Devoir en temps limité n°6 – 4h

Calculatrices interdites

On veillera à présenter très clairement sa copie : il faut rédiger les réponses et encadrer les résultats. Pour le code, il doit être indenté, on ne commence pas une fonction en bas de page et on utilise de la couleur pour les commentaires.

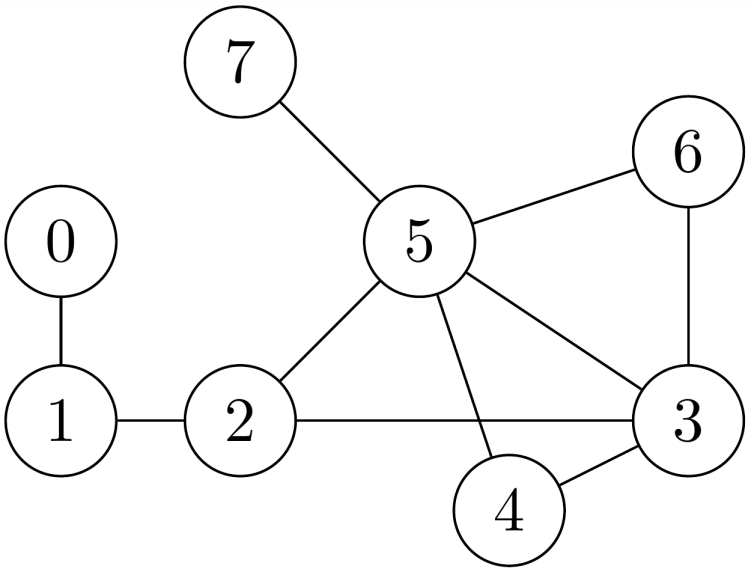
Le code doit être commenté dès qu'il dépasse les 5 lignes.

Les fonctions en C et en Ocaml doivent avoir le type précisé. Il est donc recommandé d'utiliser des fonctions auxiliaires en Ocaml.

1 Graphes

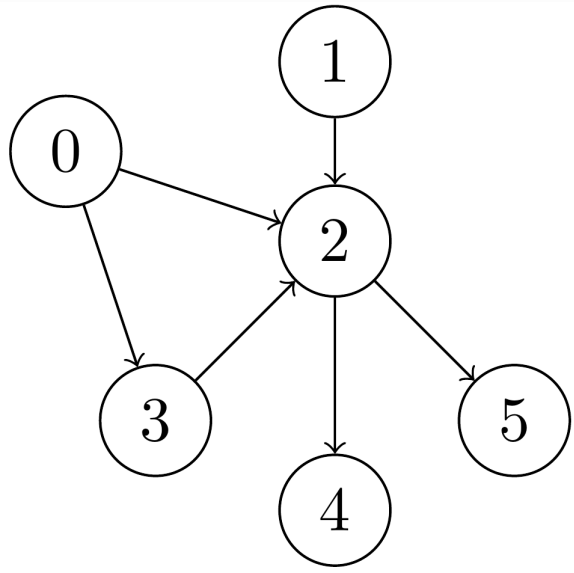
1. Questions de cours

Voici un graphe G_0 :



1. Donner la matrice d'adjacence du graphe G_0.
2. Donner les degrés des sommets dans G_0.
3. Donner un chemin de 0 à 6 dans G_0.
4. Donner un cycle dans G_0.
5. Qu'est-ce qu'un graphe connexe ? Le graphe G_0 est-il connexe ?

Voici un graphe G_1 :



6. Donner la liste d'adjacence du graphe G_1.
7. Le graphe G_1 est-il acyclique ? Pas de justification nécessaire.

8. Le graphe G_1 est-il fortement connexe ? Donner ses composantes fortement connexes.
9. Donner un tri topologique sur les sommets de G_1.

2. Manipulation en C

Dans cette section, on va, sauf indication contraire, représenter les graphes par leur liste d'adjacence en C. Comme on ne dispose pas d'une structure de listes, on utilisera un tableau de tableaux.

Un graphe est donc de type `int** g`. On rappelle que le nombre de voisins du noeud i est mis dans la case 0 du sous-tableau $g[i]$.

Dans la suite, n représente toujours le nombre de noeuds du graphe.

10. Dans cette question, on suppose que le graphe est non orienté. Écrire une fonction `int degre(int** g, int n, int i)` qui calcule le degré du sommet i.
11. Dans cette question, on suppose que le graphe est non orienté. Écrire une fonction `int* degres(int** g, int n)` qui calcule le tableau des degrés des sommets du graphe.
12. Écrire une fonction `bool sont_connectes(int** g, int i, int j)` qui renvoie `true` ssi soit l'arc (i, j) existe, soit l'arête $\{i, j\}$ existe. On n'a pas besoin de savoir si le graphe est orienté ou non pour répondre.
13. Écrire une fonction `int** matrice_to_liste(int** g, int n)` qui transforme la matrice d'adjacence du graphe en la liste d'adjacence.
    Il est interdit de réutiliser cette fonction dans la suite.
14. Écrire une fonction `int** liste_to_matrice(int** g, int n)` qui transforme la liste d'adjacence du graphe en la matrice d'adjacence.
    Il est interdit de réutiliser cette fonction dans la suite.
15. Dans cette question on suppose le graphe orienté. Écrire une fonction `int** arcs_croissants(int** g, int n)` qui prend en entrée la liste d'adjacence de G et renvoie la liste d'adjacence de G' qui est G auquel on retire tous les arcs de la forme (i, j) avec $i > j$.
    On supposera écrite une fonction `int oracle(int i)` qui permet de donner le nombre de successeurs du sommet i dans G'.

3. Coloriage

Soit $G = (S, A)$ un graphe non-orienté. Soit $n = |S|$, on supposera que les sommets sont numérotés de 0 à $n - 1$.

Un k-coloriage est un coloriage utilisant au plus k couleurs. On rappelle qu'un coloriage est l'affectation d'une couleur (un nombre) à chaque sommet de telle manière que les extrémités des arêtes sont de couleurs différentes.

Le **nombre chromatique** d'un graphe est $\chi(G) = \min\{k \in \mathbb{N} | G \text{ est k-coloriable}\}$

Un graphe est dit **complet** si toutes les arêtes possibles existent (sauf les arêtes de la forme $\{x, x\}$). On note K_p le graphe complet à $p > 0$ sommets.

Une **clique** de G est un sous-graphe complet de G.

Enfin on note $\omega(G) = \max\{p \in \mathbb{N} | K_p \text{ est une clique de } G\}$

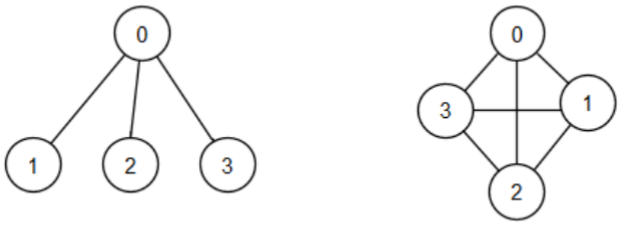


Figure 1 - de gauche à doite, le graphe G_2 et le graphe K_4

16. Le graphe G_2 de la figure 1 est-il 2 coloriable ? Justifier votre réponse.
17. Pour un entier naturel $n \geq 1$, déterminer le nombre chromatique du graphe K_n
18. Montrer que pour tout graphe G à n sommets, on a $\omega(G) \leq \chi(G) \leq n$.

4. Algorithme de coloriage

On veut maintenant écrire un algorithme pour colorier le graphe.

On se place en Ocaml. Les graphes seront représentés par le type suivant (c'est la liste d'adjacence) :

```
type graphe = int list array;;
```

On suppose écrite une fonction `degres : graphe -> (int*int) list` qui calcule la liste (degré du sommet, numéro du sommet).

19. Écrire une fonction `tri : (int*int) list -> (int*int) list` adaptée pour trier une liste de couples de manière décroissante.
    Dans cette question on pourra écrire autant de fonctions qu'on veut.

On considère ci-dessous, l'algorithme de coloriage de Welsh-Powel.

Algorithme 1 : Welsh-Powel (coloration de graphe)

Entrée : un graphe `G` à `n` sommets
Sortie tableau d'entiers contenant en position `i` la couleur du sommet numéro `i`

```
Début
    Ordonner les sommets selon les degrés décroissants dans une liste li ;
    colorie : tableau initialisé à -1 qui à terme, associera à chaque i, la couleur du sommet i ;
    Tant qu' il reste des sommets à colorier faire
        Chercher dans li le premier sommet non colorié et le colorier avec la plus petite couleur c non utilisée ;
        Colorier avec cette même couleur, en respectant leur ordre dans li, tous les sommets non coloriés et non adjacents à des sommets de couleur c ;
    Fin
    Retourner colorie
Fin
```

20. Que contient **`colorie`** si on déroule l'algorithme de coloriage ci-dessus avec le graphe G_2 de la figure 1 en entrée ?
21. Écrire une fonction `adjacent : graphe -> int array -> int -> int -> bool` qui prend en entrée le graphe, le tableau des couleurs déjà attribuées, un numéro de sommet s et un numéro de couleur c et qui renvoie **`true`** si le sommet s est adjacent à un sommet de couleur c et **`false`** sinon.
22. Implémenter l'algorithme de Welsh-Powel.

2 Arbres binaires de recherche généralisés

Dans cette partie, on étudie les B-arbres. Il s'agit d'une généralisation des arbres binaires de recherche (ABR) équilibrés à des arbres non-binaires.

Les B-arbres sont donc des arbres d'arité quelconque dont les noeuds contiennent un **ensemble de clefs**.

La définition formelle est la suivante :

Un B-arbre d'ordre t est un arbre qui vérifie toutes les propriétés suivantes :

- (S1) Tous les chemins de la racine à une feuille ont la même longueur (le même nombre de nœuds), appelée hauteur de l'arbre et dénotée par h.
- (S2) La racine est une feuille ou bien a au moins 2 enfants.
- (S3) Chaque nœud qui n'est ni racine ni feuille possède au moins $t+1$ enfants.
- (S4) Chaque nœud a au plus $2t+1$ enfants.
- (C1) Les nœuds contiennent des clefs : la racine contient de 1 à $2t$ clefs ; et les autres nœuds, entre t et $2t$ clefs.
- (C2) Dans chaque nœud, les clefs sont stockées par ordre croissant.
- (C3) Un nœud qui contient ℓ clefs (et qui n'est pas une feuille) a $\ell+1$ enfants.
- (C4) Considérons un nœud P contenant ℓ clefs $x_0, \dots, x_{\ell-1}$. Soient $P_0, \dots, P_\ell$ ses enfants, et $K(P_i)_{0\leq i\leq \ell}$ l'ensemble des clefs du sous-arbre dont la racine est P_i . Alors :
  - $\forall y \in K(P_0), y \leq x_0$

- $\forall 1 \leq i \leq \ell - 1, \forall y \in K(P_i), x_{i-1} \leq y \leq x_i$
- $\forall y \in K(P_\ell), x_{\ell-1} \leq y$

Les propriétés sont de deux ordres : quatre propriétés structurelles, préfixées par « S », et des propriétés de contenu/de clefs, préfixées par « C ». Remarquons que la propriété C4 est une généralisation de la propriété fondamentale des arbres binaires de recherche.

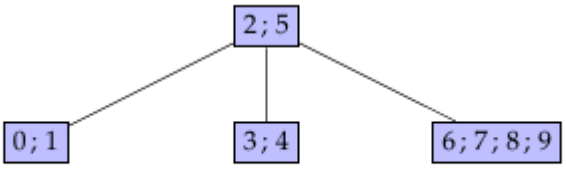
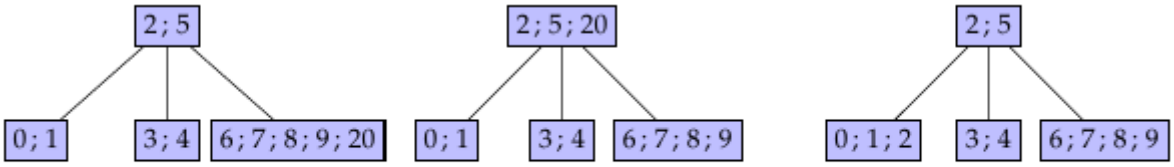


FIGURE 2 – Un exemple de B-tree d'ordre 2. Les nœuds contiennent un ensemble de clefs, triées par ordre croissant.

23. Les arbres de la Figure 3 sont-ils des B-arbres d'ordre 2 ?



(a) Arbre 1 (b) Arbre 2 (c) Arbre 3

FIGURE 3 – Quelques arbres.

On va représenter les B-arbres en Ocaml par le type suivant.

```
type bnoeud = {ordre : int; enfants : bnoeud array; clefs : int array};;
type btree = {racine : bnoeud; ordre : int};;
```

On va maintenant s'intéresser à la recherche dans un B-arbre.

La recherche d'une clef dans un B-arbre va suivre le même motif que pour un arbre binaire de recherche mais elle va combiner deux algorithmes : la recherche dans une liste triée de clés et la recherche parmi les enfants d'un nœud donné.

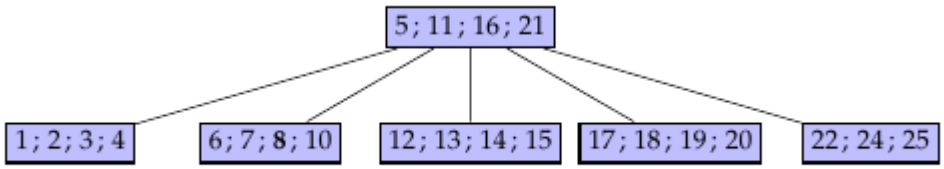


FIGURE 4 – Exemple de B-tree (d'ordre 2) pour la recherche

24. Dans le B-arbre de la figure 4, expliquer les étapes de recherche de la clef dont la valeur est 8. Il faut utiliser les caractéristiques du B-arbre pour ne pas réaliser un parcours entier de l'arbre.
25. En déduire, pour un nœud fixé, une condition d'arrêt de la recherche dans ce nœud ainsi que le numéro du sous-arbre dans lequel rechercher récursivement.
26. Écrire une fonction `recherche_cle : btree -> int -> bool` qui recherche une clé dans un B-arbre.

Étudions maintenant la complexité algorithmique de la recherche. Considérons des B-arbres non vides, d'ordre $t > 1$, de hauteur $h \geq 1$. Dans un premier temps, nous allons compter le nombre de nœuds minimum N_{min} et maximum N_{max} d'un tel B-arbre. Il est recommandé de procéder en comptant les nœuds par niveau.

27. Montrer que $N_{min} = 1 + \frac{2}{t}((t+1)^{h-1} - 1)$ pour $h \geq 2$.
28. Montrer que $N_{max} = \frac{1}{2t}((2t+1)^h - 1)$ pour $h \geq 1$.
29. En déduire un encadrement du nombre de clefs contenues dans un B-tree d'ordre t et de hauteur h.
30. Expliquer rapidement comment optimiser le coût de la recherche dans un nœud. Quelle est la complexité de cet algorithme ?
31. Conclure sur la complexité algorithmique de la recherche dans un B-arbre en fonction du nombre de nœuds n.

3 Logique

Dans ce problème, on considère des formules propositionelles qui s'écrivent avec des variables dont l'ensemble est noté X et les constructeurs $\neg$, $\wedge$, $\vee$ uniquement. Les symboles V et F désignent les deux valeurs booléennes

Dans la suite, on s'intéresse au problème Horn-Sat, qui peut être décrit de la façon suivante : étant donnée une formule de Horn P, existe-t-il une valuation v telle que $[|P|]_v = V$?

L'objectif est d'expliciter un algorithme permettant de résoudre ce problème.

On commence par introduire le vocabulaire nécessaire.

Soit x une variable propositionnelle. Un **littéral** est la formule x ou la formule $\neg x$. Le littéral x (respectivement $\neg x$) est un littéral **positif** (respectivement **négatif**).

32. Définir ce qu'est une clause (sous-entendu disjonctive).

On rappelle qu'une **forme normale conjonctive** (FNC) est une formule s'écrivant comme une conjonction de clauses.

Dans la suite on considèrera qu'il existe une clause vide, notée $()$. On considère aussi qu'il existe une FNC vide, notée $\emptyset$.

Par convention, l'évaluation de $()$ est toujours fausse et l'évaluation de $\emptyset$ est toujours vraie, peu importe la valuation.

Une **clause de Horn** est une clause contenant au plus un littéral positif.

Une **clause unitaire** est une clause composée d'un unique littéral.

Soit P une formule propositionnelle. On dit que P est une **formule de Horn** s'il existe $n \in \mathbb{N}$ et $C_1, C_2, \dots, C_n$ des clauses de Horn vérifiant $P = C_1 \wedge C_2 \wedge \cdots \wedge C_n$.

33. Pour chacune des formules suivantes, montrer qu'elles sont satisfiables ou non satisfiables.
    (a) $P_1 = (x \vee y) \wedge (\neg x \vee \neg y) \wedge (x \vee \neg y)$
    (b) $P_2 = (x) \wedge (\neg x \vee \neg y) \wedge (\neg y \vee z) \wedge (z)$
    (c) $P_3 = ()$
    (d) $P_4 = (x \vee \neg y \vee \neg t) \wedge (z \vee \neg t \vee \neg x \vee \neg y)$
34. Parmi les formules de la question précédente, lesquelles sont des formules de Horn ? Il n'est pas nécessaire de justifier.

Soit P une forme normale conjonctive contenant une clause unitaire (l) avec l un littéral. On construit une formule P' à partir de P de la façon suivante :

- supprimer de P toutes les clauses où l apparait
- enlever des autres clauses toutes les occurences du littéral opposé à l. Par exemple si $l = \neg x$, on retirera toutes les occurences du littéral x.

Cette procédure de simplification est appelée **propagation unitaire**.

Par exemple, si $P = (x \vee \neg y \vee z) \wedge (\neg x \vee y \vee z) \wedge (x)$, on a alors $P' = (y \vee z)$. Si $P = (x) \wedge (\neg x)$, on a alors $P' = ()$.

Dans la suite, on note $\Pi(P)$ une formule sans clause unitaire obtenue en itérant la propagation unitaire sur P tant qu'il reste des clauses unitaires.

On admet que si $\Pi(P)$ ne contient pas de clause vide $()$, alors $\Pi(P)$ est unique et que si $\Pi(P)$ contient au moins une clause vide alors toute formule sans clause unitaire obtenue par itération de la propagation unitaire sur P contient au moins une clause vide.

35. On pose

    $$P = (x_1) \wedge (x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_2) \wedge (\neg x_2 \vee \neg x_1 \vee x_3) \wedge (x_3 \vee \neg x_4 \vee x_5) \wedge (\neg x_1 \vee x_2 \vee x_5)$$

    Calculer $\Pi(P)$. On montrera les deux premières étapes de propagation et on elipsera la suite.
36. Soit P une formule de Horn. Montrer que $\Pi(P)$ est une formule de Horn.
37. Soit P une FNC. Montrer que P est satisfiable si et seulement si $\Pi(P)$ est satisfiable.
38. Soit P une forme normale conjonctive. Montrer que si $()$ apparaît dans $\Pi(P)$, alors P n'est pas satisfiable.
39. Montrer qu'il existe une valuation qui satisfait toutes les clauses de Horn qui ne sont pas la clause vide, ni une clause unitaire positive.
40. Soit P une formule de Horn ne contenant ni de clause vide ni de clause unitaire positive. Montrer que P est satisfiable.
41. Soit P une formule de Horn. Montrer que P n'est pas satisfiable si et seulement si $()$ apparaît dans $\Pi(P)$.