49
8.4.2 Représentation par matrice d'adjacence 50
8.5 Tests de fonctionnalités 53
8.5.1 TAD graphe 53
8.5.2 TAD liste de sommets 53
8.5.3 TAD liste d'arcs 54
8.5.4 TAD partition 54
8.6 Rapport de bogues 54
8.6.1 Bogue 1 : problème d'initialisation 54
8.6.2 Bogue 2 : (ré)allocations 54
8.6.3 Bogue 3 : prévoir la réallocation 54
8.6.4 Bogue 4 : condition manquante 54
8.6.5 Bogue 5 : consommation excessive (non résolu) 55
8.7 Performance du programme 55

9 Comparaison **56**
9.1 Gestion de la mémoire 56
9.2 Complexité en temps 56
9.3 Passage des arguments 56
9.4 Résultats des mesures 56

10 Conclusion **57**

Chapitre 1

Introduction

1.1 Présentation du problème

On dispose d'un ensemble de formats de fichiers pouvant être convertis d'un format vers un autre par le biais de programmes à usage unique (conversion d'un unique format vers un unique format). Ces programmes ne constituent pas a priori un jeu suffisant de convertisseurs pour couvrir toutes les possibilités.

De cette situation émergent les problématiques suivantes :
- Est-il possible, en combinant les programmes disponibles, de passer de n'importe quel format à n'importe quel autre ?
- Si ce n'est pas le cas, combien au minimum faudrait-il ajouter de programmes de conversion pour que cela soit possible, et lesquels ?

1.2 Objectifs techniques

Ce projet s'articule autour de trois grands axes :
- La théorie des graphes et les algorithmes s'y rattachant.
- La programmation fonctionnelle en Lisp appliquée aux graphes.
- La programmation impérative en C appliquée aux graphes.

Le premier axe devra modéliser le problème et y apporter des solutions algorithmiques, qui seront par la suite implémentées en Lisp, puis en C. Les deux derniers axes feront l'objet d'une confrontation qui permettra de prendre conscience du potentiel et des limitations de chacun de ces langages.

1.3 Démarche initiale

Après avoir défini les notions requises par ce projet, une phase d'analyse permettra de modéliser l'énoncé du problème et de le reformuler grâce à la théorie des graphes. Ceci nous conduira à développer deux solutions algorithmiques pour résoudre d'une part le problème de la forte connexité, et d'autre part celui de la connexion universelle. La synthèse de ces algorithmes permettra de définir des types abstraits optimums qui serviront lors des phases d'implémentation en Lisp et en C. Cela permettra finalement de comparer deux approches différentes de la programmation autour d'un sujet commun.

Chapitre 2

Notions sur les graphes

Les notions de théorie des graphes utilisées dans ce rapport méritent d'être définies pour éviter toute ambiguité.

2.1 Graphes

2.1.1 Graphe orienté

Définition

Un graphe orienté G est un triplet (V, E, f) tel que :
- V désigne l'ensemble des sommets de G.
- E désigne l'ensemble des arcs de G.
- f désigne l'application de E dans V^2 qui à chaque arc de G associe ses sommets de départ et d'arrivée.

Remarques

En présence de plusieurs graphes, on utilisera un indiçage pour différencier leurs éléments respectifs. Dans le cas présent, on aurait : $G = (V_G, E_G, f_G)$.

La définition actuelle de f autorise l'existence de plusieurs arcs ayant les mêmes sommets de départ et d'arrivée.

Les notions de sommet et d'arc seront définies ultérieurement.

2.1.2 Graphe simple

Définition

Un graphe (V, E, f) est dit simple si et seulement si f est injective.

Remarques

L'injectivité de f se traduit par : $\forall\,(x, y) \in E^2, f(x) = f(y) \Rightarrow x = y$. Autrement dit, deux arcs d'un graphe simple ne peuvent pas avoir mêmes sommets de départ et d'arrivée.

On comprend, par la remarque précédente, qu'il ne peut plus exister de doublons parmi les arcs (on s'intéresse au cas orienté). Ainsi, f devient superflue, à condition

que l'ensemble E exprime les arcs par des couples formés pour chacun d'eux de ses sommets de départ et d'arrivée.

Dans un graphe orienté simple, deux sommets peuvent être reliés par deux arcs sans rendre le graphe multiple. Cela suppose que les arcs en question soient en sens opposés.

2.1.3 Graphe vide

Un graphe (V, E, f) est vide si et seulement si $V = \emptyset$, autrement dit s'il n'a aucun sommet.

2.1.4 Graphe discret

Un graphe (V, E, f) est discret si et seulement si $E = \emptyset$, autrement dit s'il n'a aucun arc.

2.2 Éléments constitutifs

2.2.1 Sommets

Définition

Un sommet est un noeud dans un graphe. Il est représenté par un symbole, ce qui lui permet d'être manipulé.

Degrés

Le degré entrant (respectivement sortant) d'un sommet donné est égal au nombre d'arcs ayant ce dernier pour sommet d'arrivée (respectivement départ). Le degré d'un sommet donné est égal à la somme de ses degrés entrant et sortant.

Source et puits

Un sommet est dit puits lorsque son degré sortant est nul.

Un sommet est dit source lorsque son degré entrant est nul.

Successeurs

L'ensemble des successeurs d'un sommet v dans un graphe simple orienté est l'ensemble des sommets d'arrivée des arcs ayant v pour sommet de départ.

2.2.2 Arcs

Définition

Le terme d'arc s'applique à un graphe orienté et désigne une connexion entre deux sommets de ce graphe. Un arc est orienté de manière unique d'un sommet de départ vers un sommet d'arrivée. Le couple formé de ses derniers lui sert de représentation.

Boucle

Une boucle, dans un graphe orienté, est un arc ayant mêmes sommets de départ et d'arrivée.

Transposé

Le transposé d'un arc e est l'arc ${}^te$ dont le sommet de départ (respectivement d'arrivée) est le sommet d'arrivée (respectivement de départ) de e.

2.2.3 Chemins

Un chemin de longueur n est un n-uplet de sommets du graphe tel que pour chaque paire de sommets consécutifs, il existe un arc les reliant.

2.3 Opérations sur les graphes simples orientés

2.3.1 Initialisation

L'initialisation d'un graphe simple orienté revient à exhiber le triplet (V, E, f) tel que $V = \emptyset$ et $E = \emptyset$, f étant telle que définie précédemment.

2.3.2 Ajout d'un sommet

L'ajout d'un sommet v à un graphe simple orienté (V, E, f) revient à remplacer V par l'ensemble $V \cup \{v\}$.

2.3.3 Ajout d'un arc

L'ajout d'un arc e à un graphe simple orienté (V, E, f) revient à remplacer E par l'ensemble $E \cup \{e\}$.

2.3.4 Transposition

Le graphe transposé d'un graphe orienté (V, E, f) est le graphe orienté (V, E', f) tel que $E' = \{{}^te \mid e \in E\}$

2.4 Forte connexité

2.4.1 Remarques préalables

La notion de connexité s'applique soit à un ensemble de sommets d'un graphe, soit à un graphe tout entier.

On parle de forte connexité dans le cas des graphes orientés, qui sont les seuls à nous intéresser dans le cadre de ce projet.

2.4.2 Sommets fortement connexes

Un ensemble de sommets d'un graphe est fortement connexe si et seulement si il existe, pour chaque couple de sommets de cet ensemble, un chemin reliant ses derniers à l'intérieur du graphe restreint à cet ensemble de sommets.

2.4.3 Graphe fortement connexe

Un graphe orienté est fortement connexe si et seulement si l'ensemble de ses sommets est fortement connexe.

2.4.4 Composante fortement connexe

Une composante fortement connexe d'un graphe orienté est un ensemble de sommets maximal selon l'inclusion à être fortement connexe. En d'autres termes, un tel ensemble ne peut pas se voir ajouter de sommet sans perdre sa forte connexité.

Chapitre 3

Analyse

3.1 Modélisation et type de graphe

Une modélisation par un graphe est tout à fait adaptée à la résolution du problème posé. Voyons pourquoi il en est ainsi et intéressons-nous au type de graphe à adopter.

3.1.1 Pertinence de l'objet graphe

Les données du problème fournissent deux ensembles : des formats de fichiers ($\mathscr{F}$) et des programmes de conversion ($\mathscr{C}$). Chaque programme de conversion met en relation deux formats de fichiers.

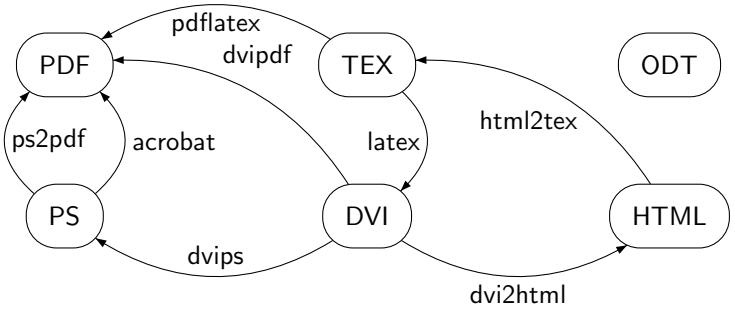
Exemple conducteur :

$$\mathscr{F} = \{\mathsf{TEX}, \mathsf{DVI}, \mathsf{PS}, \mathsf{PDF}, \mathsf{HTML}, \mathsf{ODT}\}$$

$$\mathscr{C} = \left\{\mathsf{latex}^{\mathsf{DVI}}_{\mathsf{TEX}}, \mathsf{pdflatex}^{\mathsf{PDF}}_{\mathsf{TEX}}, \mathsf{dvips}^{\mathsf{PS}}_{\mathsf{DVI}}, \mathsf{ps2pdf}^{\mathsf{PDF}}_{\mathsf{PS}}, \mathsf{acrobat}^{\mathsf{PDF}}_{\mathsf{PS}}, \mathsf{dvi2html}^{\mathsf{HTML}}_{\mathsf{DVI}}, \mathsf{html2tex}^{\mathsf{TEX}}_{\mathsf{HTML}}\right\}$$

On peut donc construire un graphe dont les sommets, définis par l'ensemble des formats de fichiers, sont reliés par des programmes de conversion.

Exemple conducteur :



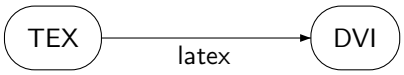
De plus, les questions posées se ramènent à des problèmes classiques de la théorie des graphes, développés dans la partie de reformulation du problème.

3.1.2 Choix d'un type de graphe

Orientation

Un programme de conversion convertit un unique format, appelé format de départ, dans un unique format, appelé format d'arrivée. Il faut donc opter pour un graphe orienté.

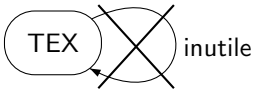
Exemple conducteur :



Absence de boucle

On suppose que les convertisseurs inutiles sont écartés, c'est à dire que les formats de départ et d'arrivée sont distincts. Il faut donc opter pour un graphe sans boucle.

Exemple conducteur :



Simplicité

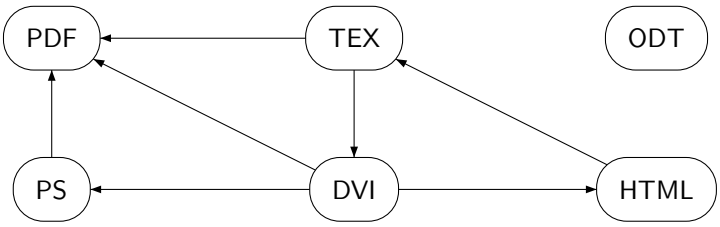
En pratique, il pourrait exister des convertisseurs équivalents en termes de fonctionnalité (effectuant la même conversion) mais qui diffèrent en termes de caractéristiques (prix, performances, plateformes, etc.).

Dans le cas présent, seule la fonctionnalité importe, si bien que deux convertisseurs équivalents ne font qu'un. Il faut donc opter pour un graphe simple.

Exemple conducteur :

On remplace $\mathsf{convertisseur}^{\mathsf{ARRIVÉE}}_{\mathsf{DÉPART}}$ par le couple $(\textsc{départ}, \textsc{arrivée})$.

$\mathscr{C} = \{(\textsc{tex}, \textsc{dvi}), (\textsc{tex}, \textsc{pdf}), (\textsc{dvi}, \textsc{ps}), (\textsc{ps}, \textsc{pdf}), (\textsc{dvi}, \textsc{html}), (\textsc{html}, \textsc{tex})\}$



3.1.3 Conclusion

En conclusion, un graphe orienté simple et sans boucle est donc parfaitement adpaté au problème posé. Ses sommets représentent les formats de fichiers et ses arcs modélisent les convertisseurs.

3.2 Reformulation du problème

Soient deux formats de fichiers. Nous devons déterminer les convertisseurs pour passer d'un format à l'autre. En terme de graphe, il faut déterminer les arcs tels qu'on puisse visiter n'importe quel sommet à partir de n'importe quel sommet.

On voit clairement apparaître la notion de forte connexité.

3.2.1 Forte connexité

Nous devons dans un premier temps décider si le graphe fourni est déjà fortement connexe. Nous allons donc rechercher toutes les composantes fortements connexes du graphe.

Dans le cas où le graphe n'est pas fortement connexe c'est à dire que l'on trouve plus d'une composante alors on doit déterminer le nombre d'arcs et les arcs à rajouter.

Bien qu'une méthode analogue à un algorithme de clôture transitive des automates existe, c'est la méthode du double parcours qui a été retenue pour des raisons de complexité.

- On parcours le graphe afin de mettre en liste les sommets dans l'ordre chronologique inverse.
- On transpose le graphe.
- On relance un parcours sur le graphe transposé à partir du dernier sommet visité (premier élément de la liste obtenu avec le premier parcours). Si tous les sommets n'ont pas été visités alors on relance un parcours à partir du deuxième sommets et ainsi de suite.

On va parcourir le graphe et mettre en liste l'ensemble des composantes fortement connexes. À ce stade, nous pouvons savoir en testant la cardinalité de la partition des CFC si le graphe est fortement connexe. S'il est connexe alors nous pouvons nous arrêter sinon nous allons passer à l'étape de complétion du graphe.

3.2.2 Connexion universelle

Dans cette seconde partie, on s'intéresse au cas où le graphe n'est pas fortement connexe. On souhaite alors ajouter un minimum de convertisseurs de manière à rendre le jeu de formats universellement convertible. Dans la modélisation du problème par la théorie des graphes, cela revient à rendre le graphe fortement connexe par ajout d'un nombre minimal d'arcs.

Graphe quotient

Pour améliorer l'efficacité des fonctions de traitement, on peut passer au graphe quotient. Chaque composante connexe sera représentée par un sommet, on réduit

donc le nombre de sommets et d'arcs dans le graphe. Cela rendra les parcours du graphe plus rapide.

Dans le graphe de départ, les sommets d'une composante fortement connexe peuvent avoir des successeurs qui n'appartiennent pas à cette dernière. Ce seront les arcs du graphe quotient. Il faut donc les chercher afin de construire le graphe quotient.

Le graphe que l'on va maintenant étudier est le graphe quotient qui est acyclique, simple, orienté et sans boucle. Il faut le rendre fortement connexe en rajoutant un minimum d'arcs.

Le nombre d'arcs

Pour le problème de la complétion, nous allons construire le graphe biparti constitué des sommets sources et des sommets puits du graphe quotient. Quand le graphe sera fortement connexe, nous ne pourrons plus définir de sommets sources ou de sommets puits. Il faut donc rajouter des arcs pour que les sommets particuliers (sources et puits) perdent ce statut.

Parmi les arcs qu'il faut rajouter, les arcs doivent obligatoirement être entrant du point de vue des sommets sources et sortant du point de vue des sommets puits.

On peut alors remarquer que le nombre d'arcs à ajouter est le maximum entre le nombre de sommets puits et le nombre de sommets source.

Détermination des arcs

Le problème reste maintenant à savoir comment définir les arcs à ajouter. Dans un premier temps nous pensions qu'il fallait choisir les puits en commençant d'abord par les sommets isolés puis en terminant par les sommets puits universels. Cependant, avec l'utilisation du graphe biparti, le choix des sommets s'avère inutile.

Nous prenons donc toujours le premier élément de la liste des sommets puits jusqu'à ce qu'on retombe sur une liste vide. Pour compléter le graphe, il faut choisir le sommet source tel qu'il n'existe pas de chemin entre le sommet source et le sommet puit.

Ensuite, nous définissons l'arc que l'on sauve dans la liste de successeurs.

Chapitre 4

Algorithmes relatifs à la forte connexité

4.1 Conventions

Graphe simple orienté sans boucle : n=m avec n le nombre de sommets et m le nombre d'arcs.

nCFC : nombre de composantes fortement connexes

CFC : composante(s) fortement connexe(s)

4.2 Recherche des composantes fortement connexes

4.2.1 Parcours du graphe

Fonction principale

Entrée : graphe orienté simple
Sortie : liste de sommets triée dans l'ordre décroissant de coloration en noir

Algorithme 1 Algorithme Fonction Graphe_Parcours

Entrées: G : graphe.
Sortie: Liste sommets
S_gris ← LS_vide()
S_noirs ← LS_vide()
S_noirs ← Sommets_Parcours(G,G.sommets, S_noirs, S_gris)
LS_supprimer(S_gris)
retourner (S_noirs)

Complexités :

Le calcul des compléxités de cette algorithme donne :

Temps : $\Theta(n)$ (linéaire au nombre de sommets)
Espace : $\Theta(n)$ (linéaire au nombre de sommets)

Preuve de terminaison :

Axiome : On suppose que :

La fonction `Sommets_Parcours` se termine.

⇒La fonction `Graphe_parcours` se termine.

Preuve de l'algorithme :

Axiome : On suppose que :
La fonction `Sommets_Parcours` est correcte.

⇒La fonction `Graphe_Parcours` est donc correcte

4.2.2 Algorithme de parcours du graphe

Algorithme

Entrée : graphe simple orienté, liste de sommets du graphe, liste vide qui contiendra les sommets parcourus, liste des sommets parcourus **Sortie :** liste de sommets parcourus (coloriés en noir)

Algorithme 2 Algorithme Fonction Sommets_Parcours

```
Entrées: G : graphe ; S : liste ; S_noirs : liste ; S_gris : liste
Sortie: Liste
  Si ! LS_estVide(S) Alors
    v ← LS_premier(S)
    Si ! LS_contient(S_gris, v) Alors
      S_gris ← LS_ajouter(S_gris, v)
      successeurs ← G_successeurs(graphe, v)
      Sommets_Parcours (G, successeurs, S_noirs, S_gris)
      LS_ajouter(S_noirs, v)
    FinSi
    S_noirs ← Sommets_parcours(graphe, LS_suivant(S), S_noirs, S_gris)
  FinSi
  Retourner S_noirs
```

Remarque (implémentation) :

On ajoute en début de liste pour que la fonction `ajouter` ait une complexité en temps constante.

Complexités :

L'algorithme de parcours ne traite pas deux fois les mêmes sommets.

Temps : $\Theta(n)$ (linéaire au nombre de sommets)
Espace : $O(n)$ (linéaire au nombre de sommets)

Preuve de terminaison :

La liste des sommets noirs permet de prouver la terminaison de l'algorithme. L'algorithme s'arrête lorsque les sommets de la liste passée en argument sont colorés en gris ou noir. Comme aucun sommet (noir ou gris) n'est coloré en blanc alors

on colorie tout les sommets.

L'ensemble bien fondé est la cardinalité de la liste S qui décroît à chaque appel de la fonction.

Preuve de correction (par récurrence) :

Initialisation : Supposons que le graphe n'est composé que d'un sommet. Alors on le choisit puis on appelle encore la fonction pour colorier ses successseurs (liste vide). Il devient noir. On colorie ensuite les sommets qui sont après S dans la liste des sommets de G. La liste est vide, la fonction se termine.

Hypothèse : On remonte les appels récursifs ("rang fixé"), les sommets parcourus sont dans la liste classée dans l'ordre chronologique inverse.

Deux cas se présentent. Il existe S un sommet ascendant, les sommets que nous venons de mettre en liste sont ses successeurs. Dans ce cas là, on le colorie en noir et on le met en liste. On conserve l'ordre chronologique inverse.

Soit il n'exite pas S, sommet ascendant aux sommets déjà en liste. Dans ce cas, on lance le parcours à partir d'un sommet (dans la liste des sommets à visiter).

4.2.3 Transposition du graphe

Nous avons à ce stade la liste des sommets (dans l'ordre inverse, comme évoqué précédemment). Le graphe a été transposé par la fonction `transpose`. Le prototype de cette fonction est :

```
fonction G_transposer (G : graphe) -> graphe
```

Elle ne s'applique qu'au graphe orienté. Sa complexité temporelle et sa complexité spatiale sont linéaires au nombre d'arcs.

4.2.4 Parcours du graphe transposé

On va parcourir le graphe et mettre en liste l'ensemble des composantes fortement connexes.

Fonction principale

Entrée : graphe simple orienté (transposé), liste de sommets à parcourir triée
Sortie : liste des composantes fortement connexes

Algorithme 3 Algorithme Fonction Graphe_transpose_Parcours

Entrées: G : graphe, S_noirs : liste
Sortie: Partition
  C_connexe = P_vide()
  S_gris = LS_vide() ;
  C_connexe ← Composantes_parcours(G, S_noirs, S_gris, C_connexe)
  Retourner (C_connexe)

Compléxités :

Temps : $\Theta(n)$ (linéaire au nombre de sommets)
Espace : $O(n)$

Preuve de terminaison :

Axiome : On suppose que :

La fonction `Composante_Parcours` se termine
La fonction `Graphe_transpose_Parcours` se termine

Preuve de correction

Axiome : `Composante_Parcours` est correcte

⇒La fonction `Graphe_transpose_Parcours` est correcte

Algorithme de parcours du graphe transposé

Entrée : graphe simple orienté, liste des sommets à parcourir, la liste des sommets parcourus, une liste qui contiendra les composantes fortement connexes

Sortie : partition des composantes fortement connexes

Algorithme 4 Algorithme Fonction Composantes_Parcours

```
Entrées: G : graphe, S_noirs : liste, S_gris : liste, C_connexe : Partition
Sortie: Partition
  L ← LS_vide()
  Si ! LS_estVide(S_noirs Alors
    v ← LS_premier(S_noirs)
    Si ! LS_contient(S_gris, v) Alors
      L ← LS_vide()
      L ← LS_ajouter(L, v)
      L ← Sommets_parcours(G, G_successeurs(G, v), L, S_gris)
      C_connexe ← P_ajouter(C_connexe, L)
    FinSi
    C_connexe ← Composantes_parcours(G, LS_suivant(S_noirs), S_gris,
    C_connexe)
  FinSi
  Retourner(C_connexe)
```

Remarque :
LS_ajouter doit être en temps constant (ajout d'une liste en tête de liste)

Calcul de compléxités :

Temps : $\Theta(n)$
Espace : $\Theta(n)$

Preuve de terminaison :

Axiome : `Sommets_Parcours` se termine.

La fin de la fonction est liée à l'existence de la liste S_gris. On traite que les sommets blancs. Dans une liste de sommets, supposons que tous les sommets sont noirs ou gris.

`Composante_Parcours` va se rappeller avec en argument une liste de sommets plus petite (privée de l'élément courant). Quand la liste est vide alors on remonte les appels récursifs et la fonction s'arrête.

Dans la liste des sommets gris, on ne fait que rajouter des sommets. L'ensemble des sommets blancs ne peut que diminuer.
La fonction se termine.

Preuve de correction :

Axiome : `Sommets_Parcours` est correcte.

D'après le cours sur les graphes, le double parcours nous permet de déterminer les composantes fortement connexes.

Nous avons une liste de sommets à parcourir (classée dans l'ordre chronologique inverse). Pour chaque sommet qui n'a pas été déjà visité (colorié), on lance un parcours à partir de ce dernier afin de mettre en liste ses successeurs. On admet qu'un sommet (s'il n'a pas de successeur) est une composante fortement connexe.

Cette liste sera une composante fortement connexes. On met en liste cette liste. Puis on rappelle la fonction avec pour argument la liste privé du sommet que l'on vient de traiter.

Graphe est-il connexe ?

Algorithme 5 Algorithme Fonction Graphe_est_connexe

Entrées: C_connexe : partition des composantes fortement connexe
Sortie: entier
Retourner(P_taille(C_connexe) = 1)

Chapitre 5

Algorithmes relatifs à la conversion universelle

5.1 Réduction des composantes fortement connexes

Nous avons une liste des composantes fortement connexes. La prochaine étape consiste à construire le graphe quotient. Chaque composante fortement connexe sera remplacée par un sommet. Ce sommet sera appelé représentant.

On a deux fonctions. La première va générer les associations entre les sommets du graphe et le représentant de la composante fortement connexe à laquelle ils appartiennent.

5.1.1 Représentant d'un sommet

Entrée : un sommet et la liste des couples (représentant sommets)
Sortie : un sommet, le représentant du sommet passé en argument

```
fonction Sommet_representant (G : graphe, s : sommet) -> sommet
```

Explications :

Cette fonction pour un sommet passé en argument donne le représentant de la composante fortement connexe auquel il appartient.

5.2 Construction du graphe quotient

5.2.1 Description du problème

Nous rappelons que l'objectif est de déterminer quels arcs il faut rajouter pour que le graphe soit fortement connexe.

Pour améliorer l'efficacité des fonctions de traitement, on peut passer au graphe quotient. Chaque composante connexe sera représentée par un sommet, on réduit donc le nombre de sommets et d'arcs dans le graphe. Cela rendra les parcours du graphe plus rapide.

5.2.2 Successeurs aux composantes fortement connexes

Dans le graphe de départ, les sommets d'une composante fortement connexe peuvent avoir des successeurs qui n'appartiennent pas à cette dernière. Ce seront les arcs du graphe quotient. Il faut donc les chercher afin de construire le graphe quotient.

Fonction principale

Cette fonction doit pour une composante fortement connexes déterminer tous les successeurs à cette composante. On regarde alors les successeurs des sommets de cette composante qui n'appartiennent pas à cette dernière.

On donne deux listes en argument : la première liste donne les sommets à traiter, à chaque traitement on traite l'élément suivant. La seconde liste de sommets sert de référence pour tester l'appartenance (elle ne sera pas modifiée).

Entrée : graphe simple orienté, liste de sommets à parcourir (x2), une liste qui contiendra pour les sommets parcourus leur successeurs (sauf ceux qui appartiennent à `CFC_Sommets`).
Sortie : une liste de sommets (successeurs)

Algorithme 6 Algorithme Fonction Composante_Successeurs

Entrées: G : graphe, Sommets : liste, CFC_Sommets : liste, Successeurs : liste
Sortie: Liste
  Si !LS_estVide(Sommets) **Alors**
    V ← LS_premier(Sommets)
    Successeurs ← AjoutRepresentant(Successeurs, G_successeurs(G, v), CFC_Sommets)
    Successeurs ← Composante_Successeurs(G, LS_suivant(Sommets), CFC_Sommets, Successeurs)
  FinSi
  Retourner (Successeurs)

Calcul de compléxités :

Temps : $\Theta(n)$
Espace : $O(n)$

Preuve de terminaison :

Axiome : La fonctions `AjoutRepresentant` se termine.

Cette fonction est appelé avec en argument la liste des sommets à parcourir pour déterminer les successeurs d'une composante connexe. Chaque appel entraîne un appel de `Composante_Successeurs` avec en argument la liste des sommets à traiter privée du sommet que l'on vient de traiter.

La cardinalité de cette liste est une suite décroissante dans $\mathbb{N}$. Le programme se termine.

Preuve de correction :

Axiome : La fonction `AjoutRepresentant` est correcte.

On doit traiter tous les sommets pour chercher tous les successeurs à la composante connexe. La fonction s'assure que l'on traite tous les sommets de la CFC.

A un instant donné, on a traité un certain nombre de sommets qui composent la CFC. Successeurs contient les successeurs de la CFC. Donc, la fonction est appelée avec en argument la suite de la liste des sommets à traiter. La liste n'est pas vide, le sommet n'a pas été traité. La fonction `AjoutRepresentant` met en liste les successeurs à ce sommet. On peut donc traiter le sommet suivant.

⇒La fonction est correcte.

Ajout des successeurs

Entrée : liste qui contiendra les successeurs de la CFC, liste des sommets à traiter, la liste référence des sommets de la CFC.
Sortie : liste des successeurs à la CFC

Remarque :

On considère que ajouter à une complexité en temps qui est constante, elle ne vérifie pas si l'élément à ajouter existe déjà. D'où l'existence d'un test en amont pour éviter les doublons.

Algorithme 7 Algorithme Fonction AjoutRepresentant

Entrées: Successeurs : liste sommets, A_Ajouter : liste de sommets, CFC_Sommets : liste de sommets
Sortie: Liste
  Si !LS_estVide(A_Ajouter) **Alors**
    v ← LS_premier(A_Ajouter)
    Si !P_appartient(CFC_Sommets, v) et !P_appartient(Successeurs, Sommet_representant(G, v)) **Alors**
      Successeurs ← LS_ajouter(Successeurs, Sommet_representant(G, v))
    FinSi
  FinSi
  Retourner(Successeurs)

Calcul de compléxités :

Temps : $\Theta(n)$
Espace : $O(n)$

Preuve de terminaison :

Axiome : La fonction `Representant` se termine.

L'ensemble bien fondé est lié `A_Ajouter`. La cardinalité de cette liste est dans $\mathbb{N}$. Si on considère l'ensemble des appels récursifs, nous pouvons construire une suite dans $\mathbb{N}$ décroissante à partir de la cardinalité de la liste (`A_Ajouter`). $T_{n+1} = T_n - 1$.

⇒La fonction se termine.

Preuve de correction :

Axiome : La fonction `Representant` est correcte

Les successeurs qui sont en listes sont des composantes fortement connexes. Les composantes connexes seront dans le graphe quotient représenté par des sommets dont le nom est celui du représentant de la composante.

Le sommet j est successeur de i (i et j représentant de deux composantes fortement connexes) s'il existe un arc entre un sommet de la CFC représentée par i vers un sommet de la CFC représentée par j.

Si on trouve un successeur, on regarde s'il n'est pas déjà dans la liste car le graphe quotient doit être simple, orienté et sans boucle.

En parcourant tous les sommets d'une CFC, on met en liste tous les représentants des CFC successeurs.

5.2.3 Le graphe quotient

Fonction principale

Entrée : graphe simple orienté, liste des composantes connexes.
Sortie : structure de listes (liste de successeurs et liste de sommets)

Algorithme 8 Algorithme Fonction Graphe_quotient

Entrées: G : graphe
Sortie: graphe
C_connexe ← G_transpose_parcours(G_transposer(G), Graphe_parcours(G))
G' ← G_vide()
G'.sommets ← Graphe_quotient_sommets(G, C_connexe)
G'.arcs ← Graphe_quotient_arcs(G, C_connexe)
Retourner (G')

Calcul de complexités :

Temps : $\Theta(n + nCFC)$
Espace : $\Theta(n)$

Preuve de terminaison :

Axiome : Les fonctions `Graphe_quotient_sommets` et `Graphe_quotient_arcs` se terminent.

⇒La fonction `Graphe_quotient` se termine

Preuve de correction :

Axiome : Les fonctions `Graphe_quotient_sommets` et `Graphe_quotient_arcs` sont correctes.
⇒La fonction `Graphe_quotient` est correcte.

Détermination de l'ensemble des sommets

Entrée : liste vide qui contiendra les sommets représentants des CFC, la liste des CFC
Sortie : liste des sommets représentants

Algorithme 9 Algorithme Fonction Graphe_quotient_sommets

Entrées: G : graphe, Comp_Connex : liste de listes
Sortie: liste de sommets
  Si !P_estVide(Comp_Connex) **Alors**
    quotient_sommets ← LS_vide()
    L ← P_premier(Comp_Connex)
    v ← Sommet_representant(G, LS_premier(L))
    quotient_sommets ← LS_ajouter(quotient_sommets, v)
    quotient_sommets ← Graphe_quotient_sommets(quotient_sommets, P_suivant(Comp_Connex))
  FinSi
  Retourner (quotient_sommets)

Calcul de compléxités :

Temps : $\Theta(n + nCFC)$
Espace : $\Theta(n)$

Preuve de terminaison :

Axiome : La fonction `Representant` se termine.

L'ensemble bien fondé est la cardinalité de la liste `Comp_Connex`. Chaque appel récursif de la fonction fait décroitre la taille de la liste de 1. Nous sommes dans $\mathbb{N}$, nous avons donc une suite décroissante dans $\mathbb{N}$.

⇒L'algorithme se termine.

Preuve de correction :

Axiome : La fonction `Representant` est correcte.

Les sommets du graphe quotient sont les composantes fortement connexes. C'est la fonction `représentant` qui va permettre d'associer un sommet à une CFC.

On parcourt toutes les CFC et on met en liste un représentant par CFC.

Construction de l'ensemble des arcs

Entrée : graphe simple, orienté, liste vide qui contiendra les listes (sommets successeurs), liste des CFC
Sortie : liste de listes (sommets successeurs)
Calcul de complexités :

Temps : $\Theta(n + nCFC)$
Espace : $\Theta(n + nCFC)$

Algorithme 10 Algorithme Fonction Graphe_quotient_Arcs

Entrées: G : graphe, quotient_arcs : liste, Comp_connex : Partition
Sortie: liste
  Si !P_vide(Comp_Connex) **Alors**
    L ← LS_premier(Comp_Connex)
    L' ← LS_vide()
    v ← Sommet_representant(G, LS_premier(L))
    Successeurs ← LS_vide()
    Successeurs ← Composante_Successeurs(G, L, L, Successeurs)
    L' ← LS_ajouter(L', v)
    L' ← LA_ajouter(L', Successeurs)
    quotient_arcs ← LA_ajouter(quotient_arcs, L')
    quotient_arcs ← Graphe_quotient_arcs(G, quotient_arcs, P_suivant(Comp_connex))
  FinSi
  Retourner (quotient_arcs)

Preuve de terminaison :

Axiome : La fonction `Composante_Successeurs` se termine.

La preuve de terminaison est analogue à celle de l'algorithme `Graphe_quotient_sommets`. L'ensemble bien fondé est la cardinalité de la liste `Comp_connex`.

Preuve de correction :

Axiome : La fonction `Composante_Successeurs` est correcte.

La représentation choisie pour représenter les arcs est `(a (bcd))` où `a` est le sommet source et `(bcd)` est la liste des successeurs.

Dans un premier temps, on met en liste les représentants des CFC successeurs. Ensuite, on met en liste le représentant de la CFC en cours de traitement avec la liste des successeurs.

Enfin, on met en liste, la liste que l'on vient de créer puis on traite la CFC suivante.

Le graphe obtenu est simple, orienté et sans boucle.

5.3 Algorithme de complétion

Algorithme 11 Algorithme Fonction Rendre_Connexe

Entrées: G : graphe
Sortie: Liste d'arcs
  sources ← Source_graphe_quotient(G)
  puits ← Puits_graphe_quotient(G)
  arcs_biparti ← Assoc_source_puits(G)
  LA ← Definir_arcs(sources, puits, arcs_biparti, LA_vide())
  Retourner(LA)

5.4 Fonctions auxiliaires

Algorithme 12 Algorithme Fonction Definir_arcs

Entrées: sources, puits : listes sommets ; arcs_biparti, LA : listes arcs
Sortie: liste arcs
  Si !LS_estVide(puits) et !LS_estVide(sources) **Alors**
    puit ← Choisir_puit(puits)
    source ← Choisir_source(arcs_biparti, sources, puit)
    LA ← Construire_arcs(source, puit, sources, puits, arcs_biparti, LA)
  FinSi
  Retourner(LA)

Algorithme 13 Algorithme Fonction Choisir_puit

Entrées: puits : liste sommets
Sortie: sommet
  Retourner(LS_premier(puits))

Algorithme 14 Algorithme Fonction Choisir_source

Entrées: arcs_biparti : liste arcs ; sources : liste sommets ; s : sommet
Sortie: sommet
  Si !LS_estVide(sources) **Alors**
    v ← LS_premier(sources)
    L ← P_contient(arcs_biparti, v)
    Si LS_contient(L, s) et !LS_estVide(LS_suivant(sources)) **Alors**
      u ← Choisir_source(arcs_biparti, LS_suivant(sources, s))
    Sinon
      u ← LS_premier(sources)
    FinSi
    Retourner(u)
  FinSi
  Retourner(s_sentinelle)

s_sentinelle est la sentinelle, valeur particulière lorsque nous ne pouvons retourner de sommets.

5.5 Manipulation de la liste d'arcs du graphe biparti

5.6 Sources et puits du graphe quotient

Algorithme 15 Algorithme Fonction Construire_arcs

Entrées: s, t : sommet ; sources, puits : liste sommets ; arcs_biparti, arcs_ajoutes : liste arcs
Sortie: Liste arcs
  sources ← Supprimer_sommet_liste(s, sources, puits)
  puits ← Supprimer_sommet_liste(t, puits, sources)
  arcs_biparti ← Actualise_assoc(arcs_biparti, s, t)
  arcs_ajoutes ← Ajouter_arcs(arcs_ajoutes, t, s)
  arcs_ajoutes ← Definir_arcs(sources, puits, arcs_biparti, arcs_ajoutes)
  Retourner(arcs_ajoutes)

Algorithme 16 Algorithme Fonction Supprimer_sommet_liste

Entrées: s : sommet ; L : liste sommets ; LP : liste sommets
Sortie: liste sommets
  Si LS_estVide(LS_suivant(LP)) ou !LS_estVide(LS_suivant(L)) **Alors**
    Retourner(LS_retirerSommet(L, s))
  Sinon
    Retourner(L)
  FinSi

Algorithme 17 Algorithme Fonction Graphe_nbArcs_ajoutes

Entrées: G : graphe quotient
Sortie: entier naturel
  Si Graphe_est_connexe(G) **Alors**
    n ← 0
  Sinon
    a ← LS_taille(Source_graphe_quotient(G))
    b ← LS_taille(Puits_graphe_quotient(G))
    n ← max(a, b)
  FinSi
  Retourner(n)

Algorithme 18 Algorithme Fonction Assoc_source_puits

Entrées: G : graphe quotient
Sortie: Liste arcs
  sources ← Source_graphe_quotient(G)
  liste_assoc ← Gen_liste_assoc(G, sources, LA_vide())
  Retourner(liste_assoc)

Algorithme 19 Algorithme Fonction Gen_liste_assoc

Entrées: G : graphe quotient ; sources : liste sommets ; liste_assoc : liste arcs
Sortie: Liste arcs

```
Si !LS_estVide(sources) Alors
  s ← LS_premier(sources)
  L_parcours ← Sommets_parcours(G, LS_ajouter(LS_vide(), s), LS_vide(),
  LS_vide())
  puits ← Puits_graphe_quotient(G)
  L ← puits_associer(L_parcours, puits, LS_vide)
  L ← LS_ajouter(L, s)
  liste_assoc ← LA_ajouter(liste_assoc, L)
  liste_assoc ← Gen_liste_assoc(G, LS_suivant(sources), liste_assoc)
FinSi
Retourner(liste_assoc)
```

Algorithme 20 Algorithme Fonction Actualise_assoc

Entrées: assoc : partition ; source, puit : sommets
Sortie: partition

```
P1 ← P_contient(assoc, source)
P2 ← P_contient(assoc, puit)
P1 ← P_concatener(P1, P_suivant(P2))
Retourner(P1)
```

Algorithme 21 Algorithme Fonction Puits_associer

Entrées: sommets, puits, sommets_puits : liste sommets
Sortie: liste sommets

```
Si !LS_estVide(sommets) Alors
  s ← LS_premier(sources)
  L ← LS_suivant(sommets)
  Si LS_contient(puits, s) Alors
    sommets_puits ← Puits_associer(L, puits, LS_ajouter(sommets_puits, s))
  Sinon
    sommets_puits ← Puits_associer(L, puits, sommets_puits)
  FinSi
FinSi
Retourner(sommets_puits)
```

Algorithme 22 Algorithme Fonction Puits_graphe_quotient

Entrées: G : graphe quotient
Sortie: liste sommets

```
sommets ← LS_vide()
G-sommets ← G_sommets(G)
G-arcs ← G_arcs(G)
Retourner(Puits_liste(G-arcs, G-sommets, sommets))
```

Algorithme 23 Algorithme Fonction Puits_liste

Entrées: arcs : liste arcs ; sommets, puits : liste sommets
Sortie: liste sommets

```
Si !LS_estVide(sommets) Alors
  s ← LS_premier(sommets)
  L ← LS_suivant(sommets)
  Si !LA_contient(arcs, s) Alors
    puits ← Puits_liste(arcs, L, LS_ajouter(puits, s))
  Sinon
    puits ← Puits_liste(arcs, L, puits)
  FinSi
FinSi
Retourner(puits)
```

Algorithme 24 Algorithme Fonction Sources_graphe_quotient

Entrées: G : graphe quotient
Sortie: liste sommets

```
sommets ← LS_vide()
G' ← G_transposer(G)
G-sommets ← G_sommets(G')
G-arcs ← G_arcs(G')
Retourner(Puits_liste(G-arcs, G-sommets, sommets))
```

Chapitre 6

Types abstraits de données

Après avoir mis en place les algorithmes résolvant les deux problèmes, voici un récapitulatif des types abstraits utilisés définis par leurs primitives.

6.1 T.A.D. Liste de Sommets (LS)

Liste de Sommets

–	vide	:		→	liste de sommets
–	estVide	:	liste de sommets	→	booléen
–	ajouter	:	liste de sommets × sommet	→	liste de sommets
–	retirer	:	liste de sommets × sommet	→	liste de sommets
–	contient	:	liste de sommets × sommet	→	booléen
–	premier	:	liste de sommets	→	sommet
–	suivant	:	liste de sommets	→	liste de sommets
–	concaténer	:	liste de sommets × liste de sommets	→	liste de sommets
–	taille	:	liste de sommets	→	entier naturel

Remarque : Afin de ne pas alourdir le jeu de T.A.D., nous considérerons qu'un sommet se réduit à sa représentation. Ce peut être un simple entier, un caractère ou encore un mot, selon les commodités du language utilisé. Ainsi, il ne sera en aucun cas défini de type abstrait Sommet à ce niveau d'abstraction.

6.2 T.A.D. Liste d'Arcs (LA)

Liste d'Arcs

–	vide	:		→	liste d'arcs
–	estVide	:	liste d'arcs	→	booléen
–	ajouter	:	liste d'arcs × sommet × sommet	→	liste d'arcs
–	contient	:	liste d'arcs × sommet × sommet	→	booléen
–	concaténer	:	liste d'arcs × liste d'arcs	→	liste d'arcs
–	premier	:	liste d'arcs	→	sommet × sommet
–	suivant	:	liste d'arcs	→	liste d'arcs
–	concaténer	:	liste d'arcs × liste d'arcs	→	liste d'arcs
–	taille	:	liste d'arcs	→	entier naturel

Remarque : Dans le même esprit que la remarque précédente sur la nature des sommets, il n'y a pas non plus de type abstrait Arcs. On considérera donc les arcs simplement comme des couples de sommets.

6.3 T.A.D. Partition (P)

Partition

–	vide	:		→	partition
–	estVide	:	partition	→	booléen
–	contient	:	partition × sommet	→	liste de sommets
–	ajouterListeSommets	:	partition × liste de sommets	→	partition
–	supprimerListeSommets	:	partition × liste de sommets	→	partition
–	premier	:	partition	→	liste de sommets
–	suivant	:	partition	→	partition
–	concaténer	:	partition × partition	→	partition
–	taille	:	partition	→	entier naturel

Remarque : Le type abstrait Partition n'a pas été pensé de manière à vérifier constamment les propriétes d'une partition. Ceci est en partie dû à des contraintes de temps car les T.A.D. n'ont été rigoureusement établis que tardivement dans l'avancement du projet. L'idée la plus rigoureuse consisterait à partir systématiquement d'une partition discrète et de procéder par unions successives. Mais ce mode de fonctionnement n'est par ailleurs pas adapté aux algorithmes. Il faut noter la particularité de la primitive `contient` qui renvoie l'ensemble vide quand le sommet n'appartient à auncun ensemble de la partition, ou l'ensemble contenant ce sommet le cas échéant.

6.4 T.A.D. Graphe (G)

Graphe

–	vide	:		→	graphe
–	estVide	:	graphe	→	booléen
–	créerDiscret	:	entier naturel	→	graphe
–	ajouterSommet	:	graphe × sommet	→	graphe
–	ajouterArc	:	graphe × sommet × sommet	→	graphe
–	sommets	:	graphe	→	liste de sommets
–	arcs	:	graphe	→	liste d'arcs
–	successeurs	:	graphe × sommet	→	liste de sommets
–	transposer	:	graphe	→	graphe

Remarque : La primitive créerDiscret appelée avec l'argument $n \in \mathbb{N}$ crée un graphe discret de n sommets numérotés de 1 à n.

6.5 Conventions de notation

On remarque que les T.A.D. ont des noms de primitives en commun, d'où la nécessité d'utiliser les préfixes suivants :

- LS pour « Liste de Sommets » : LS_vide, LS_ajouter, etc.
- LA pour « Liste d'Arcs » : LA_vide, LA_ajouter, etc.
- P pour « Partition » : P_vide, P_ajouterLS, P_supprimerLS, etc.

– G pour « Graphe » : G_vide, G_successeurs, G_transposer, etc.

Chapitre 7

Implémentation en Lisp

7.1 Représentation des graphes

La manipulation de graphes en Lisp nécessite qu'on s'intéresse à leur représentation. Trois solutions plus ou moins adaptées selon les situations ont été implémentées, ainsi qu'un moyen de passer d'une représentation à l'autre de façon transparente.

7.1.1 Structures en Lisp

Voyons tout d'abord comment se définissent et s'utilisent les structures en Lisp.

Définition

```
(defstruct ma_structure
  champ1 champ2)
```

Instanciation

```
(defparameter *instance*
  (make-ma_structure
   :champ1 1
   :champ2 'deux))
```

Accès aux champs

```
(ma_structure-champ1 *instance*) → 1
(ma_structure-champ2 *instance*) → DEUX
```

Prédicat de structure

Pour vérifier qu'une structure est du type `ma_structure`, on utilise le prédicat associé automatiquement : `ma_structure-p`.

7.1.2 Point commun entre les représentations

Dans les trois représentations, on définit une structure à deux champs. Le premier champ sommets est toujours le même, à savoir l'ensemble des sommets, stocké sous forme de liste (type naturel en Lisp) bien que l'ordre n'ait aucune importance. Seul le second champ, qui informe sur les arcs du graphe, varie d'une représentation à l'autre.

7.1.3 Représentation concrète par arcs (**RCA**)

Dans cette représentation, le graphe dispose d'un second champ arcs qui est la liste de ses arcs, chaque arc étant un couple de sommets. La structure associée s'appelle grRCA.

7.1.4 Représentation concrète par successeurs (**RCS**)

Dans cette représentation, le graphe dispose d'un second champ successeurs qui est une liste de couples hétérogènes dont le premier terme est un sommet et le second la liste de ses successeurs. La structure associée s'appelle grRCS.

7.1.5 Représentation abstraite par adjacence (**RAA**)

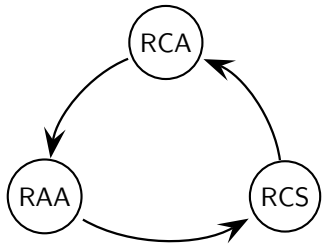
Dans cette représentation, le graphe dispose d'un second champ adjacence qui est la fonction ainsi définie :

$$\begin{array}{rccl} \text{adjacence} : & V^2 & \to & \mathbb{B} \\ & (s,t) & \mapsto & \begin{cases} \text{VRAI si } (s,t) \in E \\ \text{FAUX sinon} \end{cases} \end{array}$$

La structure associée s'appelle grRAA.

7.1.6 Passage d'une représentation à une autre

Il est amusant de noter que le problème du passage d'une représentation de graphe à une autre rejoint le problème général posé pas ce sujet. Ainsi, il suffit de trois convertisseurs pour permettre la conversion universelle entre les trois représentations proposées. Le graphe suivant illustre la solution adoptée :



Ainsi, seules trois fonctions de conversion ont été codées, tandis que les trois restantes ne sont que des compositions de deux convertisseurs précédents, ce qui n'est acceptable que dans la mesure où la complexité n'est pas un objectif prioritaire.

Concrètement, le fichier exemple-representations.lisp permet d'observer le bon fonctionnement des trois représentations et des six fonctions de conversion. Il fait appel à representations.lisp qui fait lui-même appel à representations.fonctions-annexes.lisp, par le biais de la commande load.

7.2 Implémentations des TAD

Le graphe peut avoir trois implémentations différentes lorsque nous regardons au niveau haut de l'implémentation dans le langage LISP. Cependant lorsque nous regardons l'implémentation bas niveau des graphes, c'est une structure avec deux listes.

Afin d'accomplir la complétion du graphe, il nous faut à la fois manipuler la liste de sommets et la liste d'arcs. Concrètement, la liste de sommets est une liste d'éléments simples alors que la liste d'arcs est une liste de listes d'éléments simples.

Un autre TAD à implémenter est le TAD graphe pour implémenter quelques primitives liées au graphe.

Comme nous avons choisit de représenter les partitions, ensemble d'ensembles, par une liste de listes, nous avons implémenter un TAD partition qui ressemble beaucoup à l'implémentation du TAD liste d'arcs (à bas niveau).

Nous avons fait la distinction entre les deux TAD puisque les objets que manipulent ces TAD sont de nature différente au sein des algorithmes.

Une liste de composantes fortement connexes n'est pas une liste d'arcs.

7.2.1 Liste de sommets

L'implémentation des primitives est faite dans le fichier TAD_sommet.lisp. Voici les prototypes des primitives :

```
LS_Vide : --> liste vide
LS_estVide : liste --> booléen
LS_ajouter : sommet x liste --> liste
LS_suivant : liste --> liste
LS_contient : sommet x liste --> booléen
LS_concatener : liste x liste --> liste
LS_retirerSommet : liste sommets x sommet --> liste sommets
LS_supprimer : liste sommets x liste sommets --> liste sommets
LS_premier : liste --> sommet
LS_taille : liste sommets --> entier
```

7.2.2 Liste d'arcs

L'implémentation des primitives est faite dans le fichier TAD_arc.lisp. Voici les prototypes des primitives :

```
LA_estVide : liste arcs --> booléen
LA_Vide : --> liste arcs vide
LA_contient : liste arcs de liste arcss x sommet -> liste arcs
LA_ajouter : liste arcs x liste arcs --> liste arcs
LA_concatener : liste arcs x liste arcs --> liste arcs
LA_premier : liste arcs --> sommet x sommet (arc)
LA_taille : liste arcs --> entier
```

7.2.3 Partition

L'implémentation des primitives est faite dans le fichier TAD_partition.lisp. Voici les prototypes des primitives :

```
P_estVide : partition --> booléen
P_Vide : --> partition vide
P_contient : partition x sommet -> partition
P_ajouterLS : partition x liste sommets --> partition
P_suivant : partition --> liste sommets
P_premier : partition --> liste sommets
```

```
P_concatener : partition x partition --> partition
P_supprimer : partition x liste de sommets --> partition
P_taille : partition --> entier
```

7.2.4 Graphe

L'implémentation des primitives est faite dans le fichier TAD_graphe.lisp. Voici les prototypes des primitives :

```
Sommet_representant : graphe x sommet --> sommet
G_successeurs : graphe x sommet --> liste de sommets
G_transposer : graphe --> graphe
G_sommets : graphe --> liste sommets
```

7.2.5 Implémentation du TAD graphe

Dans cette partie, nous allons passer sous silence l'implémentation des TAD liste de sommets, liste d'arcs et partition. Leur implémentation est simple, nous préférons laisser la compréhension de l'implémentation au lecteur via le code.

L'implémentation des primitives se retrouvent dans les fichiers TAD_graphe.lisp, graphe.fonctions-annexes.lisp et dans transpose.lisp.

À part pour la primitive G_transposer, les primitives peuvent prendre le graphe sous n'importe quelle représentation (abstraite, concrète par liste d'arcs et concrète par liste de successeurs). Au moyen d'un prédicat, nous vérifions la représentation et appelons le convertisseur si besoin est.

Pour la primitive G_successeur, nous devons parcourir la liste des successeurs jusqu'à ce que nous trouvons le bon sommet de départ. Nous retournons alors la liste à laquelle il est "associé".
Par exemple, si nous cherchons les successeurs de `b` dans la liste des successeurs qui est `((a (b c)) (b (e f)))` alors nous retournons la liste `(e f)`

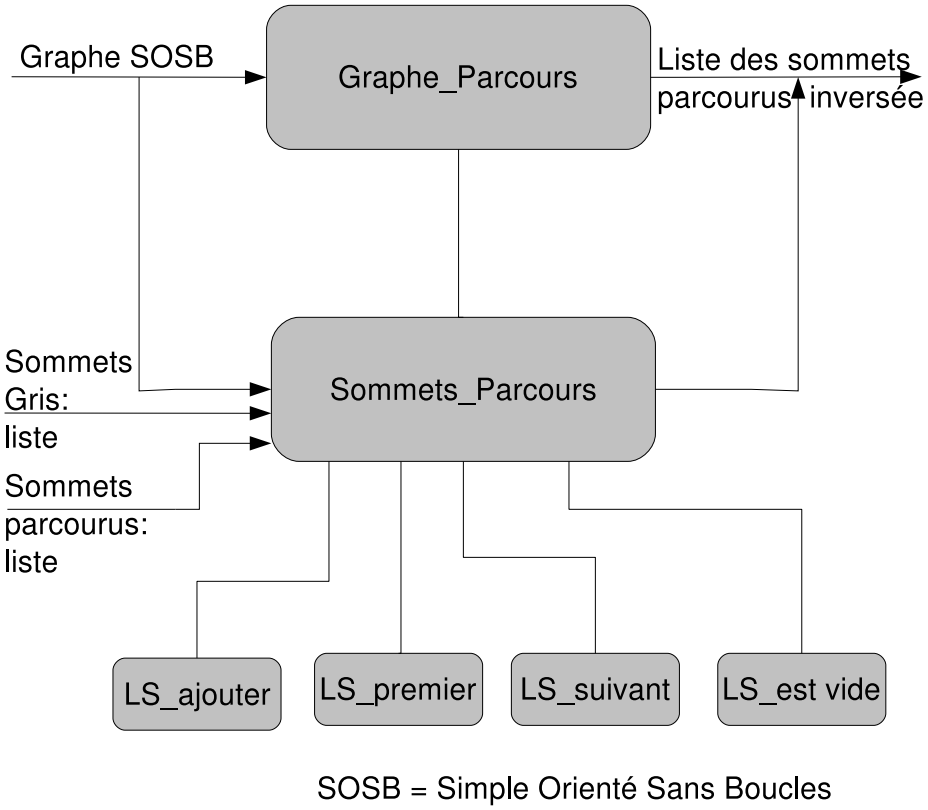
Pour la primitive G_transposer, une idée était d'utiliser la représentation concrète par liste d'arcs et d'inverser les éléments de cette liste. Pour des raisons de calculs, le graphe transposé étant utilisé à plusieurs endroits, nous avons créé une fonction qui calcule le graphe dans la représentation par liste de successeurs. On évite le passage par trois convertisseurs. Une des fonctions calcule pour chaque sommet l'ensemble des prédécesseurs pour chaque sommet, on a alors la liste de successeurs du graphe transposé.

Nous allons parler maintenant de la primitive Sommet_representant. L'implémentation de cette fonction est très simple puisque cela consiste à regarder à quelle composante fortement connexe le sommet appartient et de retourner le premier élément de la composante fortement connexe.

7.3 Implémentation de l'algorithme de forte connexité

Après avoir présenté l'implémentation des TAD qui seront utilisés dans nos algorithmes, nous allons présenté l'implémentations des algorithmes. Ceux-ci résolvent la première partie du problème : déterminer si le graphe est connexe.
Plusieurs fonctions résolvent le problème. Il y a :

- Sommet_parcours : parcours l'ensemble des sommets du graphe
- Composantes_parcours : parcours le graphe pour déterminer les composantes fortements connexes.
- Graphe_est_connexe



7.3.1 Parcours du graphe

Les fonctions citées ci-dessous sont implémentées dans le fichier parcours.lisp.

Nous avons créé une fonction qui s'appelle Graphe_parcours dont le prototype qui prend en argument un graphe et qui retourne la liste des sommets visités dans l'ordre chronologique inverse. Ce n'est juste qu'une fonction d'abstraction pour masquer les multiples arguments de Sommets_Parcours.

Sommets_Parcours prend en argument le graphe, la liste des sommets du graphe, la liste des sommets visités mais dont les successeurs ne sont pas encore visités et les la liste des sommets entièrement visités.

On passe en argument le graphe car une des fonctions utilisées dans Sommets_Parcours a besoin de graphe en argument.

On souhaite aussi que pendant l'exécution de Sommets_Parcours calculer un minimum de fois la liste des sommets du graphe. C'est donc pour cela que nous calculons la liste des sommets du graphe à l'appel de la fonction et que cette liste fait partie des arguments de la fonction.

Dans l'exécution de la fonction Sommets_Parcours, trois cas peuvent de présenter.

Le graphe est vide ou nous avons finit de parcourir les sommets du graphe donc la liste des sommets est vide. Dans ce cas, la fonction la liste des sommets parcourus.

Second cas : le sommet que nous souhaitons visiter a déjà été entièrement visité ou alors il a été grisé car il reste des successeurs de ce sommets à visiter. Dans ce cas, nous lançons le parcours sur le prochain sommet de la liste des sommets du graphe.

Dernier cas : le sommet que nous souhaitons visiter n'a pas encore été visité. On le colorie en gris puis nous visitons tous les successeurs de ce sommet. On ajoute les successeurs puis ce sommet dans la liste des sommets complètement visités.

7.3.2 Détermination des CFC

Les fonctions citées ci-dessous sont implémentées dans le fichier quotient.lisp.
CFC : composantes fortement connexes

Pour déterminer les CFC, nous devons parcourir le graphe transposer en partant du dernier sommet colorié en noir par la fonction précédente.

Pour des raisons d'abstractions, nous avons une fonction qui se nomme Graphe_transpose_Parcours qui prend en argument le graphe et la liste des sommets visités déterminées par la fonction précédente.
Cette fonction apelle G_transposer fonction qui transpose le graphe passé en argument.
Elle apelle aussi la fonction Composantes_Parcours qui prend en argument le graphe transposé, la liste des sommets triée, la partition des composantes fortement connexes et la liste des sommets partiellement visiter. Elle retourne la partition des composantes fortement connexes.

trois cas se présentent :

La liste des sommets triée est vide, on a finit le parcours du graphe transposé.

Le sommet que nous souhaitons visiter appartient déjà à un sous ensemble de la partition. Nous continuons le parcours avec le prochain sommet de la liste triée.

Dernier cas, il faut visiter le sommet et ses successeurs. La liste ainsi obtenu représente une composante que l'on peut ajouter dans la partition.

Remarque : Dans cette fonction au lieu de tester l'appartenance d'un sommet à la partition, nous aurions dû tester l'appartenance du sommet à la liste des sommets gris. Dans l'implémentation actuelle, l'ensemble sommets_gris est inutile.

7.3.3 Le graphe est-il connexe ?

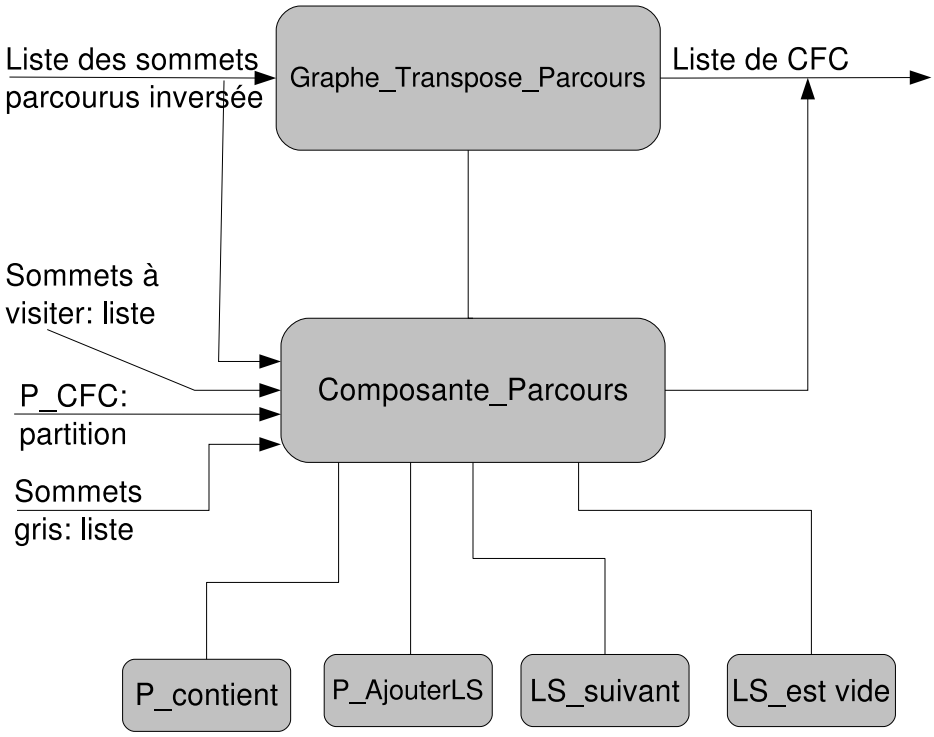
Pour répondre à cette question, il suffit de vérifier la cardinalité de la partition. Si elle vaut 1 alors, nous avons une composante fortement connexe qui est le graphe. On peut alors répondre par VRAI
Sinon, nous répondons FAUX. Nous allons voir donc comment rendre ce graphe fortement connexe.
Remarque : On considère qu'un sommet est une composante fortement connexe si il ne possède ni arcs entrants ni arcs sortants.

7.4 Implémentation de l'algorithme de connexion universelle

Nous sommes dans le cas où il va falloir compléter le graphe. Pour des raisons de complexités, nous voulons calculer le graphe quotient. Chaque sommet du graphe quotient est un sommet représentant de la composante fortement connexe.
Ensuite, nous allons calculer les arcs à ajouter pour que le graphe quotient soit fortement connexe.

7.4.1 Le graphe quotient

Pour générer le graphe quotient, nous devons créer la liste des sommets et la liste d'arcs.
Nous avions dit que les sommets sont les représentants des composantes fortement connexes.



Nous avons implémenté deux fonctions, une fonction d'abstraction Graphe_quotient_sommets qui prend en argument le graphe et qui retourne la liste des sommets. Elle appelle la fonction Generer_liste_sommets qui prend en argument la liste des composantes fortement connexes et la liste qui contiendra l'ensemble des sommets du graphe quotient.

Le représentant de la composante fortement connexe est le premier sommet dans la liste qui représente l'ensemble des sommets appartenant à cette composante.

Nous devons générer la liste des arcs.

Nous regardons, pour chaque sommet d'une composante donnée, les successeurs qui n'appartiennent à cette composante. Nous mettons en liste leur représentant. Nous veillons aussi à ne pas ajouter 2 fois le même représentant car nous travaillons sur des graphes simples orientés.

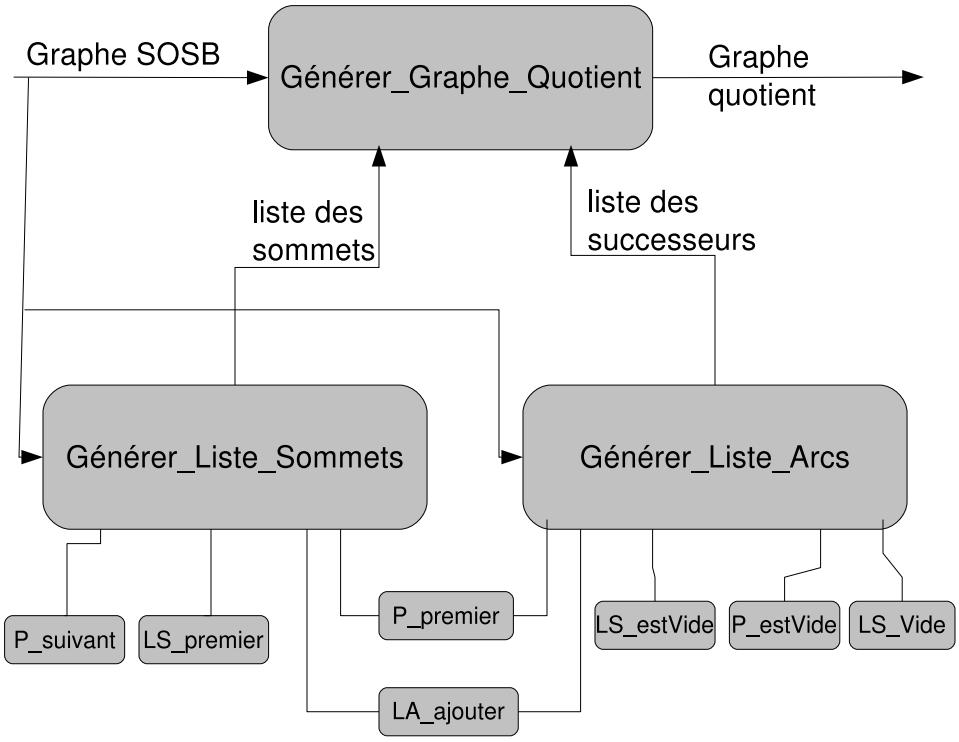
Nous avons donc implémenté 4 fonctions, une fonction d'abstraction Graphe_quotient_arcs qui prend en argument le graphe que nous cherchons à quotienter et qui retourne la liste des arcs sous la représentation liste de successeurs.

La fonction principale est Generer_liste_arcs qui prend en argument le graphe, la partition des composantes fortement connexes et la liste d'arcs à modifier.

Lors de l'exécution de cette fonction 3 cas se présentent :

- la liste des CFC est vide, on a terminé de générer la liste des arcs, on retourne alors cette liste.
- la liste des successeurs d'une composante fortement connexe n'est pas vide. Nous devons ajouter à la liste les représentants des successeurs.
- sinon nous étudions la composante fortement connexe suivante.

La liste des successeurs est établie par Composantes_Successeurs par le biais de Successeurs_cfc.



Successeurs_cfc prend en argument le graphe à quotienter, la liste qui contiendra les successeurs de la CFC étudiée, l'ensemble des sommets successeurs à la CFC appartenant au graphe à quotienter et l'ensemble des sommets de la CFC étudiée. Trois cas se présentent :

- la liste des sommets à visiter est vide, c'est la fin, on retourne la liste des successeurs de la CFC.
- Si le sommet étudié n'appartient pas à la CFC et que son représentant n'est pas déjà dans la liste des successeurs de la CFC alors on l'ajoute et on continue en étudiant le sommet suivant.
- dans le cas contraire, on passe au sommet suivant.

A ce stade, nous avons la liste des sommets et la liste d'arcs du graphe quotient. Nous le construisons à l'aide de 2 fonctions, une fonction d'abstraction Graphe_quotient qui prend en argument un graphe et qui retourne le graphe quotient et d'une fonction de construction Generer_graphe_quotient.

7.4.2 Completion du graphe

L'implémentation des fonctions citées ci-dessous se trouve dans le fichier connexite.lisp.

Dans un premier temps, nous allons déterminer le nombre d'arcs à ajouter. C'est le maximum entre le nombre de puits et le nombre de sources. Il nous faut donc déterminer l'ensemble des puits et l'ensemble des sources du graphe quotient.

La fonction qui calcule le nombre d'arcs à ajouter est Graphe_nbArcs_ajouter qui possède une fonction auxiliaire nbArcs_ajouter. C'est cette dernière qui implémente la

fonction maximum.

Par définition, un sommet puit est un sommet qui ne possède pas de successeur. Il ne figure donc pas dans la liste d'arcs (en représentation liste de successeurs) du graphe quotient. Nous avons implémenté une fonction d'abstraction Puits_graphe_quotient et Puits_liste, la fonction qui calcule la liste des puits du graphe.

Nous savons que les sommets sources du graphe quotient sont les sommets puits du graphe quotient transposé, d'où l'implémentation de Source_graphe_quotient.

Les fonctions d'abstractions que nous venons de citer prennent en argument le graphe et retourne une liste de sommets.

Nous sommes en mesure de déterminer le nombre d'arcs à ajouter. Il faudrait maintenant s'attacher au problème de la complétion. Pour simplifier au maximum le problème, nous allons travailler sur le graphe biparti, graphe composé de sommets qui sont soit puits soit sources.

Nous allons générer une liste appellée liste d'association qui est la liste des successeurs du graphe biparti. Cela est possible grâce à assoc_source_puits, gen_liste_assoc et Puits_associer. La fonction d'abstraction est assoc_source_puits qui prend pour argument le graphe quotient. Le principe de ces fonctions est de lancé un parcours à partir de chaque sommet source et de regarder quels puits nous atteignons. Ensuite, on construit la liste des successeurs.

Remarque : Plus tard, nous aurons besoin de la liste d'association puits vers sources. Il suffit d'exécuter la fonction assoc_source_puits sur le graphe transposé.

Nous avons "les listes d'associations", la liste des sommets sources et la listes des sommets puits.

Il nous faut définir les arcs à ajouter. Nous avons une première fonction nommée Definir_arcs qui appelle des sous fonctions pour choisir le puit et la source. On appelle ensuite la fonction Construire_arcs qui va sauvegarder la liste des arcs définits

Intéressons nous à leur implémentation. Definir_arcs prend la liste des sources, la liste des puits, la liste d'association source vers puits et la liste qui contiendra l'ensemble des arcs ajoutés.

Elle appelle deux fonctions nommées Choisir_source et Choisir_puits. Choisir_puits prend toujours le premier élément de la liste alors que Choisir_source s'arrange à choisir une source telle qu'il n'existe pas de chemin entre la source et le puit.

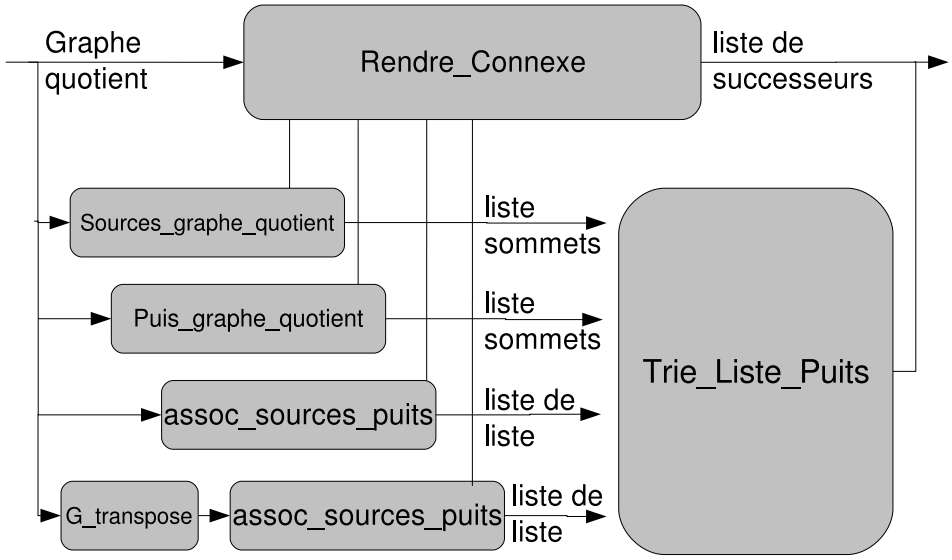
On appelle Construire_arcs qui prend pour argument, la source et le puit choisit ainsi que la liste des sources, la liste des puits, la liste d'association source vers puits et l'ensemble des arcs ajoutés.

Lorsque le nouvel arc est rajouté, elle appelle Definir_arcs. Elle appelle la fonction actualise_assoc qui "met à jour" la liste des successeurs du graphe biparti.

Expliquons l'implémentation de la fonction Ajouter_arcs. Un arc possède un sommet de départ et un sommet d'arrivée. Nous devons avant de rajouter l'arc vérifier si le sommet de départ possède déjà des successeurs.

S'il en possède, nous devons rajouter à sa liste de successeurs, le sommet d'arrivée de l'arc que nous voulons rajouter.

Dans le cas contraire, on va rajouter l'arc en respectant les spécifications imposées par la représentation liste de successeurs.



7.5 Les performances

Pour améliorer au mieux la compléxité de chaque fonction, nous avons écrit un maximum de fonctions en récursivité terminal. Cela permet de retourner le résultat lorsque nous sommes "en fond de récursivité". Nous n'avons pas besoin de remonter la pile d'appel.

Nous avons tester le programme sur un graphe discret de taille 500. Nous étudions le cas du graphe discret puisque c'est le cas critique pour la complétion du graphe. Nous avons pour avoir des résultats sur le temps d'exécution et la consommation mémoire du programme utilisé la fonction time.

Voici les résultats :

Fonctionalité	Temps d'exécution	Mémoire consommée	Cycles CPU
Graphe_est_connexe	5,57 s	59,19 Mo	6701439429
Rendre_connexe	21,18 s	218,63 Mo	25339549089

En regardant les résultats, on remarque que les fonctions qui complètent le graphe sont en terme de temps d'exécution et d'espace mémoire utilisé bien plus couteuses que celles qui servent pour décider si le graphe est connexe.

Nous garderons de coté ces résultats pour les comparer à ceux obtenus en C.

7.6 Rapport de bogues

7.6.1 Bogue 1 : parcours manquant

Un premier bogue a été détecté au niveau de la fonction Composantes_parcours. Il y avait lorsque nous calculions la liste des sommets gris, liste des sommets visités. Nous nous étions contentés de rajouter à la liste des sommets gris les successeurs du sommet que nous étions en train de traiter. Il faut en fait refaire un parcours en profondeur à partir de ce sommet pour mettre dans la liste tout les descendants de ce sommet.

7.6.2 Bogue 2 : tri futile

Nous avions implémenté un pseudo tri des sommets puits à l'étape de la complétion du graphe. En ayant corrigé le bogue précédent, il s'avère que le test devient inutile. Nous

l'avons supprimer après avoir fait de nombreux tests.

Chapitre 8

Implémentation en C

8.1 Adaptation du TAD

8.1.1 Restriction au premier problème

L'implémentation en langage C ne concerne que le premier problème, à savoir celui de la forte connexité du graphe. Il nous faut donc optimiser les TAD en éliminant les primitives inutiles, dont voici la liste :

- LS_retirer
- LA_concatener
- LA_taille
- P_supprimerLS
- P_premier
- P_suivant
- P_concatener

Nous verrons par ailleurs que l'adaptation du TAD à ce langage entraînera au contraire l'ajout de primitives spécifiques.

8.1.2 Gestion mémoire

Contrairement au Lisp qui offre une gestion souple et automatique de la mémoire, le langage C est plus contraignant puisqu'il nécessite une gestion manuelle de cette dernière. D'une part en exigeant la connaissance de la taille de tout objet déclaré, et d'autre part à travers un mécanisme d'allocation et de désallocation dynamique d'espace mémoire à la charge du programmeur, là où le Lisp s'accomode très bien de changements de types, et a recours à un ramasse-miettes natif pour libérer de la mémoire.

La conséquence de la gestion manuelle de la mémoire sur les TAD est l'ajout d'une primitive de suppression quel que soit le TAD. Ainsi, un objet sera créé en mémoire à l'aide la fonction malloc, puis désalloué par le biais de la fonction free, l'objectif visé étant que tout espace mémoire alloué au cours du programme ait été libéré avant la fin de son exécution. Ceci sera vérifié grâce à l'utilitaire de débogage valgrind.

8.1.3 Copie ou partage

Tour d'horizon

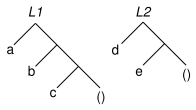
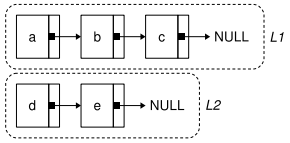
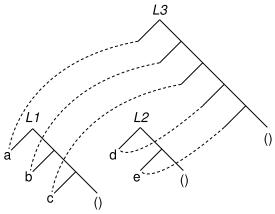
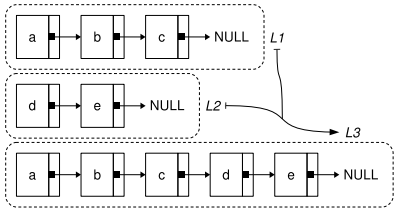
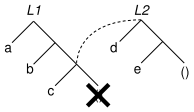
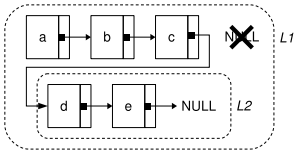
Tels que présentées dans la partie qui leur est consacrée, les primitives des différents types abstraits de données privilégient la copie au partage. En effet, la modification d'un objet consiste dans ce cas à transmettre une copie de cet objet, à la modifier, puis une fois renvoyée par la fonction modificatrice, à l'affecter à l'objet d'origine, qui est donc modifié par écrasement complet. Cette manière de procéder est polyvalente car elle permet tout aussi bien la copie que la modification.

Elle est cependant néfaste en terme de performance, puisqu'elle accroît les complexités en espace et en temps. Par exemple, si on considère une fonction qui extrait une simple information d'un objet transmis en tant que copie, le déficite en mémoire est flagrant. Voyons si la situation permet de s'afranchir de ces inconvénients.

Dans le cadre de ce projet, la plupart des primitives se contentent de modifier un objet ou d'en extraire une information. C'est pourquoi il est préférable de transmettre des objets partagés, directement accessibles par les primitives. On vise ainsi un gain de performance tout en améliorant la sémantique des fonctions. Bien entendu, une mauvaise implémentation des primitives peut faire disparaître le gain attendu. Il faudra donc être vigilent.

Cas de la concaténation

La copie a malgré tout sa place auprès de certaines primitives. Tous les cas de concaténation posent ainsi un problème que l'on peut retrouver en Lisp dans la manière de concaténer deux listes. En effet, dans ce langage, la commande append crée une nouvelle liste, tandis que nconc modifie la première liste en liant son dernier élément au premier élément de la seconde. Voici un tableau explicatif concernant la concaténation selon qu'elle est faite par copie ou par partage :

	concaténation en Lisp	**concaténation en C**
état initial		
méthode par copie		
méthode par partage		

Comme on peut le voir, la méthode par partage prend moins de place en mémoire mais altère la première liste. C'est pourquoi la solution retenue pour implémenter la concaténation est la méthode par copie.

Discussion sur les sommets

On pourrait envisager une existence propre des sommets, avec la possibilité de leur attribuer un certain nombre de propriétés (libellé, couleur, valeur...). Cependant les algorithmes privilégient l'utilisation d'ensembles indépendants pour gérer les couleurs lorsque

cela s'avère nécessaire (algorithme de parcours). D'autre part, le projet ne met pas l'accent sur la gestion des libellés, à savoir le nom des différents formats.

De plus, l'usage probable du programme final concerne une configuration de convertisseurs en relation avec des formats, ce qui ne fait intervenir qu'un seul graphe. Ainsi, la primitive successeur s'applique au TAD Graphe avec pour paramètre l'identificateur d'un de ses sommets, et non le contraire, comme ce pourrait être le cas s'il s'agissait de confronter différentes configurations de convertisseurs, et donc de graphes, s'appuyant sur un même jeu de formats de fichiers.

C'est pourquoi les sommets, qui interviennent dans tous les TAD, se réduiront à de simples entiers naturels, servant d'identificateurs de sommets. D'une certaine manière il s'agit de copie, puisqu'une valeur est à chaque fois transmise, et non une référence vers un sommet.

8.1.4 Effets de bord

La plupart des primitives implémentées en C fonctionnent par effet de bord, par opposition à leur présentation théorique faite dans la partie sur les types abstraits de données. En effet, comme les objets (graphe, liste de sommets, etc.) sont transmis par référence, les fonctions modificatrices agissent directement sur ces derniers, et renvoient un code de bon déroulement. Cette dernière information est d'autant plus utile qu'une opération d'allocation dynamique peut échouer par manque de mémoire.

Le passage du Lisp au C pose le problème de l'appréciation des résultats, les listes étant directement visualisables dans le premier langage. Une fonction d'affichage a donc été ajoutée à chaque TAD, en particulier pour faciliter le déboguage et éventuellement pour afficher des résultats. Ceci constitue un effet de bord sans grande influence sur le reste du programme.

8.1.5 Rôle des pointeurs

Les pointeurs jouent un rôle majeur dans le bon fonctionnement des TAD en C puisque ce sont eux qui permettent à la fois la gestion de la mémoire, le partage des structures en mémoire et donc les effets de bord. Voici le cycle de vie d'un objet du programme :

- *naissance* par la fonction malloc qui fournit un pointeur vers l'objet nouvellement créé ;
- *lecture* et/ou *altération* par le biais de fonctions qui prennent son pointeur en premier argument ;
- *mort* par la fonction free.

8.2 Approche modulaire

L'ensemble du TAD se décompose en quatre couples de fichiers `.h`/`.c` : `listeSommets`, `listeArcs`, `partition` et `graphe`. Ces derniers se trouvent dans le dossier `./include/tad/`.

En réalité, il y a deux versions `.c` de `graphe` correspondant aux deux implémentations et qui reposent sur le même fichier d'en-tête `graphe.h` :

- `graphe.v1.c` : version par liste d'arcs (pas de fonctions privées)
- `graphe.v2.c` : version par matrice d'adjacence (fonctions privées déclarées dans `graphe.v2.h`)

Un niveau en dessous des TAD viennent deux fichiers de déclaration :

- `booleen.h` : déclare un type booleen grâce à une énumération.
  `typedef enum Booleen_ {FAUX, VRAI} Booleen;`
- `identificateurSommet.h` : déclare un type idS comme un entier naturel.
  `typedef unsigned int idS;`

Voici le diagramme des dépendances entre ces différents fichiers, sans oublier les bibliothèques standard utilisées :

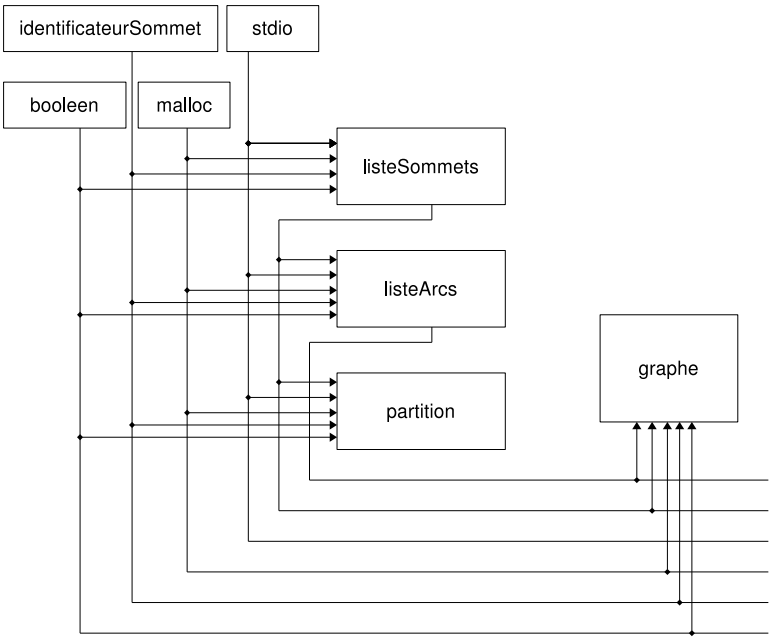


FIG. 8.1 – Diagramme de dépendances des TAD

8.3 Une implémentation des TAD auxiliaires

On s'intéresse aux TAD Liste de Sommets, Liste d'Arcs et Partition. Ces derniers ont certes leur existence propre, mais ils sont très proche les uns des autres et deux d'entre eux servent notamment dans le type de retour des primitives du TAD Graphe.

Nous avons préféré ne faire qu'une implémentation de ces TAD car d'après nos différents points de vue nous aurions convergé vers une implémentation très semblable et donc davantage source d'erreurs et de perte de temps que d'intérêt comparatif. Nous avons ainsi pu concentrer l'activité de développement concurrent sur le TAD Graphe.

8.3.1 Liste de Sommets

Structure

Le parcours ne se faisant que dans un sens, une liste simplement chaînée suffit. On définit donc une structure dotée d'un pointeur vers un suivant et d'un identificateur de sommet (idS).

Dans le fichier `listeSommets.c` :

```
struct ListeSommets_ {
  struct ListeSommets_ * suivant;
  idS id;
};
```

Dans le fichier `listeSommets.h` :

```
typedef struct ListeSommets_ LS;
```

Les trois TAD dits auxiliaires étant très proches, nous allons détailler celui-ci et aller très vite sur les deux autres.

Première allocation

La création d'une liste de sommets passe par une liste vide à laquelle on ajoute des sommets. Voici le déroulement de la création d'une liste vide :

- allouer de la mémoire à hauteur de la taille de LS
- si l'allocation a échoué
  - renvoyer le pointeur NULL
- si l'allocation a réussi
  - mettre à NULL le suivant de la LS allouée
  - mettre l'idS de la LS allouée à une valeur sentinelle
  - renvoyer le pointeur de la LS allouée

Nous avons pris -1 pour valeur sentinelle car cela permet de prendre la plus grande valeur entière possible du fait que idS est défini comme un `unsigned int`.

Notons une fois pour toute le code permettant de gérer une allocation dynamique. Ce code reviendra à de nombreuses reprises dans les TAD, en changeant bien entendu LS et ls en ce qui convient.

```
LS * ls;
ls = malloc(sizeof(LS));
if (NULL == ls) {
  return NULL;
}
else {
  ...
}
```

Cela permet de définir la primitive LS_vide qui renvoie un pointeur vers une liste de sommets vide en temps constant. Elle est complétée par une primitive LS_estVide réalisée en temps constant.

Ajout d'un sommet

Maintenant qu'on sait construire une LS vide, mettons en place la primitive d'ajout d'un sommet. Pour bénéficier d'un ajout en temps constant, on place le nouvel élément en tête de liste. Deux cas de figure peuvent se présenter :

1. la liste est vide : dans ce cas l'identificateur du sommet courant a la valeur témoin ;
   - il suffit de lui assigner la valeur idS.
2. la liste n'est pas vide :
   - dans ce cas il faut créer une nouvelle LS vide.
   - lui assigner l'identificateur courant et le suivant courant (c'est une copie du premier élément).
   - faire pointer le suivant courant vers cette copie.
   - assigner l'idS à ajouter à l'identificateur courant.

On a omis ici la sortie qui a lieu en cas de problème d'allocation, mais ce test est bien présent dans l'implémentation. Observons à quoi ressemble le code source dans le cas 2. :

```
LS * premier;
premier = LS_vide();
if (NULL == premier) {
  return FAUX;
}
else {
  premier->id = ls->id;
  premier->suivant = ls->suivant;
```

```
  ls->id = s;
  ls->suivant = premier;
  return VRAI;
}
```

Il s'agit clairement d'une fonction agissant par effet de bord sur des objets partagés dont l'un d'eux est transmis par référence à l'aide d'un pointeur.

Suivant

On dispose d'une liste dotée d'éléments qu'il s'agit de parcourir à l'aide de la primitive LS_suivant. Celle-ci prend en argument un pointeur vers une LS et renvoie le suivant de cette dernière, si la LS n'est pas vide, et NULL sinon.

Choix d'un sommet

Le choix d'un sommet lors d'un parcours de liste se fait à l'aide de la primitive LS_premier qui prend un pointeur vers une LS en argument et renvoie l'idS du premier sommet, ou -1 (valeur sentinelle) en fin de LS. Cette opération est également en temps constant.

Appartenance

Pour tester si un sommet appartient à une LS, on appelle la primitive LS_contient ayant pour arguments un pointeur vers une LS contenante et un identificateur de sommet éventuellement contenu. La liste est parcourue grâce jusqu'à rencontrer le sommet recherché, ou bien un suivant NULL.

La complexité en temps de contient est donc linéaire par rapport au nombre de sommets.

Concaténation

Comme expliqué précédemment, la concaténation de deux listes se fait par copie : une nouvelle liste est totalement reconstruite. Mais il y a des cas particuliers :

- si les deux listes sont vides, LS_concatener renvoie le pointeur NULL.
- si l'une des deux listes est nulle, LS_concatener renvoie le pointeur de l'autre liste, sans copie.

Dans le cas où les deux listes sont non vides, une liste vide est créée et remplie par les éléments de l'une puis de l'autre des listes. La contrepartie d'une complexité linéaire en le nombre de total de sommets à ajouter est que l'ordre des sommets est inversé par rapports aux listes d'origine, ce qui n'est pas gênant dans nos algorithmes. Cela vient de l'ajout par le début de liste. Il faut à nouveau surveiller que l'ajout, qui procède à une allocation dynamique, ne plante pas en cours de concaténation.

Taille

La primitive LS_taille est elle aussi de complexité en temps linéaire par rapport au nombre de sommets. Il ne s'agit que d'un parcours accompagné d'un compteur.

8.3.2 Liste d'arcs

On raisonne comme pour la liste de sommets, à la différence qu'il y a un sommet de départ et un sommet d'arrivée. Dans le fichier `listeArcs.c` :

```
struct ListeArcs_ {
  struct ListeArcs_ * suivant;
  idS depart;
  idS arrivee;
};
```

Dans le fichier `listeArcs.h` :

```
typedef struct ListeArcs_ LA;
```

Un examen approfondi de l'implémentation de ses primitives ne présente pas d'intérêt, l'essentiel étant calqué sur le Tad Liste d'Arcs que l'on vient de détailler. La seule différence résidence dans le fait de considérer non plus un sommmet, mais deux sommets, ce qui grossit certains tests et certaines opérations.

8.3.3 Partition

Une partition de sommets, dans le cas présent, se ramène à une liste de listes de sommets. On définit donc une structure dotée d'un pointeur vers un suivant et d'un champ correspondant à une liste de sommets, également sous forme de pointeur.

Dans le fichier `partition.c` :

```
struct Partition_ {
  struct Partition_ * suivant;
  LS * ls;
};
```

Dans le fichier `partition.h` :

```
typedef struct Partition_ P;
```

De même que pour le TAD Liste d'Arcs, nous ne développerons pas d'explication sur ses primitives. Une remarque doit cependant être faite en ce qui concerne la primitive P_contient. Elle s'applique bien à un sommet, et renvoie un booléen, contrairement à ce qui se faire dans le TAD général. En effet, la première partie du problème se contente de cette implémentation, ce qui est d'ailleurs plus commode en C.

8.4 Deux implémentations du TAD Graphe

8.4.1 Représentation par liste d'arcs

Structure

La représentation par liste d'arcs est extrêment simple à mettre en place dès lors qu'un TAD Liste d'Arcs existe. On définit donc une structure contenant un pointeur vers une liste de sommets – les sommets du graphe – et un pointeur vers une liste d'arcs – les arcs du graphe.

Dans le fichier `graphe.v1.c` :

```
struct Graphe_ {
  LS * sommets;
  LA * arcs;
};
```

Dans le fichier `graphe.h` :

```
typedef struct Graphe_ G;
```

Graphe vide

Cette implémentation de G_vide peut facilement s'appuyer sur les deux TAD de listes. Les précautions d'allocation mises à part, l'essentiel de la création d'un graphe vide se résume à la création d'une LS vide et d'une LA vide correspondant aux sommets et aux arcs du graphe.

Vacuité d'un graphe

La primitive G_estVide renvoie un booléen indiquant si le graphe est vide ou non. Pour cela, elle vérifie tout d'abord que le pointeur fourni n'est pas un pointeur NULL (elle renvoie faux si c'est le cas), puis regarde si son nombre de sommets est nul, grâce à la fonction LS_taille, auquel cas elle renvoie vrai. Dans les autres cas elle renvoie faux.

Graphe discret

La primitive G_creerDiscret renvoie un pointeur vers un graphe discret de n sommets nouvellement alloues. Elle renvoie un pointeur NULL en cas d'erreur.

Ajout d'un sommet ou d'un arc

La primitive G_ajouterSommet contrôle que le graphe fourni n'est pas un pointeur NULL, que le sommet ou l'arc à ajouter n'a pas d'identificateur sentinel, et qu'il n'est pas déjà dans le graphe Ensuite l'ajout se fait par l'appelle LS_ajouter sur G_sommets ou de LA_ajouter sur G_arcs.

Sommets et arcs

Les primitives G_sommets et G_arcs fournissent respectivement la liste des sommets et la liste des arcs du graphe. Cette fonction paraît superflue dans la mesure où l'implémentation contient déjà ces informations sous forme de listes, mais cela est propre à cette implémentation.

Successeurs

La primitive G_successeurs renvoie un pointeur vers une LS des successeurs d'un sommet du graphe. Voici comment procède cette primitive :

- Successeurs est une liste de sommets vide.
- Si le graphe contient le sommet visé, alors :
  - pour chaque arcs du graphe :
    - si le sommet de départ de l'arc est le sommet visé, alors :
      - ajouter le sommet dans la liste Successeurs
- Retourner Successeurs

Suppression

Une primitive est chargée de faire disparaître de la mémoire toute trace du graphe. Elle fait appel LS_supprimer et LA_supprimer, et à free en ce qui concerne la structure de graphe.

Transposition

La primitive G_transposer crée un graphe transposé d'un graphe et renvoie un pointeur vers celui-ci.

Cet algorithme commence par créer un graphe avec le même ensemble de sommets que le graphe original. Puis il ajoute chaque arc inversé, en changeant l'ordre de l'emploi des primitives LA_arrivee et LA_depart.

8.4.2 Représentation par matrice d'adjacence

Dans cette partie, l'objectif est de détailler la représentation d'un graphe par le biais de sa matrice adjacente associée. Dans un premier temps, le principe général de cette représentation sera expliqué. Nous verrons ensuite ses principales inconvénients et avantages. Dans un second temps, nous verrons comment le type abstrait Graphe à été implémenté de cette manière.

Définition

Cette représentation concerne les graphes dans lesquels seuls les sommets sont nommés. (ce qui correspond au cas présent) Tout graphe orienté simple ([1 :n],E) (cf : Notion sur les graphes) peut être représenté par sa matrice d'adjacence, c'est à dire la matrice M de booléens de taille $n * n$ définie par

$$M[i, j] := ((i, j), E)$$

Une telle représentation implique certains avantages et inconvénients en terme de complexité en temps et en espace.

Avantages
- Cette représentation est un codage : tout graphe admet une unique représentation.
- Tester si un sommet est prédecesseur d'un autre est une opération réalisée en temps constant.

Inconvénient
- La taille de la représentation est élevée : $\theta(n.n)$. Un graphe "peu dense" (avec peu d'arcs) a une représentation de même taille qu'un graphe dense.(Notion de matrice creuse)

Implémentation du TAD

Les différentes primitives à impémenter ont été décrites dans la partie sur le tad. Comme cela à été expliqué précedemment, l'objectif est de pouvoir utiliser le tad avec les deux implémentations. Pour ce faire nous redéfinissons l'objet Graphe_ afin d'utiliser la représentation par matrice d'adjacence :

```
struct Graphe_{
  int **matrice;
  int nb_sommet;
  int t_max;
};
```

Le double pointeur d'entier "matrice" permet d'accéder à la matrice adjacente, et permet donc de connaitre les relations entre les sommets.

Cette matrice est indexée par des entiers de 0 à nb_sommet.

Lorsqu'un sommet est ajouté et que la matrice n'est plus assez grande, la fonction doubler_taille_graphe se charge de faire une réallocation mémoire afin de doubler la taille de la matrice. Ceci justifie la présence de la variable "t_max" dans la structure Graphe_ : elle indique la place mémoire effectivement allouée par la matrice.

Primitives du TAD Ce paragraphe a pour objectif de détailler la manière dont les primitives ont été implémentées pour la représentation matricielle du graphe. Ainsi, pour une description plus complète des primitives, veuillez vous rapporter à la partie concernant la définition du TAD.

Créer un graphe vide

La réalisation du graphe vide se fait par des "malloc" : un pour la structure entière,un autre pour la matrice.
La matrice est allouée de sorte qu'elle puisse recevoir (de manière arbitraire) 50 sommets. Deux boucles se chargent ensuite d'initiaiser tous les champs de la matrice à 0 : ceci correspond donc à la donnée de 50 sommets non connecté entre eux.

La variable "nb_sommet" est initialisée à 0 car à cette instant on considère que le graphe est vide.

La variabe "t_max" est initiaisée à 50.

Prototype : $G * G\_vide()$;
Complexité : $\theta(t\_max * t\_max)$.

Ajouter un arc Il s'agit de remplacer un "0" dans la matrice par un "1" au bonne endroit. L'accés à cette case se fait en temps constant puisque la matrice est indexée à l'aide des sommets.

Prototype : $Booleen\ G\_ajouterArc(G * g, idSdepart, idSarrivee)$;
Complexité : $\theta(1)$.

Ajouter un sommet

L'ajout d'un sommet est l'opération a plus complexe de cette implémentation. Le but est de rajouter une ligne et un colonne à la matrice. Pour ce faire, nous avons choisi de résoudre ce problème en doublant la taille de la matrice lorsqu'il n'a plus de place pour un nouveau sommet :

```
si "nb_sommet" = "t_max"
doubler_taille_matrice

nb_sommet <- nb_sommet + 1
```

L'augmentation de la taille de la matrice se fait par l'appel à la fonction doubler_taille_graphe qui sera décrite plus loin.

Prototype : $Booleen\ G\_ajouterSommet(G * graphe, idSs)$;
Complexité : $\theta(1)$.

Obtenir la liste des sommets

Cette fonction permet de renvoyer la liste des sommets. Elle est constituée d'une bouce allant de 1 à nb_sommet qui ajoute ,à chaque itération, un sommet à la liste de sommets finalement renvoyée.

Prototype : $LS * G\_sommets(G * graphe)$;
Complexité : $\theta(n)$.

Obtenir la liste des successeurs d'un sommet

Cette fonction lit la ligne "sommet" de la matrice à la recherche de "1" (c'est à dire de sommets connecté à "sommet") puis ajoute les sommets correspondants à la liste de sommets finalement renvoyés.

Prototype : $LS * G\_successeurs(G * graphe, idSsommet)$;
Complexité : $\theta(n)$.

Connaitre la liaison entre deux sommets

Cette fonction determine si le contenu de la case (sommet1,sommet2) est à "1" ou à "0". Ainsi, elle renvoie un booléen traduisant la connexion entre deux sommets.

Prototype : $Booleen\ est\_connecte(G * graphe, idSsommet1, idSsommet2)$;
Complexité : $\theta(1)$.

Supprimer un graphe

L'appel de cette fonction détruit le graphe passé en argument à 'aide de l'instruction "free".

Prototype : $Booleen\ G\_supprimer(G * graphe)$;
Complexité : $\theta(1)$.

Fonctions particulières à l'implémentation L'existence de ces fonctions est due à l'implémentation du type. Elle ne sont en aucun cas constitutive du TAD, mais sont utiles pour l'implémentation matricielle.

La fonction nb_sommet ne fait que retourner le champs de la structure Graph_.

$int\ nb\_sommet(G * graphe)$;

Cette fonction permet de doubler la taille d'un graphe. Cette augentation se fait à l'aide d'un "réalloc". La variable "t_max" est alors doublée.

$G * doubler\_taille\_graphe(G * graphe)$;

8.5 Tests de fonctionnalités

Nous devions implémenté un jeu de tests qui teste toutes les fonctionnalités offertes par le TAD tout en vérifiant la bonne exécution des primitives quelques soient les arguments passés. Nous devions en outre créer un jeu de test qui soit de haut niveau afin de tester les 2 implémentations. Nous avons implémenté un jeu de tests par TAD.

Pour chaque test, nous vérifions la robustesse des primitives en plus de tester la correction de chaque primitive.

Nous utilisons assert et printf qui arrête le test au point où le test à échouer ou pour afficher le message de succès. L'utilisation de la macro assert oblige l'utilisateur à compiler le jeu de tests en ne passant pas le paramètre -DNDEBUG au compilateur. Dans le cas contraire, la définition de NDEBUG a pour but de "désactiver" la macro assert.

Remarques : A certains endroit, nous utilisons des primitives alors qu'elles n'ont pas encore été testées.
Nous avons veiller à ce qu'il n'y ait pas de fuites mémoires durant la procédure de test.
Les sommets sont des entiers positifs.

Nous allons maintenant présenter brièvement pour chaque TAD comment nous testons ses fonctionnalités.

8.5.1 TAD graphe

L'implémentation de ce jeu de test se trouve dans test_graphe.c. Voici les principales assertions vérifiées :

Soit G un graphe,

- G_estVide(G_creerDiscret(n)) = VRAI si et seulement si n = 0
- G_estVide(G_vide()) = VRAI
- G_estVide(G_ajouterSommet(G, j)) = FAUX pour tout j sommet
- G_estVide(G_ajouterArc(G, j, k)) = FAUX pour tout j, k sommets et j != k
- LS_taille(G_sommets(G_creerDiscret(n))) = n pour tout n ¿= 0
- LA_estVide(G_arcs(G_ajouterArc(G, j, k))) = FAUX pour tout j, k sommets

Pour les autres tests, je conseille de se référer au fichier mentionner ci-dessus.

8.5.2 TAD liste de sommets

L'implémentation de ce jeu de test se trouve dans test_listeSommets.c. Voici les principales assertions vérifiées :

Soit L une liste,

- LS_estVide(LS_vide()) = VRAI
- LS_estVide(LS_ajouter(L, s)) = FAUX pour s sommet
- LS_estVide(LS_suivant(LS_ajouter(L, s))) = VRAI pour s sommet
- LS_estVide(LS_premier(LS_ajouter(L, s))) = FAUX si s sommet

– LS_contient(LS_ajouter(L, s), s) = VRAI si s sommet
– LS_taille(LS_ajouter(L, s)) = 1 si s sommet

Pour les autres tests, je conseille de se référer au fichier mentionner ci-dessus.

8.5.3 TAD liste d'arcs

L'implémentation de ce jeu de test se trouve dans test_listeArcs.c. Voici les principales assertions vérifiées :

Soit L une liste,

– LA_estVide(LA_vide()) = VRAI
– LA_estVide(LA_ajouter(L, a, b) = FAUX si a et b sommets
– LA_contient(LA_ajouter(L, a, b), a, b) = VRAI si a et b sommets
– LA_estVide(LA_suivant(LA_ajouter(L, a, b), a, b)) = VRAI si a et b sommets

8.5.4 TAD partition

L'implémentation de ce jeu de test se trouve dans test_listeArcs.c. Voici les principales assertions vérifier :

Soit P une partition,

– P_ajouterLS(P, L) = VRAI si L est une liste de sommets.
– P_contient(P, s) = VRAI s'il existe une liste de sommets contenue dans P contenant s.
– P_taille(P) = 1 si P contient une liste de sommets
– P_estVide(P_vide()) = VRAI

8.6 Rapport de bogues

Dans cette partie, nous allons parler des bogues post-publications que nous avons détecté.

8.6.1 Bogue 1 : problème d'initialisation

Dans l'implémentation du graphe en utilisant une matrice d'ajacence, il y a un problème d'initialisation du tableau `ident`. La taille doit être augmentée de 1.

8.6.2 Bogue 2 : (ré)allocations

Dans l'implémentation du graphe avec la matrice d'adjacence, nous devons réallouer les colonnes existentes. Pour les nouvelles colonnes, il faut faire des allocations. Il faut ensuite initialiser les cases à 0 (absence d'arcs).

8.6.3 Bogue 3 : prévoir la réallocation

Dans l'implémentation de la fonction ajouter, on doit contrôler s'il faudra agrandir la matrice. On regarde un coup avant pour prévoir la réallocation. Il faut après avoir exécuté la fonction doubler taille pouvoir rajouter les arcs.

8.6.4 Bogue 4 : condition manquante

Ce bogue touche le TAD Liste de Sommets. Dans la fonction, LS_estVide(), la condition qui permet de décider si la liste est vide n'était pas complète.

8.6.5 Bogue 5 : consommation excessive (non résolu)

Lorsque nous testons le programme en utilisant l'implémentation du graphe qui utilise la matrice adjacence, on peut remarquer une consommation importante de la ressource mémoire. N'ayant pas le temps de chercher quelle est l'origine de ce problème, je ne fais que le signaler. Je tiens à dire quand même que le programme a passé les tests de fuite de mémoire avec succès et que ce problème n'est pas présent lorsque nous utilisons l'implémentation du graphe qui utilise les listes.

8.7 Performance du programme

Nous avons testé les programmes avec Gprof. On a donc compilé le programme avec l'option -pg. Nous avons donc le temps passé par le programme pour chaque fonction et primitive.

Nous avons remarqué dans l'implémentation matricielle que nous 30% du temps à redimensionner la matrice. Un meilleur choix de la dimension par défaut pourrait diminuer ce temps.

D'ailleurs, nous avons relevé un nombre important d'appel de la primitive est_connecte qui peut être appelé 1000 fois plus que les autres fonctions.

Nous avons relevé le temps passé par le programme avec la fonction système time. L'utilisation mémoire a été mesuré avec valgrind.

Chapitre 9

Comparaison

9.1 Gestion de la mémoire

LISP a un avantage sur le langage C. Nous n'avons pas besoin de gérer la mémoire. Nous disposons d'un outil qui libère la mémoire quand il faut, cela permet d'éviter les problèmes de saturation que nous pourrions avoir en C dans le cas des gros graphes.

9.2 Complexité en temps

En C, nous avons la possibilité de faire des affectations. Cette fonctionalité n'est pas disponible, en toute rigueur, dans les langages fonctionelles.

A certains moments dans les fonctions en LISP, il nous arrive de calculer deux fois de suite la même chose. Avec l'affectation, nous aurions pu éviter ceci. Il faut donc faire un compromis entre la complexité en temps et la complexité en espace.

9.3 Passage des arguments

En C, les arguments sont passés par pointeurs. En LISP, lorsque nous passons une liste en argument, il fait une copie de celle-ci pour faire les traitements sur la copie. On consomme au final bien plus de mémoire dans l'implémentation LISP que dans l'implémentation C.

9.4 Résultats des mesures

Nous allons comparer les 3 implémentations en terme d'utilisation de la mémoire et de temps d'exécution. Nous rappelons que l'implémentation LISP n'est pas compilé.

Le test consiste à décider si le graphe est connexe. Le graphe testé est un graphe discret de 1000 sommets.

Implémentation	Temps d'exécution	Mémoire utilisée
en LISP	20,95 s	221,06 Mo
C en liste	0,018 s	69 ko
C en matrice	0,348 s	16,91 Mo

Nous voyons qu'en C grâce au affectation, le temps d'exécution est négligeable devant celle de LISP.

Chapitre 10

Conclusion

Les algorithmes proposés et développés au cours de ce projet ont permis de répondre aux problématiques initiales : on sait dorénavant décider si un jeu de formats de fichiers et de convertisseurs permet la conversion universelle ; dans le cas négatif, on sait également quels convertisseurs ajouter pour atteindre cet objectif tout en minimisant leur nombre.

D'un de vue technique, ce projet nous a permis d'une part de mettre en application nos récentes connaissance en Lisp, et d'autre part d'approfondir celles de C, non seulement les difficultés liées à l'allocation dynamique mais aussi l'utilisation massive des pointeurs.

Nous avons ainsi pu apprécier deux approches de programmation assez différentes, mais dans les deux cas, il a fallu définir au préalable toute l'abstraction nécessaire et s'y tenir. Cela a été très formateur en terme de démarche de développement et de gestion de projet.

Enfin, ce projet nous a permis d'être à nouveau confrontés au travail de groupe et aux difficultés qui en découlent. La communication est en effet au coeur des décisions, et il a fallu adopter une démarche d'ingénieur en entreprise : savoir convaincre ses collègue au sujet des orientations du projet et travailler au mieux en équipe.