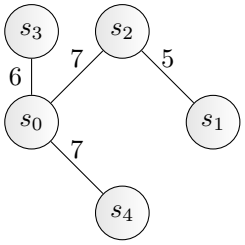
La correction proposée ne correspond sans doute pas toujours aux attentes des personnes qui ont conçu le sujet. Certaines réponses peuvent certainement être améliorées, clarifiées ou même corrigées. Le sujet original comportait quelques coquilles. Sauf incompréhension de ma part, le résultat à démontrer dans la question 22 est faux. N'hésitez pas à me faire part de vos remarques en écrivant à l'adresse suivante : jean-luc.joly@prepas.science.

I Généralités

Q. 1 Un arbre couvrant de G_2 de poids minimal est représenté par :



Q. 2

- Montrons par récurrence sur $|S|$ que si G est connexe, alors $|A| \geqslant |S| - 1$.
  - Si $|S| = 1$, alors $|A| = 0$ et l'inégalité est vérifiée.
  - On considère $n \in \mathbb{N}$ et on suppose que pour tout graphe connexe G tel que $|S| = n$, alors $|A| \geqslant |S| - 1$.
    On considère $G = (S;A)$, un graphe connexe tel que $|S| = n + 1$.
    S'il existe $s \in S$ tel que s n'admette qu'un sommet adjacent t, c'est-à-dire tel que s soit de degré 1, alors $G[S\setminus\{s\}] = (S\setminus\{s\};A\setminus\{st\})$, c'est un graphe connexe à n sommets, l'hypothèse de récurrence assure donc que :
    $$\underbrace{|A\setminus\{st\}|}_{|A|-1} \geqslant \underbrace{|S\setminus\{s\}|}_{|S|-1} - 1$$
    ce qui permet de conclure $|A| \geqslant |S| - 1$.
    Si G n'admet pas de sommet n'ayant qu'un sommet adjacent, tous les sommets de G sont au moins de degré 2 puisque G est connexe.
    Dans un graphe la somme des degrés des sommets est égal à 2 fois le nombre d'arêtes.
    Dans ce cas on a :
    $$\underbrace{\sum_{s\in S} \deg(s)}_{=2|A|} \geqslant 2\,|S|$$
    En particulier $|A| \geqslant |S| - 1$.
- Montrons que si $G = (S;A)$ est sans cycle, alors $|A| \leqslant |S| - 1$.
  On commence par justifier que G admet au moins un sommet de degré strictement inférieur à 2.
  Si ce n'est pas le cas, tous les sommets de G sont de degré au moins 2.
  On marque arbitrairement un premier sommet s_1 de S.
  On construit par induction une séquence $s_1;\cdots;s_i$ de sommets marqués deux à deux distincts en choisissant s_{i+1} un sommet non-marqué adjacent à s_i. L'algorithme s'arrête lorsque s_i ne possède pas de sommet adjacent non marqué. Puisque S est fini l'algorithme s'arrête et s_i admet alors un sommet adjacent s_j avec $j \in [\![1;i-2]\!]$. La suite de sommets distincts $(s_i;s_j;s_{j+1};\cdots;s_{i-1};s_i)$ est alors un cycle.
  On peut alors montrer par récurrence sur $|S|$ que si G ne possède pas de cycle, alors $|A| \leqslant |S| - 1$:
  - Si $|S| = 1$, alors $|A| = 0$ et l'inégalité est vérifiée.
  - On considère $n \in \mathbb{N}^*$ et on suppose l'inégalité vérifiée pour tout graphe $G = (S;A)$ sans cycle tel que $|S| = n$.
    On considère un graphe $G = (S;A)$ sans cycle tel que $|S| = n + 1$. Le résultat précédent assure que G admet un sommet s de degré 0 ou 1. $G[S\setminus\{s\}]$ est alors acyclique et possède $|A|$ ou $|A| - 1$ arêtes. L'hypothèse de récurrence assure que dans tous les cas :
    $$|A| - 1 \leqslant |S\setminus\{s\}| - 1$$

En particulier :

$$|A| \leqslant |S| - 1$$

Q. 3 Supposons que $G = (S;A)$ soit un arbre, alors G est connexe et sans cycle. Les résultats de la question **??** assurent alors que $|A| = |S| - 1$. La première propriété implique donc les deux autres.
Par ailleurs, si on démontre l'équivalence des deux dernières propriétés, on aura établi que si l'une d'entre elle est vraie, alors G est un arbre.

- On considère donc $G = (S;A)$ un graphe connexe tel que $|A| = |S| - 1$. Supposons que G admette un cycle $(s_0;s_1;\ldots;s_k)$. Alors $(S;A\setminus\{s_0s_{k-1}\})$ est connexe et :

$$\underbrace{|A\setminus\{s_0s_{k-1}\}|}_{=|A|-1} < |S| - 1$$

  ce qui contredit le résultat de la question **??** portant sur les graphes connexes. G est donc sans cycle.
- On considère $G = (S;A)$ un graphe sans cyclique tel que $|A| = |S| - 1$. Supposons que G ne soit pas connexe. On peut donc considérer s et t dans S appartenant à deux composantes connexes distinctes. En particulier $st \notin A$ et le graphe $(S;A\cup\{st\})$ est sans cycle et :

$$\underbrace{|A\cup\{st\}|}_{=|A|+1} > |S| - 1$$

  ce qui contredit le résultat de la question **??** portant sur les graphes sans cycle. G est donc connexe.

On a démontré l'équivalence des deux dernières propriétés, les trois propriétés sont finalement équivalentes.

Q. 4 *Si G n'est pas connexe, G n'admet pas d'arbre couvrant. Il y a donc une coquille dans l'énoncé original. Dans la rédaction qui suit, on suppose donc en plus que G est connexe.*

- Puisque G est connexe, l'ensemble $\mathcal{AC}_G$ des arbres couvrants de G est un ensemble fini non-vide.
  En effet, un parcours de G en largeur permet de construire un arbre couvrant et, puisque $|S|$ est fini, l'ensemble des sous-graphes de G est fini, l'ensemble des arbres couvrants de G est donc fini.
  On peut donc considérer le nombre réel positif π défini par $\pi \overset{\text{def}}{=} \min\{f(H);\quad H\in\mathcal{AC}_G\}$ et $H_0 \in \mathcal{AC}_G$ tel que $f(H_0) = \pi$ est un arbre couvrant de poids minimal de G.
- Montrons par récurrence sur $|S|$ que H_0 est unique.
  - Si $|S| \in \{1;2\}$, le résultat est immédiat, puisque G est un arbre et qu'il est alors le seul arbre couvrant de G.
  - On considère $n \in \mathbb{N}\setminus\{0;1\}$ et le résultat vrai pour $|S| = n$.
    On considère $G = (S;A;f)$ un graphe pondéré connexe tel que $|S| = n+1$, H_0 un arbre couvrant de G de poids minimal et $s \in S$.
    L'hypothèse de récurrence assure que $G[S\setminus\{s\}]$ admet un unique arbre couvrant minimum $H' = (S\setminus\{s\};B')$.
    Si $H' \neq H_0[S\setminus\{s\}]$, on a alors :

$$f(H') < f(H_0[S\setminus\{s\}])$$

    et pour tout $t \in S$ adjacent à s :

$$f(st) + f(H') < f(H_0)$$

    En notant t_0 l'unique sommet adjacent à s dans H_0, $(S;B'\cup\{st_0\})$ serait alors un arbre couvrant de G vérifiant :

$$f((S;B'\cup\{st_0\})) < f(H_0)$$

    ce qui contredit la définition de H_0.
    On a donc $H_0 = H'$ et t_0 est nécessairement l'unique sommet de $S\setminus\{s\}$ qui vérifie :

$$f(st_0) = \min\{f(st);\quad t \text{ adjacent à } s\}$$

    H_0 est donc unique.

Q. 5 On considère a une arête de poids maximal dans un cycle $(s_0;s_1;\ldots;s_k)$ de G. On peut supposer, quitte à ré-indicer, que $a = s_0s_1$. On démontre, comme dans la question précédente, que $T^*[S\setminus\{s_0\}]$ est un arbre couvrant de poids minimal de $G[S\setminus\{s_0\}]$ et :

$$f(s_0s_{k-1}) + f(T^*[S\setminus\{s_0\}]) < f(s_0s_1) + f(T^*[S\setminus\{s_0\}])$$

ce qui assure que $s_0s_1 \notin B^*$.

Q. 6 La fonction de séparation distribue les deux premiers éléments dans la première et la seconde liste respectivement :

```
let rec separation liste = match liste with
 | []                          -> ([],[])
 | [u]                         -> ([u],[])
 | (premier::deuxieme::reste) -> let (liste1,liste2) = separation reste in
   (premier::liste1,deuxieme::liste2)
```

Q. 7 La fonction `fusion` compare, le cas échéant, les deux premiers éléments des listes passées en argument :

```
let rec fusion liste1 liste2 = match (liste1,liste2) with
 | liste,[]      -> liste
 | [],liste      -> liste
 | (a::b),(c::d) -> if a < c then a::(fusion b liste2)
                    else c::(fusion liste1 d)
```

Q. 8 La fonction `tri_fusion` est la fusion des deux listes triées obtenues par séparation et tri à partir de la liste passée en argument. La complexité de `tri_fusion` est $\mathcal{O}(n\log(n))$ où n est la longueur de la liste passée en argument.

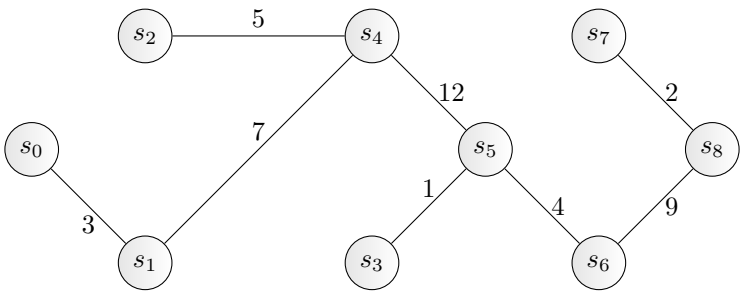
```
let rec tri_fusion liste = match liste with
 |[]  -> []
 |[e] -> [e]
 |li   -> let (liste1,liste2) = separation li in
          fusion (tri_fusion liste1) (tri_fusion liste2);;
```

II Algorithme de Kruskal inversé

Q. 9 Le principe de l'algorithme de Kruskal appliqué à un graphe pondéré connexe $G = (S;A;f)$ consiste à construire une liste ordonnée ℓ_A constituée des éléments de A par rangés par ordre croissant de poids.
On part ensuite d'un ensemble A_r initialement vide, on parcourt ensuite la liste ℓ_A en ajoutant l'élément courant à A_r si le graphe $(S;A_r)$ est sans cycle. À la fin de la boucle, on obtient un arbre couvrant de poids minimal de G.

La complexité de l'algorithme dépend de la façon dont est géré le test d'acyclicité dans la boucle. Avec une implémentation adéquate on parvient à une complexité en $\mathcal{O}(|A|\log(|S|))$.

Q. 10 L'algorithme de Kruskal inversé appliqué au graphe G_3 conduit à l'arbre suivant :



Q. 11 On peut commencer par établir la liste des éléments non triée et la trier par un tri fusion décroissant en changeant le sens de l'égalité dans la fonction `fusion` de la question **??** :

```
let tri_aretes (g:graphe) =
 let n = Array.length g in
  let rec cons liste s l = match s,l with
   | s,[] when s=n-1 -> liste
   | s,[]            -> cons liste (s+1) g.(s+1)
   | s,a::reste      -> let t,f = a in if s<t then (cons ((f,s,t)::liste) s reste)
                                      else (cons liste s reste)
 in tri_fusion (cons [] 0 g.(0))
```

Q. 12 On peut réaliser un parcours en profondeur du graphe en ne sélectionnant que des arêtes dont le poids est strictement inférieur à `poids_max`. Lors d'un parcours en profondeur, chaque arête et chaque sommet est parcourue au plus une fois, l'algorithme proposé à bien une complexité en $|S| + |A|$.

```
let connectes (g:graphe) s t poids_max =
   let vus = Array.make (Array.length g) false in
   let rec dfs u =
      if not (vus.(u)) then begin
         vus.(u) <- true;
         let rec visite_voisins liste = match liste with
            | [] -> ()
            | (v,w)::reste when w < poids_max -> dfs v; visite_voisins reste
            | _::reste -> visite_voisins reste
         in visite_voisins g.(u)
      end
   in dfs s; vus.(t)
```

Q. 13

```
let kruskal_inverse g =
   let t = Array.copy g in
   let rec aux l = match l with
      | [] -> t
      | (poids,s,t)::reste ->
         if connectes t s t poids then (
            t.(s) <- List.filter (fun (v,w) -> w < poids) t.(s);
            t.(t) <- List.filter (fun (v,w) -> w < poids) t.(t)
         );
         aux reste in
   tri_aretes g |> List.rev |> aux
```

Q. 14 « $(S;B)$ est connexe » est un invariant de boucle.
Supposons que $(S;B)$ renvoyé par l'algorithme contienne un cycle C. Alors, la première fois qu'une arête a de C a été considérée

Q. 15 On considère $G = (S;A;f)$ un graphe pondéré connexe, tel que f soit **injective** et $(S;B)$ l'arbre couvrant de G obtenu par l'algorithme de Kruskal inversé. Le résultat de la question **??** assure qu'il existe un unique arbre couvrant de poids minimal $T^* = (S;B^*)$ de G. On va montrer que $B = B^*$. Pour cela, justifions que $B^* \subset B$ est un invariant de boucle de l'algorithme.

- Avant l'entrée dans la boucle on a $B = A$. Puisque $B^* \subset A$, on a bien $B^* \subset B$ au départ.
- Supposons que $B \subset B^*$ avant la fin de boucle. Soit a l'arête supprimée par l'algorithme. Si $a \notin B^*$, $B^* \subset B$ est vérifié à l'étape suivante.
  Montrons que $a \in B^*$ est impossible.
  Si c'était le cas, en supprimant a dans $(S;B^*)$ on obtiendrait deux sous-arbres de T_1 et T_2 de T^* déconnectés. Dans l'arbre couvrant obtenu à la fin de l'algorithme, T_1 et T_2 sont nécessairement connectés : il existe s_1 un sommet de T_1 et s_2 un sommet de T_2 tels que $s_1 s_2$ soit une arête de l'arbre couvrant obtenu à la fin d'algorithme.
  Les arêtes a et $s_1 s_2$ appartiennent alors à un cycle de $(S;B)$ avant la suppression de a. En particulier :
  $$f(s_1 s_2) < f(a)$$

En supprimant a dans T^* et en ajoutant l'arête s_1s_2 on obtiendrait alors un arbre couvrant de G de poids strictement inférieur à $f(B^*)$, ce qui est absurde.

- En fin de boucle on a donc $B^* \subset B$. Par ailleurs $T^* = (S;B^*)$ et $(S;B)$ sont deux arbres couvrant de G donc :
$$|B^*| = |B| = |S| - 1$$
Finalement $B^* = B$ et l'arbre obtenu par l'algorithme de kruskal inversé est bien un arbre couvrant minimal de G.

Dans le cas général, en supposant que $|A| = p \in \mathbb{N}^*$ et en notant $A = |a_0;\dots;a_{p-1}|$, on peut montrer que pour tout $\varepsilon \in]0;+\infty[$, il existe $(\varepsilon_0;\dots;\varepsilon_{p-1}) \in [0;\varepsilon[^p$ tel que l'application g_ε définie sur A par :
$$\forall i \in [\![0;p-1]\!];\quad g_\varepsilon(a_i) = f(a_i) + \varepsilon_i$$
soit injective. Pour tout $\varepsilon \in]0;+\infty[$, l'arbre obtenu par l'algorithme de Kruskal inversé pour $(S;A;g_\varepsilon)$ est l'unique arbre couvrant minimal de $(S;A;g_\varepsilon)$. Pour $(\varepsilon_0;\dots;\varepsilon_{p-1}) = (0;\dots;0)$ c'est un arbre couvrant minimal de $(S;A;f)$.

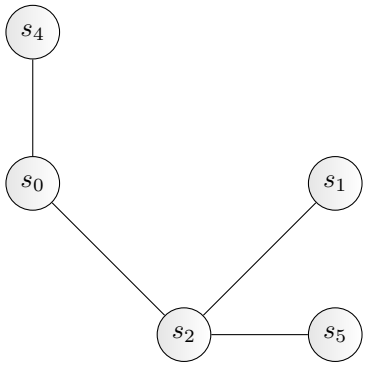
Q. 16 On peut ordonner l'ensemble des arêtes en une liste de poids décroissant avec un coût en $\mathcal{O}(|A|\log(|A|))$.
L'algorithme met en jeu une boucle de longueur $|A|$ dans laquelle on fait un test de connexité. Le test de connexité peut être mené par un parcours en largeur du graphe avec une complexité en $\mathcal{O}(|A|)$.
On obtient une complexité en $\mathcal{O}\left(|A|^2\right)$ qui est moins intéressante que dans l'algorithme classique.

On peut faire mieux qu'une complexité quadratique en adoptant un autre test de connexité, mais la comparaison des deux algorithmes reste favorable à l'algorithme classique.

III Difficulté de calcul de l'arbre optimal

Q. 17 *Le sujet comporte une erreur dans la représentation du graphe G_1.*

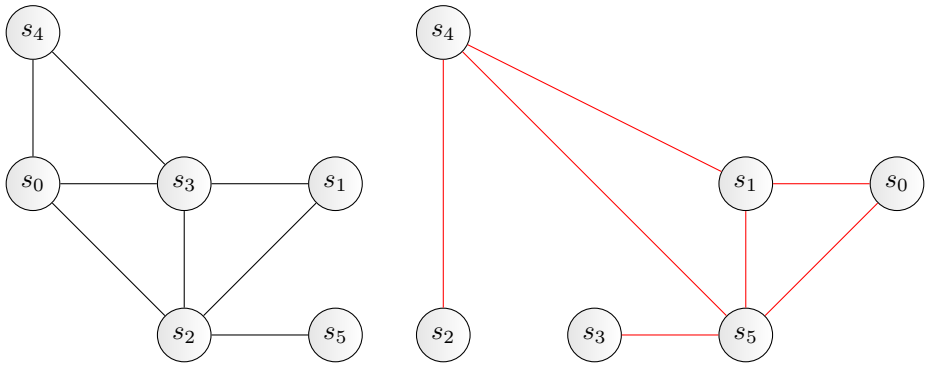
On propose l'arbre de Steiner de sommets terminaux $\{s_1;s_4;s_5\}$ dont l'ensemble des sommets de Steiner est $\{s_0;s_2\}$



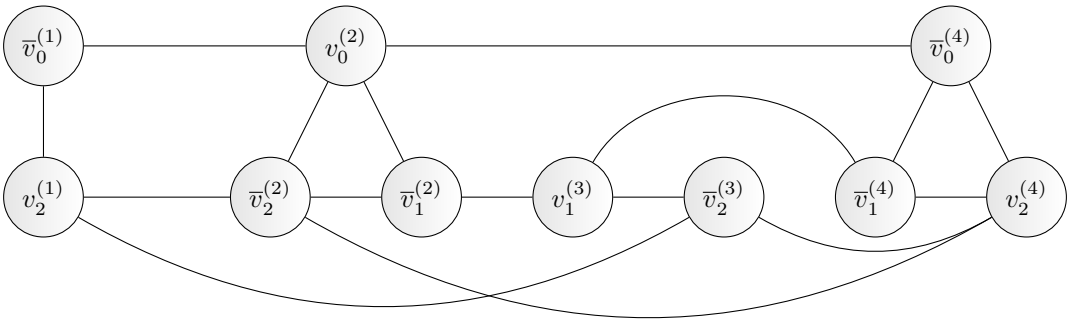
Puisque la distance de s_1 à s_4 est 2 et qu'aucun plus court chemin de s_1 à s_4 ne passe par s_5, il n'existe pas d'arbre de Steiner de sommets terminaux $\{s_1;s_4;s_5\}$ ayant moins de 1 sommet de Steiner.

III.A- De la satisfaisabilité au stable

Q. 18 On représente côte à côte les graphes G_1 et son complémentaire $\overline{G_1}$, c'est-à-dire $(S;\mathcal{P}_2(S)\backslash A)$. Un stable de G_1 est une clique de $\overline{G_1}$. Les deux stables de cardinal maximal de G_1 sont donc les ensembles $\{s_1;s_4;s_5\}$ et $\{s_0;s_1;s_5\}$.



Q. 19 Une représentation du graphe G_{φ_1} est la suivante :



Q. 20 Supposons qu'il existe un stable X de taille m dans G_φ. Puisque les littéraux apparaissant dans une même clause sont connectés, X est nécessairement de la forme $\left\{w_{i_1}^{(1)};w_{i_2}^{(2)};\ldots;w_{i_m}^{(m)}\right\}$ avec $w_{i_k}^{(k)} \in \left\{v_{i_k}^{(k)};\overline{v}_{i_k}^{(k)}\right\}$.
La formule φ est alors satisfaite par l'environnement ρ défini par :

$$\forall k \in [\![1;m]\!];\quad \rho\left(v_{i_k}\right) = \begin{cases} V & \text{si } w_{i_k}^{(k)} = v_{i_k}^{(k)} \\ F & \text{si } w_{i_k}^{(k)} = \overline{v}_{i_k}^{(k)} \end{cases}$$

Pour tout $i \in [\![1;n]\!] \setminus \{i_k;\quad k \in [\![1;m]\!]\}$, $\rho\left(v_i\right)$ est défini de façon indifférente.
Cette définition est cohérente, puisque X étant un stable de G_φ, pour tout $(k;j) \in [\![1;m]\!]^2$, on a :

- soit $i_j = i_k = i$ et $\left(w_i^{(k)};w_i^{(j)}\right) = \left(v_i^{(k)};v_i^{(j)}\right)$ ou $\left(w_i^{(k)};w_i^{(j)}\right) = \left(\overline{v}_i^{(k)};\overline{v}_i^{(j)}\right)$;
- soit $i_j \neq i_k$.

Toutes les clauses de φ sont alors satisfaites par ρ, la formule φ est donc satisfiable.

Q. 21 En considérant ρ un modèle de φ, on construit une fonction de coloration $\psi : S_\varphi \to \{\text{Noir};\text{Rouge}\}$ telle que pour tout $j \in [\![1;m]\!]$, pour tout i tel que v_i, respectivement $\neg v_i$, apparaît dans C_j, on a :

$$\psi\left(v_i^{(j)}\right) = \begin{cases} \text{Rouge} & \text{si } \rho\left(v_i\right) = V \\ \text{Noir} & \text{si } \rho\left(v_i\right) = F \end{cases}$$

respectivement :

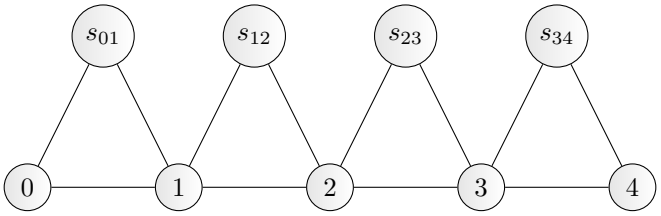
$$\psi\left(\overline{v}_i^{(j)}\right) = \begin{cases} \text{Noir} & \text{si } \rho\left(v_i\right) = V \\ \text{Rouge} & \text{si } \rho\left(v_i\right) = F \end{cases}$$

Puisque ρ est un modèle de φ, pour tout $j \in [\![1;m]\!]$, il existe un indice i tel que $\psi\left(w_i^{(j)}\right) = \text{Rouge}$.
On note que si $j \neq k$ deux noeuds adjacents de la forme $w_i^{(j)}$ et $w_i^{(k)}$ sont de couleurs différentes. Pour tout $j \in [\![1;m]\!]$, en sélectionnant i_j tel que $\psi\left(w_{i_j}^{(j)}\right) = \text{Rouge}$, l'ensemble $\left\{w_{i_j}^{(j)}\right\};\quad j \in [\![1;m]\!]$ est un stable de taille m de G_φ.

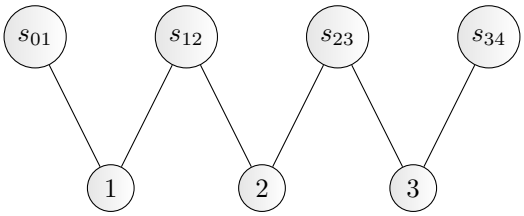
III.B- Du stable à l'arbre de Steiner

Q. 22 *La figure 6 est fausse et le résultat est évidemment faux si le graphe G n'est pas connexe. Le résultat à démontrer est lui même faux comme le prouve le contre-exemple donné dans cette correction.*

On considère la situation suivante :



$\{0;2;4\}$ est un stable de cardinal 3 du graphe G, un arbre de Steiner admettant $\{s_{01};s_{12};s_{23};s_{34}\}$ comme ensemble de sommets terminaux est :



et tous les arbres de Steiner admettant $\{s_{01};s_{12};s_{23};s_{34}\}$ comme ensemble de sommets terminaux ont au moins 3 sommets de Steiner.

Q. 23 On considère un arbre de Steiner admettant $\{s_a;\quad ai \in A\}$ comme ensemble de terminaux. Deux sommets s et t distincts de G qui ne sont pas des sommets de Steiner sont nécessairement non-adjacents sinon l'un d'entre eux au moins serait mis en jeu dans l'arbre de Steiner pour atteindre le sommet s_{st} de G'. L'ensemble des sommets de G qui ne sont pas de Steiner est donc un stable de G. Ce qui permet de conclure.

Q. 24 Les résultats des questions **??** et **??** assurent que φ est satisfiable si et seulement si G_φ admet un stable de taille m. Par construction, pour G_φ, le cardinal d'un stable est majoré par m. Avec les notations de l'énoncé, un arbre de Steiner de G'_φ admettant $\{s_a;\quad a \in A_\varphi\}$ pour ensemble de sommets terminaux, admet au moins $|S| - m$ sommets de Steiner.
En déterminant en un temps polynomial un arbre de Steiner de de G'_φ admettant $\{s_a;\quad a \in A_\varphi\}$ pour ensemble de sommets terminaux, on détermine dans le même temps si l'ensemble des sommets de Steiner est égal ou non à $|S| - m$ et on répond donc dans le même temps à la satisfiabilité de φ.

IV Approximation du problème

IV.A- Algorithme de Floyd-Warshall

Q. 25 Dans le code suivant, la fonction `f` de remplissage est itérée sur les listes d'adjacence du graphe passé en argument.

```
let adjacence (g:graphe) =
 let n = Array.length g in
  let res = Array.make_matrix n n infinity in
   let f k (s,p) = res.(s).(k) <- p; res.(k).(s) <- p
  in for i = 0 to (n-1) do
   List.iter (f i) g.(i)
  done;
 res;;
```

Q. 26 La matrice $M^{(0)}$ est simplement la matrice d'adjacence du graphe dans laquelle on a remplacé la diagonale principale par des 0.

Q. 27 Pour tout $k \in [\![0;n-1]\!]$, M_{st}^{k+1} est le poids minimal d'une chaîne de s à t dont les sommets sont inférieurs ou égaux à k. Si ce poids minimal ne met pas en jeu le sommet k on a :

$$M_{st}^{k+1} = M_{st}^{k}$$

sinon, puisque les poids sont positifs, il existe une chaîne minimale sans cycle et :

$$M_{st}^{k+1} = M_{sk}^{k} + M_{kt}^{k}$$

Finalement :

$$\forall k \in [\![0;n-1]\!]; \quad M_{st}^{k+1} = \min\left(M_{st}^{k};M_{sk}^{k} + M_{kt}^{k}\right)$$

Q. 28

```
let floydWarshall adj = let n = Array.length adj in
 let dist = Array.make_matrix n n infinity in
 let pred = Array.make_matrix n n (-1) in
 for i = 0 to n-1 do
  for j = 0 to n-1 do
   if i = j then dist.(i).(j) <- 0. else dist.(i).(j) <- adj.(i).(j)
  done;
 done;
 for k = 0 to n-1 do
  for i = 0 to n-1 do
   for j = 0 to n-1 do
    let raccourci = dist.(i).(k) +. dist.(k).(j) in
    if raccourci < dist.(i).(j) then (
     dist.(i).(j) <- raccourci ;
     pred.(i).(j) <- pred.(k).(j)
   )
    done
  done
 done;
 (dist, pred)
```

Q. 29 La complexité de `floydWarshall` est liée à la triple boucle du code de la question précédente, soit $\mathcal{O}\left(|S|^3\right)$.

Q. 30

```
let chaine_min pred s t =
 let rec chemin i j =
 match pred.(i).(j) with
  | -1 -> [i ; j]
  | k  -> (chemin i k) @ (List.tl (chemin k j))
 in chemin s t;;
```

IV.B- Solution approchée

Q. 31 Par construction $X \subset Y \subset A$ et T est un arbre qui est un sous-graphe de G puisque $A_2 \subset A$. T est donc un arbre de Steiner qui admet X comme ensemble de sommets terminaux X.

Q. 32 Le graphe obtenu par la suppression d'un sommet de degré 1 dans un arbre est encore un arbre, puisque cette action conserve la connexité et l'acyclicité.
Si on supprime un sommet de Steiner dans un arbre de Steiner, les sommets terminaux sont conservés.
La suppression des sommets de Steiner de degré 1 dans un arbre de Steiner d'ensemble de terminaux X, conduit donc à un arbre de Steiner d'ensemble de terminaux X de poids inférieur.

Q. 33 En reprenant les notations de l'énoncé, puisque T_1 est un arbre couvrant de poids minimal de H_1, on a :

$$\forall a \in B_1 \cap B; \quad f(a) = g(a)$$

Dans le graphe H_2, on a remplacé les arêtes de $B_1 \backslash A$ par des chaînes de même poids, on a donc :

$$f(H_2) \leqslant g(T_1)$$

Puisque T est un arbre couvrant minimal de H_2, on a donc en particulier :

$$f(T) \leqslant g(T_1)$$

Il suffit alors de démontrer que $g(T_1) \leqslant 2f(T^*)$ pour démontrer le résultat.
On se place dans le cas où l'ensemble des sommets terminaux n'est pas vide (l'autre cas est

trivial).
En effectuant un parcours préfixe de T^* en partant d'un sommet terminal t_0, on parcourt deux fois chacune des arêtes de l'arbre T^*, le poids du chemin parcouru est donc exactement égal à $2f(T^*)$ lorsque l'on revient au sommet t_0.
En notant $t_0;\ldots;t_{p-1}$ les p sommets terminaux distincts, rencontrés dans cet ordre lors du parcours, on peut considérer le même cycle $t_0;\ldots;t_{p-1};t_0$ parcouru dans H_1.
Puisque dans H_1 les distances entre deux sommets terminaux sont minimisées, le poids du cycle $C = t_0;\ldots;t_{p-1};t_0$ parcouru dans H_1 est inférieur au poids $2f(T^*)$ de la circum navigation de l'arbre T^*. En enlevant une arête au cycle C, on obtient un arbre T' de poids inférieur à $2f(T^*)$.
Par définition de T_1, on a $g(T_1) \leqslant g(T')$. Finalement, on a bien $g(T_1) \leqslant 2f(T^*)$.

Q. 34 On propose :

```
let renumeroter tabs =
 let n = Array.length tabs in
 let res = Array.make n (-1) in
 let rec rempli i k = match i with
  | i when i=n    -> res
  | i when tabs.(i) -> res.(i) <- k; rempli (i+1) (k+1)
  | i             -> rempli (i+1) k
 in rempli 0 0;;
```

Q. 35 *Le code suivant est un peu brutal et inélégant, il ne faut pas hésiter à faire mieux !*

On commence par coder deux fonctions.
La fonction `taille` compte le nombre de sommets terminaux

```
let taille tx =
 let n = Array.length tx in
 let rec compte k res = match k with
  | k when k=n       -> res
  | k when tx.(k)=(-1) -> compte (k+1) res
  | k                -> compte (k+1) (res+1)
 in compte 0 0;;
```

La fonction `indexe`, renvoie l'indexe d'un sommet dans une table de nombres entiers représentants les sommets d'un graphe.

```
let indexe ele tab =
let rec cherche k = if tab.(k)=ele then k else cherche (k+1) in cherche 0;;
```

On commente le code suivant en saisir les différentes parties :

```
let steiner_approche (g:graphe) tx =
 let n = Array.length g and
 di,pre = floydWarshall (adjacence g) and
 tabx = renumeroter tx in (*on renumerote tx*)
  let n1 = taille tabx in (* on prepare h1 *)
   let h1 = Array.make n1 [] in
    for i=0 to (n-1) do
     for j=(i+1) to (n-1) do
    if tab_x.(i)&& tab_x.(j)&& (i<>j) then (* on boucle sur les sommets terminaux *)
     begin
     h1.(tabx.(i)) <- (tabx.(j),di.(i).(j))::h1.(tabx.(i)); (* on adapte les numeros
         *)
     h1.(tabx.(j)) <- (tabx.(i),di.(i).(j))::h1.(tabx.(j))
     end
     done;
    done;
  let t1 = kruskal_inverse h1 in (* on cacule t1 *)
  let chem = Array.make n1 [] in (* chemin est l'ensemble des chaines entre deux
      terminaux *)
   for i=0 to (n1-1) do
    chem.(i) <- List.map (fun (s,p) -> chaine_min pre (indexe i tabx) (indexe s
```

```
      tabx)) t1.(i)
 done;
let transitions = List.concat (Array.to_list chem) in (* transitions est la liste
    de toutes les chaines *)
let sommets = List.concat ( List.map List.concat (Array.to_list chem)) (* sommets
    est l'ensemble des sommets mis en jeu dans h2 *)
in let rec numeroter liste k tab = (* il faut renumeroter les sommets de Steiner *)
match liste with
 | []                        -> tab
 | a::reste when tab.(a)=(-1) -> tab.(a) <- k; numeroter reste (k+1) tab
 | a::reste                  -> numeroter reste k tab
  in let tabx2 = numeroter sommets n1 tabx (* on construit h2 *)
   in let n2 = taille tabx2 in
  let h2 = Array.make n2 [] in
  let rec cons liste = match liste with
   | []|[_]     -> ()
   | a::b::reste -> h2.(tabx2.(a)) <- (tabx2.(b),di.(a).(b))::h2.(tabx2.(a));cons
       (b::reste)
    in List.iter cons transitions;
kruskal_inverse h2;; (* on termine avec un arbre couvrant minimal de h2 *)
```

Q. 36 *On est heureux de répondre à cette question lorsque l'on a réussi la question précédente.*