Tri topologique

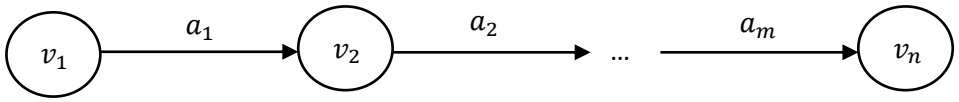
Sommaire

Introduction

Définition 1 (Graphe acyclique orienté)

Un graphe acyclique orienté est un graphe orienté $G = (V, A)$ où $V = \{v_1, \dots, v_n\}$ est un ensemble de sommets et $A = \{a_1, \dots, a_m\}$ tel que $\forall\, 1 \leq i \leq m,\ \forall 1 \leq i, j \leq n,\ a_i = (v_j, v_k) \neq (v_k, v_j)$, un ensemble d'arcs. Ce graphe ne possède aucun circuit. On définit pour tout sommet u et pour tout sommet v la relation d'ordre $R(u, v)$ telle que $R(u, v) = (u \leq v)$.

Représentation



Définition 2 (Tri topologique)

Soit $G = (V, A)$ un graphe acyclique orienté.
Le tri topologique de G donne une extension linéaire $(V, \leq_t)$ de l'ordre partiel sur les sommets $\leq$. $(V, \leq_t)$ est un ordre total :
i.e. $\forall v, u \in V,\ u \neq v \rightarrow (u \leq_t v) \vee (v \leq_t u)$ compatible avec cet ordre partiel.

Implémentation

Paradigme objet

Le choix d'implémentation s'est porté sur le paradigme objet étant plutôt intuitif à manipuler. Les différents concepts mathématiques de sommet, d'arc et de graphe sont donc représentés par des classes. La classe `vertex` modélise un sommet et la classe `directed_graph` représente un graphe. Les arcs sont représentés par la classe `arc`.

Description des classes

La classe `vertex`

Un sommet est caractérisé par une étiquette et par un symbole permettant de le distinguer. En effet, il est nécessaire qu'un sommet soit identifiable parmi ses pairs pour ainsi définir un ordre partiel. La méthode `bool operator < ( const vertex & _ ) const` de surcharge de l'opérateur de comparaison < a été donc définie. Notons qu'elle est aussi indispensable pour pouvoir utiliser la classe template `std::set <vertex>`.

Cette classe est fournie dans le fichier `vertex.hpp`.

La classe `arc`

Un arc est défini par une paire ordonnée de sommets, par un poids et pour le distinguer, par une étiquette. Pour décrire la relation entre deux sommets,
`std::pair<vertex, vertex>` a été choisi.

La classe `arc` est une classe template pour ainsi définir de manière abstraite la notion de poids entre deux sommets (pour ce projet, je me suis contentée d'entiers).

Cette classe est fournie dans le fichier `arc.hpp`.

La classe `directed_graph`

Les deux éléments de base d'un graphe étant défini plus haut, j'ai pu modélisé la classe `directed_graph` par un ensemble de sommets, `std::set <vertex>` et un ensemble d'arcs, `std::set < arc<W> >`. Il est maintenant possible de définir des opérations sur les graphes.

L'insertion, la suppression, l'affichage sont gérés facilement par la classe `directed_graph` grâce aux méthodes de `std::set<>` (notamment : il y a la garantie de ne pas avoir de doublons).

La notion de graphe vide est exprimée par la méthode `bool is_empty() const`.
L'appartenance d'un sommet à un graphe se teste avec la méthode
`bool belongs( const vertex & _ ) const`.

Cette classe est définie dans le fichier `directed_graph.hpp`.

Algorithme du tri topologique

La classe `directed_graph` possède la méthode :
`static void topological_sorting(const directed_graph & _ ) throw (graph_exception&)`
permettant d'afficher un tri topologique.

Soit `std::list<vertex> output`, la liste qui accueillera les sommets selon leur place dans le tri.

Tant que le graphe n'est pas vide, le premier sommet source (qui n'a pas d'arc incident) est mis dans la liste puis il est supprimé ainsi que ses liens dans le graphe (arcs).

Enfin, la liste contenant tous les sommets est affichée.

Correction, complétude et terminaison

Soit G un graphe orienté acyclique d'ordre n.

G possède un tri topologique $T = (v_1, \dots, v_n)$ si et seulement si `topological_sorting(G)` affiche un tri topologique. Le tri topologique donné par l'algorithme est correct et donne une solution pour tout graphe orienté acyclique.

L'algorithme est sûr de terminer car à chaque tour de boucle, un sommet est enlevé.

Complexité de l'algorithme

La représentation des données prend une place majeure dans cet algorithme, mais par soucis de lisibilité et de clarté, j'ai quand même choisi le modèle objet qui est un modèle de haut niveau.

Toutefois, cet algorithme s'exécute en temps linéaire. En effet, il dépend de l'ordre du graphe (nombre de sommets $n \in \mathbb{N}$) et du nombre d'arcs ($m \in \mathbb{N}$) . A chaque tour de boucle, on insère dans la liste le premier sommet source (l'identification se fait par le calcul du nombre d'arcs entrant, ce qui implique un parcours de l'ensemble des arcs en $\mathcal{O}(m)$) puis supprime le sommet du graphe (ce qui induit un parcours de l'ensemble des sommets $\mathcal{O}(n)$ et un parcours de l'ensemble des arcs $\mathcal{O}(m)$).

Les insertions et suppressions sont en $\mathcal{O}(\log n)$ et $\mathcal{O}(\log m)$, ceci est garantie par l'utilisation des méthodes `insert` et `erase` de `std::set<>`.

Commande `make` et tri topologique

Définition 3 (Commande `make`)

`make` est un logiciel qui construit automatiquement des fichiers à partir d'éléments de base tels que du code source.

Il sert principalement à faciliter la compilation et l'édition de liens puisque dans ce processus le résultat final dépend d'opérations précédentes. Pour ce faire `make` utilise un fichier de configuration appelé Makefile.

Définition 4 (Makefile)

Un Makefile est un fichier qui spécifie comment construire des fichiers cibles. Il se présente comme suit :

```
cible_1: dependance_1 dependance_2 … dependance_n
      commande_1
      …
      commande_n
```

La cible `cible_1` dépend de `dependance_1 … dependance_n` pour être construite. L'exécution des commandes `commande_1` à `commande_n` s'appuie sur les dépendances `dependence_1` à `dependance_n` et est nécessaire pour la génération de `cible_1`.

Le Makefile se base sur le tri topologique. Voici comment il procède :

A chaque cible et dépendance est associé un sommet du même nom. Ensuite, il y a création d'un arc entre la cible et ses dépendances. Enfin, un tri topologique est exécuté pour savoir l'ordre d'exécution des commandes.

Grammaire

Soit la GNC $\mathcal{G} = (\mathcal{V}, \mathcal{T}, \mathcal{P}, \mathcal{S})$ où $\mathcal{V}$ = {makefile, rule, target, dependence, cmd} l'ensemble fini de variables, $\mathcal{T}$ = {mot = [^ \\t\n:;]|\\[^\n], :,\t,\n} l'ensemble des symboles terminaux, $\mathcal{S}$ = {makefile} l'ensemble des axiomes (starter) et $\mathcal{P}$ l'ensemble des productions défini par :

$$\mathcal{P} = \{\ \text{makefile} \rightarrow (\text{rule})^*\ \ ;\ \ \text{rule} \rightarrow \text{target} : (\text{dependence})^+ \backslash t\ \ (\text{cmd})^* \backslash n$$

$$\text{target} \rightarrow \text{mot}\ ;\ \text{dependence} \rightarrow \text{mot}\ ;\ \ \text{cmd} \rightarrow (\text{mot})^+ \backslash n\ \ \}$$

$\mathcal{L}(\mathcal{G})$ = les fichiers Makefile (sans affectation ni commentaires).

Notons que cette grammaire n'est pas complète, en effet, elle ne prend pas en compte les affectations et les commentaires. Cependant, ce n'est pas dérangeant pour l'objectif de ce projet.

La définition de cette grammaire a été utile pour analyser les fichiers Makefile et ainsi mettre en évidence l'utilisation du tri topologique par la commande `make`. Ceci a été implémenté en *YACC* avec l'aide de *LEX*. Voir les fichiers `parser.y` et `scanner.l` pour plus de précisions.

Exemple

Soit le Makefile :

```
all: test

test: main.o src.o

	g++ -o test main.o src.o

main.o: main.cpp

	g++ -c -o main.o main.cpp

src.o: src.cpp

	g++ -c -o src.o src.cpp
```

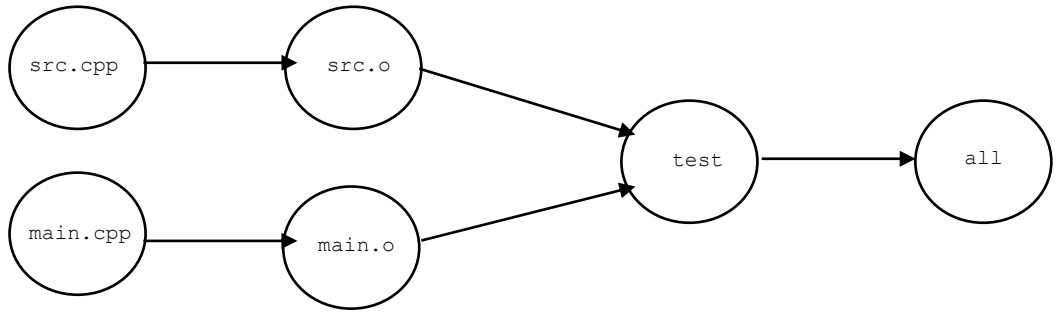
Le graphe associé à ce fichier est $G = (V, A)$ où

$$V = \{ \text{all}, \text{test} , \text{main.o} , \text{src.o} , \text{main.cpp} , \text{src.cpp} \}$$

et $A = \{(\text{test}, \text{all}), (\text{main. o}, \text{test}), (\text{src. o}, \text{test}), (\text{main. cpp}, \text{main. o}), (\text{src. cpp}, \text{src. o})\}$

C'est un graphe acyclique orienté, on peut donc appliquer le tri topologique.

Représentation



L'algorithme du tri topologique affichera : main. cpp , main. o , src. cpp , src. o , test, all . C'est en fait l'ordre total $(V, \leq_t)$ tel que :

$$\text{main. cpp} \leq_t \text{main. o} \leq_t \text{src. cpp} \leq_t \text{src. o} \leq_t \text{test} \leq_t \text{all}$$

C'est bien l'ordre duquel le Makefile procèdera pour construire `test` lors de l'exécution de la commande `make`.

Conclusion

Ce projet tente de modéliser le plus fidèlement possible le mécanisme du tri topologique implémentée par la commande `make`. Le projet se repose sur les définitions mathématiques et sur des outils de la théorie des langages comme les grammaires non contextuelles.

Bibliographie

Toutes les sources se trouvent à l'adresse
https://github.com/mbouzid/topological_sorting/wiki .

Anglais

University of Washington - "Data Structures & Algorithms" - PhD. Rajesh Rao

Wikipedia - Directed graph

MIT - "Dijkstra algorithm" - PhD. Thomas Rothvoss & Melissa Yan

Français

University of Angers - "Programmation Logique Master 1" - PhD. Igor Stéphan

InfoPrépa - TD

LABRI - "DUT info TP6" - Adrien Boussicault

Wikipedia - Make