

# 1 Étude de réseaux sociaux (d'après X-ENS 2016 – info B)

## 1.1 Réseaux sociaux

1.

$$A = \begin{pmatrix} 3 & 1 & 2 & 3 & \cdot \\ 2 & 0 & 2 & \cdot & \cdot \\ 3 & 0 & 1 & 3 & \cdot \\ 2 & 0 & 2 & \cdot & \cdot \\ 0 & \cdot & \cdot & \cdot & \cdot \end{pmatrix} \quad B = \begin{pmatrix} 1 & 1 & \cdot & \cdot & \cdot \\ 3 & 0 & 2 & 3 & \cdot \\ 3 & 1 & 3 & 4 & \cdot \\ 3 & 1 & 2 & 4 & \cdot \\ 2 & 2 & 3 & \cdot & \cdot \end{pmatrix}$$

```
2. reseau creer_reseau_vide(int n) {
    reseau R;
    R.nb_users = n;
    R.liens = (int**)malloc(n * sizeof(int*));
    for (int i = 0; i < n; i++) {
        R.liens[i] = (int*)malloc(n * sizeof(int));
        R.liens[i][0] = 0; // i possède 0 amis
    }
    return R;
}
```

```
3. bool sont_amis(reseau R, int i, int j) {
    int nb_amis = R.liens[i][0];
    for (int k = 1; k <= nb_amis; k++) {
        if (R.liens[i][k] == j) {
            return true;
        }
    }
    return false;
}
```

```
4. void ajouter_amis(reseau R, int i, int j) {
    if (sont_amis(R,i,j)) { return; }
    /* on ajoute j parmi les amis de i */
    R.liens[i][0]++;
    R.liens[i][R.liens[i][0]] = j;
    /* on ajoute j parmi les amis de j */
    R.liens[j][0]++;
    R.liens[j][R.liens[j][0]] = i;
}
```

```
5. void free_reseau(reseau R) {
    for (int i = 0; i < R.nb_users; i++) {
        free(R.liens[i]);
    }
    free(R.liens);
}
```

```
6. int nb_liens(reseau R) {
    int s = 0;
    for (int i = 0; i < R.nb_users; i++) {
        s += R.liens[i][0];
    }
    return s/2;
}
```

```
7. couple* liste_des_liens(reseau R) {
    int taille = nb_liens(R);
    couple* l_liens = (couple*)malloc(taille * sizeof(couple));
    int indice = 0; // indice de la prochaine case de la liste à remplir
    for (int i = 0; i < R.nb_users; i++) {
        int nb_amis = R.liens[i][0];
        for (int k = 1; k <= nb_amis; k++) {
            int j = R.liens[i][k];
            if (i < j) {
                couple c = {.fst = i, .snd = j};
                l_liens[indice] = c;
                indice++;
            }
        }
    }
    return l_liens;
}
```

## 1.2 Partitions

8. • Représentation filiale A  
Représentants : 5, 4, 1 et 3.

i :	0	1	2	3	4	5	6	7	8	9
parent[i] :	5	1	1	3	4	5	1	5	5	7

- Représentation filiale B  
Représentants : 9, 7 et 3.

i :	0	1	2	3	4	5	6	7	8	9
parent[i] :	3	9	0	3	9	4	4	7	1	9

```
9. int* creer_partition_en_singletons(int n) {
    int* parent = (int*)malloc(n * sizeof(int));
    for (int i = 0; i < n; i++) {
        parent[i] = i;
    }
    return parent;
}
```

```
10. (a) int representant(int* parent, int i) {
    if (parent[i] == i) { return i; }
    return representant(parent, parent[i]);
}
```

- (b) Dans le pire des cas, on a un seul groupe,  $i$  est une feuille, et l'arbre est filiforme. La complexité est donc en  $\mathcal{O}(n)$ .

Exemple :  $i = 0$  et :

$i$ :	0	1	2	3	4	5	6	...	$n-2$	$n-1$
parent[ $i$ ] :	1	2	3	4	5	6	7	...	$n-1$	$n-1$

```
11. void fusion(int* parent, int i, int j) {
    int r_i = representant(parent, i);
    int r_j = representant(parent, j);
    parent[r_i] = r_j;
}
```

12. On exécute dans l'ordre :

```
1. fusion(parent, 0, 1)
2. fusion(parent, 0, 2)
⋮
⋮
(n-1). fusion(parent, 0, n-1)
```

Le groupe de 0 a comme tailles successives  $1, 2, \dots, n-1$  et 0 s'y trouve toujours en bas (feuille de l'arbre) donc les calculs de son représentant vont avoir un coût de l'ordre de  $1 + 2 + \dots + (n-1) = \frac{n(n-1)}{2}$ . Le reste est négligeable. On aura bien une fusion totale en  $\Theta(n^2)$  opérations élémentaires.

```
13. (a) int representant(int* parent, int i) {
    if (parent[i] == i) { return i; }
    int rep = representant(parent, parent[i]);
    parent[i] = rep;
    return rep;
}
```

- (b) On effectue uniquement une opération élémentaire supplémentaire à chaque fois qu'on dépile un appel récursif : la complexité est toujours en  $\mathcal{O}(n)$  dans le pire des cas.

### 1.3 Algorithme randomisé pour la coupe minimum

Fonctions préliminaires

```
14. (a) void echange(couple* l_liens, int i, int j) {
    couple tmp = l_liens[i];
    l_liens[i] = l_liens[j];
    l_liens[j] = tmp;
}

bool pas_meme_groupe(int* parent, int i, int j) {
    return representant(parent, i) != representant(parent, j);
}
```

```

int* coupe_minimum_randomisee(reseau R) {
    int n = R.nb_users;
    int m = nb_liens(R);
    couple* l_liens = liste_des_liens(R);
    int nb_groupes = n;
    // Étape 1
    int* parent = creer_partition_en_singletons(n);
    // Étape 2
    int indMaxNonMarques = m-1; // les liens non marqués sont ceux situés avant
    // Étape 3
    while (nb_groupes > 2 && indMaxNonMarques >= 0) {
        // (a)
        int k = randint(0, indMaxNonMarques);
        int i = l_liens[k].fst;
        int j = l_liens[k].snd;
        // (b)
        if (pas_meme_groupe(parent,i,j)) {
            fusion(parent,i,j);
            nb_groupes--;
        }
        // (c)
        echange(l_liens,k,indMaxNonMarques);
        indMaxNonMarques--;
    }
    // Étape 4 : on choisit de fusionner le groupe de 0 avec les k-2 autres
    int i = 1;
    while (nb_groupes > 2) {
        if (pas_meme_groupe(parent,i,0)) {
            fusion(parent,i,0);
            nb_groupes--;
        }
        i++;
    }
    // Étape 5
    free(l_liens); // Attention aux fuites mémoires !
    return parent;
}

```

- (b)
- Pour passer de  $n$  à 2 groupes, il faut exactement  $n - 2$  fusions, chacune en  $2\alpha(n)$ , soit  $\mathcal{O}(n\alpha(n))$ .
  - Il y a  $k + \ell$  tests de `pas_meme_groupe` où  $k$  le le nombre de tours dans la première boucle et  $\ell$  le nombre de tours dans le deuxième, avec  $k \leq m$  (et même  $\min(n, m)$ ) et  $\ell \leq n$  soit  $\mathcal{O}((n + m)\alpha(n))$ .
  - Le reste est négligeable.
  - On obtient au total une complexité en  $\mathcal{O}((n + m)\alpha(n))$ .

```

15. int taille_coupe(reseau R, int* parent) {
    int m = nb_liens(R);
    couple* l_liens = liste_des_liens(R);
    int taille = 0;
    for (int k = 0; k < m; k++) {
        int i = l_liens[k].fst;
        int j = l_liens[k].snd;
        if (pas_meme_groupe(parent,i,j)) {
            taille++;
        }
    }
    free(l_liens); // Attention aux fuites mémoires !
    return taille;
}

```

## 2 Étude de relations d'ordre (d'après Mines-Ponts 2024 – option info)

### 2.1 Relation d'ordre et tri topologique

```

16. (a) let check_reflexivity r =
    let n = Array.length r in
    let res = ref true in
    for i = 0 to n-1 do
        if not r.(i).(i) then res := false
    done ;
    !res

```

```

(b) let check_reflexivity_bis r =
    let n = Array.length r in
    let rec aux i =
        if i = n then true
        else r.(i).(i) && aux (i+1)
    in aux 0

```

```

17. exception Break

let check_antisymetry r =
    let n = Array.length r in
    try
        for i = 0 to n-1 do
            for j = i+1 to n-1 do (* i < j *)
                if r.(i).(j) && r.(j).(i) then raise Break
            done
        done ;
        true
    with Break -> false

```

```

18. let check_transitivity r =
    let n = Array.length r in
    try
      for i = 0 to n-1 do
        for j = 0 to n-1 do
          for k = 0 to n-1 do
            if r.(i).(j) && r.(j).(k) && not r.(i).(k) then raise Break
          done
        done
      done ;
    true
  with Break -> false

```

```

19. let is_order r =
    check_reflexivity r && check_antisymetry r && check_transitivity r

```

```

20. let nb_sup r =
    let n = Array.length r in
    let d = Array.make n 0 in
    for i = 0 to n-1 do
      for j = 0 to n-1 do
        if r.(i).(j) then d.(i) <- d.(i) + 1
      done
    done ;
    d

```

```

21. let argmin d =
    let n = Array.length d in
    let i_min = ref 0 in
    for i = 0 to n-1 do
      if d.(i) < d.(!i_min) then i_min := i
    done ;
    !i_min

```

22. Dans cette question, on prouve que tout ensemble ordonné fini  $(V, \leq)$  possède un tri topologique.

(a) Comme  $V \neq \emptyset$ , on peut choisir un  $v_{i_0} \in V$ , puis :

- si  $\forall v_j \neq v_{i_0}$ , on n'a pas  $v_{i_0} \leq v_j$ , alors ce  $v_{i_0}$  convient ;
- sinon,  $\exists v_{i_1} \neq v_{i_0}$  tel que  $v_{i_0} \leq v_{i_1}$ , et on recommence le même raisonnement avec  $v_{i_1}$ .

On crée ainsi une suite strictement croissante d'éléments distincts de  $V$ , et comme  $V$  est fini ce processus s'arrête forcément (la suite est au plus de longueur  $n$ ). De plus, en notant  $v_{i_k}$  l'élément sur lequel le processus s'arrête, ce  $v_{i_k}$  convient. En effet,  $\forall v_j \neq v_{i_k}$  :

- si  $v_j$  fait partie de la suite, alors par transitivité de  $\leq$  on a  $v_j \leq v_{i_k}$ , et par antisymétrie de  $\leq$  on n'a pas  $v_{i_k} \leq v_j$  (car sinon, on aurait  $v_j = v_{i_k}$  : absurde) ;
- si  $v_j$  ne fait pas partie de la suite, alors on n'a pas  $v_{i_k} \leq v_j$  car sinon on aurait pu continuer la suite avec ce  $v_j$ .

(b) Montrons par récurrence sur  $n = |V|$  que tout ensemble ordonné fini  $(V, \leq)$  possède un tri topologique :

- $n = 0 : V = \emptyset$  : OK.
- Soit  $n > 0$ , supposons la propriété vraie pour  $n - 1$ . On cherche via le processus de la question 22a un élément maximal de  $V$ , c'est-à-dire un élément  $v_i \in V$  tel que  $\forall v_j \neq v_i$ , on n'a pas  $v_i \leq v_j$ . Considérons alors  $V \setminus \{v_i\}$ . Par (HR), on arrive à créer un tri topologique de  $V \setminus \{v_i\}$ , et en rajoutant  $\tau(i) = n - 1$  on obtient alors bien un tri topologique de  $V$ .

23. Un tel  $v_i$  vérifiant la propriété de la question 22a peut être calculé via l'appel `argmin d` où `d` a été calculé par la fonction `nb_sup r`.

```
let topological_sort r =
  let n = Array.length r in
  let d = nb_sup r in
  let tau = ref [] in
  let deja_ajoute = Array.make n false in
  let rec recherche_maximal () =
    let prochain = argmin d in (* trouve un élément maximal *)
    tau := prochain :: !tau ;
    deja_ajoute.(prochain) <- true ;
    (* on met à jour d pour ignorer cet élément les prochaines fois *)
    d.(prochain) <- n+1 ; (* il suffit de mettre une valeur > n *)
    for i = 0 to n-1 do
      if not deja_ajoute.(i) && r.(i).(prochain) then d.(i) <- d.(i) - 1
    done
  in
  (* on le fait n fois *)
  for _ = 0 to n-1 do
    recherche_maximal ()
  done ;
  !tau
```

24. À chaque passage dans la boucle principale, la fonction `recherche_maximal` trouve un élément maximal et le retire virtuellement de  $V$  en mettant à jour les valeurs de `d` (on met en particulier une valeur  $> n$  dans `d.(prochain)` pour qu'il ne soit jamais à nouveau renvoyé par `argmin`). Ainsi, à chaque passage dans la boucle principale, `tau` contient un tri topologique des éléments marqués comme `deja_ajoute`.

25. Initialement, `d = nb_sup r` se calcule en  $\mathcal{O}(n^2)$  et `deja_ajoute` se crée en  $\mathcal{O}(n)$ . Chaque appel à `argmin` se fait en  $\mathcal{O}(n)$ , puis la mise à jour des valeurs de `d` se font aussi en  $\mathcal{O}(n)$ . Ainsi, chaque appel à `recherche_maximal` se fait en  $\mathcal{O}(n)$ , d'où une complexité totale en  $\mathcal{O}(n^2)$ .

## 2.2 Chaines et antichaines

26. 

```
let rec is_chain r c = match c with
| [] -> true
| v::cc -> List.for_all (fun x -> r.(x).(v) || r.(v).(x)) cc
&& is_chain r cc
```

27. Chaque appel à `List.for_all` se fait en  $\mathcal{O}(|cc|)$ , d'où une complexité totale en  $\sum_{\gamma'=1}^{\gamma} \mathcal{O}(\gamma') = \mathcal{O}(\gamma^2)$ .

28. 

```
let rec is_antichain r c = match c with
| [] -> true
| v::cc -> List.for_all (fun x -> not r.(x).(v) && not r.(v).(x)) cc
&& is_antichain r cc
```

29. De même que pour `is_chain`, on obtient une complexité de `is_antichain` en  $\mathcal{O}(\gamma^2)$ .

### 2.3 Couverture par des chaines

30. Dans une chaine, tous les éléments sont comparables, donc une antichaine ne possède qu'au plus un élément dans chaque chaine  $C_\ell$ . De plus, comme les chaines  $(C_\ell)_{\ell \in \llbracket 1, \lambda \rrbracket}$  couvrent  $V$ , une antichaine ne peut pas posséder un élément qui ne soit dans aucune chaine. D'où  $\alpha \leq \lambda$ .

31. 

```
let pursuit_graph_of_order r =
  let n = Array.length r in
  let gp = Array.make (2*n) [] in
  for i = 0 to n-1 do
    for j = 0 to n-1 do
      if i <> j && r.(i).(j) then
        begin
          gp.(i) <- (n+j) :: gp.(i) ;
          gp.(n+j) <- i :: gp.(n+j)
        end
      end
    done
  done ;
  gp
```

32. Comme on souhaite que  $\lambda$  soit le plus grand possible, cela implique qu'une fois qu'un premier élément  $v_i$  est choisi pour construire un nouveau  $C_\ell$ , il faut rajouter dans  $C_\ell$  uniquement les éléments nécessaires pour avoir la première propriété :

- on commence par construire la chaine strictement croissante partant de  $v_i$  : s'il existe un couple de la forme  $(v_i, v_{n+j}) \in M$ , alors comme  $M$  est un couplage, ce  $j$  est unique, et on rajoute  $v_j$  dans  $C_\ell$  puis on recommence depuis  $v_j$ .  
 $\hookrightarrow$  ce processus s'arrête pour les mêmes raisons discutées à la question 22a. Notons  $v_{j_0}$  l'élément sur lequel ce processus s'arrête.
- on construit ensuite la chaine strictement décroissante partant de  $v_i$  : s'il existe un couple de la forme  $(v_k, v_{n+i}) \in M$ , alors comme  $M$  est un couplage, ce  $k$  est unique, et on rajoute  $v_k$  dans  $C_\ell$  puis on recommence depuis  $v_k$ .  
 $\hookrightarrow$  ce processus s'arrête également pour les mêmes raisons discutées à la question 22a. Notons  $v_{k_0}$  l'élément sur lequel ce processus s'arrête.

Il est ensuite inutile de réitérer ce processus depuis  $v_{k_0}$  ou  $v_{j_0}$  (ou n'importe quel autre élément rencontré) car, comme  $M$  est un couplage, chaque élément rencontré n'apparaît au plus qu'une fois dans la partie gauche d'un couple et au plus qu'une fois dans la partie droite d'un autre couple, et les éléments associés ont déjà été rencontrés par le processus précédent.

Finalement, par transitivité, tous les éléments rajoutés dans un tel  $C_\ell$  sont comparables :  $C_\ell$  est donc une chaine.

33. Faisons une remarque supplémentaire sur le processus décrit dans la preuve précédente : il est facile de détecter les  $v_{j_0}$  en question : ce sont les  $v_{j_0}$  n'apparaissant jamais dans la partie gauche d'un couple du couplage, autrement dit, ce sont les  $j_0$  n'apparaissant pas dans le tableau `m`. Il suffit alors de partir de ces  $j_0$  et d'effectuer uniquement la deuxième étape du processus qui crée une chaine strictement décroissante depuis l'élément  $v_{j_0}$ .

On obtient alors le code suivant :



```

let chains_of_matching m =
  let n = Array.length m in
  let rec creer_chaine elem =
    if m.(elem) = -1 then [elem]
    else elem :: creer_chaine m.(elem)
  in
  let l_chaines = ref [] in
  for j0 = 0 to n-1 do
    if not (Array.mem j0 m) (* j0 est l'élément maximal d'une chaîne *)
    then l_chaines := (creer_chaine j0) :: !l_chaines
  done ;
  !l_chaines

```

34. Pour chaque  $C_\ell$ , on calcule un tri topologique  $\tau_\ell$ . Comme  $C_\ell$  est une chaîne, on obtient alors une suite strictement croissante :

$$v_{\tau_\ell(0)} \trianglelefteq v_{\tau_\ell(1)} \trianglelefteq \cdots \trianglelefteq v_{\tau_\ell(|C_\ell|-1)}$$

Il suffit alors de rajouter dans le couplage  $M$  tous les couples de la forme  $(v_{\tau_\ell(i)}, v_{n+\tau_\ell(i+1)})$ .

Finalement, on obtient le couplage suivant, qui permet bien d'obtenir  $(C_\ell)_{\ell \in \llbracket 1, \lambda \rrbracket}$  :

$$M = \bigcup_{\ell=1}^{\lambda} \left\{ (v_{\tau_\ell(i)}, v_{n+\tau_\ell(i+1)}) \mid i \in \llbracket 0, |C_\ell| - 2 \rrbracket \right\}$$

35. On commence par montrer par récurrence sur  $|M|$  que la couverture obtenue à partir de  $M$  vérifie  $\lambda = n - |M|$  :

- si  $|M| = 0$ , tous les  $C_\ell$  sont des singletons, et  $\lambda = n$  ;
- soit  $|M| > 0$  et supposons le résultat vrai pour les couplages de cardinal  $|M| - 1$ . Comme  $M \neq \emptyset$ , on choisit  $(v_i, v_{n+j}) \in M$ , et on considère  $M' = M \setminus \{(v_i, v_{n+j})\}$ .  $M'$  est encore un couplage, donc par (HR) on obtient à partir de  $M'$  une couverture de cardinal  $\lambda' = n - |M'|$ . Notons  $C$  et  $C'$  les chaînes contenant respectivement  $v_i$  et  $v_j$  dans la couverture obtenue depuis  $M'$ . De plus, dans  $M'$ ,  $v_i$  (resp.  $v_j$ ) n'apparaît pas dans la partie gauche (resp. droite) d'un couple donc  $v_i$  (resp.  $v_j$ ) est le  $v_{j_0}$  de  $C$  (resp. le  $v_{k_0}$  de  $C'$ ). Ainsi, si on avait  $C = C'$ , on aurait  $v_j = v_{k_0} \trianglelefteq v_{j_0} = v_i$  : impossible car  $v_i \trianglelefteq v_j$  avec  $i \neq j$  et  $\trianglelefteq$  est antisymétrique. Donc  $C \neq C'$ , et à partir de  $M$  on construit la même couverture à la différence que  $C$  et  $C'$  sont fusionnées. D'où  $\lambda = \lambda' - 1 = n - |M'| - 1 = n - |M|$ .

Ainsi, considérons une couverture de  $V$  avec  $\lambda$  minimal. D'après la question précédente, il existe un couplage  $M$  associé. Et s'il existait un couplage  $M'$  avec  $|M| < |M'|$ , on obtiendrait une couverture avec  $\lambda' < \lambda$  : absurde. Donc le couplage  $M$  associé à cette couverture est maximal.

Réciproquement, considérons un couplage  $M$  de cardinal maximal, et considérons la couverture obtenue à partir de ce  $M$ , et notons  $\lambda$  sa taille. S'il existait une autre couverture de taille  $\lambda' < \lambda$ , alors d'après la question précédente, on pourrait construire un couplage  $M'$  avec (d'après le lemme établi au début de cette question)  $|M'| > |M|$  : absurde. Donc  $\lambda$  est minimal.

36. • Chaque appel de **visit**  $j$  visite un élément  $v_{j'} = v_{n+j} \in V'$ , et dans ce cas :
- soit on s'arrête si on a déjà visité ce  $v_{n+j}$  (c'est à ça que sert le tableau **visited**) ;
  - soit on s'arrête si ce  $v_{n+j}$  n'est couplé à personne ;
  - soit on appelle **augment** sur l'élément  $v_i$  tel que  $(v_i, v_{n+j}) \in M$ .
- Chaque appel de **augment**  $i$  **visited** appelle **visit**  $j$  sur tous les éléments  $v_{j'} = v_{n+j} \in V'$  tels que  $v_i \trianglelefteq v_j$  et  $j \neq i$  (il y en a donc au plus  $n - 1$ ).

Ainsi, chaque appel initial à **augment** à la ligne 16 se fait en  $\mathcal{O}(n^2)$  (on est borné par le nombre de couples  $(i, j)$  tels que  $v_i \trianglelefteq v_j$ ).

Et on passe  $n$  fois dans la boucle principale, d'où une complexité en  $\mathcal{O}(n^3)$ .

```
37. let minimum_cover r =  
    chains_of_matching (maximum_matching (pursuit_graph_of_order r))
```

- 38.
- `pursuit_graph_of_order` :  $\mathcal{O}(n^2)$ ;
  - `maximum_matching` :  $\mathcal{O}(n^3)$ ;
  - `chains_of_matching` : chaque appel à `Array.mem` et à `creer_chaine` se fait en  $\mathcal{O}(n)$ , d'où une complexité en  $\mathcal{O}(n^2)$ .

Finalement, `minimum_cover` se fait en  $\mathcal{O}(n^3)$ .