Algorithmique des graphes — mémento

Définition 2.1 *Un* graphe orienté, *ou simplement un* graphe, *est un couple* $G \stackrel{def}{=} (S, A)$, *où :*

- S *est un ensemble fini, dit de* sommets ;
- $A \subseteq S \times S$; *c'est l'ensemble des* arcs *de* G.

Définition 2.2 *Un* graphe non orienté *un un couple* $G \stackrel{def}{=} (S, A)$, *où :*

- S *est un ensemble fini, dit de* sommets ;
- A *est une famille de paires* $\{u, v\}$ *(c'est-à-dire d'ensemble de cardinal exactement* 2*), appelées* arêtes.

Définition 2.3 *Un* sous-graphe *d'un graphe orienté* $G \stackrel{def}{=} (S, A)$ *est un graphe de la forme* (S', A') *avec* $S' \subseteq S$ *et* $A' \subseteq A$.

Un sous-graphe induit *d'un graphe orienté* $G \stackrel{def}{=} (S, A)$ *est un graphe de la forme* $(S', A \cap S'^2)$ *avec* $S' \subseteq S$. *On dit qu'il est induit* par *le sous-ensemble de sommets* S'.

Un sous-graphe couvrant *d'un graphe orienté* $G \stackrel{def}{=} (S, A)$ *est un graphe de la forme* (S, A') *avec* $A' \subseteq A$.

3 Chemins, connexité, arbres non orientés

Définition 3.1 *Un* chemin *dans un graphe orienté* G *est une suite de sommets* $x_0 \to x_1 \to \cdots \to x_k$, *où* $k \in \mathbb{N}$. *Son* origine *est* x_0, *son* extrémité *est* x_k, *et sa* longueur *est* k. *On dit aussi qu'il s'agit d'un chemin* de x_0 à x_k.

Définition 3.2 *Un graphe non orienté est* connexe *si et seulement si il existe un chemin entre n'importe quel couple de sommets.*

Définition 3.3 *Les* composantes connexes *d'un graphe non orienté* G *sont les classes d'équivalence de* $—^*$.

Lemme 3.1 (BBC, section 4.3.1, p. 90) *Tout graphe non orienté connexe à* $n \geq 1$ *sommets a au moins* $n - 1$ *arêtes* $(m \geq n - 1)$.

Définition 3.4 *Une* chaîne *dans un graphe non orienté est un chemin* $x_0 — x_1 — \cdots — x_k$ $(k \in \mathbb{N})$ *dont les arêtes sont distinctes deux à deux. Un* cycle *est une chaîne non vide* $(k \geq 1)$ *telle que* $x_0 = x_k$.

Lemme 3.5 (D. Kőnig, graphes non orientés) *Si dans un graphe non orienté, il existe une chemin entre un sommet* u *et un sommet* v, *alors il existe un chemin de longueur minimale entre* u *et* v, *et tout chemin de longueur minimale entre* u *et* v *est une chaîne élémentaire.*

Lemme 3.2 (BBC, section 4.3.1, p. 91) *Tout graphe non orienté sans cycle à* $n \geq 1$ *sommets a au plus* $n - 1$ *arêtes* $(m \leq n - 1)$.

Définition 3.6 *Un* arbre non orienté *est un graphe connexe et sans cycle.*

Proposition 3.3 (BBC, section 4.3.1, pages 91, 92) *Pour un graphe non orienté* G *à* $n \geq 1$ *sommets, les six propriétés suivantes sont équivalentes.*

1. G *est un arbre non orienté ;*
2. G *est connexe et a* $n - 1$ *arêtes ;*
3. G *est sans cycle et a* $n - 1$ *arêtes ;*
4. *il existe une unique chaîne entre deux sommets quelconques ;*
5. G *est minimal connexe, autrement dit il est connexe, mais enlever une arête quelconque de* G *détruit la connexité ;*
6. G *est maximal sans cycle, autrement dit il est sans cycle, mais ajouter une arête entre deux sommets distincts crée un cycle.*

4 Accessibilité

Définition 4.1 *Soit* G *un graphe orienté, et* s *et* t *deux sommets de* G. *On dit que* t *est* accessible *depuis* s *dans* G *si et seulement s'il existe un chemin* $s = x_0 \to x_1 \to \cdots \to x_k = t$, $k \in \mathbb{N}$.

Définition 4.2 *Un chemin* $x_0 \to x_1 \to \cdots \to x_k$ *est* élémentaire *si et seulement s'il ne contient aucun sommet en double : si* $x_i = x_j$ *alors* $i = j$.

Lemme 4.3 (D. Kőnig, graphes orientés) *Si dans un graphe orienté, il existe un chemin d'un sommet* s *à un sommet* t, *alors il existe un chemin de longueur minimale de* s *à* t, *et tout chemin de longueur minimale de* s *à* t *est un chemin élémentaire.*

Définition 4.4 (Reach) *Pour tout sommet* s *dans un graphe orienté* $G \stackrel{def}{=} (S, A)$ *donné, on note* $\text{Reach}_G(s)$, *ou bien simplement* $\text{Reach}(s)$ *si aucune ambiguïté quant au graphe* G *n'est à craindre, l'ensemble des sommets* u *accessibles depuis* s *dans* G.

On note $\texttt{succ}_G(u) \stackrel{def}{=} \{v \in S \mid u \to v\}$, *ou simplement* $\texttt{succ}(u)$, *l'ensemble des successeurs d'un sommet* u *de* G *dans* G.

Lemme 4.5 *Pour tout sommet* s *d'un graphe* G, $\text{Reach}(s)$ *est le plus petit ensemble* E *de sommets tels que :*

1. $s \in E$,
2. *et pour tout* $u \in E$, $\texttt{succ}(u) \subseteq E$.

Théorème 4.6 *Pour tout sommet* `s` *d'un graphe orienté* G, `reach_set`$(G, \texttt{s})$ *termine en temps* $O(n + m)$ *et calcule* $\text{Reach}_G(\texttt{s})$. *L'accessibilité est décidée par* `reach` *en temps* $O(n + m)$.

```
   fun reach_set (G as (S, succ), s) =
1      work := succ (s);
2      marked := {s};
3      while work≠[] do
4            let u::rest=work in
5                if u ∈ marked
6                   then work:=rest;
7                else (work:=succ(u)@rest;
8                      marked:=marked ∪ {u});
9      return marked;
   fun reach(G, s,t) = t ∈ reach_set(G, s);
```

TABLE 1 – Un algorithme de test d'accessibilité

```
   fun reach_set (G as (S, succ), s) =
1      work := succ (s);
1,5    for each u ∈ S do markp(u):=false;
2      markp (s):=true;
3      while work≠[] do
4            let u::rest=work in
5                if markp[u]
6                   then work:=rest;
7                else (work:=succ(u)@rest;
8                      markp[u]:=true;
9      return;
   fun reach(G, s,t) = reach_set (G, s); markp[t];
```

TABLE 2 – Un algorithme de test d'accessibilité, avec tableau de marques

```
   fun reach_set (G as (S, succ), s) =
1      work := empty; for each u ∈ S do push(u,work);
2      marked := {s};
3      while nonempty(work)) do
4            let u=pop(work) in
5                if u ∉ marked
7                then (for each v ∈ succ(u) do
                              push (v, work);
8                      marked:=marked ∪ {u});
9      return marked;
   fun reach(G, s,t) = t ∈ reach_set(G, s);
```

TABLE 3 – Un algorithme de test d'accessibilité, avec une structure de travail abstraite `work`

5 Parcours en profondeur

Définition 5.1 *Un* circuit *dans un graphe* G *est une chemin* $x_0 \to x_1 \to \cdots \to x_k$ *non vide* $(k \geq 1)$ *et tel que* $x_0 = x_k$.

Un graphe orienté acyclique, *ou* DAG *(pour « directed acyclic graph ») est un graphe sans circuit.*

Définition 5.2 *Une* source *d'un graphe orienté* G *est un sommet à partir duquel tout sommet est accessible.*

Définition 5.3 *Un* arbre *(orienté) est un triplet* $(S, *, p)$ *où* $* \in S$, p *est une application de* $S \setminus \{*\}$ *dans* S, *et pour tout* $u \in S$, *il existe un* $n \in \mathbb{N}$ *tel que* $p^n(u) = * : *$ *est la* racine *de l'arbre,* $p(u)$ *est le* prédécesseur *de* u, *et seule la racine n'a pas de prédécesseur.*

Lemme 5.4 *Dans un arbre* $T \stackrel{def}{=} (S, *, p)$ *:*

1. $\to_T^*$ *est une relation d'ordre.*
2. *Si* $u \to_T^* v$ *alors il existe un unique chemin de* u *à* v *dans l'arbre.*
3. *Deux sommets quelconques* u *et* v *de* S *ont une borne inférieure* $u \wedge v$ *pour* $\to_T^*$, *appelée leur* ancêtre commun le plus proche.
4. *Si* $u \to_T^* w$ *et* $v \to_T^* w$, *alors* u *et* v *sont* comparables, *c'est-à-dire* $u \to_T^* v$ *ou* $v \to_T^* u$.
5. *Pour tous sommets* u, v, *s'il existe un chemin de* u *à* v *et* $u \neq v$, *alors il existe un chemin de* u *à* $p(v)$ *(et un arc de* $p(v)$ *à* v*).*

Définition 5.5 *Un* descendant *d'un sommet* u *dans un arbre* T *est un sommet* v *tel que* $u \to_T^* v$; *on dira alors aussi que* u *est un* ascendant *de* v *dans* T.

On notera $D_T(u)$ *l'ensemble des descendants de* u *dans l'arbre* T.

Définition 5.6 *Un* arbre couvrant *de* G *est un arbre qui forme un sous-graphe de* G, *et qui contient tous les sommets de* G.

Définition 5.7 *Un* parcours *de* G *est une liste de sommets* $L \stackrel{def}{=} [u_1, \cdots, u_n]$ *de* G *telle que :*

- *chaque sommet de* G *apparaît exactement une fois dans* L ;
- $u_1 = *$;
- *chaque sommet de* L, *sauf le premier, est successeur (dans* G*) d'un élément qui le précède dans* L *: autrement dit, pour tout* j $(1 < j \leq n)$, *il existe un* i $(1 \leq i < j)$ *tel que* $u_i \to u_j$.

Définition 5.8 *Un* parcours partiel *de* G *est une liste* L_* *de sommets* $[u_1, \cdots, u_k]$ *où* $1 \leq k \leq n$, *où* $u_1 = *$, *où chaque sommet de* G *apparaît* au plus *une fois dans* L_*, *et où chaque sommet sauf le premier est successeur d'un élément qui le précède dans* L_*.

Remarque 5.9 *Un parcours partiel à* n *éléments (où* n *est le nombre de sommets du graphe) est un parcours.*

Proposition 4.5 (BBC, section 4.4.4, p. 107, modifiée) *Tout choix d'arcs de liaison d'un parcours de* G *constitue un arbre couvrant de* G.

Remarque 5.10 *C'est un arbre, parce que G a une source $*$.*

Définition 5.11 *Un* sommet fermé *d'un parcours partiel L_* est un sommet de L_* dont tous les successeurs (dans G) sont aussi dans L_*.*

Un sommet ouvert *de L_* est un sommet de L_* qui a au moins un successeur qui n'est pas dans L_*.*

Remarque 5.12 *Dans un parcours (non partiel) L, tout sommet est fermé. Étant donné un parcours partiel L_*, tout sommet de G est soit ouvert dans L_*, soit fermé dans L_*, soit hors de L_*.*

Lemme 5.13 *Soit $L_* \stackrel{def}{=} [u_1, \cdots, u_k]$ un parcours partiel de G, $1 \leq k \leq n$. Si $k < n$, alors L_* contient un sommet ouvert.*

Corollaire 5.14 *Il existe au moins un parcours L de G.*

Remarque 5.15 *Dans le cas d'un graphe de la forme G_*, où il y a un arc $* \to u$ pour tout sommet u autre que $*$, la notion de parcours en largeur est triviale : tout parcours est en largeur.*

Lemme 5.16 *Soit $L \stackrel{def}{=} [u_1, \cdots, u_n]$ un parcours de G. On suppose un choix d'arcs de liaison effectué, ce qui définit un arbre couvrant T de G.*

1. *Tout sommet u de G s'écrit u_i pour un unique i ($1 \leq i \leq n$).*
   On appelle i le rang *$r(u)$ de u dans L.*
2. *Tout descendant v d'un sommet u dans T satisfait $r(v) \geq r(u)$.*
3. *Soit $1 \leq i \leq j \leq k \leq n$. Si u_i est un sommet de L_j qui est ouvert dans L_k, alors il est ouvert dans L_j.*

Définition 5.17 *Soit $L \stackrel{def}{=} [u_1, \cdots, u_n]$ un parcours de G. L est un* parcours en profondeur *si et seulement, pour chaque j ($1 < j \leq n$), le* plus grand *indice i ($1 \leq i < j$) tel que u_i soit un sommet ouvert de L_{j-1} est tel que $u_i \to u_j$.*

Nous fixons désormais un parcours en profondeur $L \stackrel{\text{def}}{=} [u_1, \cdots, u_n]$ de G (avec $u_1 = *$), et nous notons $T \stackrel{\text{def}}{=} (S, *, p)$ l'arbre couvrant associé. Un *descendant* d'un sommet u est un sommet v accessible depuis u dans T, c'est-à-dire tel que $u \to_T^* v$.

Proposition 5.18 *Pour tout sommet u de G, l'ensemble des descendants $D_T(u)$ de u dans T est un* intervalle *de L, c'est-à-dire un ensemble de la forme $\{u_i, u_{i+1}, \cdots, u_j\}$, où $i = r(u)$. On appelle j le* temps de fin *$fin(u)$ de u dans L.*

De plus, $j = fin(u)$ est le plus petit entier supérieur ou égal à i tel que tous les sommets u_i, u_{i+1}, …, u_j soient fermés dans L_j.

Proposition 5.19 *Pour tout couple de sommets u et v dans G, les trois propriétés suivantes sont équivalentes :*

1. *v est un descendant de u dans T ;*
2. *$[r(v), fin(v)] \subseteq [r(u), fin(u)]$;*
3. *$r(u) \leq r(v)$ et $fin(v) \leq fin(u)$.*

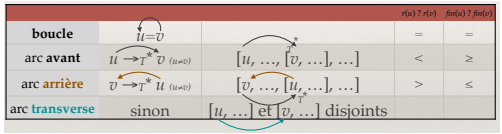

TABLE 4 – La classification des arcs par rapport à un parcours en profondeur

(Table omise : duplique la table 4.)

TABLE 5 – Classification des arcs dans un parcours en profondeur

Proposition 5.20 *Pour tout couple de sommets u et v dans G, tels qu'aucun des deux n'est descendant de l'autre dans T, les intervalles $[r(u), fin(u)]$ et $[r(v), fin(v)]$ sont disjoints.*

Remarque 5.21 *Parmi les arcs avants, on trouve bien sûr les arcs $u \to_T v$ de l'arbre T. Mais il y en d'autres.*

Lemme 4.6 (BBC, section 4.4.4, p. 108) *Pour tout arc transverse $u \to v$, $r(u) > r(v)$.*

Note 5.22 *(Omise.)*

Un *circuit* est un chemin $u \to \cdots \to u$ contenant au moins un arc.

Lemme 4.7 (BBC, section 4.4.4, pages 108–109) *Le graphe G est sans circuit si et seulement si il n'a ni boucle ni arc arrière (par rapport au parcours en profondeur L).*

Lemme 5.23 *Pour tout sommet u de G, en posant $j \stackrel{def}{=} fin(u)$:*

1. *u_j est une feuille de l'arbre T ;*
2. *pour tout sommet v de G tel que $u \to_T v \to_T^* u_j$, $fin(v) = fin(u)$.*

Proposition 5.24 *Pour tout sommet u de G, $fin(u)$ vaut :*

— *$r(u)$ si u est une feuille de T ;*
— *$\max\{fin(v) \mid u \to_T v\}$ sinon.*

Définition 5.25 *Une* extension $\sqsubseteq$ *d'une relation d'ordre $\leq$ sur un ensemble E est une relation d'ordre telle que pour tous $x, y \in E$, si $x \leq y$ alors $x \sqsubseteq y$ — autrement dit, $\leq \subseteq \sqsubseteq$.*

Une extension totale *est une extension qui est une relation d'ordre totale.*

```
fun dfs_1 (u) =
    marked := marked ∪ {u};
    r(u) := ++dfsNum;
    for each v ∈ succ(u) do
        if v ∉ marked
            then dfs_1 (v);
    fin(u) := dfsNum;
```

TABLE 6 – Un algorithme de parcours en profondeur basique

```
fun dfs_2 (u) =
    marked := marked ∪ {u};
    r(u) := ++dfsNum;
    for each v ∈ succ(u) do
        if v ∉ marked
            then dfs_2 (v);
    fin(u) := dfsNum;
    rto(u) := ++revToporder;
```

TABLE 7 – Tri topologique

Définition 5.26 *Un* tri topologique *d'un graphe $G \stackrel{def}{=} (S, A)$ est une fonction de numérotation to: $S \to \mathbb{N}$ telle que pour tous $u, v \in S$, si $u \to v$ alors $to(u) < to(v)$.*

Un tri topologique inverse *est une fonction de numérotation rto: $S \to \mathbb{N}$ telle que pour tous $u, v \in S$, si $u \to v$ alors $rto(u) > rto(v)$.*

Remarque 5.27 (Rang≠tri topologique) *Le parcours $L \stackrel{def}{=} [u_1, \cdots, u_n]$ en profondeur d'un graphe G avec une source $*$ donne une fonction de numérotation r (la fonction rang), et l'on pourrait penser que c'est un tri topologique $u_1 \sqsubset \cdots \sqsubset u_n$. Ceci est faux. Comme la table 5 le montre, ceci échoue dès qu'il existe un arc transverse.*

Lemme 5.28 *Soit G un graphe. Alors :*

1. *G est sans circuit si et seulement si G_* est sans circuit ;*
2. *si G a un tri topologique inverse rto, alors G_* a un tri topologique inverse rto_*, qui coïncide avec rto sur les sommets de G ;*
3. *si G_* a un tri topologique inverse, alors sa restriction aux sommets de G est un tri topologique inverse de G.*

Lemme 5.29 *Soit G un graphe avec une source $*$, L un parcours en profondeur de G, et T l'arbre couvrant associé. Soit rto: $S \to \mathbb{N}$ n'importe quelle fonction telle que, pour tout sommet u,*

1. *pour tout $v \in S$ tel que $u \to_T v$, $rto(u) > rto(v)$;*
2. *pour tout $v \in S$ tel que $fin(v) < r(u)$, $rto(u) > rto(v)$.*

Si G est sans circuit, alors pour tous $u, v \in S$, $u \to v$ implique $rto(u) > rto(v)$.

Toujours en supposant G sans circuit, on peut définir une telle fonction rto par :

$$rto(u) \stackrel{\text{def}}{=} \max(\max\{rto(v) \mid v \in S \text{ tel que } u \to_T v\}, \quad (1)$$
$$\max\{rto(v)) \mid v \in S \text{ tel que } fin(v) < r(u)\}) + 1,$$

où, le cas échéant, le max d'une famille vide est considéré égal à 0.

Théorème 5.30 *Un graphe G a un tri topologique (inverse) si et seulement si G est un DAG.*

Pour la même raison que pour `dfs_1`, `dfs_2 (*)` est un algorithme en temps $O(m + n)$.

Cet algorithme ne fonctionne que si G a une source $*$. Sinon, on peut soit la créer, soit simuler le déroulement de l'algorithme sur G_*. Il suffit d'écrire :

```
marked := ∅;
dfsNum := 1;
revTopOrder := 0;
for each v ∈ S do
    if v ∉ marked
        then dfs_2 (v);
```

6 Composantes fortement connexes

Définition 6.1 *Les* composantes fortement connexes *d'un graphe orienté sont ses classes d'équivalence pour la relation d'équivalence $\equiv$ associée à sa relation d'accessibilité $\to^*$. On notera $[u]$ la classe d'équivalence de u pour cette relation d'équivalence, et on l'appellera la* composante fortement connexe de u.

Remarque 6.2 *Il ne faut pas dire « composante connexe » pour un graphe orienté.*

Définition 6.3 *Pour tout graphe orienté $G \stackrel{def}{=} (S, A)$, le graphe $(S/\equiv, \Rightarrow)$ est la* condensation *de G.*

Remarque 6.4 *Dans un circuit $u_0 \to u_1 \to \cdots \to u_n = u_0$ ($n \geq 1$), tous les sommets appartiennent à la même composante fortement connexe.*

Définition 6.5 *Soit $G \stackrel{def}{=} (S, A)$ un graphe avec une source $*$, soit L un parcours en profondeur de G, et soit $T \stackrel{def}{=} (S, *, p)$ l'arbre couvrant associé. Pour toute composante fortement connexe C de G, le* point d'entrée *$*_C$ de C est le sommet de plus petit rang de C.*

Lemme 6.6 *Pour toute composante fortement connexe C de G, $C \subseteq D_T(*_C)$.*

Lemme 4.8 (BBC, section 4.4.4, p. 109, reformulé) *Pour toute composante fortement connexe C de G, $T_{|C} \stackrel{def}{=} (C, *_C, p_{|C \setminus \{*_C\}})$ est un arbre couvrant C, de racine $*_C$, et un sous-graphe induit de T.*

Définition 6.7 *Soit G un graphe avec une source $*$. On note $v \hookrightarrow w$ pour un arc arrière ou transverse de v à w. $AT(u)$ est l'ensemble des sommets w tels qu'il existe un circuit de la forme $u \to_T^* v \hookrightarrow w \to^* u$.*

Le point d'attache *$a(u)$ d'un sommet u est le sommet $w \in AT(u) \cup \{u\}$ de plus petit rang.*

Les sommets de $AT(u)$ sont les sommets w tels qu'il existe un circuit de la forme $u \to_T^* v \hookrightarrow w \to^* u$, pour un certain sommet v.

On note $low(u)$ le rang $r(a(u))$ du point d'attache de u.

Remarque 6.8 *La figure 4.8 du BBC peut donner l'impression que $a(u)$ est égal à $*_C$ pour tout sommet u de toute composante fortement connexe C. Ce n'est pas le cas.*

Remarque 6.9

1. *$low(u) \leq r(u)$;*
2. *si $u \to_T^* v \hookrightarrow w \to^* u$, alors $low(u) \leq r(w)$;*
3. *$low(u) < r(u)$ si et seulement s'il existe un circuit de la forme $u \to_T^* v \hookrightarrow w \to^* u$ où $r(w) < r(u)$.*

Lemme 6.10 *Pour toute composante fortement connexe C, pour tout sommet u de C, $a(u)$ est aussi dans C.*

Lemme 4.9 (BBC, section 4.4.4, p. 110) *Un sommet u est le point d'entrée $*_C$ de sa composante fortement connexe $C \stackrel{def}{=} [u]$ si et seulement si $u = a(u)$.*

Proposition 6.11 *Pour tout sommet u de G, le rang de $a(u)$ est l'entier le plus petit parmi :*

1. *$r(u)$;*
2. *$r(a(v))$ (c'est-à-dire $low(v)$), lorsque $u \to v$ parcourt les arcs de liaison d'origine u (autrement dit, $u \to_T v$) ;*
3. *$r(v)$, lorsque $u \to v$ parcourt les arcs arrières ou transverses d'origine u dont l'extrémité v est dans la même composante fortement connexe que u.*

Proposition 6.12 *Pour tout sommet u de G, le rang de $a(u)$ est l'entier le plus petit parmi :*

1. *$r(u)$;*
2. *$r(a(v))$, lorsque $u \to v$ parcourt les arcs de liaison d'origine u (autrement dit, $u \to_T v$) ;*
3. *$r(v)$, lorsque $u \to v$ parcourt :*
   (a) *les arcs arrières ou transverses d'origine u dont l'extrémité v est dans la même composante fortement connexe que u ;*
   (b) *et les arcs avants, ainsi que les boucles, d'origine u.*

Rappelons que nous supposons un parcours en profondeur $L \stackrel{\text{def}}{=} [u_1, \cdots, u_n]$. Pour tout sommet u du graphe G, l'unique chemin de $*$ à u dans T, $* = u_{i_1} \to_T u_{i_2} \to_T \cdots \to_T u_{i_\ell} = u$, avec $1 = i_1 < i_2 < \cdots < i_\ell = k$est la *branche* menant au sommet u.Pour tout k $(0 \leq k \leq n)$, notons π_k la sous-liste de L_k formée des éléments u_i qui sont dans la même composante fortement connexe qu'un des éléments de la branche menant à u_k.On appellera π_k la *pile à profondeur* k.

Proposition 6.13 *Soit u un sommet de G, et $k \in [r(u), fin(u)]$. Pour tout arc $u \to v$ de source u :*

1. *si $u \equiv v$ et $v \in L_k$, alors v est dans π_k ;*
2. *si v est dans π_k, alors v est dans L_k, et $u \equiv v$ ou bien $u \to_T^* v$.*

L'*algorithme de Tarjan* du calcul des composantes fortement connexes de G (toujours supposé avec origine $*$) consiste à appeler `scc (*)`, où `scc` est défini en table 8.

La pile `stk` peut être implémentée par :

— un tableau `s` de sommets, de taille n ;

— un compteur `k` du nombre d'éléments dans `s` ;

— un tableau `onStk` de bits, de taille n ;

Ce dernier tableau sera tel que `onStk`$[u]$ sera vrai, pour tout sommet u, si et seulement si u est sur la pile.

Théorème 6.14 *L'algorithme de Tarkan `scc` calcule correctement les composantes fortement connexes d'un graphe G avec une source $*$: à la fin de l'algorithme, pour tout sommet u de G, `c`(u) vaut le point d'entrée $*_C$ de la composante fortement connexe C de u, et donc deux sommets u et v sont dans la même composante fortement connexe si et seulement si `c`$(u) =$ `c`(v). Sa complexité est en $O(m + n)$.*

```
fun scc (u) =
1     marked := marked ∪ {u};
2     r(u) := ++dfsNum;
3     low(u) := dfsNum;
4     push(stk,u);
5     for each v ∈ succ(u) do
6         if v ∉ marked
7            then (scc (v);
8                  low(u) := min(low(u),low(v)));
9         else if v ∈ stk then low(u):=min(low(u),r(v));
10    if r(u)=low(u)
11       then do v := pop(stk); c(v):=u until v=u;
```

TABLE 8 – L'algorithme de Tarjan

```
fun roy-warshall (M, n) =
1     A := copy(M);
2     for i=1..n
3         A [i, i] := 1;
4     for k=1..n
5         for i=1..n
6             for j=1..n
7                 A [i, j] := A [i, j] or
8                             (A [i, k] and A [k, j]);
9     return A;
```

TABLE 9 – L'algorithme de Roy-Warshall

7 Graphes valués, automates, distances minimales et maximales

7.1 L'algorithme de Roy-Warshall

Définition 7.1 *Pour tout graphe $G \stackrel{def}{=} (S, A)$ avec $S = \{1, \cdots, n\}$, pour tout $k \in \{1, \cdots, n\}$, on note G_k le graphe (S, A_k), où A_k est l'ensemble des couples $(i, j) \in S^2$ tels qu'il existe un chemin*

$$i = x_0 \to x_1 \to \cdots \to x_\ell = j$$

*avec $\ell \in \mathbb{N}$, dont l'*intérieur $\{x_i \mid 0 < i < \ell\}$ *est inclus dans $\{1, \cdots, k\}$.*

Proposition 7.2 *Avec les notations de la définition 7.1, pour tous $(i, j) \in S^2$, on a :*

1. *$(i, j) \in A_0$ si et seulement si $i = j$ ou $i \to j$ dans G ;*
2. *pour tout $k \in \{1, \cdots, n\}$, $(i, j) \in A_k$ si et seulement si $(i, j) \in A_{k-1}$, ou bien (i, k) et (k, j) sont dans A_{k-1}.*

Lemme 7.3 *En reprenant les notations de la définition 7.1, pour tous $k, i, j \in \{1, \cdots, n\}$, on a :*

1. *$(i, k) \in A_{k-1}$ si et seulement si $(i, k) \in A_k$;*
2. *$(k, j) \in A_{k-1}$ si et seulement si $(k, j) \in A_k$.*

Théorème 7.4 *L'algorithme de Roy-Warshall `roy-warshall` calcule la clôture réflexive-transitive d'un graphe $G \stackrel{def}{=} (\{1, \cdots, n\}, A)$, lorsqu'on lui fournit en entrée sa matrice d'adjacence et n. Sa complexité est en $O(n^3)$.*

7.2 Graphes valués

Définition 7.5 *Soit K un monoïde. Un* graphe valué, *à valeurs dans K, est un couple (G, V) où $G \stackrel{def}{=} (S, A)$ est un graphe et $V : A \to K$ est une fonction de l'ensemble A des arcs vers un ensemble K. Les éléments de K sont appelés les* valeurs.

Définition 7.6 *Le* poids *d'un chemin $x_0 \xrightarrow{a_1} x_1 \xrightarrow{a_2} \cdots \xrightarrow{a_k}$ x_k est le produit $a_1 \times a_2 \times \cdots \times a_k$.*

Définition 7.7 *Un* semi-anneau *est un uplet $(K, +, 0, \times, 1)$ où :*

— *$(K, +, 0)$ est un monoïde commutatif ;*

— *$(K, \times, 1)$ est un monoïde ;*

— *les lois de distributivité suivantes sont valides :*

$$a \times (b + c) = (a \times b) + (a \times c) a \times 0 = 0$$
$$(b + c) \times a = (b \times a) + (c \times a) 0 \times a = 0,$$

pour tous $a, b, c \in K$.

Définition 7.8 *La* matrice d'adjacence *d'un graphe valué (S, A, V), avec $S \stackrel{def}{=} \{1, \cdots, n\}$, où V est une fonction de A vers un semi-anneau $(K, +, 0, \times, 1)$, est la matrice $n \times n$ dont chaque entrée (i, j) vaut :*

— *$V(i, j)$ si $(i, j) \in A$;*

— *0 sinon.*

7.3 Algèbres de Kleene continues

Définition 7.9 *Un monoïde $(K, +, 0)$ est* idempotent *si et seulement si $a + a = a$ pour tout $a \in K$. Dans ce cas, on pose $a \leq b$ si et seulement si $b = a + b$, pour tous $a, b \in K$.*

Un semi-anneau $(K, +, 0, \times, 1)$ est idempotent *si et seulement si $(K, +, 0)$ est un monoïde idempotent.*

Lemme 7.10 *Pour tout monoïde commutatif idempotent $(K, +, 0)$, $(K, \leq)$ est un* semi-treillis supérieur *: $\leq$ est une relation d'ordre, $a + b$ est la borne supérieure de $\{a, b\}$ pour tous $a, b \in K$, et 0 est le plus petit élément de K.*

Définition 7.11 *Une* algèbre de Kleene continue *est un semi-anneau idempotent $(K, +, 0, \times, 1)$ tel que :*

1. *toute famille de la forme $(b^n)_{n\in\mathbb{N}}$ a une borne supérieure, que l'on notera b^* ;*
2. *pour toute famille dénombrable $(u_n)_{n\in\mathbb{N}}$ ayant une borne supérieure $\sup_{n\in\mathbb{N}} u_n$, pour tous $a, b \in K$, la famille $(au_nb)_{n\in\mathbb{N}}$ a aussi une borne supérieure, et $\sup_{n\in\mathbb{N}} au_nb = a(\sup_{n\in\mathbb{N}} u_n)b$.*

Remarque 7.12 *La condition de continuité exprime que le produit est continu, au sens où il commute aux bornes supérieures existantes. L'addition est toujours continue.*

Remarque 7.13 *Dans la littérature, une algèbre de Kleene est un sextuplet $(K, +, 0, \times, 1, \_^*)$ où $(K, +, 0, \times, 1)$ est un semi-anneau, et $\_^*$ est une opération satisfaisant aux* axiomes de Kozen *:*

— *$1 + aa^* \leq a^*$;*

— *$1 + a^*a \leq a^*$;*

```
fun roy-warshall-gen (M, n) =
1     A := copy(M);
2     for i=1..n
3         A [i, i] := A [i, i]+1;
4     for k=1..n
5         star := A[k,k]*;
6         for i=1..n
7             for j=1..n
8                 A[i, j] := A[i, j] +
9                            (A[i, k]×star×A[k, j]);
10    return A;
```

TABLE 10 – L'algorithme de Roy-Warshall généralisé aux graphes valués

— *$ab \leq b$ implique $a^*b \leq b$;*

— *$ba \leq b$ implique $ba^* \leq b$,*

pour tous $a, b \in K$. Elle est appelée $*$-continue si et seulement si $a^* = \sup_{n\in\mathbb{N}} a^n$ pour tout $n \in \mathbb{N}$; il existe des algèbres de Kleene non $*$-continues.

Définition 7.14 *Dans un semi-anneau, on appelle* somme *d'un ensemble C d'éléments (fini ou non) la borne supérieure de C, si elle existe.*

7.4 L'algorithme de Roy-Warshall généralisé

Remarque 7.15 *Dans le cas de l'algèbre de Kleene triviale, $(\{0, 1\}, \max, 0, \min, 1)$, l'addition est un « ou », le produit est un « et », et `A[k,k]`* vaut toujours* 1. *On retrouve donc l'algorithme de Roy-Warshall usuel (table 9) dans ce cas.*

Définition 7.16 *Pour tout graphe valué $G \stackrel{def}{=} (S, A, V)$, avec valeurs dans une algèbre de Kleene continue K et $S = \{1, \cdots, n\}$, pour tout $k \in \{1, \cdots, n\}$, on note V_k la fonction qui à tout couple $(i, j) \in S^2$ associe la somme (si elle existe) des poids des chemins de la forme :*

$$i = x_0 \to x_1 \to \cdots \to x_\ell = j$$

*avec $\ell \in \mathbb{N}$, et dont l'*intérieur $\{x_i \mid 0 < i < \ell\}$ *est inclus dans $\{1, \cdots, k\}$.*

Proposition 7.17 *Avec les notations de la définition 7.16, pour tous $(i, j) \in S^2$, on a :*

1. *$V_0(i, j)$ est définie, et vaut $V(i, i) + 1$ si $i = j$, $V(i, j)$ sinon ;*
2. *pour tout $k \in \{1, \cdots, n\}$, $V_k(i, j)$ est bien définie et vaut $V_{k-1}(i, j) + (V_{k-1}(i, k) \times V_{k-1}(k, k)^* \times V_{k-1}(k, j))$.*

Lemme 7.18 *Dans un semi-anneau K,*

1. *si $A \subseteq B \subseteq K$ et si les sommes (les bornes supérieures) de A et de B existent, alors la somme de A est inférieure ou égale à celle de B ;*
2. *l'addition est monotone : si $a \leq b$ alors $a + c \leq b + c$;*
3. *le produit est monotone : si $a \leq b$ alors $ac \leq bc$ et $ca \leq cb$.*

```
  fun floyd-max (M, n) =
1     A := copy(M);
2     for i=1..n
3         A [i, i] := max(A [i, i], 0);
4     for k=1..n
5         if A[k,k]<=0 then star:=0 else star:=+∞;
6         for i=1..n
7             for j=1..n
8                 A[i, j] := max (A[i, j],
9                                 (A[i, k]+star+A[k, j]);
10    return A;
```

TABLE 11 – L'algorithme de Floyd de calcul des distances maximales

Lemme 7.19 *En reprenant les notations de la définition 7.16, pour tous $k, i, j \in \{1, \cdots, n\}$, on a :*

$$V_{k-1}(i,j) \leq V_k(i,j)$$
$$V_k(i,j) = V_{k-1}(i,j) + (a \times V_{k-1}(k,k)^* \times b)$$

pour tous $a, b \in K$ tels que $V_{k-1}(i,k) \leq a \leq V_k(i,k)$ et $V_{k-1}(k,j) \leq b \leq V_k(k,j)$.

Théorème 7.20 *Pour tout graphe valué $G \stackrel{def}{=} (\{1, \cdots, n\}, A, V)$, où V est à valeurs dans une algèbre de Kleene continue K, l'algorithme* ***roy-warshall-gen*** *appelé sur la matrice d'adjacence de G et n retourne un tableau* `A` *tel que, pour tous $i, j \in \{1, \cdots, n\}$ est la somme des poids de tous les chemins de i à j dans G. La complexité de l'algorithme est en $O(n^3)$, en supposant les opérations de l'algèbre de Kleene continue K en temps constant.*

L'hypothèse du temps constant pour les opérations de K est importante.

7.5 L'algorithme de McNaughton-Yamada

Théorème 7.21 *Il existe un algorithme qui convertit tout automate fini $\mathcal{A}$ en une expression régulière dont le language est identique au langage des mots acceptés par $\mathcal{A}$.*

7.6 L'algorithme de Floyd

Convention 7.22 (distances max) $(-\infty)+(+\infty) = -\infty$.

Proposition 7.23 (Poids strictement positifs) *Soit $G \stackrel{def}{=} (S, A, V)$ un graphe valué, à valeurs dans $(\mathbb{R} \cup \{-\infty, +\infty\}, \max, -\infty, +, 0)$, et supposons que V soit à valeurs dans $\mathbb{R}$.*

La borne supérieure des distances de tout sommet x à tout sommet y est atteinte, et est donc un max*, si et seulement si :*

1. *il existe un chemin $x \to^* y$,*
2. *et pour tout sommet z tel que $x \to^* z \to^* y$ dans G, il n'y a pas de circuit $z \to^+ z$ de poids strictement positif.*

Si la condition 1 n'est pas vérifiée, le supremum vaut $-\infty$, et si la condition 1 est vérifiée mais pas la condition 2, le supremum vaut $+\infty$.

Convention 7.24 (distances min) $(-\infty)+(+\infty) = +\infty$.

```
fun floyd-max-opt (M, n) =
    A := copy(M);
    for i=1..n
        A [i, i] := max(A [i, i], 0);
    for k=1..n
        if A[k,k]>0
           then for i=1..n
                  for j=1..n
                    if A [i, j]!=−∞ then A[i,j]:=+∞
        else for i=1..n
               for j=1..n
                 A [i, j] := max (A [i, j],
                                  (A [i, k] + A [k, j]);
    return A;
```

TABLE 12 – L'algorithme de Floyd de calcul des distances maximales, optimisé

```
fun floyd-max-simple (M, n) =
    A := copy(M);
    for i=1..n
        A [i, i] := max(A [i, i], 0);
    for k=1..n
        for i=1..n
            for j=1..n
                A [i, j] := max (A [i, j],
                                 (A [i, k] + A [k, j]);
    return A;
```

TABLE 13 – L'algorithme de Floyd de calcul des distances maximales, pour le cas des graphes valués sans circuit de poids strictement positif

Proposition 7.25 (Poids strictement négatifs) *Soit $G \stackrel{def}{=} (S, A, V)$ un graphe valué, à valeurs dans $(\mathbb{R} \cup \{-\infty, +\infty\}, \min, +\infty, +, 0)$, et supposons que V soit à valeurs dans $\mathbb{R}$.*

La borne inférieure des distances de tout sommet x à tout sommet y est atteinte, et est donc un min*, si et seulement si :*

1. *il existe un chemin $x \to^* y$,*
2. *et pour tout sommet z tel que $x \to^* z \to^* y$ dans G, il n'y a pas de circuit $z \to^+ z$ de poids strictement négatif.*

Si la condition 1 n'est pas vérifiée, l'infimum vaut $+\infty$, et si la condition 1 est vérifiée mais pas la condition 2, l'infimum vaut $-\infty$.

Remarque 7.26 *L'algorithme de Floyd est souvent considéré comme s'exécutant en temps $O(n^3)$, et ceci suppose que les opérations arithmétiques faites sur les valeurs (réelles) des arcs sont en temps constant.*

8 Distances minimales à sommet de départ fixé

On s'intéressera à des graphes valués $G \stackrel{\text{def}}{=} (S, A, c)$, où c: $A \to \mathbb{R}$. La valeur $c(x, y)$ est le *coût* de l'arc $x \to y$Le poids d'un chemin sera appelé son coût, dans ce contexte.

Définition 8.1 *Le* coût $c(x \to^* y)$ *d'un chemin est la somme des coûts de ses arcs. La* distance $d(x, y)$ *d'un sommet x à un sommet y est la borne inférieure de l'ensemble des coûts des chemins de x à y.*

Proposition 2.2 (BBC, section 7.2.2, p. 221) *Si γ: $x = x_0 \to x_1 \to \cdots \to x_n = y$ est un chemin de coût minimum dans un graphe valué G à valeurs dans $\mathbb{R}$, alors pour tous indices i, j tels que $0 \leq i \leq j \leq n$, $x_i \to \cdots \to x_j$ est un chemin de coût minimum de x_i à x_j dans G.*

8.1 L'algorithme de Bellman-Ford

Définition 8.2 *Soit G un graphe valué à valeurs dans $\mathbb{R}$, et s un sommet fixé de G. Pour tout $k \in \mathbb{N}$, pour tout sommet x de G, on note $d_k(s, x)$ la borne inférieure des coûts des chemins de s à x de longueur au plus k dans G.*

Remarque 8.3 *Dans $d_k(s, x)$, k est une* longueur*, et pas un* coût*.*

Lemme 8.4 *Soit $G \stackrel{def}{=} (S, A, c)$ un graphe valué à valeurs dans $\mathbb{R}$. On a :*

1. *$d_0(s, x) = 0$ si $s = x$, $+\infty$ sinon ;*
2. *pour tout $k \in \mathbb{N}$,*

$$d_{k+1}(s,x) = \min(d_k(s,x), \min\{d_k(s,y) + c(y,x) \mid y \in S \text{ tel que } (y,x) \in A\}).$$

Lemme 8.5 *Si G est une graphe valué, à valeurs dans $\mathbb{R}$, et à n sommets, pour tout sommet s de G, si G n'a pas de circuit de coût strictement négatif (et plus précisément, aucun circuit de coût strictement négatif accessible depuis s), alors pour tout sommet x de G, $d(s, x) = d_n(s, x)$.*

```
fun bellman-1 (G as (S, succ, c), s, n) =
1     for each x ∈ S do
2             d[x,0] := +∞;
3     d[s,0] := 0;
4     for k=1..n do
5         for each x ∈ S do
6                 d[x,k] := d[x,k-1];
7         for each y ∈ S do
8                 for each x ∈ succ(y) do
9                     let improve = d[y,k-1]+c(y,x) in
10                    if improve<d[x,k]
11                       then d[x,k] := improve;
12    return d;
```

TABLE 14 – L'algorithme de Bellman-Ford, première version

```
  fun bellman-2 (G as (S, succ, c), s, n) =
1     for each x ∈ S do
2             d[x] := +∞;
3     d[s] := 0;
4     for k=1..n do
5         for each y ∈ S do
6                 for each x ∈ succ(y) do
7                     let improve = d[y]+c(y,x) in
8                     if improve<d[x]
9                        then d[x] := improve;
10    return d;
```

TABLE 15 – L'algorithme de Bellman-Ford, calcul en place

Théorème 8.6 *Si G est une graphe valué, à valeurs dans $\mathbb{R}$, et à n sommets, pour tout sommet s de G, si G n'a pas de circuit de coût strictement négatif (et plus précisément aucun circuit de coût strictement négatif accessible depuis s), alors* `bellman-1`*(G, s, n) calcule un tableau* `d` *tel que, pour tout sommet x de G,* `d`*$[x, n]$ vaut la distance $d(s, x)$. La complexité de cet algorithme est en $O(n(m + n))$.*

Théorème 8.7 *Si G est une graphe valué, à valeurs dans $\mathbb{R}$, et à n sommets, et si G n'a pas de circuit de coût strictement négatif, alors pour tout sommet s de G,* `bellman-2`*(G, s, n) calcule un tableau* `d` *tel que, pour tout sommet x de G,* `d`*$[x]$ vaut la distance $d(s, x)$. La complexité de cet algorithme est en $O(n(m + n))$.*

Lemme 8.8 *Soit G un graphe valué, à valeurs dans $\mathbb{R}$, et s un sommet de G. Il existe un circuit $z \to^+ z$ de coût négatif avec z accessible depuis s si et seulement si $d_{n+1}(s, y) < d_n(y)$ pour au moins un sommet y de G.*

8.2 Arbres de chemins de coûts minimums

Proposition 2.3 (BBC, section 7.2.2, p. 222) *Pour tout graphe valué G à valeurs dans $\mathbb{R}$, sans circuit de coût strictement négatif, pour tout sommet s de G, il existe un sous-arbre $T \stackrel{def}{=} (\text{Reach}(s), s, p)$ de G, de racine s, dont les sommets sont les sommets accessibles depuis s dans G, et tel que pour tout $x \in \text{Reach}(s)$, l'unique chemin :*

$$s \to_T \cdots \to_T x$$

```
fun bellman-1-detect (G as (S, succ, c), s, n) =
1     for each x ∈ S do
2             d[x,0] := +∞;
3     d[s,0] := 0;
4     for k=1..n+1 do
5         for each x ∈ S do
6                 d[x,k] := d[x,k-1];
7         for each y ∈ S do
8             for each x ∈ succ(y) do
9                 let improve = d[y,k-1]+c(y,x) in
10                if improve<d[x,k]
11                   then d[x,k] := improve;
12    for each y ∈ S do
13            if d[y,n+1]<d[y,n]
14               then raise CircuitNegatif;
15    return d;
```

TABLE 16 – L'algorithme de Bellman-Ford, avec détection de circuits de coût négatifs

```
fun ford (G as (S, succ, c), s, n) =
1     for each x ∈ S do
2             d[x] := +∞;
3             p[x] := undef;
4     d[s] := 0;
5     work := empty;
6     push(work,s);
7     while nonempty(work) do
8           let x=pop(work) in
9               for each y ∈ succ(x) do
10                    let improve = d[x]+c(x,y) in
11                    if improve<d[y]
12                       then d[y] := improve;
13                            p[y] := x;
14                            push_opt(work,y);
15    return d, p;
```

TABLE 17 – L'algorithme de Ford

dans T est un chemin de coût minimum de s à x dans G.

*T est l'*arbre des chemins de coûts minimums *de G.*

Proposition 2.4 (BBC, section 7.2.2, p. 223) *Soit $G \stackrel{def}{=} (S, A, c)$ un graphe valué G à valeurs dans $\mathbb{R}$ et soit s un sommet de G. Soit $T \stackrel{def}{=} (S_T, s, p_T)$ un arbre avec $s \in S_T \subseteq \text{Reach}(s)$. Pour tout $x \in S_T$, on définit $\lambda_T(x)$ comme étant le coût de l'unique chemin de s à x dans T ; on définit $\lambda_T(x)$ comme valant $+\infty$ pour tout autre sommet x de G.*

Alors T est un arbre de chemins de coûts minimums de G si et seulement si, pour tout sommet $x \in \text{Reach}(s)$ et pour tout successeur y de x dans G,

$$\lambda_T(x) + c(x,y) \geq \lambda_T(y).$$

8.3 L'algorithme de Ford

Théorème 8.9 *Pour tout graphe valué G à valeurs dans $\mathbb{R}$ à n sommets, sans circuit de poids strictement négatif, pour tout sommet s de G, `ford`(G, s, n), s'il termine, retourne deux tableaux d et p tels que :*

1. *pour tout sommet $x \in \text{Reach}(s)$, $d[x]$ est la distance $d(s,x)$ de s à x ;*
2. *l'arbre $T \stackrel{def}{=} (\text{Reach}(s), s, p)$ est un arbre de chemins de coûts minimums.*

8.4 L'algorithme de Ford à files

```
fun ford-variant (G as (S, succ, c), s, n) =
1     for each x ∈ S do
2             d[x] := +∞;
3             p[x] := undef;
4     d[s] := 0;
5     work := empty;
6     push(work,s);
7     for k=1..n+1
8         work := ford-variant-phase (G, s, n, d,
                                      p, work, k);
9     return d, p;

fun ford-variant-phase (G as (S, succ, c), s, n, d
                        p, work, k) =
1     let later := empty in
2     while nonempty(work) do
3           let x=pop(work) in
4               for each y ∈ succ(x) do
5                   let improve = d[x]+c(x,y) in
6                   if improve<d[y]
7                      then d[y] := improve;
8                           p[y] := x;
9                           if y ∉ work then
                                push_opt(later,y);
10    return later;
```

Théorème 8.10 *La complexité de l'algorithme de Ford à files est en $O(n(m+n))$.*

8.5 L'algorithme de Ford à liste de travail

Il est en temps exponentiel, voir la figure 2.

8.6 L'algorithme de Dijkstra

Nous appelons algorithme de Dijkstra la variante de l'algorithme de Ford où `work` est implémenté par une file *à priorités*, la priorité de chaque sommet x étant la valeur courante $\mathtt{d}[x]$.

Théorème 8.11 *La complexité de l'algorithme de Dijkstra sur un graphe valué sans arc de coût strictement négatif, est en $O(m + n\log n)$, à condition d'implémenter `work` sous forme d'une liste de priorités via des tas de Fibonacci.*

8.7 Le cas des graphes sans circuit

Lemme 8.12 *Si $x_1, x_2, \cdots, x_k$ est un tri topologique de G, et si $s = x_k$, alors pour tout $i \in \{1, \cdots, k\}$,*

$$d(s, x_i) = \inf\{d(s, x_j) + c(x_j, x_i) \mid j > i, x_j \to x_i\}.$$

La formule du lemme 8.12, aussi appelée équation de Bellman dans la littérature, permet de calculer $d(s, x_i)$ par récurrence décroissante sur i. L'algorithme ainsi obtenu est

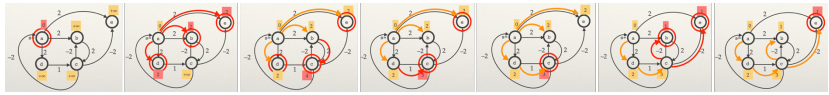

FIGURE 1 – Un exemple d'exécution de l'algorithme de Ford

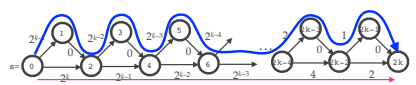

FIGURE 2 – Un exemple d'un graphe valué sur lequel l'algorithme de Ford à liste de travail prend un temps exponentiel

	détecte les circuits <0?	cas général	pas de circuit <0	pas d'arc<0	pas de circuit
Bellman-Ford	oui	$\Theta(n(m+n))$	$\Theta(n(m+n))$	$\Theta(n(m+n))$	$\Theta(n(m+n))$
Ford (FIFO)	non	–	$\Theta(n(m+n))$	$\Theta(n(m+n))$	$\Theta(n(m+n))$
Ford (LIFO)	non	–	expo.	expo.	expo.
Dijkstra	non	–	expo.	$\Theta(m+n\log n)$	expo.
Bellman	non	–	–	–	$\Theta(m+n)$

TABLE 18 – Une vue synthétique des algorithmes de calculs de distances minimums à sommet de départ fixé (on suppose les opérations arithmétiques de base en temps constant)

appelé *algorithme de Bellman* dans le BBC. Il ne faut pas le confondre avec l'algorithme de Bellman-Ford.

On a besoin d'itérer sur les sommets dans un ordre $x_1, x_2, \cdots, x_k$ obtenu par tri topologique inverse. Il suffit de modifier l'algorithme `dfs_2` et d'y ajouter l'affectation suivante en dernière ligne, juste après l'affectation à `rto(u)` :

```
sommets_tries [rto(u)] := u;
```

Théorème 8.13 *L'algorithme de Bellman décrit ci-dessus calcule correctement les distances minimums d'un sommet s de G à tout sommet, à condition que G n'ait aucun circuit. Sa complexité est en $O(m+n)$.*

8.8 Retour sur les algorithmes de distances minimums entre tous couples de sommets : l'algorithme de Johnson

Soit $G \stackrel{\text{def}}{=} (S, A, c\colon A \to \mathbb{R})$ un graphe valué. On fabrique un nouveau graphe valué $G_* \stackrel{\text{def}}{=} (S_*, A_*, c_*)$ où :

- S_* est l'union disjointe de S et d'un nouveau sommet $*$;
- $A_* \stackrel{\text{def}}{=} A \cup \{(*, u) \mid u \in S\}$, autrement dit les arcs de G_* sont ceux de G, plus un arc de $*$ à chacun des sommets de G ;
- $c_*(e) \stackrel{\text{def}}{=} c(e)$ pour tout arc $e \in A$, et $c_*(*, u) \stackrel{\text{def}}{=} 0$ pour tout $u \in S$. (Note : j'écris $c_*(u, v)$ plutôt que l'inélégant $c_*((u, v))$.)

```
fun johnson (G as (S, succ, c), n) =
1     let * = fresh() in
2     let S* = *::S in
3     let fun succ* (x,y) =
          if x=* then S else succ (x) in
4     let fun c* (u,v) =
          if u=* then 0 else c (u,v)) in
5     let G* = (S ∪ {*}, succ*, c*) in
6     let h = bellman-2 (G*, *, n+1) in
7     let fun c** (u,v) = c* (u,v)+h[u]-h[v] in
8     let G** = (S*, succ*, c**) in
9     for each s ∈ S do
10          D [s] = ford (G**, s, n+1);
11    let fun d (u, v) = D[u][v]+h[v]-h[u] in
12        return d;
```

TABLE 19 – L'algorithme de Johnson pour les distances minimum entre tous couples de sommets d'un graphe sans circuit de poids strictement négatif

Supposons que G, et donc G_*, n'a pas de circuit de coût strictement négatif. On note $h(u)$, pour tout $u \in S_*$, le coût minimum d'un chemin de $*$ à u dans G_*. On définit un nouveau graphe $G_{**} \stackrel{\text{def}}{=} (S_*, A_*, c_{**})$, de mêmes sommets et mêmes arcs que G_*, mais avec $c_{**}(u, v) \stackrel{\text{def}}{=} c_*(u, v) + h(u) - h(v)$.

Lemme 8.14 *Si G n'a pas de circuit de coût strictement négatif, alors G_{**} n'a pas d'*arc *de coût strictement négatif.*

L'algorithme de Johnson, dans sa forme la plus simple, calcule h grâce à l'algorithme de Bellman-Ford, puis G_{**}, et calcule ensuite toutes les distances minimum entre sommets de G_{**} en utilisant $O(n)$ fois l'algorithme de Dijkstra, autant de fois qu'il y a de sommets de départ `s` possibles. Les distances minimum dans le graphe G de départ s'en déduisent grâce à la proposition suivante.

Proposition 8.15 *Soit d la fonction qui à tout couple (u, v) de sommets de G associe le coût minimum d'un chemin de u à v dans G. Soit D la fonction qui à tout couple (u, v) de sommets de G_{**} associe le coût minimum d'un chemin de u à v dans G_{**}. Pour tous $u, v \in S$, $d(u, v) = D(u, v) + h(v) - h(u)$.*

Théorème 8.16 *L'algorithme de Johnson de la table 19 calcule correctement en `d` la fonction des distances minimums entre tout couple de sommets d'un graphe G à n sommets donné en entrée, si G n'a aucun circuit de coût strictement négatif. Sa complexité en temps est de $O(nm + n^2 \log n)$.*

Remarque 8.17 *On peut modifier l'algorithme de Johnson pour qu'il détecte les circuits de poids négatifs. Il suffit pour*

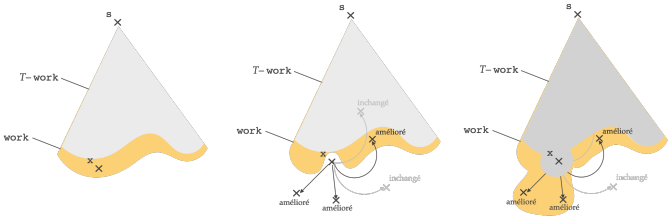

FIGURE 3 – Ce que fait l'algorithme de Dijkstra

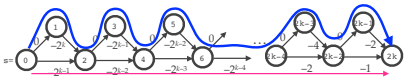

FIGURE 4 – Un exemple d'un graphe valué sur lequel l'algorithme de Dijkstra prend un temps exponentiel ; la contrainte « pas d'arc de coût strictement négatif » n'est pas respectée, mais il n'y a pas de circuit de coût strictement négatif

cela d'utiliser `bellman-1-detect` *(table 16) à la place de* `bellman-2` *à la ligne 6 de la table 19.*

Remarque 8.18 *Tous les algorithmes de la section 8 ont des complexités qui sont données en supposant les opérations de base sur les réels en temps constant, et ceci vaut donc aussi pour l'algorithme de Johnson.*

9 Réseaux, flots et coupes

Définition 9.1 *Un* réseau *est le donné d'un graphe valué* $G \stackrel{def}{=} (S, A, c_{\max})$ *sans boucle, où* $c_{\max}$ *est à valeurs dans* $\mathbb{R}_+$*, et de deux sommets* s *(la* source*) et* p *(le* puits*) tels que :*

- $s \neq p$;
- *tout sommet* u *est accessible depuis* s *(il y a un chemin* $s \to^* u$*),*
- *et est coaccessible depuis* p *(il y a un chemin* $u \to^* p$*).*

Pour tout arc e*, on appelle* $c_{\max}(e)$ *la* capacité maximale *de* e*.*

9.1 Flots et coupes

Définition 9.2 *Un* flot *dans un réseau* (G, s, p)*, où* $G \stackrel{def}{=} (S, A, c_{\max})$*, est une fonction* $f \colon A \to \mathbb{R}_+$ *telle que :*

1. *Pour tout sommet* u *de* G *différent de* s *et de* p*,* $\sum_{v/v\to u} f(v \to u) = \sum_{w/u\to w} f(u \to w)$;
2. *pout tout arc* e*,* $f(e) \leq c_{\max}(e)$.

La condition du point 1 est la condition de *conservation du flot en* u. La condition du point 2 est la condition d'*admissibilité*.

Remarque 9.3 *Rien dans la définition ne demande à ce qu'il n'y ait pas d'arc* $u \to s$ *ou* $p \to v$.

Définition 9.4 *Une* coupe *d'un réseau* (G, s, p) *est un ensemble de sommets* E *de* G *contenant* s *mais pas* p*. Pour tout fonction* $f \colon A \to \mathbb{R}_+$*, où* A *est l'ensemble des arcs de* G*, et notamment pour tout flot, on pose :*

$$\mathrm{out}(f, E) \stackrel{def}{=} \sum_{(u,w)\in A, u\in E, w\notin E} f(u, w)$$
$$\mathrm{in}(f, E) \stackrel{def}{=} \sum_{(v,u)\in A, v\notin E, u\in E} f(v, u).$$

Les arcs (u, w) *tels que* $u \in E$ *et* $w \notin E$ *sont les* arcs sortants *de* E*, les arcs* (v, u) *tels que* $v \notin E$ *et* $u \in E$ *sont les* arcs entrants *de* E*.*

Lemme 9.5 *Pour tout flot* f *sur un réseau* (G, s, p)*, la quantité* $\mathrm{out}(f, E) - \mathrm{in}(f, E)$ *est la même pour toutes les coupes* E*.*

Définition 9.6 *La* valeur d'un flot f *sur un réseau* (G, s, p) *est :*

$$\mathrm{val}(f) \stackrel{def}{=} \mathrm{out}(f, E) - \mathrm{in}(f, E),$$

pour n'importe quelle coupe E *du réseau* G*, par exemple* $\{s\}$ *ou* $A \setminus \{p\}$*,* A *étant l'ensemble des arcs de* G*.*

9.2 Flots maximaux, coupes minimales

Définition 9.7 *La* valeur d'une coupe E *d'un réseau* (G, s, p)*, où* $G \stackrel{def}{=} (S, A, c_{\max})$*, est :*

$$\mathrm{b}(E) \stackrel{def}{=} \mathrm{out}(c_{\max}, E) = \sum_{u\in E\to w\notin E} c_{\max}(u \to w),$$

la somme des capacités maximales des arcs sortants de E*.*

Lemme 9.8 *Pour tout flot* f *sur un réseau* (G, s, p)*, pour tout coupe* E*,* $\mathrm{val}(f) \leq \mathrm{b}(E)$*.*

Remarque 9.9 *Étant donné un flot* f *sur* (G, s, p)*, il suffit d'exhiber une coupe* E *telle que* $\mathrm{val}(f) = \mathrm{b}(E)$ *pour garantir que* f *est un flot maximal, par le lemme 9.8.*

De façon symétrique, étant donnée une coupe E*, il suffit d'exhiber un flot* f *tel que* $\mathrm{val}(f) = \mathrm{b}(E)$ *pour garantir que* E *est une coupe minimale.*

9.3 Le graphe d'écart, et les chemins améliorants

Définition 9.10 *Soit* (G, s, p) *un réseau et* f *un flot sur ce réseau, ou plus généralement une fonction des arcs de* G *vers* $\mathbb{R}_+$ *telle que* $0 \leq f(e) \leq c_{\max}(e)$ *pour tout arc* e *de* G*.*

Un arc $x \to y$ *de* G *est* libre *(par rapport à* f*) si* $f(x, y) < c_{\max}(x, y)$*, et* saturé *sinon.*

Un arc $x \to y$ *de* G *est* passant *(par rapport à* f*) si* $f(x, y) > 0$*, et est* inerte *sinon.*

Le graphe d'écart G_f *de* f *est le graphe valué ayant les mêmes sommets que* G*, et avec deux types d'arcs, que l'on notera* $\to_f$ *pour rappeler la dépendance en le flot* f*, et une fonction valeur* r_f*, définis comme suit :*

- *pour tout arc libre* $x \to y$ *de* G*, un arc* $x \to_f y$*, dit* arc conforme*, de valeur* $r_f(x \to_f y) \stackrel{def}{=} c_{\max}(x, y) - f(x, y)$;
- *pour tout arc passant* $x \to y$ *de* G*, un arc* $y \to_f x$*, dit* arc non conforme*, de valeur* $r_f(y \to_f x) \stackrel{def}{=} f(x, y)$.

Dans le premier cas, on dit que $x \to_f y$ *est le* représentant conforme *de l'arc libre* $x \to y$*. Dans le deuxième cas,* $y \to_f x$ *est le* représentant non conforme *de l'arc passant* $x \to y$*.*

Remarque 9.11 *Le graphe d'écart est en fait un multigraphe.*

Remarque 9.12 *Tout arc de* G *a 0, 1 ou 2 représentants.*

Le seul cas où $x \to y$ *a deux représentants est celui où* $x \to y$ *est à la fois libre et passant, c'est-à-dire où* $0 < f(x \to y) < c_{\max}(x \to y)$*. Dans ce cas, les deux représentants sont dits* conjoints*.*

Lemme 9.13 *Tout arc de* G_f *est représentant d'un arc unique de* G*. Pour tout arc* e *de* G*, la somme des valeurs de* r_f *sur les* 0*,* 1 *ou* 2 *représentants de* e *vaut toujours* $c_{\max}(e)$*.*

Définition 9.14 *Une* chemin améliorant μ *pour un flot* f *sur un réseau* (G, s, p) *est un chemin élémentaire de* s *à* p *dans le graphe d'écart* G_f*.*

Lemme 9.15 *On ne peut pas trouver à la fois le représentant conforme et le représentant non conforme d'un même arc de* G *dans aucun chemin améliorant* μ*.*

Définition 9.16 *L'*amélioration $\alpha(\mu)$ *d'un chemin améliorant* μ *est le plus petit coût (tel que calculé par* r_f*) d'un arc de* μ*.*

Lemme 9.17 *Soit* μ *un chemin améliorant pour un flot* f *sur un réseau* (G, s, p)*. La fonction* $f.\mu$ *qui à tout arc* e *de* G *associe :*

- $f(e)$ *si aucun représentant de* e *n'apparaît dans* μ*,*
- $f(e) + \alpha(\mu)$ *si* e *a un représentant conforme* e'*, et* e' *est l'un des arcs de* μ*,*
- $f(e) - \alpha(\mu)$ *si* e *a un représentant non conforme* e''*, et* e'' *est l'un des arcs de* μ*,*

est un flot, et $\mathrm{val}(f.\mu) = \mathrm{val}(f) + \alpha(\mu)$*.*

Corollaire 9.18 *Si un flot* f *sur un réseau* (G, s, p) *a un chemin améliorant* μ*, alors* $\mathrm{val}(f)$ *n'est pas maximum.*

Lemme 9.19 *S'il n'existe pas de chemin améliorant pour un flot* f *sur un réseau* (G, s, p)*, alors l'ensemble* E *des sommets accessibles depuis* s *dans le graphe d'écart* G_f *est une coupe et* $\mathrm{val}(f) = \mathrm{b}(E)$*.*

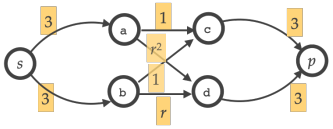

FIGURE 5 – Le réseau de Zwick ; $\lambda \approx 0,216757$ est l'unique racine réelle de $1 - 5x + 2x^2 - x^3$, $r = (1 + \sqrt{1 - 4\lambda})/2 \approx 0,682378$; les capacités maximum sont affichées sur fond orange

9.4 L'algorithme de Ford-Fulkerson, et la question de sa terminaison

Il existe des réseaux, et des stratégies de sélection de chemins améliorants qui font que l'algorithme de Ford-Fulkerson ne termine pas. L'exemple le plus petit est dû à Zwick. voir la figure 5. La stratégie consistant à choisir les chemins améliorants $sacbdp$, $sadbcp$, $sbdacp$, $sbcadp$ dans cet ordre, et à répéter ces quatre choix à l'infini, ne termine pas.

9.5 Distances estimées et le graphe d'admissibilité

Définition 9.20 *Soit* f *un flot sur un réseau* (G, s, p)*, ou plus généralement une fonction des arcs de* G *vers* $\mathbb{R}_+$ *telle que* $0 \leq f(e) \leq c_{\max}(e)$ *pour tout arc* e *de* G*. Une* distance estimée *relative au flot* f *(ou* sur *le graphe d'écart* G_f*) est une fonction* Δ *des sommets de* G *vers* $\mathbb{N}$ *telle que :*

1. $\Delta(p) = 0$;
2. *pour tout arc* $x \to_f y$ *du graphe d'écart* G_f*,* $\Delta(x) \leq \Delta(y) + 1$.

Lemme 9.21 *Pour tout flot* f *sur un réseau* (G, s, p)*, pour tout distance estimée* Δ *relative au flot* f*, pour tout sommet* x *de* G*,* $\Delta(x)$ *est inférieur ou égal à la longueur du plus petit chemin de* x *à* p *dans le graphe d'écart* G_f *(si un tel chemin existe).*

Corollaire 9.22 *Pour tout flot* f *sur un réseau* (G, s, p)*, si* Δ *est une distance estimée relative au flot* f *telle que* $\Delta(s) \geq n$*, alors il n'existe pas de chemin de* s *à* p *dans le graphe d'écart* G_f*, et donc* f *est un flot de valeur maximum, égale à la valeur de la coupe* E *formée des sommets accessibles dans le graphe d'écart* G_f*.*

Définition 9.23 *Soit* f *un flot sur un réseau* (G, s, p)*, et* Δ *une distance estimée relative à* f*. Un arc* $x \to_f y$ *du graphe d'écart* G_f *est un* arc admissible *si et seulement* $\Delta(x) = \Delta(y) + 1$.

Un chemin admissible *est un chemin améliorant dont tous les arcs sont admissibles.*

Remarque 9.24

1. *La longueur d'un chemin admissible vaut exactement* $\Delta(s)$*. C'est un chemin améliorant* de longueur minimum.
2. *Si* $\Delta(s) < n$*, alors* $\Delta(x) < n$ *pour tout sommet* x *d'un chemin admissible quelconque.*
3. *Un chemin admissible est la même chose qu'un chemin* $\mu \colon s \to_f^* p$ *dans le graphe d'écart dont tous les arcs*

```
   fun init-delta (G as (S,succ,pred),p,f,cmax,n)=
1    let Delta = new(n) in for each x ∈ S do
2                             Delta[x]:=n;
3                             (* p[x]:=undef : inutile *)
4    Delta[p]:=0;
5    work:=empty;
6    push(work,p);
7    while nonempty(work) do
8          let x=pop(work) in
9              for each y ∈ succ(x) do
10                 if f(x,y)<cmax(x,y)
11                    let improve = Delta[x]+1 in
12                    if improve<Delta[y]
13                       then Delta[y]:=improve;
14                            (* p[y]:=x : inutile *)
15                            push_opt(work,y);
16             for each y ∈ pred(x) do
17                 if f(y,x)>0
18                    let improve = Delta[x]+1 in
19                    if improve<Delta[y]
20                       then Delta[y]:=improve;
21                            (* p[y]:=x : inutile *)
22                            push_opt(work,y);
23   return Delta;
```

TABLE 20 – L'initialisation de la distance estimée ; les opérations `empty`, `push`, `pop`, `nonempty` sont des opérations de files (FIFO)

sont admissibles, et on n'a pas besoin d'imposer qu'il soit élémentaire. Il l'est toujours.

Définition 9.25 *Soit f un flot sur un réseau (G,s,p), ou plus généralement une fonction des arcs de G vers $\mathbb{R}_+$ telle que $0 \le f(e) \le c_{\max}(e)$ pour tout arc e de G, et Δ une distance estimée relative à f. Le* graphe d'admissibilité *$G_{f,\Delta}$ est le sous-graphe couvrant de G_f dont les arcs sont les arcs admissibles.*

9.6 L'algorithme de Dinic et Edmonds-Karp, et le théorème max-flow min-cut

L'algorithme de Dinic-Edmonds-Karp se décrit maintenant, de façon encore relativement abstraite, comme suit.

1. Initialisation : on initialise f au flot nul, et pour tout sommet x de G, on initialise $\Delta(x)$ à la longueur du plus court chemin $x \to_f^* p$ dans le graphe d'écart G_f.
2. Tant que $\Delta(s) < n$, on alterne entre deux opérations :
   (a) trouver un chemin admissible μ, et *améliorer le flot*, c'est-à-dire poser $f := f.\mu$;
   (b) ou *mettre à jour* la distance estimée Δ de sorte à augmenter les distances estimées, sans changer f.

Lemme 9.26 *Soit f un flot sur un réseau (G,s,p), et Δ une distance estimée relative à f. Pour tout chemin admissible μ, Δ est encore une distance estimée relative à $f.\mu$.*

Lemme 9.27 *Soit f un flot sur un réseau (G,s,p), et Δ une distance estimée relative à f. Pour tout chemin admissible μ, l'ensemble des arcs du graphe d'admissibilité $G_{f.\mu,\Delta}$ est strictement inclus dans l'ensemble des arcs du graphe d'admissibilité $G_{f,\Delta}$.*

Définition 9.28 *Soit f un flot sur un réseau (G,s,p), et Δ une distance estimée relative à f. Un* chemin admissible partiel *est un chemin $s = x_0 \to_f x_1 \to_f \cdots \to_f x_k = x$ composé d'arcs admissibles. Il est* bloqué *si et seulement si $x \neq p$ et pour tout arc $x \to_f y$ du graphe d'écart G_f, $\Delta(x) \neq \Delta(y)+1$.*

Définition 9.29 *Soit f un flot sur un réseau (G,s,p), et Δ une distance estimée relative à f. Pour tout chemin admissible partiel bloqué μ : $s \to_f^* x$, on pose $\mu.\Delta$ la fonction qui :*

1. *à x associe $\min\{\Delta(y)+1 \mid y$ successeur de x dans $G_f\}$ si x a au moins un successeur dans G_f, n sinon.*
2. *à tout autre sommet z associe $\Delta(z)$.*

Ceci reste une distance estimée, et de plus une qui améliore Δ dans le sens expliqué ci-dessous.

Lemme 9.30 *Soit f un flot sur un réseau (G,s,p), et Δ une distance estimée relative à f. Pour tout chemin admissible partiel bloqué μ : $s \to_f^* x$, $\mu.\Delta$ est une distance estimée. De plus, $(\mu.\Delta)(y) \ge \Delta(y)$ pour tout sommet y, l'inégalité étant stricte si $y = x$.*

On raffine la description abstraite de l'algorithme de Dinic-Edmonds-Karp comme suit. Les configurations sont des triplets (f,Δ,μ) formés d'un flot f sur le réseau (G,s,p), d'une distance estimée Δ relative à f, et d'un chemin admissible partiel μ :

2(a) $(f,\Delta,\mu) \rightsquigarrow (f.\mu,\Delta,s)$, où μ est un chemin admissible dans G_f (la notation s dans $(f.\mu,\Delta,s)$ dénote le chemin admissible partiel réduit au seul sommet s) ;

2(b) $(f,\Delta,\mu) \rightsquigarrow (f,\mu.\Delta,\mathrm{chop}(\mu))$, où μ est un chemin admissible partiel bloqué dans G_f ; la fonction chop retourne μ si μ est de longueur 0, et sinon μ moins son dernier arc ;

2(c) $(f,\Delta,\mu) \rightsquigarrow (f,\Delta,\mathrm{extend}(\mu,y))$ où m : $s \to_f^* x$ n'est pas bloqué ; il existe alors un sommet y successeur de x dans G_f tel que $\Delta(x) = \Delta(y)+1$, et l'on note $\mathrm{extend}(\mu,y)$ le chemin (admissible partiel) obtenu en ajoutant l'arc $x \to_f y$ à la fin de μ.

Ces règles ne sont appliquées qu'à partir de configurations (f,Δ,μ) telles que $\Delta(s) < n$.

Lemme 9.31 *Si μ est un chemin admissible partiel bloqué, relativement à une fonction distance estimée Δ, alors $\mathrm{chop}(\mu)$ est un chemin admissible partiel relativement à $\mu.\Delta$.*

Théorème 9.32 (Max-flow min-cut, Ford-Fulkerson) *Pour tout réseau (G,s,p), $\max_{f \text{ flot}} \mathrm{val}(f) = \min_{E \text{ coupe}} \mathrm{b}(E)$.*

9.7 La complexité de l'algorithme de Dinic-Edmonds-Karp

Lemme 9.33 *Soit $(f_0,\Delta_0,\mu_0) \rightsquigarrow (f_1,\Delta_1,\mu_1) \rightsquigarrow \cdots \rightsquigarrow (f_k,\Delta_k,\mu_k)$ une suite de transitions **2(a)**, **2(b)** ou **2(c)**, portant sur des configurations (f_i,Δ_i,μ_i) telles que f_i est un flot du réseau (G,s,p), Δ_i est une distance estimée relative à f_i, et μ_i est un chemin admissible partiel.*

Un couple (x,y) de sommets est un arc effacé *à la transition i si et seulement $x \to_{f_{i-1}} y$ est un arc de $G_{f_{i-1}}$ mais $x \to_{f_i} y$ n'est pas un arc de G_{f_i}. Un couple (x,y) de sommets est un* arc ajouté *à la transition i si et seulement $x \to_{f_{i-1}} y$ n'est pas un arc de $G_{f_{i-1}}$ mais $x \to_{f_i} y$ est un arc de G_{f_i}.*

Pour tout couple de sommets (x,y), si $x \to y$ est effacé ou ajouté à la transition i, alors :

1. *la transition $(f_{i-1},\Delta_{i-1},\mu_{i-1}) \rightsquigarrow (f_i,\Delta_i,\mu_i)$ est de type **2(a)** ;*
2. *$\Delta_i = \Delta_{i-1}$.*

Si $x \to y$ est un arc effacé à la transition i, alors :

3. *$x \to_{f_{i-1}} y$ est un arc du chemin μ_{i-1} ;*
4. *$\Delta_i(x) = \Delta_i(y)+1$.*

Si $x \to y$ est un arc ajouté à la transition i, alors :

5. *le conjoint $y \to_{f_{i-1}} x$ est un arc du chemin μ_{i-1} ;*
6. *$\Delta_i(y) = \Delta_i(x)+1$.*

Lemme 9.34 *Soit $(f_0,\Delta_0,\mu_0) \rightsquigarrow (f_1,\Delta_1,\mu_1) \rightsquigarrow \cdots \rightsquigarrow (f_k,\Delta_k,\mu_k)$ une suite de transitions comme au lemme 9.33, avec $\Delta_{i-1}(s) < n$ pour tout $i \in \{1,\cdots,k\}$. Pour tout couple de sommets (x,y), l'arc $x \to y$ ne peut être effacé qu'à au plus $\lceil \frac{n}{2} \rceil \le \frac{n+1}{2}$ des k transitions.*

Lemme 9.35 *Soit $(f_0,\Delta_0,\mu_0) \rightsquigarrow (f_1,\Delta_1,\mu_1) \rightsquigarrow \cdots \rightsquigarrow (f_k,\Delta_k,\mu_k)$ une suite de transitions comme au lemme 9.33, avec $\Delta_{i-1}(s) < n$ pour tout $i \in \{1,\cdots,k\}$. Alors au plus $m(n+1)$ de ces transitions sont de type **2(a)**.*

Lemme 9.36 *Soit $(f_0,\Delta_0,\mu_0) \rightsquigarrow (f_1,\Delta_1,\mu_1) \rightsquigarrow \cdots \rightsquigarrow (f_k,\Delta_k,\mu_k)$ une suite de transitions comme au lemme 9.33, avec $\Delta_{i-1}(s) < n$ pour tout $i \in \{1,\cdots,k\}$. Pour tout sommet x de G, il n'y qu'au plus n transitions $(f_{i-1},\Delta_{i-1},\mu_{i-1}) \rightsquigarrow (f_i,\Delta_i,\mu_i)$ de type **2(b)** telles que $\Delta_{i-1}(x) < \Delta_i(x)$.*

Lemme 9.37 *Soit $(f_0,\Delta_0,\mu_0) \rightsquigarrow (f_1,\Delta_1,\mu_1) \rightsquigarrow \cdots \rightsquigarrow (f_k,\Delta_k,\mu_k)$ une suite de transitions comme au lemme 9.33, avec $\Delta_{i-1}(s) < n$ pour tout $i \in \{1,\cdots,k\}$. La transition $(f_{i-1},\Delta_{i-1},\mu_{i-1}) \rightsquigarrow (f_i,\Delta_i,\mu_i)$ est la* transition i. *Disons que :*

- *un sommet z* apparaît à la transition *i si et seulement si on le trouve sur le chemin μ_i mais pas sur le chemin μ_{i-1}, ou bien si $z = s$ et la transition i est de type **2(a)** ;*
- *un sommet z il* disparaît à la transition *i si et seulement si on le trouve sur le chemin μ_{i-1} et pas sur le chemin μ_i, ou bien si $z = s$ et la transition i est de type **2(a)** ;*
- *un* segment *d'un sommet z est un intervalle entier $[j,\ell]$ avec $1 \le j < \ell \le k+1$, tel que z apparaît à la transition j, et ℓ est le plus petit entier $\ell \in \{j+1,\cdots,k\}$ tel que z disparaît à la transition ℓ, ou bien $k+1$ si un tel entier n'existe pas ;*
- *un couple de sommets (x,y) est* candidat à l'extension à la transition *i si x est l'extrémité du chemin μ_{i-1}, y est un successeur de x dans $G_{f_{i-1}}$ (donc $y \in \mathrm{succ}(x) \cup \mathrm{pred}(x)$), et $\Delta_{i-1}(x) = \Delta_{i-1}(y)+1$, autrement dit si les conditions d'applicabilité d'une transition de type **2(c)** sont vérifiées.*

Alors :

```
   fun dinic-extend(musuccx,mupredx,f,cmax,Delta,x)=
1    while musuccx≠[] do
2      let y::rest=musuccx in
3          (musuccx := rest;
4           if f(x,y)<cmax(x,y) and Delta[x]=Delta[y]+1
5              then return y, true);
6    while mupredx≠[] do
7      let y::rest=mupredx in
8          (mupredx := rest;
9           if f(y,x)>0 and Delta[x]=Delta[y]+1
10             then return y, false;
11   raise NotFound;
```

TABLE 21 – La recherche d'un sommet permettant d'étendre un chemin partiel admissible

```
   fun rf (f, cmax, x, y, est-conforme) =
1    if est-conforme
2       then return cmax(x,y)-f(x,y)
3    else return f(y,x);
```

TABLE 22 – Le calcul de $r_f(x,y)$ pour un arc $x \to_f y$ du graphe d'écart

1. *Pour tout sommet x de G, pour tout segment $[j,\ell]$ de x, pour tout sommet y de G, l'une des trois possibilités suivantes se présente, de façon exclusive :*
   (a) *(x,y) est candidat à l'extension à toute transition de $j+1, \ldots, \ell-1$;*
   (b) *(x,y) n'est candidat à l'extension à aucune transition de $j+1, \ldots, \ell-1$;*
   (c) *il existe un unique entier $p \in \{j+1,\cdots,\ell-1\}$ tel que y apparaît à la transition p, (x,y) est candidat à l'extension à toute transition $j+1, \ldots, p-1$ et ne l'est à aucune transition $p+1, \ldots, \ell-1$.*
2. *Tout sommet x disparaît à au plus $m(n+1)+n$ transitions.*
3. *Tout sommet x a au plus $(m+1)(n+1)$ segments. , il y a au plus $(m+1)(n+1)(\#\mathrm{succ}(x)+\#\mathrm{pred}(x))$ transitions i de type **2(c)** telles que l'extrémité de μ_{i-1} soit le sommet x.*

Remarque 9.38 *Dans un réseau, $2n-4 \le m$. Donc on peut simplifier $O(m+n)$ en $O(m)$. Ce genre de raisonnement est faux sur un graphe arbitraire.*

Théorème 9.39 *L'algorithme de Dinic-Edmonds-Karp calcule correctement un flot de valeur maximum d'un réseau donné en entrée en temps $O(nm^2)$.*

```
   fun alpha (f, cmax, mu, conforme, nmu) =
1    a := rf (f, cmax, mu[0], mu[1], conforme[0]);
2    for i=1 to nmu-1 do
3      let new-a=rf(f,cmax,mu[i],mu[i+1],conforme[i]) in
4          if new-a<a
5             then a := new-a;
6    return a;
```

TABLE 23 – Le calcul de $\alpha(\mu)$

```
   fun update-flow (f, cmax, mu, conforme, nmu) =
1      a := alpha (f, cmax, mu, conforme, nmu);
2      for i=0 to nmu-1 do
3          if conforme[i]
4             then f (mu[i], mu[i+1]) += alpha;
5          else f (mu[i+1], mu[i]) -= alpha;
```

TABLE 24 – La mise à jour d'un flot par un chemin améliorant (lemme 9.17)

```
   fun min-delta (succ, pred, f, cmax, Delta, x) =
1      d := n;
2      for each y ∈ succ(x) do
3              if f(x,y)<cmax(x,y) do
4                 d := min (d, Delta[y]+1);
5      for each y ∈ pred(x) do
6              if f(y,x)>0 do
7                 d := min (d, Delta[y]+1);
8      return d;
```

TABLE 25 – Le calcul de $\min\{\Delta(y) + 1 \mid y \text{ successeur de } x \text{ dans } G_f\}$ (ou n si l'ensemble est vide)

```
   fun dinic-edmonds-karp(G as (S,succ,pred),s,p,cmax,n)=
1     let f=new-flow(n) in for each x ∈ S do
2                                for each y ∈ succ(x) do
3                                    f(x,y) := 0;
4     Delta := init-delta (G, p, f, cmax, n);
5     mu:=new(n); mu[0]:=s;
      musucc[0]:=succ(s); mupred[0]:=pred(s); nmu:=0;
6     conforme := new(n);
7     while Delta[s]<n do
8       try (while mu[nmu]≠p do
9              (mu[nmu+1], conforme[nmu] :=
                  dinic-extend (&musucc[nmu],
                                &mupred[nmu], f,
                                cmax, Delta, mu[nmu]);
10             nmu := nmu+1;
11             musucc[nmu] := succ(mu[nmu]);
12             mupred[nmu] := pred(mu[nmu]));
13           update-flow (f, cmax, mu, conforme, nmu);
14           mu[0] := s; nmu := 0);
15      match NotFound =>
16        (Delta[mu[nmu]]:=min-delta(succ,pred,f,
                                   cmax,Delta,mu[nmu]);
17         if nmu≠0 then nmu := nmu-1;)
18    return f;
```

TABLE 26 – L'algorithme de Dinic-Edmonds-Karp

```
   fun flow-val (G as (S, succ, pred), f, s) =
1      v := 0;
2      for each v ∈ succ(s) do
3              v := v+f(s,v);
4      for each v ∈ pred(s) do
5              v := v-f(v,s);
6      return v;
```

TABLE 27 – Le calcul de la valeur d'un flot $\text{val}(f, \{s\})$

Remarque 9.40 *Le BBC (théorème 3.6, section 8.3.2, p. 262) annonce une complexité en $O(n^2m)$; il semble que ce soit une erreur.*

Remarque 9.41 *L'analyse de complexité donnée de l'algorithme de Dinic-Edwards-Karp suppose que les opérations arithmétiques faites sur les coûts sont en temps constant.*

9.8 L'algorithme par échelonnement d'Edmonds-Karp et Dinic

Ce nouvel algorithme s'applique dans le cas particulier où les capacités maximales sont *entières*, ou de façon plus générale, toutes multiples entières d'un même réel strictement positif, ce qui est notamment le cas si les capacités maximales sont rationnelles.

Définition 9.42 *Soit f un flot sur un réseau (G, s, p). Pour tout $D > 0$, le* graphe d'écart tronqué $G_f(D)$ *est le sous-graphe couvrant de G_f formé des arcs e de coûts $r_f(e) \geq D$.*

Lemme 9.43 *Soit (G, s, p) un réseau à m arcs et $v_{\max}$ la valeur de son flot maximum. Soit f un flot sur (G, s, p), et $D > 0$. S'il n'existe pas de chemin améliorant dans $G_f(D)$, alors $\text{val}(f) \geq v_{\max} - mD$.*

L'algorithme par échelonnement (« capacity scaling », en anglais) d'Edmonds-Karp et Dinic fonctionne comme suit :

1. Initialisation. On pose f égal au flot nul, et on calcule un entier $k \geq 1$ tel que $2^k \geq c_{\max}(e)$ pour tout arc e de G, le plus petit possible, en temps $O(n + m)$ (par un parcours de S puis des listes de successeurs de chaque sommet ; pour tout entier a, trouver le plus petit entier $k \geq 1$ tel que $2^k \geq a$ est supposé effectué en temps constant).
2. Tant que $k \geq 1$:
   (a) Tant qu'il existe un chemin améliorant μ dans $G_f(2^{k-1})$, $f := f.\mu$ (améliorer le flot) ;
   (b) décrémenter k.

L'invariant principal de cet algorithme, vrai au début de l'étape 2, est :

(Inv 1) $\text{val}(f) \geq v_{\max} - m2^k$.

Théorème 9.44 *Sur un réseau (G, s, p) à capacités maximums entières donné en entrée, l'algorithme par échelonnement d'Edmonds-Karp et Dinic retourne un flot maximum en temps $O(m^2 \max(1, \log C))$, où C est la capacité maximum des arcs de G.*

Remarque 9.45 *Dans le cas d'entiers machine, il faut faire attention aux débordements arithmétiques ; dans le cas d'entiers en grande précision, la complexité de l'algorithme est en fait en $O(m^2 \log C(\log C + \log n))$.*

9.9 Chemins disjoints, et le théorème de Menger

Théorème 9.46 *Pour tout réseau (G, s, p), si les capacités maximums des arcs sont toutes entières, alors il en est de même de la valeur du flot maximum, et il existe un flot maximum f tel que $f(e)$ soit entier pour tout arc e de G.*

On a le même résultat en remplaçant « entier » par « multiple entier d'une valeur $c > 0$ fixée » ou par « rationnel ».

```
   fun augmenting-path (G as (S,succ,pred),s,f,cmax,D)=
1    work := empty; for each u ∈ S do push (u, work);
2    marked := {s};
3    while nonempty(work)) do
4      let u=pop(work) in
5        if u ∉ marked
7        then (for each v ∈ succ(u) do
8                if f(u,v)<cmax(u,v) and
                    cmax(u,v)-f(u,v)>=D
9                then (push (v, work);
10                     pT[v]:=u; conforme[v]:=true);
11           for each v ∈ pred(u) do
12             if f(v,u)>0 and f(v,u)>=D
13                then (push (v, work);
14                     pT[v]:=u; conforme[v]:=false);
15           marked:=marked ∪ {u});
```

TABLE 28 – La recherche de chemin améliorant dans $G_f(D)$

```
   fun alpha-pT (f, cmax, pT, conforme, s, p) =
1      u := pT[p];
2      a := rf (f, cmax, u, p, conforme[p]);
3      while u≠s do
4        let new-a = rf(f,cmax,pT[u],u,conforme[u]) in
5            (u := pT[u];
6             if new-a<a
7                then a := new-a);
8      return a;
```

TABLE 29 – Le calcul de $\alpha(\mu)$ où μ est un chemin améliorant donné par prédécesseurs `pT` dans un arbre couvrant du graphe d'écart

```
   fun update-flow-pT (f, cmax, pT, conforme, s, p) =
1      a := alpha-pT (f, cmax, pT, conforme, s, p);
2      u := p;
3      while u≠s do
3            (if conforme[u]
4                then f (pT[u], u) += alpha;
5             else f (u, pT[u]) -= alpha;
6             u := pT[u])
```

TABLE 30 – La mise à jour d'un flot par un chemin améliorant donné par prédécesseurs `pT` dans un arbre couvrant du graphe d'écart

```
   fun fitting-bit-mask (n) =
1      if n=0 then return 1;
2      let a = n & (-n) in
3          if a=n
4             then return a;
5          else return a«1;
```

TABLE 31 – Le calcul de 2^k où k est le plus petit entier $k \geq 0$ tel que $2^k \geq \mathtt{n}$ (les entiers sont supposés codés en complément à 2)

```
   fun init-bit-mask (S, succ, cmax) =
1      D := 2;
2      for each x ∈ S do
3          for each y ∈ succ(x) do
4              D:=max(D, fitting-bit-mask(cmax(x, y)));
5      return D;
```

TABLE 32 – Le calcul de 2^k où k est le plus petit entier $k \geq 1$ tel que $2^k \geq \mathtt{n}$ (les entiers sont supposés codés en complément à 2)

```
   fun echelonnement (G as (S,succ,pred),s,p,cmax) =
1    for each x ∈ S do
2        for each y ∈ succ(x) do
3            f(x,y) := 0;
4    D = init-bit-mask (S, succ, cmax);
5    while D≥2 do
6      (while (pT[p]:=undef, augmenting-path(G,s,f,cmax,D»1),
7              pT[p]≠undef) do
8        update-flow-pT (f, cmax, pT, conforme, s, p);
9      D := D»1);
10   return f;
```

TABLE 33 – L'algorithme par échelonnement d'Edmonds-Karp et Dinic

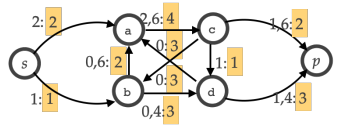

FIGURE 6 – Un flot maximum à valeurs non entières sur un réseau dont les capacités maximums sont toutes entières

```
   fun disjoint-paths(G as (S, succ, pred), s, p) =
1      let reach = reach_set ((S, succ), s) in
2      let coreach = reach_set ((S, pred), p) in
3      let fun ok(u) = u ∈ reach and v ∈ coreach in
4      let S' = [u | u ∈ S such that ok(u)] in
5      let memofun succ'(u) =
           [v | v ∈ succ(u) such that ok(v)] in
6      let memofun pred'(u) =
           [v | v ∈ pred(u) such that ok(v)] in
7      let trivial-cmax(u,v) = 1 in
8      let f = echelonnement ((S',succ',pred'),s,p,
                             trivial-cmax) in
9          return flow-val (f, G, s)
```

TABLE 34 – L'algorithme de calcul du nombre maximal de chemins disjoints de s à p ($s \leq p$) dans un graphe G

Remarque 9.47 *Supposons que les capacités maximums des arcs soient entières. Le théorème 9.46 ne dit pas que tout flot maximum est à valeur entières, juste qu'il en existe un qui est à valeurs entières. Voir la figure 6.*

Définition 9.48 *Le problème des* chemins disjoints *est le suivant :*
***Entrée** : un graphe orienté G, deux sommets distincts s et p de G, et un entier k écrit en binaire ;*
***Question** : G a-t-il au moins k chemins deux à deux disjoints, c'est-à-dire n'ayant aucun arc en commun, de s à p ?*

Lemme 9.49 *Soit G un graphe orienté, et s et p deux sommets distincts de G. Posons G' le sous-graphe induit de G dont les sommets sont les sommets qui sont à la fois accessibles depuis s et coaccessibles depuis p dans G. On définit la capacité maximum de tout arc de G' comme valant 1. Pour tout $k \in \mathbb{N}$, il y a au moins k chemins deux à deux disjoints de s à p dans G si et seulement si la valeur du flot maximum du réseau (G', s, p) est supérieure ou égale à k.*

Théorème 9.50 *Le problème des chemins disjoints est décidable en temps $O(n + m^2)$.*

Remarque 9.51 *La démonstration du théorème 9.50 commence par élaguer G et fabrique le sous-graphe G' induit par les sommets accessibles depuis s et coaccessibles depuis p. Ne pas le faire resterait correct, mais la complexité passerait à $O(m(n + m))$.*

Remarque 9.52 *Le terme en $+n$ dans la complexité énoncée au théorème 9.50 est souvent oublié. Il n'est pas nécessaire si $n \leq m^2$, ce qui est souvent le cas.*

Théorème 9.53 (Menger) *Soit G un graphe, et s et p deux sommets disjoints de G. Le nombre maximum de chemins disjoints de s à p dans G est égal au nombre minimum d'arcs de G dont la suppression rend p inaccessible depuis s.*

9.10 Couplages dans les graphes bipartis

Définition 9.54 *Un* couplage *(« matching » en anglais) dans un graphe non orienté G est un ensemble d'arêtes sans sommet commun.*

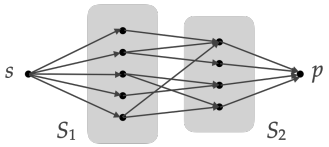

FIGURE 7 – La réduction du problème des couplages dans un graphe biparti (sommets sur fond gris) à un problème de flot maximum

Définition 9.55 *Un* graphe biparti *est un triplet (S_1, S_2, A), où S_1 et S_2 sont deux ensembles finis disjoints, et $A \subseteq S_1 \times S_2$.*

Remarque 9.56 *Un graphe biparti est souvent défini comme un graphe non orienté $G \stackrel{def}{=} (S, A)$ est* biparti *si et seulement son ensemble de sommets S s'écrit comme l'union disjointe $S_1 \uplus S_2$ de deux ensembles, de sorte que toute arête u — v relie un sommet de S_1 à un sommet de S_2. Ce n'est pas une définition équivalente à la définition 9.55, mais ceci ne causera aucun problème dans les développements qui suivent.*

Lemme 9.57 *Soit $G \stackrel{def}{=} (S_1, S_2, A)$ un graphe biparti. Notons G_0 le réseau obtenu en ajoutant à G deux sommets distincts s et p, plus :*

- *des arcs $s \to u$ pour chaque sommet $u \in S_1$, de capacité maximum 1 ;*
- *des arcs $u \to v$ pour chaque couple $(u, v) \in A$, de capacité maximum 1 ;*
- *des arcs $v \to p$ pour chaque sommet $v \in S_2$, de capacité maximum 1.*

La fonction φ qui à chaque couplage C de G associe l'unique flot f tel que $f(u, v) = 1$ pour tout arc (u, v) de C, $f(u, v) = 0$ pour tout arc $(u, v) \in A \setminus C$ est une bijection de l'ensemble des couplages de G vers l'ensemble des flots à valeurs entières de (G_0, s, p).

Un flot f est *à valeurs entières* si et seulement si $f(e)$ est entier pour tout arc e. On prendra soin de ne pas confondre cette notion avec un flot *à valeur entière*, qui est un flot f tel que $\mathrm{val}(f)$ est entier.

Théorème 9.58 *Étant donné un graphe biparti G, on peut trouver un couplage de cardinalité maximum sur G en temps $O(n + m^2)$.*

9.11 Algorithmes de préflots

Définition 9.59 *Un* préflot *sur un réseau (G, s, p) est une fonction $f \colon A \to \mathbb{R}_+$, où A est l'ensemble des arcs de G, telle que :*

1. *Pour tout sommet u de G différent de s, l'*excès $\mathrm{e}_f(u) \stackrel{def}{=} \sum_{v/v\to u} f(v, u) - \sum_{w/u\to w} f(u, w)$ *de f en u soit positif ou nul ;*
2. *pout tout arc e, $f(e) \leq c_{\max}(e)$.*

Le préflot initial sera le suivant.

Lemme 9.60 *Soit (G, s, p) un réseau, et $c_{\max}$ sa fonction de capacité maximum. La fonction f_0 qui à tout arc e d'origine s associe $c_{\max}(e)$ et à tout autre arc associe 0 est un préflot. On l'appelle le* préflot initial.

Définition 9.61 *Soit f un préflot sur un réseau (G, s, p). Un sommet u de G est* actif *pour f si et seulement si $u \neq s$, $u \neq p$ et $\mathrm{e}_f(u) > 0$.*

Définition 9.62 (Poussée) *Pour tout préflot f sur un réseau (G, s, p), pour tout arc $a \colon u \to_f v$ du graphe d'écart donc l'origine est un sommet actif u pour f, on définit la fonction $f.a$ des arcs de G vers $\mathbb{R}_+$ par :*

- *si a est représentant conforme d'un arc $u \to v$ de G, alors $(f.a)(u, v) \stackrel{def}{=} f(u, v) + \epsilon$, où $\epsilon \stackrel{def}{=} \min(\mathrm{e}_f(u), c_{\max}(u, v) - f(u, v))$, et $(f.a)(e) \stackrel{def}{=} f(e)$ pour tout autre arc e de G ;*
- *si a est représentant non conforme d'un arc $v \to u$ de G, alors $(f.a)(v, u) \stackrel{def}{=} f(v, u) - \epsilon$, où $\epsilon \stackrel{def}{=} \min(\mathrm{e}_f(u), f(v, u))$, et $(f.a)(e) \stackrel{def}{=} f(e)$ pour tout autre arc e de G.*

Lemme 9.63 *Sous les hypothèses de la définition 9.62, $f.a$ est un préflot.*

Définition 9.64 *Soit (G, s, p) un réseau. Une* transition de poussée *est de la forme $(f, \Delta) \rightsquigarrow (f.a, \Delta)$, où f est un flot sur (G, s, p), Δ est une distance estimée relative à f, a est un arc admissible du graphe d'écart, d'origine un sommet actif u.*
Elle est saturante *si et seulement si $\mathrm{e}_f(u) \geq r_f(a)$.*

Remarque 9.65 *Dans les conditions de la définition 9.64, on ne retrouve pas l'arc $u \to_{f.a} v$ dans le graphe d'écart $G_{f.a}$ si et seulement si la poussée est saturante et $r_f(a) > 0$.*

Lemme 9.66 *Soit f un préflot sur un réseau (G, s, p), et Δ une distance estimée relative à f. Pour tout arc admissible $a \colon u \to_f v$ du graphe d'écart, d'origine un sommet actif u, Δ est encore une distance estimée relative à $f.a$.*

Lemme 9.67 *Soit f un préflot sur un réseau (G, s, p), et Δ une distance estimée relative à f. Pour tout arc admissible a du graphe d'écart G_f, d'origine un sommet actif u, l'ensemble des arcs du graphe d'admissibilité $G_{f.a,\Delta}$ est inclus dans l'ensemble des arcs du graphe d'admissibilité $G_{f,\Delta}$, et strictement inclus si la poussée $(f, \Delta) \rightsquigarrow (f.a, \Delta)$ est saturante.*

Définition 9.68 *Soit f un préflot sur un réseau (G, s, p), et Δ une distance estimée relative à f. Un sommet u est* bloqué *dans la configuration (f, Δ) si et seulement s'il est actif pour f et n'est l'origine d'aucun arc admissible dans $G_{f,\Delta}$.*

Lorsqu'un sommet est bloqué, on peut le débloquer par une mise à jour de Δ, c'est-à-dire un *réétiquetage*.

Définition 9.69 (Réétiquetage) *Soit (G, s, p) un réseau. Une* transition de réétiquetage *est de la forme $(f, \Delta) \rightsquigarrow (f, u.\Delta)$, où f est un flot sur (G, s, p), Δ est une distance estimée relative à f, u est un sommet bloqué dans la configuration (f, Δ), et $u.\Delta$ est définie par :*

- *$(u.\Delta)(u) \stackrel{def}{=} \min\{\Delta(v) + 1 \mid u \to_f v\}$ si cet ensemble est non vide, n sinon ;*
- *$(u.\Delta)(v) \stackrel{def}{=} \Delta(v)$ pour tout sommet $v \neq u$.*

Lemme 9.70 *Dans les conditions de la définition 9.69, $u.\Delta$ est une distance estimée relative au préflot f ; $(u.\Delta)(v) \geq \Delta(v)$ pour tout sommet v, l'inégalité étant stricte si $u = v$.*

L'algorithme de Goldberg-Tarjan et Karzanov consiste à itérer les transitions de poussée et les transitions de réétiquetage, partant de (f_0, Δ_0), où f_0 est le préflot du lemme 9.60 et Δ_0 est la fonction calculée à la table 20, jusqu'à ce que l'on ne puisse plus opérer ni poussée ni réétiquetage.

Lemme 9.71 *Soit (G, s, p) un réseau à n sommets, et $(f_0, \Delta_0) \rightsquigarrow (f_1, \Delta_1) \rightsquigarrow \cdots \rightsquigarrow (f_k, \Delta_k)$ une suite de transitions de poussée ou de réétiquetage, où f_0 est le préflot du lemme 9.60 et Δ_0 est la fonction calculée par la procédure **`init-delta`** de la table 20, c'est-à-dire la fonction $\min(d(\_, p), n)$.*
Pour tout $i \in \{0, 1, \cdots, k\}$, on a $\Delta_i(s) = n$.

Remarque 9.72 *On peut en fait montrer que le puits p n'est accessible depuis s dans* aucun *des graphes d'écart G_{f_i} (ce qui implique $\Delta_i(s) = n$).*

Théorème 9.73 *Soit (G, s, p) un réseau à n sommets, et $(f_0, \Delta_0) \rightsquigarrow (f_1, \Delta_1) \rightsquigarrow \cdots \rightsquigarrow (f_k, \Delta_k)$ une suite de transitions de poussée ou de réétiquetage, où f_0 est le préflot du lemme 9.60 et Δ_0 est la fonction calculée par la procédure **`init-delta`** de la table 20, c'est-à-dire la fonction $\min(d(\_, p), n)$.*
Si (f_k, Δ_k) est une configuration stoppée*, c'est-à-dire si aucune transition ni de poussée ni de réétiquetage ne peut lui être appliquée, alors f_k est un flot de valeur maximum de (G, s, p).*

Lemme 9.74 *Soit f un préflot sur un réseau (G, s, p). Pour tout sommet actif u pour f, il existe un chemin $\mu \colon s \to^* u$ dans G qui est* positif*, c'est-à-dire tel que pour tout arc e de μ, $f(e) > 0$; de façon équivalente, le chemin $\mu^{op} \colon u \to_f^* s$ obtenu en prenant les représentants non conformes de chaque arc de μ est un chemin du graphe d'écart G_f.*

Corollaire 9.75 *Soit (G, s, p) un réseau à n sommets, et $(f_0, \Delta_0) \rightsquigarrow (f_1, \Delta_1) \rightsquigarrow \cdots \rightsquigarrow (f_k, \Delta_k)$ une suite de transitions de poussée ou de réétiquetage, où f_0 est le préflot du lemme 9.60 et Δ_0 est la fonction $\min(d(\_, p), n)$.*
Pour tout $i \in \{0, 1, \cdots, k-1\}$, pour tout sommet actif u pour f_i, $\Delta_i(u) \leq 2n - 1$. Par conséquent, le nombre de transitions $(f_{i-1}, \Delta_{i-1}) \rightsquigarrow (f_i, \Delta_i)$ $(1 \leq i \leq k)$ qui sont des transitions de réétiquetage est d'au plus $(n - 1)(2n - 1)$.

Proposition 9.76 *Sous les conditions du corollaire 9.75, au plus $2m(n - 1)$ des transitions sont des transitions de poussée saturantes.*

Lemme 9.77 *Soit $(f, \Delta) \rightsquigarrow (f.a, \Delta)$ une transition de poussée, où $a \colon u \to_{r_f} v$. Alors :*

1. *u est actif pour f ;*
2. *v est actif pour $f.a$, sauf si $v = s$ ou si $v = p$;*
3. *pour tout sommet $w \neq u, v$, w est actif pour f si et seulement s'il l'est pour $f.a$;*

```
   fun init-preflot (G as (S,succ,pred),s,cmax,n) =
1      let exces = new(n), f = new-flow(n) in
2      for each u ∈ S do
3              (exces[u] := 0;
4               for each v ∈ succ(u) do
5                       f(u,v) := 0);
6      for each v ∈ succ(s) do
7              (f(s,v) := cmax(s,v);
8               exces[v] := cmax(s,v));
9      return f, exces;
```

TABLE 35 – Le calcul du préflot et des excès initiaux

```
   fun init-partition (S, exces, Delta, s, p, n) =
1      dmax := -1;
2      let partition = new(2n) in
3          for i=0..2n-1 do
4              partition[i] := [];
5          for each u ∈ S do
6              if u≠s and u≠p then
7                 let i=d[u] in
8                     (partition[i] := u::partition[i];
9                      if exces(u)>0 and i>dmax
10                        then dmax := i);
11     return partition, dmax;
```

TABLE 36 – Le calcul des partitions de Dinic-Karzanov à partir d'une distance estimée

4. si la poussée est non saturante, alors u n'est pas actif pour f.a.

Proposition 9.78 *Sous les conditions du corollaire 9.75, au plus $8n^2m$ des transitions sont des transitions de poussée non saturantes.*

Karzanov a eu l'idée que l'on pouvait toujours choisir un sommet de distance estimée maximale, à condition d'utiliser la structure de donnée adéquate, en temps constant. Cette structure de données adéquates est une partition de l'ensemble des sommets par distances estimées, et remonte à Dinic (1970). De plus, ce choix va aussi diminuer le nombre de poussées non saturantes, qui forment le goulot d'étranglement de l'algorithme.

Proposition 9.79 *Soit (G, s, p) un réseau à n sommets, et $(f_0, \Delta_0) \rightsquigarrow (f_1, \Delta_1) \rightsquigarrow \cdots \rightsquigarrow (f_k, \Delta_k)$ une suite de transitions de poussée ou de réétiquetage, où f_0 est le préflot du lemme 9.60 et Δ_0 est la fonction $\min(d(\_, p), n)$. Supposons de plus que la stratégie de sélection de la transition à effectuer à chaque étape soit de sélectionner un sommet actif u de distance estimée maximale, puis d'effectuer toutes les poussées sur les arcs admissibles d'origine u dans le graphe d'écart courant.*

Alors le nombre de transitions de poussée non saturantes est majoré par $2n(n^2 + 1)$.

La réalisation de cette idée consiste à maintenir un tableau `partition`, tel que pour tout $i \in \{0, 1, \cdots, 2n-1\}$, `partition`$[i]$ contienne une liste de tous les sommets de distance estimée égale à i exactement.

Théorème 9.80 *La complexité de l'algorithme de Goldberg-Tarjan et Karzanov (table 40) est en $O(n^3)$.*

```
   fun push-flow ((u, v), Delta, f, exces, epsilon)) =
1      if Delta[u]>Delta[v]
2         then (f(u,v) := f(u,v)+epsilon;
3               exces[v] := exces[v]+epsilon);
4      else
5              (f(u,v) := f(u,v)-epsilon;
6               exces[v] := exces[v]-epsilon);
```

TABLE 37 – Une poussée de flot, avec maintien du tableau `exces`

```
   fun select (partition, dmax) =
1      d := dmax;
2      while d≥0 do
3            if partition[d]=[]
4               then d := d-1
5            else let u::rest = partition[d] in
6                     (partition[d] := rest;
7                      return u, d);
8      raise Fini;
```

TABLE 38 – La sélection de sommet par la stratégie de Karzanov

```
   fun relabel(G as (S,succ,pred),u,f,cmax,Delta,partition,n)=
1      new_d := n;
2      for each v ∈ succ(u) do
3          if Delta[u]>Delta[v] and f(u,v)<cmax(u,v)
                               and Delta[v]+1<new_d
4              then new_d := Delta[v]+1
5      for each v ∈ pred(u) do
6          if Delta[u]>Delta[v] and f(v,u)>0
                               and Delta[v]+1<new_d
7              then new_d := Delta[v]+1
8      Delta[u] := new_d;
9      partition[new_d] := u::partition[new_d];
10     return max(dmax, new_d);
```

TABLE 39 – L'opération de réétiquetage, avec calcul d'une nouvelle valeur de `dmax` et maintien de la table `partition`

```
   fun goldberg-tarjan-karzanov(G as (S,succ,pred),s,p,cmax,n)=
1    let f, exces = init-preflot(G,s,cmax,n) in
2    let Delta = init-delta(G,p,f,cmax,n) in
3    let partition = init-partition(S,exces,Delta,s,p,n) in
4    try while true do
5        let u, d = select(partition,dmax) in
6            (bloque := true;
7             for each v ∈ succ(u) do
8                 if exces[u]>0 and Delta[u]=Delta[v]+1
                                and f(u,v)<cmax(u,v)
9                    then (bloque := false;
10                         push-flow((u,v),Delta,f,exces,
                                   min(exces[u],cmax(u,v)-f(u,v))));
11           for each v ∈ pred(u) do
12               if exces[u]>0 and Delta[u]=Delta[v]+1
                                and f(v,u)>0
13                   then (bloque := false;
14                         push-flow((u,v),Delta,f,exces,
                                   min(exces[u],f(v,u));
15           if bloque
16              then dmax:=relabel(G,u,f,cmax,Delta,partition,n));
17   match Fini => return f;
```

TABLE 40 – L'algorithme de Goldberg-Tarjan et Karzanov