Devoir en temps limité n°6 – 4h

Calculatrices interdites

1 Graphes

1. Questions de cours

1. *Donner la matrice d'adjacence du graphe G_0.*

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

2. *Donner les degrés des sommets dans G_0.*

sommet	degré
0	1
1	2
2	3
3	4
4	2
5	5
6	2
7	1

3. *Donner un chemin de 0 à 6 dans G_0.*
   0, 1, 2, 3, 6
4. *Donner un cycle dans G_0.*
   3, 4, 5 est un cycle
5. *Qu'est-ce qu'un graphe connexe ? Le graphe G_0 est-il connexe ?*
   Un graphe est connexe si pour tous sommets x et y il existe un chemin de x à y.
6. *Donner la liste d'adjacence du graphe G_1.*
   [[2,3], [2], [4,5], [2], [], []]
7. *Le graphe G_1 est-il acyclique ? Pas de justification nécessaire.*
   Oui il est acyclique
8. *Le graphe G_1 est-il fortement connexe ? Donner ses composantes fortement connexes.*
   Non il n'est pas fortement connexe. Chaque sommet est sa propre composantes (car il n'y a pas de cycles).
9. *Donner un tri topologique sur les sommets de G_1.*
   0,1,3,2,4,5 fonctionne.

2. Manipulation en C

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.*

```c
int degre(int** g, int n, int i){
  return g[i][0];
}
```

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.*

```
int* degres(int** g, int n){
  int* res = malloc(n*sizeof(int));
  for(int i=0; i<n; i+=1){
    res[i] = degre(g,n);
  }
  return res;
}
```

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.*

```
bool sont_connectes(int** g, int i, int j){
  for(int k=0; k<g[i][0]; k+=1){
    if (g[i][k]==j){
      return true;
    }
  }
  return false;
}
```

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.*

```
int** matrice_to_liste(int** g, int n){
  int** liste = malloc(n*sizeof(int*));

  for(int i=0; i<n; i+=1){
    int deg = 0;
    for (int j=0; j<n; j+=1){
      if (g[i][j]==1){
        deg+=1;
      }
    }
    liste[i] = malloc((deg+1)*sizeof(int));
    liste[i][0] = deg;

    int k=1;
    for (int j=0; j<n; j+=1){
      if (g[i][j]==1){
        liste[i][k] = j;
        k+=1;
      }
    }
    return liste;
  }
}
```

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.*

```
int** liste_to_matrice(int** g, int n){
  int** matrice = malloc(n*sizeof(int*));
  for (int i=0; i<n; i+=1){
    matrice[i] = malloc(n*sizeof(int))
    for (int j=0; j<n; j+=1){
      matrice[i][j] = 0;
    }
    for (int j=0; j<g[i][0]; j+=1){
      matrice[i][g[i][j]] = 1;
    }
  }
  return matrice;
}
```

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* $i > j$.

```
int** arcs_croissants(int** g, int n){
  int** liste = malloc(n*sizeof(int*));
```

```
    for(int i=0; i<n; i+=1){
      liste[i] = malloc(oracle(i)*sizeof(int));
      int k=0;
      for(int j=0; j<g[i][0]; j+=1){
        if (g[i][j]<i){
          liste[i][k] = j;
          k+=1;
        }
      }
    }
    return liste;
  }
```

3. Coloriage

16. *Le graphe G_2 de la figure 1 est-il 2 coloriable ? Justifier votre réponse.*
Oui, G_2 est 2-coloriable. En effet on peut donner la couleur 0 à 0 et la couleur 1 aux autres.

17. *Pour un entier naturel $n \geq 1$, déterminer le nombre chromatique du graphe K_n*
Dans un graphe complet, on ne peut pas donner la même couleur à deux sommets différents puisqu'ils sont toujours reliés. Donc il faut n couleurs.

18. *Montrer que pour tout graphe G à n sommets, on a $\omega(G) \leq \chi(G) \leq n$.*
Le nombre maximal de couleurs qu'on peut donner est n. Reste à montrer $\omega(G) \leq \chi(G)$.
Le graphe G contient une clique de taille $\omega(G)$. Cette clique est un graphe complet de taille $\omega(G)$ et ne peut pas être colorié avec moins de $\omega(G)$ couleurs. Or un coloriage de G implique un coloriage de la clique. Donc G ne peut pas être colorié avec moins de $\omega(G)$ couleurs.

4. Algorithme de coloriage

19. *Écrire une fonction `tri : (int*int) list -> (int*int) list` adaptée pour trier une liste de couples de manière décroissante.*
On fait un tri fusion.

20. *Que contient **`colorie`** si on déroule l'algorithme de coloriage ci-dessus avec le graphe G_2 de la figure 1 en entrée ?*
[0,0,0,1]

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.*

```
let adjacent g couleurs s c =
   let rec aux l = match l with
     |[] -> false
     |t::q -> (couleurs.(t)=c) || aux q
   in
   aux g.(s);;
```

22. *Implémenter l'algorithme de Welsh-Powel.*

```
let welsh-powel g =
 (*Pour réduire un peu le coût de parcours de la liste, on éliminera son début au fur et à
mesure*)
  let li = ref (tri (degres g)) in
  let n = Array.length g in
  let colorie = Array.make n (-1) in
  let nb_colorie = ref 0 in
  let c = ref 0 in (* plus petite couleur non utilisée *)
  let rec aux l = match l with (*Renvoie le plus petit sommet non colorié*)
    |[] -> ([],-1)
    |(dt,t)::q -> if colorie.(t) = -1 then (q,t)
                  else aux q
  in
  let rec aux2 l = match l with (*Effectue la 2ème partie de la boucle*)
    |[] -> ()
    |(dt,t)::q -> if colorie.(t)=-1 && not (adjacent g colorie t !c) then begin
                        colorie.(t) <- !c;
                        nb_colorie := !nb_colorie+1
```

```
                end
in
while !nb_colorie <> n do
  (*Récupérer le premier sommet non colorié*)
  let q,s = aux !li in
  (*exclure les éléments déjà coloriés*)
  li := q;
  (*le colorier*)
  colorie.(s) <- !c;

  (*Faire la deuxième phrase*)
  aux2 !li;
  (*On peut passer à la prochaine couleur, celle-ci ne changera plus*)
  c := !c+1
done;
colorie;;
```

2 Logique

23. *Définir ce qu'est une clause (sous-entendu disjonctive).*
C'est une disjonction de littéraux, une formule de la forme $ł_1 \vee l_1 \vee \cdots \vee l_m$.

24. *Pour chacune des formules suivantes, montrer qu'elles sont satisfiables ou non satisfiables.*
    - P_1 est satisfiable puisque pour la valuation v définie par $v(x) = \mathrm{V}$ et $v(y) = \mathrm{F}$, on a $[|P_1|]_v = \mathrm{V}$.
    - P_2 est satisfiable puisque pour la valuation v définie par $v(x) = \mathrm{V}$, $v(y) = \mathrm{F}$ et $v(z) = \mathrm{V}$, on a $[|P_2|]_v = \mathrm{V}$.
    - P_3 n'est pas satisfiable puisque pour toute valuation v on a $[|()|]_v = \mathrm{F}$.
    - P_4 est satisfiable puisque pour une valuation v définie par $v(x) = \mathrm{V}$, $v(y) = \mathrm{F}$, $v(z)$ et $v(t)$ quelconques, on $[|P_4|]_v = \mathrm{V}$.

25. *Parmi les formules de la question précédente, lesquelles sont des formules de Horn ? Il n'est pas nécessaire de justifier.*
Seule P_1 n'est pas une formule de Horn.

26. *Calculer* $\Pi(P)$. *On montrera les deux premières étapes de propagation et on elipsera la suite.*
(je fais le calcul en entier pour bien montrer le résulat attendu). À partir de P par propagation unitaire sur x_1 on obtient :
$$(x_3 \vee x_4) \wedge (\neg x_2) \wedge (\neg x_2 \vee x_3) \wedge (x_3 \vee \neg x_4 \vee x_5) \wedge (x_2 \vee x_5)$$
Par propagation unitaire sur $\neg x_2$ on obtient ensuite :
$$(x_3 \vee x_4) \wedge (x_3 \vee \neg x_4 \vee x_5) \wedge (x_5)$$
Par propagation unitaire sur x_5 on obtient ensuite le résultat final :
$$(x_3 \vee x_4)$$

27. *Soit P une formule de Horn. Montrer que* $\Pi(P)$ *est une formule de Horn.*
On montre qu'un invariant de l'algorithme de construction de $\Pi(P)$ est que les formules obtenues à chaque étape sont des formules de Horn.
La formule originale est une formule de Horn.
Soit $P' = C'_1 \wedge C'_2 \wedge \cdots \wedge C'_n$ une formule de Horn obtenue après une certaine étape qui n'est pas la dernière et P'' la formule obtenue après une nouvelle étape de propagation.
La propagation unitaire supprime des clauses, ou des littéraux dans les clauses. On suppose que les clauses supprimées sont C'_{m+1} à C'_n et on note C''_1 à C''_m les clauses restantes modifiées.
Pour $i \in [|1, m|]$, C'_i est une clause de Horn, donc elle possède au plus un littéral positif. C''_i est C'_i avec des littéraux en moins. Elle contient elle aussi au plus un littéral positif. Donc C''_i est une clause de Horn.
On obtient donc $P'' = C''_1 \wedge C''_2 \wedge \cdots \wedge C''_m$ conjonction de clauses de Horn. P'' est une formule de Horn.
Notre propriété est bien un invariant.
À la fin du processus, qui termine car le nombre de clauses diminue strictement à chaque étape, la formule obtenue $\Pi(P)$ est une formule de Horn.

28. *Soit P une FNC. Montrer que P est satisfiable si et seulement si $\Pi(P)$ est satisfiable.*
Supposons P satisfiable. Soit v une valuation qui satisfait P, alors elle satisfait toute clause de P.
On considère P' obtenu par une étape de propagation unitaire sur une clause unitaire l. Soit C' une clause de P', elle correspond à une unique clause C de P.
Si $C' = C$, alors v satisfait C'.
Sinon, on sépare le cas où l est positif ou négatif :

- Si c'est un littéral positif x, alors $C = C' \vee \neg x$. Or comme (x) apparait comme clause dans P, $v(x)$ est vrai.
  Donc $V = [|C|]_v = [|C'|]_v$ OU $[|\neg x|]_v = [|C'|]_v$. C' est satisfaite par v.
- Sinon c'est un littéral négatif $\neg x$, alors $C = C' \vee x$. Or comme $(\neg x)$ apparait comme clause dans P, $v(x)$ est faux.
  Donc $V = [|C|]_v = [|C'|]$ OU $[|x|]_v = [|C|]_v$. C' est satisfaite par v.

On conclut que v satisfait toutes les clauses de P', donc v satisfait P'.
En répétant ce raisonnement autant de fois qu'on effectue la propagation unitaire, on obtient que v satisfait $\Pi(P)$.

Réciproquement, supposons $\Pi(P)$ satisfiable et soit v un valuation qui le satisfait.
On considère la formule P' le prédécesseur de $\Pi(P)$ dans l'algorithme de propagation unitaire itéré. Si le littéral retiré est x pour x une variable, on définit v' la valuation qui pour y dans le domaine de v associe $v(y)$ et à x associe V. Si le littéral retiré est $\neg x$, on fait la même chose avec $v'(x) = F$
Ici il est important de comprendre que la variable x n'existant pas dans $\Pi(P)$, il faut lui affecter une valeur pour obtenir une valuation de P'
Soit C' une clause de P', on va montrer que v' la satisfait :

- soit C' disparait car elle contient l. Dans ce cas I' la satisfait puisque $[|l|]_{v'} = V$.
- Soit C' ne contient ni l ni son contraire et est une clause de $\Pi(P)$, qui est satisfiable par v donc par v'.
- Soit C' contient l'inverse de l et donne dans $\Pi(P)$ une clause C telle que $C' = C \vee \neg l$. Or $[|C'|]_{v'} = [|C|]_{v'}$ OU $[|\neg l|]_{v'} = [|C|]_v$ OU $F = [|C|]_v = V$. Donc C' est satisfaite par v'.

On conclut que I' satisfait toutes les clauses de P', donc I' satisfait P'. En répétant ce raisonnement autant de fois qu'on effectue la propagation unitaire, on construit une valuation satisfaisant $\Pi(P)$.

29. *Soit P une forme normale conjonctive. Montrer que si $()$ apparaît dans $\Pi(P)$, alors P n'est pas satisfiable.*
Si () apparait dans $\Pi(P)$, $\Pi(P)$ est de la forme $() \wedge C_2 \wedge \cdots \wedge C_n$.
Soit v une valuation, $[|\Pi(P)|]_v = [|()|]_v$ ET $[|C_2 \wedge \cdots \wedge C_n|]_v = \text{F}$ ET $[|C_2 \wedge \cdots \wedge C_n|]_v = \text{F}$.
Donc $\Pi(P)$ est non satisfiable. D'après la question précédente P n'est pas satisfiable.

30. *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.*
On considère la valuation v_0 qui à toute variable associe F.
Considérons une clause de Horn qui n'est pas vide et pas unitaire et positive. Alors cette clausse contient un littéral négatif.
Elle est donc satisfaite par v_0, puisqu'un de ses littéraux est satisfait.

31. *Soit P une formule de Horn ne contenant ni de clause vide ni de clause unitaire positive. Montrer que P est satisfiable.*
Si P est une formule de Horn ne contenant ni clause vide, ni clause unitaire positive, alors il est de la forme : $P = C_1 \wedge C_2 \wedge \cdots \wedge C_m$ avec pour tout i, C_i qui est une clause de Horn qui n'est ni la clause vide, ni une clause unitaire positive.
D'après la question précédente, l'interprétation qui à toute variable associe F satisfie C_i pour tout i, donc satisfait P.

32. *Soit P une formule de Horn. Montrer que P n'est pas satisfiable si et seulement si $()$ apparaît dans $\Pi(P)$.*
Si () apparait dans $\Pi(P)$ alors P n'est pas satisfiable d'après Q30.
Pour la réciproque, on raisonne par contraposée. On suppose que () n'apparait pas dans $\Pi(P)$.
$\Pi(P)$ est une formule de Horn puisque P en est une (Q28). De plus par définition $\Pi(P)$ ne contient pas de clause unitaire.
On en déduit que $\Pi(P)$ est une formule de Horn sans clause vide ni clause unitaire positive. D'après la question précédente, $\Pi(P)$ est satisfiable et donc d'après Q29, P également.

3 Arbres binaires de recherche généralisés

33. *Les arbres de la Figure 3 sont-ils des B-arbres d'ordre 2 ?*
L'arbre 1 n'est pas un B-arbre d'ordre 2 car le fils le plus à droite contient 5 clés et la limite est 4.
L'arbre 2 n'est pas un arbre car la racine contient 3 clés et a 3 fils.
L'arbre 3 est bien un B-arbre. (les inégalités sont larges dans les propriétés)

34. *Dans le B-arbre de la figure 4, expliquer les étapes de recherche de la clef dont la valeur est 8.*
On parcourt la liste des clés. On trouve que le 8 s'insère entre le 5 et le 11 (et n'est pas présent dans la liste). Notre clé ne peut donc se trouver que dans le deuxième fils.
On se déplace donc dans le deuxième fils grâce au tableau des enfants. On parcourt la nouvelle liste des clés et là on trouve notre clé. Si on avait pas trouvé, on saurait qu'elle n'est pas présente.

35. *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.*
Si le noeud actuel est une feuille et qu'on rencontre une clé strictement supérieure à celle cherchée, on peut s'arrêter. Dans n'importe quel noeud, si on trouve la clé dans la liste, on peut s'arrêter.
Si on est pas sur une feuille, on cherche i tel que $x_{i-1} < k < x_i$ (on a pas égalité car sinon on s'arrête). Il faut alors aller chercher la clé dans le fils de numéro i. Si on a $k < x_0$ il faut aller dans le fils 0 et si $k > x_{\ell-1}$, il faut aller chercher dans le fils ℓ.

36. *Écrire une fonction* `recherche_cle : btree -> int -> bool` *qui recherche une clé dans un B-arbre.*

```
let rec recherche_cle ba k =
  (* Chercher l'indice *)
  let i = ref 0 in
  while ba.clefs.(!i)<k do
    i:=!i+1
  done;
  if ba.clefs.(!i)=k then true
  else if Array.length enfants = 0 then false (* si c'est uen feuille*)
  else recherche_cle ba.enfants[i] k;;
```

37. *Montrer que* $N_{min} = 1 + \frac{2}{t}((t+1)^h - 1)$ *pour* $h \geq 1$.
Montrons par récurrence qu'au niveau $i \geq 1$, le nombre minimum de noeuds est $2(t+1)^{i-1}$. Au niveau 0 on a 1 noeud.
Au niveau 1 on a au minimum deux noeuds.
Supposons la formule vraie au niveau i. Chacun des noeuds a au minimum $t+1$ enfants. Cela fait donc au total $2(t+1)^{i-1+1}$.
Par principe de récurrence, notre formule est vraie.
Pour finir, on doit sommer les sommets sur tous les niveaux. Cela fait $1 + \sum_{i=1}^{h} 2(t+1)^{i-1} = 1 + 2\sum_{i=0}^{h-1}(t+1)^i =$
$1 + 2 * \frac{(t+1)^h - 1}{t+1-1} = 1 + \frac{2}{t}((t+1)^h - 1)$.

38. *Montrer que* $N_{max} = \frac{1}{2t}((2t+1)^{h+1} - 1)$ *pour* $h \geq 0$.
Montrons par récurrence qu'au niveau i, le nombre maximum de noeuds est $(2t+1)^i$.
Au niveau 0 on a 1 noeud.
Supposons la formule vraie au niveau i. Chacun des noeuds a au meximum $2t+1$ enfants. Cela fait donc au total $(2t+1)^{i+1}$.
Par principe de récurrence, notre formule est vraie.
Pour finir, on doit sommer les sommets sur tous les niveaux. Cela fait $\sum_{i=0}^{h}(2t+1)^i = \frac{(2t+1)^{h+1}-1}{2t+1-1} = \frac{1}{2t}((2t+$
$1)^{h+1} - 1)$.

39. *En déduire un encadrement du nombre de clefs contenues dans un B-tree d'ordre t et de hauteur h.*
Au minimum la racine contient 1 clé et les autres noeuds t clés.
En combinant avec N_{min}, on obtient que le B-arbre contient au minimum $1 + 2((t+1)^h - 1)$ clés.
Au maximum un noeud contient $2t$ clés.
En combinant avec N_{max}, on obtient que le B-arbre contient au maximum $(2t+1)^{h+1} - 1$ clés.

40. *Expliquer rapidement comment optimiser le coût de la recherche dans un nœud. Quelle est la complexité de cet algorithme ?*

    Plutôt que de faire une bête recherche linéaire, on peut utiliser le fait que le tableau des clés est trié dans l'ordre croissant. On peut donc effectuer une recherche dichotomique, de complexité logarithmique.

41. *Conclure sur la complexité algorithmique de la recherche dans un B-arbre en fonction du nombre de nœuds n.*

    Notre parcours parcourt dans le pire cas un chemin de la racine à une feuille h. Pour chaque noeud, on effectue une recherche dichotomique qui est logarithmique en t.

    Cela fait donc une complexité en $O(h * log(t))$.

    Étant donné que chaque noeud a $t+1$ enfants, on peut voir que n est très grand devant t.

    En reprenant N_{max}, on peut estimer h par rapport à n dans le pire cas.

$$\frac{1}{2t}((2t+1)^{h+1}-1) = n$$

$$2tn + 1 = (2t+1)^{h+1}$$

$$\log(2tn+1) = (h+1)\log(2t+1)$$

$$\log(2tn+1-2t-1) = h+1$$

$$\log(2t(n-1)) = h+1$$

$$\log(2t) + \log(n-1) = h+1$$

    En conclusion : $h = O(log(n))$ et la complexité de notre fonction est en $O(log(n)^2)$ (estimation très large, on aurait probablement pu estimer t comme un logarithme de n aussi.