FACULTÉ DES SCIENCES
UNIVERSITÉ D'ANGERS

Licence 3 Informatique
TP Algorithmique des Graphes
2023-2024

TP 1 : Implémentations de graphes, parcours de graphes, calcul de composantes connexes et applications

Ce TP est centré sur les parcours de graphe, en largeur et en profondeur, avec des premières applications directes. Il est à noter que par rapport aux parcours d'arbre vus en L2, le cas d'un graphe et de cycles potentiels implique de devoir marquer des sommets visités, pour ne pas parcourir un graphe à l'infini.

La recherche des composantes connexes d'un graphe, ou le certificat de connexité sont des premières applications des parcours simples de graphe. D'autres applications suivront, telle que le calcul des degrés de séparation, qui peut être vu comme un plus court chemin dans un graphe non pondéré, cas particulier de l'algorithme de Dijkstra.

Un code générique vous est fourni en C++, ré-implémentant des classes de graphe à partir des conteneurs de la `std`. On aurait pu utiliser une bibliothèque de graphe telle que celle fournie `boost`. Avoir une implémentation ouverte permettra de mieux utiliser de telles bibliothèques par la suite en étant sensibilisé aux subtilités des implémentations de graphes.

Consignes et pré-requis au TP, à lire attentivement

Rendu de TPs

Les TP seront évalués dans votre note de Contrôle Continu. De manière générale, les TPs sont volontairement longs, vous n'êtes pas obligés de tout terminer, nous indiquons pour chaque TP la partie minimale à réaliser. Le code sont à déposer sur Moodle, vous avez jusqu'au lundi 24 février pour déposer ce TP1. Comme en POO Java de L2, vous présenterez votre code à chaque session de TP. Pour ce TP, la partie minimale à réaliser va jusqu'à la section 4.2 incluse.

Il ne sera pas répondu aux questions en dehors des séances de TP. Assurez vous d'avoir bien compris les différentes notions du travail à finir sur la séance de TP. Les TP seront donnés en avance, pour vous permettre de les préparer avant la séance, et d'arriver avec vos questions et points de déblocages, pour finir en toute autonomie.

Notations et hypothèses pour ce TP

On considérera des graphes non orientés et non pondérés dans ce TP, que l'on note $G = (V, E)$ où l'ensemble des sommets est noté V, de cardinal n, et l'ensemble des arêtes E est de cardinal m. On étendra à des graphes pondérés aux TP 2 et 3.

On supposera que $V = [\![0, n-1]\!] = [0, n-1] \cap \mathbb{Z}$, de manière à pouvoir repérer exactement un sommet avec un indice de tableau. Les graphes fournis comme données d'entrée, du format DIMACS, sont numérotés entre 1 et n, la conversion sera faite au moment de l'import pour avoir des sommets indexés de 0 à $n-1$. De manière générale, si on a des sommets étiquetés comme des chaîne de caractères, il suffirait d'utiliser un `vector<string>` pour nommer chaque sommet suivant son indice entre 0 et $n-1$.

$\overline{G}$ désignera le graphe complémentaire de G, $\overline{G}$ a les mêmes sommets que G, mais les arrêtes "inverses" : v et v' sont reliés dans $\overline{G}$ si et seulement si v et v' ne sont pas reliés dans G.

Implémentation C++, le cas de vector<bool>

Pour l'implémentation C++, on pourra utiliser largement les `vector<bool>` dans ces TP. L'implémentation fournie des matrices d'adjacences les utilise, on il est intéressant aussi de les utiliser pour marquer des sommets parcourus, ou comme définition d'un sous- ensemble.

La taille minimale en mémoire est un octet. Déclarer `bool b` peut prendre en mémoire 1 ou 4 octets, alors qu'on pourrait espérer naïvement que cela ne prenne en mémoire qu'un bit. Un tableau de booléens, déclaré `bool* tab` et initialisé avec `tab = new bool[1024]` prend en mémoire 1024 octets soit 8192 bits au minimum

(ou 32768), alors que on a besoin de 1024 bits pour encoder une telle information. `vector<bool>` correspond à une telle implémentation, comme si 8 bits étaient encodés sur un seul octets. Cela est utile pour des grands graphes, les graphes de la vie réelle étant potentiellement très grands, la mémoire peut être un facteur limitant. Il est à noter qu'utiliser `bitset` peut présenter les mêmes avantages, on préférera `vector<bool>` pour sa facilité d'utilisation, `v[i]]` pour de i 0 à $n-1$ indique le booléen relatif à i, apr exemple, si i est marqué car déjà parcouru.

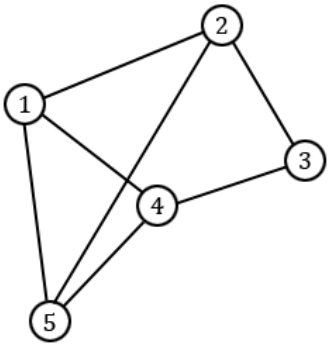
Un désavantage et une précaution à prendre avec un `vector<bool> v`, `&v[i]]` n'a plus forcément de sens, les implémentations fournies éviteront de tels usages. En pratique, vous n'aurez qu'à utiliser `v[i]]`

Implémentations de graphes, rappels de cours

Pour implémenter un graphe, deux grands types d'implémentations sont usuelles, par *liste d'adjacence* ou par *matrice d'adjacence*, comme illustré Figure 1.

La matrice d'adjacence est une matrice de taille $n \times n$ à valeur dans $\{0,1\}$, la coordonnée (i,j) dans la matrice indiquant avec un 1 si i et j sont reliés dans le graphe. Dans le cas d'un graphe non orienté, cette matrice est symétrique (égale à sa transposée, symétrique par rapport à la diagonale de gauche à droite).

Les listes d'adjacences associent à chaque sommet i, l'ensemble de ses voisins, comme une liste d'entiers distincts à valeur dans $\{0, n-1\}$




```
1 -> {2, 4, 5}
2 -> {1, 3, 5}
3 -> {2, 4}
4 -> {1, 3, 5}
5 -> {1, 2, 4}
```

Listes d'adjacence

$$M = \begin{pmatrix} 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 \end{pmatrix}$$

Matrice d'adjacence

FIGURE 1 – Illustration des représentations d'un graphe non orienté par matrice d'adjacence et liste d'adjacence

Code fourni

Le code fourni comprend une implémentation de matrices d'adjacence, `adjmat.hpp`, avec une classe abstraite et plusieurs implémentations. Ces classes seront explicitées plus loin.

Un canevas est fourni pour une classe abstraite dans `graph.hpp`, avec des méthodes virtuelles pures que chaque implémentation de graphe devra fournir :

```
using gint = unsigned int;
using vertex = gint;

class Graph {
public:
   virtual bool is_edge(vertex v1, vertex v2)=0; // true iff edges (v1,v2) exists
   virtual gint nb_vertex()=0;
   virtual gint nb_edges()=0;
   virtual gint degree(vertex v1)=0;
   virtual vector<vertex> neighbors(vertex v1)=0; //get neighbors sorted  in increasing order

   virtual void intersect_neighbors(vector<vertex>& vect, vertex v1);
   virtual void  union_neighbors(vector<vertex>& vect, vertex v1);
   virtual void  diff_neighbors(vector<vertex>& vect, vertex v1);

   void display_screen();
```

```
    void export_as_dimacs_file();

protected:
    virtual void reset(vertex nb_vertex)=0; // clear the graph, set a new  size  nb_vertex
    void generate_random( vertex nb_vertex, float proba_density);
    void import_dimacs_file( string & filename);
    virtual void add_edge(vertex v1, vertex v2)=0;
};
```

Différentes implémentations vous sont fournies. Dans un premier temps, il n'est pas nécessaire de regarder leurs implémentations spécifiques. On réalisera des implémentations avec les fonctions génériques de la classe `graph`, pour qu'elles soient valables (mais avec des efficacités et complexités potentiellement différentes) pour toutes les implémentations réalisées.

Plutôt que d'alourdir la classe mère, on écrira les implémentations dans un fichier à part, comme des fonctions statiques. On pourrait prendre en argument un `Graph * g`, pour être utilisées sur toutes les implémentations par un appel de constructeur différent, en utilisant du polymorphisme, comme cela était couramment utilisé dans le cours de C++. De manière assez équivalente, les fonctions prendront une référence constante en argument, `const Graph &g`, de manière à ne pas utiliser la syntaxe par pointeur dans le code de la fonction.

Pour tester les fonctions, on peut utiliser les implémentations `GraphAdjVectorSorted` ou `GraphAdjMatrix` efficaces pour des listes d'adjacences et matrice d'adjacence respectivement.

Un générateur de graphe aléatoire `generate_random` est fourni, permettant de spécifier une taille de noeuds et la probabilité de génération d'un arc, qui statistiquement correspondra à la densité du graphe généré. L'import et l'export de fichier DIMACS est fourni, pour se focaliser sur les différentes implémentations de graphes et sur des algorithmes de graphe. On trouvera en annexe de ce TP des détails sur ce format. Ces fonctions d'imports utilisent uniquement des méthodes virtuelles pures, et sont ainsi commune aux différentes implémentations de graphe.

On n'autorisera comme constructeur que la construction depuis un fichier DIMACS ou par génération aléatoire, les différents constructeurs appellent les méthodes `protected` implémentées dans la classe mère `import_dimacs_file` ou `generate_random`.

Instances

Des jeux de données DIMACS sont fournis dans le dossier instance, qui doit se trouver à côté des sources pour les chemins relatifs utilisés à l'import. Le nom de l'instance `filename` est converti en un chemin, `filename` correspond au nom de l'instance, le suffixe .col est ajouté en suffixe, ainsi que le chemin relatif en préfixe. On pourra trouver un ensemble complet d'instances DIMACS au lien : https://github.com/Cyril-Grelier/gc_instances/tree/main/original_graphs.

1 Premières fonctions

Cette partie réalise des petites implémentations, qui peuvent être réalisées avec les fonctions génériques de la classe `graph`, pour qu'elles soient valables (mais avec des efficacités et complexités potentiellement différentes) pour toutes les implémentations réalisées.

On écrira ces fonctions dans un fichier à part, tp1.cpp, les signatures sont fournies dans tp1.hpp, et un fichier de test est fourni dans main.cpp.

1.1. La fonction `float density(const Graph &g)`, qui calcule la densité, ie le nombre d'arêtes divisé par le nombre d'arête du graphe complet (ou clique) de même nombre de sommets. Attention à ne pas compter deux fois des arêtes, en non orienté, une clique de taille n a $n(n-1)/2$ arêtes.

1.2. La fonction `vertex max_degree(const Graph &g)` qui calcule le degré maximum d'un sommet du graphe.

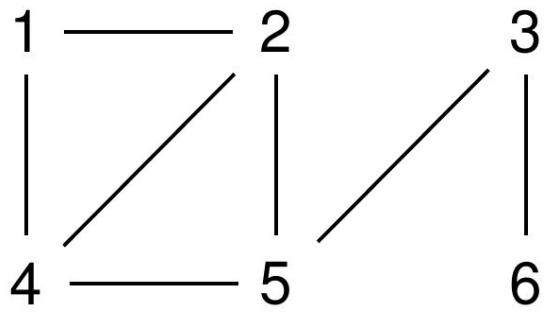
1.3. La fonction `std::pair<vertex,vertex> max_degree_vertex(const Graph &g)` qui calcule le degré maximum d'un sommet du graphe et renvoie l'indice d'un sommet de degré maximum. (Par exemple, le premier champ est le degré, le second l'indice d'un sommet de degré max).

2 Parcours complet : BFS, DFS

Comme le parcours d'un arbre (vus en L2), les graphes (orientés et non-orientés) peuvent être aussi parcourus en largeur ou en profondeur.

Le **parcours en largeur (BFS)** partant d'un sommet v, commence par visiter tous les voisins de v en visitant ensuite les voisins des voisins de v qui n'ont pas encore été visités, en les stockant dans un conteneur. L'algorithme se termine lorsque l'ensemble des sommets à visiter est vide. Un conteneur adapté au BFS est une file, les nouveaux sommets à ajouter seront parcourus les derniers.

A partir d'un sommet de départ v, le **parcours en profondeur (DFS)** poursuit toutes les sommets accessibles jusqu'à trouver un sommet déjà visité (ou jusqu'à un cul-de-sac). Après v, le premier voisin de v est visité, et on itère ainsi de suite le parcours sur ce voisin, en gardant en mémoire qu'on aura les autres voisins de v à visiter. Un conteneur adapté au DFS est une pile, les nouveaux sommets à ajouter seront parcourus les premiers.



BFS(G,1) = 1,2,4,5,3,6.

DFS(G,1) = 1,2,5,4,3,6.

Il est à noter qu'un arbre correspond à un graphe non orienté connexe et sans cycles (c'est une définition équivalente). Ces deux cas particuliers impliquent deux différences majeures.

Tout d'abord, on ne parcourt qu'une composante connexe, il n'est pas forcément possible de visiter tous les noeuds du graphe, on parcours uniquement la composante connexe du sommet de départ. Mettre comme critère d'arrêt de visiter les n sommets n'est pas un critère de terminaison valide en général, uniquement pour des graphes connexes. Le critère d'arrêt est ainsi d'avoir un ensemble vide des sommets à parcourir.

Ensuite, l'existence de cycles fait qu'on peut visiter plusieur fois les mêmes noeuds. Les parcours boucleraient sur un cycle si on n'y veille pas. Il faut ainsi marquer les soimmets lorsqu'on les visite, pour ne plus les visiter par la suite. On utilisera `vector<bool>`, comme expliqué en préambule.

2.1. Coder deux fonctions `vector<t_vertex> breadth_first_search(const Graph &g, const vertex start)` et `vector<t_vertex> depth_first_search_it(const Graph &g, const vertex start)` correspondants aux parcours BFS et DFS du graphe g, en démarrant du sommet $start$, remplissant le vector `vector<t_vertex>` avec les sommets ajoutés dans l'ordre du parcours (ajouts successifs par `push_back`). On codera ces fonctions de manière itérative, et on pourra avoir des implémentations très proches. On pourra utiliser des piles et files (`std::stack` et `std::queue`) ou une liste doublement chaînée `std::deque` permettant ajout et suppression en O(1) en début et fin de liste, pour les deux types d'implémentation.

2.2. (facultatif) Un parcours DFS se fait naturellement avec de la récursivité, la pile des travaux en cours étant équivalente à la pile de récursivité. Coder de telle manière une fonction récursive de parcours DFS.

3 Applications

3.1 Connexité, Calcul de composantes connexes

La première application des parcours, DFS ou BFS, est qu'ils génèrent la composante connexe du sommet de départ. On peut alors en déduire siu un graphe est connexe, et réaliser la décomposition d'un graphe en composantes connexes.

3.1.1. Coder la fonction `bool is_connex(const Graph &g)` qui renvoie true si et seulement si le graphe est connexe. On pourra faire un parcours en largeur ou en profondeur d'un sommet, et marquer les sommets atteignables par un tel parcours. Si au moins un des sommets n'est pas parcouru, le graphe n'est pas connexe.

3.1.2. Coder la fonction `vector<vector<t_vertex>> connected_components(const Graph &g)` réaliser la décomposition du graphe g en composantes connexes. Le premier vector indiquant les différentes composantes connexes, chacune étant codée avec un `vector<t_vertex>`. N.B : on aurait pu choisir d'encoder les composantes connexes comme un `vector<vector<bool>>`.

3.2 Théorème d'Euler-Hierholzer

Le théorème d'Euler, appelé aussi théorème d'Euler-Hierholzer, se décline en deux caractérisations :

- Un graphe connexe admet un parcours eulérien si et seulement si ses sommets sont tous de degré pair sauf au plus deux.
- Un graphe connexe admet un cycle eulérien si et seulement si tous ses sommets sont de degré pair.

3.2.1. Coder les fonctions `bool has_eulerian_path(const Graph &g)` et `bool has_eulerian_cycle(const Graph &g)` en utilisant la caractérisation du théorème d'Euler.

3.3 Plus courts chemins non pondérés, six degrés de séparation

Dans un graphe non pondéré, le problème de plus court chemin peut avoir un sens. C'est le cas dans la théorie des six degrés de séparation (aussi appelée théorie des six poignées de main) est une théorie qui évoque la possibilité que toute personne sur Terre peut être reliée à n'importe quelle autre au travers d'une chaîne de relations individuelles comprenant au maximum six maillons.

Une autre application est le nombre d'Erdös. Un graphe de chercheurs est défini avec une arrête entre chercheurs s'ils ont publié ensemble. Le nombre d'Erdös est la distance au célèbre et prolifique mathématicien Paul Erdös dans ce graphe.

Un parcours BFS peut être utilisé pour calculer de telles distances.

3.3.1. Coder une fonction `vector<t_vertex> dist(const Graph &g, const vertex start)` calculant les distances de chaque point du graphe g au sommet de départ, en considérant que chaque arrête est de poids un, en adaptant un parcours BFS. Lorsqu'un sommet est hors de la composante connexe de g, on lui affectera la valeur $n + 1$.

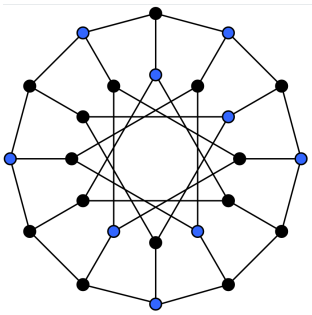
3.3.2. Quelle est la complexité de l'algorithme précédent ? Quelle est la complexité en utilisant l'algorithme de Dijkstra ? Expliquer la différence de complexité.

3.3.3. Calculer par `vector<vector<t_vertex>> distances(const Graph &g)` la la matrice des distances entre chaque couple de sommets.

3.3.4. Quelle est la complexité de l'algorithme précédent ? Quelle est la complexité en utilisant l'algorithme de Floyd Warshall ? Expliquer la différence de complexité.

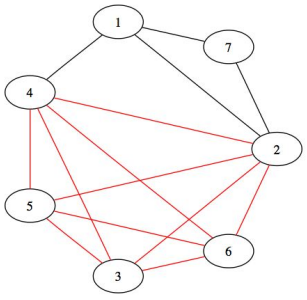
3.4 Stables maximums dans un graphe de degré deux

Dans un graphe non orienté $G = (V, E)$, un sous-ensemble de sommets $V' \subset V$ est un stable ("independent set" en anglais) ou une clique si :



V' est un stable ssi le sous graphe induit par V' dans G n'a pas d'arête :

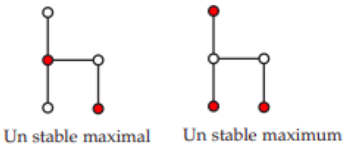
$$\forall v_1, v_2 \in V', \quad (v_1, v_2) \notin E$$



V' est une clique ssi le sous graphe induit par V' dans G est complet :

$$\forall v_1, v_2 \in V', \quad (v_1, v_2) \in E$$

Dans le graphe complémentaire, une clique (resp un stable) du graphe originel est un stable (resp une clique)


Un stable maximal Un stable maximum

Un stable $S \subset V$ d'un graphe $G = (V, E)$ peut être maximal ou maximum :

— S est un stable MAXIMAL : on ne peut plus ajouter de sommets dans S et avoir toujours un stable. C'est un max "au sens de l'inclusion"

— S est un stable MAXIMUM : il n'existe pas de stable dans G ayant un cardinal strictement plus élevé que S. C'est un max "de cardinalité"

— Un stable maximum est donc forcément maximal.

Idem pour les cliques, maximums et maximales se définissent de même. Une clique maximum (resp maximale) dans G est un stable maximum (resp maximal) dans $\overline{G}$ Un stable maximum (resp maximal) dans G est une clique maximum (resp maximale) dans $\overline{G}$

La recherche de stables et cliques maximums, ou de très grès grande taille est un problème fondamental de l'optimisation combinatoire, qui a de nombreuses applications. On verra ces problèmes régulièrement au cours des TPs. Ces deux problèmes étant équivalents par passage au complémentaire, on se focalisera sur le problème de recherche de stable maximums, le problème Maximum Independent Set (MIS) cherchant des stables les plus grands possibles au sens de la cardinalité.

Le problème MIS, se résout facilement dans un graphe de degré deux, cad où tous les sommets sont de degrés au plus deux. Un tel graphe a pour composantes connexes des cycles, des lignes, ou des sommets isolés. Un sommet isolé est nécessairement sélectionné dans MIS. Le long d'une ligne, en sélectionnant un sommet sur deux en partant d'une extrémité, on a une solution optimale de MIS. Sur un cycle, on a une solution optimale de MIS en sélectionnant également un sommet sur deux, en partant de n'importe quel sommet.

Au final, MIS sur un graphe de degré deux, se résout en calculant des composantes connexes par parcours. On résoudra un tel problème, après avoir codé des fonctions de vérification.

3.4.1. Coder une fonction `bool deg2(const Graph &g)` qui vérifie que le graphe est de degré 2.

3.4.2. Coder la fonction `bool is_clique(const Graph &g, vector<vertex> & candidate)` indiquant si un sous ensemble de sommets est une clique.

3.4.3. Coder la fonction `bool is_stable(const Graph &g, vector<vertex> & candidate)` indiquant si un sous ensemble de sommets est un stable (on dit aussi sous ensemble indépendant).

3.4.4. Coder une fonction `bool stable_maximal(const Graph &g, vector<bool> stab)` qui vérifie que `stab` est un stable dans le graphe g, et qu'on ne peut pas ajouter de sommet et rester stable, ie le stable est maximal (mais pas nécessairement maximum).

3.4.5. Coder une fonction`vector<bool> mis_deg2(const Graph &g)` qui après avoir vérifié que g est de degré 2, renvoie un `vector<bool>` où les sommets marqués à true sont sélectionnés dans la solution optimale de MIS renvoyée. la la matrice des distances entre chaque couple de sommets.

3.4.6. Pour tester une telle fonction, on codera un générateur de graphe de degré 2. On pourra le générer au langage de votre choix en générant un fichier au format DIMACS (solution la plus simple).

4 Implémentations de graphes non-orientés et non pondérés

Plusieurs implémentations de graphes par matrices d'adjacence et des listes d'adjacences ont été fournies. Ici, on se posera les questions différentes efficacités et cas appropriés des différentes implémentations. Plus précisément, on comparera des complexités des opérations élémentaires que sont :

— savoir si une arête existe (`is_edge`)

— ajouter une arête existe (`add_edge`), utilisée dans le parseur DIMACS

— calculer le degré d'un noeud, (`degree`), ie le nombre de ses voisins.

— récupérer les indices des voisins dans un `vector` trié dans l'ordre croissant.

— l'espace mémoire occupé pour encoder un graphe. On notera que le nombre d'arête m peut être de l'ordre de n ou de n^2, et on pourra penser à ces différents cas.

Des méthodes peuvent être virtuelles et implémentées génériquement à l'aide de méthodes virtuelles pures. Dans des tels cas, des implémentations spécifiques pourront être plus efficaces que la fonction générique, dans de tels cas, la méthode sera overridée. C'est par exemple le cas des fonctions ensemblistes qui font l'intersection/l'union/ la différence entre en ensemble donné et les voisins d'un sommet v qui utilisent dans l'implémentation fournie ici la méthode `neighbors()` et l'implémentation de ces opérations ensemblistes entre vector triés (comme `std::set_intersection`).

4.1. Y a t'il un intérêt à combiner liste d'adjacence et matrice d'adjacence ? Si oui, écrire une telle implémentation de graphe, et préciser la complexité des opérations élémentaires.

Annexe : Le format DIMACS

Le format DIMACS est le résultat d'un effort de standardisation de fichiers de entrée et sortie pour les différents DIMACS Implémentation Challenges, qui permet de comparer plus facilement les résultats obtenus. Ce format peut varier légèrement entre chaque challenge pour mieux répondre aux besoins de la compétition.

DIMACS pour les problèmes de Clique et Coloration de Graphes

Le *challenge* de 1992 portait sur les algorithmes de résolution de clique et coloration de graphes non-orientés et non-pondérés.

Les **fichiers d'entrée** considèrent que, pour un graphe $G = (V, E)$, l'ensemble des sommets V est numéroté entre 1 et n et le graphe contient exactement m arêtes.

Paramètre	Description	Exemple
Commentaires	Information utile pour les utilisateurs mais pas pour les programmes, donc ils doivent être ignorés	`c Ceci est un commentaire`
Graphe	Il y a exactement une ligne "`p edge`" dans chaque fichier. Le champ `nodes` indique le nombre des sommets. Le champ `arcs` indique le nombre des arcs.	`p edge <nodes> <arcs>`
Arêtes/Arcs	Caractère `e`. Les champs `<w>` et `<v>` indiquent les sommets concernés.	`e <w> <v>`

Les **fichiers de sortie** sont crées après l'exécution d'un algorithme sur un graphe. Ces fichiers contiennent une ou plusieurs lignes, qui prennent en considération le type d'algorithme et le problème résolu.

Paramètre	Description	Exemple
Solution	Caractère `s`. Le champ `<type>` indique le type du valeur à présenter. Le champ `<solution>` indique le valeur même.	`s <type> <solution>`
Délimitation	Caractère `b`. Limite inférieure du nombre de couleurs nécessaires pour colorier le graphe.	`b <bound>`
Étiquettes	Caractère `l`. Le champ `<v>` est le numéro du sommet. Le champ `<lb>` indique l'étiquette correspondante. Il y a une ligne d'étiquette pour chaque sommet du graphe.	`l <v> <lb>`

Annexe : Affichage d'un graphe

Un des langages le plus connu pour décrire un graphe est **DOT**, utilisé dans le logiciel de visualisation de graphes GraphViz. Pour obtenir une image PNG à partir d'un fichier DOT, on utilise la commande suivante : `dot -Tpng <monfichier.dot> -o <monfichier.png>`

Un autre option d'affichage est utiliser le langage **LaTeX avec le package TiKz** pour la création d'éléments graphiques. LaTeX est un langage de traitement de texte considéré comme "intermédiaire" entre les langages WYSIWYM (comme LibreOffice Write ou Microsoft Word) et les langage de balisage (comme HTML, XML,

etc.). Pour compiler un fichier `tex` vous pouvez utiliser `pdflatex <monfichier.tex>` dans votre terminal. Il y a aussi des éditeurs en ligne, comme LatexBase (https://latexbase.com)

Ci-dessous, un exemple (même graphe) dans les deux langages :

DOT

```
digraph D {
  rankdir=LR;

  1 [shape=circle]
  2 [shape=circle]
  3 [shape=circle]
  4 [shape=circle]
  5 [shape=circle]
  6 [shape=circle]

  1 -> 2 [arrowhead=none]
  1 -> 4 [arrowhead=none]
  2 -> 4 [arrowhead=none]
  2 -> 5 [arrowhead=none]
  3 -> 6 [arrowhead=none]
  4 -> 5 [arrowhead=none]
  5 -> 6 [arrowhead=none]
}
```

LaTeX+ TikZ

```
\documentclass{standalone}
\usepackage{tikz}

\begin{document}
\begin{tikzpicture}[
    vertex/.style={draw,circle,
       minimum height=.55cm},
    edge/.style={thick=1pt}
]
\node[vertex](1) at (0,2){};
\node[vertex](2) at (2,2){};
\node[vertex](3) at (4,2){};
\node[vertex](4) at (0,0){};
\node[vertex](5) at (2,0){};
\node[vertex](6) at (4,0){};

\draw(1)node{\tt1};
\draw(2)node{\tt2};
\draw(3)node{\tt3};
\draw(4)node{\tt4};
\draw(5)node{\tt5};
\draw(6)node{\tt6};

\draw[edge](1) to (2);
\draw[edge](1) to (4);
\draw[edge](2) to (4);
\draw[edge](2) to (5);
\draw[edge](3) to (6);
\draw[edge](4) to (5);
\draw[edge](5) to (6);
\end{tikzpicture}
\end{document}
```