Fiche de SDA

Algorithme et complexité

Def. Algorithme : suite finie d'instruction non ambiguës pouvant être exécutées de façon automatique.

Def. Complexité de $\mathcal{A}$: ordre de grandeur du nombre d'opérations élémentaires effectuées pendant le déroulement de l'algorithme.

Premières structures de données

Def. Structure de données : manière d'organiser et de représenter des données ainsi que méthodes d'accès et de transformation.

Def. Liste : suite ordonnée d'éléments d'un type donné. Implémentations possibles : tableau, liste chaînée.

Def. Pile (LIFO) : liste dans laquelle l'insertion ou la suppression d'un élément s'effectue toujours à partir de la même extrémité (début). Fonctions standards : empiler, dépiler, tester si vide, renvoyer premier élément.

Def. File (FIFO) : liste dans laquelle les insertions s'effectuent d'un même côté (fin) et les suppressions à partir de l'autre extrémité (début). Opérations standards : enfiler, défiler.

Def. Un **arbre binaire** est soit vide, soit constitué d'un nœud racine et de deux sous-arbres binaires disjoints appelés sous-arbre gauche et sous-arbre droit. Il est dit **localement complet** si tout nœud interne a exactement deux fils. Il est dit **parfait** ou presque-complet si, avec h la hauteur de l'arbre, les niveaux de profondeur $p < h$ sont complètement remplis alors que le niveau de profondeur h est rempli en partant de la gauche. Il est dit **équilibré** si, pour tout nœud, les sous-arbres gauche et droit ont des hauteurs qui diffèrent au plus de 1.

Recherche et tri

Algorithme 1 : Recherche dichotomique $O(\ln(n))$

Entrées : Tableau $T[1, n]$ trié et élément x.
```
gauche ← 1, droite ← n ;
found ← false ;
tant que ¬found et gauche ⩽ droite faire
    milieu ← (gauche + droite)/2 ;
    si x < T[milieu] alors
        droite ← milieu − 1 ;
    sinon si x > T[milieu] alors
        gauche ← milieu + 1 ;
    sinon
        found ← true
```
Sorties : found

Th. *Tout algorithme de tri comparatif (fondé sur des comparaisons entre les éléments pour déterminer la permutation correspondant à l'ordre croissant des données) possède une complexité $\Omega(n \log_2(n))$ (au pire et en moyenne).*

Algorithme 2 : Tri sélection ($C \in O(n^2)$)

Entrées : Tableau $T[1, n]$.
```
pour tous les i ∈ ⟦1 ; n − 1⟧ faire
    i_min ← i, min ← T[i] ;
    pour tous les j ∈ ⟦i + 1 ; n⟧ faire
        si T[j] < min alors
            i_min ← j, min ← T[j] ;
    echanger(T, i, i_min) ;
```

Algorithme 3 : Tri insertion ($C \in O(n^2)$)

Entrées : Tableau $T[1, n]$.
```
pour tous les i ∈ ⟦2 ; n⟧ faire
    j ← i, key ← T[j] ;
    tant que j ⩾ 2 et T[j − 1] > key faire
        T[j] ← T[j − 1] ;
        j ← j − 1 ;
    T[j] ← key ;
```

Algorithme 4 : Tri rapide

Entrées : Tableau $T[1, n]$.
```
Fonction partition(g,d)
    p ← T[g], i ← g + 1, j ← d ;
    tant que i ⩽ j faire
        tant que i ⩽ j et T[i] ⩽ p faire i ← i + 1;
        tant que T[j] > p faire j ← j − 1;
        si i < j alors
            echanger(T,i,j);
            i ← i + 1, j ← j − 1;
    echanger(T,g,j);
    Sorties : j
Procedure triRapide(g,d)
    si g < d alors
        j ← partition(g,d) ;
        triRapide(g,j − 1);
        triRapide(j + 1,d);
triRapide(1,n);
```

Def. Un arbre binaire est un **arbre binaire de recherche** (ABR) s'il est vide ou égal à (F_g, c, F_d) où : F_g et F_d sont des ABRs, toute clé de F_g est inférieure à c et toute clé de F_d est supérieure à c.

Un parcours en ordre infixe d'un ABR donne donc la liste triée de ses clés.

Prop. *Le tri rapide et le tri par ABR ont une complexité au pire en $O(n^2)$ et en moyene en $O(n \ln(n))$.*

Def. Tas-max : arbre binaire parfait sur un ensemble totalement ordonné tel que l'étiquette de chaque

Algorithme 5 : Insertion dans un ABR

Entrées : Racine r de l'arbre et clé c à insérer.
Fonction inserer(r,c)
  si $r = nil$ **alors**
    Soit n un nouveau nœud ;
    n.data $\leftarrow c$, $n.g \leftarrow$ nil, $n.d \leftarrow$ nil ;
    Sorties : n
  sinon
    si $c \leqslant r.data$ **alors** $r.g \leftarrow$ inserer($r.g, c$);
    sinon $r.d \leftarrow$ inserer($r.d, c$);
    Sorties : r

nœud autre que la racine est inférieure ou égale à l'étiquette de son père.

Prop. *Le tri tas a une complexité en $O(n \ln(n))$.*

Le hachage

Def. Fonction de hachage : application $h \colon \mathbf{K} \mapsto [\![0\,;m-1]\!]$ où $\mathbf{K}$ est l'ensemble des clés possibles et $[\![0\,;m-1]\!]$ l'ensemble des indices (adresses de hachage).

Def. Collision primaire : deux clés ont le même indice. Collision secondaire : une case d'indice i est déjà occupée par une clé d'indice $j \neq i$.

Def. Hachage linéaire : en cas de collision on se déplace cycliquement vers la droite dans la table. Hachage avec chaînage interne : en cas de collision on remplit par la droite une zone de débordement à la fin du tableau, puis les positions libres du tableau, par la droite encore. Hachage avec chaînage externe : liste chaînée à partir de chaque case du tableau.

L'algorithme de Huffman

Def. Codage binaire : application injective $E \colon \Sigma \to \{0,1\}^+$ avec Σ un alphabet.

On peut l'étendre à Σ^+ par concaténation et on dit que E est uniquement décodable si E est injectif sur Σ^+.

Def. Codage préfixe : $\forall \sigma, \sigma' \in \Sigma, \sigma \neq \sigma'$, $E(\sigma)$ n'est pas un préfixe de $E(\sigma')$ (suit la règle du préfixe).

Prop. *Un codage préfixe est uniquement décodable et équivalent à la donnée d'un arbre binaire dont les feuilles sont étiquetées par les lettres de Σ.*

Objectif : associer à la suite des caractères à coder un AB tel que, toute feuille étant associé à un caractère c_i de fréquence f_i, à la hauteur l_i, $\sum_i l_i f_i$ est minimum. L'arbre est alors dit optimum.

Lem. *Dans un AB optimum tout nœud interne a deux fils et les deux plus petites occurrences se trouvent à la profondeur maximum de l'arbre.*

Lem. *Il existe un AB optimum dans lequel les deux plus petites occurrences sont dans des feuilles "frères" à la profondeur maximum.*

Lem. *Soit A un AB où les deux plus petites occurrences sont "frères" à la profondeur maximum. On note x et y leurs caractères contenus. On transforme A en A' en fusionnant ces deux feuilles en $x+y$ avec nombre d'occurrences conservé. L'arbre A est optimum si et seulement si A' l'est.*

Algorithme 6 : Montée, descente et tri dans un tas

Entrées : Tableau $T[1,n]$
Procedure montee(p)
  // p est le numéro du dernier élément.
  $i \leftarrow p$;
  cle $\leftarrow T[p]$;
  tant que $i \geqslant 2$ *et* $cle > T\left[\left\lfloor \frac{i}{2} \right\rfloor\right]$ **faire**
    $T[i] \leftarrow T\left[\left\lfloor \frac{i}{2} \right\rfloor\right]$;
    $i \leftarrow \left\lfloor \frac{i}{2} \right\rfloor$;
  $T[i] \leftarrow$ cle ;
Procedure descente(q,p)
  // q est le numéro de l'élément à descendre.
  found $\leftarrow$ false ;
  $i \leftarrow q$;
  cle $\leftarrow T[q]$;
  tant que $\neg found$ *et* $2i \leqslant p$ **faire**
    si $2i = p$ **alors** $i_{\max} \leftarrow p$;
    sinon
      si $T[2i] \geqslant T[2i+1]$ **alors** $i_{\max} \leftarrow 2i$;
      sinon $T[2i] \geqslant T[2i+1]$;
      $i_{\max} \leftarrow 2i+1$
    si $cle < T[i_{\max}]$ **alors**
      $T[i] \leftarrow T[i_{\max}]$;
      $i \leftarrow i_{\max}$;
    sinon found $\leftarrow$ true;
  $T[i] \leftarrow$ cle ;
Procedure triTas(p)
  pour p *qui varie de* 2 *à* n **faire** montee(p);
  pour p *qui varie de* n *à* 2 **faire**
    echanger(T,1,p) ;
    descente(1,p - 1) ;

Th. *La compression optimale de données sans perte est toujours possible en utilisant un code sans préfixe.*

Th. *L'algorithme de Huffman permet de construire un codage préfixe optimal.*

Graphes et arbres couvrant

On note $X_2 = \{Y \in \mathcal{P}(X) \mid \mathrm{Card}(Y) = 2\}$.

Def. Graphe (simple) non orienté : $G = (V, E)$ avec $E \subset V_2$.

Def. Graphe (simple) orienté : $G = (V, E)$ avec $E \subset V^2$.

Def. L'**ordre** d'un graphe est $|V|$ et sa **taille** est $|E|$.

Def. Graphe **complet** (ou clique) $K_n = (V, V_2)$ avec $|V| = n$.

Def. Graphe **partiel** de $G = (V, E)$: $G' = (V, E')$ avec $E' \subset E$. Soit $W \subset V$, on appelle $F = (W, E_W)$, avec $E_W = E \cap W_2$ ou $E_W = E \cap W^2$, sous-graphe de G engendré par W.

Def. Arbre : graphe acyclique connexe.

Algorithme 7 : Algorithme de Huffman

Entrées : alpabet Σ et fonction occurrence `occ`

```
soit A un tableau dynamique d'arbres ;
pour tous les x ∈ Σ faire
    A.push(nil,(x,occ (x)),nil);
A.sort() ;
// tri par occurrence décroissante des racines
n ← A.size() ;
tant que n > 1 faire
    (_, (x, _), _) ← A[n] ;
    (_, (y, _), _) ← A[n − 1] ;
    A[n − 1] ← (A[n − 1], (x + y, occ(x) + occ(y)), A[n]) ;
    A.popBack(), n ← n − 1 ;
    A.sort() ;
```

Sorties : $A[1]$

Th. *Pour un graphe G d'ordre n et de taille m, sont équivalents :*

i) G est un arbre,

ii) G est connexe et on a $m = n - 1$,

iii) G est sans cycle et on a $m = n - 1$,

iv) G est connexe et supprimer une arête le déconnecte,

v) G est acyclique et l'ajout d'une arête crée un cycle,

vi) entre deux sommets quelconques, il existe une chaîne élémentaire unique.

Problème : étant donné un graphe $G = (V, E)$ pondéré par $w\colon E \to \mathbf{R}$, trouver un arbre couvrant T (graphe partiel connexe) de poids $w(T) = \sum_{e \in T} w(e)$ minimal.

Algorithme 8 : Algorithme de Kruskal $O(m \log(m))$

Entrées : graphe connexe $G = (V, E)$ et fonction coût w

```
A ← ∅;
E_t = triCroissant(E) ;
pour tous les (a, b) ∈ E_t faire
    si A ∪ {(a, b)} est acyclique alors
        A ← A ∪ {(a, b)} ;
```

Sorties : (V, A)

```
/* on peut vérifier l'acyclicité par un tableau donnant le numéro de composante connexe ou une structure union-find */
```

Algorithme 9 : Algorithme de Prim $O(m \log(n))$

Entrées : graphe connexe $G = (V, E)$ et fonction coût w

```
S ← {s_o};
// sommet choisi arbitrairement
A ← ∅;
p ← s_o ;
pour tous les s ∈ V \ {s_0} faire d(x) ← +∞;
tant que S ≠ V faire
    pour tous les s ∈ voisins(p) ∩ S^C faire
        si w(p, s) < d(s) alors
            proche[s] ← p;
            d(s) ← w(p, s) ;
    p ← arg min_{s∈S^C} (d(s));
    S ← S ∪ {s} ;
    A ← A ∪ {(proche[p], p)} ;
```

Sorties : (S, A)

```
// on représente S^C avec un tas
```

orienté sous-jacent G' est un arbre et $\forall v \in V$, l'unique chaîne entre r et v dans G' correspond à un chemin de r vers x dans l'arborescence.

Algorithme 10 : Algorithme de Dijkstra, calcul des plus courts chemins d'un sommet à tous les autres, $O(n^2)$

Entrées : graphe $G = (V, E)$ avec coût $w\colon E \to \mathbf{R}_+$ et sommet $s \in V$.

```
S ← {s};
pour tous les v ∈ V faire
    d[v] ← w(s, v)
tant que S^C ≠ ∅ faire
    u ← arg min_{v∈S^C} (d[v]);
    S ← S ∪ {u};
    pour tous les v ∈ S^C faire
        si d[v] > d[u] + w(u, v) alors
            d[v] ← d[u] + w(u, v) ;
            pere[v] ← u ;
```

Sorties : vecteurs d (distances à s) et `pere`

Def. On appelle **tri topologique** des sommets d'un graphe orienté acyclique une numérotation des sommets telle que, pour $(u, v) \in E$ et n la fonction de numérotation, $n(u) \leqslant n(v)$.

Prop. *Un graphe est sans circuit si et seulement s'il admet une numérotation topologique.*

Problèmes de plus court chemin

Not. On note $\sim$ la relation binaire telle que $x \sim y$ s'il existe un chemin de x à y dans le graphe.

Def. Chemin élémentaire : passant par des sommets distincts. **Circuit** : chemin dont les extrémités coïncident. Circuit **absorbant** C : tel que $\sum_{e \in C} w(e) < 0$.

Def. Racine de G : $r \in V$ tel que $\forall s \in V, r \sim s$. **Arborescence** de racine r : graphe orienté tel que le graphe

Parcours de graphes

C'est sur la façon de choisir v que diffèrent les algorithmes : parcours en largeur en utilisant une file et parcours en profondeur en utilisant une pile.

Not. M : ensemble des sommets marqués par un algorithme de parcours à partir de r.

Def. Arborescence du parcours : $A = (M, \{(\mathtt{pere}(x), x) \mid x \in M \setminus \{r\}\})$ de racine r.

Algorithme 11 : Algorithme de Bellman $O(n^2)$

Entrées : graphe acyclique orienté avec coût w $G = (V, E)$

Fonction `numTopo`(V,E)

trouver x qui n'a pas de prédecesseur ;

$n_{-1} = \texttt{numTopo}(V \setminus \{x\}, E \cap (V \setminus \{x\})^2)$;

Sorties : $n \colon y \mapsto \begin{cases} 1 & \text{si } y = x \\ n_{-1}(y) + 1 & \text{sinon} \end{cases}$

$n = \texttt{numTopo}(V,E)$;

$r \leftarrow n^{-1}(1)$;

$d[r] \leftarrow 0$;

pour tous les $v \in V \setminus \{r\}$ **faire** $d[v] \leftarrow \infty$;

pour i *variant de* 2 *à* n **faire**

$x \leftarrow n^{-1}(i)$;

$\texttt{pere}[x] \leftarrow \arg\min_{y \in V | n(y) < y, (y,x) \in E} (d[y] + w(y, x))$;

$d[x] \leftarrow d[\texttt{pere}[x]] + w(\texttt{pere}[x], x)$;

Sorties : vecteurs d et `pere`

Algorithme 12 : Algorithme de Ford

Entrées : Graphe $G = (V, E)$ avec coût $w \colon E \to \mathbf{R}$ et sommet $s \in V$.

$k \leftarrow 0$;

$d^{(0)}(s) \leftarrow 0$;

pour tous les $v \in V \setminus \{s\}$ **faire** $d^{(0)}(v) \leftarrow \infty$;

répéter

`changement` $\leftarrow$ `faux` ;

$k \leftarrow k + 1$;

pour tous les $v \in V$ **faire**

$d^{(k)}(v) \leftarrow \min_{u|(u,v) \in E} (d^{(k-1)}(u) + w(u, v))$;

si $d^{(k)}(v) \neq d^{(k-1)}(v)$ **alors**

`changement` $\leftarrow$ `vrai` ;

$\texttt{pere}(v) \leftarrow \underset{u|(u,v)\in E}{\arg\min} (d^{(k-1)}(u) + w(u, v))$;

jusqu'à $k = n$ *ou* $\neg$*`changement`*;

si *`changement`* **alors**

Sorties : "cycle absorbant"

sinon

Sorties : vecteurs d et `pere`

Algorithme 13 : Algorithme de Dantzig, plus courts chemins de tout sommet à tout sommet, cas général $O(n^3)$

Entrées : graphe $G = (V, E)$ avec coût $w \colon E \to \mathbf{R}$

$D^{(1)}(1, 1) \leftarrow 0$;

pour k *allant de* 1 *à* $n - 1$ **faire**

pour $i \in [\![1 ; k]\!]$ **faire**

$D^{(k+1)}(i, k + 1) \leftarrow \min_{j \in [\![1;k]\!]} (D^{(k)}(i, j) + w(j, k + 1))$;

$D^{(k+1)}(k + 1, i) \leftarrow \min_{j \in [\![1;k]\!]} (D^{(k)}(j, i) + w(k + 1, j))$;

$D^{(k+1)}(k + 1, k + 1) \leftarrow 0$;

pour $(i, j) \in [\![1 ; k]\!]^2$ **faire**

$D^{(k+1)}(i, j) \leftarrow \min(D^{(k)}(i, j), D^{(k+1)}(i, k + 1) + D^{(k+1)}(k + 1, j))$;

Sorties : matrice $D^{(n)}$ des plus courtes distances

Algorithme 14 : Parcours abstrait d'un graphe

Entrées : $G = (V, E)$, sommet $r \in V$.

$T \leftarrow \{r\}$;

marquer r ;

tant que $T \neq \varnothing$ **faire**

choisir $v \in T$;

visiter v ;

$T \leftarrow T \setminus \{v\}$;

pour tous les v' *voisin de* v **faire**

si v' *n'est pas marqué* **alors**

$T \leftarrow T \cup \{v'\}$;

marquer v' ;

Les arcs $(\texttt{pere}(x), x)$ s'appellent **arcs arborescents**.

Prop. *M est l'ensemble des sommets x pour lesquels il existe dans G un chemin de r à x.*

Algorithme 15 : Parcours en profondeur $O(n+m)$

```
Entrées : G = {V, E}, r un sommet particulier.
(n_pre ← 1, n_post ← 1) ;
Fonction DFS(s ∈ V)
    (Numéroter s en préfixe, n_pre ← n_pre + 1) ;
    Marquer s ;
    pour tous les v voisin de s faire
        si v n'est pas marqué alors
            pere(v) ← s ;
            DFS(v) ;
    Numéroter s en suffixe, n_post ← n_post + 1 ;
DFS(r);
```

Dans un DFS en non orienté il peut être utile d'orienter les arêtes traversées (lors de la découverte des voisins) dans le sens de cette traversée.

Def. **Arête arborescente** : arête qui a été orientée en un arc arborescent. **Arc arrière** : arc obtenu en orientant une arête non arborescente. Toute arête conduit à un arc arborescent ou à un arc arrière.

Lem. *Les arcs arrières vont d'un sommet x à un des ses ancêtres autre que son père dans l'arborescence.*

Def. Soit $\bowtie$ la relation d'équivalence sur V définie par $x \bowtie y \iff (x \sim y) \wedge (y \sim x)$. Les classes d'équivalences de x pour cette relation sont les composantes fortement connexes de G. G est dit fortement connexe s'il n'admet qu'une seule telle composante.

Algorithme 16 : Algorithme de calcul des composantes fortement connexes $O(n+m)$

```
Entrées : G = {V, E} un graphe orienté.
tant que tous les sommet ne sont pas numérotés en
  post-fixe faire
    soit x un sommet non numéroté ;
    DFS(x) avec numérotation post-fixe ;
E⁻ ← {(v, u) | (u, v) ∈ E} ;
G⁻ ← {V, E⁻} ;
effacer les marquages ;
effectuer des DFS dans G⁻ successivement au
    départ du sommet de plus grand numéro
    post-fixe restant // chaque DFS donne une
    composante fortement connexe
Sorties : composantes fortement connexes
```

Def. **Sommet d'articulation** d'un graphe non orienté : sommet dont la suppression augmente le nombre de composantes connexes. C'est équivalent à dire qu'il existe $(x, y) \in V^2$ tel que toute chaîne entre x et y passe par ce point. Un graphe est dit 2-**connexe** si et seulement si il est réduit à une arête ou n'a pas de sommet d'articulation.

Th. *Soit $G = (V, E)$ un graphe connexe, non orienté, et A une arborescence DFS de G. Alors $s \in V$ est un point d'articulation de G si et seulement si*

- *s est la racine de A et possède au moins deux fils dans A,*
- *ou s n'est pas la racine de A et, pour un fils f de s dans A, il n'y a pas d'arc arrière entre les descendants de f (f compris) et les ancêtres de s.*

Def. **Numérotation basse** : $\texttt{basse}(v)$ est le minimum de : $\texttt{prefixe}(v)$, $\texttt{prefixe}(z)$ où $z \in V$ et (v, z), et $\texttt{basse}(u)$ où u est fils de v dans l'arborescence DFS.

Th. *Le point de départ de DFS est sommet d'articulation si et seulement si il a au moins deux fils dans l'arborescence. Un sommet s autre que la racine est un sommet d'articulation si et seulement s'il a au moins un fils f dans l'arborescence tel que $\texttt{basse}(f) \geqslant \texttt{prefixe}(s)$.*

Def. Soit G un graphe non orienté. On dit que G est 2-connexe s'il est connexe et n'admet aucun sommet d'articulation. On appelle composante 2-connexe tout sous-graphe 2-connexe maximal pour l'inclusion.

Algorithme 17 : Calcul des composantes 2-connexes $O(m)$

```
Entrées : G = {V, E}, r un sommet particulier.
pre ← 1 ;
pileAretes ← nil ;
Fonction parcours(x ∈ V)
    Marquer x ;
    prefixe[x] ← pre ;
    pre ← pre + 1 ;
    basse[x] ← prefixe[x] ;
    pour tous les y voisin de x faire
        si {x, y} n'a pas été traversée alors
            /* i.e. si y n'est pas marqué ou
               (y n'est pas le père de x et
               prefixe[y] ⩽ prefixe[x])          */
            pileAretes.empiler( { x,y } ) si y est
             déjà marqué alors
                basse[x] ←
                  min(basse[x], prefixe[y]) ;
            sinon
                pere[y] ← x ;
                parcours(y) ;
                si basse[y] ⩾ prefixe[x] alors
                    dépiler pileAretes jusqu'à
                      {x, y} (compris) ;
                    les arêtes dépilées sont celles
                      d'une composante 2-connexe ;
                sinon
                    basse[x] ←
                      min(basse[x], basse[y]) ;
parcours(r);
```

Flot max et coupe min

Def. **Réseau** : quadruplet (G, c, s, t) où $G = (V, E)$ est un graphe orienté, $c \colon E \to \mathbf{R}_+^*$ une fonction de capacité et s et v des sommets distincts de V appelés source et puits.

Not. Soit $S \subset V$ contenant s et non p. On dit que S sépare s de p. **Coupe** : ensemble noté $(S, S^{\mathcal{C}})$ des arcs d'origine dans S et d'extrémité dans $S^{\mathcal{C}}$. On note $E(S,T) = \{(u,v) \in E \mid u \in S, v \in T\}$ et $\delta_f(v) = \sum_{e \in E(v, V\setminus\{v\})} f(e) - \sum_{e \in E(V\setminus\{v\}, v)} f(e)$.

Def. **Flot** : application $f\colon E \to \mathbf{R}_+$ telle que

i) $\forall u \in E, f(e) \leqslant c(e)$

ii) $\forall v \in V \setminus \{s,t\}, \delta_f(v) = 0$.

Flux d'un arc u : $f(u)$.

Def. **Valeur du flot** f : $\mathrm{val}(f) = \delta_f(s) = -\delta_f(p) = f(S, S^{\mathcal{C}}) - f(S^{\mathcal{C}}, S)$ avec $(S, S^{\mathcal{C}})$ une coupe. En particulier cette dernière valeur est indépendante de la coupe choisie. Capacité d'une coupe : $c(S, S^{\mathcal{C}}) = \sum_{u \in (S, S^{\mathcal{C}})} c(u)$.

Def. Étant donné un réseau, un flot maximal est un flot qui maximise $\mathrm{val}\, f$. Problème : trouver un tel flot.

Th. *Soit f un flot et $(S, \bar{S})$ une coupe, alors*

1. $\mathrm{val}(f) \leqslant c(S, S^{\mathcal{C}})$,
2. *si* $\mathrm{val}(f) = c(S, S^{\mathcal{C}})$, $\mathrm{val}(f)$ *est maximum et* $c(S, S^{\mathcal{C}})$ *est minimum,*
3. $\mathrm{val}(f) = c(S, S^{\mathcal{C}})$ *si et seulement si* $\forall u \in (S, S^{\mathcal{C}}), f(u) = c(u)$ *et* $\forall u \in (S^{\mathcal{C}}, S), f(u) = 0$.

Not. Soit $e = (u,v)$ un arc, on note $\overleftarrow{e} = (v,u)$ (arc inverse). Pour f donné, on a le graphe résiduel (G_f, c', s, t) où $G_f = (V, E')$, $E' = \{e \in E \mid f(e) < c(e)\} \cup \{\overleftarrow{e} \mid e \in E, f(e) > 0\}$ et $\forall e \in E', c'(e) = \begin{cases} c(e) - f(e) & \text{si } e \in E \\ f(\overleftarrow{e}) & \text{si } \overleftarrow{e} \in E \end{cases}$.

Def. **Chemin f-augmentant** P : chemin de s à t dans G_f.

Soit $\gamma = \min_{e \in P} c'(e)$. On peut augmenter f le long de P par les formules $f(e) \leftarrow f(e) + \gamma$ si $e \in E$ et $f(e') \leftarrow f(e') - \gamma$ si $\overleftarrow{e} \in E$.

Th (de Ford-Fulkerson). *Sont équivalents*

i. *f est un flot maximal,*

ii. *G_f ne contient pas de chemin f-augmentant,*

iii. *$\exists (S, \bar{S})$ une coupe, telle que $|f| = c(S, \bar{S})$.*

Th. *Dans un réseau à capacités entières, il existe un flot maximum tel que tous les flux soient entiers.*

Algorithme 18 : Algorithme de Ford-Fulkerson $O(m \cdot \max_f(\mathrm{val}(f)))$

Entrées : (G, c, s, t)
Fonction `rechercheCheminAugmentant()`
 `marque`$[s] \leftarrow (\Delta, +\infty)$;
 considérer les sommets de $V \setminus \{s\}$ non marqués ;
 considérer tous les sommets non examinés ;
 tant que *p est non marqué et il y a un sommet marqué mais non examiné* **faire**
  soit x un tel sommet ;
  `// on prolonge la chaîne augmentant après` x
  soit α la valeur absolue de la deuxième marque de x;
  pour tous les *y successeur de x non marqué* **faire**
   si $c(x,y) > f(x,y)$ **alors**
    `marque`$[y] \leftarrow (x, \min(\alpha, c(x,y) - f(x,y)))$;
  pour tous les *z prédécesseur de x non marqué* **faire**
   si $f(z,x) > 0$ **alors**
    `marque`$[z] \leftarrow (x, -\min(\alpha, f(z,x)))$;
  considérer x comme examiné ;
on initialise f à 0 (flot nul) ;
`rechercheCheminAugmentant()` ;
tant que *p est marqué* **faire**
 reconstituer la chaîne augmentante C ;
 soit α la valeur absolue de la deuxième marque de p;
 pour tous les *arc $(x,y) \in C$ parcouru à l'endroit* **faire**
  $f(x,y) \leftarrow f(x,y) + \alpha$;
 pour tous les *arc $(x,y) \in C$ parcouru à l'envers* **faire**
  $f(x,y) \leftarrow f(x,y) - \alpha$;
 `rechercheCheminAugmentant()` ;
Sorties : f

Applications de la théorie des flots

Def. G est dit k-**connexe** s'il a au moins $k+1$ sommets et si la suppression d'au plus $k-1$ sommets quelconques résulte en un graphe connexe. Sommet-connectivité de G : plus grand entier k tel que G soit k-connexe.

Def. G est dit k-**arête-connexe** si la suppression d'au plus $k-1$ arêtes quelconques résulte en un graphe connexe. Arête-connectivité de G : plus grand entier k tel que G soit k-arête-connexe. La forte arc-connectivité est analogue dans un graphe orienté.

Forte arc-connectivité

Not. Soit $G = (V, e)$ orienté et $(a,b) \in V^2$ distincts. On considère le réseau R_{ab} déterminé par : ce graphe G, le sommet source a et le sommet puits b, une capacité de 1 sur tous les arcs de G.

Not. Soit P et N de à valeurs de E dans $\mathbf{N}$ tels que $P(a,b)$ est le nombre maximum de chemins de a vers b, deux à deux arc-disjoints, et $N(a,b)$ le nombre minimum d'arcs à supprimer pour qu'il n'existe plus de chemin de a vers b.

Th (Menger, "Max Flow - Min Cut"). *Si f_{ab} est maximal dans le réseau R_{ab}, alors* $\mathrm{val}(f_{ab}) = P(a,b) = N(a,b)$.

Lem (Zorn). *Supposons définie une numérotation des sommets de G de 0 à $n-1$. La forte arc-connectivité de G est $\min_{0\leqslant i\leqslant n-1} N(x_i, x_{i+1})$, en posant $x_n = x_0$.*

Puisque l'algorithme de Ford-Fulkerson est en $O(nm)$, on peut déterminer une forte arc-connectivité en $O(mn^2)$.

Détermination de l'arête-connectivité

Not. Symétrisé $G^* = (V, E^*)$ de $G = (V, E) : \forall\{u,v\} \in E$, $(u,v) \in E^*$ et $(v,u) \in E^*$. Par ailleurs, on attribue à chaque arc de G^* une capacité de 1.

Not. Soit a et b dans V, on note :

- $N(a,b)$ le nombre min d'arêtes à supprimer de G pour qu'il n'existe plus de chaîne entre a et b,
- $N^*(a,b)$ le nombre min d'arcs à supprimer de G^* pour qu'il n'existe plus de chemin de a vers b,
- $N(a,b)$ le nombre max de chaînes deux à deux arête-disjointes de G entre a et b,
- $N^*(a,b)$ le nombre max de chemins deux à deux arc-disjoints de G^* de a vers b.

Th. *Si f_{ab} de a à b est maximal dans G^*, alors* $N(a,b) = N^*(a,b) = P(a,b) = P^*(a,b) = \mathrm{val}(f_{ab})$.

Forte connectivité et connectivité

Th (Menger). *Un graphe non orienté est k-connexe si et seulement si, entre deux sommets quelconques, il existe k chaînes n'ayant en commun que leurs extrémités.*

Th (Menger). *Un graphe orienté est k-fortement connexe si et seulement si, entre deux sommets quelconques, il existe k chemins de l'un vers l'autre n'ayant en commun que leurs extrémités.*

Couplage maximum

Def. Graphe biparti : il existe une bipartition des sommets $V = V_1 \cup V_2$ telle que $E \subset V_1 \times V_2$. **Couplage** : ensemble d'arêtes non incidentes entre elles.

Problème : étant donné un graphe $G = (V,E)$ biparti, trouver un couplage de cardinal maximal.

Modélisation : on construit un réseau à partir de G en adjoignant deux sommets s et p et en plaçant des arcs de capacité 1 :

- entre s et v pour tout $v \in V_1$
- entre v et p pour tout $v \in V_2$
- entre u et v si $(u,v) \in E$

Prop. *Les flots à valeur entière sont en bijection avec les couplages de G.*

L'algorithme de Ford-Fulkerson permet alors de résoudre le problème d'affectation posé.

Complexité

Def. Une machine de Turing déterministe à une bande est formée de :

- un ruban de mémoire infini,
- une tête de lecture et d'écriture qui se déplace sur le ruban,
- un ensemble d'états avec une table de transition (comme une unité centrale munie d'un programme),
- un état particulier qui est l'état initial,
- un état final

Un calcul sur une machine de Turing se fait :

- en inscrivant l'entrée sur le ruban,
- en appliquant la table de transition,
- si on arrive dans l'état final le calcul s'arrête et est accepté.

Modèle : $(Q, \Gamma, b, \Sigma, \delta, q_0, q_s)$ avec Q l'ensemble des états, Γ l'alphabet de travail, $b \in \Sigma$ le symbole blanc, $\Sigma \subset \Gamma$ l'alphabet d'entrée, $\delta : Q \times \Gamma \to Q \times \Gamma \times \{\leftarrow, \rightarrow\}$, q_0 l'état initial et q_s l'état final.

Classe de complexité P

Def. Une fonction $f : \mathbf{N} \to \mathbf{R}$ est dite polynomiale en $g : \mathbf{N} \to \mathbf{R}$ si $\exists p \in \mathbf{R}[g], \forall n \in \mathbf{N}, f(n) \leqslant p(g(n))$.

Def. Problème de décision (ou problème de reconnaissance) : admet oui ou non pour réponse. Il est **en temps polynomial** (ou appartient à la classe P) s'il existe une machine de Turing permettant de décider ce problème et qui fonctionne en un nombre d'étapes polynomial en la taille de l'instance écrite sur le ruban d'entrée.

Not. A est polynomialement réductible à B si toute instance a de A peut être transformée en temps polynomial en une instance b de B, de longueur polynomiale en la longueur de a, telle que A est solvable sur a si et seulement si B est solvable sur b. On note $A \leqslant_P B$.

Classe de complexité NP

Def. Un problème est dans la classe NP s'il est possible de vérifier une solution en temps polynomial.

Prop. *Le problème du voyageur de commerce est dans la classe NP.*

Rem. $P \subset NP$ mais on ne sait pas dire si il y a égalité ou non.

Def. On dit que A est NP-complet si $A \in NP$ et, pour tout $B \in NP$, $B \leqslant_P A$.

Prop. *Si A est NP-complet et $C \in NP$ tel que $A \leqslant_P C$, alors C est aussi NP-complet.*

Problème de satisfiabilité

Voc.
- Variables booléennes : $x_1, \ldots, x_n$.
- Littéraux : $\lambda_i = x_i$ ou $\lambda_i = \neg x_i$.
- Clauses : $C_j = \lambda_i \vee \lambda_2 \vee \ldots \vee \lambda_k$.
- Formule : $C_1 \wedge C_2 \wedge \ldots \wedge C_z$.

Def. Problème SAT : étant donné une formule F, peut-on trouver des valeurs de $x_1, \ldots, x_n$ qui rendent F vraie ?

Th (Cook-Levin). *Le problème SAT est NP-complet.*

Def. Problème 3-SAT : étant donné une formule F dont les clauses sont toutes formées de 3 littéraux, peut-on satisfaire F ?

Th. *Le problème 3-SAT est NP-complet.*

Def. Si le problème de décision associé à une problème d'optimisation O est NP-complet, alors O est lui-même NP-difficile.

FIGURE 1 – Ada Lovelace