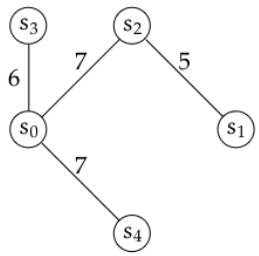
DS 3

CORRIGÉ

I Généralités

► **Question 1**



► **Question 2**
Par récurrence :
- pour $i = 0$, G_0 n'a pas d'arêtes, donc $n = n - 0$ composantes connexes ;
- si on suppose le résultat vrai pour i, alors ajouter l'arête a_{i+1} peut diminuer le nombre de composantes connexes d'au plus 1.

On conclut par récurrence. Dès lors, G possède au moins $n - k$ composantes connexes. Si G est connexe, on en déduit que $n - k \leqslant 1$, soit $|S| - 1 \leqslant |E|$.

► **Question 3** Dans la récurrence précédente, si le nombre de composantes connexes ne diminue pas lors d'une étape c'est que l'arête ajoutée relie deux sommets de la même composante et crée donc un cycle.

Par conséquent, si G est acyclique chacune des p étapes a diminué le nombre de composantes, donc G a au plus $n - p$ composantes connexes. Ce nombre de composantes valant au moins 1, on a donc $n - p \geqslant 1$, c'est-à-dire $|V| - 1 \geqslant |E|$.

► **Question 4** En notant c le nombre de composantes connexes de G, les arguments précédents montrent en fait que $n - p \geqslant c$ avec égalité si et seulement si G est acyclique. Dès lors :

$(a) \Rightarrow (b)$ G est connexe donc $c = 1$ et acyclique donc $n - p = c$, on a bien $n - p = 1$.

$(b) \Rightarrow (c)$ G est connexe donc $c = 1$ et $n - p = 1$, donc $n - p = c$ donc G est acyclique.

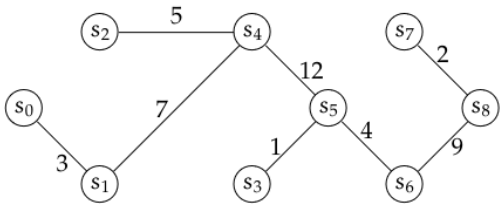
$(c) \Rightarrow (a)$ G est acyclique donc $n - p = c$ et d'autre part $n - p = 1$, donc $c = 1$ et G est connexe (et donc un arbre).

► **Question 5** Pour commencer, on remarque que $E(X)$ est non vide puisque G est connexe (s'il admet un arbre couvrant). Soient T un arbre couvrant de G et $e = xy$ l'arête de poids minimal de $E(X)$, supposons $e \notin T$. En posant $T' = T + e$, on voit que T' admet nécessairement un cycle (il a trop d'arêtes) et que ce cycle passe par e. Comme $x \in X$ et $y \in \overline{X}$, le cycle contient nécessairement au moins une autre arête de $E(X)$, que l'on note E'. On a alors $T'' = T + e - e'$ qui est connexe et vérifie $p = n - 1$, donc est un arbre couvrant. Or par définition de e et injectivité de f on a $f(e) < f(e')$, d'où $f(T'') < f(T)$, ce qui contredit le caractère minimal de T. On conclut par l'absurde.

► **Question 6** Il manque l'hypothèse G connexe, qui est bien sûr nécessaire pour que G ait un arbre couvrant. Si T est un arbre couvrant minimal, alors pour chaque arête e qu'il contient, $T - e$ possède exactement composantes connexes, qui sont donc de la forme X et $\overline{X}$ comme à la question précédente. On en déduit que e est l'arête de poids minimal de $E(X)$, et que tous les arbres couvrants minimaux contiennent e. Ainsi, tous les arbres couvrants minimaux contiennent T dans son ensemble, ce qui montre l'unicité.

2 Principe de l'algorithme de Prim

▸ **Question 7**



▸ **Question 8** En notant $T = (X, F)$, on a l'invariant « T est un arbre ». C'est vrai initialement, et à chaque étape :

- on ajoute une arête et un sommet, donc on préserve $|X| - 1 = |F|$;
- on relie le nouveau sommet à l'arbre qu'on avait, donc on garde la connexité (et donc l'acyclicité d'après la question 4).

L'étape « trouver $xy \in E$ tel que… » ne peut échouer puisque $X \neq V$ et que G est connexe, et $|V| - |X|$ fournit un variant de boucle. Donc l'algoritme termine, en renvoyant T qui est un arbre d'après l'invariant et qui a $X = V$ comme ensemble de sommet : il s'agit bien d'un arbre couvrant de G.

▸ **Question 9** Chaque arête ajoutée par l'algorithme est de poids minimal pour $E(X)$ (avec le X qu'on a à ce moment), donc appartient à l'unique arbre couvrant minimal de G. On renvoie donc un arbre couvrant inclus dans l'arbre couvrant minimal, et donc égal puisqu'ils possèdent autant d'arêtes.

▸ **Question 10** Le plus simple est de rétablir l'injectivité : il suffit de choisir une numérotation $e_1, \dots, e_p$ des arêtes et de poser $f'(e_i) = (f(e), i)$ (avec l'ordre lexicographique). Rien dans la preuve ne dépendait du fait que f soit à valeurs réelles, donc Prim renvoie l'unique arbre couvrant minimal pour f', qui est évidemment minimal (mais non nécessairement unique) pour f.

2.1 Implémentation

▸ **Question 11**

```c
int nb_edges(graph_t *g){
    int p = 0;
    for (int i = 0; i < g->n; i++){
        p += g->degrees[i];
    }
    return p;
}
```

▸ **Question 12** Pour insérer un élément dans un tas binaire, on commence par créer une nouvelle feuille puis l'on effectue une percolation vers le haut pour rétablir la propriété d'ordre des tas, ce qui se fait en temps linéaire en la hauteur, et donc en $O(\log n)$ puisque l'arbre est complet. Pour extraire le minimum, on remplace la racine par la dernière feuille puis l'on effectue une percolation vers le bas, à nouveau en temps $O(\log n)$.

▸ **Question 13** Le tableau de booléens `covered` code l'ensemble X, et la file contient les arêtes ayant au moins une extrémité dans X qui n'ont pas encore été traitées. Á chaque étape, on extrait l'arête de poids minimal de la file, on l'ignore si sa deuxième extrémité est dans X, et sinon on ajoute son autre extrémité y à X et les arêtes incidentes à y à la file (sauf celles dont l'autre extrémité est dans X).

```c
void add_edges(graph_t *g, heap_t *heap, bool *covered, int x){
    int degree = g->degrees[x];
    for (int i = 0; i < degree; i++){
        edge_t e = g->adj[x][i];
        if (!covered[e.y]) heap_push(heap, e);
    }
}

edge_t *prim(graph_t *g){
    edge_t *selected = malloc((g->n - 1) * sizeof(edge_t));
    int nb_selected = 0;

    bool *covered = malloc(g->n * sizeof(bool));
    for (int i = 0; i < g->n; i++) covered[i] = false;

    heap_t *heap = heap_create(nb_edges(g));

    covered[0] = true;
    add_edges(g, heap, covered, 0);

    while (!heap_is_empty(heap)){
        edge_t e = heap_extract_min(heap);
        if (covered[e.y]) continue;
        covered[e.y] = true;
        selected[nb_selected] = e;
        nb_selected++;
        add_edges(g, heap, covered, e.y);
    }

    free(covered);
    heap_free(heap);
    return selected;
}
```

Pour la complexité, on a, en notant $n = |V|$ et $p = |E|$:

- une initialisation en $O(n)$ (pour `covered`) plus $O(p)$ (pour la file) ;
- une file dont la taille est majorée par p ;
- chaque arête est insérée et extraite une fois de la file, en temps $O(\log p)$, donc $O(P \log p)$ au total.

On obtient donc du $O(n + p + p \log p)$. On remarque alors que $n = O(p)$ (puisque G est connexe) et $\log p = O(\log n)$ (puisque $p \leqslant n^2$). On a donc bien du $O(p \log n)$.

▸ **Question 14** Dans l'algorithme de Kruskal, on part d'un graphe H contenant tous les sommets de V et aucune arête puis l'on considère les arêtes de E une par une par poids croissant. Pour chaque arête, si elle relie dans sommets de deux composantes connexes distinctes de H on l'ajoute, sinon on ne l'ajoute pas. En utilisant une structure UnionFind pour les composantes connexes, le coût des tests et des fusions est en $o(p \log n)$, et le coût du tri initial est en $O(p \log p)$ pour un tri par comparaison. On obtient donc du $O(p \log p) = O(p \log n)$ au total, comme pour Prim.

On peut cependant remarquer que si l'on dispose déjà des arêtes triées, ou si l'on s'autorise à sortir des tris par comparaison (tri radix sur des entiers machine ou des flottants par exemple), la complexité devient dominée par celle des tests et fusions, et est strictement meilleure que celle de Prim.

3 Application au problème du goulot maximal

▸ **Question 15** On obtient $s_0, s_2, s_1, s_4, s_7, s_6, s_8$, de capacité 7.

▸ **Question 16** Soit T un arbre couvrant maximal de G et λ le chemin élémentaire de s à t dans T. Supposons que ce chemin ne soit pas un goulot maximal de s à t dans G, et soit alors $e \in \lambda$ telle que $f(\lambda) < c^*(s,t)$. $T - e$ possède deux composantes connexes X et $\overline{X}$, et un goulot maximal de s à t dans G contient forcément une arête $e' \in E(X)$. On a nécessairement $f(e') \geqslant c^*(s,t)$, donc $f(e') > f(e)$ et $f(T - e + e') < f(T)$. Or $T - e + e'$ est un arbre couvrant (l'arête e' ne crée pas de cycle puisqu'elle est dans $E(X)$), donc c'est absurde. On conclut que λ est un goulot maximal.

▸ **Question 17** On peut remplacer le tas min par un tas max dans Prim, ou simplement exécuter Prim sur $G' = (V, E, -f)$.

▸ **Question 18** On effectue un parcours à base de pile, mais un parcours en profondeur récursif aurait bien sûr été possible en écrivant une fonction auxiliaire.

```c
int *parent(graph_t *g, int u){
    int *t = malloc(g->n * sizeof(int));
    for (int i = 0; i < g->n; i++) t[i] = -1;
    int *stack = malloc(g->n * sizeof(int));
    int top = 0;
    stack[0] = u;
    while (top != -1) {
        int x = stack[top];
        top--;
        for (int i = 0; i < g->degrees[x]; i++){
            int y = g->adj[x][i].y;
            // On a un arbre mais le test suivant est quand même
            // nécessaire pour ne pas reprendre la même arête en sens
            // inverse.
            if (t[y] != -1) continue;
            t[y] = x;
            top++;
            stack[top] = y;
        }
    }
    free(stack);
    return t;
}
```

On a une initialisation en $O(n)$ puis on traite chaque arc du graphe en temps constant, ce qui donne du $O(n+p)$. Comme G est connexe (c'est un arbre), $n = O(p)$ et l'on a en fait du $O(p)$.

▸ **Question 19** L'énoncé donne un argument de type `graphe`, c'est une erreur : on utilisera un `graph_t*`. Autre problème : il faut libérer l'arbre en fin de fonction, et l'on ne dispose pas de fonction pour le faire. C'est un oubli de l'énoncé, mais on peut facilement en écrire une. Pour la fonction en elle-même, on calcule l'arbre, on l'enracine en t et l'on parcourt la branche de s à la racine. On alloue un tableau de taille n, ce qui sera toujours suffisant puisque le chemin est de longueur au plus $n-1$.

```c
void graph_free(graph_t *g){
    free(g->degrees);
    for (int i = 0; i < g->n; i++) {
        free(g->adj[i]);
    }
    free(g->adj);
    free(g);
}
```

```c
int *goulot_max(graph_t *g, int s, int t, int* lc){
    graph_t *tree = mst(g);
    int *uptree = parent(g, t);
    int *path = malloc(g->n * sizeof(int));
    *lc = 0;
    int x = s;
    while (x != -1){
        path[*lc] = x;
        *lc = *lc + 1;
        x = uptree[x];
    }
    free(uptree);
    graph_free(tree);
    return path;
}
```

4 Interlude : médiane en temps linéaire

▶ Question 20

```
let rec separe u seuil =
  match u with
  | [] -> ([], [], 0, 0)
  | x :: xs ->
    let petits, grands, nb_petits, nb_egaux = separe xs seuil in
    if x < seuil then (x :: petits, grands, 1 + nb_petits, nb_egaux)
    else if x > seuil then (petits, x :: grands, nb_petits, nb_egaux)
    else (petits, grands, nb_petits, nb_egaux + 1)
```

▶ Question 21

```
let rec medianes_5 u =
  match u with
  | [] -> []
  | a :: b :: c :: d :: e :: xs ->
    mediane_bloc [a; b; c; d; e] :: medianes_5 xs
  | _ -> [mediane_bloc u]
```

▶ Question 22 La fonction complétée :

```
let rec select u k =
  let n = List.length u in
  if n = 1 then List.hd u
  else
    let liste_medianes = medianes_5 u in
    let nb_medianes = 1 + (n - 1) / 5 in
    let mom = select liste_medianes (nb_medianes / 2) in
    let petits, grands, nb_petits, nb_egaux = separe u mom in
    if k < nb_petits then select petits k
    else if k < nb_petits + nb_egaux then mom
    else select grands (k - nb_petits - nb_egaux)
```

En considérant que dans la liste triée on aurait `nb_petits` éléments inférieurs strictement à `mom`, puis `nb_egaux` copies de `mom`, puis des éléments strictement supérieurs à `mom`, on a un cas si l'élément cherchée est dans la partie de gauche, un s'il est dans celle du milieu et un s'il est dans celle de droite (et l'on cherche alors son indice *dans la partie de droite*).

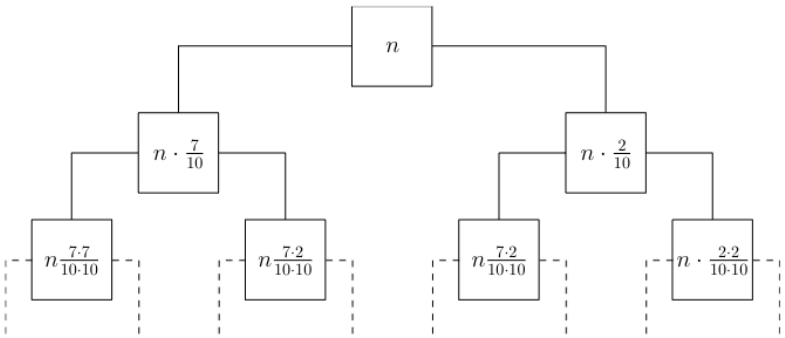
▸ **Question 23** On effectue :

- un appel à `medianes_5`, en temps $O(n)$;
- un appel à `separe` sur une liste de taille $n/5$, donc en $O(n)$;
- un appel récursif sur une liste de taille $n/5$, donc en temps $T(n/5)$;
- potentiellement un appel récursif sur `petits` ou `grands`.

Il reste donc à justifier que $|\texttt{petits}| \leqslant 7n/10$ et $|\texttt{grands}| \leqslant 7n/10$. On suppose pour simplifier que n est divisible par 5 et l'on note $k = n/5$ et $m_0, \dots, m_{k-1}$ les médianes des blocs de taille 5. $\texttt{mom}$ est la médiane des m_i (d'où son nom : *Median Of Medians*), donc inférieure ou égale à au moins $k/2$ d'entre eux. Chacun de ces m_i est lui-même inférieur ou égal à au moins 3 éléments du bloc. Donc $\texttt{mom}$ est inférieure ou égale à au moins $3k/2 = 3n/10$. On en déduit que $|\texttt{grands}| \leqslant 7n/10$, et le même raisonnement prouve la même chose pour $|\texttt{petits}|$.
Finalement, on a bien $T(n) \leqslant T(n/5) + T(7n/10) + An$.

▸ **Question 24** Deux démonstrations possibles (au choix) :

Avec l'arbre d'appel On a représenté ci-dessous les premiers niveaux de l'arbre d'appel, en mettant dans chaque nœud la taille de l'argument :



D'après la formule obtenue à la question précédente, le travail effectué à chaque niveau de l'arbre est majoré par A fois la somme des tailles. Or on voit facilement que cette somme des tailles est multipliée par $9/10$ à chaque niveau : chaque fils gauche vaut $7/10$ de son père et chaque fils droit $2/10$. En notant h la hauteur de l'arbre (qu'il est inutile de déterminer), le travail total vaut alors :

$$\begin{aligned}\sum_{k=0}^{h} An(9/10)^k &= An\sum_{k=0}^{h}(9/10)^k \\ &= O(n) && \text{puisque la somme est bornée.}\end{aligned}$$

Par récurrence Soit $n > 1$, supposons qu'il existe une constante réelle B telle que $T(k) \leqslant Bk$ pour tout $k < n$. On a alors $T(n) \leqslant B\frac{n}{5} + B\frac{7n}{10} + An \leqslant n\left(A + \frac{9}{10}B\right)$. On a l'hérédité à condition que $A + \frac{9}{10}B \leqslant B$, ce qui équivaut à $B \geqslant 10A$.
D'autre part, l'initialisation pour $n = 1$ fonctionne si $B \geqslant T(1)$. On pose donc $B = \max(T(1), 10A)$ et l'on a le résultat souhaité par récurrence.

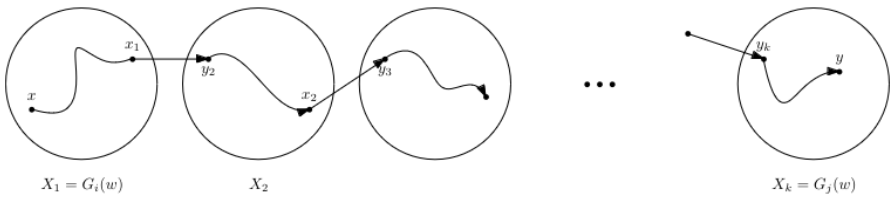
5 Goulot maximal en temps linéaire

5.1 Étude théorique

▸ **Question 25** Il suffit d'effectuer un parcours de graphe depuis x en ignorant les arêtes de poids strictement inférieur à $c^*(G, x, y)$ et de reconstruire le chemin par un moyen quelconque.

▸ **Question 26**

- Si x et y sont dans la même composante, alors il existe un chemin à l'intérieur de cette composante de capacité au moins w. Comme tout chemin sortant de la composante a une capacité strictement inférieure, on peut ne garder que la composante (et que les arêtes de poids au moins w).
- Sinon, on procède par double inégalité, en se basant sur le schéma ci-dessous :



  — Soit $\lambda = (X_1, \dots, X_k)$ un chemin de $G_i(w) = X_1$ à $G_j(w) = X_k$ dans $\overline{G}(w)$. Ce chemin correspond à une suite d'arêtes $x_i y_{i+1}$ avec $x_i \in X_i$ et $y_{i+1} \in X_{i+1}$ dans G, et sa capacité est $c(\lambda) = \min(f(x_i y_{i+1}) \mid 1 \leqslant i < k) < w$ par définition de $\overline{G}(w)$. Pour chaque i, on peut trouver un chemin λ_i de y_i à x_i de capacité au moins w dans G, et donc étendre λ en un chemin λ' de x à y dans G tel que $c(\lambda') = c(\lambda)$. On a donc montré $c^*(\overline{G}(w), G_i(w), G_j(w)) \leqslant c^*(G, x, y)$.

  — Inversement, en partant d'un chemin λ de x à y dans G, on peut considérer la suite des composantes connexes $G_i(w) = X_1, \dots, X_k = G_j(w)$ traversées. L'arête prise pour passer de X_i à X_{i+1} est de poids au plus $f_{\overline{G}(w)}(X_i X_{i+1})$, donc il existe un chemin de capacité au moins $c(\lambda)$ de $G_i(w)$ à $G_j(w)$ dans $\overline{G}(w)$, ce qui montre l'autre inégalité.

▸ **Question 27** On propose l'algorithme suivant :

Algorithme 1 – Capacité maximale en temps linéaire

Entrées : Un graphe pondéré connexe $G = (V, E, f)$, deux sommets distincts x et y.
Sorties : $c^*(G, x, y)$

```
fonction Capacité(G, x, y)
    si |V| = 2 alors renvoyer f(xy)
    w ← med(f(e) | e ∈ E)
    Calculer les G_i(w) = (V_i, E_i, f)
    Soient i tel que x ∈ V_i, j tel que y ∈ V_j
    si i = j alors
        renvoyer Capacité(G_i(w), x, y)
    sinon
        Calculer Ḡ(w)
        renvoyer Capacité(Ḡ(w), G_i(w), G_j(w))
```

Pour la correction partielle :

- le cas de base est correct, $|V| = 1$ est impossible puisque $x \neq y$ par hypothèse, et dans le cas $|V| = 2$ il n'y a qu'un seul chemin ;
- les égalités utilisées dans les appels récursifs sont celles prouvées à la question précédente ;
- G_{bar} est connexe puisque G l'était et chacune des $G_i(w)$ est connexe par définition, donc les appels récursifs se font bien sur des graphes connexes ;
- de plus, les appels récursifs se font bien sur des sommets distincts.

Pour la complexité (qui donnera aussi la terminaison), on commence par remarquer que $|V| = O(|E|)$ comme G est connexe, on ne garde donc que $p = |E|$ comme paramètre. Il est clair que le calcul des $G_i(w)$ et de $\overline{G}(w)$ peut se faire en temps $O(n+p) = O(p)$ (c'est une variante du calcul des composantes connexes). Le calcul de w est en temps linéaire si l'on utilise l'algorithme de la partie précédente. $\overline{G}(w)$ contient au plus $\lfloor p/2 \rfloor$ arêtes puisque w est la médiane. L'ensemble des $G_i(w)$ contient au plus $\lceil p/2 \rceil$ arêtes (on utilise ici l'injectivité de f), donc dans les deux cas on fait un appel récursif sur un graphe ayant au plus $p/2$ arêtes, à un près. On a donc $T(p) \leqslant T(p/2) + Ap$ avec A une constante. La même méthode qu'en question 24 permet de conclure que $T(p) = O(p)$, ce qui est le résultat cherché.

▶ **Question 28** Non, si l'on ne garde que les arcs de poids au moins w et qu'on considère les composantes fortement connexes, deux sommets x et y peuvent se retrouver dans deux composantes distinctes alors qu'il y a un chemin de x à y de capacité supérieure ou égale à w (et ce même si l'on ajoute l'hypothèse que le graphe initial est fortement connexe).

5.2 Implémentation

▶ **Question 29**

```
let construit_graphe aretes =
  let g = Hashtbl.create 1 in
  let traite_arete (x, y) w =
    match Hashtbl.find_opt g x with
    | None -> Hashtbl.replace g x [(y, w)]
    | Some l -> Hashtbl.replace g x ((y, w) :: l)
  in
  Hashtbl.iter traite_arete aretes;
  g
```

▶ **Question 30**

```
let calcule_composantes g w =
  let t = Hashtbl.create 1 in
  let i_composante = ref 0 in
  let rec explore v =
    match Hashtbl.find_opt t v with
    | None ->
      Hashtbl.replace t v !i_composante;
      let aretes = Hashtbl.find g v in
      List.iter (fun (v', w') -> if w' >= w then explore v') aretes
    | _ -> ()
  in
  let traite v _ =
    if not (Hashtbl.mem t v) then (
      explore v;
      incr i_composante
    )
  in
  Hashtbl.iter traite g;
  t
```

On utilise le dictionnaire associant un sommet à sa composante comme ensemble de sommets visités, et un compteur qu'on incrémente à chaque nouvelle composante.

▶ **Question 31**

```
let calcule_g_bar (g : pondere) t : pondere =
  let aretes = Hashtbl.create 1 in
  let traite_sommet x voisins =
    let cx = Hashtbl.find t x in
    let traite_arete (y, w) =
      let cy = Hashtbl.find t y in
      if cx <> cy then
        match Hashtbl.find_opt aretes (cx, cy) with
        | Some w' when w' >= w -> ()
        | _ ->
          Hashtbl.replace aretes (cx, cy) w;
          Hashtbl.replace aretes (cy, cx) w;
    in
    List.iter traite_arete voisins
  in
  Hashtbl.iter traite_sommet g;
  construit_graphe aretes
```

On commence par créer un dictionnaire d'arêtes (de type `ensemble_aretes`). Pour ce faire, pour chaque arête, on vérifie si elle relie deux sommets de deux composantes distinctes. Si c'est le cas, on crée une arête de $\overline{G}(w)$ ou met à jour son poids. On termine en convertissant en `pondere`.

Remarque

L'argument w donné par l'énoncé était inutile, on l'a omis ici.

▶ **Question 32**

```
let extrait_composante g t i =
  let aretes = Hashtbl.create 1 in
  let traite_sommet x voisins =
    let cx = Hashtbl.find t x in
    let traite_arete (y, w) =
      let cy = Hashtbl.find t y in
      if cx = cy then
        match Hashtbl.find_opt aretes (x, y) with
        | Some w' when w' >= w -> ()
        | _ -> Hashtbl.replace aretes (x, y) w
    in
    if cx = i then List.iter traite_arete voisins
  in
  Hashtbl.iter traite_sommet g;
  construit_graphe aretes
```

Cette fois, on ne garde que les arêtes entre sommets de la composante i.

▶ **Question 33**

```
let poids_median g =
  let liste_poids = ref [] in
  let traite _ l = List.iter (fun (_, w) -> liste_poids := w :: !liste_poids) l in
  Hashtbl.iter traite g;
  select !liste_poids (List.length !liste_poids / 2)
```

▶ **Question 34** Il ne reste qu'à transcrire l'algorithme de la question 27.

```
let rec capacite_max g u v =
  if Hashtbl.length g = 2 then (
    match Hashtbl.find g u with
    | [(x, w)] when x = v -> w
    | _ -> assert false
  ) else (
    let w_limite = poids_median g in
    let t = calcule_composantes g w_limite in
    let cu = Hashtbl.find t u in
    let cv = Hashtbl.find t v in
    if cu = cv then capacite_max (extrait_composante g t cu) u v
    else capacite_max (calcule_g_bar g t) cu cv
  )
```