Diplôme inter-universitaire “Enseigner l’Informatique au Lycée” (DIU-EIL)

Polycopié d’exercices (Bloc 5)

Table des matières

Résumé

Ce cahier Ce cahier d'exercices a été réalisé pour l'enseignement "Bloc 5" du DIU "Enseigner l'Informatique au Lycée", proposé par l'université Lyon1 en collaboration avec d'autres partenaires (Lyon2, ENSL, ENSSIB ...)

Il peut être librement distribué (License CC-BY-SA 4.0).

Intervenants Les personnes suivantes ont été impliquées dans l'écriture de ces exercices et des supports Python associés, et l'enseignement correspondant au sein du Bloc 5 :

— 2020 : Laure Gonnord, Julien Velcin, ...

`https://diu-eil.univ-lyon1.fr/bloc5/index.html`

Contenu du bloc 5 À la fin on mettre les compétences acquises, normalement.

TD 1
Graphes (1) - exercices

Sources

- `https://www.irif.fr/~francoisl/DIVERS/l3algo1617-TD2.pdf`
- `https://www.irif.fr/~francoisl/DIVERS/l3algo-td5-1011.pdf`
- `http://www.gymomath.ch/javmath/polycopie/th_graphe4.pdf`
- Feuille de TD L3 ENSL.
- Diverses Annales

1.1 Exercices élémentaires (TD)

1.1.1 Degré, connexité

EXERCICE #1 ▸ **Une preuve par construction**
Un graphe est dit k-régulier si tous ses sommets sont de degré k. Prouver la propriété suivante :

Pour tout entier n pair, $n > 2$, il existe un graphe 3-régulier composé de n sommets.

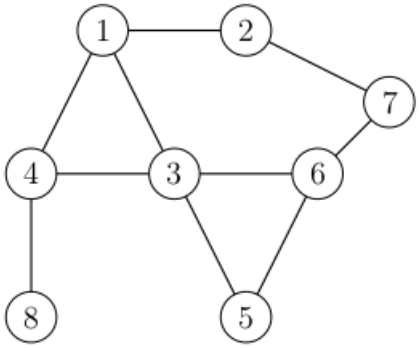
EXERCICE #2 ▸ **Degré**
Si m est le nombre d'arêtes d'un graphe G, montrer que

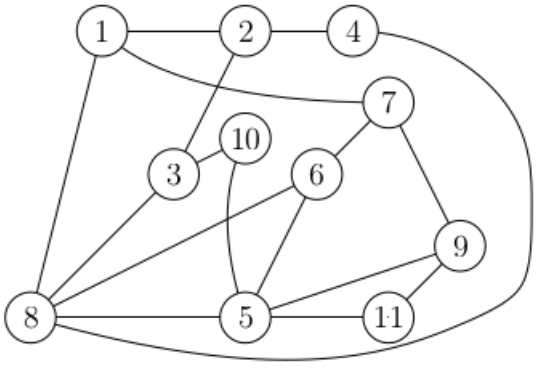
$$\sum_{v \in V(G)} d_G(v) = 2m.$$

EXERCICE #3 ▸ **Connexité**
Montrez que tout graphe connexe à n sommets a au moins n arêtes.

1.1.2 Parcours

EXERCICE #4 ▸ **Parcours en largeur**
Pour chacun des graphes suivants, donner l'ordre des noeuds rencontrés lors d'un parcours en largeur, en partant du sommet 1. Donner l'arbre résultant de ce parcours.





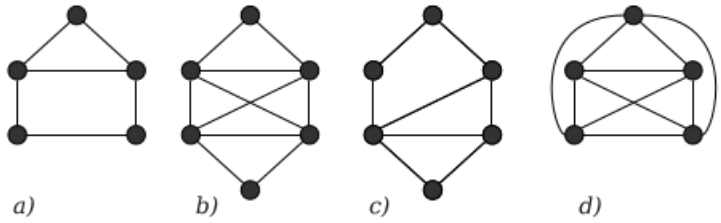
Quelle est la complexité du parcours en largeur avec les listes d'adjacence ?

EXERCICE #5 ▸ **Profondeur**
Donner l'ordre des noeuds visités dans le parcours en profondeur des deux graphes précédents à partir du noeud 1.

EXERCICE #6 ▸ **Applications des parcours**
Proposer un algorithme qui permet de déterminer si un graphe contient un cycle.

1.1.3 Cycles : graphes eulériens/hamiltoniens

EXERCICE #7 ▸ **Cycles Eulériens**
Un chemin est dit eulérien si il passe une et une seule fois par chacune des *arêtes* du graphe. Les graphes suivants possèdent-ils un cycle eulérien ?



EXERCICE #8 ▸ **CNS Chemin Eulérien**
Montrer qu'un graphe admet un chemin eulérien ssi il est connexe et au plus 2 de ses sommets sont de degré impair.

EXERCICE #9 ▸ **Graphes Hamiltoniens**
Un cycle est dit Hamiltonien ssi il passe une et une seule fois par chaque sommet du graphe.
Les graphes suivants possèdent-ils un cycle hamiltonien ?

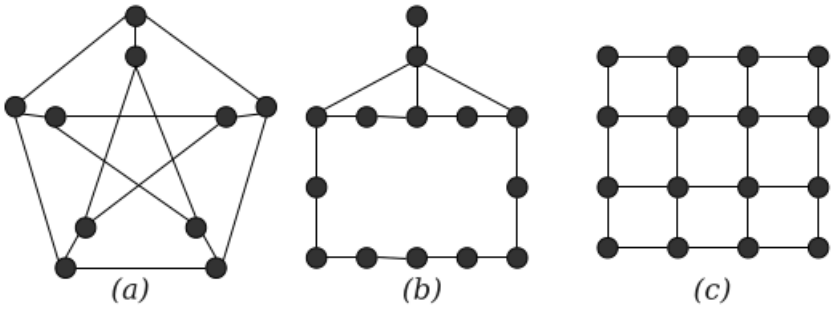
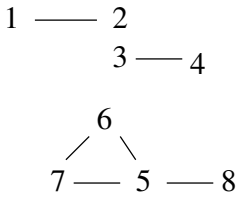


FIGURE 3 – (a) Graphe de Petersen 3-régulier, (b) maison agrandie, (c) grille

Donner un algorithme pour déterminer si un tel cycle existe.

1.1.4 Coloriages

EXERCICE #10 ▸ **Un exemple simple**
Avec l'algorithme polynomial vu en cours (simplification de Kempe), colorier le graphe suivant :

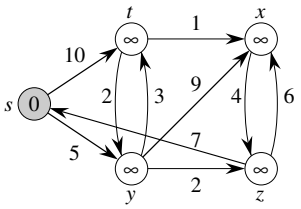


EXERCICE #11 ▸ **Coloriage et Bipartisme**
On dit qu'un graphe est biparti si on peut partitionner ses sommets en deux ensembles V_1 et V_2 de sorte qu'il n'y ait aucune arête entre deux sommets de V_1 (resp. de V_2). Les seules arêtes joignent donc un sommet de V_1 à un sommet de V_2.

1. Montrer qu'un graphe à n sommets est n-coloriable. Donner un graphe à 5 sommets qui n'est pas 4 coloriable.
2. Montrer qu'un graphe est deux coloriable ssi il est biparti.

1.1.5 Distances

EXERCICE #12 ▸ **Dijkstra**
Appliquer l'algorithme au graphe suivant (source = s) :

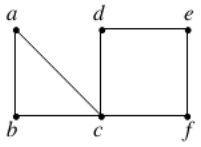


1.1.6 Exercices plus avancés

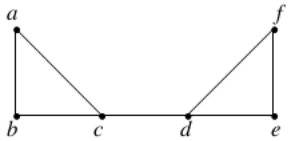
EXERCICE #13 ▸ **Application : fiabilité des réseaux - notion de coupure - facultatif**
Si on considère un réseau informatique où tout le monde doit communiquer avec tout le monde, il est important que le graphe associé soit connexe. Maintenant, il faut aussi que le retrait d'une machine (ou d'un lien) soit sans douleur, d'où les définitions suivantes :

> Le retrait d'un sommet et de toutes les arêtes incidentes à ce sommet conduit à former un sous-graphe avec plus de composantes connexes que dans le graphe initial. Ces sommets sont appelés **points de coupure**. Le retrait d'un point coupure à partir d'un graphe connexe produit un sous-graphe qui n'est pas connexe. De façon similaire, une arête dont le retrait produit un graphe avec plus de composantes connexes que dans le graphe initial est appelée un **séparateur**.

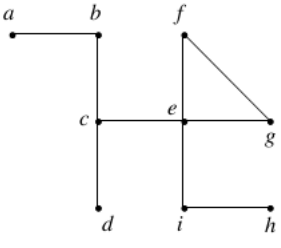
Dans les graphes suivants trouver les points de coupure et les séparateurs :



F



G



H

TD 2

Graphes (2) : Programmation Python

Nous fournissons une archive avec du code "à trous". Pour afficher les graphes, il y a une dépendance au logiciel `graphviz` *et son interface python 3*

2.0.1 Graphes non orientés

EXERCICE #1 ▸ **Décortiquons la classe fournie**
Regarder l'implémentation de la classe `Graph` dans le fichier `LibGraphes.py`. Cette classe fournit un constructeur, et des méthodes de parcours (non implémentées), une méthode heuristique de coloriage, et une méthode d'affichage utilisant Graphviz.

1. Dans `Main.py`, modifier le graphe fourni et vérifier que l'affichage correspond. *Il est normal que les fonctions de parcours ne fonctionnent pas, elles ne sont pas implémentées*
2. Discuter de l'intérêt d'utiliser une classe pour les Graphes.

EXERCICE #2 ▸ **Parcours**
Implémenter les deux parcours : en largeur (BFS) et en profondeur (DFS) *à partir d'un noeud distingué* en remplissant les trous "`TODO`" dans le code fourni de la bibliothèque. [1].

EXERCICE #3 ▸ **Composantes connexes**
Implémenter la recherche des composantes connexes.

EXERCICE #4 ▸ **Passage d'une représentation à une autre**
Les fonction utilitaires de changement de représentation sont explicitement au programme de Terminale, les questions de conception de cet exercice un peu moins . . .

Discuter des choix d'implémentation (critiquer!), et proposer des modifications de la librairie pour disposer des deux implémentations :

1. Quel type interne pour les matrices d'adjacence?
2. Comment garder les deux représentations à jour? Est-ce utile? qu'est-ce que cela impose sur le code des fonctions déjà implémentées?
3. Implémenter les méthodes de transformation d'une représentation à une autre.

2.1 Graphes Orientés

EXERCICE #5 ▸ **Une nouvelle classe?**
Discuter des choix d'implémentation pour réaliser "à partir" de la classe fournie une librairie de graphes orientés. Implémenter une des solutions discutées.

EXERCICE #6 ▸ **Composantes fortement connexes**
Dans des graphes orientés on appelle composante fortement connexe un sous-graphe maximal pour la propriété "pour tout couple (u, v) de noeuds dans ce sous-graphe, il existe un chemin (orienté) de u à v."

à ce stade du TP vous pouvez passer à l'implémentation à rendre"

EXERCICE #7 ▸ **Facultatif**
Implémenter l'algorithme de clôture transitive avec les produits de matrice. Tester.

EXERCICE #8 ▸ **Digicode (Graphe Eulérien) - facultatif**
Doc : `https://fr.wikipedia.org/wiki/Graphe_eul%C3%A9rien`

Implémenter le problème de recherche de séquence minimale pour ouvrir un digicode à 4 chiffres :

1. ce code à trou est automatiquement généré à partir du code prof solution avec le magnifique script `generate-skeletons` que vous pouvez trouver là :`https://gitlab.com/moy/generate-skeletons`

— Montrer que le problème se ramène à un problème de recherche de circuit eulérien.
— Implémenter les fonctions suivantes :
  1. `condEuler(self)` qui teste les conditions d'Euler.
  2. `findEulerianCircuit()` qui retourne (si il existe) un circuit eulérien.
— Répondre à la question. On écrira aussi une fonction qui teste qu'une séquence de chiffres donnée contient bien tous les codes.

2.1.1 Graphes Pondérés

Les graphes pondérés ne sont pas explicitement au programme de Terminale, mais certains algorithmes de flot sont des bons exemples de programmation dynamique.

EXERCICE #9 ▸ **Plus court chemin avec Dijsktra - facultatif**
On vous fournit une classe graphe pondéré (et `Main2.py`, implémentez l'algo de Dijkstra (cf slides).

— Implementez `mini(Q,d)` qui retourne un élément de Q qui minimise d. (d dictionnaire, Q ensemble)
— Les initialisations de l'algo sont déjà faites, il vous reste à implémenter la boucle.
— Améliorez l'affichage.
— Vérifiez l'algo sur plusieurs exemples "faits à la main".

EXERCICE #10 ▸ **Floyd-Warshall - facultatif**
Implémenter les plus courts-chemins avec Floyd Warshall. Implémenter la variante pour calculer la clôture transitive d'un graphe orienté (cf slides).

2.1.2 Évaluation partie Graphes

En utilisant la classe Graphe fournie augmentée par vos implémentations des parcours, et l'exercice d'annales 3.1 vous implémenterez un tri topologique de DAG orienté. Vous fournirez avec cette implémentation :

— Un fichier `README.txt` décrivant un peu vos choix, ainsi qu'une "histoire" adaptée au tri-topologique différente de l'habillage de superman (utilisable en classe de Terminale).
— Un `Main.py` comportant quelques exemples conformes à votre scénario pédagogique

Vous rendrez une archive **en format zip ou tgz** (pas autre chose) sur TOMUSS.

TD 3

Graphes (3) : annales

3.1 Annale 1 : habiller Superman (tri topologique)

Sujet adapté par L. Gonnord du poly d'algo de L3 de l'ENS de Lyon, pour la préparation à l'option informatique du CAPES de Mathématiques, 2018-19

On travaille sur des graphes *orientés*, que l'on suppose représentés par des *listes d'adjacence* : soit S l'ensemble des sommets et A l'ensemble des arcs, on associe à chaque sommet u dans S la liste $Adj[u]$ des éléments v tels qu'il existe un arc de u à v.

Parcours en profondeur :

Le parcours en profondeur d'un graphe G fait usage des constructions suivantes :

— A chaque sommet du graphe est associée une *couleur* : au début de l'exécution de la procédure, tous les sommets sont blancs. Lorsqu'un sommet est rencontré pour la première fois, il devient gris. On noircit enfin un sommet lorsque l'on est en fin de traitement, et que sa liste d'adjacence a été complètement examinée.

— Chaque sommet est également *daté* par l'algorithme, et ceci deux fois : pour un sommet u, $d[u]$ représente le moment où le sommet a été rencontré pour la première fois, et $f[u]$ indique le moment où l'on a fini d'explorer la liste d'adjacence de u. On se servira par conséquent d'une variable entière **globale** "temps" comme compteur événementiel pour la datation.

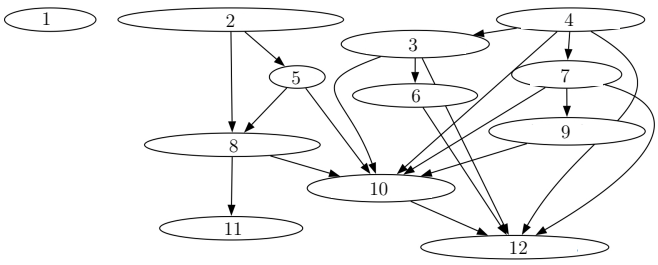
— La manière dont le graphe est parcouru déterminera aussi une relation de paternité entre les sommets, et l'on notera $\pi[v] = u$ pour dire que u est le père de v (selon l'exploration qui est faite, c'est à dire pendant le parcours v a été découvert directement à partir de u).

On définit alors le parcours en profondeur (**PP**) à l'aide des Algorithmes 1 et 2.

début
  pour tous les *sommets* $u \in S$ `/* Initialisation */`
  faire
    $couleur[u] \leftarrow BLANC$;
    $\pi[u] \leftarrow NIL$;
  $temps \leftarrow 0$;
  pour tous les *sommets* $u \in S$ `/* Exploration */`
  faire
    si $couleur[u] = BLANC$ **alors**
      Visiter_PP(u);
    fin
fin

Algorithme 1 : PP(S, $Adj[]$, $temps$)

1. Dessiner la liste d'adjacence du graphe suivant :



```
début
    couleur[u] ← GRIS;
    d[u] ← temps;
    temps ← temps + 1;
    pour tous les v ∈ Adj[u] faire
        si couleur[v] = BLANC alors
            π[v] ← u;
            Visiter_PP(v);
        fin
    couleur[u] ← NOIR;
    f[u] ← temps;
    temps ← temps + 1;
fin
```

Algorithme 2 : Visiter_PP(u)

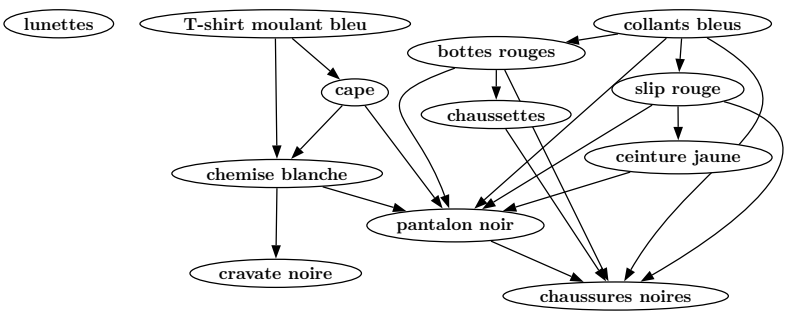
2. Faire tourner l'algorithme sur ce graphe. Lorsqu'il y a un choix entre deux sommets, on prendra le sommet de plus petit numéro.
   On donnera la liste des sommets dans l'ordre de parcours, ainsi que $\pi[u]$, $d[u]$ et $f[u]$ pour tout sommet u.
3. Justifiez le fait que l'on n'appelle **Visiter_PP**(u) qu'une et une seule fois par sommet u du graphe, puis une et une seule fois par arc.
4. En déduire la complexité de PP en fonction de |S| et |A|.
5. Soient u et v deux sommets du graphe. Supposons $d[u] < d[v]$ (on rappelle que d est la "date" de première visite). Si v est un descendant de u, que peut-on dire des valeurs relatives de $d[u]$, $d[v]$, $f[u]$, $f[v]$? Et si v n'est pas un descendant de u ? *FAIRE UN DESSIN*
6. Montrer que si v est un descendant de u, alors à l'instant $d[u]$, il existe un *chemin blanc* de u à v, c'est à dire un chemin qui ne contient que des noeuds coloriés par Blanc. *On s'appuiera sur l'exemple pour raisonner.*
7. Montrer que si à l'instant $d[u]$, il existe un *chemin blanc* de u à v, alors forcément v est un descendant de u. *On raisonnera par l'absurde.*

Tri topologique. n tâches sont données avec des contraintes de précédence $A < B$ signifie que la tâche A doit être effectuée avant la tâche B. L'objectif est de trouver un ordre des tâches qui respecte les contraintes de précédence. Pour cela, on modélise le problème par un graphe orienté :

— les sommets sont les tâches et

— il y a un arc $A \to B$ si et seulement si $A < B$, i.e. si A doit être exécuté avant A.

Voici par exemple le graphe de contraintes pour permettre à Clark Kent de s'habiller le matin avec son habit de Superman sous son costume (par exemple, le slip est mis après les collants et avant le pantalon noir) :



Il s'agit alors de trouver un ordre des sommets qui respecte les dépendances, ie qui respecte l'ordre "père-fils" dans le graphe. On remarque que si le graphe admet un circuit, ce n'est pas possible.

8. Rajouter dans le graphe exemple des premières questions l'arc $12-8$ (afin de créer un circuit). Que se passe-t-il lors du parcours en profondeur ?
9. Montrer que si le parcours en profondeur produit un arc arrière (c'est-à-dire un arc uv tel que v est un ancêtre de u dans l'arborescence produite π), alors il y a un circuit dans le graphe.
10. Montrer (par l'absurde) que si un graphe est sans circuit, alors le parcours en profondeur ne produit aucun arc arrière.

    On donne l'algorithme suivant pour faire un tri topologique d'un graphe G :

    TRI-TOPOLOGIQUE(G)
    1 appeler PP(G) pour calculer
      les dates de fin de traitement $f[v]$ pour chaque sommet v
    2 chaque fois que le traitement d'un sommet se termine,
      insérer le sommet début d'une liste chaînée
    3 **retourner** la liste chaînée des sommets

11. Appliquer cet algorithme pour habiller Superman grâce à l'exemple de la question 1.
12. Quelle est la complexité de cet algorithme ?
13. (Difficile) Expliquer pourquoi cet algorithme donne le résultat voulu.

3.2 Annale 2 : coloriage de graphes

Sujet adapté par L. Gonnord du sujet de concours X-ENS 2018, pour la préparation à l'option informatique du CAPES de Mathématiques, 2018-19

Préliminaires

Complexité Par **complexité en temps** d'un algorithme A, on entend le nombre d'opérations élémentaires (comparaison, addition, soustraction, multiplication, division, affectation, test, etc) nécessaires à l'exécution de A dans le cas le pire.

Lorsque la complexité en temps ou en espace dépend d'un ou plusieurs paramètres $k_0,\dots,k_{r-1}$, on dit que A a une complexité $O(f(k_0,\dots,k_{r-1}))$ s'il existe une constante $C>0$ telle que, pour toutes les valeurs de $k_0,\dots,k_{r-1}$ suffisamment grandes (c'est à dire plus grandes qu'un certain seuil), pour toute instance du problème de paramètres $k_0,\dots,k_{r-1}$, la complexité est au plus $Cf(k_0,\dots,k_{r-1})$.

On dit que la complexité en temps est **linéaire** quand f est une fonction linéaire des paramètres $k_0,\dots,k_{r-1}$, **polynomiale** quand f est une fonction polynomiale des paramètres $k_0,\dots,k_{r-1}$ et enfin **exponentielle** quand $f=2^g$ où g est une fonction polynomiale des paramètres $k_0,\dots,k_{r-1}$.

Les complexités (en temps) des algorithmes **devront être justifiées**.

Graphes Rappelons qu'un graphe non-orienté est la donnée (S,A) de deux ensembles finis :

— un ensemble S de **sommets**, et

— un ensemble $A\subset S\times S$ d'**arêtes**, tel que pour tout couple de sommets (s,t), $s\neq t$ et, on a $(s,t)\in A$ si et seulement si $(t,s)\in A$.

Etant donné un graphe $G=(S,A)$, le **sous-graphe induit** par un ensemble de sommets $T\subset S$ est $(T, A\cap(T\times T))$.

Soit $G=(S,A)$ un graphe et soit $s\in S$ un sommet de G. Un **voisin** de s est un sommet t de G qui est relié à s par une arête, c'est à dire tel que $(s,t)\in A$. On note $V(s)$ l'ensemble des voisins de s. Le **degré** $d(s)$ de s est le cardinal de $V(s)$. Le **degré** $d(G)$ de G est le maximum des degrés de ses sommets.

Un graphe est dit **étiqueté** lorsque l'on dispose d'une fonction, dite d'étiquetage, de l'ensemble de ses sommets vers un ensemble non vide arbitraire, que l'on appelle ensemble des étiquettes. Les étiquettes peuvent par exemple êter des entiers, des listes ou des chaînes de caractères.

On dit qu'une fonction d'étiquetage L est un **coloriage** des sommets de $G=(S,A)$ lorsque deux sommets voisins ont toujours deux étiquettes distinctes (alors appelées **couleurs**), c'est à dire lorsque L vérifie la condition

$$\forall s,t\in S,\ (s,t)\in A\Rightarrow L(s)\neq L(t)$$

Un graphe est dit k-coloriable s'il admet un coloriage avec au plus k couleurs. Un graphe est dit colorié s'il est k-coloriable pour un $k > 0$.

Le **nombre chromatique** d'un graphe non orienté G, noté $\chi(G)$, est le nombre minimal k tel que G est k-coloriable. Cet énoncé porte sur le calcul de nombres chromatiques et de coloriages.

Représentation des graphes étiquetés On se fixe dans cet énoncé une représentation des graphes par matrices d'adjacence (avec 0/1). On se fixe également comme convention que les étiquetages des graphes sont tous à valeurs entières. L'étiquetage d'un graphe sera donné par une liste d'entiers. Un graphe non orienté $G = (S, A)$ avec $S = \{0, \ldots, n-1\}$ est représenté par une valeur `gphe` de type `graphe` telle que pour $i, j \in S$, `gphe[i,j]`=1 si et seulement si $(i, j) \in A$. Le graphe G étant supposé non orienté, on a alors également par symétrie `gphe[j,i]`=1. Pour un étiquetage `etiq` de `gphe`, l'étiquette du sommet i de `gphe` est donnée par `etig[i]`.

En Python/igraph, on crée une matrice d'adjacence comme ceci :

```
# ceci est un graphe avec 6 sommets
adj = np.array([[0, 1, 1, 0, 0, 0],
                [1, 0, 0, 1, 0, 0],
                [1, 0, 0, 0, 1, 0],
                [0, 1, 0, 0, 1, 0],
                [0, 0, 1, 1, 0, 1],
                [0, 0, 0, 0, 1, 0]])
```

et un étiquetage comme ceci :

```
etiq = [1, 1, 2, 2, 1, 0]
```

3.2.1 Coloriage

1. Indiquer, pour chacun des graphes de la figure 3.1, si l'étiquetage proposé est un coloriage.

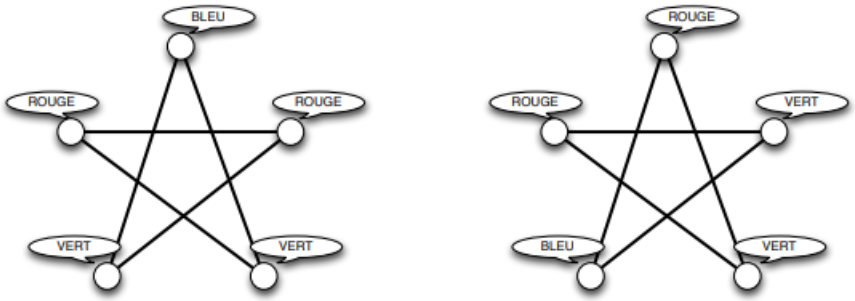


FIGURE 3.1 – Graphes (bien) coloriés ?

2. Donner le nombre chromatique, ainsi qu'un exemple de coloriage pour le **graphe de Petersen** de la Figure 3.2.

   La vérification de la propriété de coloriage est le problème suivant.

   — Entrée : un graphe G et un étiquetage L de G.

   — Question : L est-il un coloriage de G ?

3. Ecrire une fonction `est_col`, telle que `est_col(gphe,etiq, taille)` renvoie `True` si et seulement si `etiq` est un coloriage de `gphe` (taille est le nombre de sommets).

   Dans le cas où la taille de l'étiquetage est strictement inférieure au nombre de sommets du graphe, la fonction renvoie `False`. On demande une complexité quadratique en le nombre de sommets du graphe.

4. Démontrer que le calcul du nombre chromatique d'un graphe peut s'effectuer en temps exponentiel en le nombre de sommets. *indication : on se demandera combien il existe de coloriages à k couleurs, pour k inférieur au nombre de sommets du graphes.*

3.2.2 2-coloriage

Nous avons vu à la question 4 que le calcul du nombre chromatique peut s'effectuer en temps exponentiel en le nombre de sommets du graphe. Dans le cas général, on ne sait aujourd'hui pas faire mieux. Pour obtenir de meilleures bornes de complexité, il faut donc se limiter à des sous-probèmes. On considère dans cette partie le cas du 2-coloriage.

Graphe biparti. Un graphe G est **biparti** lorsque l'ensemble de ses sommets S peut être divisé en deux sous-ensembles disjoints T et U (non vides), tels que chaque arête a une extrémité dans T et l'autre dans U.

5. Démontrer (proprement) qu'un graphe G est biparti si et seulement s'il est 2-coloriable.

   On se propose de programmer la vérification de la 2-colorabilité des graphes en procédant comme suit. On effectue un parcours du graphe en profondeur au cours duquel on construit une 2-coloration du graphe. On se donne pour ce faire trois étiquettes, disons -1, 0 et 1. L'étiquetage est initialisé à -1 pour tous les sommets, et on teste la 2-colorabilité avec 0 et 1. Le principe de l'algorithme est le suivant.

   (1) On choisit un sommet s d'étiquette -1.

   (2) On colorie les sommets rencontrés lors du parcours en profondeur à partir de s, en alternant entre les couleurs 0 et 1 à chaque incrémentation de la profondeur, et en vérifiant si les sommets déjà coloriés rencontrés sont d'une couleur compatible.

   (3) Enfin, s'il reste des sommets d'étiquette -1, alors on revient au point (1).

6. Écrire une fonction récursive `explo(gr, i, k, taille, etiq)` qui réalise le parcours en profondeur du graphe `gr` à partir du sommet `i`, en fixant la couleur de ce sommet à `k` dans `etiq`. Avec quelle couleur doivent être coloriés les voisins du sommet k?

   Comme les paramètres des fonctions python sont mutables, toute modification apportée à `etiq` *sera (automatiquement) enregistrée à la sortie de la fonction.* -1 *dans le tableau* `etiq` *dénote le fait qu'un sommet n'a pas encore été vu.*

7. En utilisant la fonction précédente, écrire une fonction `deuxcol(gphe, taille)` qui calcule et retourne un 2 coloriage (0/1) du graphe si celui-ci est 2-coloriable. Le sommet 0 sera colorié en 0. Faire un exemple.

   On demande une complexité quadratique en le nombre de sommets du graphe (justifier!). Le comportement de la fonction est laissé au choix du candidat lorsque le graphe n'est pas 2-coloriable.

   Indication : l'initialisation des étiquettes peut être réalisée avec : `etiq = [-1] * taille`

3.2.3 Glouton

Dans cette partie, nous allons étudier un algorithme permettant de colorier un graphe en temps polynomial, mais donnant en général un coloriage sous-optimal : le coloriage obtenu peut dans certains cas utiliser plus de couleurs que le coloriage optimal.

Cet algorithme prend en paramètre un ordre sur les sommets du graphe, que l'on appelera **ordre de numérotation**.
Par exemple, $1 < 3 < 4 < 0 < 2 < 6 < 5 < 9 < 8 < 7$ et $0 < 7 < 2 < 5 < 4 < 6 < 8 < 1 < 3 < 9$ sont deux ordres de numérotation des sommets du graphe de Petersen (Figure 3.2).

Pour un graphe `gphe` à n sommets, on implémente un ordre de numérotation de ses sommets par un tableau `num` de n valeurs entières, tel que `num[k]`$=j$ si et seulement si le sommet j apparaît en $(k+1)$-ième position dans l'ordre.

L'**algorithme glouton** construit un coloriage L d'un graphe G en utilisant au plus $d(G)+1$ couleurs. Son principe est le suivant :
On parcourt la liste des sommets du graphe, dans l'ordre de numérotation des sommets donné. Pour chaque sommet s parcouru :

(1) On calcule l'ensemble $C(s) = \{L(t) / \ t \in V(s)\}$ des couleurs déjà données aux voisins de s.

(2) On cherche le plus petit entier naturel c qui n'appartient pas à $C(s)$.

(3) On pose $L(s) = c$.

8. Considérons le graphe de Petersen (Figure 3.2) et les deux ordres de numérotation :

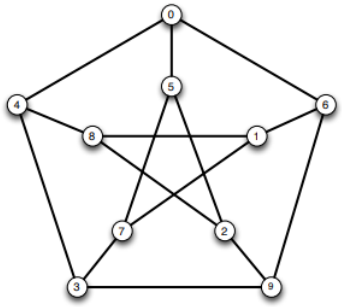


FIGURE 3.2 – Le graphe de Peterson à 10 sommets

```
num1 = [1, 3, 4, 0, 2, 6, 5, 9, 8, 7]
num2 = [0, 7, 2, 5, 4, 6, 8, 1, 3, 9]
```

Donner les coloriages obtenus par l'algorithme glouton décrit ci-dessus pour le graphe de Petersen et chacun de ces deux ordres de numérotation, ainsi que les nombres de couleurs correspondants.

9. Écrire une fonction `min_couleur_possible(gr, etiquetage, taille, sommet)` qui pour un graphe `gphe` à $taille$ sommets, un étiquetage `etiquetage` à valeurs dans $\{-1,\dots,n-1\}$, renvoie le plus petit entier naturel n'appartenant pas à l'ensemble $\{\texttt{etiq}[t] / \ t \in V(\texttt{s})\}$. On demande une complexité $O(n)$.
10. Écrire une fonction `colore_glouton` : pour un graphe `gphe` de taille $taille$ et un ordre de numérotation `num` de ses sommets, l'appel `colore_glouton(gphe, num, taille)` renvoie le coloriage glouton de `gphe` selon l'ordre, avec au plus $d+1$ sommets, où d est le degré de `gphe`. On demande une complexité $O(n^2)$, où n est le nombre de sommets de `gphe`.

    Dans le cas où le tableau `num` contient autre chose qu'un ordre de numérotation des sommets de `gphe`, le résultat de la fonction est laissé au choix, mais il faudra préciser.
11. Montrer que l'algorithme de coloriage glouton construit toujours un coloriage, et que ce coloriage utilise au plus $d+1$ couleurs, où d est le degré du graphe en entrée.
12. Soit G un graphe. Montrer que pour tout coloriage L de G, il existe un ordre de numérotation des sommets tel que le coloriage glouton L' associé vérifie $L'(s) \leq L(s)$ pour tout sommet s de G. En déduire qu'il existe une numérotation des sommets telle que l'algorithme glouton renvoie un coloriage optimal.

    Les questions 7 et 11 indiquent que l'efficacité de l'algorithme glouton est en grande partie dépendante de l'ordre dans lequel on choisit de parcourir les sommets du graphe. L'ordre correspondant à la représentation choisie du graphe (dans notre cas, les indices de la matrice d'adjacence, c'est à dire la permutation identité) est le plus simple à calculer, mais a peu de chances d'être efficace. A contrario, on pourrait essayer de déterminer l'ordre optimal, dont on a prouvé l'existence à la question 11, mais cela n'apporte aucun bénéfice vis-à-vis de la complexité temporelle du problème.

    Une alternative est donnée par l'optimisation de Welsh-Powell. L'idée est de parcourir l'ensemble des sommets du graphe par ordre de degré décroissant. Le tri des sommets par degré décroissant ne prend pas plus de temps que le parcours glouton, mais permet d'obtenir un algorithme raisonnablement efficace en pratique.
13. Écrire une fonction de tri **décroissant** d'un tableau / d'une liste d'entiers en Python, et donner sa complexité.
14. Écrire une fonction `degres(gr, taille)` qui retourne la liste des degrés des sommets du graphe.
15. En déduire une fonction `welsh_powell` qui implémente l'optimisation de Welsh-Powell. *Une modification des deux algorithmes précédents pourra être utile.* Pourquoi un tri quadratique est-il suffisant ?

3.3 Annale 3 : Chemins dans un plan

Sujet adapté par L. Gonnord du sujet de concours X-ENS PSI/PT 2015, pour la préparation à l'option informatique du CAPES de Mathématiques, 2018-19

Préliminaires

Objectif Le but de cette épreuve est de décider s'il existe, entre deux villes données, un chemin passant par exactement k villes intermédiaires distinctes, dans un plan contenant au total n villes reliées par m routes. L'algorithme d'exploration naturel s'exécute en temps $O(nkm)$. L'objectif est d'obtenir un algorithme qui s'exécute en un temps $O(f(k) * n(n+m))$, qui croit polynomialement en la taille $(n+m)$ du problème quelle que soit la valeur de k demandée.

Complexité. La complexité, ou le temps d'exécution, d'un programme Π (fonction ou procédure) est le nombre d'opérations élémentaires (addition, multiplication, affectation, test, etc...) nécessaires à l'exécution de Π . Lorsque cette complexité dépend de plusieurs paramètres n, m et k, on dira que Π a une complexité en $O(\varphi(n,m,k))$, lorsqu'il existe quatre constantes absolues A, n_0 , m_0 et k_0 telles que la complexité de Π soit inférieure ou égale à $A * \varphi(n,m,k)$, pour tout $n > n_0$, $m > m_0$ et $k > k_0$. Lorsqu'il est demandé de préciser la complexité d'un programme, le candidat devra justifier cette dernière si elle ne se déduit pas directement de la lecture du code.

Implémentation On suppose que l'on dispose d'une fonction `creerTableau(n)` qui alloue un tableau de taille n indexé de 0 à $n-1$ (les valeurs contenues dans le tableau initialement sont arbitraires). L'instruction `b = creerTableau(n)` créera un tableau de taille n dans la variable b

On pourra ainsi créer un tableau a de p tableaux de taille q par la suite d'instructions suivante :

```
a = creerTableau(p)
for i in range(p):
    a[i] = creerTableau(q)
```

On accédera par l'instruction $a[i][j]$ à la j-ème case du i-ème tableau contenu dans le tableau a ainsi créé. Par exemple, la suite d'instructions suivante remplit le tableau a case par case :

```
for i in range(p):
    for j in range(q):
       a[i][j] = i+j
```

On supposera l'existence de deux valeurs booléennes `True` et `False`. On supposera l'existence d'une procédure : `affiche(...)` qui affiche le contenu de ses arguments à l'écran. Par exemple :

```
 x = 1; y = x+1; affiche("x = ",x," et y = ",y);
```

affiche à l'écran :

```
x = 1 et y = 2
```

Dans la suite, nous distinguerons fonction et procédure : les fonctions renvoient une valeur (ou un tableau) tandis que les procédures ne renvoient aucune valeur.

3.3.1 Préliminaires : stockage sans redondance

On souhaite stocker en mémoire de manière non ordonnée un ensemble d'au plus n entiers sans redondance (aucun entier n'apparaîtra plusieurs fois). Nous proposons d'utiliser un tableau `tab` de longueur $n+1$ tel que :

— `tab[0]` contient le nombre d'éléments de l'ensemble.

— `tab[i]` contient le i-ème élément de l'ensemble (non-ordonné).

1. Écrire une fonction `creerTabVide(n)` qui crée, initialise et renvoie un tableau de longueur $n+1$ correspondant à la table vide pouvant stocker un ensemble à n éléments.

2. Écrire une fonction `estDansTab(tab, x)` qui renvoie `True` si l'élément x apparaît dans l'ensemble représenté par le tableau tab, et renvoie `False` sinon. Quelle est la complexité en temps de votre fonction dans le pire cas en fonction du nombre maximal n d'éléments dans l'ensemble?
3. Écrire une procédure `ajouteDansTab(tab, x)` qui modifie de façon appropriée le tableau tab pour y ajouter x si l'entier x n'appartient pas déjà au tableau, et ne fait rien sinon.
4. Quel est le comportement de votre procédure (ajout) si la table est pleine initialement? (On ne demande pas de traiter ce cas) Quelle est la complexité en temps de votre procédure dans le pire cas en fonction du nombre maximal n d'éléments dans l'ensemble?

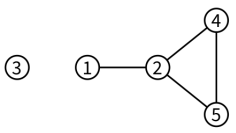
3.3.2 Création et manipulation de plans

Un plan P est défini par : un ensemble de n villes numérotées de 1 à n et un ensemble de m routes (toutes à double-sens) reliant chacune deux villes ensemble. On dira que deux villes $x, y \in [\![n]\!]$ sont voisines lorsqu'elles sont reliées par une route [1] , ce que l'on notera par $x \sim y$. On appellera chemin de longueur k toute suite de villes $v_1, \ldots, v_k$ telle que $v_1 \sim v_2 \sim \ldots \sim v_k$. On représentera les villes d'un plan par des ronds contenant leur numéro et les routes par des traits reliant les villes voisines (voir Figure 3.3).

Structure de données . Nous représenterons tout plan P à n villes par un tableau `plan` de $(n+1)$ tableaux où :

— `plan[0]` contient un tableau à deux éléments où :
  — `plan[0][0] = n` contient le nombre de villes du plan
  — `plan[0][1] = m` contient le nombre de routes du plan
— Pour chaque ville $x \in [\![n]\!]$, `plan[x]` contient un tableau à n éléments représentant l'ensemble à au plus $n-1$ éléments des villes voisines de la ville x dans P dans un ordre arbitraire en utilisant la structure sans redondance définie dans la partie précédente. Ainsi :
  — `plan[x][0]` contient le nombre de villes voisines de x
  — `plan[x][1]`, ..., `plan[x][plan[x][0]]` sont les indices des villes voisines de x.

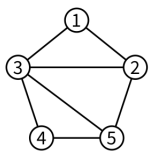
La figure 3.3 donne un exemple de plan et d'une représentation possible sous la forme de tableau de tableaux (les * représentent les valeurs non-utilisées des tableaux).



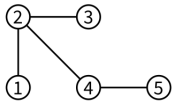

```
plan = [ [ 5  4],
         [ 1  2    *   *   * ],
         [ 3  4    1   5   * ],
         [ 0  *    *   *   * ],
         [ 2  2    5   *   * ],
         [ 2  4    2   *   * ] ]
```

FIGURE 3.3 – Un plan à 5 villes et 4 routes et une représentation possible en mémoire sous forme d'un tableau de tableaux `plan`

5. Représenter sous forme de tableaux de tableaux les deux plans suivants :



Plan 1



Plan 2

On pourra utiliser dans la suite, les fonctions et procédures définies dans la partie précédente.

6. Écrire une fonction `creerPlanSansRoute(n)` qui crée, remplit et renvoie le tableau de tableaux correspondant au plan à n villes n'ayant aucune route.

1. En langage graphes, on appelerait cela arête

7. Écrire une fonction `estVoisine(plan, x, y)` qui renvoie `True` si les villes x et y sont voisines dans le plan codé par le tableau de tableaux `plan`, et renvoie `False` sinon.
8. Écrire une procédure `ajouteRoute(plan, x, y)` qui modifie le tableau de tableaux `plan` pour ajouter une route entre les villes x et y si elle n'était pas déjà présente et ne fait rien sinon. (On prendra garde à bien mettre à jour toutes les cases concernées dans le tableau de tableaux `plan`) Y a-t-il un risque de dépassement de la capacité des sous-tableaux?
9. Écrire une procédure `afficheToutesLesRoutes(plan)` qui affiche à l'écran l'ensemble des routes du plan codé par le tableau de tableaux plan où chaque route apparaît exactement une seule fois. Par exemple, pour le graphe codé par le tableau de tableaux de la figure 3.3, votre procédure pourra afficher à l'écran :

   ```
   Ce plan contient 4 route(s): (1-2) (2-4) (2-5) (4-5)
   ```

   Quelle est la complexité en temps de votre procédure dans le pire cas en fonction de n et m?

3.3.3 Recherche de chemins arc-en-ciel

Étant données deux villes distinctes s et $t \in [\![n]\!]$, nous recherchons un chemin de s à t passant par exactement k villes intermédiaires toutes distinctes. L'objectif de cette partie et de la suivante est de construire une fonction qui va détecter en temps linéaire en $n(n+m)$, l'existence d'un tel chemin avec une probabilité indépendante de la taille du plan $n+m$.

Le principe de l'algorithme est d'attribuer à chaque ville $i \in [\![n]\!] \setminus \{s,t\}$ une couleur aléatoire codée par un entier aléatoire uniforme $couleur[i] \in 1,\dots,k$ stocké dans un tableau $couleur$ de taille $n+1$ (la case 0 n'est pas utilisée). Les villes s et t reçoivent respectivement les couleurs spéciales 0 et $k+1$, i.e. $couleur[s] = 0$ et $couleur[t] = k+1$. L'objectif de cette partie est d'écrire une procédure qui détermine s'il existe un chemin de longueur $k+2$ allant de s à t dont la j-ème ville intermédiaire a reçu la couleur j. Dans l'exemple de la figure 3.4, le chemin $6 \sim 7 \sim 8 \sim 3 \sim 4$ de longueur $5 = k+2$ qui relie $s = 6$ à $t = 4$ vérifie cette propriété pour $k = 3$.

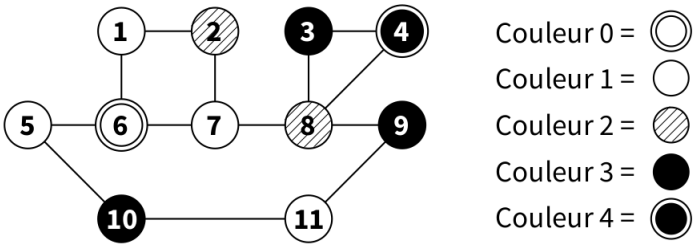


FIGURE 3.4 – Exemple de plan colorié pour $k = 3$, $s = 6$, $t = 4$

On suppose l'existence d'une fonction `entierAleatoire(k)` qui renvoie un entier aléatoire uniforme entre 1 et k (i.e. telle que $Pr\{entierAleatoire(k) = c\} = 1/k$ pour tout entier $c \in \{1,\dots,k\}$).

10. Écrire une procédure `coloriageAleatoire(plan, couleur, k, s, t)` qui prend en argument un plan de n villes, un tableau couleur de taille $n+1$, un entier k, et deux villes s et $t \in [\![n]\!]$, et remplit le tableau couleur avec : une couleur aléatoire uniforme dans $\{1,\dots,k\}$ choisie indépendamment pour chaque ville $i \in [\![n]\!] \setminus \{s,t\}$; et les couleurs 0 et $k+1$ pour s et t respectivement.

    Nous cherchons maintenant à écrire une fonction qui calcule l'ensemble des villes de couleur c voisines d'un ensemble de villes donné. Dans l'exemple de la figure 2, l'ensemble des villes de couleur 2 voisines des villes $\{1,5,7\}$ est $\{2,8\}$.
11. Écrire une fonction `voisinesDeCouleur(plan, couleur, i, c)` qui crée et renvoie un tableau codant l'ensemble sans redondance des villes de couleur c voisines de la ville i dans le plan plan colorié par le tableau couleur.

12. Écrire une fonction `voisinesDeLensCouleur(plan, couleur, tab, c)` qui crée et renvoie un tableau codant l'ensemble sans redondance des villes de couleur c voisines d'une des villes présente dans l'ensemble sans redondance tab dans le plan $plan$ colorié par le tableau $couleur$. Quelle est la complexité de votre fonction dans le pire cas en fonction de n et m?
13. Écrire une fonction `existeCheminArcEnCiel(plan, couleur, k, s, t)` qui renvoie `True` s'il existe dans le plan plan, un chemin $s \sim v_1 \sim \ldots \sim v_k \sim t$ tel que $couleur[v_j] = j$ pour tout $j \in \{1,\ldots,k\}$; et renvoie `False` sinon.

    Quelle est la complexité de votre fonction dans le pire cas en fonction de k, n et m?

3.3.4 Recherche de chemin passant par exactement k villes intermédiaires distinctes

Si les couleurs des villes sont choisies aléatoirement et uniformément dans $\{1,\ldots,k\}$, la probabilité que j soit la couleur de la j-ème ville d'une suite fixée de k villes, vaut $1/k$ indépendamment pour tout j. Ainsi, étant données deux villes distinctes s et $t \in [[n]]$, s'il existe dans le plan plan un chemin de s à t passant par exactement k villes intermédiaires toutes distinctes et si le coloriage couleur est choisi aléatoirement conformément à la procédure `coloriageAleatoire(plan, couleur, k, s, t)`, la procédure `existeCheminArcEnCiel(plan, couleur, k, s, t)` répond `True` avec probabilité au moins k^{-k}; et répond toujours `False` sinon. Ainsi, si un tel chemin existe, la probabilité qu'une parmi k^k exécutions indépendantes de `existeCheminArcEnCiel` réponde `True` est supérieure ou égale à $1-(1-k^{-k})^{k^k} = 1-exp(k^k ln(1-k^{-k})) \geq 1-\frac{1}{e} > 0$ (admis).

14. Écrire une fonction `existeCheminSimple(plan, k, s, t)` qui renvoie `True` avec probabilité au moins $1-1/e$ s'il existe un chemin de s à t passant par exactement k villes intermédiaires toutes distinctes dans le plan plan; et renvoie toujours `False` sinon. Quelle est sa complexité en fonction de k, n et m dans le pire cas? Exprimez-la sous la forme $O(f(k) * g(n,m))$ pour f et g bien choisies.
15. Expliquer comment modifier votre programme pour renvoyer un tel chemin s'il est détecté avec succès.

TD 4
Algorithmes randomisés

On conseille ici d'aller lire l'excellente page Wikipédia sur le sujet :

`https://fr.wikipedia.org/wiki/Algorithme_probabiliste`

4.1 Marches Aléatoires

Énoncé, code, corrigé par S. Gonnord remis en forme par L. Gonnord, mai 2020.

Objectifs pédagogiques

— Réaliser et visualiser des marches aléatoires dans **Z**, puis le plan, l'espace et même le cercle trigonométrique.

— Faire des statistiques sur de telles marches pour vérifier/tester des résultats plus ou moins standards.

On fournit du code "à trous".

4.1.1 Marches dans Z

On commence par des marches dans **Z** avec une fonction réalisant un pas. Elle prend en entrée $p \in [0,1]$, puis renvoie 1 avec probabilité p et -1 avec probabilité $1-p$. On utilisera la fonction `random` qui, appelée sans argument, renvoie un flottant entre 0 et 1, avec une loi de répartition uniforme, de sorte que `random() <= p` est vrai avec probabilité p.

EXERCICE #1 ▸ **Un pas, une marche**
Écrire une telle fonction réalisant un pas, puis une marche.

```
>>> pas(0.5)
1
>>> pas(0.5)
-1
>>> pas(0.5)
-1
>>> pas(0.8)
1
>>> pas(0.2)
-1
```

```
>>> marche(10, 0.5)
[0, -1, -2, -1, 0, -1, 0, -1, -2, -3, -2]
>>> marche(10, 0.5)
[0, -1, 0, 1, 2, 1, 0, 1, 2, 3, 2]
>>> marche(10, 0.5)
[0, 1, 0, 1, 0, -1, -2, -3, -2, -1, -2]

>>> marche(10, 0.3)
[0, -1, -2, -3, -2, -1, 0, -1, -2, -3, -4]
```

EXERCICE #2 ▸ **Dessins**
Utiliser la fonction fournie pour dessiner des marches aléatoires sur un axe. On doit obtenir des dessins comme sur les Figures 4.1 et 4.2.

On va maintenant tenter d'observer expérimentalement quelques résultats portant sur une marche $S_n = X_1 + \cdots + X_n$ (les X_i sont des pas indépendants) tels que (dans le cas non biaisé $p = 1/2$) :

— L'espérance de S_n est nulle.

— Sa variance est $Var(S_n) = E(S_n^2) = n$.

— L'espérance de sa valeur absolue est majorée par la racine de l'espérance de son carré : $E(|S_n|) \leq n$.

— Avec probabilité 1, on revient à l'origine.

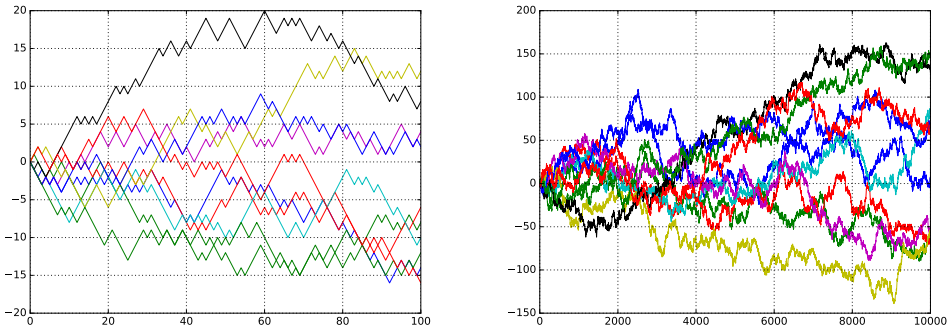


FIGURE 4.1 – Des marches aléatoires non biaisées...

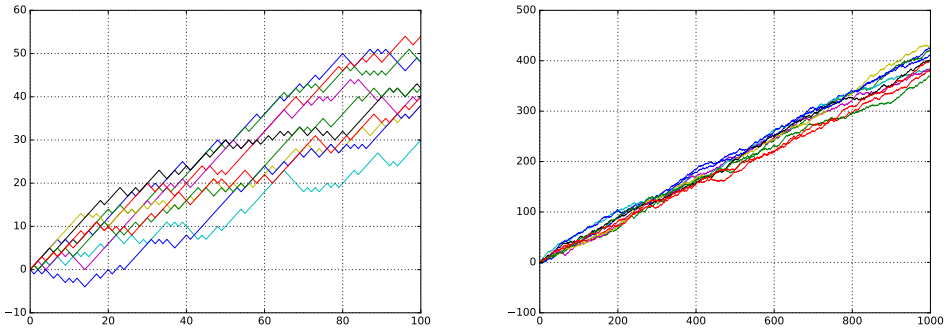


FIGURE 4.2 – Et des marches biaisées (devinez la valeur de p).

EXERCICE #3 ▶ **Moyenne**
Écrire une fonction permettant de calculer la moyenne des positions issues de marches successives : on réalise des marches, et on garde pour chacune la dernière valeur (la position à la fin de la marche), et on fait la somme de toutes ces dernières valeurs avant de diviser par le nombre de marches. On trouvera en testant des résultats du style :

```
>>> valeur_moyenne(100, 0.5, 10)
0.0
>>> valeur_moyenne(100, 0.5, 10)
1.0
>>> valeur_moyenne(100, 0.5, 10)
-5.4
```

```
>>> valeur_moyenne(100, 0.5, 10**4)
0.1628
>>> valeur_moyenne(100, 0.7, 10**3)
39.87
>>> valeur_moyenne(100, 0.2, 10**3)
-59.858
```

EXERCICE #4 ▶ **Valeurs moyennes et espérance**
En s'inspirant de ce qui précède, évaluer les valeurs moyennes de $|S_n|$ et de S_n^2 pour $n = 10$ puis $n = 100$ (à vous de déterminer le nombre de marches qui vous semble pertinent).

On s'intéresse maintenant à la question du passage par un point imposé : on fixe un objectif (un entier relatif, ici), et on réalise un certain nombre de marches pour savoir si on passe par cet objectif. La théorie dit qu'avec probabilité 1 on passe par cet objectif dans $\mathbf{Z}$ et $\mathbf{Z}^2$... mais pas $\mathbf{Z}^3$! Bien entendu si on s'impose un nombre de pas maximum, cette probabilité est < 1 (elle augmente avec le nombre de pas maximum, et diminue avec la distance entre l'objectif et l'origine).

EXERCICE #5 ▶ **Revenir au même point**
Écrire une fonction réalisant une marche d'au plus N pas (N est un paramètre), retournant `True` si la marche est passée par l'objectif (paramètre de la fonction), et `False` sinon. On pourra prendre un troisième paramètre

$p \in [0,1]$ permettant de biaiser la marche (sans quoi, on fait des marches symétriques, ce qui est déjà intéressant). Par exemple on peut trouver :

```
>>> repasse_par(0, 10, 0.5)
True
>>> repasse_par(0, 10, 0.5)
True
>>> repasse_par(0, 10, 0.5)
False
>>> repasse_par(0, 10, 0.5)
True
```

```
>>> repasse_par(0, 10, 0.7)
False
>>> repasse_par(0, 10, 0.7)
True
>>> repasse_par(0, 10, 0.7)
False
>>> repasse_par(0, 10, 0.7)
False
>>> repasse_par(0, 10, 0.7)
False
```

EXERCICE #6 ▸ **Probabilités - statistiques**
Écrire une fonction réalisant des statistiques sur la question du passage par un objectif (on donne comme paramètre supplémentaire le nombre de marches).

```
>>> proba_passage(0, 2, 10**4, 0.5)
0.4978
>>> proba_passage(0, 3, 10**4, 0.5)
0.5021
>>> proba_passage(0, 4, 10**5, 0.5)
0.62563

>>> proba_passage(0, 10, 10**5, 0.5)
0.7553
>>> proba_passage(0, 100, 10**5, 0.5)
0.91979
>>> proba_passage(0, 10**4, 10**4, 0.5)
0.9913

>>> proba_passage(-5, 4, 10**5, 0.5)
0.0
>>> proba_passage(-5, 10, 10**5, 0.5)
0.10912
```

```
>>> proba_passage(-5, 100, 10**5, 0.5)
0.61873
>>> proba_passage(-5, 10**4, 10**4, 0.5)
0.9637

>>> proba_passage(-5, 10, 10**5, 0.8)
0.00071
>>> proba_passage(-5, 10, 10**5, 0.2)
0.75681
>>> proba_passage(-5, 100, 10**5, 0.8)
0.00107
>>> proba_passage(-5, 100, 10**5, 0.2)
1.0
>>> proba_passage(-5, 10**4, 10**3, 0.8)
0.001
>>> proba_passage(-5, 10**4, 10**3, 0.2)
1.0
```

4.1.2 Dans le plan, puis l'espace

On passe dans le plan ! Cette fois, un pas est un couple (ou une liste de deux) entiers : il y a 4 déplacements **équiprobables**.

EXERCICE #7 ▸ **Un pas**
Écrire une fonction réalisant un pas dans le plan. On pourra choisir un pas dans la liste des quatre possibles, grâce à la fonction `randint` qui prend en entrée deux arguments entiers, disons a et b, et renvoie un entier n tel que $a \leq n < b$ (oui, une inégalité stricte à droite), chacun avec probabilité $\frac{1}{b-a}$.

```
>>> randint(0, 4)
3
>>> randint(0, 4)
2
>>> randint(0, 4)
3
>>> randint(0, 4)
0

>>> pas2()
[0, 1]
>>> pas2()
[-1, 0]
```

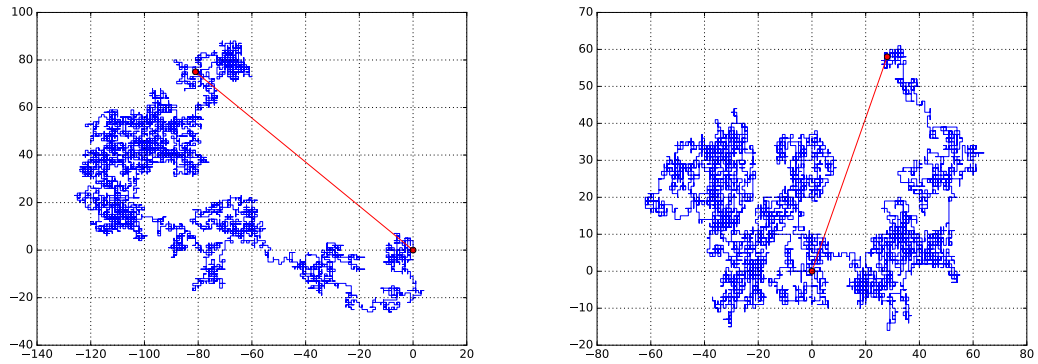


FIGURE 4.3 – Des marches (de 10^4 pas) dans le plan – j'ai relié l'origine à l'arrivée.

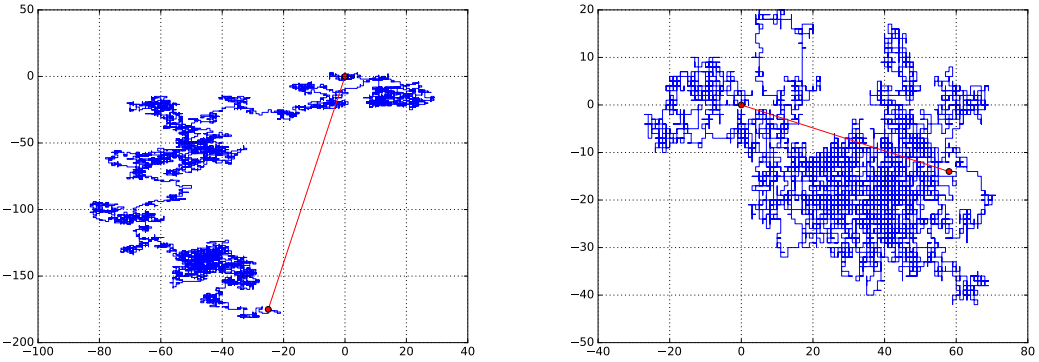


FIGURE 4.4 – Et deux de plus

EXERCICE #8 ▸ **Marche, dessin de marche, etc.**
Remplir les `TODO` du code fourni, pour obtenir des dessins ressemblants à ceux des Figures 4.3 et 4.4.

Vous pouvez voir qu'il y a une grande variété de scénarios possibles ; en particulier sur la question du retour vers l'origine !

EXERCICE #9 ▸ **Distance moyenne**
Écrire des fonctions permettant d'évaluer la distance moyenne (resp. : au carré) après une marche de n pas. Expérimenter.

On s'intéresse enfin à la question du retour à l'origine :

EXERCICE #10 ▸ **Retour au même point**
Écrire des fonctions permettant d'évaluer la probabilité pour qu'une marche (d'une longueur imposée) atteigne un point donné.

Et maintenant, dans l'espace ! Ici il y a un grand changement : la probabilité pour que l'on repasse par l'origine est strictement plus petite que 1.

EXERCICE #11 ▸ **En autonomie**
Reprendre les questions précédentes (distance (au carré) moyenne, probabilité de passage par l'origine), et tester !

4.1.3 Sur le cercle

On termine avec une marche aléatoire trigonométrique, avec un marcheur qui se déplace d'un angle $\pm\alpha$ à chaque étape – on part du point de coordonnées $(1,0)$. La théorie dit que la position moyenne (enfin, son espérance ...) après n pas est $(\cos^n \alpha, 0)$.

EXERCICE #12 ▸ **Et sur le cercle ?**
Reprendre les questions précédentes dans ce cas.

Les maths en jeu Les 4 premiers points sont valables pour "le cas de l'axe".

— par linéarité de l'espérance, $\mathbb{E}(S_n) = \sum_{i=1}^{n} \mathbb{E}(X_i) = n(p-(1-p)) = n(2p-1)$ (ce qui vaut 0 sans surprise dans le cas non biaisé $p = 1/2$).

— les variables aléatoires X_i **étant indépendantes** on a $\texttt{Var}\,(S_n) = \sum_{i=1}^{n} \texttt{Var}\,(X_i)$ et $\texttt{Var}\,(X_i) = \mathbb{E}(X_i^2) - \mathbb{E}(X_i)^2 = 1-(2p-1)^2 = 4p(1-p)$ (X_i^2 est constante égale à 1, d'où son espérance !). Dans le cas $p = 1/2$, on a donc $\texttt{Var}\,(X_n) = n$ (et donc $\mathbb{E}(S_n^2) = n$).

— L'inégalité de Cauchy-Schwarz (par exemple, mais Jensen passe aussi !) donne $\mathbb{E}(|S_n|.1) \leq \sqrt{\mathbb{E}(S_n^2)}\sqrt{\mathbb{E}(1^2)}$, majorant qui vaut $\sqrt{n}$ dans le cas $p = 1/2$.

— Le fait que la marche repasse par l'origine (et en fait par tout point) avec probabilité 1 est à la fois accessible avec des maths de licence … et tout de même non évident (exercice de niveau L2/L3).

— dans le plan, on a cette fois un couple de variables aléatoires $Z_n = (S_n, T_n) = (X_1 + \cdots + X_n, Y_1 + \cdots + Y_n)$. La linéarité de l'espérance nous donne encore $\mathbb{E}(S_n) = 0$ (les X_i et les Y_i sont d'espérance nulle, et de variance égale à $\mathbb{E}(X_i^2) = 2/4 = 1/2$). On obtient de même (par indépendance) $\mathbb{E}(\|S_n\|^2) = \sum_{i=1}^{n} \mathbb{E}(X_i^2 + Y_i^2) = n$, puis $\mathbb{E}(\|S_n\|) \leq \sqrt{n}$. Le passage presque sûr (avec probabilité 1) par l'origine ou tout autre point est plus difficile que dans le plan.

— sur le cercle trigonométrique, tout se comprend bien avec peu de calcul si on note que cette fois $Z_n = e^{i\alpha Z_n} = e^{i\alpha X_1} e^{i\alpha X_2} \ldots e^{i\alpha X_n}$, et l'espérance du produit est le produit des espérances car les $e^{i\alpha X_k}$ sont indépendantes !

4.2 Miller - Rabin

Un des algorithmes randomisés classiques est le test de primalité de Miller-Rabin, qui n'est pas difficile à implémenter. Comme d'habitude, c'est l'analyse qui est plus compliquée. https://fr.wikipedia.org/wiki/Test_de_primalit%C3%A9_de_Miller-Rabin

TD 5
Complexité : langages P, langages NP, NP-complétude

5.1 Quelques problèmes P

Soient les deux problèmes suivants :

— 2-SAT : Soit une forme normale conjonctive F dans laquelle chaque clause a exactement 2 littéraux. F est-elle satisfaisable?

— 2-COLOR : Soit un graphe G = (V, E). Existe-t-il une fonction c : $V \to \{0,1\}$ telle que pour toute arête (u,v) de G, $c(u) \neq c(v)$?

EXERCICE #1 ▸ **2-SAT dans P**
Prouver directement que 2-SAT est dans P. On prendra comme exemple la formule

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

et on essaiera de construire un graphe.

EXERCICE #2 ▸ **2-COLOR dans P**
Prouver directement que 2-COLOR est dans P.

EXERCICE #3 ▸ **Complémentation**
Montrer que la classe P est close par complémentation.

5.2 Appartenance à P via réduction polynomiale

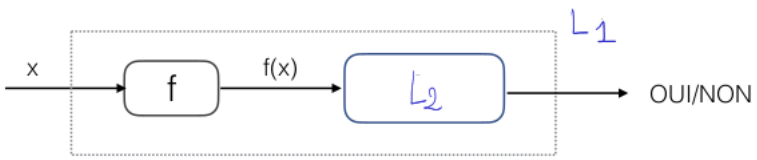
On rappelle la définition d'une réduction polynomiale :

DÉFINITION 1. Soient L_1 et L_2 deux langages de Σ^*. Une réduction polynomiale de L_1 *vers* L_2 est une fonction $f : \Sigma^* \to \Sigma^*$ *calculable en temps polynomial* telle que $x \in L_1$ ssi $f(x) \in L_2$. On note alors $L_1 \leq^P L_2$.

Informellement, cela signifie que L_1 n'est pas plus difficile que L_2.
On a ensuite le résultat suivant

PROPOSITION 1. Si $L_1 \leq^P L_2$ et $L_2 \in P$, alors $L_1 \in P$.

Avec un dessin, c'est vraiment mieux :



ou encore avec du code :

```
let y = f(x) in f_2(y) #algo pour décider L1.
```

EXERCICE #4 ▸ **2-COLOR dans P, par réduction**
Démontrer que le langage 2-COLOR se réduit polynomialement à 2-SAT. En déduire qu'il est dans P.

5.3 NP-Complétude

EXERCICE #5 ▸ **Questions de cours**
Soient P_1 et P_2 deux problèmes de décision, et supposons qu'on connaisse une transformation polynomiale (une réduction) de P_1 en P_2. Répondre aux sept questions suivantes avec un maximum de deux lignes de justification par question.

1. Si $P_1 \in \mathsf{P}$, a-t-on $P_2 \in \mathsf{P}$?
2. Si $P_2 \in \mathsf{P}$, a-t-on $P_1 \in \mathsf{P}$?
3. Si P_1 est NP-complet, P_2 est-il NP-complet ?
4. Si P_2 est NP-complet, P_1 est-il NP-complet ?
5. Si on connaît une transformation polynomiale de P_2 en P_1, P_1 et P_2 sont-ils NP-complets ?
6. Si P_1 et P_2 sont NP-complets, existe-t-il une transformation polynomiale de P_2 en P_1 ?
7. Si $P_1 \in \mathsf{NP}$, P_2 est-il NP-complet ?

5.4 Quelques petites réductions "faciles"

On rappelle la proposition qui permet de prouver qu'un problème donné est NP-complet :

PROPOSITION 2. Si $L_1 \leq^P L_2$, $L_2 \in NP$, et L_1 NP-complet, alors L_2 est NP-complet.

Il s'agit donc de réduire polynomialement le problème connu vers notre problème à montrer NP-complet. Dans la suite, nous allons prouver NP-complets des variantes de "problèmes connus".

EXERCICE #6 ▸ **3-NAE "not all equal"**

— *Instance :* Un ensemble de clauses $C_1, \ldots, C_m$, chacune contenant exactement 3 littéraux.
— *Question :* Existe-t-il une instantiation des variables telle que chaque clause contient un littéral évalué à vrai et un littéral évalué à faux ?

Montrer que 3-NAE est NP-complet. *Indice : On pourra créer m+1 nouvelles variables, et construire une instance avec 2m clauses.* Pour l'exercice suivant, on rappelle la définition du problème 2-PARTITION, qui est NP-complet (on ne demande pas de preuve).

DÉFINITION 2 (2-PARTITION). *Instance* $S = \{a_1, \ldots, a_n\}$ des entiers
Question : Existe-t-il $I \subset S$ tel que $\sum_{i \in I} a_i = \sum_{i \notin I} a_i$?

EXERCICE #7 ▸ **Deux variantes de 2-PARTITION**
Montrer que les deux variantes suivantes de 2-PARTITION sont NP-complètes.

1. **2-Part-Even :**
   Instance : Un ensemble V de $2n$ entiers strictement positifs $a_1, a_2, \ldots, a_{2n}$.
   Question : Existe-t-il un sous-ensemble $I \subset [1..2n]$ tel que $\sum_{i \in I} a_i = \sum_{i \notin I} a_i$?
2. 2-PART-EQ :
   Instance : Un ensemble V de $2n$ entiers strictement positifs $a_1, a_2, \ldots, a_{2n}$.
   Question : Existe-t-il un sous-ensemble $I \subset [1..2n]$ tel que $\sum_{i \in I} a_i = \sum_{i \notin I} a_i$ et $|I| = n$.

5.5 Deux réductions plus compliquées

Dans les deux cas il s'agit de réduction à partir de 3-SAT.

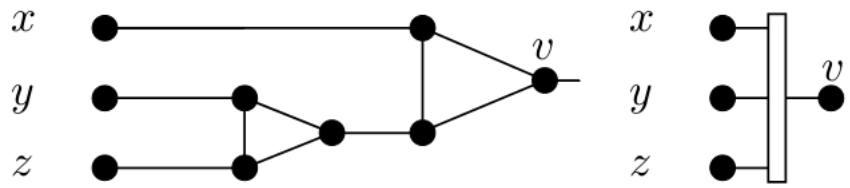
EXERCICE #8 ▸ **SUBSET-SUM**
Montrer que le problème suivant SUBSET-SUM est NP-complet.

SUBSET-SUM
Instance : un ensemble fini S d'entiers positifs et un entier objectif t.
Question : existe-t-il un sous-ensemble $S' \subseteq S$ tel que $\sum_{x \in S'} x = t$?

Indication : vous pouvez par exemple effectuer une réduction à partir de 3-SAT. A partir d'un ensemble de clauses $C_0, ..., C_{m-1}$ sur les variables $x_0, ..., x_{n-1}$, considérer S l'ensemble des entiers $v_i = 10^{m+i} + \sum_{j=0}^{m-1} b_{ij}10^j$ et $v'_i = 10^{m+i} + \sum_{j=0}^{m-1} b'_{ij}10^j$, $0 \leq i \leq n-1$, où b_{ij} (resp. b'_{ij}) vaut 1 si le littéral x_i (resp. $\overline{x_i}$) apparaît dans C_j et 0 sinon, et des entiers $s_j = 10^j$ et $s'_j = 2.10^j$, $0 \leq j \leq m-1$. Trouver alors un entier objectif t tel qu'il existe un sous-ensemble $S' \subseteq S$ de somme t si et seulement si l'ensemble initial de clauses est satisfiable. Conclure. Quels autres entiers auraient aussi marché ?

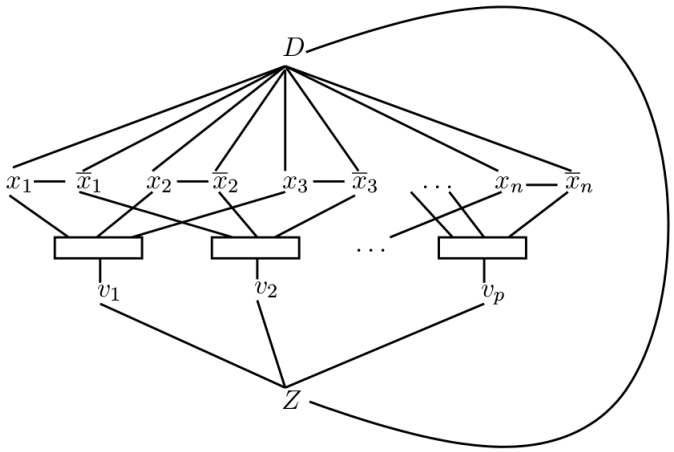
EXERCICE #9 ▸ **NP-complétude de 3-COLOR**

On va faire une réduction à partir de 3-SAT, montré NP-complet dans le cours.
Soit le gadget suivant (à droite son symbole dans la suite) :



1. Montrer que 3-COL est dans la classe NP.
2. Montrer que le gadget satisfait les 2 propriétés suivantes [1] :
   (a) Si $x = y = z = 0$ alors v est *nécessairement* colorié à 0.
   (b) Toute autre entrée (x, y, z) permet de colorier v à 1 ou 2 (au sens où l'on est capable d'exhiber un coloriage *tel que...*)

Soit I une instance de 3-SAT avec p clauses C_k (numérotées de 1 à p) et n variables x_i. On construit le graphe G suivant :



3. Vérifier que ce graphe a une taille polynomiale en l'instance I considérée.
4. Prouver que I a une solution ssi G est 3 coloriable.

EXERCICE #10 ▸ **REGISTER-ALLOC**
Nous allons montrer que le problème suivant est NP-complet. **REGISTER-ALLOC**

Instance : un graphe de flot de contrôle (général) avec des temporaires, $k \geq 4$ registres.
Question : existe-t-il une allocation des variables aux registres sans conflit ?

On supposera dans la suite que les variables ne sont pas splittées et que la construction du graphe de conflit est polynomiale. On va montrer la NP-complétude par réduction à partir de la NP-complétude de $k-1$-COLOR.

1. On pourra utiliser la notation suivante : $x, y \in \{0,1,2\}$ étant distincts, $\varphi(x, y)$ désigne le troisième élément de $\{0,1,2\}$.

1. Montrer que k-REGISTERALLOC est dans NP.

Soit un graphe $G = (V, E)$. Construisons le graphe de flot de contrôle sur les variables $V \cup \{x\}$ suivant :

— Pour chaque arête (u, v), on définit un bloc $B_{u,v}$ qui définit (au sens *def* du cours de compilation) les variables u et v (en les initialisant à une constante) puis `x=u+v`.

— Pour chaque sommet u de V, on construit un bloc B_u qui lit u et x et retourne une nouvelle valeur déduite de u et x (par exemple `return u+x`)

— Chaque bloc $B_{u,v}$ est un prédécesseur direct de B_u et de B_v.

— Les blocs $B_{u,v}$ ont un unique prédécesseur commun vide appelé B_{switch}.

2. Construire le graphe de flot de contrôle produit par la construction précédente pour le graphe cyclique de taille 4 dont les arêtes sont (a, b), (b, c), (c, d) et (d, a).
3. Quel est le graphe d'interférence associé à ce graphe de flot ?
4. Si le graphe initial est k coloriable, que peut-on dire du graphe de conflit du programme après transformation ?
5. Finir la preuve.
6. (Bonus) Que se passe-t-il si on autorise le *split* de variables ? On pourra raisonner sur le graphe précédent en essayant de découper la durée de vie de la variable dans B_a avant que a soit utilisée.

Remarques Source de la preuve : `www.ens-lyon.fr/LIP/Pub/Rapports/RR/RR2006/RR2006-13.pdf`, dans ce rapport il est aussi montré les résultats intéressants suivants :

— Si on peut "splitter" les variables, le problème reste NP-complet.

— Si le graphe n'a pas d'arc critique et que l'on peut splitter les variables uniquement à des points fixés, le problème reste NP complet.

— Le problème reste NP complet même si l'on peut splitter les variables, en ajoutant l'hypothèse suivante : *seulement si il existe des instructions machine qui peuvent créer deux variables à la fois.*

5.6 Annale : graphes et complexité

Le problème suivant a été proposé aux préparationnaires du Capes NSI, Lyon1, en mars 2020.

Préambule

On rappelle les notions de base suivantes sur les graphes et la logique propositionnelle, suffisantes pour la bonne compréhension des trois problèmes à suivre.

Graphes

— Un graphe (orienté) G est un couple (S, A) où S est un ensemble fini de sommets et $A \subseteq S \times S$ est l'ensemble de ses arcs.

— Deux sommets x et y d'un graphe G=(S, A) seront dits *adjacents* si et seulement si $(x, y) \in A$ ou $(y, x) \in A$ (i.e., s'il existe un arc entre les deux sommets). L'adjacence est donc une relation symétrique.

— Soit un ensemble C fini de k *couleurs*. Un *k-coloriage* d'un graphe G est une fonction qui affecte une et une seule couleur à chaque sommet. Il est "correct" si et seulement si deux sommets adjacents n'ont pas la même couleur. Un graphe est *k-coloriable* s'il existe un *k-coloriage* correct pour ce graphe.

Logique propositionnelle

- Étant donné un vocabulaire logique V constitué de v variables propositionnelles $p_1, \ldots, p_v$, une conjonction de clauses est une formule logique de la forme $E_1 \wedge \ldots \wedge E_i \ldots \wedge E_m$. Chaque clause E_i est une disjonction de littéraux (différents deux à deux) $l_{i_1} \vee \ldots \vee l_{i_{k_i}}$, et un littéral est soit une variable propositionnelle p de V soit la négation $\neg p'$ d'une variable propositionnelle p' de V.
- Une interprétation I d'un ensemble $V = \{p_1, \ldots, p_v\}$ de variables propositionnelles est une fonction qui associe à chaque variable propositionnelle p_i la valeur $Vrai$ ou $Faux$.
  - L'évaluation d'une conjonction de clauses dans une interprétation I de ses variables propositionnelles renvoie $Vrai$ si *toutes* les clauses sont évaluées à $Vrai$ dans I, et $Faux$ sinon.
  - L'évaluation d'une clause dans une interprétation I renvoie $Vrai$ si *au moins* un de ses littéraux est évalué à $Vrai$ dans I et $Faux$ sinon.
  - Un littéral positif p_i est évalué à $Vrai$ dans I si $I(p_i) = Vrai$, à $Faux$ sinon.
  - Un littéral négatif $\neg p_i$ est évalué à $Faux$ dans I si $I(p_i) = Vrai$, à $Vrai$ sinon.
- Une conjonction de clauses est satisfaisable s'il existe une interprétation de ses variables propositionnelles dans laquelle toutes les clauses sont évaluées à $Vrai$.

  Exemple : La conjonction de clauses $(\neg p_1 \vee p_3) \wedge (p_2 \vee \neg p_3) \wedge (\neg p_2)$
  - est évaluée à $Vrai$ dans l'interprétation I définie par : $I(p_1) = I(p_2) = I(p_3) = Faux$,
  - est évaluée à $Faux$ dans l'interprétation I' : $I'(p_1) = I'(p_2) = Faux, I'(p_3) = Vrai$.

  Elle est satisfaisable (I suffit pour le prouver).

NP-complétude, réduction polynomiale Un problème $\mathscr{L}_2$ est NP-complet s'il est dans NP et s'il est NP-dur, i.e, si tout problème dans NP peut être réduit à $\mathscr{L}_2$, ce qui est équivalent à montrer qu'il existe une réduction polynômiale d'un problème NP-complet $\mathscr{L}_1$ *connu* vers $\mathscr{L}_2$.

On rappelle la définition formelle d'une réduction polynomiale d'un langage vers un autre langage : Une réduction polynomiale de L_1 *vers* L_2 est une fonction $f : \Sigma^* \to \Sigma^*$ *calculable en temps polynomial* telle que $x \in \mathscr{L}_1$ ssi $f(x) \in \mathscr{L}_2$ (transfert des solutions).

Il est demandé d'expliquer et justifier les exemples et les algorithmes.

5.6.1 Exemple 1 : coloriage de graphe non orienté

On code la relation d'adjacence associée à un graphe G de n sommets par un tableau T de taille $n \times n$ tel que : $T[i, j] = 1$ si les sommets i et j sont adjacents (cf. préambule), et $T[i, j] = 0$ sinon. On code un k-coloriage de G par un tableau d'entiers C à une dimension de taille n tel que $C[i]$ vaut la couleur affectée au sommet i dans le coloriage.

1. Dessiner le graphe dont la matrice d'adjacence est déclarée par : (on numérote les sommets de 0 à 5, les indices débutent à 0) :

```
adj = np.array ([[0, 1, 1, 0, 0, 0],
  [1, 0, 0, 1, 0, 0],
  [1, 0, 0, 0, 1, 0],
  [0, 1, 0, 0, 1, 0],
  [0, 0, 1, 1, 0, 1],
  [0, 0, 0, 0, 1, 0]])
```

   Dans la suite, nous nommons ce graphe G_1.

2. Écrire un programme Python prenant en entrée la matrice d'adjacence d'un graphe, un tableau représentant un coloriage, le nombre de noeuds du graphe, et qui vérifie si ce coloriage est correct :

```
coloriage=[1,1,2,2,1,0]
print is_ok (adj,coloriage,6)
#False
coloriage=[0,1,2,3,4,5]
print is_ok (adj,coloriage,6)
#True
```

3. Quel est l'ordre de grandeur de la complexité de ce programme en fonction du nombre n de sommets du graphe ?

   Une instance du problème de *k−coloriabilité* d'un graphe est la donnée d'un graphe et d'un entier, ie une paire (G,k). La question associée est "existe-t-il un k-coloriage pour G?"

4. Montrer que ce problème est NP.

   Intéressons nous maintenant à la 3-coloriabilité : étant donné un graphe G à n sommets, on modélise par la variable propositionnelle $x_{i,c}$ ($i \in [0..n-1]$ et $c \in [0..2]$) que le sommet i est colorié avec la couleur de numéro c.

5. — Traduire par une conjonction de clauses que chaque sommet est colorié par une et une et une seule des 3 couleurs.
   Indication : on exprimera par la clause $K(i)$ que le sommet i est colorié en au moins une des 3 couleurs. On exprimera ensuite par la clause $U(i)$ (respectivement $V(i)$ et $W(i)$) que le sommet i ne peut pas être colorié à la fois par la couleur 0 et la couleur 1 (respectivement la couleur 0 et la couleur 2, la couleur 1 et la couleur 2).
   — Écrire cette conjonction de clauses dans le cas particulier du graphe G_1 de la question 1.*Pour des raisons de longueur, vous pouvez vous restreindre aux sommets 3 et 4.*
   — Traduire par une conjonction de clauses que deux sommets voisins n'ont pas la même couleur.
   — Écrire cette conjonction de clauses dans le cas particulier du graphe G_1 de la question 1. *Pour des raisons de longueur, vous pouvez vous restreindre au sommet 4 et à ses arêtes adjacentes.*

6. En déduire l'existence d'une réduction polynômiale du problème de 3-coloriabilité de graphes vers SAT.

7. Peut-on en déduire que le problème de 3-coloriabilité de graphes est NP-complet ?

   Étant donnée une instance ϕ de 3-SAT, c'est-à-dire une conjonction $E_0 \wedge \ldots \wedge E_i \ldots \wedge E_{m-1}$ de m clauses, construite à partir de v variables propositionnelles $(p_0,\ldots,p_{v-1})$, où chaque clause contient exactement 3 littéraux (différents), on construit le graphe G_ϕ à $3v+m$ sommets défini de la manière suivante :

   — Ses sommets sont $p_0,\ldots,p_{v-1}$, $\neg p_0,\ldots,\neg p_{v-1}$, $y_0,\ldots,y_{v-1}$, $E_0,\ldots,E_{m-1}$.
   — Chaque sommet p_i est relié au sommet $\neg p_i$.
   — Chaque sommet y_i est relié :
     * à tous les sommets y_j tels que $j \neq i$,
     * à tous les sommets p_j tels que $j \neq i$,
     * et à tous les sommets $\neg p_j$ tels que $j \neq i$.
   — le sommet p_i est relié au sommet E_j si p_i *n'est pas un littéral* de la clause E_j.
   — le sommet $\neg p_i$ est relié au sommet E_j si $\neg p_i$ *n'est pas un littéral* de la clause E_j.

   Dans la suite on considérera l'ordre des sommets du graphe de taille $3v+m$ dans l'ordre suivant : d'abord les p_i, puis les $\neg p_i$, puis les y_i, puis les E_i. p_0 est donc le sommet de numéro 0, $\neg p_0$ le sommet de numéro $v,\ldots$.

8. Soit ϕ_1 l'instance de 3-SAT suivante :

$$(p_0 \vee \neg p_1 \vee \neg p_2) \wedge (\neg p_0 \vee p_1 \vee \neg p_2).$$

   Que valent v, m? Dessiner le graphe G_{ϕ_1} et donner sa matrice d'adjacence. *Les points de suspension sont autorisés...*

   On code en Python une instance de 3-SAT ϕ (avec les notations précédentes) sous la forme d'une matrice de F taille $m \times v$ telle que $F[i,j]$ vaut :

— 1 si p_i est un littéral de la clause E_j,
— -1 si $\neg p_i$ est un littéral de la clause E_j,
— et 0 si ni p_i ni $\neg p_i$ n'est un littéral de la clause E_j,

9. — Que vaut F dans le cas de la formule ϕ_1 ? On nomme cette matrice F_1.
   — Quel est le numéro du noeud représentant la clause E_j ?
   — Coder dans un programme Python la transformation qui prend le codage d'une instance de 3-sat (F, matrice de taille $m \times v$) et qui construit en sortie la matrice codant le graphe correspondant :

```
def transforme_formule(F,m,v):
    T=np.zeros((3*v+m,3*v+m),dtype=np.int32) #init de T avec des 0
    #à compléter
    return T
```

   L'appel `transforme_formule(F1,..,..)` doit retourner la matrice d'adjacence de G_{ϕ_1} obtenue à la question 8.
   — Quelle est la taille de l'entrée, et le nombre d'instructions effectuées par votre programme en fonction de m et v ?
10. Montrer que si $k < v$, alors il n'existe pas de k-coloriage correct du graphe.
11. On considère un $(v+1)$-coloriage qui associe la couleur i à l'un des deux membres de chaque paire de sommets $\{p_i, \neg p_i\}$, et la couleur $v+1$ à l'autre.
    Montrer que si $v > 4$, alors, dans tout $(v+1)$-coloriage correct qui étend aux sommets E_j le $(v+1)$-coloriage précédent, aucun sommet E_j n'a la couleur $v+1$.
12. Montrer comment associer une couleur parmi $1, \ldots, v$ à chaque sommet E_j de sorte que G soit $(v+1)$-coloriable si et seulement si l'ensemble de toutes les clauses E_j est satisfaisable. On pourra raisonner sur un exemple.
13. Peut-on en déduire que le problème de k-coloriage d'un graphe est NP-complet ?

5.6.2 Exemple 2 : chemins et circuits dans un graphe orienté

On considère ici la représentation des graphes orientés par liste d'adjacence. En Python on propose l'implémentation à l'aide d'un dictionnaire, par exemple :

```
g = {"a": ["d"],        "b": ["c"],        "c": ["b", "d"],
     "d": ["c", "b"],   "e": ["c"],        "f": []}
```

Recherche en Python

1. Donner un algorithme en Python qui permet de décider si un sommet d est accessible à partir d'un sommet s dans un graphe orienté.
2. Quelle est la complexité algorithmique de votre algorithme en considérant la représentation par liste d'adjacence ?
3. Donner un algorithme en Python qui permet de décider si deux sommets donnés font partie d'un même circuit. Quelle est sa complexité ?

Complexité de la recherche On s'intéresse à la réduction polynomiale de 2-SAT vers la recherche de chemin dans un graphe orienté. On rappelle que 2-SAT est la satisfiabilité d'une FNC (Forme Normale Conjonctive) F comportant **au maximum** 2 littéraux par clause.
On considère la transformation suivante d'une instance F de 2-SAT en un graphe orienté G appelé le *graphe d'implications* de F.

— Pour chaque variable propositionnelle x_i apparaissant dans F, G a deux sommets étiquetés x_i et $\overline{x_i}$.
— Pour chaque clause binaire $x_i \vee x_j$ de F, on crée une arête du sommet $\overline{x_i}$ vers le sommet x_j et une arête du sommet $\overline{x_j}$ vers le sommet x_i traduisant les implications "si x_i est faux alors x_j doit être vrai" et "si x_j est faux alors x_i doit être vrai". On rappelle que $\overline{\overline{x_i}} = x_i$.

— Pour chaque clause unaire x_i, on crée une arête du sommet $\overline{x_i}$ vers le sommet x_i.

4. Dessinez le graphe d'implications correspondant à la conjonction de clauses $(\overline{x_1} \vee x_2) \wedge (\overline{x_2} \vee x_3) \wedge (\overline{x_3} \vee x_1) \wedge x_1$.
5. Donnez l'ordre de grandeur de la complexité de la construction du graphe d'implications en fonction du nombre n de variables et du nombre m de clauses de l'instance 2-SAT à transformer.
6. Soit F une instance de 2-SAT et G le graphe d'implications correspondant.
   Montrez que, s'il existe dans G un circuit passant par deux sommets x_i et $\overline{x_i}$, alors F est insatisfiable.
   Indication : Montrez (par double implication) que les littéraux reliés par une chaîne fermée d'implications (un circuit du graphe G) ne peuvent qu'avoir la même valeur de vérité (Vrai ou Faux), dans une interprétation satisfiant F.
7. On considère que deux sommets sont équivalents ($u \approx v$) s'ils sont sur un même circuit. On construit le graphe G' à partir de G en remplaçant chaque ensemble de sommets reliés 2 à 2 par un circuit par un sommet unique $[u]$ correspondant à tous les sommets v tels que $u \approx v$), et en conservant les sommets non impliqués dans un circuit.
   Dans G', une arête relie deux sommets distincts $[u]$ et $[v]$ si et seulement si il existe une arête dans G reliant un des sommets regroupés dans $[u]$ avec un des sommets regroupés dans $[v]$.
   Remarques. Si G est sans circuit, $G' = G$. Par construction, si G a des circuits, G' est sans circuit. G' a donc au moins un sommet sans prédécesseur, on peut donc l'explorer en largeur à partir d'un tel sommet.
   Dessinez le graphe G' du graphe d'implications G de la question 1. Pourquoi G' est-il ainsi réduit ?
8. Les résultats des deux parties (chemins dans un graphe orientés) sont-ils cohérents ?

TD 6
Calculabilité

Rappel Pour montrer qu'un problème est indécidable, soit on montre de manière directe (cf. cours), soit on effectue une réduction, dont on rappelle la définition :

DÉFINITION 3. Soient L_1 et L_2 deux langages de Σ^*. Une *réduction* de L_1 à L_2 est une fonction récursive (*i.e.* calculable par machine de Turing) $\rho : \Sigma^* \to \Sigma^*$ telle que :

$$w \in L_1 \text{ ssi } \rho(w) \in L_2.$$

PROPOSITION 3. Pour montrer qu'un langage L n'est pas récursif (i.e est indécidable), il suffit d'exhiber un langage L' non récursif tel que L' se réduit à L.

En général, au lieu d'exhiber une machine de Turing qui réalise la réduction, on se contente de fournir un algorithme dans un langage "raisonnable". Ceci sera justifié par la section 6.3.

6.1 Trois problèmes indécidables

Le Xième problème de Hilbert peut s'énoncer de la façon suivante :

HILBERTX Soit p un polynôme à n variables à coefficients dans $\mathbb{Z}$, existe-t-il une racine entière, c'est à dire $x_1, \dots x_n \in \mathbb{Z}^n$ tels que $p(x_1, \dots, x_n) = 0$?

Il a été montré indécidable par Matiiassevitch en 1970.

EXERCICE #1 ▸ **Xth Hilbert**
Montrer par réduction à partir du Xième problème de Hilbert que le problème de satisfiabilité sur $\mathbb{R}$ des formules construites sur $\{0, 1, +, *, \sin\}$ est indécidable.

On rappelle $LU = \{< M, w > | M \text{ accepte } w\}$. Ce problème a été montré indécidable dans le cours.

EXERCICE #2 ▸ **Machines de Turing**
Soient M_1 et M_2 deux machines de Turing. Montrer que le problème $L(M_1) =? L(M_2)$. est indécidable.

EXERCICE #3 ▸ **Miroir**
Soit L un langage accepté par une machine de Turing et soit L^R le langage constitué des mots de L vus dans un miroir. Par exemple, si $L = \{a, ab, abb, abbb\}$, alors $L^R = \{a, ba, bba, bbba\}$. Montrer que le problème "est-ce que $L = L^R$?" est indécidable.

6.2 Indécidabilité du Problème de Correspondance de Post (PCP)

Soit Σ un alphabet fini [1], et $P \subseteq \Sigma^\star \times \Sigma^\star$ un ensemble fini de dominos étiquetés par des mots sur l'alphabet Σ (des paires de mots, donc). Le Problème de Correspondance de Post (PCP), introduit par Emil Post en 1946, consiste à déterminer s'il existe une séquence de dominos de P tels que le mot obtenu par la concaténation des premières composantes est identique à celui formé par la concaténation des secondes composantes. Plus formellement, on cherche à déterminer l'existence d'une suite $(u_i, v_i)_{0 \leqslant i \leqslant n}$ telle que : $\forall 0 \leqslant i \leqslant n, (u_i, v_i) \in P$ et $u_0 \cdot u_1 \cdots u_n = v_0 \cdot v_1 \cdots v_n$.

EXERCICE #4 ▸ **PCP**

1. Résoudre PCP **"à la main"** pour les instances suivantes. *Si une correspondance existe, on la donnera explicitement. Sinon, on justifiera soigneusement qu'une telle correspondance n'existe pas.*

1. *Rédaction par P. Brunet pour l'examen de MIF15 de déc. 2014*

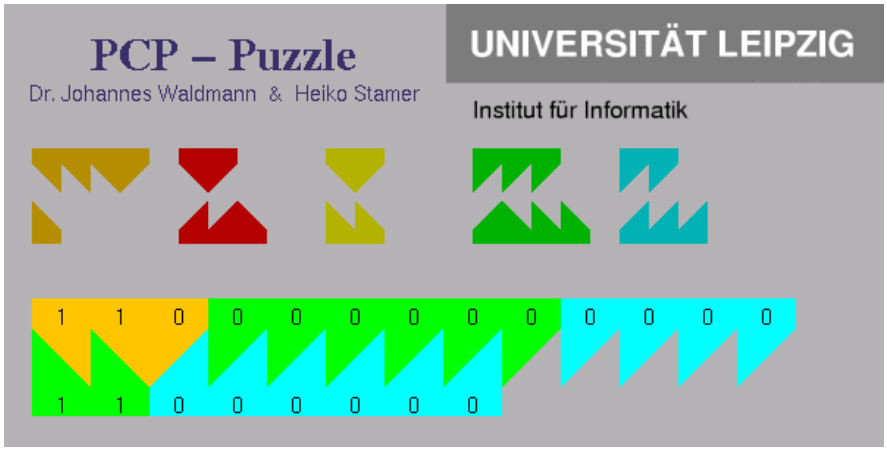


FIGURE 6.1 – PCP, version graphique, `http://freecode.com/projects/pcp-puzzle`

(a) $P_0 = \{(a, aaa), (aaaa, a)\}$;

(b) $P_1 = \{(aab, ab), (bab, ba), (aab, abab)\}$;

(c) $P_2 = \{(a, ab), (ba, aba), (b, aba), (bba, b)\}$;

(d) $P_3 = \{(ab, bb), (aa, ba), (ab, abb), (bb, bab)\}$.

2. Donner un algorithme pour décider PCP dans le cas où Σ ne contient qu'une seule lettre.

On considère maintenant une variante de PCP, le Problème de Correspondance de Post Modifié (PCPM). Pour ce problème, on considère un ensemble fini de dominos P et un domino initial $(u_0, v_0) \in P$, et on cherche à déterminer l'existence d'une correspondance qui commence par (u_0, v_0).

On considère comme acquis que PCPM est indécidable : en effet il existe une réduction du problème de l'arrêt d'une machine de Turing vers PCPM. On va maintenant montrer que PCP est indécidable, en réduisant depuis PCPM.

EXERCICE #5 ▸ **Réduction**

On commence par introduire deux fonctions p et s. Soit \$ un nouveau symbole n'appartenant pas à Σ, pour tout mot $w = a_1 a_2 \cdots a_n$ (les a_i sont les lettres), on définit :

$$\begin{cases} p(w) = \$a_1\$a_2 \cdots \$a_n \\ s(w) = a_1\$a_2\$ \cdots a_n\$ \end{cases}$$

1. Montrer que les fonctions p et s vérifient les propriétés suivantes :
   — pour deux mots v et w, $p(vw) = p(v)p(w)$ et $s(vw) = s(v)s(w)$;
   — pour tout mot w, $p(w)\$ = \$s(w)$.

   Soit $P, (u_0, v_0)$ une instance une PCPM, on définit $P' = \{(p(u_0), \$s(v_0))\} \cup P_1 \cup P_2$ avec :

$$\begin{cases} P_1 = \{(p(u), s(v)) \mid (u, v) \in P\} \\ P_2 = \{(p(u)\$, s(v)) \mid (u, v) \in P\} \end{cases}$$

2. Montrer que $P, (u_0, v_0)$ admet une solution pour PCPM si et seulement si P' admet une solution pour PCP.
3. En déduire que PCP est indécidable.

6.3 Machines alternatives

On a déjà vu que les programmes WHILE ont le même pouvoir que les machines de TUring, ici on explore un peu plus différents autres modèles de calcul.

6.3.1 Machines à piles

Une machine à k piles possède un nombre fini k de piles $r_1, \dots r_k$ qui correspondent à des piles d'éléments de Σ. Les instructions d'une machine à piles permettent seulement d'empiler un symbole sur l'une des piles, tester la valeur du sommet d'une pile, ou dépiler le symbole au sommet d'une pile. Si l'on préfère, on peut voir une pile d'éléments de Σ comme un mot w sur l'alphabet Σ. Empiler (*push*) le symbole a correspond à remplacer w par aw. Tester la valeur du sommet d'une pile (*top*) correspond à tester la première lettre du mot w. Dépiler (*pop*) le symbole au sommet de la pile correspond à supprimer la première lettre de w.

EXERCICE #6 ▸ **Simulation**
Montrer que tout machine de Turing peut être simulée par une machine à 2 piles. En déduire un problème indécidable pour les machines à 2 piles.

6.3.2 Machines à compteur

Une machine à n compteurs a plusieurs registres $r_1, \dots r_n$, appelés *compteurs*, chacun capable de stocker un nombre naturel. Un programme est une suite d'instructions de la forme :

1. `q:x++;goto p`
2. `q:x-;goto p`
3. `q:if x=0 then goto p else goto r`
4. `q:stop`

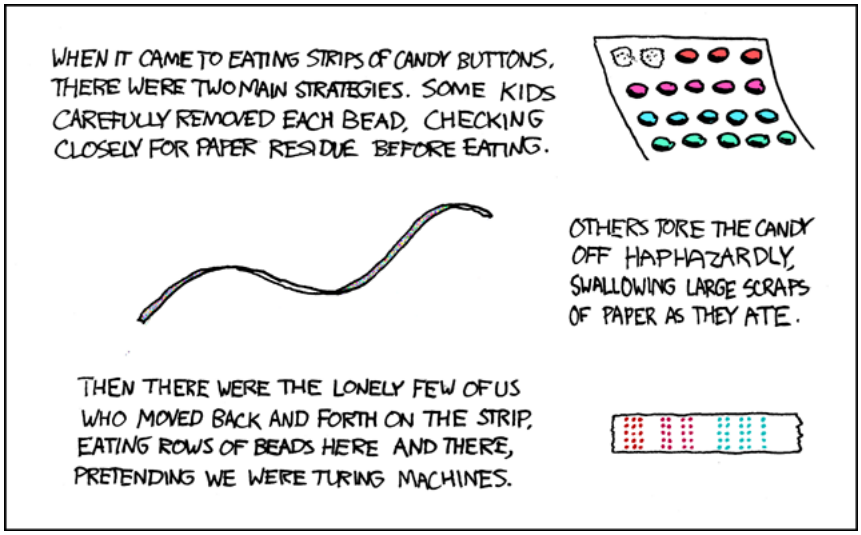
Calculer `0-` produit une erreur. Une configuration de la machine à n compteurs est le $n+1$-uplet $(q, x_1, \dots, x_n)$ qui représente l'étiquette de l'instruction courante et la valeur actuelle des compteurs. Pour l'entrée x, la configuration initiale est $(init, x, 0, \dots, 0)$, ou "init" est une instruction particulière, et qu'elle calcule jusqu'à *stop*. Le résultat est le contenu du registre x_1 après l'arrêt.

EXERCICE #7 ▸ **Calculons**
Montrer comment calculer $x \mapsto 2x$, et $x \mapsto x \mod 2$ avec une telle machine.

EXERCICE #8 ▸ **Indécidabilité de l'arrêt des machines à compteur**
Prouver que le problème de l'arrêt pour les machines à 4 compteurs est indécidable. On utilisera les étapes suivantes :

— Comment représenter un contenu de pile par un entier ?
— Comment réaliser les opérations Empiler et Depiler ?
— Comment finalement simuler le fonctionnement d'une machine à deux piles par une machine à 3 compteurs ?

Ce résultat est encore valable pour une machine à deux compteurs, via un encodage adhoc. En conséquence, les problèmes d'arrêt, calcul d'invariant, d'accessibilité d'un état mauvais, ... sont indécidables pour les automates à deux compteurs. Ce modèle de calcul plus pratique que les Machines de Turing est beaucoup utilisé en analyse de programme.



http://xkcd.com/205/