Corrigé du concours blanc 2025

① 1) Supposons que G est récursive et donc qu'il existe un symbole non terminal X tel que $X \Rightarrow^* \alpha X \beta$ avec α et β composés à la fois de symboles terminaux et non terminaux.

Soit $d \in \mathbb{N}$ qcq, on peut construire une dérivation qui enchaîne d fois la dérivation $X \Rightarrow^* \alpha X \beta$ et on obtient $X \Rightarrow^+ \alpha^d X \beta^d$ avec la portion de branche correspondant à l'enchaînement des X de longueur au moins d. Ainsi l'arbre d'analyse ainsi construit est de hauteur au moins d.

Soit G non récursive, si on considère une branche d'un arbre d'analyse alors tous les symboles non terminaux qu'elle contient sont $\neq$ et elle contient au plus un symbole terminal donc la hauteur de l'arbre est majorée par le nb de symboles non terminaux plus 1 et la grammaire est bien d-bornée pour $d = \#0 + 1$.

2) En prenant la numérotation suivante :

A : 0	B : 1
a : 2	b : 3
S : 4	

let g = ([(4, [0; 1]); (0, [1; 1]); (1, [3; 2])], 4, 5)

3)

3) let terminaux g =

```
let r, s, taille = g in
let res = Array.make taille true in
res.(s) ← false;
let apply (a, l) = res.(a) ← false in
List.iter apply r;
res
```

4) let rec est_unaire t (x,l) = [(List.length l = 1) && (t.(List.hd l) = false)] || [List.forall (fun x → t.(x)) l]

(inutile)

on peut donc simplifier en:

```
let rec est_unaire t (x, l) =
(List.length l = 1) || (List.forall (fun x → t.(x)) l)
```

```
let g_est_unaire g =
    let r, s, t = g in
    let tableau = terminaux g in
    List.forall (r_est_unaire tableau) r
```

5)
```
let x_est_linéaire t (x,l) =
    let rec aux liste cpt = match liste with
        |[] -> cpt<= 1
        |h::next -> if (t.(h)) then aux next cpt
                    else aux next (cpt+1)
    in aux l 0

let g_est_linéaire g =
    let x,s,t = g in
    let tab= terminaux g in
    List.forall (x_est_linéaire tab) x
```

6)
```
let construct g =
    let x,s,t = g in
    let graphe = ARRAY.make t [] in
    let traite (x,l) =
        List.iter (fun i ->(graphe.(x)<- i::graphe.(x))) l
    in
    List.iter traite x;
    graphe
```

7) A l'aide d'un parcours en profondeur, on peut détecter l'existence d'un cycle dans un graphe orienté.
On obtient alors le code suivant:

```
exception Cycle
let detect_cycle g =
    let n = Array.length g in
    let dec = Array.make n false in
    let traite = Array.make n false in
    let rec parcours s =
        if (not (dec.(s))) then
          begin
          dec.(s) <- true;
          if (List.exists (fun i -> dec.(i) && not (traite.(i)))
                          g.(s))
            then raise Cycle;
          List.iter parcours g.(s);
          traite.(s) <- true;
          end
    in try for i = 0 to n-1 do
          parcours i;
       done;
       false
    with | Cycle -> true
```

```
let est_non_rec g = let graphe = construct g in
                    not (detect_cycle graphe)
```

8) On écrit d'abord une fonction qui gère la liste des résultats de la dérivation d'un non terminal.

```
let rec derivation h r = match r with
  |[] -> []
  |(x,l)::next when x = h -> l::(derivation h next)
  |_::next -> derivation h next

let est_successeur_immédiat g mot =
  let r,s,t = g in
  let tab = terminaux g in
  let rec aux mot = match mot with
    |[] -> []
    |h::t when tab(h) -> let liste = aux t in
                         List.map (fun l -> h::l) liste
    |h::t -> let deriv = derivation h r in
             let liste = aux t in
             (List.map (fun x -> x@t) deriv)
             @
             (List.map (fun l -> h::l) liste)
  in aux mot
```

9) Il suffit de faire un parcours en profondeur avec une garantie d'arborescence grace à la non récursivité qui garantit l'arrêt.

```
let est_produit g mot =
    let t = terminaux g in
    let liste_mots = ref [] in
    let rec aux mot_c =
        let liste_voisins = est_successeur_immediat
                            g mot_c in

        List.iter (fun m -> if (List.for_all
        (fun x -> t(x)) m) then
        liste_mots := m::(!liste_mots)
        else aux m) liste_voisins;

    in aux mot;
    !liste_voisins.
```

10) Si la grammaire est non récursive alors le nombre de mots générés est fini et donc le nombre d'états du jeu est bien fini.

(11) $Att_0(J) = \{((m,p), j) \quad m \in L$ et p et j q.c.q.s $\}$

$\forall n \geq 0$, $Att_{n+1}(J) = Att_n(J) \cup$

$\{((m,p), j) : j = 0$ et $\exists p' \in [\![0, |m|+1]\!]$ t.q. $((m,p'), 1) \in Att_n(J)\} \cup$

$\{((m,p), j) : j = 1$ et $\forall m'$ t.q. $m \Rightarrow m'$ en dérivant la position p, on a $((m', p), 0) \in Att_n(J)\}$

(12)

```
let attr_j init =
  let tab = Hashtbl.create() in
  let rec aux (m,p), j =
    if (not (Hashtbl.mem tab (m,p), j)) then
      begin
      if (j=0) then
        begin
        let rec aux_0 i =
          if (i = List.length m) then false
          else if aux ((m,i), 1) then true
          else aux_0 (i+1)
        in
        Hashtbl.add tab ((m,p), j) (aux_0 0)
        end
      else
        begin
        let l = successeurs_immediats g m in
        let rec aux_1 l = match l with
          [] -> true
          | h::n when (not (aux(h,p), 0)) -> false
          | h::n -> aux_1 n
        in
        Hashtbl.add tab ((m,p), j) (aux_1 l)
        end
      end
    Hashtbl.find tab ((m,p), j)
in aux init;
tab
```

2) Graphes de flots

① Soit $(u,v) \in S^2$ alors $f(u,v) = -f(u,v)$

Supposons que $f(u,v) > 0$ alors le respect de la capacité garantit que $0 < f(u,v) \leqslant c(u,v)$ et si $(u,v) \notin A$ on aurait $c(u,v) = 0$ ce qui est contradictoire donc $(u,v) \in A$.

Si $f(u,v) < 0$ alors $-f(u,v) = f(v,u) > 0$ et donc $(v,u) \in A$ d'après ce qui précède.

② Supposons que $u \in S \setminus \{s,t\}$ alors

$$\begin{aligned}\phi(u) &= \phi_+(u) - \phi_-(u) \\ &= \sum_{v : f(u,v) > 0} f(u,v) - \sum_{v : f(u,v) < 0} f(v,u) \\ &= \sum_{v : f(u,v) > 0} f(u,v) + \sum_{v : f(u,v) < 0} f(u,v) \quad \text{(antisymétrie)} \\ &= \sum_{v \in S} f(u,v)\end{aligned}$$

On a donc bien $\phi(u) = 0$ ssi $\sum_{v \in S} f(u,v) = 0$

③ Comme s est une source alors $\phi_-(s) = 0$
Comme t est un puit alors $\phi_+(t) = 0$
donc $\boxed{|f| = \phi(s) = \phi_+(s)}$

De plus, la q°2 garantit que :

$$\sum_{u \in S} \phi(u) = \phi(s) + \phi(t)$$

car en dehors des termes $u = s$ et $u = t$ ils sont tous nuls

et d'autre part :

$$\sum_{u \in S} \phi(u) = \sum_{u \in S} \phi_+(u) - \sum_{u \in S} \phi_-(u)$$

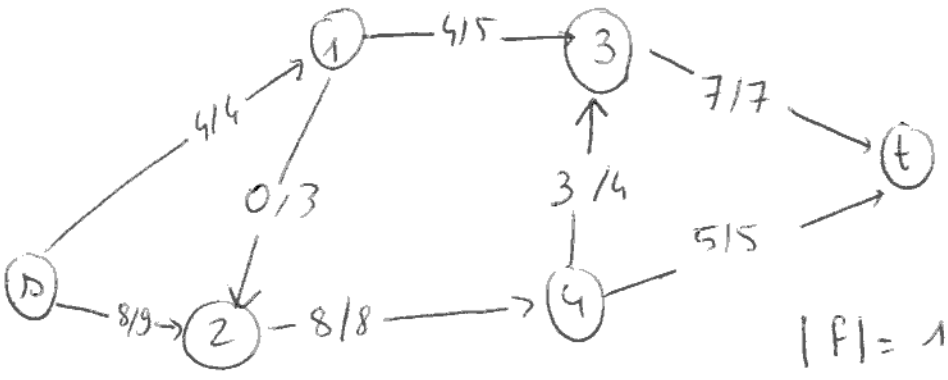
$$= \sum_{\substack{u,v: \\ f(u,v)>0}} f(u,v) - \sum_{\substack{u,v: \\ f(u,v)<0}} f(v,u)$$

$$= \sum_{\substack{u,v: \\ f(u,v)>0}} f(u,v) - \sum_{\substack{u,v: \\ f(v,u)>0}} f(v,u)$$

$$= 0$$

donc $\boxed{|f| = \phi(s) = -\phi(t)}$

(4)



$|f| = 12$

Le flot est maximal car la somme des débits entrants en t vaut 12 et est une borne supérieure sur la valeur cherchée.

(5) On construit un graphe avec les mêmes sommets où on ne garde que les arêtes non saturées. Si dans ce graphe, on trouve un chemin à l'aide d'un parcours de s à t alors ce chemin est un chemin non saturé et dans le cas contraire il n'existe pas de chemin saturé.

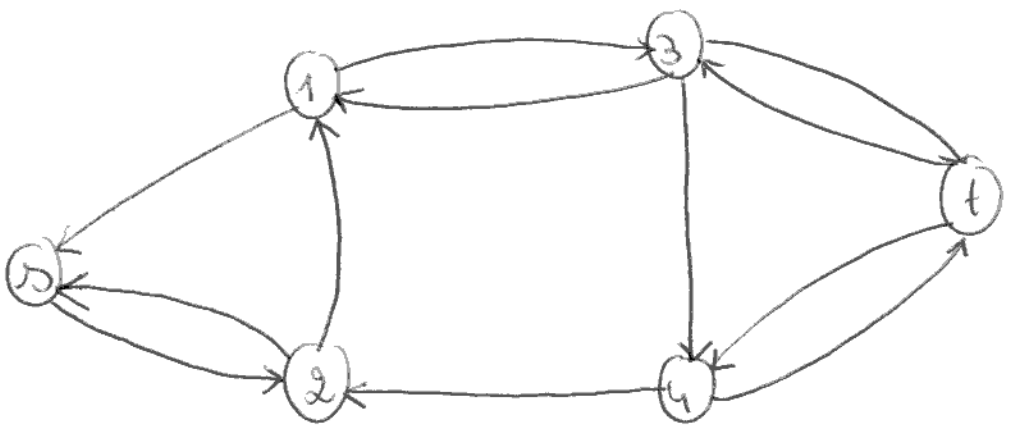
(6) Le seul chemin non saturé est $(s, 1, 3, t)$ que l'on peut augmenter de 1. Ceci termine l'algo et le débit obtenu ne vaut que 9, il n'est donc pas maximal.

(7) $(u,v) \in A_f$ ssi $c(u,v) > f(u,v)$

ssi $f(u,v) < 0$ ou $c(u,v) > f(u,v) > 0$

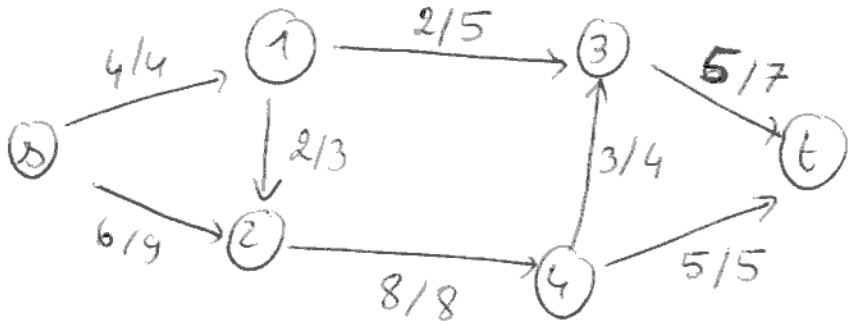
et donc $\begin{cases} (v,u) \in A \text{ dans le } 1^e \text{ cas} \\ \text{et } (u,v) \in A \text{ dans le second.} \end{cases}$

(8)

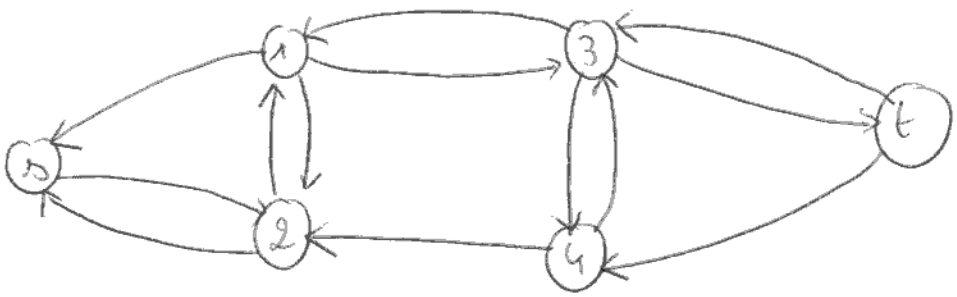


(9) Il y a deux chemins améliorant : $(s, 2, 1, 3, t)$
et $(s, 2, 1, 3, 4, t)$

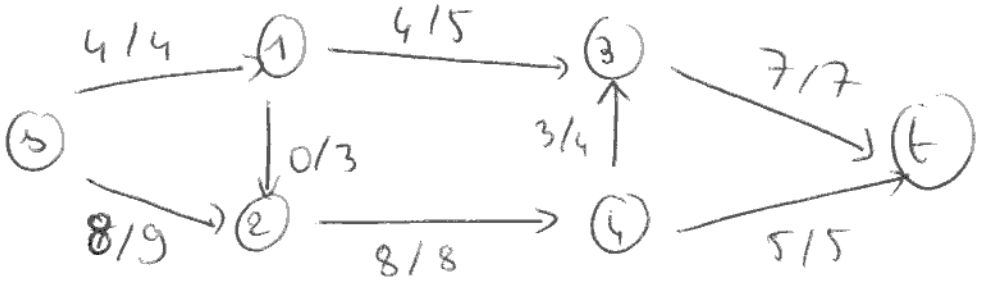
Saturons le 2e : on peut augmenter le flot de 1



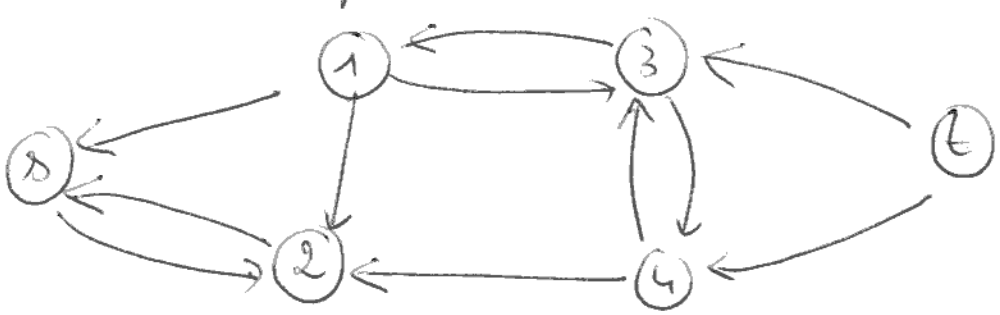
on obtient le graphe résiduel suivant :



(s, 2, 1, 3, t) est un chemin augmentant : on peut l'augmenter de 2 tout du long et on oblient



et le graphe résiduel serait



dans lequel il n'y a pas de chemin de s à t : l'algo est terminé

(10)

```
int** zeroes (int n) {
    int** res = malloc (n * sizeof(int*));
    for (int i = 0; i < n; i++) {
        res[i] = malloc (n * sizeof(int));
        for (int j=0; j<n; j++) {
            res[i][j] = 0;
        }
    }
    return res;
}
```

(11)

```
void free_matrix (int** mat, int n) {
    for (int i=0; i< n; i++) {
        free (mat[i]);
    }
    free (mat);
}
```

(12)

```
queue* queue_create (void) {
    queue* res = malloc (sizeof (queue));
    res -> len = 0;
    res -> left = NULL;
    res -> right = NULL;
    return res;
}
```

(13)
```
int queue_length (queue* q) {
    return (q->len);
}
```

(14)
```
void queue_push (queue* q, vertex v) {
    cell* new = malloc (size_of (cell));
    new->data = v;
    new->next = NULL;
    q->len ++;
    if (q->right == NULL) {
        q->right = new;
        q->left = new;
    }
    else { q->right->next = new;
        q->right = new;
    }
}
```

(15)
```
vertex queue_pop (queue* q) {
    cell* deb = q->left;
    assert (deb != NULL);
    vertex v = deb->data;
    if (q->len == 1) {
        q->left = NULL;
        q->right = NULL;
    }
    else { cell* effacer = q->left;
        q->left = q->left->next;
        free (effacer); }
    q->len--;
    return v; }
```

(16)

```
void cell_free (cell* c) {
    if (c != NULL) {
        cell_free (c -> next);
        free(c);
    }
}
```

(17)

```
void queue_free (queue* q) {
    cell_free (q -> left);
    free (q);
}
```

(18) insérer à gauche ne pose pas de problème pour supprimer à droite, il faudrait récupérer l'avant-dernière cellule ce que l'on ne peut pas faire efficacement sans parcourir la liste, il faudrait une liste doublement chainée.

(19)

```
vertex* bfs_residual (flow_graph* g, int** f) {
    queue* q = queue_create();
    queue_push (q, g -> s);
    vertex* parent = malloc (g -> n * sizeof (vertex));
    for (int i = 0; i < g -> n; i++) parent[i] = -1;
    parent [g -> s] = g -> s;
    while (queue_length (q) > 0) {
        vertex v = queue_pop (q);
        for (int i = 0; i < g -> degrees [v]; i++) {
            vertex w = g -> adj [v] [i];
            if (parent [w] == -1 &&
            g -> capacity [v] [w] > f [v] [w]) {
                parent [w] = v;
                queue_push (q, w);
            }
        }
    }
}
```

```
    queue_free q;
    return parent;
}
```

(20)
```
int path_capacity(flow_graph* g, int** f, vertex* parent){
    int freec = INT_MAX;
    int sommet = g->t;
    if (parent[sommet] == -1) return 0;
    while (v != g->s){
        vertex p = parent[v];
        freec = min(freec, g->capacity[p][v]
                           - f[p][v],
        v = p;
    }
    return freec;
}
```

(21)
```
bool saturate_path(flow_graph* g, int** f, vertex* parent){
    int c = path_capacity(g, f, parent);
    if (c == 0) return false;
    int v = g->t;
    while (v != g->s){
        vertex p = parent[v];
        f[p][v] += c;
        f[v][p] -= c;
        v = p;
    }
    return true;
}
```

(22)

```
bool step (flow-graph* g, int** f){
    int* parent = bfs_residual(g, f);
    bool result = saturate_path(g, f, parent);
    free(parent);
    return result;
}
```

(23) $O(|A|+|S|)$ pour le parcours du graphe puis deux parcours du chemin obtenu de s vers t.

(24)

```
int** ff (flow-graph* g) {
    int** f = zeroes(g->n);
    while (step(g,f)) {}
    return f;
}
```

(25) Saturer le chemin augmente la valeur du flot à chaque étape d'au moins un donc l'algo termine et la complexité est en $O(\underbrace{|S|^2}_{\text{créat° du tableau pour } f} + M(|S|+|A|))$

(26) elle vaut 15.

(27)

(27) Soit f un flot,

$$\forall (u,v) \in X \times \bar{X}, \quad f(u,v) \leq c(u,v)$$

$$\underline{\text{donc}} : \sum_{(u,v) \in X \times \bar{X}} f(u,v) \leq c(X)$$

on va montrer que $\sum_{(u,v) \in X \times \bar{X}} f(u,v) = |f|$ par rec sur la taille de X

* si $|X| = 1$ alors $X = \{s\}$ et $\bar{X} = S \setminus \{s\}$
  Ainsi $\sum_{(u,v) \in X \times \bar{X}} f(u,v) = \phi_+(s) = \phi(s) = |f|$

* Soit X de cardinal $n+1$ avec $n \geq 1$:
  soit $i \in X$, on pose $X' = X \setminus \{i\}$ et $\bar{X}' = \bar{X} \cup \{i\}$
  on a par HR $\sum_{(u,v) \in X' \times \bar{X}'} f(u,v) = |f|$

Or:

$$\sum_{(u,v) \in X \times \bar{X}} f(u,v) = \sum_{(u,v) \in X' \times \bar{X}'} f(u,v) + \sum_{v \in \bar{X}} f(i,v) - \sum_{v \in X'} f(v,i)$$

$$= \sum_{(u,v) \in X' \times \bar{X}'} f(u,v) + \sum_{v \in \bar{X} \cup X'} f(i,v) = \sum_{(u,v) \in X' \times \bar{X}'} f(u,v)$$

d'après la conservation au sommet i

et donc, on a bien

$$\sum_{(u,v)\in X\times\overline{X}} f(u,v) = |f|$$

---

(28) (a) $\Rightarrow$ (b) si le flot est maximal alors il ne peut pas y avoir de chemin améliorant car un tel chemin permettrait d'augmenter la valeur du flot.

(b) $\Rightarrow$ (c) On pose X l'ensemble des sommets accessibles depuis s alors $s\in X$ et $t\in\overline{X}$ car il n'y a pas de chemin améliorant.

De plus si $(u,v)\in X\times\overline{X}$ alors $(u,v)\notin G_f$ ce qui signifie que $f(u,v)=c(u,v)$, on a alors avec la question précédente que :

$$|f| = \sum_{u,v\in X\times\overline{X}} f(u,v) = \sum_{u,v\in X\times\overline{X}} c(u,v) = C(X)$$

(c) $\Rightarrow$ (a) s'il existe X t.q $|f| = C(X)$ alors f est maximal car pour tout autre flot f', on aura $|f'| \leqslant C(X) = |f|$ d'après la question précédente.

(29) On a déjà vu que l'algo termine et il va renvoyer un flot n'admettant pas de chemin améliorant ce qui garantit qu'il est maximal d'après la question précédente (le point b) $\Rightarrow$ c) plus précisément).