Chapitre 12 : Compléments de théorie des graphes

Table des matières

1 Composantes fortement connexes

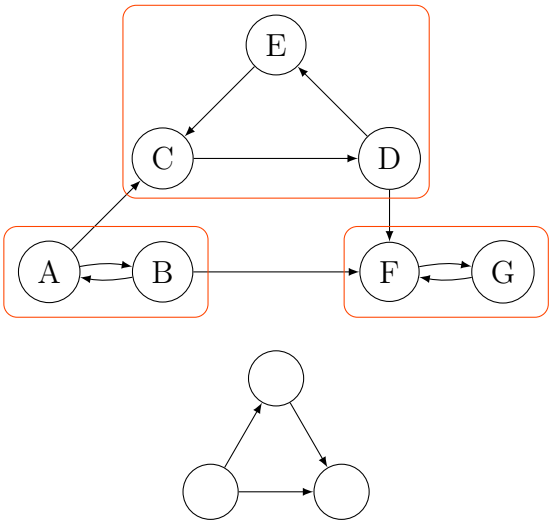
1.1 Algorithme de Kosaraju

1.1.1 Rappels (*cf* chap 10, 2.2.2)

• Définition (*connexité forte*) : Soit $G = (S, A)$ un GO, et $s, s' \in S$.
On dit que s et s' sont *fortement connectés*, noté $s \sim_G s'$, $\Leftrightarrow$ il existe un chemin de s à s' et un chemin de s' à s.
G est dit *fortement connexe* $\Leftrightarrow \forall s, s' \in S, s \sim_G s'$.

• Définition (*composantes fortement connexes*) : Soit $G = (S, A)$ un GO, et $s \in S$.
La *composante fortement connexe* de G contenant s est la classe d'équivalence de s pour $\sim_G$.

• Exemple :



1.1.2 Remarque

Soit $G = (S, A)$ un GO.
On peut définir le graphe des composantes fortement connexes de G en plaçant un nœud par composante et en définissant l'ensemble des arcs comme l'ensemble des (C, C') avec C, C' des composantes distinctes telles que $\exists s \in C,\ s' \in C' \mid (s, s') \in A$.

Exo Mq le graphe des composantes fortement connexes est acyclique.

On en déduit par le TD_{30} que l'on peut effectuer un tri topologique de ce graphe.
Rappel : un tri topologique est un ordre sur les nœuds tel que $\forall (s, s')$ arc, $s \leqslant s'$.

Question : que se passe-t-il si on parcourt G en choisissant les points de départ successifs selon leur appartenance aux composantes fortement connexes dans l'odre topologique inverse ?

L'ensemble des sommets accessibles à partir d'un point de départ donné est constitué de la composante fortement connexe du point de départ et de sommets de composante déjà parcourus.

Les parcours successifs donnent les composantes.

Remarque : l'idée de l'algorithme de KOSARAJU est de déterminer cet ordre de parcours sans connaître les composantes, en exploitant les propriétés du parcours en profondeur.

1.1.3 Algorithme de KOSARAJU

Découvert de manière indépendante par KOSARAJU (1978) et SHAMIR (1981), cet algorithme permet de déterminer les composantes fortement connexes d'un GO en seulement deux parcours en profondeur.

Idée : l'ordre de traitement des sommets du premier parcours permet de déterminer le choix des points de départ pour le second, qui est effectué dans le graphe transposé.

• Définition (*graphe transposé*) : Soit $G = (S, A)$ un GO.

Le *graphe transposé* de G est le graphe $G^T = (S, A^T)$ où $A^T = \{(s', s) \mid (s, s') \in A\}$.

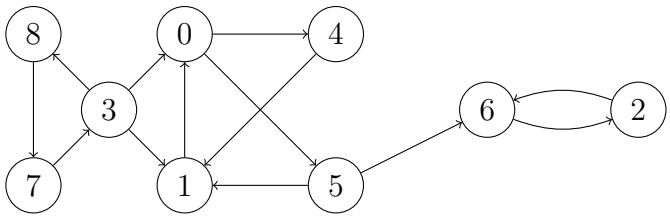
• Algorithme :

Entrée : $G = (S, A)$ un GO

> Ranger S par ordre de fin de traitement décroissante dans un parcours en profondeur de G
> Déterminer G^T
> Effectuer un parcours en profondeur de G^T en choisissant les points de départ successifs dans leur ordre d'apparition dans le tri à la première étape

Résultat : pour chaque point de départ, l'ensemble des sommets atteints constitue sa composante fortement connexe.

• Exemple :



(1) 387056241 (parcours en profondeur)

(3) (378)(0145)(62)

• Code OCaml (code C dans le TD_{30}) :

On utilise la représentation par listes d'adjacence :

```
type graphe = int list array

let rec visite (g : graphe) (l : int list) (acc : int list) (vu :
   bool array) : int list =
    match l with
    | [] -> acc
    | s::q when vu.(s) -> visite g q acc vu
    | s::q ->
        vu.(s) <- true;
        visite g q (s::visite g g.(s) acc vu) vu

let post_dfs (g : graphe) : int list =
    let vu = Array.make (Array.length g) false in
    let acc = ref [] in

    for i = 0 to Array.length g - 1 do
        if not vu.(i) then
            acc := visite g [i] !acc vu
    done;
    !acc;;

let transpose (g : graphe) : graphe =
    let gT = Array.make (Array.length g) [] in
    Array.iteri (
        fun i vi -> List.iter (
            fun j -> gT.(j) <- i::gT.(j)
        ) vi
    ) g;
    gT

(*Fonction Arry.iteri HP, iter + indice accessible*)

let kosaraju (g : graphe) : int list list =
    let l = ref (post_dfs g) in
    let gT = transpose g in
    let vu = Array.make (Array.length g) false in
    let res = ref []
    in
    while !l <> [] do
        if not vu.(List.hd !l) then
            res := (visite gT [List.hd !l] [] vu) :: !res;
        l := List.tl !l
    done;
    !res

```

• Complexité : deux parcours en $\mathcal{O}(|S| + |A|)$ + le calcul du graphe transposé en $\mathcal{O}(|S| + |A|)$.
Au total : $\mathcal{O}(|S| + |A|)$.

1.1.4 Correction de l'algorithme de Kosaraju

Soit $G = (S, A)$ un GO.

• Pour chaque composante fortement connexes C de G, on note fin(C) le maximum des dates de fin de traitement de ses éléments dans le premier parcours.

> *Lemme* : Soient C, C' des composantes fortement connexes de G distinctes.
> Si $\exists (s, s') \in A \mid (s, s') \in C \times C'$, alors $\text{fin}(C) > \text{fin}(C')$.

☐ Démonstration :

On note $u \in C \cup C'$ le premier sommet parmi ceux de $C \cup C'$ rencontré lors du parcours.

Si $u \in C$: par définition de C, tout sommet de C est accessible depuis u.

En particulier s.

Comme $(s, s') \in A$ et comme tout sommet de C' est accessible depuis s', tout sommet de C' est aussi accessible depuis u.

Comme aucun sommet de $(C \cup C') \setminus \{u\}$ n'a été vu lorsqu'on atteint la date de fin de traitement de u est supérieure à celle de tous les autres sommets de $C \cap C'$ donc $\text{fin}(C) > \text{fin}(C')$.

Si $u \in C'$:

Comme le graphe des composantes fortement connexes est acyclique, aucun sommet de C n'est accessible à partir de u.

Donc tous les sommets de C' sont traités avant même d'avoir atteint un sommet de C donc $\text{fin}(C) > \text{fin}(C')$ ■

• Exo G^T et G on les mêmes composantes connexes.

• Théorème :

> *Théorème* : les arborescences du second parcours en profondeur dans l'algorithme de KOSARAJU donnent les composantes fortement connexes de G.

☐ Démonstration :

On montre par récurrence finie forte que les k premiers sous-parcours sont des composantes fortement connexes de G.

• Initialisation : $k = 0 \to 0$ ensemble $\to$ ok.

• Hérédité : on suppose que les k premiers parcours sont des composantes fortement connexes, on note s le point de départ du $(k+1)$-ème parcours, et C la composante fortement connexe de G contenant s.

Par hypothèse de récurrence, on n'a visité que des composantes fortement connexes de G, donc aucun sommet de C n'a déjà été rencontré.

Donc C est inclus dans le parcours lancé à partir de s.

Si $\exists s' \in S \setminus C$ dans l'ensemble des sommets atteints, alors il existe un chemin $s \rightsquigarrow u \to v \rightsquigarrow s'$ constitué uniquement de sommets pas encore rencontrés dont on distingue les sommets u et v tels que $u \in C$, $v \notin C$ et (u, v) est le premier arc à sortir de C.

Comme $(v, u) \in A$ ($(u, v) \in A^T$), le lemme donne $\text{fin}(C') > \text{fin}(C)$.

Donc selon l'ordre de traitement des sommets dans le parcours, un sommet de C' a déjà été considéré avant de choisir s.

Donc C' a déjà été parcourue et v a déjà été vu : absurde.

• Comme tous les sommets sont rencontrés dans le parcours, toutes les composantes fortement connexes on bien été visitées. ■

1.2 Application : résolution de 2-SAT

1.2.1 Rappels (*cf* chap.8)

• Un *littéral* est une variable ou la négation d'une variable ;

• Une *clause disjonctive* est une disjonction de littéraux ;

• Une formule est en FNC (Forme Normale Conjonctive) si c'est une conjonction de clauses ;

• 2-SAT est le problème de décision suivant : étant donné une FNC dont les clauses ont deux littéraux, est-elle satisfiable ?

1.2.2 Objectif

Nous allons étudier un algorithme permettant de résoudre 2-SAT en temps polynomial en la taille de la formule.

Idée : on étudie les composantes fortement connexes du graphe d'implication de la formule.

1.2.3 Définition (*graphe d'implication*)

Soit φ une FNC.

Le *graphe d'implication* de φ est le graphe $G_\varphi = (S_\varphi, A_\varphi)$ défini comme suit :

$$S_\varphi = \bigcup_{x \in \text{Vars}(\varphi)} \{x, \neg x\}$$

$$A_\varphi = \{(l, l') \mid l, l' \text{ littéraux tel qu'il existe une clause de } \varphi \text{ équivalente à } l \to l'\}$$

Remarque :

$x \vee y \equiv \neg x \to y \equiv \neg y \to x$

$\neg x \vee y \equiv x \to y \equiv \neg y \to \neg x$

$\neg x \vee \neg y \equiv x \to \neg y \equiv y \to \neg x$

1.2.4 Résolution de 2-SAT

• Algorithme :

Entrée : une 2-FNC φ

Pseudo-code :

$V \leftarrow \text{Vars}(\varphi)$
$S_\varphi \leftarrow \bigcup_{x \in V} \{x, \neg x\}$
$A_\varphi \leftarrow \varnothing$

Pour chaque clause c dans φ :
    Ajouter dans A_φ les deux arcs équivalents à c

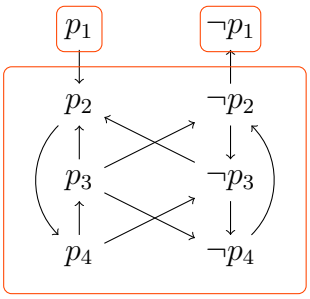
Déterminer les composantes fortement connexes de $G_\varphi = (S_\varphi, A_\varphi)$

Pour chaque $x \in V$:
    Si x et $\neg x$ appartient à la même composante fortement connexe :
        Renvoyer « Impossible »

Renvoyer « Satisfiable »

• Exemple :

$$(p_3 \vee \neg p_4) \wedge (p_2 \vee \neg p_3) \wedge (p_4 \vee \neg p_2) \wedge (\neg p_1 \vee p_2) \wedge (\neg p_4 \vee p_3) \wedge (p_3 \vee p_2)$$



Donc insatisfiable.

• Complexité : si φ a n variables et m clauses, on calcule V, S_φ, A_φ en temps $\mathcal{O}(n+m)$.
$|A_\varphi| = 2n$, $|A_\varphi| = 2m$, donc le calcul des composantes fortement connexes se fait en temps $\mathcal{O}(n+m)$ (par l'algorithme de Kosaraju).
Modulo un prétraitement en $\mathcal{O}(n)$, la dernière boucle s'exécute en temps $\mathcal{O}(n)$.
Au total, la complexité de l'algorithme est en $\mathcal{O}(n+m)$.

1.2.5 Correction de l'algorithme

Soit φ une 2-FNC.
On veut montrer que φ est satisfiable ssi $\nexists x \mid x$ et $\neg x$ sont dans la même composante fortement connexe de g_φ.

Lemme : Si φ est satisfiable et si v est une valuation telle que $[\![\varphi]\!]_v = \top$
Alors pour toute composante fortement connexe C de G_φ et $\forall l, l' \in C$, $[\![l]\!]_v = [\![l']\!]_v$.

□ Démonstration :
    Il existe dans G_φ un chemin $l \rightsquigarrow l'$ et un chemin $l' \rightsquigarrow l$.
    Notons $l_0 \dots l_k$ un chemin $l \rightsquigarrow l'$.

Par définition, $\forall i \in [\![0\ ;\ k-1]\!]\ ,\ \exists c_i \in \varphi$ une clause équivalente à $l_i \to l_{i+1}$.
Or $[\![\varphi]\!]_v = \top$, donc $[\![c_i]\!]_v = \top$ donc $[\![l_i \to l_{i+1}]\!]_v = \top$.
Par transitivité, $[\![l' \to l]\!]_v = \top$, donc $[\![l \leftrightarrow l']\!]_v = \top$.
D'où $[\![l]\!]_v = [\![l']\!]_v$ ■

> *Corollaire* : Si φ est satisfiable, alors il n'existe pas de variable x telle que x et $\neg x$ sont dans la même composante fortement connexe de G_φ.

☐ Démonstration :

Par contraposée, si x et $\neg x$ sont dans la même composante, $\forall v$ valuation telle que $[\![\varphi]\!]_v = \top$, $\underbrace{[\![x]\!]_v = [\![\neg x]\!]_v}_{\text{impossible}}$ d'après le lemme.

Donc il n'existe pas de valuation v telle que $[\![\varphi]\!]_v = \top$.

Donc φ n'est pas satisfiable ■

> *Proposition* : S'il n'existe pas x tel que x et $\neg x$ sont dans la même composante fortement connexe de G_φ, alors φ est satisfiable.

☐ Démonstration :

On remarque que si $C = \{l_i \mid i \in [\![1\ ;\ k]\!]\}$ est une composante fortement connexe de G_φ, alors $\neg C = \{\neg l_i \mid i \in [\![1\ ;\ k]\!]\}$ en est une aussi Exo.
On note aussi que si il existe un arc $C \to C'$ dans le graphe des composantes fortement connexe, alors il existe aussi un arc $\neg C' \to \neg C$.

On construit alors un modèle v de φ en considérant les composantes dans l'ordre topologique inverse : lorsqu'on considère une composante C dont les littéraux sont sans valeur par v, on prolonge v de telle sorte que tous les littéraux de C prennent la valeur $\top$ (donc ceux de $\neg C$ prennent la valeur $\bot$).

Par choix de l'ordre topologique inverse, pour toute composante C dont les littéraux ont la valeur $\top$, on ne peut atteindre depuis C que des composantes dont les littéraux ont la valeur $\top$ et pour toute composante C dont les littéraux ont la valeur $\bot$, C ne peut être atteinte que depuis des composantes dont les littéraux ont la valeur $\bot$.

On le montre par induction bien fondée : lorsque l'on fixe la valeur des littéraux de C à $\top$, on ne peut pas atteindre depuis C une composante C' dont les littéraux ont la valeur $\bot$.
En effet, un chemin $C \rightsquigarrow C'$ implique un chemin $\neg C' \rightsquigarrow \neg C$.
Lorsque $\neg C'$ a été choisie, cela contredit l'ordre topologique inverse car $\neg C$ n'avait pas de valeur.
De même, il est impossible d'atteindre une composante dont les littéraux sont sans valeur depuis C.
Par symétrie, on ne peut atteindre $\neg C$ que depuis des composantes dont les littéraux ont la valeur $\bot$.

Ainsi, v est choisie de telle sorte que toute implication déduite de φ est de la forme $\bot \to \bot$ $\quad \bot \to \top$ ou $\top \to \top$ donc toutes les clauses sont satisfaites.
On remarque que v est bien une valuation : chaque prolongement définit simultanément une valeur de vérité pour un ensemble de littéraux qui sont non contradictoires car on ne peut avoir une variable et sa négation dans le même ensemble. ■

2 Arbres couvrant de poids minimum

2.1 Problème

2.1.1 Rappels (*cf* chap.10 4.1)

• Définition (*sous-graphe induit / couvrant*) :
Soit $G = (S, A)$ un graphe.
– Le sous-graphe de G induit par $T \subseteq S$ est le graphe $G_T = (T, A_T)$ où

$$A_T = \{a \in A \mid \text{les extrémités de } a \text{ sont dans } T\}$$

– Le sous-graphe de G induit par $A' \subseteq A$ est le graphe $G_{A'} = (S', A')$, où

$$S' = \{s \in S \mid \exists a \in A' \mid s \text{ est une extrémité de } a\}$$

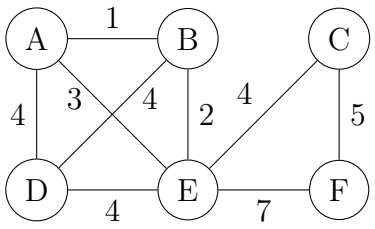
– Un sous-graphe (S', A') de G est *couvrant* $\Leftrightarrow$ $S' = S$.

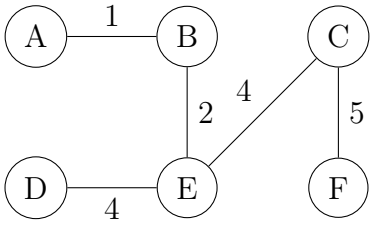
• Remarque : un parcours d'un graphe permet de définir une forêt couvrante du graphe, cette forêt est une arbre si le graphe est connexe.

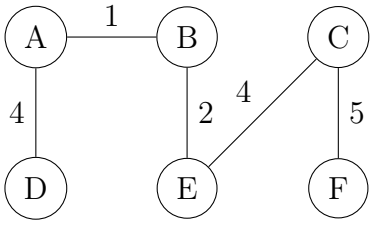
2.1.2 Objectif

Soit $G = (S, A, w)$ un GNO pondéré connexe.
On veut déterminer un sous-arbre $T = (S, A')$ qui minimise $w(T) = \sum_{a \in A'} w(a)$.

Exemple :







2.1.3 Proposition

Soit $G = (S, A, w)$ un GNO pondéré connexe.

Si w est injective, alors G admet un unique arbre couvrant de poids minimal.

☐ Démonstration :

Par contraposée, supposons deux arbres couvrants de poids minimum distincs.

$T_0 = (S, A_0)$ et $T_1 = (S, A_1)$

$\varnothing \neq A_0 \Delta A_1 = (A_0 \setminus A_1) \cup (A_1 \setminus A_0)$ (différence symétrique).

Donc $\exists e \in A_0 \Delta A_1$ de poids minimum.

Sans perte de généralité, on suppose $e \in A_0$.

$T_1 \cup \{e\} = (S, A_1 \cup \{e\})$ contient un cycle car T_1 est un arbre (*cf* chap 10 2.1.14)

En retirant une arête de ce cycle, on obtient de nouveau un arbre couvrant.

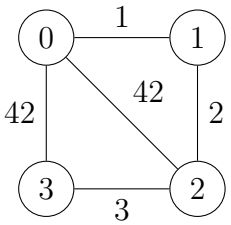
Comme T_0 est un arbre, ce cycle contient une arête $e' \in A_1 \setminus A_0 \subseteq A_0 \Delta A_1$ donc $w(e) \leqslant w(e')$.

Le nouvel arbre obtenu est alors de poids $w(T_1) + w(e) - w(e') \leqslant w(T_1)$

Par minimalité de T_1, on en déduit $w(e) = w(e')$, donc w n'est pas injective ($e \neq e'$ par définition). ■

2.1.4 Remarques

• La réciproque est fausse :



• Si G est un arbre, il admet un unique arbre couvrant, à savoir G lui-même.

• Il existe plusieurs algorithmes permettant de déterminer un arbre couvrant de poids minimum d'un graphe donné. Ces algorithmes sont souvent gloutons et reposent sur une idée commune : on sélectionne une à une, par des choix gloutons, les arêtes qui appartiennent à l'arbre : on peut le faire de plusieurs manières : on peut par exemple

se focaliser sur la connexité du sous-graphe en cours de construction, ou alors sur la minimalité du poids des arêtes sélectionnées.

L'algorithme de KRUSKAL (le seul au programme) repose sur la seconde idée.

On construit une forêt couvrante du graphe qui devient progressivement un unique arbre par sélection d'arêtes.

L'ajout d'une arête pouvant créer un cycle, il faut pouvoir le détecter efficacement. Il existe une structure de données adaptée pour cela : cette structure permet de représenter une forêt et de vérifier efficacement si deux sommets appartiennent au même sous-arbre.

2.2 Structure *union et trouver* (*union-find*)

2.2.1 Définition (*union et trouver*)

Soit E un ensemble.

Une structure *union et trouver* sur E est une structure de données servant à représenter une famille finie de sous-ensembles finis et disjoints de E, chacun muni d'un représentant unique. Un telle structure est munie des opérations suivantes :

- `create` : étant donné $x \in E$ qui n'appartiennent à aucun des ensembles de la structure, crée l'ensemble $\{x\}$ (représentant de x) ;
- `union` : étant donné $x, y \in E$ appartenant aux ensembles E_x et E_y, remplace E_x et E_y par $E_x \cup E_y$ et choisit un représentant pour cet ensemble ;
- `find` : étant donné $x \in E$ appartenant à l'ensemble E_x, renvoie le représentant de E_x.

2.2.2 Remarques

- Cette structure est adaptée pour construire incrémentalement une partition d'un ensemble ou alors une relation d'équivalence, comme par exemple les composantes connexes d'un graphe, que l'on peut calculer ainsi : on utilise l'opération `create` sur chaque sommet, puis on parcourt les arêtes et pour chaque arête, si les représentants de ses extrémités sont distincts, alors on fusionne les classes. Cela revient à maintenir l'ensemble des composantes connexes d'un sous-graphe couvrant qui n'a pas d'arête initialement et auquel on ajoute progressivement toutes les arêtes du graphe.

- Pour évaluer l'efficacité d'une implémentation d'une telle structure, on étudie en général la complexité totale d'un ensemble de m opérations sur une famille d'ensembles contenant en tout n éléments. Cela signifie qu'il y a exactement n opérations `create` et que $m \geqslant n$. On pourra supposer que les opérations `create` sont effectuées avant les autres.

2.2.3 Première implémentation

On présente chaque ensemble par la liste chaînée de ses éléments. Le représentant d'un ensemble est la tête de la liste associée. L'union se fait par concaténation des listes. On utilise des listes doublement chaînées pour pouvoir remonter jusqu'à la tête.

```
typedef struct maillon {
    int val;
    struct maillon* prev;
    struct maillon* next;
} element ;

typedef element* ensemble;
```

La création d'un ensemble se fait en temps constant par allocation d'un unique maillon.

La recherche du représentant d'un élément se fait en temps linéaire en la taille de l'ensemble parce qu'il faut remonter les pointeurs jusqu'à la tête.

L'union se fait en mettant à jour mes pointeurs du dernier élément d'un ensemble et du premier de l'autre.

```
void union(element* x, element* y) {
    element* last_x, first_y;
    last_x = x->next;

    while (last_x != NULL) {
        x = last_x;
        last_x = x->next;
    }

    first_y = y->prev;

    while (first_y != NULL) {
        y = first_y;
        first_y = y->prev;
    }

    x->next = y;
    y->prev = x;
}
```

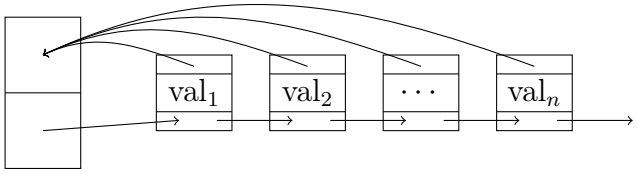
Ici le représentant de l'union est le représentant de x.

La fonction `union` est de complexité $\mathcal{O}(|E_x| + |E_y|)$

Remarque : on peut construire une séquence de $2n-1$ opérations de complexité totale $\mathcal{O}(n^2)$ Exo

2.2.4 Optimisation

On peut obtenir une opération find de complexité $\mathcal{O}(1)$ en construisant une structure pointant vers le premier et le dernier élément de la liste et en gardant un pointeur de chaque élément vers la structure.



L'opération find se fait en temps constant en suivant deux pointeurs (le premier vers l'ensemble et le second vers la tête de la liste).

L'union se fait encore par concaténation. On accède bien au premier élément d'un ensemble et au dernier de l'autre en temps constant, mais il faut mettre à jour tous les pointeurs des éléments de l'ensemble qui finit en queue de liste pour pointer vers l'autre ensemble.

Exo On peut encore construire une séquence de $2n-1$ opérations de complexité totale $\mathcal{O}(n^2)$.

Remarque : on peut faire mieux en utilisant l'heuristique de l'union pondérée qui consiste à toujours placer en queue de liste l'ensemble de plus petit cardinal.

Théorème :

> *Théorème* : avec l'heuristique de l'union pondérée, une séquence de m opérations sur des ensembles totalisant n éléments est de complexité $\mathcal{O}(m + n\log n)$

□ Démonstration :

On remarque qu'il y a au plus $n-1$ opérations d'union, car il y a en tout n éléments.

Le coût d'une union est défini oar la mise à jour des pointeurs des éléments du plus petit des deux ensembles vers le premier ensemble.

Étant donné un élément x, évaluons le nombre de fois que ce pointeur sera mis à jour. Lors d'une mise à jour, le pointeur de x n'est mis à jour que s'il appartient au plus petit des deux ensembles.

Donc dans une telle union, le cardinal de l'ensemble contenant x double (au moins).

Or un ensemble contient au plus n éléments.

Donc le pointeur de x est mis à jour au plus $\mathcal{O}(\log n)$ fois.

La totalité des unions est donc de complexité $\mathcal{O}(n\log n)$.

Il reste les opérations `create` et `find`, de complexité $\mathcal{O}(1)$, qui sont répétées au plus m fois.

D'où une complexité $\mathcal{O}(m + n\log n)$ ■

2.2.5 Implémentation par forêt

On utilise des arbres pour représenter les ensembles, la racine de l'arbre étant le représentant de l'ensemble.

La création d'un ensemble est la création d'un arbre à un seul nœud.

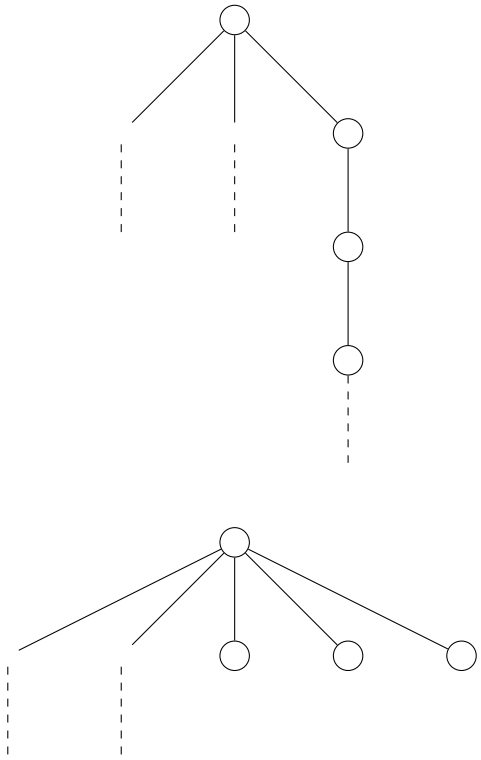
L'opération `find` consiste à remonter les ancêtres d'un nœud donné jusqu'à la racine.

L'opération d'union consiste à placer l'une des deux racines comme fille de l'autre.

Remarque : sans plus de précision, on peut construire une séquence d'opérations faisant que les arbres sont tous des listes chaînées, donc ce n'est pas mieux que les essais précédents.

On peut utiliser deux heuristiques pour améliorer la situation.

La première, appelée *compression de chemin*, consiste à remplacer le père de chaque nœud rencontré lors d'une opération `find` par la racine.



La seconde, appelée *union par rang*, est semblable à l'union pondérée des listes.

La différence est qu'on n'utilise pas le cardinal de l'ensemble mais le rang des racines pour déterminer laquelle sera fille de l'autre.

Le rang est un majorant de la hauteur de l'arbre. On n'utilise pas la valeur exacte de la hauteur pour limiter le nombre d'opérations à effectuer lors de la compression de chemin.

Code :

```
struct noeud {
    struct noeud* pere;
    int val;
    int rang;
};

typedef struct noeud* element;

element create (int x) {
    element e = (element) malloc(sizeof(struct(noeud)));

    e->pere = NULL;
    e->val = x;
```

```
14      e->rang = 0;
15
16      return e;
17  }
18
19  element find(element e) {
20      if (e->pere == NULL)
21          return e;
22
23      e->pere = find(e->pere);
24
25      return e->pere;
26  }
27
28  void link(element x, element y) {
29      // On suppose que x et y sont des racines
30
31      if (x->rang > y->rang)
32          y->pere = x;
33
34      else if (x->rang < y->rang)
35          x->pere = y;
36
37      else
38          y->pere = x;
39          x->rang++;
40  }
41
42  void union(element x, element y) {
43      link(find(x), find(y));
44  }
```

Complexité de `create` : $\mathcal{O}(1)$

Complexité de `find` : $\mathcal{O}(p_e)$ où p_e est la profondeur du nœud e.

Complexité de `link` : $\mathcal{O}(1)$

Complexité de `union` : $\mathcal{O}(p_x + p_y)$

Théorème (admis) :

> *Théorème* : avec les heuristiques de l'union par rang et de la compression de chemin, une séquence de m opérations dont n `create` est de complexité totale
>
> $$\mathcal{O}(m\alpha(n))$$
>
> où α est définie comme suit :
>
> $$\forall n \in \mathbb{N},\ \alpha(n) = \min\{k \in \mathbb{N} \mid A_k(1) \geqslant n\}$$
>
> où
>
> $$\forall (k,j) \in \mathbb{N}^2,\ A_k(j) = \begin{cases} j+1 & \text{si } k = 0 \\ A_{k-1} \circ A_{k-1} \circ \cdots \circ A_{k-1}(j) & \text{sinon} \end{cases}$$

Remarque : en pratique, c'est linéaire en le nombre d'opérations.

En effet, $A_k(1) \gg 2^{2048} \gg 10^{80}$ (nombre d'atomes dans l'univers)

Donc $\alpha(n) \leqslant 4$ pour toutes les valeurs "pratiques" de n.

2.3 Algorithme de Kruskal

2.3.1 Rappel

L'algorithme de Kruskal est un algorithme glouton qui permet de calculer un arbre couvrant de poids minimum dans un graphe non orienté pondéré connexe (une forêt couvrante de poids min si le graphe n'est pas connexe).

Le principe de l'algorithme est de construire une forêt couvrante et de la faire évoluer en un unique arbre par sélection gloutonne d'arêtes.

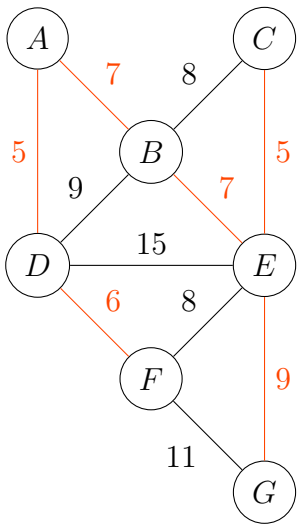
2.3.2 Algorithme

Entrée : $G = (S, A, w)$ un GNO pondéré (connexe)

Pseudo-code :

> $A' \leftarrow \varnothing$
> Pour tout $a \in A$ choisie par $w(a)$ croissant :
> Si $(S, A' \cup \{a\})$ est acyclique :
> $A' \leftarrow A' \cup \{a\}$
>
> Renvoyer (S, A')

2.3.3 Exemple



2.3.4 Proposition

Soit $G = (S, A, w)$ un GNO pondéré connexe.

Le sous-graphe de G produit par l'algorithme de Kruskal est un arbre couvrant de G

□ Démonstration :

Il est couvrant par définition.

On note (S, A') le résultat de l'algorithme.

Montrons que (S, A') est un arbre, *i.e* que c'est un GNO connexe acyclique.

"(S, A') est acyclique" est clairement un invariant de l'algorithme.

Soient $s, s' \in S$. Comme G est connexe, il existe un chemin $s_0 s_1 \dots s_n$ de s à s' dans G.

$\forall i \in [\![0\ ;\ n-1]\!]$, si $\{s_i, s_{i+1}\} \in A$, on la conserve.

Sinon, en notant A'' la valeur de la variable A' au moment où $\{s_i, s_{i+1}\}$ est rejetée, on sait que l'ajout de $\{s_i, s_{i+1}\}$ à A'' crée un cycle.

Donc il existe dans (S, A'') (donc aussi dans (S, A') car $A'' \subseteq A'$) un chemin de s_i à s_{i+1}.

On remplace alors dans le chemin initial $\{s_i, s_{i+1}\}$ par ce chemin.

Résultat : on a construit un chemin de s à s' dans (S, A'), donc (S, A') est connexe ■

2.3.5 Proposition

Soit $G = (S, A, w)$ un GNO pondéré connexe.

L'arbre couvrant de G produit par l'algorithme de KRUSKAL est de poids minimum.

□ Démonstration :

On note (S, A') le résultat de l'algorithme.

On va démontrer l'invariant "*il existe* $A_{\text{min}} \subseteq A \mid (S, A_{\text{min}})$ *est un arbre couvrant de poids minimum et* $A' \subseteq A_{\text{min}}$"

• Initialisation : au début, $A' = \varnothing$, donc tout arbre couvrant de poids minimum convient.

Il en existe un car G est connexe.

• Invariance : on suppose qu'à un certain point, $\exists A_{\text{min}} \subseteq A \mid (S, A_{\text{min}})$ est un arbre couvrant de poids minimum et $A' \subseteq A_{\text{min}}$.

On note $a \in A$ la prochaine arête considérée par l'algorithme.

– Si $(S, A' \cup \{a\})$ contient un cycle, on a toujours $A' \subseteq A_{\text{min}}$ et l'invariant reste vrai.

– Si $(S, A' \cup \{a\})$ est acyclique :

+ Si $a \in A_{\text{min}}$, alors A_{min} convient encore pour $A' \cup \{a\}$.

+ Si $a \notin A_{\text{min}}$: $(S, A_{\text{min}\cup\{a\}})$ contient un cycle car (S, A_{min}) est un arbre (d'après chap.10 2.2.14.)

Retirer une arête de ce cycle donne de nouveau un arbre (couvrant).

Les arêtes de ce cycle n'appartiennent pas toutes à $A' \cup \{a\}$ (sinon $(S, A' \cup \{a\})$ ne serait pas acyclique)

On note a' une arête du cycle telle que $a' \notin A' \cup \{a\}$.

Si on démontre que $w(a) \geqslant w(a)$, alors $(A_{\min} \cup \{a\}) \setminus \{a'\}$ conviendra pour démontrer l'invariant.

Si $w(a') < w(a)$, alors on a considéré a' avant a dans l'algorithme.

On note A'' la valeur de la variable A' à ce moment.

$a' \notin A'$, donc $(S, A'' \cup \{a'\})$ contient un cycle.

Or $A'' \subseteq A' \subseteq A_{\min}$ et $a' \in A_{\min}$.

Donc $A'' \cup \{a'\} \subset A_{\min}$ et $(S, A_{\min})$ contient un cycle.

Absurde car c'est un arbre !

• Optimalité de l'algorithme : l'invariant assure qu'en fin d'algorithme, il existe $A_{\min} \subseteq A \mid (S, A_{\min})$ est un arbre couvrant de poids minimum, et $A' \subseteq A_{\min}$.

Or par 2.3.4, (S, A') est un arbre couvrant.

Alors $A' = A_{\min}$ et (S, A') est un arbre couvrant de poids minimum.

En effet, si $A' \neq A_{\min}$, comme $A' \subseteq A_{\min}$, $\exists a \in A_{\min} \setminus A'$.

Comme a n'a pas été sélectionnée par l'algorithme, comme avant on sait que $(S, A' \cup \{a\})$ contient un cycle.

Donc comme $A' \cup \{a\} \subseteq A_{\min}$, $(S, A_{\min})$ ne peut être un arbre : absurde. ■

2.3.6 Implémentation

Pour parcourir les arêtes par poids croissant, on peut commencer par les trier (complexité $\mathcal{O}(|A| \log |A|)$) pour les parcourir ensuite, ou alors les insérer dans une file de priorité min où les priorités sont les poids et les extraire une à une (complexité $\mathcal{O}(|A| \log |A|)$).

Si on implémente l'algorithme naïvement, en testant l'algorithme du sous-graphe à chaque étape : il y a $|A|$ tests de complexité $\mathcal{O}(|S| + |A'|)$.

$|A'|$ vaut au plus $|S| - 1$ car un arbre à n sommets a $n - 1$ arêtes.

La complexité totale des tests est donc $\mathcal{O}(|A|\,|S|)$.

Au total, une telle implémentation serait de complexité $\mathcal{O}(|A| \log |A| + |A|\,|S|) = \mathcal{O}(|A| \log |S| + |A|\,|S|) = \mathcal{O}(|A|\,|S|)$ car $|A| = \mathcal{O}(|S|^2)$.

On peut essayer de faire mieux à l'aide d'une structure union et trouver. En effet, avec une implémentation par forêt, on représente directement (S, A').

Test d'acyclicité : Soit $\{u, v\} \in A$.

$(S, A' \cup \{u, v\})$ est acyclique $\Leftrightarrow$ u, v appartiennent au même arbre $\Leftrightarrow$ u, v ont le même représentant.

On peut donc réécrire ainsi l'algorithme de KRUSKAL :

> Pour tout $s \in S$:
> `créer` $\{s\}$
>
> Pour tout $\{u, v\} \in A$ choisie par $w(\{u, v\})$ croissant :
> Si `find(`u`)` $\neq$ `find(`v`)` :
> `union(`$\{u, v\}$`)`

> Renvoyer l'état de la structure union et trouver.

Complexité totale des opérations de la structure union et trouver :
On a $n = |S|$ opérations `create`, $2|A|$ opérations `find`, au plus $|S| - 1$ opérations `union`.
Il y a donc au plus

$$\begin{aligned} m &= |S| + 2|A| + |S| - 1 \\ &= 2(|S| + |A|) - 1 \\ &= \mathcal{O}(|S| + |A|) \end{aligned}$$

opérations.
Or G est connexe, donc $|A| \geqslant |A| - 1$, donc $n = \mathcal{O}(|A|)$.
Au total, les opérations sur la structure union et trouver sont de complexité $\mathcal{O}(|A|\,\alpha(|S|))$.
la complexité globale est alors $\mathcal{O}(|A|\log|A| + |A|\,\alpha(|S|)) = (\,|A|\log|S|)$ car $\alpha(|S|) = \mathcal{O}(\log|S|)$.

Code : on utilise la représentation par liste d'adjacences pondérées pour le graphe.

```c
typedef struct {
    int target;
    double weight;
} neighbour;

struct item {
    neighbour val;
    struct item* next;
};

typedef struct item* neigh_list;
typedef neight_list* graphe;

typedef struct {
    int source, target;
    double weight;
} edge;
```

Exo coder une fonction de prototype :

```c
void sort_edges(graphe g, int n, int* m, edge** edges);
```

qui prend un graphe `g` à n sommets et calcule le nombre d'arêtes et trie les arêtes dans le tableau pointé par `edges`.
On utilise le type `element` et les fonctions `create`, `find`, `union` vues en 2.2.5.

```c
element* kruskal(graphe g, int n) {
    element* forest = (element*) malloc(n * sizeof(element));

    for (int i = 0 ; i < n ; i++)
        forest[i] = create(i);

    int m;
    edge* edges;

```

```
    sort_edges(g, n, &m, &edges);

    for (int i = 0 ; i < m ; i++) {
        int u = edges[i].source, v = edges[i].target;

        if (find(forest[u]) != find(forest[v]))
            union(forest[u], forest[v]);
    }

    free(edges);

    return forest;
}

```

3 Couplages

3.1 Définitions et problème

3.1.1 Définition (*couplage*)

Soit $G = (S, A)$ un GNO, et $M \subseteq A$.
M est un *couplage* dans $G \Leftrightarrow$

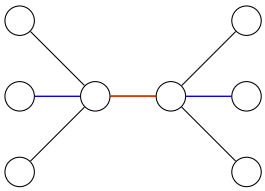
$$\forall a, a' \in M \mid a \neq a',\ a \cap a' = \varnothing$$

i.e $\Leftrightarrow$ les arêtes de M sont deux à deux non adjacentes.

Un couplage M est *maximum* $\Leftrightarrow$ M est de cardinal maximum.

3.1.2 Remarques

• Ne pas confondre maximum et maximal (au sens de l'inclusion)



M maximal
M' maximum

• La recherche d'un couplage maximum a plusieurs applications, comme l'étude de la structure chimique de certaines molécules (liaisons doubles), la résolution du problème d'affectation (associer de manière optimale des agents à des tâches), la résolution approchée du problème du voyageur de commerce (trouver un plus court chemin passant

exactement une fois par chaque ville d'un plan) par l'algorithme de CHRISTOFIDES qui nécessite aussi la recherche d'un arbre couvrant de poids minimum.

• Le programme se limite à la recherche d'un couplage maximum dans un GNO biparti.

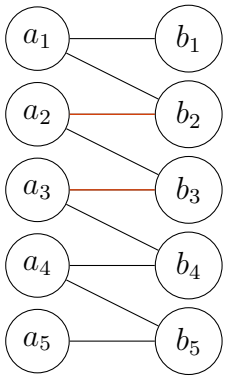
3.1.3 Définition (*chemin alternant / augmentant*)

Soit $G = (S, A)$ un GNO, $M \subset A$ un couplage de G, et $s_0 \dots s_n$ un chemin.

• On dit que $s_0 \dots s_n$ est *alternant* $\Leftrightarrow$ ce chemin alterne entre des arêtes de M et des arêtes de $A \setminus M$, *i.e* soit $\forall i \in [\![0\ ;\ n-1]\!]\,,\ \{s_i, s_{i+1}\} \in M \quad \Leftrightarrow \quad i \in 2\mathbb{N}$, soit $\forall i \in [\![0\ ;\ n-1]\!]\,, \{s_i, s_{i+1}\} \notin M$.

• On dit que $s_0 \dots s_n$ est *augmentant* $\Leftrightarrow$ ce chemin est alternant et s_0 et s_n sont libres, *i.e* $\forall a \in M,\ s_0 \notin a$, et $s_n \notin a$.

3.1.4 Exemple



$M = \{\{a_2, b_2\}, \{a_3, b_3\}\}$
$a_1, b_1, a_4, b_4, a_5, b_5$ sont libres.
$a_2b_2, a_1b_2a_2, a_1b_2a_2b_3$ sont alternant
$a_1b_2a_2b_1a_3b_4, a_4b_4$ sont augmentant.

3.1.5 Exercice

Soit $G = (S, A)$ un GNO, $M \subseteq A$ un couplage, et $s_0, \dots, s_n$ un chemin augmentant. Alors n est impair, et on si note $E = \{\{s_i, s_{i+1}\} \mid i \in [\![0\ ;\ n-1]\!]\}$, on a

$$|E \setminus M| = 1 + |E \cap M|$$

3.1.6 Théorème

Soit $G = (S, A)$ un GNO, et $M \subseteq A$ un couplage.

> M est maximum $\Leftrightarrow$ M n'admet aucun chemin augmentant.

□ Démonstration :

On montre plutôt que M n'est pas maximum $\Leftrightarrow$ M admet un chemin augmentant.

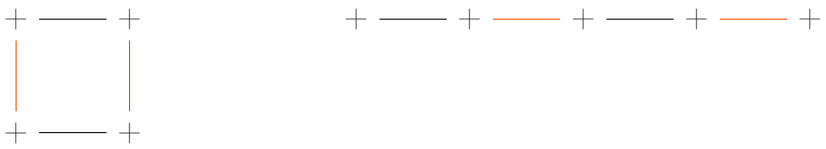
$\boxed{\Rightarrow}$ On considère M' un couplage maximum (existe car il existe un nombre fini non nul de couplages).

On considère le sous-graphe $G' = (S', A')$ induit par $A' = M \Delta M'$.

Chaque sommet de G' est de degré au plus 2 car il ne peut être extrémité que d'une seule arête de M et d'une arête de M'.

Tout chemin élémentaire dans G' alterne entre des arêtes de M et de M', donc est un chemin alternant dans G pour M.

Les composantes connexes de G' sont de plus soit des cycles simples de longueur paire, soit des chemins simples.



C'est vrai grâce aux propriétés des couplages et à la condition sur les degrés dans G'.

Comme M' est maximum et M ne l'est pas, on a

$$|M'| > |M|$$

et $|M' \setminus M| > |m \setminus M'|$

un cycle de longueur paire dans G' contient autant d'arêtes de $M' \setminus M$ que de $M \setminus M'$.

Il existe donc une composante connexe de G' qui est un chemin simple contenant une arête de plus de $M' \setminus M$ que de $M \setminus M'$.

Ce chemin est augmentant dans G par M : on sait qu'il est alternant et ses extrémités sont libres.

En effet, la première et la dernière arête de ce chemin appartient à $M' \setminus M$, et il n'y a pas d'arête de M incidente à l'une des extrémités (sinon, elle appartient à $M \setminus M'$ et aurait du être considérée dans G').

$\boxed{\Leftarrow}$ On note $s_0 \dots s_n$ un chemin augmentant, et $E = \{\{s_i, s_{i+1}\} \mid i \in [\![0\ ;\ n-1]\!]\}$

On considère $M' = M \Delta E$.

$$|M'| = |M \setminus E| + |E \setminus M| = |M| + 1$$

($|E \setminus M| = |E \cap E| + 1$ d'après 3.1.5, page 22)

Il reste à montrer que M' est un couplage.

Soit $s \in S$, montrons qu'il existe au plus une arête $a \in M' \mid s\ ina$.

Si $s \notin \{s_0, \dots, s_n\}$, alors il existe au plus une arête $a \in M \mid s \in a$, et $s \notin a' \; \forall a' \in E$: ok.

Si $s \in \{s_0, s_n\}$, alors s est libre pour M, et il existe une unique arête de E qui contient s ($\{s_0, s_1\}$) ou $\{s_{n-1}, s_n\}$: ok.

Si $s \in \{s_0, \dots, s_{n-1}\}$, $\exists i \in [\![1 \; ; \; n-1]\!] \mid s_n s_i$

s appartient à une unique arête de $M \cap E$ (soit $\{s_{i-1}, s_i\}$, soit $\{s_i, s_{i+1}\}$) qui est remplacée dans M' par l'unique arête de $E \setminus M$ incidente à s (soit $\{s_i, s_{i+1}\}$, soit $\{s_{i-1}, s_i\}$).

Donc M' est un couplage tel que $|M'| > |M|$, donc M n'est pas maximum ■

3.2 Algorithme de recherche de couplage maximum

3.2.1 Algorithme générique

On se sert du théorème 3.1.6 (page 22) et de sa démonstration.

Entrée : $G = (S, A)$ un GNO.

Pseudo-code :

> $M \leftarrow \varnothing$
> Tant qu'il existe un chemin augmentant $s_0 \dots s_n$:
> $\quad E \leftarrow \{\{s_i, s_{i+1}\} \mid i \in [\![0 \; ; \; n-1]\!]\}$
> $\quad M \leftarrow M \Delta E$
> Renvoyer M

Problème : trouver un chemin augmentant n'est pas forcément simple.

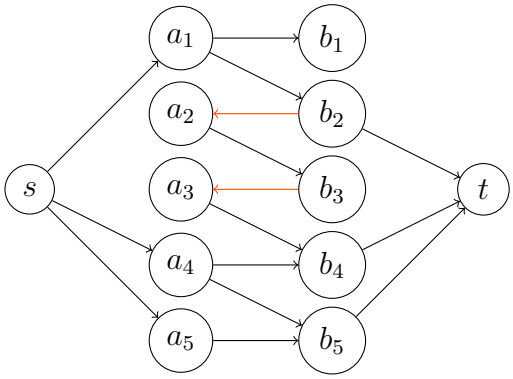
Donc on va se restreindre aux graphes bipartis.

3.2.2 Recherche d'un chemin augmentant

Soit $G = (S, A)$ un GNO biparti, et (U, V) une partition convenable de S.

Remarque : comme tout chemin augmentant est de longueur impaire (*cf* 3.1.5, page 22), il y a une extrémité dans U et l'autre dans V.

Principe : on oriente G de telle sorte que tout chemin dans le GO soit alternant puis on ajoute un sommet "source" s et un sommet "cible" t de manière que tout chemin de s à t (privé de s et t) soit augmentant. Il suffit alors de chercher un chemin de s à t à l'aide d'un parcours.



$M = \{\{a_2, b_2\}, \{a_3, b_3\}\}$

Formalisation : on définit le GO $G_M = (S, A_M)$, où

$$A_M = \{(u,v) \in U \times V \mid \{u,v\} \in A \setminus M\} \cup \{(v,u) \in V \times U \mid \{u,v\} \in M\}$$

Puis $G' = (S \cup \{s,t\}, A')$, où

$$A' = A_M \cup \{(s,u) \mid u \in U \text{ libre}\} \cup \{(v,t) \mid v \in V \text{ libre}\}$$

3.2.3 Proposition

Soit $G = (S, A)$ un GNO biparti et M un couplage.
Soit $G' = (S \cup \{s,t\}, A')$ définit comme en 3.2.2, et $s_0 \dots s_n$ un chemin de s à t.
Alors $s_1 \dots s_{n-1}$ est un chemin augmentant pour M dans G.

□ Démonstration :

Remarque : cela a du sens d'écrire $s_1 \dots s_{n-1}$ car $n \geqslant 3$ par définition de G'.
s_1 est libre car $(s, s_1) \in A'$
s_{n-1} est libre car $(s_{n-1}, t) \in A'$.
Il reste à montrer que $s_1 \dots s_{n-1}$ est alternant.
On montre par récurrence finie que $\forall i \in [\![1\ ;\ n-2]\!]$,

$$\begin{cases} i \in 2\mathbb{N} \ \Rightarrow\ s_i \in V,\ s_{i+1} \in U,\ \{s_i, s_{i+1}\} \in M \\ i \in 2\mathbb{N}+1 \ \Rightarrow\ s_i \in U,\ s_{i+1} \in V,\ \{s_i, s_{i+1}\} \in M \end{cases}$$

• Initialisation :
$s_1 \in U$ car $(s, s_1) \in A'$, donc

$$\begin{cases} s_2 \in V & \text{car } (s, s_1) \in A_M \\ \{s_1, s_2\} \notin M \end{cases}$$

• Hérédité : si $i \in [\![1\ ;\ n-3]\!]$ vérifie la propriété, on suppose $i+1 \in 2\mathbb{N}$ (l'autre cas similaire).

Par hypothèse de récurrence, $s_i \in U,\ s_{i+1} \in V,\ \{s_i, s_{i+1}\} \notin M$
Comme $s_{i+1} \in V$ et $(s_{i+1}, s_{i+2}) \in A_M$, on a

$$s_{i+2} \in U \{s_{i+1}, s_{i+2}\} \in M$$

Donc $s_1 \dots s_{n-1}$ est augmentant ■

3.2.4 Complexité de l'algorithme

On suppose $G = (S, A)$ biparti.
Comme chaque sommet ne peut être extrémité que d'une arête d'un couplage et que chaque arête a deux extrémités, le cardinal maximum d'un couplage est $\leqslant \left\lfloor \frac{|S|}{2} \right\rfloor = \mathcal{O}(|S|)$.
Vu les arguments développés en 3.1.6 (page 22), et comme on part du couplage vide, l'algorithme effectue $\mathcal{O}(|S|)$ tours de boucle.
La recherche d'une partition adaptée de S peut être effectuée une seule fois en temps $\mathcal{O}(|S| + |S|)$ *via* un parcours en largeur.
Le calcul $M \Delta E$ se fait en temps $\mathcal{O}(|E|) = \mathcal{O}(|A|)$ si on représente M avec une table qui à chaque $a \in A$ associe le booléen $a \in M$
Le calcul de G' et la recherche d'un chemin de s à t dans G' se fait en temps $\mathcal{O}(|S|+|A|)$ (parcours de M pour choisir l'orientation et déterminer les sommets libres, parcours de U et V pour définir les arcs depuis s et vers t et parcours en profondeur depuis s pour chercher un chemin de s à t).
Au total, on a une complexité $\mathcal{O}(|S|\,(|S| + |A|))$ $(= \mathcal{O}\big(|S|^3\big)$ si le graphe est dense $(|A| = \Theta\big(|S|^2\big)))$.