Sujet 0
Étude d'un problème informatique

Les questions de programmation doivent être traitées en langage Python. L'usage ou l'importation de tout module sera interdit. Lorsque le candidat écrira une fonction, il pourra faire appel à des fonctions définies dans les questions précédentes, même si elles n'ont pas été traitées. Il pourra également définir des fonctions auxiliaires, mais devra préciser leurs rôles ainsi que les types et significations de leurs arguments. Les candidats sont encouragés à expliquer les choix d'implémentation de leurs fonctions lorsque ceux-ci ne découlent pas directement des spécifications de l'énoncé. Si les paramètres d'une fonction à coder sont supposés vérifier certaines hypothèses, il ne sera pas utile dans l'écriture de cette fonction de tester si les hypothèses sont bien satisfaites.

On identifiera une même grandeur écrite dans deux polices de caractères différentes, en italique du point de vue mathématique (par exemple n) et en Computer Modern à chasse fixe du point de vue informatique (par exemple `n`).

Sans précision supplémentaire, lorsqu'une question demande la complexité d'une fonction, il s'agira de la complexité temporelle dans le pire des cas. On considérera que toutes les opérations arithmétiques sur les entiers s'effectuent en temps constant. La complexité sera exprimée sous la forme $\mathcal{O}(f(n,m))$ où n et m sont les tailles des arguments de la fonction, et f une expression simple. Les calculs de complexité seront justifiés succinctement. Lorsqu'une question de programmation précise qu'une complexité est attendue, sauf demande explicite, il n'est alors pas nécessaire de justifier que la fonction écrite vérifie cette contrainte.

1 Ordonnancement de tâches

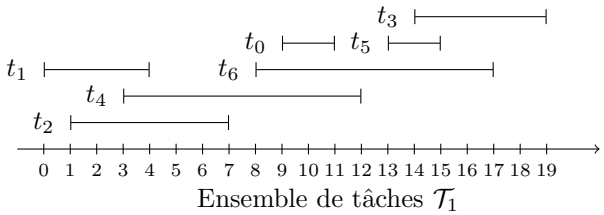
Cette partie est dédiée à l'étude d'un problème d'ordonnancement de tâches sur des machines. Chaque tâche est donnée sous la forme d'un intervalle de temps durant lequel une machine doit être dédiée à l'exécution de cette tâche. Étant donné un ensemble de tâches $t_0, ..., t_{n-1}$, l'objectif est de minimiser le nombre k de machines $M_0, ..., M_{k-1}$ utilisées pour toutes les exécuter. L'exécution d'une tâche représentée par un intervalle t_i sur une machine M_j occupe M_j durant la totalité de l'intervalle de temps t_i. Chaque machine ne peut donc exécuter qu'un ensemble de tâches dont les intervalles de temps sont disjoints.

Une tâche t_i est représentée en Python par un couple d'entiers $(\texttt{deb}_\texttt{i}, \texttt{fin}_\texttt{i})$ représentant respectivement la date de début et la date de fin de la tâche t_i. Un ensemble de tâches $t_0, ..., t_{n-1}$ est représenté par la liste $[(\texttt{deb}_0, \texttt{fin}_0), ..., (\texttt{deb}_{\texttt{n}-1}, \texttt{fin}_{\texttt{n}-1})]$ en Python et on suppose que **toutes les valeurs de débuts et de fins de tâches sont distinctes**. La numérotation des tâches étant arbitraire, plusieurs représentations sont possibles pour un ensemble donné.

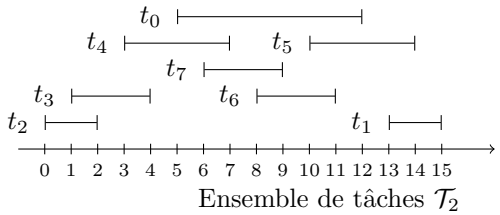
Un *ordonnancement* est une liste Python `ordo` de longueur `n` à valeur dans $[\![0, k-1]\!]$. Si `ordo[i] = j`, cela signifie que la tâche t_i est exécutée sur la machine M_j. On dit qu'un ordonnancement est *cohérent* si pour tout couple (i, i') tel que $i \neq i'$ et `ordo[i] = ordo[i']`, on a $t_i \cap t_{i'} = \emptyset$; c'est-à-dire que deux tâches exécutées sur la même machine correspondent toujours à des intervalles disjoints. Le nombre de machines utilisé par un ordonnancement `ordo` est le nombre de valeurs différentes dans la liste `ordo`. Un ordonnancement cohérent est *optimal* s'il n'existe pas d'ordonnancement cohérent utilisant strictement moins de machines que `ordo`.

1.1 Représentations des ensembles de tâches

Pour faciliter les raisonnements sur les ensembles de tâches, on les représente graphiquement comme suit :



Ensemble de tâches $\mathcal{T}_1$



Ensemble de tâches $\mathcal{T}_2$

L'ensemble de tâches $\mathcal{T}_1$ représenté ci-dessus est donc constitué de la liste d'intervalles

$$\texttt{T1} = [(\texttt{9, 11}),\ (\texttt{0, 4}),\ (\texttt{1, 7}),\ (\texttt{14, 19}),\ (\texttt{3, 12}),\ (\texttt{13, 15}),\ (\texttt{8, 17})].$$

Ici, l'axe horizontal représente le temps, il est gradué sur les entiers à partir de 0. Les différences de hauteur des intervalles dans les représentations graphiques n'ont pas d'importance, elles ne sont présentes que pour la lisibilité.

1. Sans justifier, donner la liste d'intervalles de l'ensemble de tâches $\mathcal{T}_2$ représenté ci-dessus.
2. Donner un ordonnancement optimal de l'ensemble $\mathcal{T}_1$, puis de l'ensemble $\mathcal{T}_2$. Justifier que l'on ne pourrait pas utiliser moins de machines.

Avant de s'intéresser au calcul d'un ordonnancement optimal, on commence par voir comment calculer le nombre de machines nécessaire à un ordonnancement optimal grâce à un simple parcours des dates de débuts et de fins de tâches dans l'ordre croissant. On définit la notion d'*événement* (date de début ou date de fin d'une tâche) par un triplet `(date, ind, type)` où `date` est l'entier représentant la date de l'événement, `ind` est l'indice de la tâche dont cette date est une extrémité, `type` vaut 0 ou 1 selon si la date est le début de la tâche t_{ind} ou sa fin. La liste des tâches peut alors être représenté par une liste d'événements triés dans l'ordre croissant des dates. Les dates étant toutes différentes, cette représentation est unique.

L'ensemble de tâches $\mathcal{T}1$ évoqué précédemment est donc représenté par la liste d'événements suivante :

`Ev1` = [(0, 1, 0), (1, 2, 0), (3, 4, 0), (4, 1, 1), (7, 2, 1), (8, 6, 0), (9, 0, 0), (11, 0, 1), (12, 4, 1),
(13, 5, 0), (14, 3, 0), (15, 5, 1), (17, 6, 1), (19, 3, 1)].

3. Sans justifier, donner les 7 premiers éléments de la liste triée d'événements représentant l'ensemble de tâches $\mathcal{T}_2$.
4. Écrire une fonction `formate(T)` qui prend en argument une liste d'intervalles représentant un ensemble de tâches et renvoie la liste des événements triés dans l'ordre croissant des valeurs. Ainsi, l'appel `formate(T1)` doit renvoyer la liste `Ev1`. Cette fonction doit agir par insertions successives dans une liste triée des événements associées aux intervalles. L'usage d'une fonction de tri pré-implémentée est proscrite. La complexité attendue est quadratique en la longueur de la liste passée en argument.
5. Justifier que la fonction `formate` est bien de complexité quadratique en la longueur de la liste passée en argument et expliquer brièvement comment améliorer la complexité de cette fonction.

Dans la suite de cette partie, on considère que les ensembles de tâches sont donnés sous forme d'une liste triée d'événements.

1.2 Calcul du nombre maximum de tâches simultanées

Pour un ensemble $\mathcal{T}$ de tâches donné, on souhaite calculer $\mathcal{K}(\mathcal{T})$ le nombre maximum de tâches simultanément en cours d'exécution. Par exemple, pour l'ensemble $\mathcal{T}_1$, il y a au plus $\mathcal{K}(\mathcal{T}_1) = 3$ tâches qui s'exécutent au même moment (par exemple, les tâches t_1, t_2 et t_4 entre les dates 3 et 4).

6. Sans justifier, donner $\mathcal{K}(\mathcal{T}_2)$.
7. Écrire une fonction `nb_simultanees(Ev)` qui prend en argument la liste des événements triés d'un ensemble $\mathcal{T}$ de tâches et qui renvoie $\mathcal{K}(\mathcal{T})$. Cette fonction devra avoir une complexité linéaire en la taille de la liste.
8. Donner et prouver un invariant de la fonction précédente et l'utiliser pour prouver sa correction

1.3 Calcul d'un ordonnancement optimal

En s'inspirant du calcul du nombre maximum de tâches simultanées, on peut écrire un algorithme glouton qui calcule un ordonnancement pour un ensemble $\mathcal{T}$ de tâches donné. Pour cela, on parcourt la liste triée des événements : à chaque fois que l'on rencontre un début de tâche, on lui attribue la machine disponible de plus petit indice et cette machine ne sera à nouveau disponible qu'au moment de la date de fin de cette tâche. Pour l'ensemble $\mathcal{T}_1$, l'ordonnancement obtenu est : [1, 0, 1, 2, 2, 1, 0].

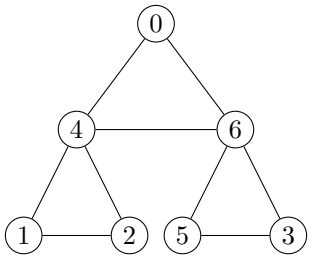
9. Sans justifier, donner l'ordonnancement obtenu de cette façon pour $\mathcal{T}_2$.
10. Écrire une fonction `ordo_glouton(Ev)` qui prend en argument la liste des événements triés d'un ensemble $\mathcal{T}$ de tâches et qui renvoie l'ordonnancement construit comme décrit ci-dessus.
11. Prouver que l'ordonnancement calculé par la fonction `ordo_glouton` est cohérent.
12. Prouver que cet ordonnancement est optimal.
13. Quelle est la complexité de la fonction `ordo_glouton` en fonction de n ?

2 Lien avec les graphes

Dans cette partie, on étudie une représentation des ensembles de tâches sous forme d'un graphes. On ne considère que des graphes sans boucle et sans multi-arête.

Un *graphe* est un couple (S, A) où $S = \{0, ..., n-1\}$ est l'*ensemble des sommets* et $A \subseteq S \times S \setminus \{(s, s) | s \in S\}$ est l'*ensemble des arêtes*. Dans ce sujet, tous les graphes sont non-orientés, donc pour tout couple $(s, s') \in S \times S$, $(s, s') \in A$ si et seulement si $(s', s) \in A$. Dans la suite, on représente les graphes à l'aide de la représentation graphique usuelle dès que possible. De plus, les graphes sont implémentés **par listes d'adjacence**.

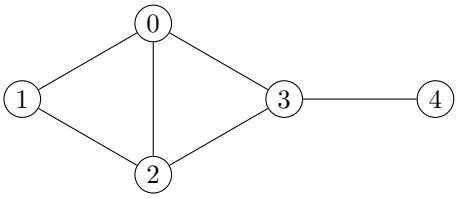
Soit $\mathcal{T} = \{t_0, ..., t_{n-1}\}$ un ensemble de tâches. On définit le graphe $G_\mathcal{T} = (S_\mathcal{T}, A_\mathcal{T})$ associé à $\mathcal{T}$ comme suit :
— $S_\mathcal{T} = \{0, ..., n-1\}$
— $A_\mathcal{T} = \{(s, s') \in S_\mathcal{T} \times S_\mathcal{T} | t_s \cap t_{s'} \neq \emptyset\}$.
Par exemple, le graphe associé à $\mathcal{T}_1$ est le suivant :



Un graphe construit de cette façon est un *graphe d'intervalles* et l'ensemble de tâches utilisé est une *réalisation* de ce graphe.

Le graphe d'intervalles est une représentation moins précise qu'une réalisation car on ne garde que l'information sur les intersections entre les intervalles des tâches. Deux ensembles de tâches distincts peuvent être des réalisations du même graphe d'intervalle. Par exemple, l'ensemble de tâches [(5, 8), (9, 11), (10, 12), (1, 3), (6, 13), (2, 4), (0, 7)] est une autre réalisation du graphe d'intervalles associé à $\mathcal{T}_1$.

14. Représenter graphiquement le graphe d'intervalles associé à l'ensemble $\mathcal{T}_2$.
15. Écrire une fonction `liste_vers_graphe(T)` qui prend en argument une liste d'intervalles représentant un ensemble de tâches et renvoie le graphe associé.
16. Déterminer une réalisation du graphe d'intervalles suivant :



    Représenter graphiquement l'ensemble des tâches pour plus de lisibilité.

Cette construction n'est pas possible pour tous les graphes. Considérons par exemple les graphes cycliques $\mathcal{C}_n = (S_n, A_n)$ pour $n \geq 4$ définis comme suit :
— $S_n = \{0, ..., n-1\}$
— $A_n = \{(i, i+1) | i \in \{0, ..., n-2\}\} \cup \{(n-1, 0)\}$.

17. Prouver qu'il n'existe pas de réalisation pour $\mathcal{C}_4$.
18. Généraliser la démonstration aux graphes $\mathcal{C}_n$ avec $n \geq 4$.

L'ensemble des graphes d'intervalles est donc strictement inclus dans l'ensemble des graphes. Il ne contient pas des graphes simples comme les graphes cycliques, mais il est stable par sous-graphes induits. On rappelle qu'un sous-graphe induit d'un graphe $G = (S, A)$ est un graphe $G' = (S', A')$ tel que $S' \subseteq S$ et $A' = A \cap (S' \times S')$, c'est-à-dire un graphe obtenu à partir de G en supprimant certains sommets et les arêtes incidentes à ces sommets.

19. Prouver que tout sous graphe induit d'un graphe d'intervalles est un graphe d'intervalles.

Dans un graphe G, on dit que $s_0, s_1, ..., s_k$ avec $s_k = s_0$ est un *cycle* de longueur k si pour tout $i \in [\![0, k]\!]$ $(s_i, s_{i+1}) \in A$. On dit qu'un tel cycle admet une *corde* s'il existe deux sommets non-consécutifs du cycle qui sont adjacents.

20. Que peut-on alors dire d'un graphe qui contient un cycle de longueur supérieure ou égale à 4 sans corde ?

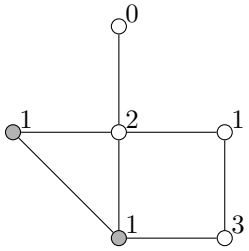
3 Coloration de graphes

Le problème de l'ordonnancement d'un ensemble de tâches correspond au problème de la coloration d'un graphe d'intervalles. Dans cette partie, on étudie la coloration des graphes quelconques. La dernière partie est dédiée à la résolution dans la cas particulier des graphes d'intervalles.

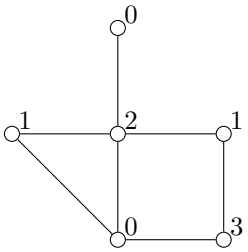
3.1 Préliminaires

Pour un graphe $G = (S, A)$, une *coloration* de G est une numérotation des sommets de G de telle sorte que deux sommets adjacents n'ont jamais la même couleur, c'est-à-dire jamais le même numéro. Plus formellement, pour un entier $k \in \mathbb{N}^*$, une *k-coloration* de G est une fonction $c : S \to \{0, \dots, k-1\}$ telle que pour toute arête $(s, t) \in A$, $c(s) \neq c(t)$.

Voici deux numérotations possibles d'un même graphe G_0 :



Une numérotation qui n'est pas une coloration

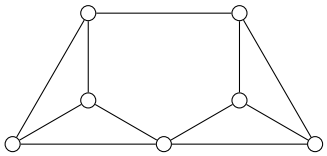


Une 4-coloration

La première n'est pas une coloration car les sommets grisés sont adjacents et ont le même numéro. La seconde est une 4-coloration. Attention, dans les représentations ci-dessus, les numéros à l'extérieur des sommets ne correspondent pas aux indices des sommets mais bien à la numérotation des couleurs.

G est dit *k-colorable* s'il possède une k-coloration. On définit également le *nombre chromatique* de G, noté $\chi(G)$, comme étant le plus petit entier k tel que G est k-colorable.

21. Déterminer le nombre chromatique du graphe G_0 représenté ci-dessus. On exhibera une coloration optimale, et on justifiera rigoureusement qu'il est impossible d'utiliser moins de numéros différents.
22. Faire de même pour le graphe G_1 ci-dessous.



Le graphe G_1

23. (a) Déterminer en justifiant le nombre chromatique de $\mathcal{C}_n$ en fonction de n.
    On rappelle que $\mathcal{C}_n$ est défini avant la question 16.
    (b) On rappelle qu'un graphe est complet si tous ses sommets sont deux à deux adjacents. Déterminer en justifiant le nombre chromatique d'un graphe complet à n sommets en fonction de n.

Dans un graphe $G = (S, A)$, le *degré* d'un sommet $s \in S$ est le nombre de sommets adjacents à s. Une *clique* de G est un ensemble de sommets deux à deux adjacents. Le *nombre clique* de G, noté $\omega(G)$, est la taille maximale d'une clique de G.

24. Pour un ensemble de tâches $\mathcal{T}$, comparer $\mathcal{K}(\mathcal{T})$ et $\omega(G_{\mathcal{T}})$.
    On rappelle que $\mathcal{K}$ est défini au début de la partie 1.2 et que $G_{\mathcal{T}}$ est défini au début de la partie 2.
25. Déterminer en justifiant une inégalité entre $\omega(G)$ et $\chi(G)$. Pour n un entier naturel $\geqslant 3$ arbitraire, décrire un graphe connexe à n sommets pour lequel cette inégalité est en fait une égalité et un graphe connexe à n sommets pour lequel cette inégalité est stricte.

On note $\Delta(G)$ le degré maximal d'un sommet de G.

26. Déterminer sans justifier une inégalité entre $\Delta(G)$ et $\chi(G)$. Pour n un entier naturel $\geqslant 3$ arbitraire, décrire un graphe connexe à n sommets pour lequel cette inégalité est en fait une égalité et un graphe connexe à n sommets pour lequel cette inégalité est stricte.

3.2 Calcul du nombre chromatique

Dans cette partie, on cherche à calculer la valeur exacte du nombre chromatique d'un graphe. Pour ce faire, on va naïvement tester toutes les numérotations possibles, ne considérer que celles qui sont des colorations, et déterminer une de celles qui utilisent le plus petit nombre de couleurs. On représentera une numérotation $c : S \to \{0, \dots, k-1\}$ par une liste `c` de taille $|S|$ contenant des éléments de $\{0, \dots, k-1\}$, telle que `c[s]` prend la valeur $c(s)$, pour tout sommet $s \in S$.

27. Pour un graphe G à n sommets, quelle est la valeur maximale possible pour $\chi(G)$ en fonction de n ? Justifier.
28. Écrire une fonction `est_coloration(G, c)` qui prend en arguments un graphe `G` représenté par liste de listes d'adjacence et une numérotation `c` et renvoie `True` si `c` est une coloration de `G` et `False` sinon. On supposera sans le vérifier que `G` et `c` sont de mêmes tailles.

Pour la suite, on souhaite énumérer toutes les numérotations possibles d'un graphe à n sommets, utilisant $k \leqslant n$ couleurs distinctes, c'est-à-dire les k^n listes de taille n contenant des éléments de $\{0, \dots, k-1\}$. L'ordre choisi pour cette énumération est l'ordre lexicographique.

29. Écrire une fonction `incrementer(c, k)` qui prend en arguments une numérotation `c` et un entier `k`, modifie `c` en la numérotation suivante selon l'ordre lexicographique de $\{0, \dots, k-1\}^n$, et renvoie `True` si (avant modification) `c` est la liste ne contenant que la valeur $k-1$ (c'est-à-dire si on a atteint la dernière numérotation) et `False` sinon. Si `c` ne contient que la valeur $k-1$, on modifiera `c` pour qu'elle ne contienne que la valeur 0.

    Cette fonction devra avoir une complexité en $\mathcal{O}(n)$.

    Par exemple, `incrementer([2, 1, 0, 2, 2], 3)` modifie la liste en `[2, 1, 1, 0, 0]` et renvoie `False`. De plus, `incrementer([2, 2, 2, 2, 2], 3)` modifie la liste en `[0, 0, 0, 0, 0]` et renvoie `True`.
30. En utilisant les deux fonctions précédentes, en déduire une fonction `k_colorable(G, k)` qui prend en arguments un graphe `G` et un entier `k` et renvoie `True` si G est k-colorable et `False` sinon.
31. En déduire une fonction `chi(G)` qui calcule et renvoie $\chi(G)$.
32. Déterminer les complexités temporelles et spatiales dans le pire des cas de la fonction `chi` en fonction de n, le nombre de sommets de G. Cette fonction est-elle utilisable en pratique pour des graphes de taille 50 ? de taille 20 ?

3.3 Problème de décision et NP-complétude

Dans cette partie, on cherche à montrer que le problème général du calcul du nombre chromatique d'un graphe est difficile. Pour ce faire, on étudie plutôt des problèmes de décision liés à la coloration. On définit les problèmes de décision suivants :

— Problème k-coloration (pour k fixé)
  * Entrée : un graphe G.
  * Question : le graphe G est-il k-colorable ?

— Problème Coloration
  * Entrée : un graphe G et un entier k.
  * Question : le graphe G est-il k-colorable ?

33. Montrer que pour tout entier k, k-coloration est réductible en temps polynomial à Coloration.
34. Montrer que pour $h \leqslant k$ deux entiers fixés, h-coloration est réductible en temps polynomial à k-coloration.
35. Écrire une fonction `deux_colorable(G)` qui prend en argument un graphe `G` et renvoie `True` si G est 2-colorable et `False` sinon. On utilisera un algorithme de parcours de graphe au choix. Cette fonction devra avoir une complexité linéaire en le nombre de sommets et d'arêtes de G.

Les deux questions précédentes montrent que les problèmes 1-coloration et 2-coloration sont dans P. Les questions suivantes montrent la NP-complétude de 3-coloration.

36. Montrer que Coloration est dans NP. On déterminera un certificat de taille polynomiale prouvant qu'un graphe G est k-colorable.

On rappelle les définitions suivantes : pour un ensemble de variables $V = \{v_1, \dots, v_n\}$, un *littéral* est une variable ou sa négation, c'est-à-dire de la forme v_i ou $\overline{v_i}$. Une *clause* est une disjonction de littéraux (séparés par des $\vee$). Une formule est dite en *forme normale conjonctive* si c'est une conjonction de clauses (séparées par des $\wedge$). Elle est dite en 3-FNC si de plus chaque clause contient exactement 3 littéraux. Par exemple, la formule φ_0 suivante est en 3-FNC :

$$\varphi_0 = (a \vee b \vee c) \wedge \overbrace{(\overline{a} \vee c \vee \overline{d})}^{\text{clause}} \wedge (b \vee \underbrace{\overline{c}}_{\text{littéral}} \vee d)$$

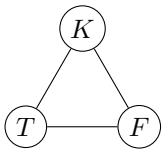
On définit alors le problème de décision :

— Problème 3-SAT
  * Entrée : une formule booléenne φ en 3-FNC.
  * Question : la formule φ est-elle satisfiable ?

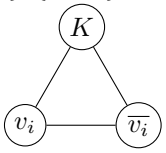
On admet que ce problème est NP-complet. La suite de cette partie a pour objectif de montrer qu'il se réduit en temps polynomial au problème 3-coloration. Pour ce faire, en considérant une formule booléenne φ en 3-FNC, on construit un graphe G_φ qui est 3-colorable si et seulement si φ est satisfiable. L'idée est que deux des couleurs utilisées correspondent à l'affectation *Vrai* ou *Faux* pour chaque littéral, et la troisième couleur est une couleur de contrôle.

Dans le détail, si φ est une formule en 3-FNC sur un ensemble $V = \{v_1, \dots, v_n\}$ de n variables, alors on définit le graphe $G_\varphi = (S, A)$ par :

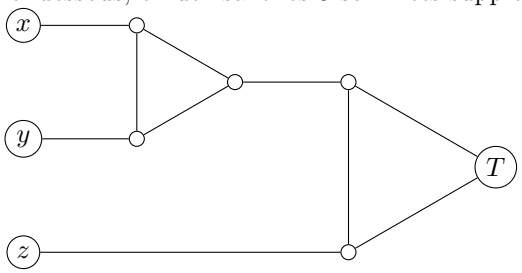
— S contient trois sommets T (*True*), F (*False*) et K (*Control*), un sommet par littéral possible, et 5 sommets supplémentaires par clause ;
— A contient les arêtes suivantes :
  — $\{T, F\}, \{T, K\}, \{F, K\}$



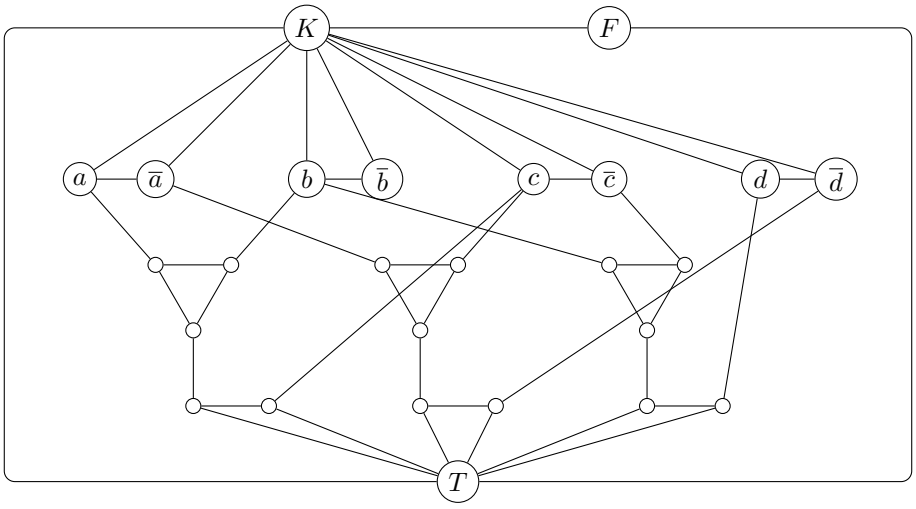
  — pour chaque variable v_i, les arêtes $\{v_i, \overline{v_i}\}, \{v_i, K\}, \{\overline{v_i}, K\}$



  — pour chaque clause $C = x \vee y \vee z$ apparaissant dans φ (où x, y, z sont des littéraux), on rajoute les 10 arêtes telles que décrites dans le graphe ci-dessous, en utilisant les 5 sommets supplémentaires associés à la clause.

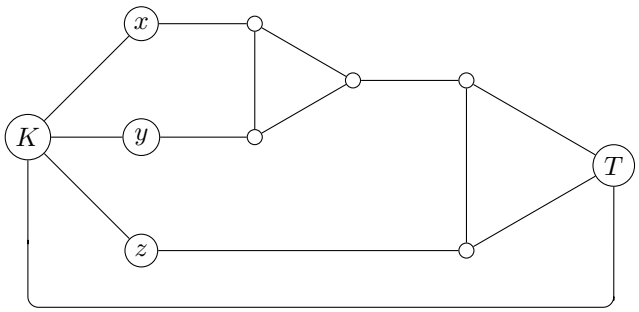


Par exemple, voici le graphe G_{φ_0} obtenu pour la formule booléenne φ_0 sur l'ensemble de variables $\{a, b, c, d\}$.



Le graphe G_{φ_0} pour $\varphi_0 = (a \vee b \vee c) \wedge (\overline{a} \vee c \vee \overline{d}) \wedge (b \vee \overline{c} \vee d)$

37. Représenter graphiquement le graphe G_{φ_1} pour $\varphi_1 = (\overline{a} \vee b \vee \overline{c}) \wedge (a \vee b \vee \overline{c}) \wedge (a \vee \overline{b} \vee c)$.
38. Justifier que la construction de G_φ à partir d'une formule φ en 3-FNC se fait en temps polynomial en fonction de la taille de φ (la taille d'une formule en 3-FNC est son nombre de clauses).
39. Montrer que le graphe suivant est 3-colorable et que pour toute 3-coloration, au moins l'un des sommets x, y ou z a la même couleur que T.



40. Montrer que φ est satisfiable si et seulement si G_φ est 3-colorable. On construira une distribution de vérité de φ à partir d'une 3-coloration de G_φ, et réciproquement.
41. En déduire que 3-coloration est NP-difficile.

3.4 Algorithmes gloutons

Comme vu précédemment, le calcul du nombre chromatique d'un graphe est un problème difficile. On se propose donc d'étudier dans cette partie des algorithmes gloutons efficaces permettant de déterminer des k-colorations d'un graphe G, sans garantir que $k = \chi(G)$. Le schéma général de tous ces algorithmes est le même, seul l'ordre de traitement des sommets pouvant différer.

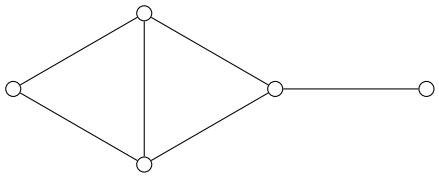
Début algorithme
  Entrée : graphe $G = (S, A)$
  Pour chaque sommet $s \in S$ non coloré **Faire**
    Colorer s en utilisant la plus petite couleur possible

La ligne « Colorer s en utilisant la plus petite couleur possible » consiste à parcourir tous les sommets colorés adjacents à s et à numéroter s en utilisant le plus petit numéro n'apparaissant pas parmi les couleurs des voisins de s. On attribue toujours la couleur 0 au premier sommet traité.

42. Écrire une fonction `plus_petit_absent(L)` qui prend en argument une liste `L` d'entiers et renvoie le plus petit entier positif ou nul n'apparaissant pas dans `L`. Cette fonction devra avoir une complexité linéaire en la taille de `L` dans tous les cas.

L'algorithme glouton le plus élémentaire consiste à parcourir les sommets dans l'ordre croissant des indices.

43. Écrire une fonction `colo_glouton(G)` qui prend en argument un graphe `G` et renvoie une coloration de G en suivant l'algorithme glouton.
44. Déterminer la complexité temporelle de `colo_glouton`.
45. Déterminer une indexation des sommets telle que pour le graphe G_2 ci-dessous, la fonction `colo_glouton` ne renvoie pas une coloration optimale (c'est-à-dire utilisant strictement plus de $\chi(G_2)$ couleurs).



Le graphe G_2

46. Montrer que pour tout graphe G, il existe une indexation des sommets telle que la fonction `colo_glouton` renvoie une coloration optimale.

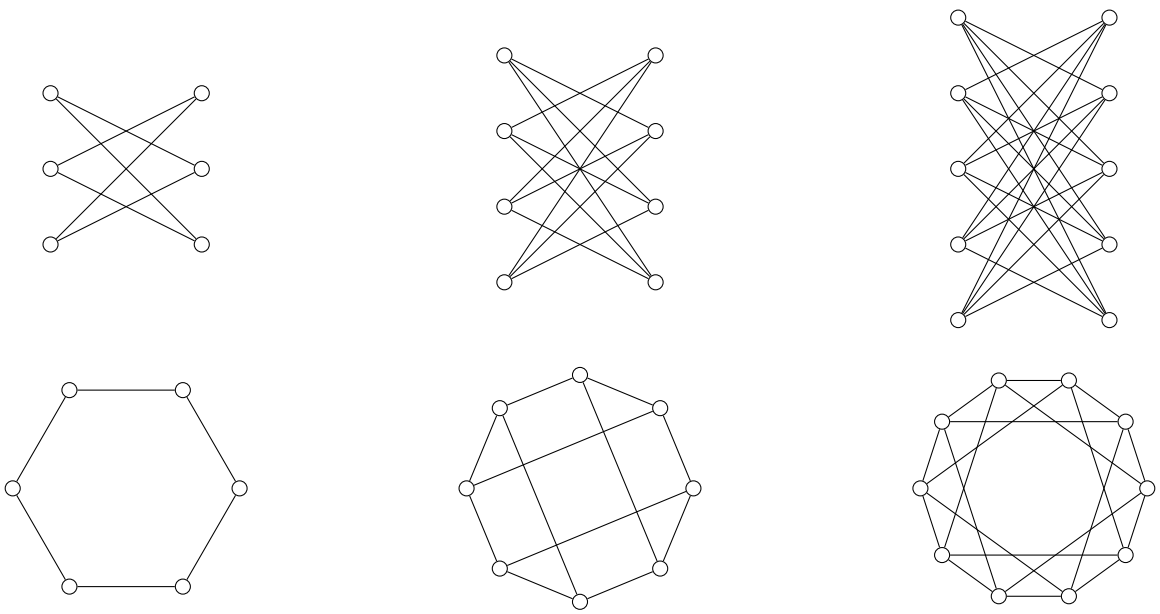
L'algorithme de Welsh-Powell consiste à traiter les sommets par ordre décroissant de degré. En cas d'égalité de degrés, l'algorithme choisit le sommet de plus petit indice.

47. Montrer que l'algorithme de Welsh-Powell renvoie toujours une coloration optimale pour le graphe G_2, quelle que soit l'indexation des sommets.

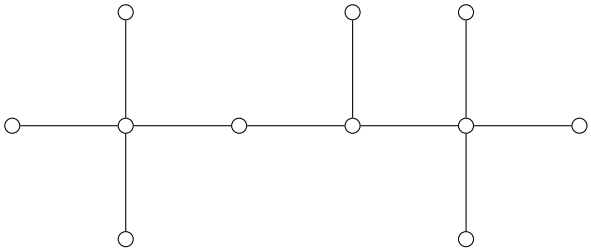
Pour n un entier supérieur ou égal à 3, on appelle *graphe couronne à $2n$ sommets*, noté J_n, le graphe biparti $J_n = (S, A)$ défini par :

— $S = X \sqcup Y$ où $X = \{x_1, \dots, x_n\}$ et $Y = \{y_1, \dots, y_n\}$;
— $A = \{\{x_i, y_j\}, i \neq j\}$, c'est-à-dire qu'il existe une arête entre un sommet x_i et un sommet y_j si i et j sont distincts.

Ci-après, deux représentations possibles des graphes couronnes à 6, 8 et 10 sommets.



48. Montrer que pour n supérieur ou égal à 3, il existe une indexation des sommets de J_n telle que l'algorithme de Welsh-Powell renvoie une 2-coloration, et une indexation telle que l'algorithme de Welsh-Powell renvoie une n-coloration.
49. Montrer que quelle que soit l'indexation des sommets du graphe G_3 suivant, l'algorithme de Welsh-Powell renvoie une coloration non optimale.



Le graphe G_3

4 Coloration de graphes d'intervalles

Dans la partie précédente, on a montré les limites de l'approche gloutonne pour la coloration d'un graphe quelconque. Dans cette partie, on fait le lien entre ordonnancement et coloration.

50. Soit $G = (S, A)$ un graphe d'intervalles et r une réalisation de G. Prouver que `ordo` est un ordonnancement cohérent de r si et seulement si c qui à $s \in S$ associe la couleur `ordo[s]` est une coloration de G.
51. Soit G un graphe d'intervalles et r une réalisation de G. Déterminer un ordre de traitement des sommets permettant à l'algorithme glouton de renvoyer une coloration optimale de G.

Le problème de la coloration de graphes d'intervalles peut ainsi être résolu efficacement si l'on dispose d'une réalisation de ce graphe. La partie suivante montre que même sans connaître de réalisation, le problème de la coloration admet toujours une résolution efficace pour les graphes d'intervalles. Pour cela, on étudie le parcours en largeur lexicographique et ses propriétés.

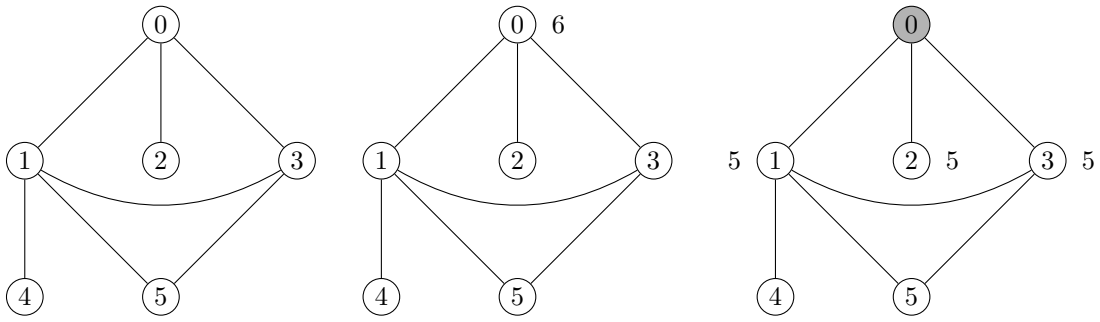
4.1 Parcours en largeur lexicographique

Le parcours en largeur lexicographique est un parcours en largeur où l'on contraint l'ordre de parcours des voisins d'un même sommet. Dans le parcours en largeur classique, on parcourt les voisins de chaque sommet dans un ordre arbitraire (ici on considère que le parcours se fait par indices de sommets croissants). Pour le parcours en largeur lexicographique, on associe à chaque sommet une étiquette que l'on construit au fur et à mesure et le parcours des voisins d'un sommet se fait en sélectionnant toujours le sommet dont l'étiquette est maximale pour l'ordre lexicographique. **Si deux sommets ont la même étiquette, on commence par celui d'indice minimum.**

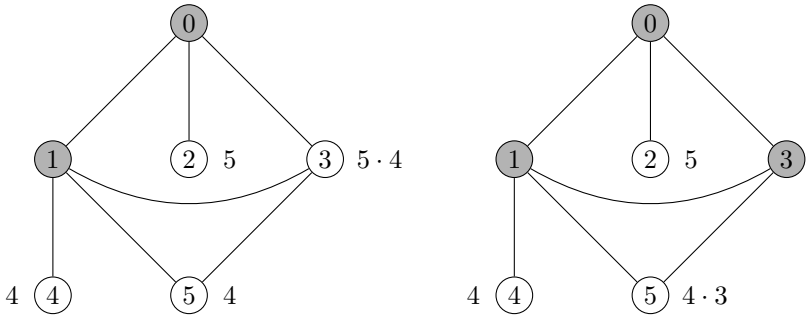
On appelle LexBFS l'algorithme de parcours en largeur lexicographique, cet algorithme est résumé ci-après. On attribue pour cela une étiquette à chaque sommet. Lorsque $S = \{0, 1, \ldots, n-1\}$, une *étiquette* est un mot sur l'alphabet $\Sigma = \{1, 2, \ldots, n\}$. La concaténation entre deux mots u et v de Σ^* est dénotée par $u \cdot v$. Le mot vide est dénoté ε.

Début algorithme
  Entrée : graphe $G = (S, A)$ connexe
  Sortie : ordre de parcours σ
  Pour chaque sommet $s \in S$ **Faire**
    étiquette$(s) := \varepsilon$
  étiquette$(0) := |S|$
  Pour i de $|S| - 1$ à 1 **Faire**
    Choisir s un sommet non traité d'étiquette maximale
    $\sigma(|S| - i - 1) := s$
    Pour chaque sommet t non traité, voisin de s **Faire**
      étiquette$(t) :=$ étiquette$(t) \cdot i$
  Renvoyer σ

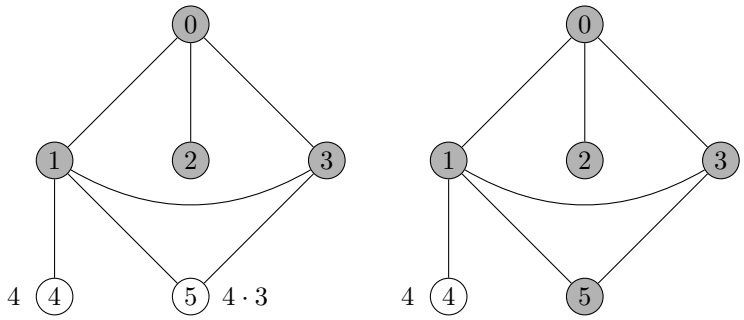
Voici le déroulement de cet algorithme pas à pas sur un graphe simple. Pour une meilleure lisibilité, les étiquettes vides et celles des sommets déjà traités ne sont pas représentées.



Au départ, les étiquettes sont toutes vides sauf le sommet 0 qui prend l'étiquette 6. Le parcours commence donc au sommet d'indice 0. Lors du traitement de 0, on concatène la valeur 5 aux étiquettes de tous ses voisins.

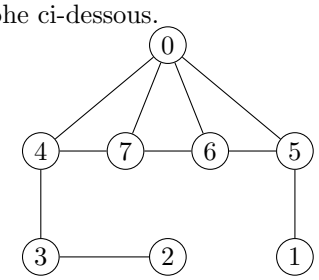


Le parcours continue à partir du sommet d'indice 1 car c'est le plus petit indice des sommets de d'étiquettes maximales. Lors du traitement du sommet 1, on concatène la valeur 4 aux étiquettes de tous ses voisins qui n'ont pas encore été traités. Le parcours continue alors sur le sommet d'indice 3 dont l'étiquette $5 \cdot 4$ est maximale pour l'ordre lexicographique. La valeur 3 est concaténée à l'étiquette du voisin non encore traité.



Le parcours continue sur le sommet 2 dont l'étiquette est à présent maximale, mais qui n'a pas de voisin. Puis le parcours se termine par les sommets 5 puis 4 qui n'ont pas de voisins non-traités. L'ordre renvoyé est donc $\sigma = (0, 1, 3, 2, 5, 4)$.

52. Appliquer l'algorithme LexBFS sur le graphe ci-dessous.



53. Appliquer l'algorithme glouton de coloration en suivant l'ordre obtenu avec LexBFS. Faire de même avec l'algorithme de parcours en largeur classique où les voisins sont parcourus par indices croissants.

4.2 Application à la coloration de graphes d'intervalles

Dans cette section, nous montrons que l'ordre de parcours obtenu avec LexBFS pour un graphe d'intervalles permet toujours à l'algorithme glouton de renvoyer une coloration optimale de ce graphe.

On dit qu'un ordre de parcours σ est *simplicial* si pour chaque sommet, ses voisins qui le précèdent forment une clique.

54. Prouver que l'ordre renvoyé par LexBFS lorsqu'il est appliqué à un graphe d'intervalles est simplicial.
55. En déduire que l'algorithme glouton de coloration appliqué avec l'ordre renvoyé par LexBFS est optimal sur les graphes d'intervalles.

4.3 Implémentation

56. Implémenter l'algorithme LexBFS en détaillant les choix de modélisation, de structures de données, les complexités des sous-fonctions et de la fonction principale.