DS4 (éléments de réponse)

Exercice 1

Question 1. De manière générale, la recherche dans une ABR se fait en $O(h)$ où h est la hauteur de l'arbre. Si l'arbre est équilibré, on a $h = O(\log n)$. Si l'arbre est filiforme, on a $h = O(n)$.

Question 2. L'ordre d'insertion 1, 2, 3, 4, 5, 6, 7 produit un arbre filiforme (ou arbre peigne). L'ordre d'insertion 4, 2, 6, 1, 3, 5, 7 produit un arbre complet.

Question 3. Si $R_n = i$, par définition de R_n, le sous-arbre gauche de l'ABR considéré est un ABR aléatoire contenant $i-1$ éléments et le sous-arbre droit est un ABR aléatoire contenant $n-i$ éléments. Par conséquent $H_n = 1+\max(H_{i-1}, H_{n-i})$ puis $X_n = 2^{H_n} = 2 \times \max(X_{i-1}, X_{n-i})$.

Question 4. $Z_{n,i}$ vaut 1 pour exactement une valeur de $i \in [\![1, n]\!]$. La question précédente permet de conclure.

Question 5. On a :

$$\begin{aligned}
\mathbb{E}[X_n] &= 2\sum_{i=1}^{n} \mathbb{E}\left[Z_{n,i} \max(X_{i-1}, X_{n-i})\right] && \text{(linéarité et question 3)}\\
&= 2\sum_{i=1}^{n} \mathbb{E}\left[Z_{n,i}\right] \mathbb{E}\left[\max(X_{i-1}, X_{n-i})\right] && \text{(indépendance rappelée dans l'énoncé)}\\
&= 2\sum_{i=1}^{n} \frac{1}{n} \mathbb{E}\left[\max(X_{i-1}, X_{n-i})\right] && \text{(équiprobabilité des permutations)}\\
&\leqslant 2\sum_{i=1}^{n} \frac{1}{n} \left(\mathbb{E}[X_{i-1}] + \mathbb{E}[X_{n-i}]\right) && \text{(positivité des } X_i\text{)}
\end{aligned}$$

Finalement :

$$\mathbb{E}[X_n] \leqslant \frac{4}{n}\sum_{i=0}^{n-1} \mathbb{E}[X_i]$$

Question 6. Lorsque n est suffisamment grand, d'après l'énoncé on a :

$$\mathbb{E}[X_n] \leqslant \frac{1}{4}\begin{pmatrix} n+3 \\ n \end{pmatrix} = \frac{1}{4}\frac{(n+3)(n+2)(n+1)}{2} \leqslant Kn^3$$

où K un constante positive indépendante de n. Comme $x \mapsto 2^x$ est convexe, l'inégalité de Jensen affirme que :

$$2^{\mathbb{E}[H_n]} \leqslant \mathbb{E}\left[2^{H_n}\right] = \mathbb{E}[X_n] \leqslant Kn^3$$

et en prenant le $\log_2$ de cette inégalité on aboutit à : $\mathbb{E}[H_n] \leqslant \log_2(Kn^3) = O(\log n)$.
Ce résultat exprime que si on construit aléatoirement un ABR à n noeuds, en espérance cet arbre est équilibré.

Exercice 2

Question 1. Il s'agit de proposer un certificat. Ici, on peut proposer un sous-ensemble E des sommets de G, de taille polynomiale en $|G|$. On doit alors vérifier que $|E| \geqslant k$ et que, pour toute arête de G, ses deux extrémités sont dans E ou pas. Ces opérations se font effectivement en temps polynomial en la taille de l'entrée. Ce qui établit que Stable $\in$ NP.

Question 2. Si G possède un stable σ de taille m alors, par construction, les sommets de ce stable vérifient les propriétés suivantes.

- Un sommet exactement de σ choisi dans l'ensemble des (au plus 3) sommets correspond à une même clause car les sommets correspondant aux littéraux d'une clause sont adjacents et qu'il y a m clauses.
- Pour toute variable propositionnelle x, il n'y a jamais à la fois un sommet étiqueté par x et un sommet étiqueté par $\neg x$ dans σ puisque les sommets correspondants sont liés.

Notons v la valuation qui à chaque littéral l associe 1 si l étiquette un sommet de σ et 0 sinon. D'après les propriétés énoncées ci-dessus, cette valuation est bien définie et satisfait chacune des clauses. Donc φ est satisfiable.

La réciproque est vraie : si φ est satisfiable, il existe dans chacune des clause un littéral l_i tel que $\varphi(l_i) = 1$ et il suffit de considérer l'ensemble des sommets correspondant aux l_i dans G_φ pour exhiber un stable de taille m.

Question 3. Construire G_φ se fait en temps polynomial en $|\varphi|$. Or on vient de montrer que φ est une instance positive de 3Sat si et seulement si G_φ est une instance positive de Stable. On en déduit que 3Sat $\leq$ Stable et comme 3Sat est NP-dur, il en va de même pour Stable. Comme ce problème est aussi dans NP d'après la question 1, il est NP-complet.

Question 4. Clique est effectivement dans NP. En outre, si $G = (S, A)$ est une instance de Stable alors $G' = (S, S^2 \setminus A)$ est une instance de Clique constructible en temps polynomial en $|G|$ et G est une instance positive de Stable si et seulement si G' est une instance positive de Clique. En conséquence, Stable se réduit à Clique et comme le premier est NP-difficile, le second l'est aussi.

Exercice 3

Flots

Question 1. Il n'existe pas de flot de débit 13 car si c'était le cas, on aurait $\phi(s, 1) = 4$ et $\phi(s, 2) = 9$. Or la capacité sortante de 2 est égale à 8. Par conséquent, il ne serait pas possible de respecter la conservation du flot.

Question 2. Comme s est une source, $\phi_-(s) = 0$ car il n'existe pas d'arête (v, s). On a donc bien $|\phi| = \phi_+(s) = \Delta\phi(s)$. Par ailleurs, on remarque que $\sum_{u\in S} \phi_+(u) = \sum_{u\in S} \phi_-(u)$, chaque arête étant comptée exactement une fois dans chaque somme. On en déduit que $\sum_{u\in S} \Delta\phi(u) = 0 = \Delta\phi(s) + \Delta\phi(t)$ par conservation du flot. Ce qui établit la deuxième égalité.

Question 3. L'idée est la suivante : parcourir chaque liste d'adjacence pour modifier le tableau `dphi`. La fonction `modif_dphi` se charge de modifier, pour une arête (u, v), les cases d'indices u et v du tableau. On prend garde au fait qu'on manipule des matrices de flottants.

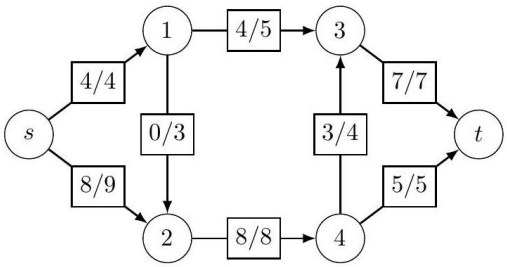
```
let delta_phi g phi =
  let n = Array.length g in
  let dphi = Array.make n 0. in
  for u = 0 to n - 2 do
    let modif_dphi v =
      dphi.(u) <- dphi.(u) +. phi.(u).(v);
      dphi.(v) <- dphi.(v) -. phi.(u).(v)
    in
    List.iter modif_dphi g.(u)
  done;
  dphi;;
```

Question 4. Il suffit de vérifier que les valeurs sont nulles, sauf pour s et t.

```
let conservation g phi =
  let n = Array.length g in
  let dphi = delta_phi g phi in
  let u = ref 1 in
  while !u < n - 1 && dphi.(!u) = 0. do incr u done;
  !u = n-1;;
```

Calcul de flot maximal

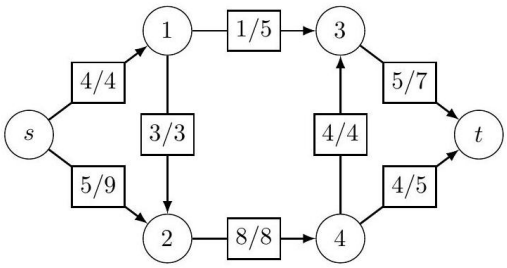
Question 5. On propose le flot suivant.



Il s'agit effectivement d'un flot maximal car $|f| = -\phi(t) = \sum_{u\in S} c(u, t) = 12$.

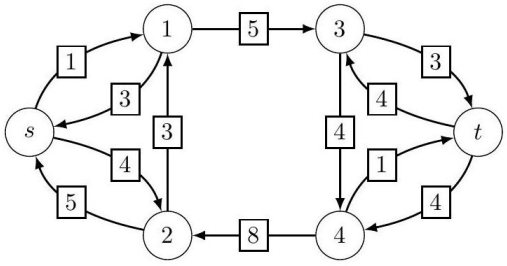
Saturation des chemins

Question 6. Le chemin $(s, 1, 3, t)$ n'est pas saturé. Il est de capacité 1. Après saturation, on obtient le flot suivant. Avec ce flot, tous les chemins de s à t sont saturés, donc l'algorithme termine. Ce flot n'est pas maximal car son débit est égal à 9 (et on a trouvé un flot de débit 12).



Algorithme de Ford-Fulkerson

Question 7. On obtient.



Question 8. On distingue les deux cas où on ajoute une arête.

```
let residuel g c phi =
  let n = Array.length g in
  let gres = Array.make n [] in
  for u = 0 to n - 2 do
    let ajout_aretes v =
      if phi.(u).(v) < c.(u).(v) then gres.(u) <- v :: gres.(u);
      if phi.(u).(v) > 0. then gres.(v) <- u :: gres.(v)
    in
    List.iter ajout_aretes g.(u)
  done;
  gres;;
```

Comme pour les fonctions précédentes, la complexité est en $O(|A| + |S|)$.

Question 9. Il s'agit d'implémenter un parcours de graphe. La fonction auxiliaire `dfs` prend en argument deux sommets. Si le sommet s n'a pas encore été vu (c'est-à-dire s'il n'a pas de parent), on lui attribue p comme parent et on relance un appel récursif sur ses voisins, s devenant l'éventuel parent de ses voisins.

```
let parcours g =
  let n = Array.length g in
  let parent = Array.make n (-1) in
  let rec dfs p s =
    if parent.(s) = -1 then begin
      parent.(s) <- p;
      List.iter (dfs s) g.(s)
    end
  in
  dfs 0 0;
  parent;;
```

On ne parcourt la liste d'adjacence d'un sommet qu'au plus une fois, ce qui garantit une complexité $O(|S|+ |A|)$ (la création de parent est en $O(|S|)$ et les opérations autre que les appels à dfs sont en temps constant).

Question 10. L'idée est de partir de t et de remonter dans l'arborescence jusqu'à s en utilisant la relation de parenté. Si t n'a pas de parent, il n'existe pas de tel chemin.

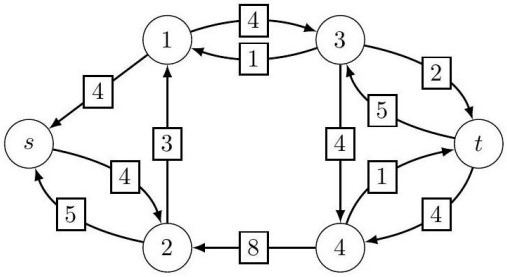
```
let chemin parent =
  let n = Array.length parent in
  if parent.(n-1) = -1 then None
  else begin
    let sigma = ref [n-1] in
    while List.hd !sigma <> 0 do
```

```
      sigma := parent.(List.hd !sigma) :: !sigma
    done;
    Some !sigma
  end;;
```

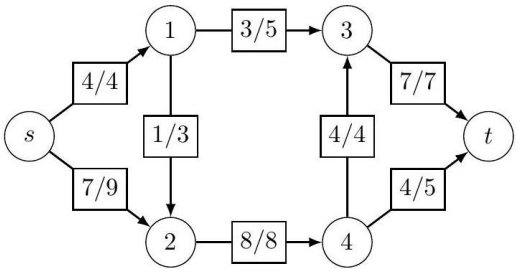
Question 11. On écrit une fonction récursive qui parcourt le chemin. Si la liste est de taille 0 ou 1, le chemin est vide et on renvoie une erreur d'après les hypothèses. Si le chemin ne contient qu'une arête (u, v), on calcule $r_\phi(u, v)$ (on teste s'il existe une arête (u, v) ou (v, u) dans le graphe de flot selon que $c(u, v)$ est strictement positif ou non). Sinon, on calcule le minimum entre la capacité résiduelle de la première arête et celle du chemin restant.

```
let rec residu sigma c phi = match sigma with
  | [] | [_] -> failwith "Chemin vide"
  | [u; v] -> if c.(u).(v) > 0.
              then c.(u).(v) -. phi.(u).(v)
              else phi.(v).(u)
  | u :: v :: q -> min (residu [u; v] c phi) (residu (v :: q) c phi);;
```

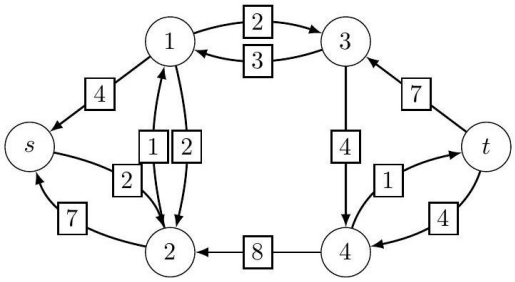
Question 12. À partir du flot obtenu précédemment, on obtient le graphe résiduel suivant.



Il existe un chemin améliorant $(s, 2, 1, 4, t)$, de capacité résiduelle 2. En améliorant le flot, on obtient :



Puis le graphe résiduel suivant.



Il existe un chemin améliorant $(s, 2, 1, 3, 4, t)$, de capacité résiduelle 1. En améliorant le flot, on obtient le flot maximal représenté question 5.

Question 13. On commence par calculer la capacité résiduelle du chemin. Ensuite, pour chaque arête du chemin, soit on augmente le flot de cette capacité résiduelle, soit on diminue le flot de l'arête inverse.

```
let ameliorer sigma c phi =
  let cres = residu sigma c phi in
  let rec amelio = function
    | [] | [_]    -> ()
    | u :: v :: q -> if c.(u).(v) > 0.
                     then phi.(u).(v) <- phi.(u).(v) +. cres
                     else phi.(v).(u) <- phi.(v).(u) -. cres;
                     amelio (v :: q)
  in
  amelio sigma;;
```

Question 14. On suit l'algorithme décrit dans l'énoncé en utilisant les fonctions précédentes.

```
let ford_fulkerson g c =
  let n = Array.length g in
  let phi = Array.make_matrix n n 0. in
  let fini = ref false in
    while not !fini do
      let gphi = residuel g c phi in
      match chemin (parcours gphi) with
        | None -> fini := true
        | Some sigma -> ameliorer sigma c phi
    done;
  phi;;
```

Question 15. Si les capacités sont entières, à chaque passage dans la boucle **tant que**, le débit du flot augmente d'au moins 1. Comme les capacités sont finies, il en est de même pour le débit du flot maximal. On en déduit qu'il y a au plus $|\phi^*|$ passages dans la boucle. Chaque passage dans la boucle s'effectue en complexité $O(|S| + |A|)$. De plus, avant la boucle, on crée une matrice de taille $|S|^2$. La complexité totale est donc en $O\left(|S|^2 + |\phi^*|\,(|S| + |A|)\right)$.

Flot maximal, coupe minimale

Question 16. On a $C(X) = c(s,2) + c(1,2) + c(3,t) = 19$.

Question 17. On a :

$$\phi(X) = \sum_{(u,v)\in A\cap(X\times\overline{X})} \phi(u,v) - \sum_{(v,u)\in A\cap(\overline{X}\times X)} \phi(u,v)$$

puis :

$$\phi(X) \leqslant \sum_{(u,v)\in A\cap(X\times\overline{X})} \phi(u,v)$$

et :

$$\phi(X) \leqslant \sum_{(u,v)\in A\cap(X\times\overline{X})} c(u,v) = c(X)$$

De plus, remarquons que si $X = \{s\}$, alors par définition, $\phi(X) = |\phi|$.
Si $X \neq \{s\}$, il existe $x \in X \setminus \{s\}$. Alors, pour calculer $\phi(X \setminus \{x\}) - \phi(X)$, il faut :

- ajouter les $\phi(u,x)$ pour $u \in X \setminus \{x\}$;
- retirer les $\phi(x,v)$ pour $v \in \overline{X}$;
- retirer les $\phi(x,u)$ pour $u \in X \setminus \{x\}$;
- ajouter les $\phi(v,x)$ pour $v \in \overline{X}$.

Or, ce qui est ajouté correspond exactement à $\phi_-(x)$ et ce qui est retiré à $\phi_+(x)$. Par la loi de conservation du flot, comme $x \notin \{s,t\}$, on a bien $\phi(X\setminus\{x\}) - \phi(X) = 0$. Par récurrence sur le cardinal de X, on en déduit que $\phi(X) = \phi(\{s\}) = |\phi|$.

Question 18. On montre ce résultat par implications successives.

- **1** $\Rightarrow$ **2**. On remarque que l'action d'amélioration d'un flot par rapport à un chemin améliorant augmente strictement le débit du flot. Si ϕ est maximal, on en déduit donc qu'il n'existe pas de chemin améliorant (sinon on pourrait trouver un flot de débit plus grand).
- **2** $\Rightarrow$ **3**. On pose X l'ensemble des sommets accessibles depuis s dans le graphe résiduel G_ϕ. Si on suppose qu'il n'existe pas de chemin améliorant pour ϕ, alors $t \notin X$.
  Par définition de X, il n'existe aucune arête de X vers $\bar{X}$ dans G_ϕ (sinon ces sommets seraient accessibles). On en déduit que pour $(u,v) \in A \cap (X \times \bar{X})$, $\phi(u,v) = c(u,v)$ (sinon l'arête (u,v) existe dans G_ϕ). De même, pour $(v,u) \in A \cap (\bar{X} \times X)$, $\phi(v,u) = 0$ (sinon l'arête (u,v) existe dans G_ϕ).
  On en déduit que $|\phi| = \phi(X) = C(X)$.
- **3** $\Rightarrow$ **1**. Par la question précédente, $|\phi| \leqslant C(X)$ pour toute coupe X. S'il existe une coupe pour laquelle c'est une égalité, alors nécessairement ϕ est maximal (car sinon il existerait un flot ϕ' tel que $|\phi'| > |\phi| = C(X)$).

Question 19. L'algorithme de Ford-Fulkerson termine lorsqu'il n'existe plus de chemin améliorant. Comme **2** $\Rightarrow$ **1** dans les équivalences précédentes, on en déduit la correction de l'algorithme.

Résolution du problème de couplage maximum

Question 20. On applique l'algorithme de Ford-Fulkerson dans R_G. Par principe de l'algorithme, on obtient un flot maximal ϕ dans R_G, dont toutes les valeurs du flot sont entières. On associe alors le couplage C défini par : $C = \{\{x, y\} \mid \phi(x, y) = 1\}$.

Question 21. Montrons que l'ensemble C défini précédemment est bien un couplage de cardinal maximum de G :

- si $\phi(x, y) = 1$, alors l'arête (x, y) existe dans R_g, donc $\{x, y\} \in A$. On a donc $C \subseteq A$;
- supposons qu'il existe deux arêtes dans C ayant le sommet $x \in X$ en commun. Alors $\phi_+(x) \geqslant 2$. Mais comme $\phi_-(x) = 1$ par construction, le sommet x ne peut pas respecter la conservation du flot. On raisonne de même si deux arêtes ont un sommet de y en commun. C est donc bien un couplage ;
- par construction, un chemin σ est augmentant pour C si et seulement si (s, σ, t) est améliorant pour ϕ. Comme il n'existe pas de chemin améliorant pour ϕ, il n'existe pas de chemin augmentant pour C. On en déduit que C est bien maximum.

De plus, la construction de R_G se fait en temps $O(|S| + |A|)$. Le débit de ϕ vaut au plus $|X| \leqslant |S|$. On en déduit une complexité totale en $O\left(|S|^2 + |S|(|S| + |A|)\right) = O(|S|(|S| + |A|))$. Cela correspond à la complexité qu'on aurait eu par l'algorithme de recherche de chemins augmentants.

Algorithme d'Edmonds-Karp

Question 22. Si on trouve $(s, 1, t)$ puis $(s, 2, t)$ comme chemins améliorants, il n'y aura que deux itérations. Si on alterne entre des chemins améliorants $(s, 1, 2, t)$ et $(s, 2, 1, t)$, alors chaque amélioration n'augmentera le débit que de 1 . Comme le débit du flot maximal est de 2000, il y aura bien 2000 itérations.

Implémentation de file

Question 23. L'opération d'enfilage ne pose pas problème car on peut facilement créer un nouveau maillon, le faire pointer vers l'actuel début, puis modifier le début vers le nouveau maillon.

L'opération de défilage est problématique car il faudrait faire pointer la nouvelle fin vers le maillon qui précède celui qui est défilé. Il est impossible de trouver ce maillon à moins de parcourir entièrement la file, ce qui donnerait une complexité de défilage linéaire.

Question 24. On propose :

```
let creer () = {debut = sentinelle; fin = sentinelle};;
```

Question 25. On vérifie si le maillon de début est la sentinelle.

```
let est_vide f = f.debut.valeur = -1;;
```

Question 26. On crée un maillon qu'on fait pointer vers la sentinelle. Si la file est vide, ce maillon devient le début et la fin, sinon il devient la fin, et on pense à faire pointer vers lui l'ancienne fin.

```
let enfiler f x =
  let m = {valeur = x; suivant = sentinelle} in
  if est_vide f then f.debut <- m else f.fin.suivant <- m;
  f.fin <- m;;
```

Question 27. On récupère le maillon de tête. Si c'est le dernier maillon de la file, on fait pointer la fin de la file vers la sentinelle. Dans tous les cas, le début de la file devient le maillon suivant de celui de tête (éventuellement la sentinelle). On renvoie finalement la valeur.

```
let defiler f =
  if est_vide f then failwith "File vide";
  let m = f.debut in
  if m.suivant.valeur = -1 then f.fin <- sentinelle;
  f.debut <- m.suivant;
  m.valeur;;
```

Question 28. Toutes ces opérations ont une complexité constante (pas de boucle, pas de récursivité).

Algorithme d'Edmonds-Karp

Question 29. On utilise la structure de file pour implémenter le parcours. On effectue ici un marquage précoce des sommets pour gagner en mémoire.

```
let parcours_largeur g =
  let n = Array.length g in
  let parent = Array.make n (-1) in
  parent.(0) <- 0;
  let f = creer () in
  enfiler f 0;
  while not (est_vide f) do
    let u = defiler f in
    let ajout v = if parent.(v) = -1 then begin
      parent.(v) <- u;
      enfiler f v
    end
    in
    List.iter ajout g.(u)
  done;
  parent;;
```

Question 30. Par définition de v, on a $d_i(u) \leqslant d_{i+1}(u)$. Dès lors, distinguons :

- si (u, v) est une arête de G_{ϕ_i}, alors $d_i(v) \leqslant d_i(u) + 1 \leqslant d_{i+1}(u) + 1 = d_{i+1}(v) < d_i(v)$, ce qui est absurde ;
- sinon, comme (u, v) est une arête de σ_{i+1}, nécessairement, l'arête (v, u) fait partie de σ_i (sinon l'arête (u, v) ne serait pas non plus dans $G_{\phi_{i+1}}$). On en déduit que $d_i(v) = d_i(u) - 1 \leqslant d_{i+1}(u) - 1 = d_{i+1}(v) - 2 < d_i(v)$, ce qui est absurde.

On arrive bien à une contradiction dans les deux cas.

Question 31. Si (u, v) est une arête critique de σ_i alors (u, v) n'est pas une arête de $G_{\phi_{i+1}}$. Pour que (u, v) devienne à nouveau une arête critique, il faut que (v, u) apparaisse sur un chemin améliorant, pour un certain $j > i$. Mais alors on a :

$$d_j(u) = d_j(v) + 1 \geqslant d_i(v) + 1 = d_i(u) + 2$$

L'inégalité découlant de la question précédente. La distance de u augmente donc d'au moins 2 à chaque fois que (u, v) devient une arête critique. Comme la distance maximale est $|S| - 1$, on en déduit le résultat.

Question 32. Par la question précédente, il y a au plus $\frac{|S|}{2}|A|$ passages dans la boucle **tant que**. Comme l'algorithme d'Edmonds-Karp n'est qu'une variante de l'algorithme de Ford-Fulkerson (le parcours en largeur reste de complexité linéaire), la complexité totale est alors :

$$O\left(\min\left(|S|^2 + |\phi^*|\,|A|, |S||A|^2\right)\right.$$