

On s'intéresse ici à un problème d'appariement entre deux groupes d'individus U et V , que l'on suppose de même cardinalité n .

On cherche à appairer les éléments de V aux éléments de U , chaque élément devant être apparié à exactement un élément de l'autre ensemble. Chaque élément de V (respectivement de U) a, de plus, un ordre de préférence total entre tous les éléments de U (respectivement de V).

Définition 1 (Couplage, couplage parfait)

Un ensemble de paires $A \subseteq V \times U$ est un couplage si tout $x \in V$ et tout $y \in U$ apparaissent dans au plus un élément de A . Le couplage est dit parfait si tout $x \in V$ et tout $y \in U$ apparaissent dans un et un seul élément de A .

Notations

Dans toute la suite on notera :

- $|E|$ le cardinal de l'ensemble E ,
- $>^x$ l'ordre associé à l'élément $x \in V \cup U$: si $u_1 \in U$ préfère strictement $v_1 \in V$ à $v_2 \in V$ alors $v_1 >^{u_1} v_2$.
Par extension, on définit de même la notion de préférence $\geq^x$.
- $m_A(x)$ le partenaire de $x \in V \cup U$ dans le couplage parfait A .

Définition 2 (Couplage parfait stable)

Un couplage parfait A est dit stable si, pour tous couples (v_1, u_1) et (v_2, u_2) de A , il est impossible que l'on ait $u_2 >^{v_1} u_1$ et $v_1 >^{u_2} v_2$.

Définition 3 (Pareto-optimalité)

Soient A et A' deux couplages parfaits stables et $E \subseteq V \cup U$. On dit que A Pareto-domine A' sur E si et seulement si pour tout $x \in E$, $m_A(x) \geq^x m_{A'}(x)$ et il existe $x \in E$ tel que $m_A(x) >^x m_{A'}(x)$. Un couplage est Pareto-optimal s'il est stable et n'est pas Pareto dominé sur $V \cup U$.

En économie, la Pareto-optimalité d'un système exprime le fait que l'on ne peut améliorer la satisfaction d'un individu du système sans diminuer la satisfaction d'un autre.

III.1 - Recherche de couplages stables

L'algorithme de Gale-Shapley (**algorithme 3**) permet de montrer qu'il existe toujours au moins un couplage parfait stable et d'en trouver un de façon efficace. L'idée de l'algorithme est tout d'abord de choisir un des deux ensembles (ici pour illustration V). Les éléments de V font des propositions aux éléments de U . Quand un élément de V fait une proposition à un élément de U , celui-ci peut soit la refuser définitivement, soit l'accepter provisoirement en attendant une éventuelle meilleure offre. Toute nouvelle acceptation vaut rejet définitif des propositions précédemment acceptées.

Algorithme 3 - Algorithme de Gale-Shapley

Entrées : $n = |V| = |U|$, les listes de préférences des éléments de V et U

Sorties : Un couplage parfait stable A

début

$A := \emptyset$

tant que il existe un élément de V non apparié faire

Choisir un tel v

$u :=$ élément de U que v préfère parmi ceux à qui il n'a pas encore $\leftarrow$ proposé

si u n'est pas apparié dans A alors

$A := A \cup \{(v, u)\}$

sinon

si u est apparié dans A à v' ET $v >^u v'$ alors

$A := A \setminus \{(v', u)\}$

$A := A \cup \{(v, u)\}$

retourner A

Soit le jeu de données $\mathcal{D}$ suivant : $V = \{v_1, v_2, v_3\}, U = \{u_1, u_2, u_3\}$ et :

V	U
$u_2 >^{v_1} u_1 >^{v_1} u_3$	$v_1 >^{u_1} v_3 >^{u_1} v_2$
$u_1 >^{v_2} u_2 >^{v_2} u_3$	$v_2 >^{u_2} v_1 >^{u_2} v_3$
$u_1 >^{v_3} u_2 >^{v_3} u_3$	$v_2 >^{u_3} v_1 >^{u_3} v_3$

Q17. Donner la trace de l'**algorithme 3** sur ces données. Le choix de v dans l'algorithme se fera par ordre croissant des indices.

correction

On choisit v_1 .

$u := u_2$

$A := \{(v_1, u_2)\}$

On choisit v_2 .

$u := u_1$

$A := \{(v_1, u_2), (v_2, u_1)\}$

On choisit v_3 .

$u := u_1$, déjà apparié à v_2 , mais qui préfère v_3 .

$A := \{(v_1, u_2), (v_3, u_1)\}$

On choisit v_2 .

$u := u_2$, déjà apparié à v_1 , mais qui préfère v_2 .

$A := \{(v_2, u_2), (v_3, u_1)\}$

On choisit v_1 .

$u := u_1$, déjà apparié à v_3 , mais qui préfère v_1

$A := \{(v_2, u_2), (v_1, u_1)\}$

On choisit v_3 .

$u := u_2$, déjà apparié à v_2 , et qui préfère v_2 à v_3 .

On choisit v_3 .

$u := u_3$

$A := \{(v_2, u_2), (v_1, u_1), (v_3, u_3)\}$

Q18. Montrer que l'**algorithme 3** termine.

correction

On considère comme variant le nombre de couples (v, u) , où v a fait une proposition à u .

Ce nombre augmente strictement à chaque passage dans la boucle **tant que**, et est majoré par le nombre de couples (n^2) , donc l'algorithme termine en au plus n^2 itérations.

Q19. Donner, sans le prouver, un invariant de boucle permettant de prouver que l'**algorithme 3** retourne un couplage parfait.

correction

On considère l'invariant «à la fin de chaque itération, A est un couplage».

Q20. Montrer que le couplage parfait calculé est stable.

correction

Raisonnons par l'absurde. Supposons qu'il existe deux couples (v_1, u_1) et (v_2, u_2) renvoyés par l'algorithme tels que $u_2 >^{v_1} u_1$ et $v_1 >^{u_2} v_2$.

- Si v_1 choisit u_2 avec que v_2 ne le fasse. Lorsque u_2 quittera v_1 , se sera pour un v_i tel que $v_i >^{u_2} v_1 >^{u_2} v_2$, et donc u_2 ne pourra jamais être apparié avec v_2 : absurde.
- Si v_2 choisit u_2 avant que v_1 ne le fasse. Mais alors, v_1 (ou un v encore plus intéressant pour u_2) choisira u_2 après, et v_2 ne sera pas apparié avec u_2 : absurde.

Q21. Montrer que cet algorithme effectue moins de n^2 itérations de boucles.

correction

Déjà prouvé ci-dessus lors de la réponse sur la terminaison.

Pour des préférences données, il peut exister plus d'un couplage parfait stable. Soient A et A' deux tels couplages pour un problème donné. On dira qu'un élément de $V \cup U$ préfère A à A' s'il n'a pas le même partenaire dans les deux couplages et s'il préfère celui de A à celui de A' .

Q22. Montrer que si v préfère le couplage A , alors son partenaire u dans A préfère A' .

correction

Supposons que $(v, u) \in A$, $(v, u') \in A'$, $(v', u) \in A'$, $u \neq u'$, $v \neq v'$ et $u >^v u'$ (i.e. v préfère A à A'). On souhaite montrer que u préfère A' , i.e. $v' >^u v$.

Si ça n'était pas le cas, on aurait $v >^u v'$, et les 4 propriétés suivantes forment une contradiction avec la définition de la stabilité de A' :

- $(v, u') \in A'$
- $(v', u) \in A'$
- $v >^u v'$
- $u >^v u'$

Donc u préfère bien A' à A .

On peut montrer qu'en même temps le partenaire u' de v dans A' préfère A .

La correction de l'algorithme de Gale-Shapley ne dépend pas de l'élément $v \in V$ qui fait sa proposition à chaque tour. En fait, on montre dans la suite que, quel que soit l'élément $v \in V$ non encore apparié qui fait une proposition à chaque tour, l'algorithme calcule toujours le même couplage parfait stable. Dans la suite, on dira qu'un élément $u \in U$ est *réalisable* pour un élément $v \in V$ s'il existe un couplage parfait stable qui contient (v, u) .

Q23. Montrer qu'au cours de l'algorithme 3, chaque $v \in V$ n'est rejeté que par les éléments $u \in U$ qui ne sont pas réalisables pour v .

correction

TODO

Soit $M(v) \in U$ l'élément réalisable le mieux classé sur la liste des préférences de v et $P(u) \in V$ l'élément réalisable le moins bien classé sur la liste des préférences de u . On note enfin $A^* = \{(v, M(v)), v \in V\}$.

Q24. Montrer que lors de l'exécution de l'algorithme 3 :

1. Tout $v \in V$, s'il est apparié, l'est avec $u \in U$ supérieur ou égal à $M(v)$ dans sa liste de préférences.
2. Tout $v \in V$, s'il n'est pas apparié, n'a fait des propositions qu'à des $u \in U$ strictement supérieurs à $M(v)$ dans sa liste de préférences.

correction

On montre que les 2 propriétés forment un invariant.

Elles sont trivialement vraies avant le premier passage dans le **tant que** – aucun v n'est apparié et il n'y a eu aucune proposition.

Montrer que les propriétés sont préservées par un passage dans la boucle. Distinguons des cas :

- Si v fait une proposition à u qui n'est pas apparié, l'appariement a lieu. En fin d'exécution, si v sera apparié avec u' (éventuellement égal à u) – à qui il aura fait une proposition après (au sens large) u . u' sera donc réalisable, et u est après u' donc après $M(v)$ dans sa liste de préférence.
- Si v fait une proposition à u mais que u refuse : en fin d'exécution, si v sera apparié avec $u' \neq u$ – à qui il aura fait une proposition après u . u' sera donc réalisable, et u est strictement après u' donc strictement après $M(v)$ dans sa liste de préférence.

La propriété reste vraie trivialement pour les précédents u_i auquel v a fait une proposition précédemment, qui sont encore après dans la liste.

- Si v fait une proposition à u , déjà apparié à v' , et que u accepte. Alors
 - Par l'argument du premier point, u est supérieur ou égal à $M(v)$ dans la liste des préférences de u .
 - v' n'est plus apparié. Par hypothèse, il n'a fait que des proposition à des éléments supérieures ou égaux à $M(v')$. Reste à montrer que u n'est pas égal à $M(v')$ – i.e. que (v', u) n'est pas réalisable. C'est le cas d'après la question précédente, car en considérant une exécution de l'algorithme qui aurait apparié v à u avant que v' ne fasse une proposition à u , u aurait rejeté v' .

NDMP. Argument qui mériterait d'être précisé, je suis preneur de preuves alternatives.

Q25. En déduire que quel que soit l'élément $v \in V$ choisi dans la boucle Tant que de l'**algorithme 3**, l'algorithme termine en produisant l'ensemble A^* .

correction

À la fin de l'algorithme, on obtient un couplage stable – donc par définition, v est apparié avec un u réalisable.

Or d'après la question précédente, il est avec un u supérieur ou égal à $M(v)$ – qui est l'élément réalisable le mieux classé.

Il est donc nécessairement apparié avec $M(v)$.

Ceci est vrai pour tout v , donc l'algorithme termine en produisant A^* .

Ainsi, l'algorithme calcule le meilleur couplage parfait stable possible du point de vue des éléments de V . Il s'avère également que l'algorithme de Gale-Shapley fait correspondre u avec $P(u)$, pour tout $u \in U$. L'algorithme avantage donc de manière claire le groupe ayant l'initiative (ici V) : le couplage stable produit est dit *Pareto-V-optimal* en ce sens qu'il n'est pas Pareto-dominé sur V .

III.2 - Couplages Pareto-optimaux

Dans cette sous-partie, on utilise également le jeu de données $\mathcal{D}$.

L'algorithme de Gale-Shapley ne retourne pas nécessairement un couplage Pareto-optimal. On s'intéresse ici à un algorithme susceptible de produire cette propriété.

Pour construire un couplage Pareto-optimal, on représente les données à l'aide d'un graphe orienté $\mathcal{G} = (S, E)$ où :

- les sommets S sont les éléments de $V \cup U$,
- les arcs E représentent les premiers choix : $(v, u) \in E$ si et seulement si u est le premier choix de v dans sa liste de préférences et $(u, v) \in E$ si et seulement si v est le premier choix de u .

Q26. Montrer que $\mathcal{G}$ contient au moins un circuit, c'est-à-dire un chemin dont les deux sommets extrémités sont identiques.

correction

On part d'un sommet v arbitraire, on suit l'unique arc (v, u) partant de v , puis on recommence à partir de u .

On crée ainsi un chemin infini (car il y a un arc sortant de tout sommet), qui passe donc deux fois par un même sommet – ce qui prouve l'existence d'un circuit dans le graphe.

Q27. Justifier par l'absurde que tout $x \in V \cup U$ est dans au plus un circuit.

correction

Il ne paraît pas utile de raisonner par l'absurde. Pour tout $n \in \mathbb{N}$, une récurrence immédiate montre qu'il existe une unique chemin de longueur n commençant par u (car il y a exactement un arc sortant de chaque sommet).

Quitte à changer son point de départ, on peut toujours considérer qu'un circuit contenant u part de u .

Il existe donc un unique circuit contenant u .

On propose alors l'**algorithme 4** pour calculer un couplage, dont on admettra qu'il est Pareto-optimal pour V .

Algorithme 4 - **Algorithme** de couplage par graphe orienté

Entrées : $n = |V| = |U|$, les listes de préférences des éléments de V et U

Sorties : Un couplage A

début

$A := \emptyset$

tant que il existe un élément de V **non** apparié **faire**

 Construire le graphe $\mathcal{G}$ sur les données courantes

pour chaque arc (v, u) présent dans un circuit de $\mathcal{G}$ **faire**

$A = A \cup \{(v, u)\}$

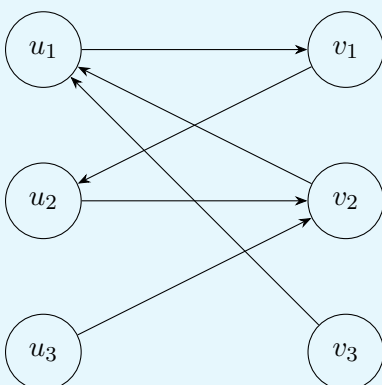
 Supprimer du problème tous les $v \in V$ et les $u \in U$ appariés

 Mettre à jour les listes de préférences des individus restants

retourner A

Q28. Donner les graphes et la construction itérative de A produits par l'**algorithme 4** sur $\mathcal{D}$.

correction



On peut considérer le circuit $u_2 \rightarrow v_2 \rightarrow u_1 \rightarrow v_1 \rightarrow u_2$, d'où les appariements (v_2, u_1) et (v_1, u_2) .

À l'étape suivante, le graphe est réduit aux sommets u_3 et v_3 , et a deux arcs les reliant. On aura donc l'appariement (v_3, u_3) .

Q29. En utilisant les données $\mathcal{D}$, montrer que l'**algorithme 4** ne produit pas toujours un couplage stable.

correction

L'appariement obtenu à la question précédente n'est pas stable, car $u_1 >^{v_3} u_3$, et $v_3 >^{u_1} v_2$.

On s'intéresse alors à une autre forme de stabilité pour assurer à la fois une propriété de Pareto et une stabilité dans les couplages.

Définition 4 (P-stabilité)

Un couplage A est P -stable s'il n'existe aucune paire $(v, v') \in V^2$ telle que v préfère $m_A(v')$ et v' préfère $m_A(v)$.

Q30. Montrer par l'absurde que l'algorithme 4 produit un couplage P -stable.

correction

Raisonnons par l'absurde, et raisonnons par l'absurde, en supposant que l'algorithme 4 peut produire un couplage P -stable, i.e. dans lequel il existe une paire (v, v') telle que v préfère $m_A(v')$ à $m_A(v)$, et v' préfère $m_A(v)$ à $m_A(v')$.

Distinguons plusieurs cas :

- $(v, m_A(v))$ est apparu dans un cycle strictement avant $(v', m_A(v'))$. Mais alors, lors du choix de $(v, m_A(v))$, $m_A(v')$ était encore présent dans le graphe, et comme v préfère $m_A(v')$ à $m_A(v)$, l'arc $(v, m_A(v))$ ne pouvait être présent à ce moment là : absurde.
- De même dans le cas symétrique.
- Si les deux appariements sont apparus à la même itération, c'est que les 4 sommets $v, v', m_A(v)$ et $m_A(v')$ étaient présents dans le même graphe, ce qui est à nouveau absurde pour la même raison que précédemment.

Q31. Montrer que l'algorithme 3 ne produit pas nécessairement un couplage P -stable.

correction

On reprend le résultat renvoyé pour $\mathcal{D}$ par cet algorithme : $\{(v_1, u_1), (v_2, u_2), (v_3, u_3)\}$. Ce couplage n'est pas P -stable car v_1 préfère $u_2 = m_A(v_2)$ à $u_1 = m_A(v_1)$, et réciproquement.

III.3 - Implémentation

Dans la suite, les individus sont numérotés de 0 à $n-1$. Ainsi $V = \{0, \dots, n-1\}$ et $U = \{0, \dots, n-1\}$. On suppose disposer de deux matrices de taille $n \times n$ L_v et L_u donnant respectivement les listes de préférences des éléments de V et U : ainsi, $L_v[i][j]$ est le j -ème élément de V préféré par l'élément $i \in U$.

On définit ces tableaux en C par l'intermédiaire du code suivant :

```
/* La directive #define est utilisée pour définir une valeur pour la ←
   constante n qui
   servira à déclarer des tableaux de taille fixe.*/
#define n 4
int main(void) {
    int L_v[n][n], L_u[n][n];
    (...)
}
```

III.3.1 - Algorithme de Gale-Shapley

Pour connaître rapidement le classement des éléments de V dans les listes de préférences des éléments de U , on définit un tableau **Rang** de taille $n \times n$, tel que **Rang** $[i][j]$ donne le rang de $j \in V$ dans la liste de préférences de l'élément $i \in U$. Par convention, les rangs commencent à 0.

Q32. Écrire une fonction de prototype

`void calcule_rang (int Lu[n][n], int Rang[n][n])` qui construit le tableau Rang.

correction

```
1 void calcule_rang(int Lu[n][n], int Rang[n][n]) {  
2     for (int i = 0; i < n; i++)  
3         for (int j = 0; j < n; j++)  
4             {  
5                 Rang[i][Lu[i][j]] = j;  
6             }  
7 }
```

Afin de pouvoir ajouter ou supprimer rapidement des couples (v, u) au couplage parfait stable A , on définit deux tableaux `CoupleV` et `CoupleU` tels que `CoupleV[i]` ou `CoupleU[i]` est le partenaire actuel de i . On définit enfin un dernier tableau `USuiv` de taille n stockant pour chaque élément de V le rang du prochain élément de U à qui il peut faire une proposition.

Q33. Proposer des valeurs d'initialisation pour ces trois tableaux.

correction

`CoupleV[i] = CoupleU[i] = -1` – convention possible pour dire que les sommets correspondant ne sont pas appariés.
`USuiv[i] = 0`.

Q34. Avec ces structures de données, écrire une fonction de prototype `void gale_shapley(int Lv[n] [↔ n], int Lu[n][n], int CoupleU[n], int CoupleV[n])` qui réalise l'**algorithme 3**. On ne retournera pas le couplage parfait stable, on remplira seulement les variables `CoupleU` et `CoupleV` passées en argument. Pour le choix de v , on prendra les éléments de V par ordre croissant de leur numéro en choisissant à chaque fois le premier libre.

```

1 void gale_shapley(int Lv[n][n], int Lu[n][n], int CoupleU[n], int ←
    CoupleV[n]) {
2     int Rang[n][n];
3     calcule_rang(Lu, Rang);
4
5     int Usuiv[n];
6     for (int i = 0; i < n; i++) {
7         CoupleV[i] = CoupleU[i] = -1;
8         Usuiv[i] = 0;
9     }
10    int nb_app = 0; // nombre de v appariés
11    while (nb_app < n) {
12        int v = 0;
13        while (CoupleV[v] != -1) v++;
14        // v contient le premier v non apparié
15        int u = Usuiv[v];
16        Usuiv[v]++;
17        if (CoupleU[u] == -1) {
18            CoupleU[u] = v;
19            CoupleV[v] = u;
20            nb_app++;
21        }
22        else if (Rang[u][v] < Rang[u][CoupleU[u]]) {
23            CoupleV[CoupleU[u]] = -1; // on dés-apparie v' = CoupleU[u]
24            CoupleU[u] = v;
25            CoupleU[v] = u;
26        }
27    }
28 }

```

III.3.2 - Algorithme de couplage par graphe orienté

Sans explicitement coder tout l'algorithme 4 qui requiert de rechercher les circuits dans $\mathcal{G}$, on se propose de coder quelques fonctions utiles à la réalisation de l'algorithme.

Pour pouvoir numéroté les sommets du graphe de manière continue, les individus sont maintenant numérotés de 0 à $2n - 1$ de la manière suivante : $V = \{0, \dots, n - 1\}$ et $U = \{n, \dots, 2n - 1\}$.

On code le graphe orienté $\mathcal{G}$ par listes d'adjacence. On propose les types suivants :

```

struct noeud {
    int individu; // Element de U union V
    struct noeud* suivant;
};
typedef struct noeud *liste;
struct graphe_s {
    int nb_sommets;
    liste *liste_adjacence;
};
typedef struct graphe_s graphe;

```

Q35. Écrire une fonction de prototype

`void ajoute_arc (graphe *g, int i, int j)` qui ajoute l'arc (i, j) au graphe $\mathcal{G}$.

correction

Le type proposé «overshoote» largement dans la mesure où chaque sommet du graphe considéré a exactement un sommet sortant...

```
1 void ajoute_arc (graphe *g, int i, int j) {
2     liste cell = malloc(sizeof(liste));
3     cell->individu = j;
4     cell->suivant = g->liste_adjacence[i];
5     g->liste_adjacence[i] = cell;
6 }
```

Q36. Écrire une fonction de prototype

`void supprime_arc (graphe *g, int i, int j)` qui supprime l'arc (i, j) du graphe $\mathcal{G}$.

correction

```
1 void supprime_arc (graphe *g, int i, int j) {
2     liste cell = g->liste_adjacence[i];
3     if (cell -> individu == j) g->liste_adjacence[i] = g->↵
        liste_adjacence[i] -> suivant;
4     else {
5         while (cell != NULL && cell -> suivant -> individu != j) cell = ↵
            cell -> suivant;
6         if (cell != NULL) cell -> suivant = cell -> suivant -> suivant;
7     }
8 }
```

Q37. Écrire une fonction de prototype

`void supprime_sommet (graphe *g, int s)`

qui supprime le sommet s du graphe $\mathcal{G}$. Pour ce faire, on supprimera uniquement tous les arcs dont ce sommet est origine ou destination.

correction

```
1 void supprime_sommet (graphe *g, int s) {
2     g->liste_adjacence[s] = NULL;
3     for (int i = 0; i < n; i++) {
4         supprime_arc(g, i, s); // supprime_arc prend bien en compte le ↵
            cas où l'arc est absent
5     }
6 }
```

Q38. Écrire une fonction de prototype

`graphe *construit_graphe (int Lv[n][n], int Lu[n][n])`

qui construit le graphe $\mathcal{G}$ utilisé dans l'algorithme 4 et dont la définition est donnée juste avant la Q 26. On

prendra bien garde à la numérotation des éléments de U .

correction

```
1 graphe *construit_graphe (int Lv[n][n], int Lu[n][n]) {
2     graphe* g = malloc(sizeof(graphe));
3     g-> nb_sommets = 2*n;
4     g-> liste_adjacence = malloc(2*n * sizeof(liste));
5     for (int i = 0; i < n; i++) {
6         ajoute_arc(g, i, n + Lv[i][0]);
7     }
8     for (int i = n; i < 2*n; i++) {
9         ajoute_arc(g, i, Lv[i - n][0]);
10    }
11    return g;
12 }
```

FIN