Partie I - Coloration de graphes

L'objectif de cette partie est de proposer une implémentation d'une solution au problème de coloration d'un graphe.

I.1 - Définitions et propriétés

Soit $G = (S, A)$ un graphe fini non orienté avec S son ensemble de sommets et A son ensemble d'arêtes. On suppose que le graphe est simple c'est-à-dire qu'il ne comporte pas d'arêtes $\{s, s\}$ et que chaque paire de sommets est reliée par au plus une arête. On note n, le cardinal de l'ensemble S. Les sommets sont numérotés de 0 à $n - 1$.
Étant donné un entier naturel k, une k-coloration des sommets de G est une application $c : S \to \{0, 1, \dots, k-1\}$ telle que pour chaque arête $\{x, y\}$ d'extrémités x et $y, c(x) \neq c(y)$. Si $c(x) = i$, on considérera que la couleur i est affectée au sommet x. Si G admet une k-coloration, il est k-coloriable. On définit le nombre chromatique $\chi(G)$ d'un graphe G fini par $\chi(G) = \min\{k \in \mathbb{N}, G$ est k-coloriable $\}$.
Une clique est un sous-ensemble de sommets du graphe, adjacents 2 à 2 . On dit qu'un graphe est complet si il est une clique. On notera K_p le graphe complet à p sommets.
On pose $\omega(G) = \max\{p \in \mathbb{N} \mid K_p \text{ est une clique de G}\}$, avec $\mathbb{N}$ l'ensemble des entiers naturels.

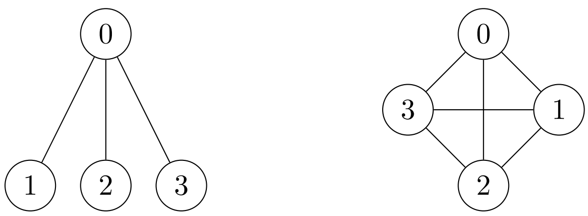

FIGURE 1 – De gauche à droite : le graphe G_1 et le graphe K_4

Q1. Le graphe G_1 de la figure 1 ci-dessus est-il 2-coloriable ? Justifier votre réponse.

Q2. Pour un entier naturel $n \geqslant 1$, déterminer le nombre chromatique du graphe K_n.

Q3. Montrer que pour tout graphe G à n sommets, on a $\omega(G) \leqslant \chi(G) \leqslant n$.

I.2 - Algorithmique et programmation

La coloration d'un graphe G avec $\chi(G)$ couleurs est un problème complexe. Dans cette sous-partie, nous présentons une heuristique permettant de construire une coloration d'un graphe donné.
Dans la suite, on implémente un graphe par sa représentation en liste d'adjacence, de type Ocaml `type graphe = int list array`.

Q4. Définir en Ocaml la représentation en liste d'adjacence du graphe G_1.

Q5. Écrire une fonction `degres_sommets : graphe -> int*int array` qui prend en paramètre la représentation d'un graphe et renvoie un tableau `t` tel que `t.(i)` contient un couple (d_i, i) où d_i est le degré du sommet i.

Q6. On suppose qu'on dispose d'une fonction Ocaml `tri : 'a array -> unit` qui trie un tableau dans l'ordre décroissant. En particulier sur un tableau de couples cette fonction trie selon le premier élément du couple. En déduire une fonction `tri_degres : graphe -> int array` qui prend en paramètre la représentation d'un graphe et renvoie un tableau contenant les numéros des sommets, triés par degrés décroissants.

Q7. Écrire une fonction Ocaml `test : graphe -> int array -> bool` qui prend en paramètre la représentation d'un graphe G et un tableau `tc` tels que `tc.(i)` contient la couleur du sommet i. La fonction renvoie `true` si `tc` est une 2-coloration pour G.

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

Algorithme 1 Welsh-Powel (coloration de graphe)

```
fonction WP(G)
    ▷ Entrée : un graphe G à n sommets ◁
    ▷ Sortie : un tableau d'entiers contenant en position i la couleur du sommet numéro i ◁
    Ordonner les sommets selon les degrés décroissants dans un tableau td
    colorie : tableau de taille n initialisé à -1
    ▷ À terme, colorie associera à chaque i, la couleur du sommet i ◁
    tant que il reste des sommets à colorier faire
        Chercher dans td le premier sommet non colorié
        Le colorer avec la plus petite couleur c non utilisée
        Colorier avec cette même couleur, en respectant leur ordre dans td, tous
        les sommets non coloriés et non adjacents à des sommets de couleur c
    renvoyer colorie
```

Q8. Que contient `colorie` à la fin si on déroule l'algorithme de coloriage ci-dessus avec le graphe G_1 en entrée ?

Q9. Écrire une fonction Ocaml `adjacent : graphe -> int array -> int -> int` qui prend en paramètre la représentation d'un graphe, un tableau `tc` contenant la couleur des sommets coloriés, le numéro d'un sommet `s`, une couleur `c` et renvoie `true` si le sommet `s` est adjacent à un des sommets de couleur `c`, `false` sinon.

Q10. Proposer une implémentation en Ocaml de l'algorithme de Welsh-Powel.

Application

Le tableau ci-dessous représente les liens d'amitiés entre huit étudiants : Alice (A), Béatrice (B), Carl (C), David (D), Eloïs (E), Fanny (F), Gary(G) et Hedge (H).

Prénom	A	B	C	D	E	F	G	H
Ami-e avec	B,C,G	A,C,E,F	A, B	E,F	B,D,F	B,D,E,H	A,H	F,G

On souhaite créer des groupes de travail. Dans le contexte de l'application, un groupe contient au moins 2 étudiants tel que chaque étudiant soit dans un groupe différent de celui de ses amis.

Q11. Modéliser la situation par un graphe et en déduire une solution.

Partie II - Satisfiabilité d'une formule propositionnelle

Une formule propositionnelle est construite à l'aide de constantes propositionnelles, de variables propositionnelles et de connecteurs logiques. Les connecteurs logiques seront notés $\neg$ (négation), $\wedge$ (conjonction), $\vee$ (disjonction). Dans cette partie, on étudie le problème de satisfiabilité d'une formule et son application à la détermination d'une conséquence logique entre 2 formules propositionnelles. Le problème CNF-SAT est défini de la façon suivante. Étant donné une formule sous forme normale conjonctive, admet-elle un modèle, c'est-à-dire une valuation des variables, qui rende la formule vraie ? On souhaite écrire un programme qui teste si une valuation donnée rend une telle formule vraie.

Dans cette partie, on considère que si une formule contient n variables propositionnelles, elles seront désignées par $x_0, x_1, \ldots, x_{n-1}$.
On définit le type OCaml suivant :

```
type clause = Var of int
            | Non of clause
            | Ou of clause * clause
```

L'argument du constructeur var correspond au numéro de la variable concernée.
Une formule sous forme normale conjonctive ayant m clauses sera implémentée par une liste de m clauses. Les tableaux seront implémentés par le module `Array` dont les éléments suivants pourront être utilisés :

- `type 'a array`, notations `[| |]`
- création d'un tableau : `make : int -> 'a -> 'a array`
- accès à l'élément d'indice `i` du tableau `t` : `t.(i)`
- modification de l'élément placé à l'indice `i` du tableau `t` : `t.(i) <- v`
- taille du tableau : `length : 'a array -> int`

Q12. Donner le code OCaml correspondant à la clause $c = (x_0 \vee x_1) \vee \neg x_2$.

Q13. Donner le code OCaml permettant de définir la formule : $f = (x_0 \vee x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2)$.

Q14. Écrire une fonction de signature `evalue_clause : clause -> bool array -> bool` qui prend en paramètre une clause et une valuation représentée par un tableau contenant à l'indice i, la valeur de vérité de la variable x_i et renvoie la valeur de vérité de la clause.

Q15. Écrire une fonction de signature `evalue_FNC : clause list -> bool array -> bool` qui prend en paramètre une liste de clauses et une valuation représentée par un tableau contenant à l'indice i, la valeur de vérité de la variable x_i et évalue une formule donnée sous forme normale conjonctive.

Q16. Quel résultat obtient-on avec la formule F et le tableau de valuations `[|false; true; true|]` ? Justifier.

On souhaite énumérer toutes les valuations possibles pour un nombre de variables fixé. Étant donné une valuation, on considérera que si la valeur `true` correspond à 1 et la valeur `false` correspond à 0 , la valuation suivante correspond à l'ajout de 1 au nombre binaire associé. Ainsi, la valuation suivante de `[|false; true; false|]` est `[|false; true; true|]`. On considère que la valuation suivante de `[|true; true; true|]` n'existe pas.

Q17. Écrire une fonction de signature `suivant : bool array -> bool` qui prend en paramètre un tableau de booléens, lui attribue la valuation "suivante" si possible et renvoie `true` ; sinon renvoie `false`.

Q18. En déduire une fonction de signature `satisfiable : clause list -> int -> bool` qui prend en paramètre une formule en forme normale conjonctive, son nombre de variables et renvoie `true` si il existe une valuation qui rend la formule vraie, `false` sinon.

Q19. Quelle est la complexité en temps de cette fonction par rapport aux paramètres d'entrée ?

Q20. Proposer une stratégie de retour sur trace pour résoudre le problème de satisfiabilité d'une formule.

Conséquence logique entre 2 formules

Définition Une formule ϕ est une conséquence logique d'un ensemble fini de n formules $\Gamma = \{F_1, \ldots, F_n\}$, n étant un entier naturel supérieur ou égal à 1 , si tout modèle de ϕ est un modèle de Γ. On note $\Gamma \models \phi$. On admettra que toute formule admet une formule équivalente sous forme normale conjonctive.

Q21. Déduire de la fonction précédente, un algorithme en pseudo-code permettant de déterminer si une formule F est une conséquence logique d'un ensemble de formules $\Gamma : F_1, \ldots, F_n$.

Q22. Afin de déterminer si $\Gamma \models \phi$, on peut prouver le séquent $\Gamma \vdash \phi$. Justifier cette méthode, puis construire un arbre de preuve qui démontre le séquent $\Gamma : P \to Q, Q \to R, P \vdash P$ où P, Q, R désignent des variables propositionnelles représentant des formules logiques, à partir des règles d'inférence de la déduction naturelle ; les règles et notations utilisées seront clairement mentionnées.

Partie III - Autour des tas

L'objectif est ici d'étudier et d'implémenter quelques outils autour d'une structure de données appelée *tas binomial*. Un tas binomial est une structure assez proche du tas binaire (utilisé par exemple pour réaliser une file de priorité), pour lequel la procédure de fusion de deux tas est efficace et peu complexe.

III.1 - Arbre binomial

Définition 5 (Arbre enraciné)
Un *arbre enraciné* est une généralisation des arbres binaires dans laquelle un noeud peut avoir plus de 2 fils.
La figure 2 présente un exemple d'arbre enraciné dans lequel 0 est la racine.

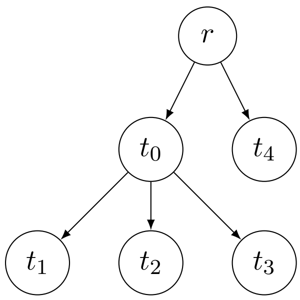

FIGURE 2 – Exemple d'arbre enraciné

On définit un arbre enraciné (non vide) par la valeur de sa racine r et $[t_0, \cdots, t_{n-1}]$ la liste de ses fils, chaque t_i étant un arbre. Un arbre vide est défini par `Vide`.
On utilisera le type suivant en Ocaml :

```
type 'a arbre = Vide | Noeud of 'a * 'a arbre list;;
```

Par abus de notation, on confond dans la suite la racine de l'arbre et sa valeur, ainsi que les fils d'un arbre avec leur racine.

Q23. Écrire des fonctions :
- `vide : 'a arbre -> bool` qui prend en entrée un arbre `a` et renvoie `true` si l'arbre `a` est vide, `false` sinon ;
- `racine : 'a arbre -> 'a` qui prend en entrée un arbre `a` et renvoie la racine de `a` si `a` est non vide ;
- `fils : 'a arbre -> 'a arbre list` qui prend en entrée un arbre `a` et renvoie la liste des arbres, fils de la racine de `a`.

Définition 6 (Arbre binomial)
Un *arbre binomial* a_k d'ordre $k \geq 0$ est un arbre enraciné dans lequel les fils de chaque nœud sont ordonnés. II est défini récursivement comme suit :

i) $a_0 =$ `Noeud(r,[])` est constitué d'un nœud unique, la racine. Cet arbre est d'ordre 0 ;
ii) pour $k \in \mathbb{N}$, soit $a_k =$ `Noeud`$(r, [t_0; ...; t_{n-1}])$ un arbre enraciné non vide. a_k est un arbre binomial d'ordre k si :
   — t_0 est un arbre binomial d'ordre $(k-1)$;
   — `Noeud`$(r, [t_1; ...; t_{n-1}])$ est un arbre binomial d'ordre $(k-1)$;
   — la racine de t_0 a une valeur supérieure ou égale à `r`.

La figure 3 donne un exemple d'arbre binomial d'ordre 3. Les valeurs dans l'arbre sont des entiers.

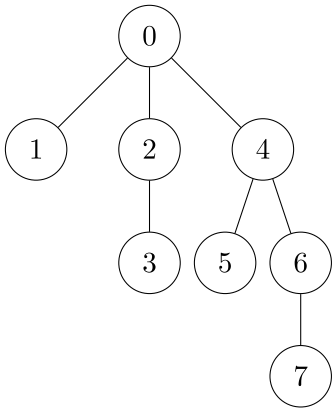

FIGURE 3 – Exemple d'arbre binomial d'ordre 3

Q24. Écrire une fonction `arbreBin : 'a arbre -> 'a arbre -> 'a arbre` qui construit, à partir de deux arbres binomiaux a_1 et a_2 d'ordre $k-1$, un arbre binomial a_k d'ordre k contenant les mêmes valeurs que a_1 et a_2.
Dans la suite, on note $a_1 \oplus a_2$ cette opération.

Q25. Montrer par récurrence que la racine d'un arbre binomial d'ordre k a exactement k fils.

Q26. En déduire une fonction `ordre : 'a arbre -> int` qui renvoie l'ordre d'un l'arbre binomial `a`.

Q27. Montrer qu'un arbre binomial `a` d'ordre k possède 2^k nœuds.

Q28. Écrire une fonction récursive `estUnArbreBinomial : 'a arbre -> bool` qui renvoie `true` si `a` est un arbre binomial, `false` sinon.

III.2 - Tas binomial

Un *tas* est une structure de données de type arbre qui permet en particulier de retrouver directement un élément qui doit être traité en priorité.

Définition 7 (Tas binomial)
Soient $k \geq 0$ et `T` $= \{a_0, \cdots, a_k\}$ un ensemble d'arbres. `T` est un *tas binomial* de longueur $k+1$ si, pour tout $i \in [|0, k|]$, a_i est soit un arbre vide, soit un arbre binomial d'ordre i. Si $i = k, a_i$ ne peut, de plus, pas être vide et a_k est donc un arbre binomial d'ordre k.

On note dans la suite |`T`| le nombre de nœuds d'un tas `T`, qui est le nombre total de noeuds des arbres qui constituent `T`.
En Ocaml, on représentera un tas binomial par le type suivant :

```
type 'a tasbin = {arbres : 'a arbre array; mutable taille :
    int};;
```

Le tableau `arbres` sera d'une grande taille $N = 100$ et contiendra les différetns arbres composant le tas binomial. Le champ `taille` sert à savoir exactement combien de cases du tableau sont utilisées.
Par exemple si $T = \{a_0; a_1; a_2\}$, alors `taille` vaudra 3 et `arbres` aura des arbres non vides dans les cases 0,1,2 ; toutes les autres cases contenant `Vide` (l'arbre vide).

Définition 8 (Signature d'un tas) Soit $\mathtt{T} = \{a_0, \cdots, a_k\}$ un tas binomial de longueur $k+1$. On appelle *signature* de T la suite $s_0 \cdots s_k$ telle que pour tout $i \in [|0, k|]$, $s_i = 0$ (respectivement $s_i = 1$) si l'arbre a_i est vide (respectivement n'est pas vide).

Q30. Soit T un tas binomial de longueur $k+1$. En utilisant sa signature, calculer $|\mathtt{T}|$ et montrer que $2^k \leq |\mathtt{T}| < 2^{k+1}$. En déduire k en fonction de $|\mathtt{T}|$.

Q31. Écrire une fonction `minimumTas : 'a tasbin -> 'a` qui prend en entrée un tas T et retourne la valeur minimum du tas. En donner la complexité en fonction de $|\mathtt{T}|$.

Les tas se construisent itérativement à partir de données. On est donc amené, pour un tas T, à ajouter un à un des éléments.
Soit p un élément que l'on souhaite ajouter à un tas binomial T non vide et déjà construit. L'insertion de la valeur p dans le tas T se fait alors selon l'algorithme 2.

Algorithme 2 Insertion de p dans T

```
fonction INSERTION(T, p)
    ▷ Entrée : un tas T = {a0...ak}, une valeur p ◁
    ▷ Sortie : un tas (T augmenté de la valeur p) ◁
    i = 0
    Coder p dans un arbre binomial a d'ordre 0
    tant que i<k+1 et a non vide faire
        si ai est vide alors
            ai devient a
            a devient vide
        sinon
            a devient a ⊕ ai (***)
            ai devient vide
        i=i+1
    si a n'est pas vide alors
        Ajouter a au tas T (qui devient donc de longueur k + 1)
```

Q32. Coder l'algorithme 2 sous la forme d'une fonction `insertion : 'a -> 'a tasbin -> unit`. L'algorithme doit modifier le tas par effet de bord.

Q33. Évaluer la complexité de cet algorithme en fonction de $|\mathtt{T}|$. On suppose que l'étape marquée $(***)$ s'effectue en temps constant.

Q34. Donner la signature du tas résultant de l'insertion de p dans T en fonction de la signature de T.

Q35. Donner, sans justification, un invariant de boucle pour la boucle de l'algorithme 2 permettant de prouver la correction de ce dernier.