GraphEx

Langage de programmation orienté graphes

Réalisé par :

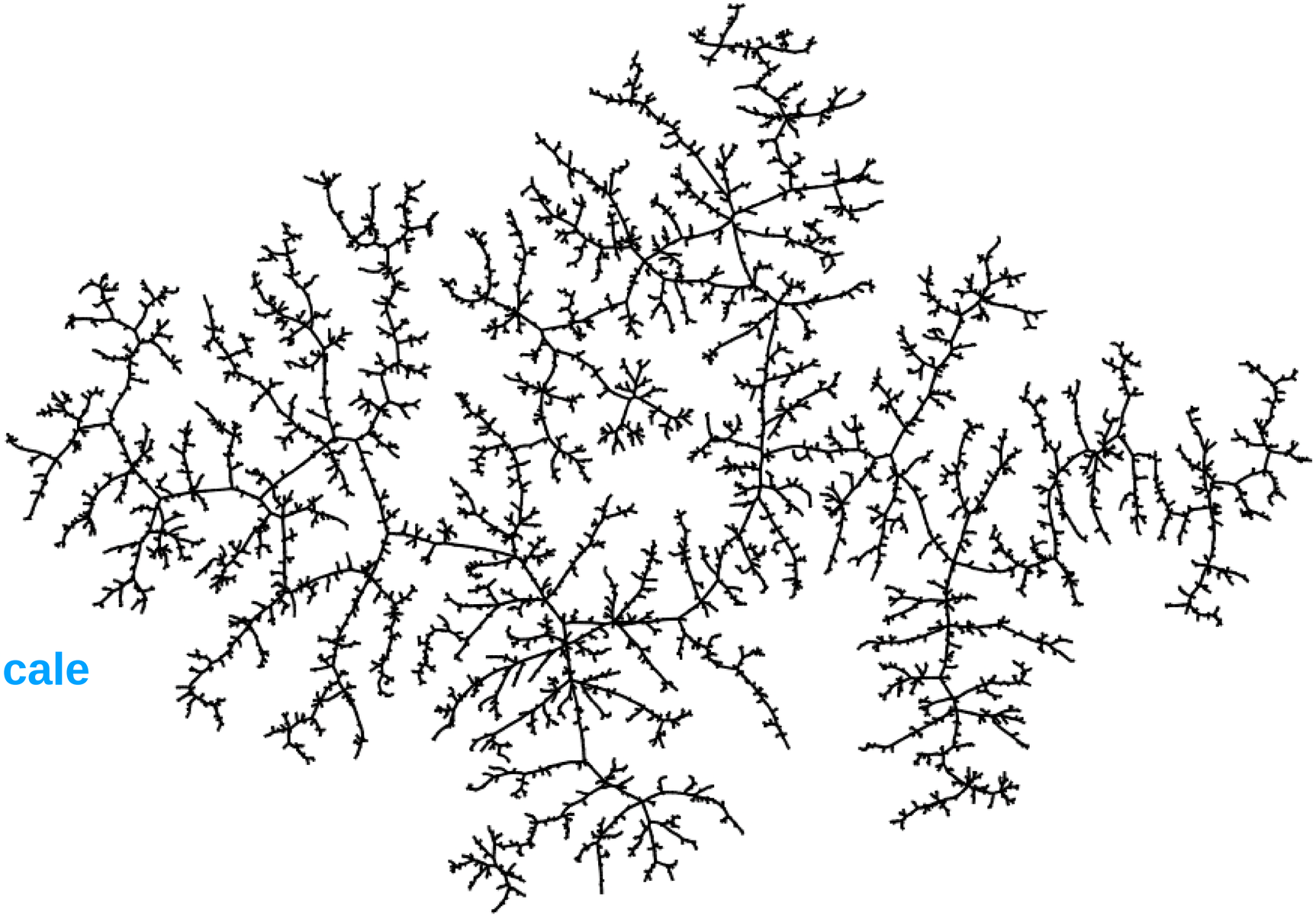
EZZAAOUI Rahali Hamza
HAJAZI Soufiane
JAMAI Mohammed Amine
KAHLAOUI Ismaïl

Encadré par:

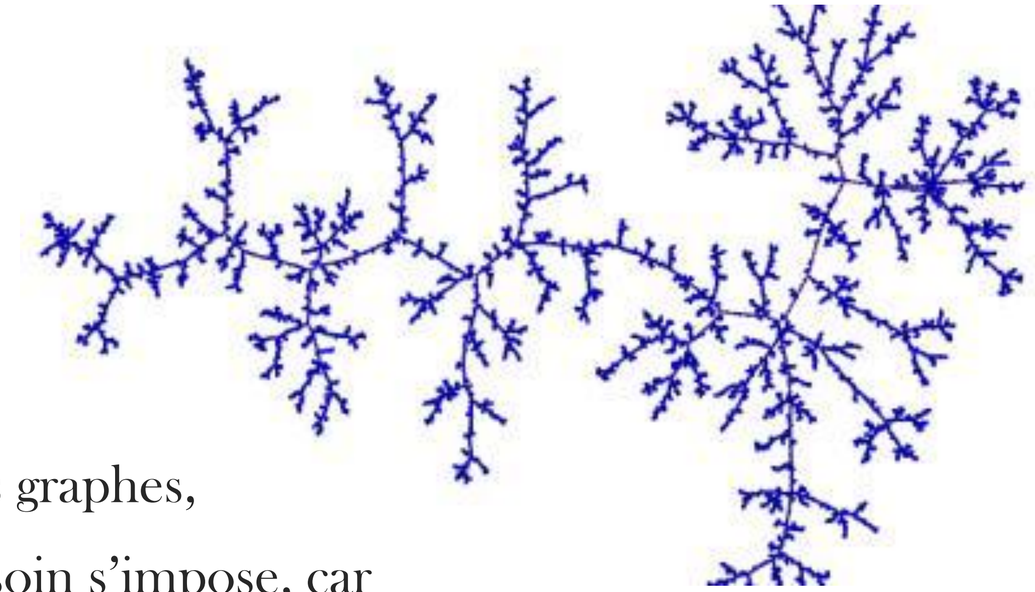
M. RACHID OULAD HAJ THAMI

Groupe: GL2

Plan



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 découvrirleurs 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.

Pourquoi GraphEx ?

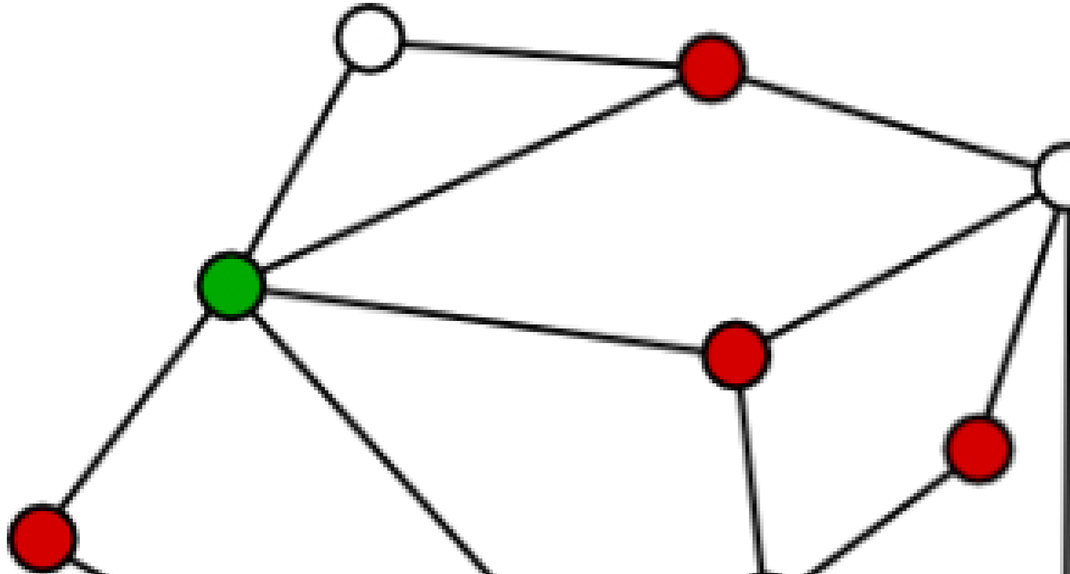
- Une grammaire simplifiée
- Un langage optimal
- Une grande liberté d’expression
- Un langage visuel
- Une large offre d’outils de traitement

Comment fonctionne GraphEx ?

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

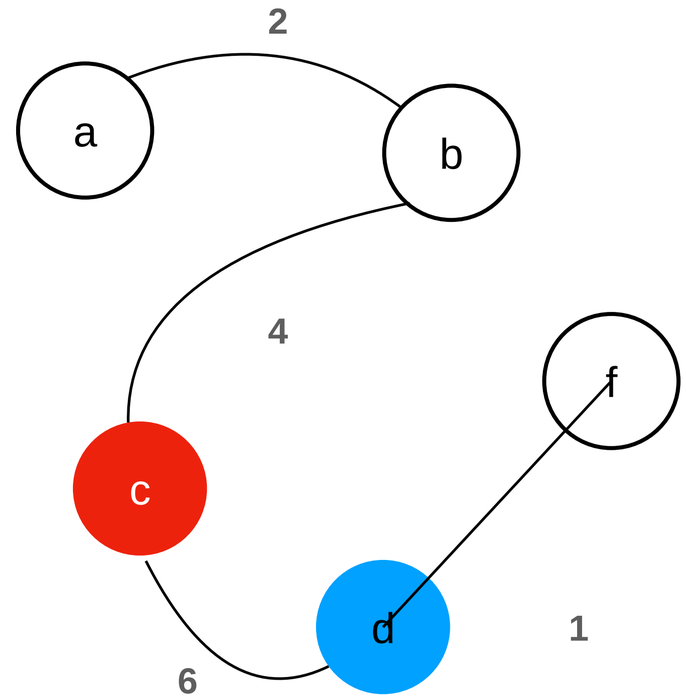
La structure du fichier:

- Declaration de sous-graphes en premier
- Declaration du graphe main obligatoire (la partie executable)
- Tout graphe déclaré en dehors du Main doit être relié au Main
- Extension des fichiers : chaque fichier doit porter l'extension .gx

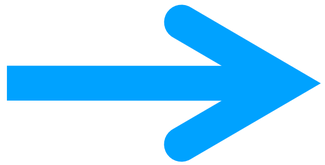


Exemples de code

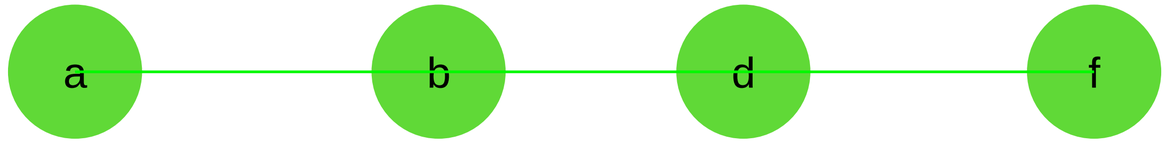
```
myFirstGraph{
        %TYPE{undirected}
        %DECLARE
                a -> b,2;
                b -> c,4;
                c -> d,6;
                d -> f,1;
}

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);
}
```



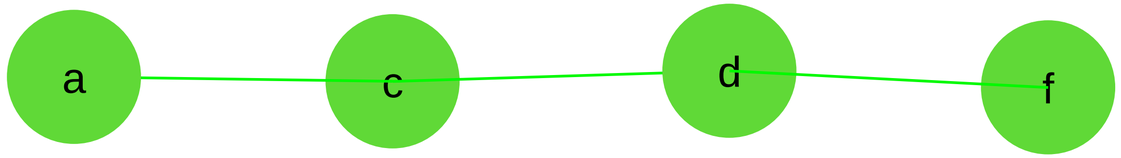
```
myFirstGraph{
        %TYPE{indirected}
        %DECLARE
                a->b,1;
                a->c,3;
                a->d,6;
                b->d,1;
                c->d,6;
                d->f,2;
}

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);
}
```

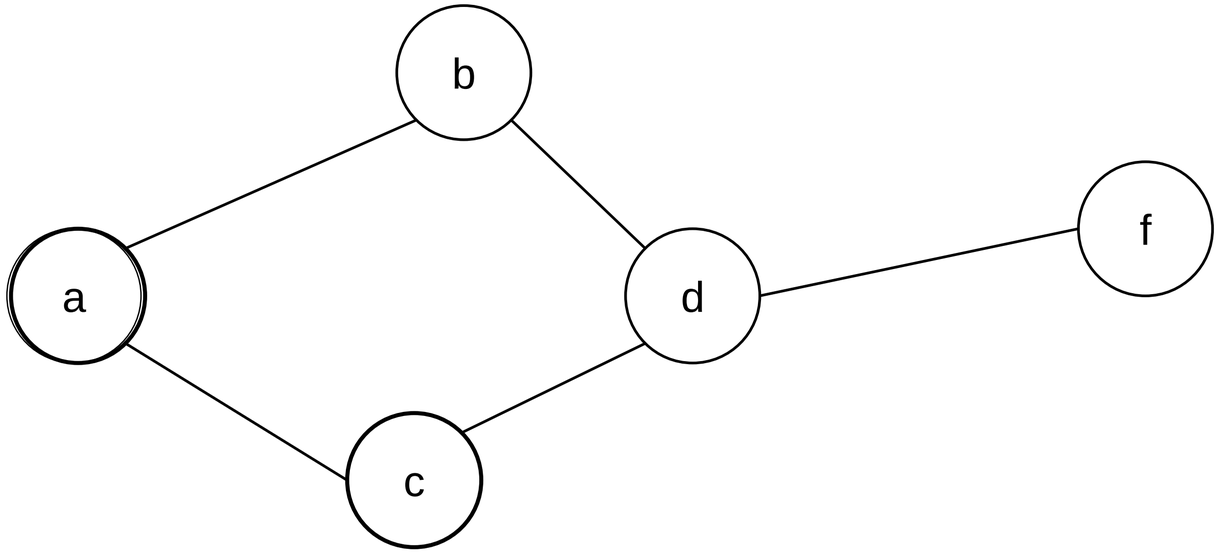


>>chemin 1 :

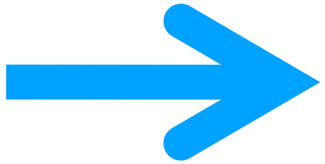


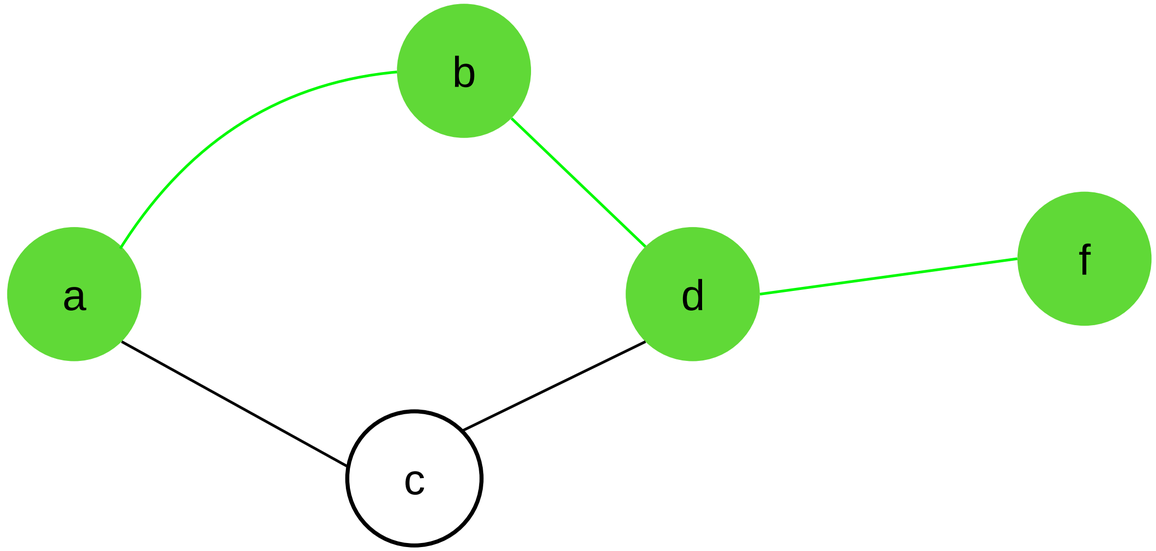
>>chemin 2 :



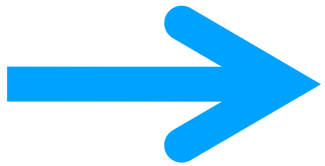


```
myFirstGraph{
        %TYPE{indirected}
        %DECLARE
                a -> b,1;
                a -> c,3;
                a -> d,6;
                b -> d,1;
                c -> d,6;
                d -> f,2;
}

Main{
        %SUBGRAPH
                myFirstGraph : test;
        %DECLARE
                Racine->test(a);
        %OPERATIONS

        Dijkstra(a, f , myFirstGraph);

}
```





```
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}]
```

La grammaire :

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

TYPE :: %type { TYPE' }

TYPE' :: directed | undirected

SUBGRAPH :: %subgraph ID : ID OTHERID INSTANCE | epsilon

INSTANCE :: ID : ID OTHERID INSTANCE | epsilon

OTHERID :: , ID OTHERID | ;

DECLARE :: %declare ID CHILD CHILDS

CHILD :: -> ID CHILD' | ;

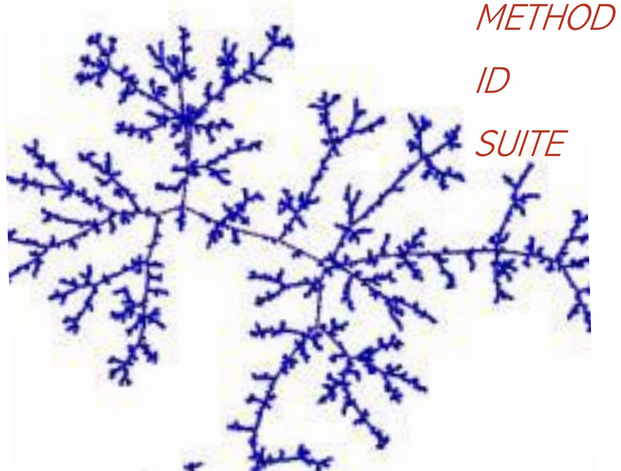
CHILD' :: POIDS CHILD | (ID') POIDS ;

ID' :: ID | epsilon

POID :: , chiffre | epsilon

CHILDS :: ID CHILD CHILDS | epsilon

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 }) ;

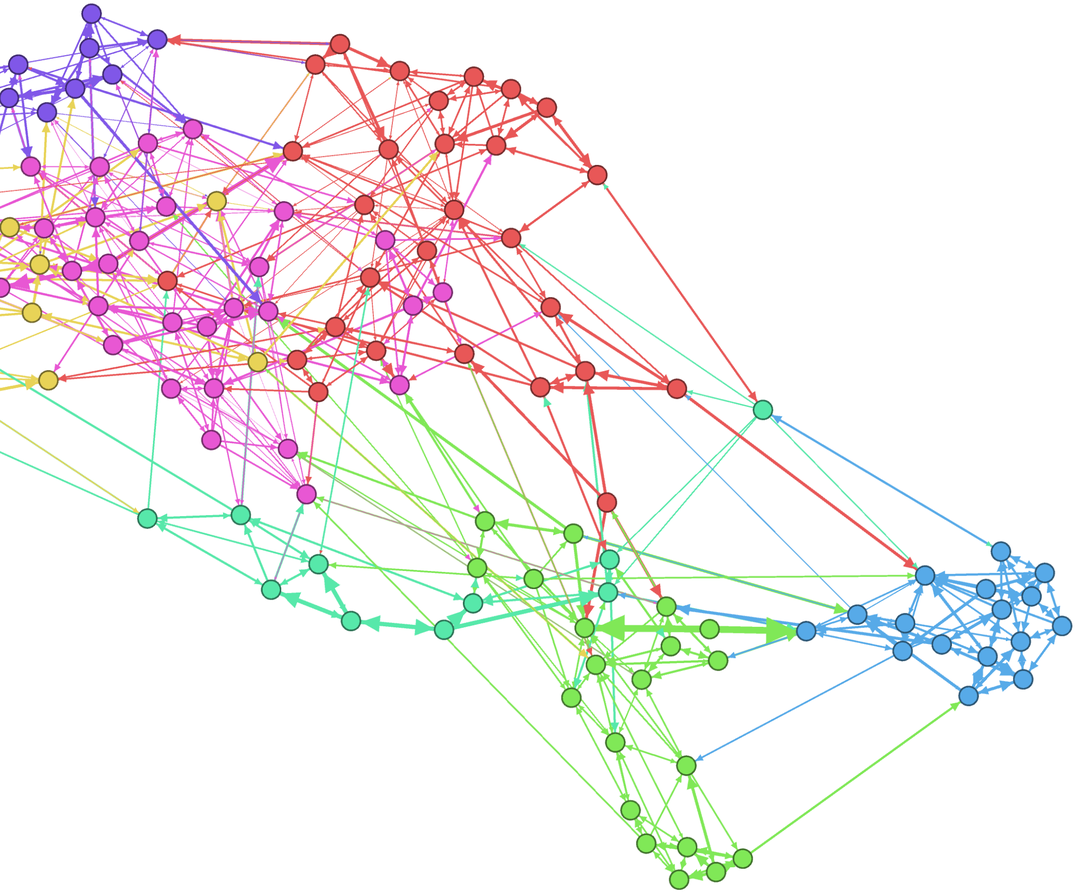
ID'' :: ID , | epsilon

METHOD :: dfs | bfs

ID :: _ SUITE | lettre SUITE

SUITE :: lettre SUITE | chiffre SUITE | epsilon

La syntaxe



```
GraphSecondaire {
        %TYPE {directed | undirected}
        %SUBGRAPH
            // Instanciation des sous-graphes.
        %DECLARE
            // Declaration des noeuds et arcs.
}
Main {
        %TYPE {directed | undirected}
        %SUBGRAPH
            // Declaration de sous-graphes.
        %DECLARE
            // Declaration de noeuds et arcs.
        %OPERATION
            // Appel des fonctions, et operations sur nos graphes
}
```

Les opérations possibles sur les graphes

- printAll([NomSousGraphe1]); // Affiche des listes de noeuds/arcs et les poids
- printNodes([NomSousGraphe1]); // Affiche la liste des noeuds dans un graphe
- printEdges([NomSousGraphe1]); // Affiche la liste des arcs du graphe main
- getChemin(node1, node2, [NomSousGraphe1]); // Affiche une liste contenant tous les chemins possibles entre node1 et node2
- getWeight(node1, node2, [NomSousGraphe1]);
- getNode(edge) // Retourne une liste des deux noeuds attachés à l'arc.
- exists({node1, node4, …}, [NomSousGraphe1]); // Verifie si un chemin existe dans le graphe mentionné (sinon main).
- minCost(noeudDebut, noeudFin, [NomSousGraphe1]); // Donne le coût minimal nombreChromatique([NomSousGraphe1]);
- colorier(node, color); // color -> #RED, #YELLOW, #GREEN, ...
- colorerGraph([NomSousGraphe1]);
- plot([NomSousGraph1]); // Affiche le graphe mentionné dans une nouvelle fenêtre (si aucun graphe n'est mentionné, on considère le main).
- Traverse([NomSousGraph1 ,] (DFS | BFS), (startNode, endNode, edge) => { }); // Traverse un graphe donné par l'une méthode (DFS ou BFS)
- Dijkstra(node1, node2 [, sousGraph]);
- Bellman(node1, node2 [, sousGraph]);
- DijkstraGeneralise(node1, node2 [, sousGraph]);
- Kruskal([sousGraph]);
- Prime([sousGraph]);

Merci pour votre aimable attention