

# TP 1 – Graphes orientés : implémentation et applications

Gaëtan Staquet

2026

Ce TP est découpé en deux parties. La première consiste à implémenter une structure de graphes et des algorithmes génériques. La seconde se concentre sur des applications de ces algorithmes.

Avant de vous lancer dans le TP, lisez attentivement les consignes suivantes.

- La première partie est la plus importante pour la suite du cours. Essayez quand même d’avancer le plus possible dans la seconde partie.
- Le langage à utiliser pendant les TP est Python.
- Vous êtes libre d’utiliser les outils que vous voulez. Vous trouverez en fin de ce PDF quelques conseils pour l’utilisation d’IA générative (Section A).
- Veuillez limiter votre usage de modules externes. En particulier, les bibliothèques de graphes (comme `NetworkX` ou `graph-tool`) sont interdites, étant donné que le but est d’implémenter les algorithmes par vous-même.
- **Il vous est demandé d’utiliser le même projet Python pendant l’ensemble des TP. Ceci vous permettra de mettre en application des bonnes pratiques à plus long terme, dans un contexte dans lequel les besoins évoluent et vous sont inconnus. Notamment, pensez à bien structurer, documenter et tester votre code en amont afin de pouvoir facilement le modifier plus tard.**
- À cette fin, créez un ou des modules pour la première partie. Le code pour la seconde partie peut ensuite se reposer sur ces modules. En d’autres termes, séparez les algorithmes génériques des applications spécifiques.

# Partie I — Graphes et algorithmes

À la fin de cette partie, vous aurez une implémentation de graphes orientés, ainsi que les algorithmes de base : DFS, BFS, tri topologique et identification des composantes fortement connexes.

## I.1. Exercice de modélisation

Avant de passer à l'implémentation, votre première tâche est de modéliser un problème via un graphe. De plus, en utilisant un algorithme vu en cours, vous serez en mesure de trouver une solution. **Travaillez sur papier**, un logiciel de dessin ou via une autre manière qui ne demande pas de programmer.

Considérons le problème des vases d'eau. Pour ce problème, nous avons trois vases : un qui peut contenir 4 litres, un de 3 litres et un de 1 litre. Une configuration des vases sera dénotée par un triplet indiquant le contenu de chaque vase dans le même ordre. Par exemple,  $(3, 0, 1)$  veut dire que le premier vase (de 4 litres) contient 3 litres, le deuxième (de 3 litres) en contient 0 et le dernier (de 1 litre) en contient 1.

Initialement, le vase de 4 litres est plein et les deux autres sont vides, c'est-à-dire la configuration  $(4, 0, 0)$ . On veut arriver à avoir 2 litres dans le grand vase et 2 litres dans le vase moyen. Pour y arriver, on peut choisir un vase  $A$  non vide et verser son contenu dans un autre vase  $B$  qui n'est pas déjà rempli. Cette opération s'arrête dans deux situations :

- Le vase  $A$  est vide, auquel cas tout ce qui était dans  $A$  se trouve maintenant dans  $B$ . Par exemple, depuis la configuration  $(3, 0, 1)$ , on peut verser le contenu du troisième vase dans celui du milieu pour obtenir  $(3, 1, 0)$ .
- Le vase  $B$  est complètement rempli, auquel cas  $A$  n'est pas nécessairement vide. Par exemple, depuis la configuration  $(4, 0, 0)$ , on peut verser 3 litres dans le vase du milieu et arriver en  $(1, 3, 0)$ , ou verser 1 litre dans le dernier vase et obtenir  $(3, 0, 1)$ .

Une fois l'opération terminée et la nouvelle configuration atteinte, on peut à nouveau choisir un vase et verser son contenu dans un autre et ainsi de suite.

Depuis la configuration  $(3, 0, 1)$ , il ne faut pas considérer l'opération de verser le vase du milieu vers un autre vase, vu qu'il est vide. Il ne faut pas non plus considérer l'opération de verser le grand vase dans le petit. En effet, le petit vase étant déjà plein, rien ne se passerait. Autrement dit, chaque opération effectuée doit changer le contenu d'exactly deux vases.

Attention : il n'est pas possible de verser uniquement deux litres et d'obtenir  $(2, 2, 0)$  depuis  $(4, 0, 0)$ . Une opération ne peut être interrompue en cours d'exécution. Ainsi, depuis la configuration  $(1, 2, 1)$ , on peut arriver en  $(0, 3, 1)$ , mais l'inverse n'est pas possible (étant donné qu'il faudrait réussir à verser 1 seul litre depuis le deuxième vase dans le premier, ce qui ne suffirait pas à vider l'un, ni à remplir l'autre).

**Exercice I.1.1.** Est-ce qu'un graphe orienté ou non orienté est le plus adapté pour modéliser le problème ? Justifiez brièvement.

**Exercice I.1.2.** Modéliser le problème via un graphe dont les sommets sont les configurations possibles de vases.

Le sommet de départ est  $(4, 0, 0)$ , *i.e.*, la situation initiale du problème. Seuls les sommets accessibles depuis ce sommet de départ doivent être dessinés. Par exemple, il est inutile d'ajouter un sommet pour la configuration  $(2, 0, 2)$ , vu qu'il est impossible de l'atteindre.

Le graphe final doit contenir 8 sommets et 26 arcs.

**Exercice I.1.3.** On souhaite déterminer le nombre minimum d'opérations nécessaires pour obtenir  $(2, 2, 0)$  depuis  $(4, 0, 0)$ . Quel algorithme vu en cours vous semble le plus adapté ? Justifiez brièvement.

**Exercice I.1.4.** Appliquez l'algorithme de votre choix sur le graphe. Donnez les grandes étapes permettant de retracer vos calculs. Par exemple, indiquez dans quel ordre vous parcourez les sommets ou les arcs.

## I.2. Implémentation de graphes

Nous allons maintenant passer à la partie implémentation en elle-même. Comme dans l'introduction de ce document, le langage à utiliser est Python. Il vous est demandé de créer un ou des modules pour cette partie et d'appliquer de bonnes méthodes de développement. Le code que vous écrirez aujourd'hui sera réutilisé et, potentiellement, modifié pour les autres TP. Pensez notamment à bien tester votre implémentation.

### I.2.1. Graphes orientés et programmation orientée objet

Pendant le cours, nous avons vu deux manières de représenter un graphe orienté en mémoire : les matrices d'adjacence et les listes de successeurs. Ici, vous implémenterez une troisième approche qui se repose sur une vision orientée objet.

Dans cette vision, un graphe est découpé en plusieurs classes. Pour fluidifier ce TP (et les suivants), veuillez suivre la structure suivante.

- Une classe **Vertex** (pour le nom anglais de sommet) qui contient le label du sommet et tous les arcs sortants.
- Une classe **Edge** (arc) qui contient les deux extrémités de l'arc.
- Une classe **DirectedGraph** qui possède tous les sommets.

Vous pouvez choisir librement comment stocker les arcs et les sommets (liste, ensemble, dictionnaire, *etc.*).

**Exercice I.2.1.** Définissez les classes comme indiqué ci-dessus. La classe **DirectedGraph** doit avoir les méthodes suivantes :

- `add_vertex(self, label: str)` qui crée un sommet avec le label donné et l'ajoute au graphe. La fonction doit retourner un objet permettant d'identifier le sommet ajouté. Vous pouvez choisir librement la façon d'identifier. Par exemple, vous pouvez retourner l'instance de **Vertex** directement ou encore la position

du sommet dans la liste de `DirectedGraph`.

- `add_edge(self, source, target)` qui prend en entrée deux sommets (retournés par la fonction précédente) et qui crée un arc de `source` vers `target`.
- `get_successors(self, source)` qui retourne un itérable (une liste ou un ensemble) de sommets qui sont les successeurs du sommet `source`.
- `get_vertices(self)` qui retourne un itérable (une liste ou un ensemble) contenant tous les sommets du graphe.

Pendant l'entièreté des TP, vous êtes libre de définir des méthodes supplémentaires. De même, vous pouvez ajouter autant de méthodes que vous désirez dans les `Vertex` et `Edge`.

**Exercice I.2.2.** Ajoutez une méthode à `DirectedGraph` qui construit un sous-graphe induit par un ensemble de sommets.

**Exercice I.2.3.** Décrivez en quelques phrases comment vous modifieriez la structure de classes pour supporter les graphes non orientés.

## I.2.2. Lecture et écriture de fichiers et visualisation

Afin de ne pas avoir à encoder les graphes directement dans le code, nous allons permettre de lire et écrire les graphes dans des fichiers. Ces fichiers seront au format DOT décrit sur la page Wikipédia : [https://en.wikipedia.org/wiki/DOT\\_\(graph\\_description\\_language\)](https://en.wikipedia.org/wiki/DOT_(graph_description_language)).

Sur la page Hippocampus du cours, vous trouverez une archive avec

- un fichier `dot.py` permettant de lire et écrire un graphe dans un fichier au format DOT ;
- plusieurs fichiers au format DOT. Ces fichiers peuvent vous servir pour construire une base de tests. Ils sont répartis dans plusieurs dossiers. Pour l'instant, contentez-vous du dossier `graph` qui contient plusieurs graphes provenant du cours ou de ce TP, ainsi que des nouveaux graphes, dont certains sont acycliques. Les autres dossiers seront utilisés dans la Partie II.

**Exercice I.2.4.** Ajoutez ces fonctions dans votre projet. Vous pouvez soit garder les fonctions dans leur module `dot.py` séparé et modifier les imports pour que le code s'exécute correctement, soit copier les fonctions dans votre module de graphe.

Il est possible de générer une image à partir d'un fichier DOT avec un des programmes de Graphviz (<https://graphviz.org/>). Par exemple, pour générer une image PNG à partir du fichier `FICHIER.dot`, exécutez la commande suivante dans un terminal :

```
dot -Tpng -O FICHIER.dot
```

L'exécutable crée une image `FICHIER.dot.png` que vous pouvez ouvrir pour visualiser le graphe.

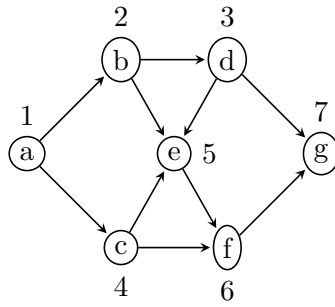


FIGURE 1 – Exemple d'un tri topologique.

### I.3. Algorithmes de parcours

Pour rappel, un parcours en profondeur d'abord se base sur une pile pour mémoriser les sommets traversés, mais qui ont des successeurs non encore vus. Ce parcours explore le graphe en allant le plus profondément possible, *i. e.*, sans garantir de trouver un chemin le plus court possible. À l'inverse, un parcours en largeur d'abord utilise une file pour stocker les sommets à parcourir et cherche un chemin de longueur minimum.

**Exercice I.3.1.** Implémentez une fonction permettant de trouver un chemin reliant deux sommets donnés via un parcours en **profondeur** d'abord.

**Exercice I.3.2.** Décrivez comment vous vous y prendriez pour ajouter les fonctions `previsite`, `visite` et `postvisite` vues en cours.

**Exercice I.3.3.** Implémentez une fonction permettant de trouver un chemin reliant deux sommets donnés via un parcours en **largeur** d'abord.

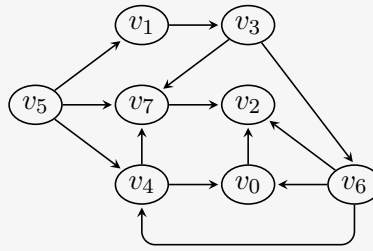
### I.4. Tri topologique

Le tri topologique indique dans quel ordre explorer les sommets d'un graphe **acyclique** de telle sorte que chaque sommet n'est parcouru qu'après avoir vu tous ses prédécesseurs. Un exemple est donné dans la Figure 1. Une application (voir Section II.3) du tri topologique est d'ordonnancer des tâches à effectuer afin de garantir d'avoir fini toutes les tâches préliminaires à une nouvelle tâche.

**Définition I.4.1 (Tri topologique).** Soit un graphe orienté  $\mathcal{G} = (V, E)$  et  $n = |V|$ . Une bijection  $f : V \rightarrow \{1, \dots, n\}$  est appelée un **tri topologique** si

$$\forall (u, v) \in E : f(u) < f(v).$$

**Exercice I.4.2.** Donnez un tri topologique pour le graphe suivant.



**Exercice I.4.3.** Prouvez qu'un graphe orienté est acyclique si et seulement s'il admet un tri topologique de ses sommets.

**Indice pour  $\Rightarrow$  :** si le graphe est acyclique, il existe un sommet sans arc sortant. Que peut-on en faire ?

**Indice pour  $\Leftarrow$  :** si on suppose avoir un tri topologique et un cycle, qu'est-ce qui est incohérent ?

---

**Algorithme 1.** Pseudo-code pour un algorithme de tri topologique.

---

```

1 : procédure TRI_TOPOLOGIQUE( $\mathcal{G} = (V, E)$ )
2 :    $V_{us} \leftarrow \emptyset$ 
3 :    $f : V \rightarrow \{1, \dots, |V|\}$  initialement vide
4 :   pour chaque sommet  $v$  de  $\mathcal{G}$  faire
5 :     si  $v \notin V_{us}$  alors
6 :        $V_{us} \leftarrow \text{PARCOURS}(\mathcal{G}, v, V_{us}, f)$ 
7 :        $f(v) \leftarrow \min\{\min_{u \in V} f(u), |V| + 1\} - 1$      $\triangleright$  On suppose que  $\min \emptyset = +\infty$ 
8 :   retourner  $f$ 

9 : procédure PARCOURS( $\mathcal{G} = (V, E), v, V_{us}, f$ )
10 :    $V_{us} \leftarrow V_{us} \cup \{v\}$ 
11 :   pour chaque successeur  $s$  de  $v$  dans  $\mathcal{G}$  faire
12 :     si  $s \notin V_{us}$  alors
13 :        $V_{us} \leftarrow \text{PARCOURS}(\mathcal{G}, s, V_{us})$ 
14 :        $f(s) \leftarrow \min\{\min_{u \in V} f(u), |V| + 1\} - 1$ 
15 :     sinon si  $f(s)$  n'est pas défini alors
16 :       Erreur
17 :   retourner  $V_{us}$ 

```

---

**Exercice I.4.4.** Expliquez avec vos propres mots le fonctionnement des deux procédures données dans l'Algorithme 1.

Pourquoi est-ce qu'atteindre la ligne 16 provoque une erreur ? Quelle condition doit satisfaire le graphe  $\mathcal{G}$  pour provoquer cette erreur ?

**Exercice I.4.5.** Implémentez l'algorithme pour calculer un tri topologique.

## I.5. Composantes fortement connexes

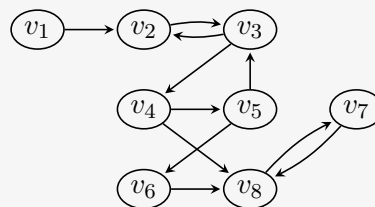
Pour rappel, un graphe est dit **fortement connexe** si, pour toute paire de sommets  $u$  et  $v$ , il existe un chemin qui relie  $u$  à  $v$ . Une **composante fortement connexe** d'un graphe est un sous-graphe induit, fortement connexe et maximal.

**Exercice I.5.1.** Est-ce que l'affirmation suivante est vraie ? Justifiez.

Si deux sommets  $u$  et  $v$  dans la même composante fortement connexe, alors il n'existe aucun chemin qui relie  $u$  à  $v$  et qui passe par des sommets hors de la composante fortement connexe.

En cours, vous avez vu l'algorithme de Kosaraju-Sharir pour identifier les composantes fortement connexes d'un graphe orienté. Il en existe un autre qui est bien connu : **l'algorithme de Tarjan**, donné dans l'Algorithme 2.

**Exercice I.5.2.** Appliquez l'Algorithme 2 sur le graphe suivant.



En vous basant sur votre exécution, expliquez, dans les grandes lignes, comment fonctionne l'algorithme.

Décrivez comment vous implémenteriez cet algorithme (choix des structures de données, éventuelles fonctions secondaires, *etc.*).

**Exercice I.5.3.** Quelle est la complexité de l'algorithme de Tarjan ?

Dressez une comparaison avec l'algorithme de Kosaraju-Sharir (voir les slides du cours), à la fois sur la complexité théorique et sur la facilité d'implémentation à priori.

**Exercice I.5.4.** Implémentez l'algorithme de Tarjan.

---

**Algorithme 2.** Algorithme de Tarjan

---

**procédure** TARJAN( $\mathcal{G} = (V, E)$ )

$indices : V \rightarrow \{1, \dots, |V|\}$  initialement vide

$lowlink : V \rightarrow \{1, \dots, |V|\}$  initialement vide

    pile  $\leftarrow$  nouvelle pile

    index  $\leftarrow 0$

    composantes  $\leftarrow$  liste (ou ensemble) vide

**pour chaque**  $v \in V$  **faire**

**si**  $indices(v)$  n'est pas défini **alors**

            CONNECTE( $\mathcal{G}, v, indices, lowlink, pile, index, composantes$ )

**retourner** composantes

**procédure** CONNECTE( $\mathcal{G}, v, indices, lowlink, pile, index, composantes$ )

$indices(v) \leftarrow index$

$lowlink(v) \leftarrow index$

    index  $\leftarrow index + 1$

    pile.EMPILER( $v$ )

**pour chaque** successeur  $w$  de  $v$  **faire**

**si**  $indices(w)$  n'est pas défini **alors**

            CONNECTE( $\mathcal{G}, w, indices, lowlink, pile, index, composantes$ )

$lowlink(v) \leftarrow \min(lowlink(v), lowlink(w))$

**sinon si**  $w$  est dans pile **alors**

$lowlink(v) \leftarrow \min(lowlink(v), lowlink(w))$

$\triangleright v$  est le début d'une nouvelle composante fortement connexe

**si**  $lowlink(v) = indices(v)$  **alors**

        compo  $\leftarrow$  liste (ou ensemble) vide

**répéter**

$w \leftarrow$  pile.RETIRER()

            compo.AJOUTER( $w$ )

**jusqu'à**  $w = v$

        composantes.AJOUTER(compo)

---



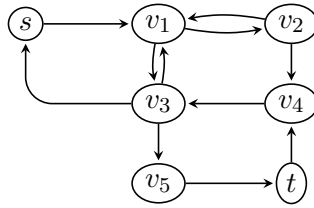


FIGURE 2 – Un exemple d’un environnement de type labyrinthe.

## Partie II — Applications

Cette partie se concentre sur des applications des algorithmes implémentés dans la première partie. Vous êtes libre de choisir sur quelle(s) application(s) vous vous concentrez et dans quel ordre. Il n’est pas demandé de tout réaliser. Le but est de proposer des exercices pour manipuler les graphes.

### II.1. Le problème des vases

Maintenant que vous avez les outils pour trouver un plus court chemin dans un graphe, vous pouvez vérifier vos réponses aux exercices de la Section I.1.

**Exercice II.1.1.** Implémentez une fonction qui génère le graphe des configurations accessibles depuis  $(4, 0, 0)$ .

Vérifiez vos réponses aux questions de la Section I.1 grâce à ce graphe.

**Note :** si vous exportez le graphe au format DOT, faites attention à retirer les espaces, virgules, parenthèses et autres caractères « spéciaux » des noms des sommets. Par exemple, le sommet pour la configuration  $(4, 0, 0)$  doit apparaître en tant que 400 dans le fichier.

### II.2. Exploration et génération d’environnements

Une application du parcours des graphes est de trouver un chemin entre deux points dans un environnement. Nous allons ici nous concentrer sur les environnements de types **labyrinthe** : nous avons un ensemble fini de positions, dont une position d’entrée et une position de sortie, et des couloirs qui relient des positions. Nous supposons que ces environnements sont décrits via des graphes **orientés** où chaque sommet représente une position et les arcs permettent de passer d’une position à l’autre. De plus, le sommet de départ est toujours  $s$  et celui d’arrivée est toujours  $t$ . La Figure 2 donne un exemple d’un tel environnement.

Dans l’archive des fichiers DOT, vous trouverez un dossier **environnements** qui contient plusieurs environnements sous la forme de graphes.<sup>1</sup>

1. Certains environnements ont été générés de manière aléatoire, sans aucune garantie de structure.

**Exercice II.2.1.** En supposant avoir toute la connaissance d'un tel environnement, est-il toujours possible de trouver un chemin de  $s$  à  $t$ ? Argumentez.  
Quels algorithmes vus en cours pourraient servir de base pour calculer un tel chemin?  
Quelles adaptations sont nécessaires, s'il y en a?  
À votre avis, quel algorithme minimiserait le temps d'exécution en pratique? Est-ce toujours le cas?

Les deux sections suivantes sont indépendantes et il n'est pas nécessaire d'avoir fini l'exploration pour faire la génération. À nouveau, vous pouvez choisir sur quoi vous travaillez.

### II.2.1. Découverte d'un environnement à la volée

On n'a pas nécessairement une connaissance complète de l'environnement. On voudrait donc pouvoir permettre de construire le graphe en même temps qu'on l'explore et qu'on découvre les arcs. Par exemple, on pourrait imaginer un robot qui doit livrer un colis dans un environnement inconnu.

On suppose que le robot n'a **aucune** connaissance a priori. En revanche, il est capable de repérer si des murs sont autour de lui et d'énumérer les positions par lesquelles il passe, tout en se rendant compte s'il repasse par une position déjà vue.<sup>2</sup>

Pour l'implémentation, nous allons séparer les tâches en deux parties :

- Une classe qui gère l'environnement et qui indique au robot sur quelle case il se trouve et les mouvements qui sont possibles. Plus précisément, l'environnement associe un nombre à chaque position du robot : sa position de départ a le numéro 0, la position suivante le numéro 1, *etc.* Si le robot passe plusieurs fois par le même sommet, le même numéro doit être donné. Autrement dit, on abstrait le graphe qu'on explore en changeant, à la volée, le nom des sommets.

Les mouvements possibles sont également énumérés. Par exemple, l'environnement retourne la liste (ou l'ensemble)  $[0, 1, 2]$  pour signifier que le robot a trois déplacements possibles, qui correspondent à trois successeurs dans le graphe. Faites attention à ce que les déplacements soient toujours les mêmes : le mouvement 0 depuis un sommet doit toujours amener sur le même sommet.

- Une fonction ou une classe qui représente le robot et les décisions qu'il prend durant son exploration.

**Exercice II.2.2.** Commencez par implémenter une classe permettant de manipuler un environnement tout en le cachant. La classe doit gérer la position du robot dans son monde, sans la révéler, et avoir les méthodes suivantes :

- `possible_moves` qui retourne une liste ou un ensemble avec les déplacements possibles du robot.
- `move_robot` qui déplace le robot selon le mouvement choisi.

---

2. En pratique, on peut justifier ce choix grâce à des capteurs sur le robot qui permettent d'identifier des caractéristiques propres à chaque position, comme la typologie du sol, la présence et la position de rochers ou de plantes, *etc.*

- `is_on_exit` qui retourne **True** si et seulement si le robot est sur la case de sortie de l'environnement.

Implémentez ensuite un robot qui découvre son environnement et cherche un chemin, tout en construisant un graphe des endroits où il est déjà passé et des arcs empruntés. Pensez à bien mémoriser quel mouvement vous faites à partir de chaque sommet, afin de pouvoir explorer d'autres mouvements plus tard. Pour ce faire, vous pouvez, par exemple, étendre la classe `Edge` pour y rajouter un nombre.

**Plusieurs robots.** Allons encore plus loin en considérant un cas où on a **plusieurs robots** dans l'environnement. Ces robots peuvent communiquer entre eux pour partager leurs connaissances du monde.<sup>3</sup> Plus précisément, **les robots vont construire séparément un graphe de l'environnement parcouru**. Lorsque deux robots se croisent, ils échangent leurs connaissances sur le monde et forment un graphe commun qui regroupe toutes les informations.

Notez que chaque robot a son propre graphe. La position numérotée 1 n'est pas nécessairement le même sommet pour tous les robots. En revanche, chaque robot démarre de  $s$  (*i.e.*, de la position numérotée 0) et mémorise les mouvements effectués. Vu qu'il n'est, en général, pas facile de fusionner les graphes de deux robots différents (étant donné les différences de numérotation des sommets des deux robots), vous pouvez, à la place, suivre l'idée suivante :

- Depuis le sommet 0 (qui est commun à tous les robots et qui correspond à  $s$ ), si les deux robots ont choisi le même mouvement et sont arrivés dans les sommets  $v_1$  et  $v'_1$ , alors  $v_1$  et  $v'_1$  correspondent à la même position de l'environnement.
- Une fois qu'on sait que  $v_1 \equiv v'_1$ , on peut recommencer le processus : vérifier les mouvements communs et fusionner les sommets d'arrivée.

**Exercice II.2.3.** Donnez un pseudocode qui formalise cet algorithme.

**Exercice II.2.4.** Modifiez la classe implémentée à la section précédente pour avoir plusieurs robots dans le monde et ajoutez la méthode `are_on_the_same_position` (qui indique si deux robots se trouvent au même endroit). Placez tous les robots sur  $s$  initialement.

Implémentez une fonction qui va se comporter comme une flotte de robots : pour chaque robot, le déplacer dans une direction et mettre à jour le graphe connu de **ce robot**. Lorsque deux robots sont sur la même case, ils partagent leurs informations et construisent un ou plusieurs graphes qui englobent les connaissances des deux robots.

**Exercice bonus : positions initiales aléatoires.** Finalement, considérons le cas où les robots commencent à des positions **aléatoires et inconnues** dans l'environnement. (L'environnement sait toujours où se trouvent les robots, mais eux ne le savent pas.) À nouveau, les robots peuvent se déplacer dans quatre directions et tester s'ils sont arrivés

3. On va ici supposer que la communication est parfaite (c'est-à-dire qu'il n'y a aucune perte lors des partages) et instantanée.

sur  $t$ . De plus, ils peuvent dorénavant tester s'ils sont sur  $s$ . Leur but est toujours de trouver un chemin de  $s$  à  $t$ .

**Exercice II.2.5.** Dans un premier temps, revenons au cas avec un seul robot. Qu'est-ce qui doit changer si la position initiale du robot est choisie au hasard ?

Le cas avec plusieurs robots est plus complexe. Vu que chaque robot a une information imparfaite et partent de positions différentes, la fusion des connaissances est (pratiquement) impossible dans un premier temps ; il faut que les deux robots qui communiquent soient préalablement passés par  $s$  et le sachent, afin de pouvoir associer les deux sommets des deux robots qui correspondent à  $s$ .

**Exercice II.2.6.** Décrivez un algorithme permettant de gérer la fusion des graphes en cas de positions initiales aléatoires.  
Que changeriez-vous dans votre code pour implémenter cet algorithme ?

### II.2.2. Génération d'environnements

Pour la génération aléatoire d'environnements, nous allons nous concentrer sur un dérivé du parcours en profondeur. L'énoncé donne ici les grandes lignes. Plus de détails sont disponibles sur Wikipédia : [https://en.wikipedia.org/wiki/Maze\\_generation\\_algorithm#Randomized\\_depth-first\\_search](https://en.wikipedia.org/wiki/Maze_generation_algorithm#Randomized_depth-first_search).

En partant du point de départ de l'environnement, on choisit aléatoirement une direction. Si la case correspondante n'a pas déjà été visitée, on ajoute un arc entre les deux sommets dans le graphe. On recommence récursivement sur ce nouveau sommet. Quand tous les voisins ont déjà été visités, on remonte dans la pile d'appels récursifs et on retente avec une autre direction depuis le sommet précédent.

**Exercice II.2.7.** Implémentez une fonction pour générer un environnement via un DFS aléatoire.

## II.3. Ordonnancement de tâches

Lors de la réalisation d'un projet, la complétion de certaines tâches est nécessaire pour pouvoir commencer de nouvelles tâches. Un problème principal est alors de déterminer un bon ordre de réalisation de ces tâches.

**Exercice II.3.1.** Dans un projet, on a 7 tâches (a à g) à réaliser avec les contraintes suivantes :

- la tâche a doit être finie avant de commencer b et c ;
- la tâche b doit être finie avant de commencer d et e ;
- la tâche c doit être finie avant de commencer e et f ;
- la tâche d doit être finie avant de commencer e et g ;
- la tâche f doit être finie avant de commencer g ;
- la tâche e doit être finie avant de commencer f.

Donnez un graphe qui modélise ce projet. Dans quel ordre réaliser les tâches ? Que se passe-t-il si la tâche f est un pré-requis de la tâche b ? Déduisez-en une condition importante que le graphe doit satisfaire.

**Exercice II.3.2.** Expliquez en quelques mots pourquoi un tri topologique donne un bon ordre pour réaliser les tâches.  
Déduisez-en un pseudo-code pour calculer dans quel ordre réaliser les tâches d'un projet donné sous la forme d'un graphe.

Vu qu'une version plus étendue de l'ordonnancement, prenant en compte le temps nécessaire pour réaliser les tâches, sera vue plus tard, aucune implémentation n'est demandée ici.

## II.4. Résolution de formules 2-CNF (2-SAT)

Dans le cours d'algorithmique avancée, vous avez vu le problème de satisfiabilité de formules booléennes, données sous forme normale conjonctive (**CNF**), connu sous le nom de SAT. Pour rappel, SAT est un problème NP-complet ; à ce jour, il n'y a pas d'algorithmes pouvant trouver une réponse en temps polynomial. Cependant, certaines instances spécifiques du problème sont plus faciles à résoudre. Notamment, un groupe d'instances faciles a un algorithme de résolution basée sur les graphes.

Cette section est longue et exploite tous les algorithmes implémentés dans la Partie I, mais peut déboucher à des utilisations concrètes. Comme le but de ce cours est l'utilisation de graphes, nous nous arrêterons à l'algorithme de résolution. Quelques pistes d'applications seront cependant données.

**Rappels de la terminologie.** Avant d'introduire l'algorithme, un petit rappel des termes s'impose.

- Une **variable propositionnelle** peut être soit vraie, soit fausse. Le but du problème est de trouver une **assignation** des variables qui rende la formule vraie.
- Un **littéral** est soit une variable positive, soit sa négation. Par exemple, si  $x$  est une variable, deux littéraux sont possibles :  $x$  et  $\neg x$ .
- Une **clause** est, dans notre cas, une disjonction de littéraux.
- Une **formule CNF** est une conjonction de clauses.

### II.4.1. Formules 2-CNF et graphes d'implication

On va se concentrer sur les formules CNF dont chaque clause contient exactement deux littéraux. Par exemple,

$$(x \vee \neg y) \wedge (\neg x \vee y) \wedge (\neg x \vee \neg y) \wedge (x \vee \neg z)$$

est une formule **2-CNF**.<sup>4</sup> L'algorithme présenté ici vient de [1].

---

4. Dans le cas où la formule contient une clause qui est juste  $x$ , on la remplace par  $x \vee x$  pour respecter ce format.

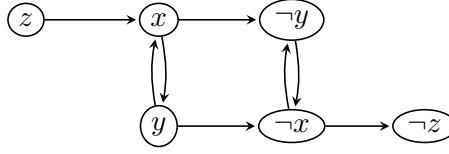


FIGURE 3 – Graphe d'implication pour la formule  $(x \vee \neg y) \wedge (\neg x \vee y) \wedge (\neg x \vee \neg y) \wedge (x \vee \neg z)$ .

Dans un premier temps, à partir de cette formule, on va construire un graphe orienté appelé **graphe d'implication**. L'idée se repose sur les équivalences

$$(x \vee y) \equiv (\neg x) \Rightarrow y \quad \text{et} \quad (x \vee y) \equiv (\neg y) \Rightarrow x.$$

Le graphe d'implication va regrouper toutes les implications qui découlent des clauses de la formule 2-CNF. Ainsi, on va utiliser les deux équivalences ci-dessus pour générer **deux arcs par clause**.

Dans un deuxième temps, on va analyser ce graphe pour en déduire si des implications sont contradictoires. Par exemple, si on obtient que la variable  $x$  doit être vraie et fausse à la fois, alors il n'existe pas de bonne assignation des variables.

Commençons par définir le graphe d'implication. Étant donné une formule  $\phi$  à  $n$  variables propositionnelles  $x_1, \dots, x_n$  et  $m$  clauses, on construit un graphe  $\mathcal{G} = (V, E)$  avec

- $V = \{x_i, \neg x_i \mid i \in \{1, \dots, n\}\}$ , *i. e.*, on a  $2n$  sommets : pour chaque variable, on a un sommet qui représente sa valeur positive et un pour sa valeur négative.
- Pour chaque clause  $x_i \vee x_j$ , on définit deux arcs : un qui part de  $\neg x_i$  et qui va en  $x_j$ , et un qui part de  $\neg x_j$  pour aller en  $x_i$ . De la même manière, pour chaque clause  $x_i \vee \neg x_j$ , on définit les arcs  $(\neg x_i, \neg x_j)$  et  $(x_j, x_i)$  ; chaque  $\neg x_i \vee x_j$  donne  $(x_i, x_j)$  et  $(\neg x_j, x_i)$  ; et chaque  $(\neg x_i \vee \neg x_j)$  donne  $(x_i, \neg x_j)$  et  $(x_j, \neg x_i)$ . Ainsi,  $\mathcal{G}$  a  $2m$  arcs.<sup>5</sup>

La Figure 3 donne le graphe d'implication pour la formule donnée en exemple plus haut. Remarquez que ce graphe a une structure qui est assez symétrique : **si on a l'arc  $(x, y)$ , on a l'arc  $(\neg y, \neg x)$** .

**Exercice II.4.1.** Prouvez l'affirmation suivante.

Soient  $\mathcal{G}$  le graphe d'implication d'une formule 2-CNF  $\phi$  et  $v_\ell, v_{\ell'}$  les deux sommets de  $\mathcal{G}$  pour les littéraux  $\ell$  et  $\ell'$  de  $\phi$ . S'il existe un chemin de  $v_\ell$  à  $v_{\ell'}$ , alors il existe un chemin de  $v_{\neg \ell'}$  à  $v_{\neg \ell}$ .

À partir du graphe d'implication  $\mathcal{G}$  d'une formule 2-CNF  $\phi$ , on peut décider si  $\phi$  est satisfiable en cherchant les **composantes fortement connexes** de  $\mathcal{G}$  et en vérifiant que nous n'avons **pas à la fois  $x$  et  $\neg x$  dans la même composante fortement connexe**.

**Théorème II.4.2.** Soit  $\mathcal{G}$  le graphe d'implication d'une formule 2-CNF  $\phi$ . La formule  $\phi$  est satisfiable si et seulement si, pour toute composante connexe  $\mathcal{C}$  de  $\mathcal{G}$  et pour toute variable  $x$  de  $\phi$ , on a que  $x$  n'est pas un sommet de  $\mathcal{C}$  ou  $\neg x$  n'est pas un sommet de  $\mathcal{C}$ .

5. On suppose ici qu'il n'y a pas deux fois la même clause dans la formule 2-CNF.

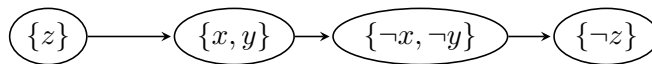


FIGURE 4 – Le graphe des composantes fortement connexes de la Figure 3.

**Exercice II.4.3.** Donnez un pseudo-code pour tester si une formule 2-CNF est satisfiable. Quelle est la complexité de cet algorithme ?

## II.4.2. Trouver une assignation des variables

On peut aller plus loin et trouver une assignation des variables qui rend la formule vraie (si cela est possible) grâce aux composantes fortement connexes et à un tri topologique.

Dans un premier, nous **réduisons** le graphe d'implication  $\mathcal{G} = (V, E)$  à ses **composantes fortement connexes**  $\mathcal{C}_1, \dots, \mathcal{C}_k$ . Pour ce faire, on définit un nouveau graphe  $\mathcal{R} = (V^{\mathcal{R}}, E^{\mathcal{R}})$  tel que

- les sommets de ce graphe sont les composantes fortement connexes, *i. e.*,  $V^{\mathcal{R}} = \{\mathcal{C}_1, \dots, \mathcal{C}_k\}$ , et
- il existe un arc de  $\mathcal{C}_i$  à  $\mathcal{C}_j$  si et seulement si il existe un arc qui part d'un sommet de  $\mathcal{C}_i$  pour arriver dans un sommet de  $\mathcal{C}_j$ . Autrement dit, on peut passer de la première composante à la seconde. Formellement,

$$E^{\mathcal{R}} = \{(\mathcal{C}_i, \mathcal{C}_j) \mid \exists u \in \mathcal{C}_i, v \in \mathcal{C}_j : (u, v) \in E\}.$$

La Figure 4 donne le graphe des composantes fortement connexes du graphe de la Figure 3. Remarquez que ce graphe est **acyclique**.

**Exercice II.4.4.** Est-ce que l'affirmation suivante est vraie ? Justifiez en quelques mots ou donnez un contre-exemple.

Pour tout graphe orienté  $\mathcal{G}$  et son graphe des composantes fortement connexes  $\mathcal{R}$ ,  $\mathcal{R}$  est acyclique.

Supposons maintenant que, pour toute variable  $x$  de la formule  $\phi$ ,  $x$  et  $\neg x$  n'appartiennent **pas** tous deux à la même composante fortement connexe. Par le Théorème II.4.2, il existe donc une assignation des variables qui rend  $\phi$  vraie. Pour la calculer, on va exploiter le fait que  $\mathcal{R}$  est acyclique, ce qui nous permet de calculer un **tri topologique** des composantes fortement connexes.

On parcourt ensuite les composantes fortement connexes dans l'**ordre inverse** du tri topologique (on commence par la composante avec le plus grand nombre et on descend jusqu'à la composante numérotée 1). Si les variables de la composante fortement connexe n'ont pas encore de valeur, on fixe une valeur de manière que chaque littéral soit vrai. Par exemple, si la composante est  $\{\neg x, \neg y\}$ , on fixe  $x$  et  $y$  à faux.

L'Algorithme 3 donne un pseudo-code qui résume les étapes de calcul.

---

**Algorithme 3.** Trouver une valuation des variables dans une formule 2-CNF.

---

**Nécessite :**  $\phi$  une formule 2-CNF

```
1 : procédure CALCULERVALUATION( $\phi$ )
2 :    $\mathcal{G} \leftarrow$  le graphe d'implication de  $\phi$ 
3 :    $\{\mathcal{C}_1, \dots, \mathcal{C}_k\} \leftarrow$  les composantes fortement connexes de  $\mathcal{G}$ 
4 :   pour chaque composante  $\mathcal{C}_i$  faire
5 :     si il existe une variable  $x$  telle que  $x \in \mathcal{C}_i$  et  $\neg x \in \mathcal{C}_i$  alors
6 :       retourner  $\phi$  n'est pas satisfiable
7 :    $\mathcal{R} \leftarrow$  le graphe réduit des composantes fortement connexes de  $\mathcal{G}$ 
8 :    $[\mathcal{C}_{i_1}, \dots, \mathcal{C}_{i_k}] \leftarrow$  tri topologique de  $\mathcal{R}$ 
9 :   pour chaque  $j$  de  $k$  à 1 faire
10 :    si les variables de  $\mathcal{C}_{i_j}$  n'ont pas encore de valeur alors
11 :      Assigner une valeur à chaque variable de manière à rendre chaque littéral
12 :      vrai.
12 : retourner l'assignation
```

---

**Exercice II.4.5.** Appliquez cet algorithme sur les formules suivantes

$$(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2) \wedge (x_1 \vee \neg x_3) \\ (x_1 \vee x_2) \wedge (\neg x_1 \vee x_3) \wedge (x_2 \vee x_2) \wedge (\neg x_4 \vee \neg x_2) \wedge (\neg x_3 \vee x_4)$$

**Exercice II.4.6.** Quelle est la complexité de l'algorithme ?

**Exercice II.4.7.** Implémentez une fonction qui prend en entrée une formule 2-CNF (dont vous choisissez librement l'encodage en mémoire) et qui retourne une valuation des variables qui rend la formule vraie, si elle existe.

**En pratique.** Il n'est **pas** possible de transformer n'importe quelle formule CNF en une formule 2-CNF. Par conséquent, les problèmes qui peuvent être résolus avec l'algorithme sont plus limités que dans le cas général. En particulier, n'importe quel problème NP-complet ne peut pas être résolu via cette méthode.

Il existe quand même plusieurs problèmes concrets pour lesquels on peut trouver une solution en temps polynomial grâce à une formule 2-CNF. Vous trouverez une liste (non exhaustive) sur Wikipédia : <https://en.wikipedia.org/wiki/2-satisfiability>.



# Partie III — Annexe

## A. Notes sur l'utilisation d'une IA générative

Mon but ici n'est pas de vous dire comment écrire vos prompts, mais plutôt de se concentrer sur **quand** utiliser une IA générative. Vu que l'objectif de ce cours est de vous faire manipuler les graphes et leur implémentation, évitez de demander directement « une fonction qui calcule le tri topologique ». À la place, décrivez la structure générale de l'algorithme (via un pseudo-code, par exemple).

De même, si vous voulez générer des classes et méthodes, faites le travail en amont de découper le problème en sous-classes et décrivez ce que chaque méthode doit faire. Par exemple, c'est ce qui est fait dans l'Exercice I.2.1.

Vérifiez également attentivement que le code généré respecte bien vos instructions. Cela vous entraînera à faire de la relecture de code ce qui vous sera utile dans la gestion de projets informatiques.

En quelques mots, je vous demande de penser par vous-même à la partie *ingénierie* du projet. Vous êtes libre d'utiliser une IA pour « boucher les trous » du découpage.

## Références

- [1] Bengt ASPVALL, Michael F. PLASS et Robert Endre TARJAN. « A Linear-Time Algorithm for Testing the Truth of Certain Quantified Boolean Formulas ». In : *Inf. Process. Lett.* 8.3 (1979), p. 121-123. DOI : 10.1016/0020-0190(79)90002-4.