DS4

N.B. : le candidat attachera la plus grande importance à la clarté, à la précision et à la concision de la rédaction. Si un candidat est amené à repérer ce qui peut lui sembler être une erreur d'énoncé, il le signalera sur sa copie et devra poursuivre sa composition en expliquant les raisons des initiatives qu'il a été amenées à prendre.

RAPPEL DES CONSIGNES

- Utiliser uniquement un stylo noir ou bleu foncé non effaçable pour la rédaction de votre composition ; d'autres couleurs, excepté le vert, peuvent être utilisées, mais uniquement pour les schémas et la mise en évidence des résultats.
- Ne pas utiliser de correcteur.
- Écrire le mot FIN à la fin de votre composition.

Les calculatrices sont interdites.
Le sujet est composé de quatre parties, toutes indépendantes.

1 Exercice 1

On s'intéresse au problème SAT pour une formule en forme normale conjonctive. On se fixe un ensemble fini $\mathcal{V} = \{v_0, v_1, \ldots, v_{n-1}\}$ de variables propositionnelles.

Un *littéral* ℓ_i est une variable propositionnelle v_i ou la négation d'une variable propositionnelle $\neg v_i$. On représente un littéral en OCaml par un type énuméré : le littéral v_i est représenté par `V i` et le littéral $\neg v_i$ par `NV i`. Une *clause* $c = \ell_0 \vee \ell_1 \vee \cdots \vee \ell_{|c|-1}$ est une disjonction de littéraux, que l'on représente en OCaml par un tableau de littéraux. On ne considérera dans cet exercice que des formules sous forme normale conjonctive, c'est-à-dire sous forme de conjonctions de clauses $c_0 \wedge c_1 \wedge \cdots \wedge c_{m-1}$. On représente une telle formule en OCaml par une liste de clauses, soit une liste de tableaux de littéraux. On n'impose rien sur les clauses : une clause peut être vide et un même littéral peut s'y trouver plusieurs fois. De même une formule peut n'être formée d'aucune clause, elle est alors notée $\top$ et est considérée comme une tautologie. Une *valuation* $v : \mathcal{V} \to \{V, F\}$ est représentée en OCaml par un tableau de booléens.

On utilisera les types suivants pour représenter les formules :

```
(* 'V i' correspond à la variable propositionnelle d'indice 'i' et 'NV
   i' à la négation de la variable propositionnelle d'indice 'i' *)
type litteral =
  | V of int
  | NV of int

type clause = litteral array

type formule = clause list

type valuation = bool array

(* indice : litteral -> int *)
let indice x = match x with
 |V i -> i
 |NV i -> i

let initialise (n : int) : valuation =
  Array.init n (fun _ -> Random.int 2 = 0)
```

La fonction `initialise : int -> valuation` permet d'initialiser une valuation aléatoire.

1. Implémenter la fonction `evalue : clause -> valuation -> bool` qui vérifie si une clause est satisfaite par une valuation.

   Etant donnée une formule f constituée de m clauses $c_0 \wedge c_1 \wedge \ldots \wedge c_{m-1}$ définies sur un ensemble de n variables, la fonction `random_sat` a pour objectif de trouver une valuation qui satisfait la formule, s'il en existe une et qu'elle y arrive. Cette fonction doit effectuer au plus k tentatives et renvoyer un résultat de type `valuation option`, avec une valuation qui satisfait la formule passée en paramètre si elle en trouve une et la valeur `None` sinon. L'idée de ce programme est d'effectuer une assignation aléatoire des variables puis de vérifier que chaque clause est satisfaite. Si une clause n'est pas satisfaite, on modifie aléatoirement la valeur associée à un littéral de cette clause, qui devient ainsi satisfaite, puis on recommence.

```
(* Objectif : prendre en entrée une formule ‘f‘ sur ‘n‘ variables, un
   nombre d'essais maximum ‘k‘ supposé strictement positif et s'évaluer
   en une option sur une valuation qui satisfait ‘f‘ si on en trouve
   une ou ‘None‘ si on échoue après ‘k‘ tentatives *)
let random_sat (f : formule) (n : int) (k : int) : valuation option =
  assert (k >= 1);
  let v = initialise n in
  let rec test (g : formule)  (k : int) : valuation option =
    match g with
    | [] -> Some v
    | c :: cs ->
       if evalue c v then
         test cs k
       else if k < 1 then
         None
       else begin
         let i = Random.int (Array.length c) in
         v.(indice c.(i)) <- not v.(indice c.(i));
         test g (k - 1)
       end
  in
  test f k
```

2. Si ce programme renvoie `None` , peut-on conclure que la formule f en entrée n'est pas satisfiable ? De quel type d'algorithme probabiliste s'agit-il ?
3. Proposer un jeu de 5 tests élémentaires permettant de tester la correction de ce programme.
4. Ce programme est-il correct par rapport à sa spécification ? Si cela s'avère nécessaire, corriger ce programme pour qu'il remplisse ses objectifs.

   On s'intéresse maintenant au problème MAX-SAT qui consiste, toujours pour une formule en forme normale conjonctive comme ci-dessus, à trouver une valuation qui satisfait le plus grand nombre de clauses de cette formule simultanément satisfiables. Un algorithme d'approximation probabiliste naif pour obtenir une solution approchée consiste à générer aléatoirement k valuations et retenir celle qui maximise le nombre de clauses satisfaites.
5. Implémenter cette approche en OCaml. Quelle est sa complexité dans le meilleur et dans le pire cas ?
6. Sous l'hypothèse $P \neq NP$, peut-il exister un algorithme de complexité polynomiale pour résoudre MAX-SAT ? Justifier.

2 Exercice 2

On dispose de $n \geq 1$ objets $\{o_0, \ldots, o_{n-1}\}$ de valeurs respectives $(v_0, \ldots, v_{n-1}) \in \mathbb{N}^n$ et de poids respectifs $(p_0, \ldots, p_{n-1}) \in \mathbb{N}^n$. On souhaite transporter dans un sac de poids maximum $pmax$ un sous-ensemble d'objets ayant la plus grande valeur possible. Formellement, on souhaite donc maximiser

$$\sum_{i=0}^{n-1} x_i v_i$$

sous les contraintes
$(x_0, \ldots, x_{n-1}) \in \{0,1\}^n$ et $\sum_{i=0}^{n-1} x_i p_i \leq pmax$

Intuitivement, la variable x_i vaut 1 si l'objet o_i est mis dans le sac et 0 sinon.

On propose d'utiliser un algorithme glouton dont le principe est de considérer les objets $o_0, o_1, \ldots, o_{n-1}$ dans l'ordre et de choisir à l'étape i l'objet i (donc poser $x_i = 1$) si celui-ci rentre dans le sac avec la contrainte de poids maximal respectée et ne pas le choisir (donc poser $x_i = 0$) sinon. On remarque que les valeurs $v_0, v_1, ..., v_{n-1}$ ne sont pas directement utilisées par cet algorithme, elle le seront lors du tri éventuel des objets.

1. L'algorithme glouton ci-dessus donne-t-il toujours une solution optimale ?
   (a) si on ne suppose rien sur l'ordre des objets a priori ;
   (b) si les objets sont triés par ordre de valeur décroissante ;
   (c) si les objets sont triés par ordre de poids croissant ;
   (d) si les objets sont triés par ordre décroissant des quotients v_i/p_i.

   Justifier à chaque fois votre réponse à l'aide d'un contre-exemple ou d'une démonstration.
2. Le problème du sac à dos étudié dans cet exercice est un problème d'optimisation. Donner le problème de décision associé en utilisant une valeur seuil `vseuil`. Montrer que ce problème de décision est dans la classe NP.
3. Quelle serait la complexité d'une méthode qui examinerait tous les choix possibles pour retenir le meilleur ? Quelles stratégies d'élagage pourrait-on mettre en œuvre pour réduire l'espace de recherche ?

   On peut montrer que ce problème de décision est NP-complet.
   On rappelle que le problème SubsetSum est défini comme suit :
   Instance : $x_0, \ldots, x_{n-1} \in \mathbb{N}^*$ et $S \in \mathbb{N}$
   Question : existe-t-il $I \subseteq [0 \ldots n-1]$ tel que $\sum_{i \in I} x_i = S$?
4. En réduisant SubsetSum, montrer que le problème du sac à dos avec seuil est NP-complet.

3 Exercice 3 : Bases de données

On s'intéresse dans cette partie à un site Internet d'échange de supports de cours entre enseignants de NSI. Chaque personne désirant proposer ou récupérer du contenu doit commencer par se créer un compte sur ce site et peut ensuite accéder à du contenu ou en proposer.

Ce site repose sur une base de données contenant en particulier trois tables.

— La table comptes possède un enregistrement par utilisateur ou utilisatrice, et ses attributs sont :
— id, un identifiant numérique, unique pour chaque compte ;
— nom, le nom de la personne possédant le compte ;
— et d'autres informations, concernant le mot de passe, l'adresse mail, des préférences sur le site, etc., que nous ne détaillons pas ici.
— La table ressources possède un enregistrement par document téléversé sur le site. Ses attributs sont :
— id, un identifiant numérique, unique pour chaque ressource ;
— owner, l'identifiant de la personne ayant créé la ressource ;
— titre, une chaine de caractères décrivant la ressource ;
— type, chaine de caractères pouvant être cours, ds, tp ou td.
— La table chargement mémorise chaque fois qu'un utilisateur télécharge une ressource sur le site. Ses attributs sont :
— date, date du téléchargement, par exemple '2021-02-28' pour le 28 février 2021 (on peut utiliser des opérations de comparaison classiques avec ce format) ;
— id_u, identifiant de l'utilisateur qui télécharge la ressource ;
— id_r, identifiant de la ressource téléchargée.

Voici un extrait de chacune de ces tables :

comptes		
id	nom	…
1	Ada Lovelace	…
4	Alan Turing	…
…	…	…

ressources			
id	owner	titre	type
4	1	Machine à décalage	cours
13	4	Intelligence artificielle	td
…	…	…	…

chargement		
date	id_u	id_r
'1931-06-29'	4	4
'2020-05-30'	27	458
…	…	…

1. Écrire une requête SQL permettant de connaitre le nombre total de ressources de type cours présentes sur le site.
2. Que fait la requête suivante : ?

```
SELECT ressource.titre, comptes.nom
FROM chargement
  JOIN ressources ON ressouces.id = chargement.id_r
  JOIN comptes ON comptes.id = chargement.id_u
ORDER BY chargement.date DESC
LIMIT 1
```

3. Écrire une requête SQL qui permet de déterminer la liste des couples (x, y), signifiant que la personne possédant l'identifiant x a téléchargé des documents téléversés par la personne possédant l'identifiant y.

   On définit le graphe non-orienté $G(V, E)$ où V est l'ensemble des identifiants de comptes sur le site et $E \subset V \times V$ l'ensemble des paires d'identifiants telles que le premier compte a déjà téléchargé des documents téléversés par l'autre et réciproquement. Ainsi, si $(x, y) \in E$, alors on doit avoir $(y, x) \in E$.
4. Écrire une requête SQL qui renvoie la table des couples (x, y) de E.

4 Calcul d'une coupe minimum d'un graphe

Un agent secret a pour mission de perturber le réseau de communications ennemi en coupant certains fils dans le réseau. Ayant peu de moyens, l'agent a pour consigne de couper le moins de fils possibles pour accomplir cette tâche. Le réseau de communications est modélisé à l'aide d'un *multigraphe* non orienté $G = (S, A)$, où S est l'ensemble des sommets du graphe et A l'ensemble de ses arêtes. Résoudre le problème de l'agent secret, c'est rechercher une *coupe minimum* dans G.

Définition 4 (Multigraphe)
Un multigraphe est un graphe dans lequel des couples de mêmes sommets peuvent être reliés par plus d'une arête.

Dans toute la suite, on considérera les multigraphes sans arêtes du type (s, s) (boucles).

Définition 5 (Coupe)
Une coupe d'un multigraphe non orienté $G = (S, A)$ est une partition non triviale (S_1, S_2) de S, c'est-à-dire telle que $S_1 \cup S_2 = S$, $S_1 \cap S_2 = \emptyset$ et $S_1, S_2 \neq \emptyset$. On dira que les arêtes reliant S_1 à S_2 sont les arêtes de la coupe.

La *taille* d'une coupe (S_1, S_2) est définie par $|(S_1, S_2)| = |\{(s, t) \in A, s \in S_1, t \in S_2\}|$. Autrement dit, c'est le nombre d'arêtes de G qui relient S_1 et S_2.

Définition 6 (Coupe minimum)
Une coupe minimum est une coupe de taille minimale.

Pour trouver une coupe minimum d'un multigraphe $G = (S, A)$, on pourrait énumérer l'ensemble des coupes possibles (il y en a $2^{|S|}$...), ou encore utiliser un algorithme de flot (le plus efficace étant en $\mathcal{O}\left(|S|^3\right)$). Nous proposons dans la suite un algorithme probabiliste, l'algorithme de Karger, basé sur la notion de contraction d'arête.

4.1 Contraction d'arête

Soient $G = (S, A)$ un multigraphe et $a = (s, t) \in A$ une arête de G de sommets s et t. *Contracter* l'arête a consiste à :

i Créer un nouveau sommet $u = (st)$ dans S. u est un *supersommet* du nouveau graphe.

ii Pour toute arête (r, s) ou (r, t), $r \in S$, ajouter une arête (r, u) à A. Dans le cas où (r, s) et (r, t) existent, deux arêtes reliant r à u sont créées.

iii Supprimer de A toutes les arêtes ayant s ou t comme extrémité.

iv Supprimer s et t de S.

Un supersommet peut donc être vu comme un ensemble de deux sommets du graphe initial, obtenu après une contraction d'arête.

Le multigraphe obtenu est appelé *graphe contracté* et est noté G/st. Si plusieurs arêtes relient s à t dans G, contracter l'une d'entre elles supprime toutes les autres du graphe G/st. Dans toute la suite, on notera G_0 le graphe initial, avant transformations par contractions.

On considère le graphe G_0 de la **figure 1**.

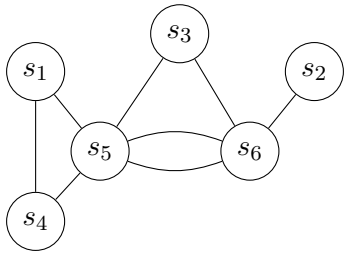

FIGURE 1 – Graphe G_0 exemple

Q1 Donner le graphe obtenu par contraction d'une arête (s_5, s_6).

Après plusieurs contractions, un supersommet u du graphe résultant contient un sous-ensemble $S_u \subseteq S$ de sommets de G_0.

Q2 Soit u un supersommet et $s, t \in S_u$. Montrer qu'il existe un chemin Γ entre s et t dans G_0 tel que chaque arête de Γ a été contractée.

Q3 Montrer qu'en contractant une arête s, t dans un multigraphe G, la taille d'une coupe minimum du graphe contracté G/st est au moins égale à la taille d'une coupe minimum de G.

Q4 Montrer que la taille d'une coupe minimum de G/st est strictement plus grande que celle d'une coupe minimum de G si et seulement si l'arête (s, t) est une arête de toutes les coupes minimum de G.

4.2 Premier algorithme

On construit un premier algorithme qui essaye de trouver une coupe minimum en contractant aléatoirement une arête de G, jusqu'à ce qu'il ne reste que deux sommets (**algorithme 1**).

Algorithme 1 – Algorithme de Karger

```
Karger(G)
Entrees : G = (S, A) multigraphe non orienté
Sorties : Une coupe minimum de G.
Algorithme :
 pour i = |S| à 3 en décrémentant de 1 faire
   Tirer aléatoirement (loi uniforme) une arête a = (s, t) ∈ A
   G = G/st
 finPour
 retourner G
```

Q5 Appliquer l'**algorithme 1** au graphe de la **figure 1**, en contractant successivement (s_5, s_6), $((s_5s_6), s_2)$, (s_1, s_4) et $((s_5s_6s_2), s_3)$. Chaque graphe contracté sera représenté. La coupe calculée est-elle une coupe minimum ?

D'après la **Q4**, tant que l'on ne contracte pas une arête faisant partie de toutes les coupes minimum de G, alors l'**algorithme 1** trouvera une coupe minimum. On cherche alors la probabilité de ne jamais contracter une telle arête.

Q6 Soit $d(s)$ le degré d'un sommet $s \in S$. Montrer que $\sum d(s) = 2|A|$ (lemme des poignées de main).

Q7 En déduire qu'une coupe minimum a une taille d'au plus $\dfrac{2|A|}{|S|}$.

Soit $C = (S_1, S_2)$ une coupe minimum de G. L'objet des questions **Q8** et **Q9** est de minorer la probabilité que l'**algorithme 1** renvoie C. Pour cela, on remarque que cet algorithme ne renvoie pas C si et seulement si lors d'une itération on choisit une arête qui traverse C, c'est-à-dire telle qu'un sommet est dans S_1 et l'autre dans S_2.

Q8 Quelle est la probabilité maximum de choisir une arête qui traverse C ?

Q9 En déduire que la probabilité $P_{|S|}$ que l'**algorithme 1** renvoie C est supérieure ou égale à $\dfrac{2}{|S|(|S|-1)}$.

Q10 Déduire de la question précédente qu'un multigraphe non orienté $G = (S, A)$ possède au plus $\dfrac{|S|(|S|-1)}{2}$ coupes minimum.

4.3 Deuxième algorithme

Le résultat précédent n'est pas satisfaisant d'un point de vue complexité. On peut cependant l'améliorer facilement en utilisant une technique d'amplification : on itère l'**algorithme 1** plusieurs fois et on retourne la valeur minimale obtenue (**algorithme 2**).

Algorithme 2 – Algorithme de Karger amplifié

```
KargerAmplifier(G, N)
Entrees : G = (S, A) multigraphe non orienté, N ∈ ℕ
Sorties : Une coupe minimum de G.
Algorithme :
 t = ∞
 pour i = 1 à N faire
   X = Karger(G)
   si |X| < t alors
     t = |X|
     C = X
   finSi
 finPour
 retourner C
```

Q11 Montrer que la probabilité maximale que l'**algorithme 2** ne trouve pas une coupe minimum est égale à $\left(1 - \frac{2}{|S|(|S|-1)}\right)^N$.

Q12 On rappelle que pour tout $x \in \mathbb{R}$, $1 + x \leqslant \mathrm{e}^x$. Montrer que la probabilité que l'**algorithme 2** renvoie une coupe minimum est supérieure à $1 - \mathrm{e}^{\frac{-2N}{|S|(|S|-1)}}$. En déduire, pour $c > 0$ fixé, la plus petite valeur de N telle que cette probabilité soit supérieure à $1 - \frac{1}{N^c}$.

Q13 Les algorithmes proposés dans cette partie sont des algorithmes probabilistes. Sont-ils de type Monte Carlo ou Las Vegas ? Justifier la réponse.

4.4 Implémentation en langage C

Cette partie comporte des questions de programmation qui seront abordées en utilisant **exclusivement le langage C**. Les codes seront commentés de manière pertinente.

On suppose qu'un graphe est codé à l'aide de l'ensemble de ses arêtes, chaque arête étant définie par ses deux sommets, représentés par des entiers.

Q14 Définir en C des types structurés `Arete` et `Graphe`. Pour ce dernier, on s'attachera à avoir un accès direct au nombre de sommets et au nombre d'arêtes. On rappelle la définition d'un type structuré par `struct nom_s {type`$_1$ `champ`$_1$`;...; type`$_n$ `champ`$_n$`;}` et ensuite `typedef struct nom_s nom;`.

Pour tout u supersommet, les S_u forment une partition de S. Pour coder ces sous-ensembles, on a donc naturellement recours à la structure de données Unir & Trouver (Union-Find), qui va permettre de trouver rapidement à quel sous-ensemble S_u un sommet $s \in S$ appartient (Trouver) et de fusionner deux supersommets S_u et S_v déjà construits (Unir).

On suppose donc disposer :

(i) d'un type subset, décrivant un supersommet d'un graphe contracté.

```
struct subset {
  int parent;  // recherche par compression de chemin pour Trouver
  int rang;    // Union par rang .
};
typedef struct subset subset;
```

(ii) d'une fonction de prototype `int Trouver(subset subsets[], int s)`, qui retourne le numéro du supersommet dans lequel le sommet s se trouve (par compression de chemin),

(iii) d'une fonction de prototype `void Unir(subset subsets[], int Su, int Sv)` qui fusionne les deux supersommets S_u et S_v (union par rang) et maintient à jour la liste des supersommets.

Il n'est pas demandé d'écrire ces deux dernières fonctions.

Q15 Écrire une fonction de prototype `int contracteArete(Graphe G, subset subsets[], Arete a)` qui contracte l'arête `a`. On veillera à ne pas contracter l'arête si ses deux extrémités sont dans le même supersommet. La fonction renvoie 0 si aucune arête n'est contractée et -1 sinon.

Q16 Écrire une fonction de prototype `int compteAretesCoupe(Graphe G, subset subsets[])` qui compte le nombre d'arêtes du graphe contracté final G résultant de l'**algorithme 1**.

`subsets` est la partition des sommets du graphe initial S calculée par l'**algorithme 1**.

Q17 En déduire une fonction de prototype `int kargerMinCut(Graphe GO)` qui renvoie la taille de la coupe calculée par l'**algorithme 1** sur le graphe G_0 . Les supersommets initiaux sont les $|S|$ sommets de G_0, chacun ayant un rang nul et un numéro parent égal au numéro du sommet.

Pour effectuer un tirage aléatoire uniforme, on utilisera la fonction `int rand();` de la librairie `stdlib.h` qui renvoie un nombre aléatoire entre 0 et `RAND_MAX` $\geqslant 32767$. `RAND_MAX` est une constante définie dans `stdlib.h`.

Q18 Donner la complexité au pire des cas de cet algorithme.