1 Coloration de graphes

1. Le graphe G_1 est 2-coloriable en attribuant au sommet 0 la couleur 0 et aux autres sommets la couleur 1.

2. Pour tout graphe à n sommets, une coloration attribuant la couleur i au sommet i est valide; ainsi tout graphe est n-coloriable et $\chi(K_n) \leqslant n$. De plus, deux sommets de K_n étant toujours voisins, une couleur ne peut être attribuée à deux sommets, donc $\chi(K_n) = n$.

3. On a montré ci-dessus que pour tout graphe G à n sommets, on a $\chi(G) \leqslant n$. De plus, si G' est un sous-graphe de G, alors $\chi(G') \leqslant \chi(G)$. Or $K_{\omega(G)}$ est un sous-graphe de G, donc, par la question précédente, $\omega(G) \leqslant \chi(G)$.

4. `d1 = {0: [1,2,3], 1: [0], 2: [0], 3: [0]}`.

5.
```
def degres_sommets(g) :
    return [(len(d[i]), i) for i in d]
```

6.
```
def tri_degres(d) :
    ds = degres_sommets(d)
    return [x[1] for x in tri(ds)]
```

7. On vérifie successivement que seules les couleurs 0 et 1 sont présentes et que la condition sur les voisins est respectée. Je considère que la coloration proposée donne une couleur à chaque sommet.
```
def test(d, c) :
  for s in c :
    if c[s] != 0 and c[s] != 1 :
      return False
  for s in d :
    for voisin in d[s] :
      if c[s] == c[voisin] :
        return False
 return True
```

8. L'énoncé est ambigu; je considère que l'algorithme renvoie un dictionnaire. On obtient `{0: 0, 1: 1, 2: 1, 3: 1}`.

9.
```
def adjacent(g, dc, s, c) :
    for voisin in g[s] :
      if voisin in dc and dc[voisin] == c :
        return True
    return False
```

10.
```
def coloriage_glouton(g) :
    li = tri_degres(g)
    colorie = {}
    couleur = 0
    while len(colorie) != len(g) :
      for s in li :
        if s not in colorie :
          break
      colorie[s] = couleur
      for x in li :
        if x not in colorie and not adjacent(g, colorie,
x, couleur) :
          colorie[x] = couleur
      couleur = couleur + 1
    return colorie
```

11. On modélise la situation par un graphe dont les sommets sont les étudiants et les arêtes les liens d'amitiés. On peut voir le problème de construction des groupes comme une coloration.

[insérer ici le graphe, flemme de le faire en TikZ]

On peut ainsi considérer les groupes $\{B, D, G\}$, $\{A, F\}$ et $\{C, E, H\}$.

2 Satisfiabilité d'une formule propositionnelle

12.
```
let c = Ou(Ou(Var 0, Var 1), Non(Var 2))
```

13.
```
let f = [c ; Ou(Non(Var 1), Var 2)]
```

14.
```
let rec evalue_clause c v =
    match c with
      | Var i -> v.(i)
      | Non(phi) -> not (evalue_clause phi v)
      | Ou(phi, psi) -> (evalue_clause phi v) || (evalue_clause psi v)
```

15. Je considère que cette fonction traite une formule et non une clause comme écrit dans l'énoncé.
```
let rec evalue_FNC f v =
  match f with
    | [] -> true
    | h :: t -> (evalue_clause h v) && (evalue_FNC t v)
```

16. La première clause est satisfaite grâce à x_1, la deuxième avec x_2 : on obtient `true`.

17. On ajoute 1 à droite et on propage la retenue vers la gauche jusqu'à trouver un zéro. Si la retenue se propage tout à gauche, il n'y a pas de suivant.

```
let suivant t =
  let i = ref (Array.length t - 1) in
  let retenue = ref t.(!i) in
  t.(!i) <- not t.(!i); (* 0 + 1 = 1 ; 1 + 1 = 0 et retenue 1 *)
  decr i;
  while !i >= 0 && !retenue do
      retenue := t.(!i);
      t.(!i) <- not t.(!i);
      decr i
  done;
  not !retenue
```

18.
```
let satisfiable f m =
  let sigma = Array.make m false in
  let res = ref (evalue_FNC f sigma) in
  while not !res && suivant sigma do
    res := evalue_FNC f sigma
  done;
  !res
```

19. On note n le nombre de littéraux de la formule et m le nombre de variables en jeu. La création de la valuation initiale coute $O(m)$. Surtout, chaque appel à `evalue` coute $O(n)$ et chaque appel à `suivant` coute $O(m)$. Il y a 2^m valuations à essayer, donc la complexité est $O((n+m)2^m)$.

20. On fixe à faux la valeur de la première variable. Récursivement, on cherche une affectation des autres variables qui rend la formule vraie. S'il n'y en a pas, on fixe à vrai la première variable et on procède récursivement de même. Si cela échoue aussi, la formule n'est pas satisfaisable.

21. On construit la formule $\bigwedge_{i=1}^{n} F_i \wedge \neg F$, on la met en FNC, et on vérifie avec l'algorithme précédent qu'elle n'est pas satisfaisable.

22. L'ensemble des règles du système $\vdash$ n'étant pas rappelé, je considère qu'il s'agit de la déduction naturelle classique. Ce système est correct : si $\Gamma \vdash \varphi$, alors $\Gamma \models \varphi$. De plus ce système est complet : si $\Gamma \models \varphi$, il existe une preuve de $\Gamma \vdash \varphi$. On peut donc se ramener de l'un à l'autre.

Dans l'exemple, P est présent des deux côtés de $\vdash$, donc il suffit d'appliquer la règle Axiome.

3 Automates et reconnaissance de motifs

La définition de période n'est pas claire. Je considère que p est une période de x si pour *tout* $i \in [\![0\,,\,|x|-p-1]\!]$, $x_i = x_{i+p}$.

23. La longueur d'un mot non vide en est une période, l'ensemble $[\![0\,,\,|x|-p-1]\!]$ étant alors vide.

24. La fonction `occ_ici` indique si `x` est une occurrence de `y` à partir de l'indice `i`.
```
let occ_ici x y i =
  let j = ref 0 in
  while !j < String.length x && x.[!j] = y.[i+!j] do
    incr j
  done;
  !j = String.length x

let occurrence x y =
  let i = ref 0 in
  let lx = String.length x and ly = String.length y in
  while !i <= ly - lx && not (occ_ici x y !i) do
    incr i
  done;
  !i <= ly - lx
```

25. La complexité de `occ_ici` est $O(|x|)$, celle de `occurrence` est donc $O(|x||y|)$.

26. 3 est une période : $(aab)(aab)(aa\varepsilon)$. 2 n'est pas une période : $(aa)(ba)$.... 1 non plus. Donc 3 est la période.

27.
```
let est_periode s p =
    let i = ref 0 in
    while !i + p < String.length s && s.[!i] = s.[!i+p] do
      incr i
    done;
    !i + p = String.length s

let periode s =
    let p = ref 1 in
    while not (est_periode s !p) do
      incr p
    done;
    !p
```

28. Supposons que p est une période de x et notons $u = x_0 \dots x_{p-1}$ et $w = x_p \dots x_{n-1}$, de sorte que $x = uw$. Comme p est une période, les premières lettres de w sont les premières lettres de x : on peut donc écrire $x = wv$. Et on a $|v| = |x| - |w| = |u| = p$.

Supposons que $x = uw = wv$ avec $|u| = |v| = p$. Soit $i \in [\![0\,,\,|x| - p - 1]\!]$: on a $x_i = u_i$ car $x = uw$, et $x_i = w_i$ car $x = wv$, mais aussi $x_{i+p} = (uw)_{i+p} = w_i$, donc $x_i = x_{i+p}$. Ainsi, p est une période.

29. Montrons le résultat sur les bords par récurrence sur $|x|$.

Pour x réduit à une lettre, le seul bord est ε. Par conséquent $n = 1$ et il n'y a rien à montrer.

Supposons la propriété établie pour tout mot de longueur strictement inférieure à $|x|$. Soit $x' = Bord(x)$. $|x'| < |x|$, donc on peut appliquer l'hypothèse de récurrence : la suite $(Bord^i(x'))$ est la suite des bords de x' dans l'ordre décroissant de longueur. Or il s'agit de la suite $(Bord^{i+1}(x))$: la suite $(Bord^i(x))$ vérifie donc la propriété voulue, à nouveau car $|x'| > |Bord(x')|$, ie. $|Bord(x)| > |Bord^2(x)|$.

Pour les périodes : on a vu que p est une période de x si et seulement s'il existe u, v, w tels que $x = uw = wv$, ie s'il existe un bord w de longueur $|x| - p$. Il en résulte la croissance de la suite des périodes $|x| - |Bord^i(x)|$.

30. *Cette question délirante est à traiter une fois que vous avez fini tout le reste (sauf si vous avez l'intuition géniale d'emblée). Il ne peut y avoir beaucoup de points dessus parce que quasiment personne ne va répondre, et encore moins de façon pertinente. Par contre si vous avez une idée à moitié finie, ça peut rapporter une partie des points...*

Quand on a un échec dans `occ_ici`, on sait quand même que les j premières lettres coïncident. Naïvement, la tentative suivante sera décalée de 1 : mais si 1 n'est pas une période de x, ça n'a aucune chance de marcher. On peut faire un décalage correspondant à la plus petite période (ou au plus grand bord) de la chaine coïncidente.

31.
```
let automate =
  {etats = [0;1;2;3;4];
   alphabet = ['a'; 'b'; 'c'];
   initial = 0;
   final = [2;4];
   transition = fun e l -> match e,l with
                | 0, 'a' -> 1
                | 0, 'b' | 0, 'c' -> 0
                | 1, 'a' -> 2
                | 1, 'b' -> 3
                etc.}
```

32. Tous les états sont accessibles (ie. depuis l'état initial) et coaccessibles (vers un état final), donc l'automate est émondé.

33. L'énoncé étant imprécis, je considère qu'il s'agit de l'algorithme qui élimine les états d'un automate généralisé pour obtenir une expression régulière dénotant le langage reconnu.

On ajoute un nouvel état initial α et un nouvel état final ω, reliés respectivement vers 0 et depuis 2 et 4 par des transitions étiquetées ε.

L'élimination de 0 crée les transitions :

- α vers 1 étiquetée $(b|c)^*a$,
- 1 vers 1 étiquetée $c(b|c)^*a$,
- 2 vers 1 étiquetée $c(b|c)^*a$,
- 3 vers 1 étiquetée $(b|c)(b|c)^*a$,
- 4 vers 1 étiquetée $c(b|c)^*a$.

34. Chaque terme dénote une occurrence du mot cherché, en indiquant où celle-ci se termine.
Dans l'exemple, on a `[3;4]`.

35. Le procédé de construction des listes en OCaml conduit à avoir `lst` en ordre décroissant. Ce n'est pas un problème car la spécification ne donne pas de contrainte d'ordre.
Il n'y a pas de blocage car l'automate est supposé complet.

```
let cherche m t =
  let e = ref m.initial in
  let lst = ref [] in
  for i = 0 to String.length t - 1 do
    let lambda = t.[i] in
    e := m.transition !e lambda;
    if List.mem !e m.final then
      lst := i :: !lst
  done;
  !lst
```

Ici, j'ai suivi l'algorithme en style impératif. Il arrive parfois à CCINP qu'il soit explicitement demandé de programmer en OCaml fonctionnel pur (bien repérer cette consigne car son non-respect entraine la note zéro). On peut alors systématiquement transformer :

- *la boucle devient une fonction récursive,*
- *les variables qui sont propagées d'un tour à l'autre sont des paramètres de cette fonction,*
- *le cas de base est le cas d'arrêt de la boucle et renvoie la ou les variables qui sont utiles dans la suite de l'algorithme,*
- *le ou les cas récursifs correspondent aux évolutions des variables d'un tour de boucle au suivant.*
- *On peut poser des variables correspondant aux valeurs initiales (avant la boucle) pour garder les choses dans le même ordre que dans l'algorithme itératif ; dans le cas contraire ces valeurs apparaissent dans l'appel initial à la fonction récursive.*
- *De même, on peut noter explicitement de nouvelles variables correspondant aux mises à jour faites peu à peu dans la boucle ou reporter cette mise à jour dans l'appel récursif.*

```
let cherche m t =
  let rec loop i e lst =
    if i = String.length t then
      lst
    else
      let lambda = t.[i] in (* pas besoin d'en faire un paramètre de loop car d'usage interne à un tour de boucle
donné *)
      let e' = m.transition e lambda in
      let lst' = if List.mem e' m.final then i :: lst else lst in
      (* on pourrait définir de même i' = i+1 mais cela n'apporte rien *)
      loop (i+1) e' lst'
  in
  loop 0 m.initial []
```

36. Par construction, l'algorithme indique les indices i tels que $t_0 \dots t_i$ est dans le langage reconnu par M, soit A^*X, donc admet un mot de X comme suffixe. Or une occurrence d'un tel mot dans t correspond bien à cette situation, la partie A^* servant à dénoter les lettres de t qui sont avant l'occurrence. La terminaison est claire (boucle bornée).