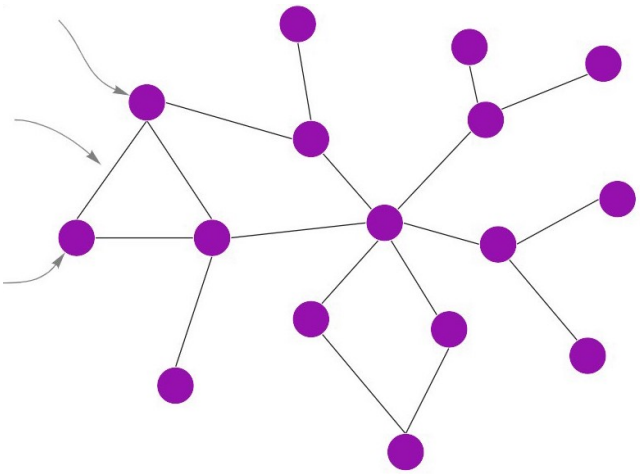
Projet de compilation - GraphEx



:

Réalisé par	EZZAAOUI Rahali Hamza
	HAJAZI Soufiane
:	JAMAI Mohammed Amine
	KAHLAOUI Ismaïl
Encadré par:	M. RACHID OULAD HAJ THAMI
Groupe:	GL2
Année académique:	2020 - 2021

Il n'y a pas de mauvais projets. Il n'y a que de mauvais ingénieurs.

Mise en page terminée.

Le:: 5 mars 2021

Table des matières

Remerciement

Avant d'entamer tout développement de cette expérience fructueuse, nous aimerons tout d'abord exprimer notre gratitude à un nombre de personnes qui ont contribué à la réalisation et au bon déroulement de notre projet. Nous commençons ainsi par présenter nos remerciements à l'Ecole Nationale supérieure d'informatique et d'analyse des systèmes qui ne cesse de prôner de telles expériences afin d'assurer une meilleure intégration de ses lauréats dans le milieu professionnel.

Toute notre gratitude et profonde reconnaissance s'adressent à Monsieur. TABII Youness de nous avoir garanti les meilleures conditions et facilités pour mener à bien ce type de projets.

Par la même occasion, nous tenons a remercier profondément notre encadrant Monsieur. RACHID OULAD HAJ THAMI qui a su se rendre disponible pour nous aider quand le besoin s'en faisait ressentir.

Chapitre 1

Présentation du projet

Sommaire

1.1 Problématique :

Face à une panoplie de concepts abstraits relatifs à la théorie des graphes, et à la rigueur que demande la manipulation des graphes, un besoin s'impose, car non seulement nous devons bien visualiser les graphes, mais nous devons aussi les parcourir et découvrir leurs détours et leur souffler la vie. Le marché est saturé de bibliothèques basé sur du Javascript qui permettent de visualiser les graphes mais qui offrent des fonctionnalités limitées, d'où l'intérêt que nous portons pour un langage orienté graphes qui soit à 100% Ensiaste.

1.2 Introduction au GraphEx

Le langage que nous proposons est un langage orienté graphe donc dédié à la manipulation des graphes de différentes nature. Nous avons choisi de le nommer : **GraphEx** (abréviation de Graphe Explorer).

Extension des fichiers : chaque fichier doit porter l'extension .gx

La structure du fichier :

- Déclaration de sous-graphes.
- Déclaration du graphe main.

Ça veut dire que chaque ligne de code doit avoir une relation avec au moins un graphe.

1.3 Objectifs

Afin de permettre aux élèves ingénieurs Ensiastes de bien assimiler le module de la théorie de graphe, on propose de réaliser un langage de programmation, orienté graphe, son objectif est de permettre une meilleure visualisation, la manipulation et l'application des différents algorithmes dispensé en cours, à savoir : **Djikstra, Bellman, Kruskal**,...

Chapitre 2

Syntaxe & Grammaire

Sommaire

2.1 Syntaxe du GraphEx

Structure du langage

Notre langage de programmation est constitue principalement de deux parties, une première dédie a la définition des sous-graphes afin de les utilisées dans notre graphe main. Cette première partie est pas obligatoire, alors que la définition du graphe main est indispensable dans un fichier .gx.

Le graphe main représente un point d'entrée dans GaphEx.

Le code suivant, vous présente la structure globale d'un fichier .gx.

```
1000 nomGraphSecondaire {

1002     %TYPE {directed | undericted}

1004         // On fait le choix entre un graphe orient  ou non.

1006     %SUBGRAPH

1008         // Instanciation des sous-graphes

1010     %DECLARE

1012         // D claration des noeuds et arcs.

1014 }

1016 // On peut d clarer ici autant de sous-graphes qu'on veut.

1018 main {

1020     %TYPE {directed | undericted}

1022         // On fait le choix entre un graphe orient  ou non.

1024     %SUBGRAPH

1026         // Instanciation des sous-graphes.

1028     %DECLARE

1030         // Declaration de noeuds et arcs.

1032     %OPERATIONS

1034         // C'est la partie dedie aux operations sur notre graphe
     main.

1036         // On peut faire appel aux differentes fonctions predefinies

1038 }
```

Instanciation des sous-graphes

Pour un sous graphe on peut créer plusieurs instances, afin d'assurer la réutilisation. Voici un exemple d'instanciation d'un sous-graphe :

```
%SUBGRAPH

    nomSousGraph : premierInstance [, autreInstance,    ];
```

Déclaration de père, fils et poids :

Évidement chaque graphe contient des noeuds et des arcs, alors dans notre langage on les présentes comme ci-dessous :

```
%DECLARE

    Racine -> Fils1[, poids1];

    Fils1 -> Fils2, poids2 [-> Fils3, poids3];
```

Appel de sous graphe :

Pour faire appel a une instance de sous-graphe il y a quelques règles a respecter, a noter qu'un sous-graphe ne peut pas avoir des fils. Voici des exemples décrivant l'appel des instances des sous-graphes :

```
    // Exemple valide :

        noeudPere -> NomSousGraphe1(nomNoeud) [, poids];

        // ... nomNoeud c'est le noeud qui va etre liee avec
    noeudPere

    // Exemples non valide :

        noeudPere -> instanceSousGraphe(nomNoeud) [, poids] -> ...;

        instanceSousGraphe() -> ...;
```

2.1.1 Opérations :

La partie opérations, c'est une partie propre au graphe main dont l'utilisateur peut faire des traitement sur son graphe main.

Dans cette partie, l'utilisateur peut faire appel a un ensemble des fonctions prédéfinies qui vont facilite la manipulation de son graphe.

Dans ce qui suit, on vous présente l'ensemble des fonctions prédéfinies :

```
1000 // NB: On ne peut faire l'appel d'operations que dans le graphe main
     .

1002 printAll([instanceSousGraphe]) :

1004     // Affiche des listes de noeuds/arcs et les poids associes.
         // S'il y a pas de parametre, le graphe concerne par cette
         operation est le main.

1006
     printNodes([instanceSousGraphe]):

1008
         // Affiche la liste des noeuds dans un graphe (main dans le cas
         d'absence de parametre).

1010
     printEdges([instanceSousGraphe]):

1012
         // Affiche la liste des arcs du graphe (main dans le cas d'
         absence de parametre).

1014
     getChemin(node1, node2, [instanceSousGraphe]):

1016
         // Affiche une liste contenant tous les chemins possibles entre
         node1 et node2 pour un graphe donnee en troisieme parametre (
         main dans le cas d'absence de ce parametre).

1018
     getWeight(node1, node2, [instanceSousGraphe]);

1020
     getNode(edge):

1022
         // Retourne une liste des deux noeuds attaches    l'arc.

1024
     exists({node1, node4,..}, [instanceSousGraphe]):

1026
         // Verifie si un chemin  existe dans le graphe mentionn  (sinon
          main).

1028
     minCost(noeudDebut, noeudFin, [instanceSousGraphe]):

1030
         // Donne le cout minimal possible dans les chemins du noeud de
         debut vers celui de la fin dans un graphe.
```

```
nombreChromatique([instanceSousGraphe]):

    // Retourne le nombre chromatique d'un graphe donne ou bien de
    main.

colorier(node, color):

    // permet de colorier un noeud

    // color -> #RED, #YELLOW, #GREEN, ...

colorerGraph([instanceSousGraphe]):

    // Permet de colorer un graphe on se basant sur le resultat de
    nombreChromatique avec des couleurs aleatoires.

plot([NomSousGraph1]):

    // Affiche le graphe mentionne dans une nouvelle fenetre (si
    aucun graphe n'est mentionne, on considere le main).

Traverse([instanceSousGraphe] METHOD, (startNode, endNode, edge) =>
    {

        If (getWeight(edge) > 5) {

            colorier(startNode, #RED);

            colorier(endNode, #RED);

        }

}):

    // Exemple de Traverser, qui est pour objectif de traverser un
    graphe donne par l'une des m thodes (DFS ou BFS) et fait des
    operations sur chaque noeud ou sur l'arc qui les lient.

    // Dans ce qui suit des algorithmes de base qu'on peut appliquer
     sur nos graphes et avoir des resultats graphique :

Dijkstra(node1, node2 [, sousGraph]);

Bellman(node1, node2 [, sousGraph]);

DijkstraGeneralise(node1, node2 [, sousGraph]);

Kruskal([sousGraph]);
```

```
Prime([sousGraph]);
```

2.2 Grammaire

La grammaire en cours d'étude s'agit bien d'une grammaire LL1, la vérification de cette dernière à été faite à travers d'un outil qui s'appelle **smlweb**, vu l'absence de toute récursivité à gauche ou de la factorisation à gauche, on a pu conclure que notre grammaire est **LL1**

```
1000 Graph        :: ID { TYPE SUBGRAPH DECLARE } Graph
                  | main { TYPE SUBGRAPH DECLARE OPERATIONS }
1002              | epsilon

1004 TYPE         :: %type { TYPE' }

1006 TYPE'        :: directed
                  | undirected
1008
     SUBGRAPH     :: %subgraph ID : ID OTHERID INSTANCE
1010              | epsilon

1012 INSTANCE     :: ID : ID OTHERID INSTANCE
                  | epsilon
1014
     OTHERID      :: , ID OTHERID
1016              | ;

1018 DECLARE      :: %declare ID CHILD CHILDS

1020 CHILD        :: -> ID CHILD'
                  | ;
1022
     CHILD'       :: POIDS CHILD
1024              | (ID') POIDS ;

1026 ID'          :: ID
                  | epsilon
1028
     POID         :: , chiffre
1030              | epsilon

1032 CHILDS       :: ID CHILD CHILDS
                  | epsilon
1034
```

```
OPERATIONS  :: %operation INST ; INSTS

INSTS       :: INST ; INSTS
            | TRAVERSER INSTS
            | IF INSTS
            | epsilon

INST        :: operationName ( PARAMS )

PARAMS      :: PARAMS'
            | epsilon

PARAMS'     :: ID OTHERPARAM
            | #color OTHERPARAM
            | INST OTHERPARAM

OTHERPARAM  :: , PARAMS'
            | epsilon

IF          :: if (COND) { INSTS }

COND        :: EXPR OP EXPR

EXPR        :: chiffre
            | INST

OP          :: = | <> | < | > | <= | >=

TRAVERSER   :: traverse (ID'' METHOD, (ID , ID , ID) => { INSTS });

ID''        :: ID ,
            | epsilon

METHOD      :: dfs | bfs

ID          :: _  SUITE
            | lettre SUITE

SUITE       :: lettre SUITE
            | chiffre SUITE
            | epsilon
```

2.3 Les variables

Les variables de notre grammaire se subdivise principalement en 3 trois grands catégories, à savoir :

- Les graphes.
- Les noeuds.
- Les arcs.

L'ensemble des opérations qu'on peut les appliquer sur ces variables varient des opérations de déclarations(ex : %DECLARE,...), des opérations d'affichage(ex :printEdges, printNodes,...), ainsi que des opérations conditionnelles et de parcours(ex : Traverse-pour parcourir un graphe).

Il est à noter que les opérations arithmétique ou binaires ne sont pas inclues dans la grammaire en cours d'étude pour la simple raison qu'il n'ont pas d'utilité dans la théorie des graphes, ainsi que l'équivalent de la fonction **Scanf** qui nécessite des données à saisir par l'utilisateur, chose qu'on l'a pas pu exploité vu l'absence de son utilité pour notre cas d'étude.

2.4 Unités lexicales

> Cette partie contient l'ensemble des unités lexicales utilise pour implémenter notre grammaire.

Voir la figure ci-dessous.

TokenType	Désignation
ID	ID_TOKEN
NUMBER	NUM_TOKEN
(	OP_TOKEN
)	CP_TOKEN
=	EQ_TOKEN
<>	NEQ_TOKEN
>	GT_TOKEN
<	LT_TOKEN
<=	LEQ_TOKEN
>=	BEQ_TOKEN
{	OB_TOKEN
}	CB_TOKEN
main	MAIN_TOKEN
%TYPE	PTYPE_TOKEN
%DECLARE	PDECLARE_TOKEN
%SUBGRAPH	PSUBGRAPH_TOKEN
%OPERATIONS	POPERATIONS_TOKEN
Directed | Undirected	GTYPE_TOKEN
->	EDGE_TOKEN
,	COMMA_TOKEN
;	SEMICOLON_TOKEN
#COLOR	COLOR_TOKEN
IF	IF_TOKEN
TRAVERSE	LOOP_TOKEN
OperationName	OPERATION_TOKEN
=>	ARROW_TOKEN
DFS | BFS	GSEARCH_TOKEN
EOF	EOF_TOKEN
:	COLON_TOKEN

FIGURE 2.1 – Les unités lexicales du langage GraphEx

Chapitre 3

L'analyseur lexicale

Sommaire

3.1 Définition

L'analyseur lexical constitue la première phase d'un compilateur. Sa tâche principale est de lire les caractères d'entrée et de produire comme résultat une suite d'unités lexicales que l'analyseur syntaxique va l'utiliser. A la réception d'une commande ¨prochaine unité lexicale¨manant de l'analyseur syntaxique, l'analyseur lexical lit les caractères d'entrée jusqu'à ce qu'il puisse identifier la prochaine unité lexicale.

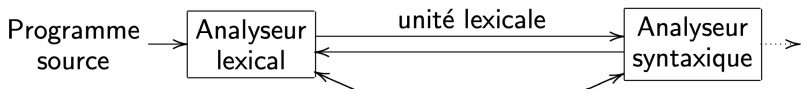


FIGURE 3.1 – Principe de fonctionnement d'un analyseur lexical

3.1.1 La fonction NextToken

Afin de générer une suite d'unités lexicales, ou de Tokens, l'analyseur lexical s'appuie principalement sur la fonction **NextToken**, cette fonction fonctionne comme suit :

- Si le mot commence par une lettre, cette fonction vérifie s'il s'agit bien d'un mot clé ou d'un ID.
- S'il commence par le caractère *#;* on va vérifier s'il s'agit bien d'un couleur, sinon il va générer une erreur lexicale.
- pour les mots qui commencent par le caractère '‰', ´rifier s'il s'agit des Tags prédéfinies, un message d'erreur va être affiché dans le cas échéant.
- lors de l'insertion des caractères spéciaux, la fonction va retourner le Token adéquat.
- La fonction génère un erreur dans le cas où un caractère ne correspond pas à un Token valide dans la table des unités lexicales.

3.1.2 La détection d'erreur

On plus de la division du fichier sous forme de tokens, et de stocker le type de chaque unité lexicale, l'analyseur permet également de détecter les erreurs et d'afficher un message qui contient un ensemble des informations relatives à l'erreur, à savoir(la ligne, la colonne,...) lors de l'insertion d'un caractère inconnue ou non valide, comme le montre la figure ci-dessous.

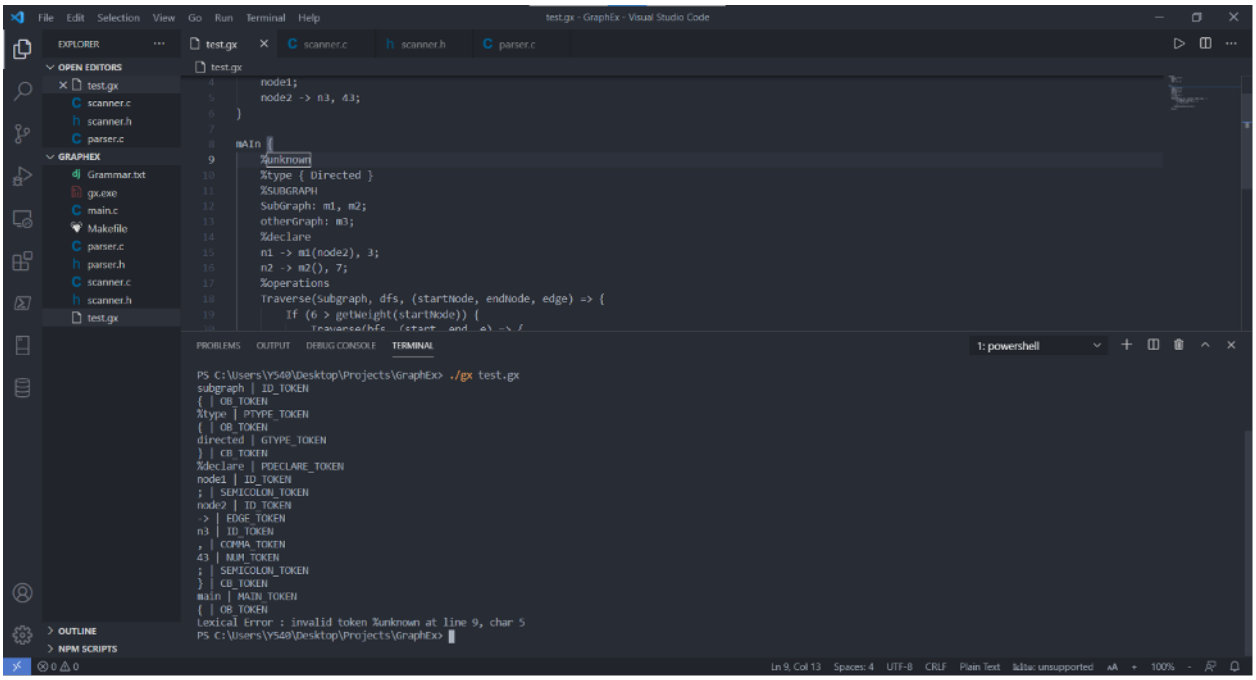


FIGURE 3.2 – Détection des erreurs lexicales

Chapitre 4

L'anlyseur syntaxique

Sommaire

4.1 Définition

L'analyse syntaxique vise à produire une représentation sous forme d'arbre des relations grammaticales entre les mots.

4.1.1 Principe de fonctionnement

L'analyseur syntaxique fait appel à chaque fois, à la fonction next_Token depuis l'analyseur lexical, afin de déterminer le token suivant et son type. L'analyseur syntaxique permet également de vérifier le respect de notre grammaire.

4.1.2 La détection des erreurs

Le programme renvoi un message d'erreur qui contient un ensemble des informations relatives à l'erreur, à savoir la ligne, la colonne,le type du token attendue,... Si par exemple, on s'attendait à un token spécifique, et on trouve un autre, comme le montre la figure ci-dessus :

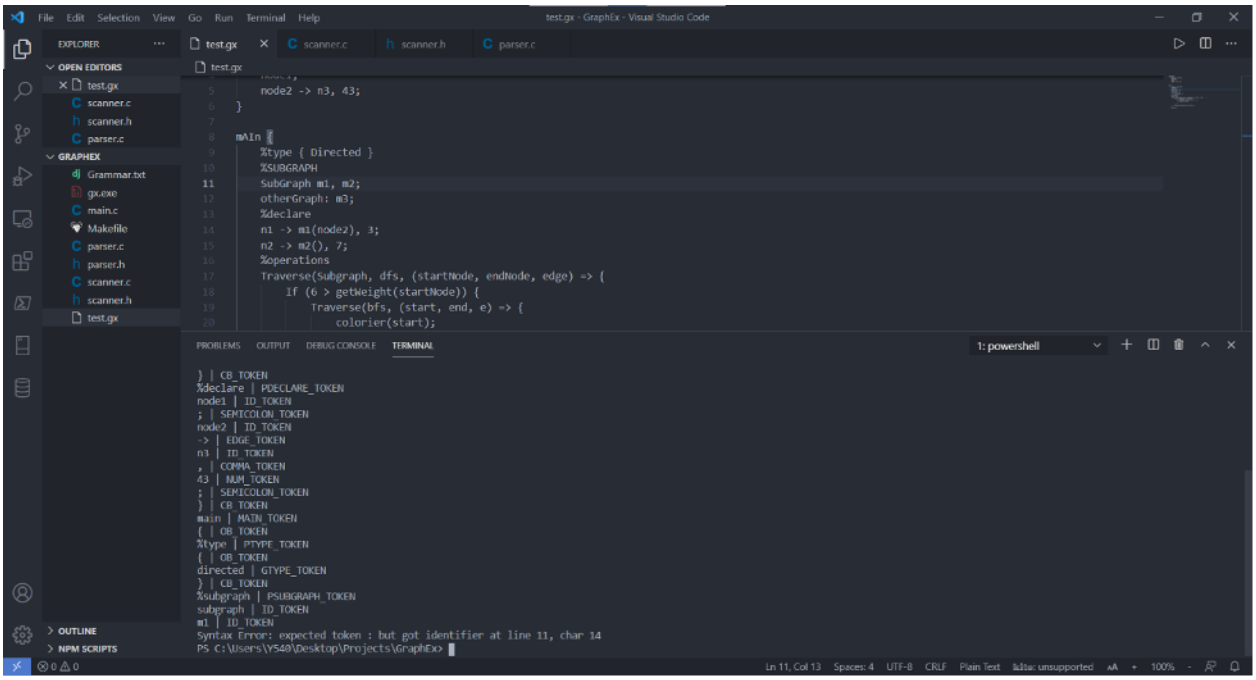


FIGURE 4.1 – Détection des erreurs syntaxiques

Chapitre 5

Le code en pratique

Sommaire

5.1 Exemple de code

Déclaration des sous-graphes

L'exemple de la figure 5.1 montre le processus de création d'un graphe non orienté et d'un sous graphe, ainsi que le coloriage des noeuds en se basant sur le résultat d'une condition prédéfinie.

```
myFirstGraph{
        %TYPE{undirected}
        %DECLARE
                a -> b,2;
                b -> c,4;
                c -> d,6;
                d -> f,1;
}

Main{
        %TYPE{undirected}
        %SUBGRAPH
                myFirstGraph : test;
        %DECLARE
                racine->test(a);
        %OPERATIONS
                Traverse(test, DFS , (a, f, edge) => {
                        If (getWeight(edge) > 5) {
                                colorier(startNode(edge), #RED);
                                colorier(endNode(edge), #BLUE);
                        }
                });
                plot(test);
}
```

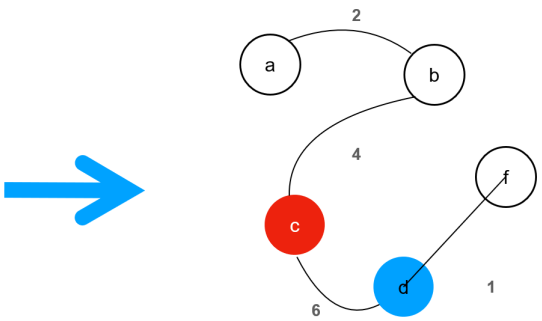


FIGURE 5.1 – La déclaration des sous-graphes & coloriage des noeuds

L'application des algorithmes de construction des chemins

L'instance de code de la figure 5.2 a pour but de trouver le chemin le plus court à partir d'un graphe qu'on la définie à l'avance sous le nom de *myFirstGraph*.

```
myFirstGraph{
        %TYPE{indirected}
        %DECLARE
                a->b,1;
                a->c,3;
                a->d,6;
                b->d,1;
                c->d,6;
                d->f,2;
}

Main{
        %TYPE{indirected}
        %SUBGRAPH
                myFirstGraph : test;
        %DECLARE
                racine->test(a);
        %OPERATIONS
                getChemin(a, f, myFirstGrapgh);
        //retourne tous les chemins possibles du graphe de a à f

                plot(myFirstGrapgh);
}
```

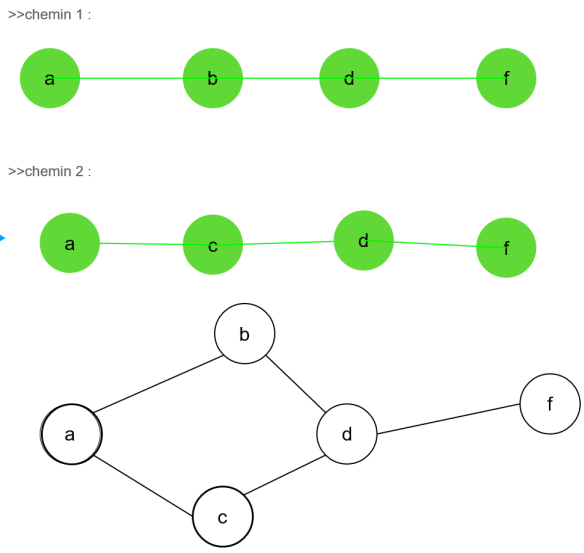


FIGURE 5.2 – L'application de l'algorithme de Djikstra pour trouver le chemin le plus court

L'affichage des chemins possibles entre 2 points

Afin d'afficher l'ensemble des chemins possibles entre 2 points d'un même graphe, on utilise la commande *getChemin(point_de_départ,Point_d'arriver,graphe)*, et le résultat de l'exécution va nous retourner l'ensemble des combinaisons possibles, comme le montre la figure 5.3 .

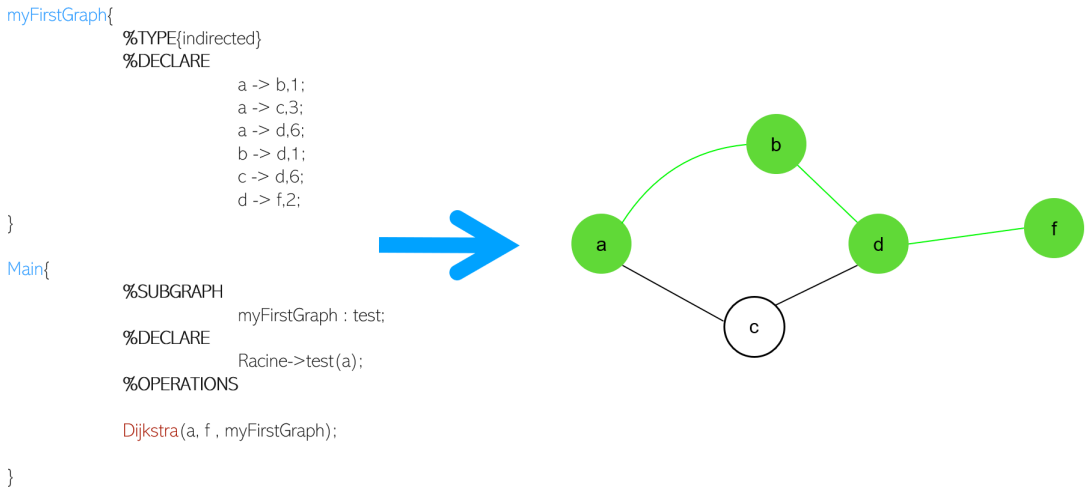


FIGURE 5.3 – L'extraction des chemins possibles entre 2 noeuds

L'affichage des noeuds et des arcs d'un graphe

Les 2 commandes printNodes et printEdges ont pour rôles d'afficher les noeuds et les arcs d'un graphe et d'imprimer le résultat dans la console, la figure 5.4 montre un cas d'exécution de ces commandes.

```
myFirstGraph{
        %TYPE{indirected}
        %DECLARE
                a->b,1;
                a->c,3;
                a->d,6;
                b->d,1;
                c->d,6;
                d->f,2;
}
mySecondGraph{
        %TYPE{indirected}
        %DECLARE
                a->b,1;
                b->d,7;
}
Main{
        %SUBGRAPH
                myFirstGraph : test;
                mySecondGraph: test2;
        %DECLARE
                racine->test(a);
                racine->test2(a);
        %OPERATIONS
                exists(a,test);
                printNodes(test);
                printEdges(test2);
```

Résultat exécution

```
>> true
>> {a,b,c,d,e,f}
>> [{a,b,1},{b,d,7}]
```

FIGURE 5.4 – L'affichage des noeuds et des arcs d'un graphe

Chapitre 6

Amélioration & Suggestions

Sommaire

6.1 GraphEx 1.0

Dans la construction de notre grammaire, on a mis l'accent sur les fonctionnalités qu'on a jugé utile pour la manipulation, la construction, et le parcours des graphes, grâce au langage GraphEx, un étudiant ou un utilisateur peut facilement appliquer les algorithmes de la théorie de langage sur un graphe qui peut être personnalisé selon son besoin, ainsi que l'affichage des noeuds, des arcs, le coloriage et plein d'autres fonctionnalités.

6.2 Améliorations possibles

parmi les améliorations que notre langage peut entretenir, on trouve au première lieu :

- Introduire la notion des données dans les noeuds(String, Caractère, ...).
- Ces notions vont nous permettre d'ajouter les différents opérations dans un langage de programmation ordinaire(opérations binaires, arithmétiques, traitement des inputs, gestion des exceptions,..).
- Introduire la notion de la programmation orientée objet.