Des arbres dynamiques efficaces pour l'algorithme de Dinic

Laurent PIERRE, université de Nanterre

I Introduction

En 1984 Sleator et Tarjan ont inventé les arbres dynamiques, qui permettent de gérer un ensemble d'arbres comportant n sommets, en faisant les opérations suivantes en $O(\lg n)$ amorti chacune:

- `link(a→b)` ajoute l'arc `a→b`.
- `cut(a→b)` supprime l'arc `a→b`.
- `root(a)` donne la racine de l'arbre contenant `a`.

Les arcs peuvent être valués, et on peut effectuer en $O(\lg n)$ amorti des opérations sur un chemin comme

- `min_path()` qui donne le minimum des poids des arcs du chemin,
- `add_path()` qui ajoute une même valeur à chacun des arcs du chemin.

Ces opérations peuvent être utilisées dans l'algorithme de Dinic, qui cherche le flux maximal dans un réseau de transport avec n sommets et m arcs. Quand on fait ces opérations sur les arbres bêtement, en $O(n)$, l'algorithme de Dinic est en $O(n^2m)$. Avec des arbres dynamiques le temps passe en $O(nm \lg n)$.

Mais la gestion des arbres dynamique est assez lourde et la constante impliquée par $O(\lg n)$ peut être très grande. Si bien que sur des réseaux de transport réels ou tirés au hasard, l'algorithme en $O(n^2m)$ est souvent plus rapide que l'algorithme en $O(nm \lg n)$. C'est pourquoi les arbres dynamiques sont rarement utilisés. On les voit comme une belle construction théorique sans intérêt pratique. En fait on peut simplifier la gestion des arbres dynamiques de telle sorte que la constante diminue, et que l'algorithme en $O(nm \lg n)$ soit effectivement plus rapide que l'algorithme en $O(n^2m)$. C'est le sujet cet article.

Après l'introduction en section I, on rappelle en section II l'algorithme de Dinic, puis en section III le principe des arbres dynamiques et des arbre binaires de recherche évasés (splay trees). Puis la section IV décrit comment simplifier et améliorer les arbres dynamiques pour les adapter efficacement à l'algorithme de Dinic. La section V analyse le temps de calcul de cet algorithme modifié. La section VI décrit en détail toutes les procédures utilisées. La section VII décrit rapidement un programme en C qui mesure l'efficacité de cet algorithme. La section VIII montre comment appliquer ces améliorations à des arbres dynamiques plus généraux, en dehors de l'algorithme de Dinic. La section IX analyse une variante de l'algorithme qui gère différemment la saturation simultanée de plusieurs arcs. La section X améliore les majorants des temps de calcul, sans changer l'algorithme. Les références bibliographiques sont en section XI.

II Algorithme de Dinic

Un réseau de transport est un graphe orienté valué, chaque arc ayant une certaine capacité positive. Un sommet est la source, un autre est le puit. On veut faire passer dans chaque arc un certain flux, positif inférieur à la capacité. Le flux total entrant dans un sommet doit être égal au flux total sortant, pour tout sommet autre que la source ou le puit. On veut maximiser le flux total de la source vers le puit. Quand un arc `a→b` a une capacité `capa(a→b)` et fait passer un flux `flux(a→b)` avec `0≤flux(a→b)≤capa(a→b)`, cet arc a une capacité résiduelle `cap(a→b)=capa(a→b)-flux(a→b)`. Mais on peut aussi faire diminuer le flux traversant l'arc d'une quantité inférieure à `flux(a→b)`. Tout ce passe comme si on avait un arc `b→a` de capacité résiduelle `cap(b→a)=flux(a→b)`. Le graphe résiduel est l'ensemble de tous ces arcs. On peut se passer des capacités et des flux des arcs, si on ne garde que les capacités résiduelles. Par exemple quand on fait passer un flux `c` à travers l'arc `a→b` on fait `cap(a→b)-=c` et `cap(b→a)+=c` et quand on fait passer un flux `c` à travers l'arc `b→a` on fait `cap(b→a)-=c` et `cap(a→b)+=c`. Ainsi `a` et `b` jouent des rôles symétriques. Normalement, il faut enlever les arcs de capacité nulle, du graphe résiduel. Mais il est plus simple de les y laisser et de les enlever virtuellement, en les ignorant quand on les rencontre.

La méthode de Ford et Fulkerson consiste à prendre un chemin augmentant, c'est-à-dire un chemin dans le graphe résiduel allant de la source au puit, à le saturer, c'est-à-dire à faire passer dans chacun de ses arcs un même flux égal au minimum de la capacité de ses arcs, et à recommencer jusqu'à ce qu'il n'y ait plus de chemin augmentant. Alors on aura fait passer le flux maximal entre la source et le puit. Malheureusement cela peut être très lent. Avec des capacités incommensurables, il est même possible que cela boucle indéfiniment avec un flux total qui croît et converge vers une valeur autre que le maximum.

Edmonds et Karp ont montré que si on choisit à chaque étape, un chemin augmentant de longueur minimale, en nombre d'arcs, alors l'algorithme se termine et prend un temps en $O(nm^2)$. En fait la longueur

du chemin augmentant choisi à chaque étape croît au sens large au court du temps. De plus si on appelle G_i le graphe formé de l'union de tous les chemins augmentants possibles de longueur minimale l_i à l'étape i, alors ou bien $l_{i+1} > l_i$, ou bien $l_{i+1} = l_i$ et G_{i+1} est strictement inclus dans G_i car on a éliminé de G_i tous les arcs saturés dans le chemin augmentant. Au pire il n'y a qu'un arc saturé dans le chemin augmentant et G_{i+1} a seulement un arc de moins que G_i. Donc le nombre d'étapes passées avec la même valeur de l_i est inférieur à m. Puisque $0 < l_i < n$, le nombre de valeurs différentes de l_i est inférieur à n. Le nombre de chemins augmentants pris est donc majoré par nm. La recherche d'un chemin augmentant de longueur minimale peut se faire par un parcours en largeur d'abord en un temps $O(m)$. Donc le temps total de l'algorithme est $O(nm^2)$.

Il est tentant de regrouper toutes les étapes correspondant à une même valeur de l_i, et de calculer explicitement le graphe G_i, en éliminant explicitement certains arcs de G_i pour obtenir G_{i+1}. C'est exactement ce qu'a fait Dinitz (sic). L'algorithme de Dinic (sic) (reconstitué par Even après lecture de l'article de 4 pages de Dinitz dans doklady [1], c'est-à-dire le compte-rendu de l'académie des sciences de l'URSS) est:

```
répéter
  On fait un parcours en largeur d'abord du graphe résiduel, qui pour chaque sommet s
    détermine dist[s], le nombre d'arcs du chemin le plus court allant de la source à s.
  Si dist[puit]==∞ alors l'algorithme est fini.
  On fabrique le graphe G formé des arcs du graphe résiduel qui éloignent de la source,
    c'est-à-dire des arcs a→b, pour lesquels dist[b]==dist[a]+1 (et cap(a→b)>0).
  Tant que G contient un chemin de la source au puit,
    on le sature.
    On enlève de G les arcs saturés dans ce chemin.
```

La boucle interne "`Tant que G contient ...`" revient à chercher un "flux bloquant". On peut étudier le temps passé dans cette recherche d'un flux bloquant. La recherche d'un chemin de la source au puit dans G, peut se faire par un parcours en profondeur d'abord, que l'on arrête dès qu'on en a trouvé un. Mais à chaque fois que l'on revient de l'exploration d'un arc `a→b` sans avoir trouvé de chemin vers le puit, on peut éliminer cet arc de G. La boucle est faite au plus m fois, puisqu'à chaque itération G perd au moins un arc. Chaque arc disparaît au plus une fois. Donc le temps passé dans le parcours en profondeur d'abord en dehors du chemin trouvé est globalement en $O(m)$. Le temps passé dans le parcours en profondeur d'abord sur le bon chemin est proportionnel à la longueur du chemin et est donc en $O(n)$. Donc le temps total de la recherche du flux bloquant est $mO(n) + O(m) = O(nm)$. Le parcours en largeur d'abord qui remplit le tableau `dist[]` prend un temps en $O(m)$. Donc le temps total d'une itération de la boucle externe ("`répéter ...`") est $O(nm)$. Elle est effectuée moins de n fois, car à chaque itération `dist[puit]` augmente strictement. Donc l'algorithme de Dinic est en $O(n^2m)$.

En fait l'algorithme de Dinic utilise la même suite de chemins augmentants qu'Edmonds et Karp. Il y en a au plus nm. Mais le traitement de chaque chemin augmentant est en $O(m)$ pour Edmonds et Karp et en $O(n)$ pour Dinic. Il passe en $O((\lg n)^2)$ chez Galil et Naamad [2] et en $O(\lg n)$ chez Sleator et Tarjan [3].

Pour mémoire, selon [6], le vrai algorithme de Dinitz était légèrement différent. Dinitz gérait une structure de données un peu plus compliquée, qui lui permettait d'éliminer de G les arcs dès qu'ils ne permettaient plus d'atteindre le puit. Donc la recherche du chemin de la source au puit, n'était pas faite pas un parcours en profondeur d'abord, mais par un simple parcours en avant sans aucun retour en arrière.

II.2 forêt des arcs courants

Pour avoir ces temps de calcul, on suppose évidemment que l'on dispose pour chaque sommet `s`, de la liste des arcs de G sortant de `s`. Le premier arc de cette liste non encore éliminé de G sera appelé l'arc courant sortant de `s`. Quand on élimine de G l'arc courant sortant de `s`, on se contente donc de prendre pour nouvel arc courant sortant de `s`, le suivant dans la liste. Donc l'arc `s→t` est éliminé, et on passe au suivant dans la liste des arcs sortant de `s`, quand il est saturé ou quand `t` n'est pas le puit et a épuisé sa liste d'arcs sortant. L'ensemble des arcs courants forme un graphe sans cycle, car de chaque sommet il ne sort qu'un seul arc, et il augmente strictement la distance à la source. Chaque composante connexe de ce graphe est donc un arbre. Un ensemble d'arbres peut être appelé une forêt. La recherche d'un flux bloquant peut se ramener à une suite d'opérations sur des chemins dans la forêt des arcs courants :

```
Tant que la source a un arc courant :
  On cherche le plus long chemin partant de la source, en suivant les arcs courants.
  Si il atteint le puit alors
    On détermine la capacité D du chemin, qui est le minimum des capacités de ses arcs.
    On sature ce chemin en diminuant la capacité de chacun de ses arcs de D.
    On cherche dans le chemin un arc saturé.
    On coupe le chemin juste derrière cet arc.
  On élimine son dernier arc que l'on remplace éventuellement
    par un autre arc sortant du même sommet.
```

Ces opérations sur des chemins dans des arbres, peuvent être faites efficacement en utilisant des arbres dynamiques.

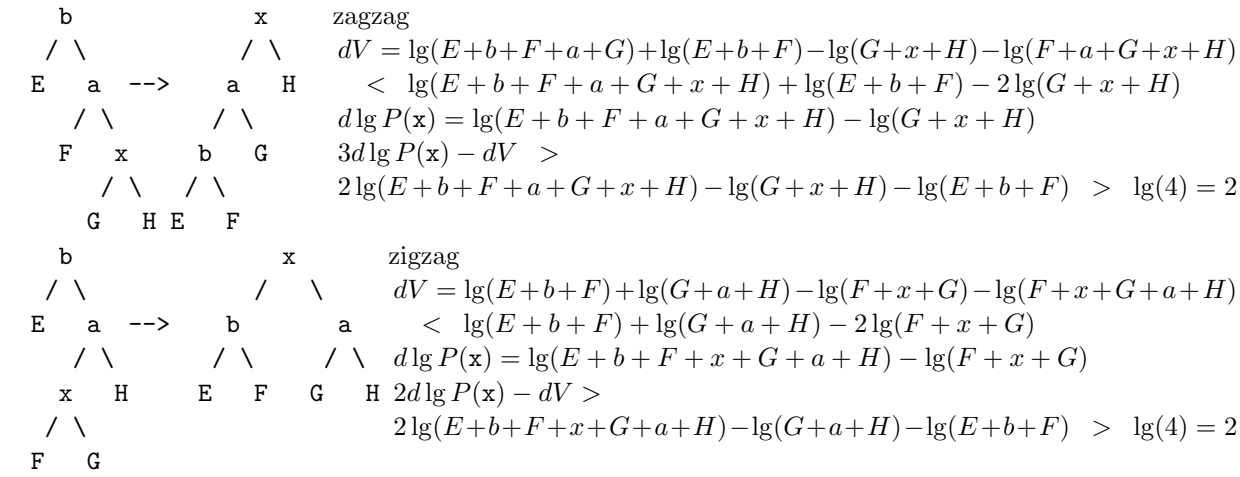
III Arbres dynamiques

On décompose les arbres en divers chemins disjoints, appelés les chemins préférés. Chaque chemin est mis dans un arbre binaire de recherche (abr). Dans la racine d'un abr, il faut rajouter une information qui indique quel arc utiliser pour se rapprocher de la racine de l'arbre (path_parent). Dans chaque noeud de l'abr on peut rajouter des informations qui permettent de gérer efficacement `min_path()` et `add_path()`. Si les abr sont équilibrés, l'algorithme de Dinic passe en $O(nm(\lg n)^2))$ selon [2]. Sleator et Tarjan on montré dans [3] que si les abr ont un équilibrage biaisé, en donnant des poids différents à chacun des sommets le temps passait en $O(nm \lg n)$. Ils ont aussi inventé dans [4] les abr évasés, qui sont autoajustés et se comportent aussi bien que les abr équilibrés de façon biaisée, sans avoir à calculer les poids des noeuds et des sous arbres. Les arbres dynamiques traités avec des "splay trees" sont par exemple décrits sous le nom de "link-cut tree" dans un cours du MIT [5] qui a inspiré l'article de wikipedia. Cette façon de faire donne un algorithme de Dinic en $O(nm \lg n)$, avec une grande constante.

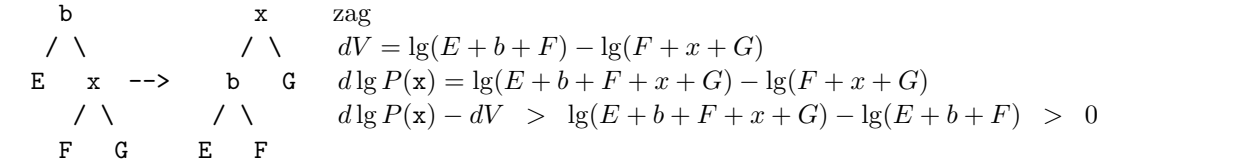
Les arbres dynamiques sont modélisés par des abr (arbres binaires de recherche) qui sont donc aussi des arbres. Pour éviter les ambiguïtés on utilisera toujours le "mot" abr pour les arbres binaires de recherche, le mot arbre étant réservé aux arbres modélisés par les abr. Par exemple, un sous-arbre d'un abr sera plutôt appelé un sous-abr.

III.2 Abr évasés (splay trees) d'après [4].

En général on utilise des abr équilibrés dans lesquels les deux fils d'un noeud ont à peu près le même poids si bien que les plus longues branches ont une longueur en $O(\lg n)$. Donc toute opération sur un tel abr prend un temps en $O(\lg n)$. Les abr évasés sont gérés autrement. Ils peuvent avoir n'importe quelle forme et avoir des branches très longues. Mais chaque opération a un coût amorti en $1 + 3 \lg n$. Cela signifie qu'on a une cagnotte V, dans laquelle on pioche quand une opération a un coût réel supérieur à $1 + 3 \lg n$, et qu'on approvisionne quand le coût réel est inférieur. Il suffit donc de s'assurer que chaque opération a un coût réel inférieur à $1 + 3 \lg n - dV$, en notant dV la variation de V. Si V reste toujours positif et qu'il n'est pas trop grand au début du calcul, après une longue suite d'opérations, le coût amorti d'une opération est assez proche de son coût réel moyen. La valeur de V dépend en fait uniquement de la forme et du contenu de l'abr. V est une fonction de l'abr, qu'on appelle aussi un potentiel. Chaque noeud `a` de l'abr a un poids $p(\mathtt{a}) > 0$. Chaque sous-abr `Y` a un poids $P(\mathtt{Y})$ qui est la somme des poids de ses noeuds. Le potentiel V d'un abr est la somme des logarithmes des poids de tous ses sous-abr. Chaque fois que l'on parcourt une branche d'un abr de la racine jusqu'à un de ses noeuds `x`, dans un sens ou dans l'autre, on réarrange l'abr en ramenant `x` à la racine, par une suite d'étapes qui mettent `x` à la place de son grand-père, tant que c'est possible. Si la profondeur initiale de `x` était impaire, il faut rajouter une dernière étape où `x` prend la place de son père. Tout cela a un coût réel, en nombre de rotations simples effectuées, égal à la profondeur initiale de `x`, mais un coût amorti de $1 + 3d \lg P(\mathtt{x})$. Si par exemple les noeuds ont tous un poids de 1, alors les sous-abr ont des poids entiers compris entre 1 et n, donc $\lg P(\mathtt{x})$ reste entre 0 et $\lg n$, donc $d \lg P(\mathtt{x}) \leq \lg n$. Pour démontrer tout cela il suffit de remarquer que la remontée de `x` à la place de son grand-père a un coût réel de 2 et un coût amorti de $3d \lg P(\mathtt{x})$ et que la remontée de `x` à la place de son père a un coût réel de 1 et un coût amorti de $1 + 3d \lg P(\mathtt{x})$. Dans les schémas suivants `a`, `b` et `x` sont des noeuds tandis que `E`, `F`, `G` et `H` sont des sous-abr. Mais dans les formules les mêmes lettres représentent les poids. Par exemple on note a et E au lieu de $p(\mathtt{a})$ et $P(\mathtt{E})$.



Pour les inégalités, on a seulement utilisé que lg est une fonction croissante et que $2\lg(a+b) \geq \lg(a)+\lg(b)+\lg(4)$ car $(a+b)^2 = 4ab+(a-b)^2 \geq 4ab$.



Dans les inégalités précédentes le coefficient de $d\lg P(\mathtt{x})$ est 3 ou 2 ou 1, mais on peut mettre partout 3 car $d\lg P(\mathtt{x}) > 0$. Le zag est traité par une rotation simple à gauche sur `b`, le zig par une rotation simple à droite, le zigzag par une rotation simple à droite sur `a` suivie d'une rotation simple à gauche sur `b`, c'est-à-dire comme un zig suivi d'un zag. Par contre le zagzag est traité par une rotation simple à droite sur `b`, le grand-père de `x`, suivie d'une autre rotation simple à droite sur `a`, le père de `x`, et non pas l'inverse comme ferait un zag suivi d'un zag. On appellera `splay(x)` la procédure qui amène le noeud `x` à la racine de l'abr qui le contient. Dans ce qui suit, on mettra des arcs dans les noeuds de l'abr.

IV Améliorations

On peut simplifier la gestion des arbres dynamiques, en l'adaptant à l'algorithme de Dinic

IV.1 Gestion statique des sommets

Dans la version la plus générale des arbres dynamiques, l'ensemble des sommets est dynamique lui aussi. On peut ajouter des sommets au graphe et en enlever. Pour cela un sommet est représenté par un pointeur sur une structure contenant toutes les données relatives au sommet. Dans l'algorithme de Dinic l'ensemble des sommets est fixé une fois pour toute. Il est donc plus simple de considérer que les n sommets sont représentés par les entiers de 0 à $n-1$. Ainsi les champs de la structure sont remplacés par des tableaux de taille n. Cela ne change pas beaucoup la description de l'algorithme : La propriété `xxx` du sommet `s` sera notée `xxx[s]` au lieu de `s->xxx`, en C, ou `s.xxx`, en java. Mais cela évite plein d'allocations dynamiques de mémoire coûteuses. En fait les tableaux seront plutôt de taille $n+1$, pour pouvoir utiliser `xxx[n]`, le sommet n étant un sommet fictif qui peut représenter l'absence de sommet.

IV.2 sens des arcs

Au lieu de prendre des chemins qui vont de la racine de l'arbre vers un de ses descendants en suivant le tableau `fils_préféré[]`, on suit des chemins qui remontent vers la racine en suivant l'arc courant sortant d'un sommet. Donc au lieu de l'arc `t`→`s` avec `s=fils_préféré[t]` on utilisera l'arc `s`→`t` avec `t=cour[s]`. On n'a plus besoin du tableau `fils_préféré[]`. Dans la suite on notera `s`→ l'arc courant sortant du sommet `s` au lieu de `s`→`cour[s]`.

IV.3 partition des arcs, plutôt que des sommets

Au lieu de faire une partition des sommets du graphe en chemins préférés, on va faire une partition des arcs courants en chemin actifs. Cela revient à peu près au même si on considère que tous les arcs d'un chemin actifs, sauf le dernier, sont des arcs préférés. Un chemin actif a→b→c→d correspond donc à un chemin préféré, c→b→a et a est le fils préféré de b, qui est le fils préféré de c, qui n'est pas le fils préféré de d. Mais dans la racine de l'abr représentant le chemin préféré c→b→a on met l'arc courant c→d comme "path-parent". Au lieu de cela l'arc courant c→d est le dernier arc du chemin actif, et dans la racine de l'abr représentant le chemin actif a→b→c→d on met le sommet d. L'abr représentant le chemin préféré a 3 noeuds contenant les sommets c, b et a. L'abr représentant le chemin actif a 3 noeuds contenant les arcs a→b, b→c et c→d, qui sont en fait représentés par leur sommets de départ a, b et c. Cela revient donc au même. Cela diffère uniquement pour un chemin allant à la racine de l'arbre. Par exemple, si la racine est r, et que le chemin préféré partant de cette racine est r→s→t→u. Ce chemin préféré est représenté par un abr ayant 4 noeuds contenant les sommets r, s, t et u, et sa racine n'a pas de "path-parent". Le chemin actif correspondant est u→t→s→r. Il est représenté par un abr à 3 noeuds contenant les arcs u→t, t→s et s→r ou les sommets u, t et s. La racine contient le sommet r. On n'a pas de tableau fils_préféré[]. Dans la racine d'un abr on n'a pas besoin d'un indicateur, pour savoir s'il existe un "path_parent", on a toujours un "bout" contenant le dernier sommet du chemin. Pour pouvoir exécuter splay(s→) chaque noeud d'un abr a un père, sauf la racine. On peut donc ranger le bout du chemin à la place du père de la racine de l'abr. Mais on y mettra plutôt bout-n. De la sorte father[s]<0 signifie que s→ est la racine d'un abr représentant un chemin actif dont le bout est father[s]+n.

IV.4 recherhe des chemins

Au lieu d'utiliser la procédure access(a) qui fait en sorte que le chemin de la racine à a soit un chemin préféré complet, on utilise une procédure extend(t→) qui prolonge le chemin actif contenant t→ jusqu'à la racine de l'arbre. Elle ne coupe donc pas le chemin avant le sommet t. C'est inutile car le seul chemin que l'on prolongera est celui qui part de la source, et aucun arc n'arrive sur la source. Cela économise plein de splay(source→) inutiles. Donc t→ peut être n'importe quel arc du chemin à prolonger, mais pour éviter de commencer par un splay() on suppose que c'est déjà la racine de son abr.

```
fonction extend(t→)
  répéter
    b=father[t]+n      // bout du chemin
    si cour[b]==n      // b est la racine de l'arbre.
      return t→
    splay(b→)
    b→ renie son fils gauche // Dans le chemin actif contenant b→ on remplace ce qui précède b
    b→ adopte t→ comme fils gauche // par le chemin actif contenant t→.
    t=b
```

La fonction extend() renvoie la racine de l'abr contenant le chemin prolongé. En général ce n'est pas la même racine qu'au début de la procédure. La valeur renvoyée permet de faire immédiatement une opération sur le chemin allongé, sans devoir faire un splay(). Autrement dit, l'argument et le résultat de extend() sont des chemins actifs, qui sont représentés plus efficacement par la racine d'un abr.

IV.5 link() et cut()

Les procédures cut(s→) et link(s,t) qui dans [5] font respectivement un et deux appels très coûteux à acces(), ne les font plus. Ici link(s,t) fabrique un nouveau chemin actif réduit à s→t en O(1). De même cut(s→) ramène s→ à la racine de son abr par un splay(s→) puis renie ses deux fils et détruit sa racine. Pour l'algorithme de Dinic, on utilisera plutôt la fonction relink(s→) qui après le splay(s→) ne renie que le fils droit de s→ et ne détruit pas la racine de l'abr mais y remplace s→t par s→t', le suivant dans la liste des arcs sortant de s, s'il existe. Sinon elle se comporte comme cut(). Ainsi relink(s→) coupe le chemin actif contenant l'arc s→ juste derrière cet arc, qui est ensuite remplacé ou éliminé. Elle retourne la nouvelle racine de l'abr. Sans compter le splay() initial, cut() et relink() sont en $O(1)$.

IV.6 mémoire statique pour les noeuds des abr

L'arc courant `s`→ est un noeud d'un abr qui a donc entre autres propriétés, un fils gauche et un fils droit. Ils seront rangés dans `lson[s]` et `rson[s]`, les tableaux `lson[]` et `rson[]` étant alloués une fois pour toute au début de l'algorithme de Dinic. Si le fils gauche de `s`→ est le sous-abr de racine `a`→ on aura `lson[s]=a`. Si le fils gauche de `s`→ est vide (`s`→ n'a pas de fils gauche) alors `lson[s]=n`. Le noeud contenant `s`→ existe dès que et tant que `cour[s]`$\neq$`n`. Juste après la création du graphe G, au début de la recherche du flux bloquant, les arcs courants existants sont tous seuls dans leur chemin actif, et donc dans leur abr : `lson[s]=n`, `rson[s]=n`, `father[s]=cour[s]-n`.

V Analyse du temps de calcul

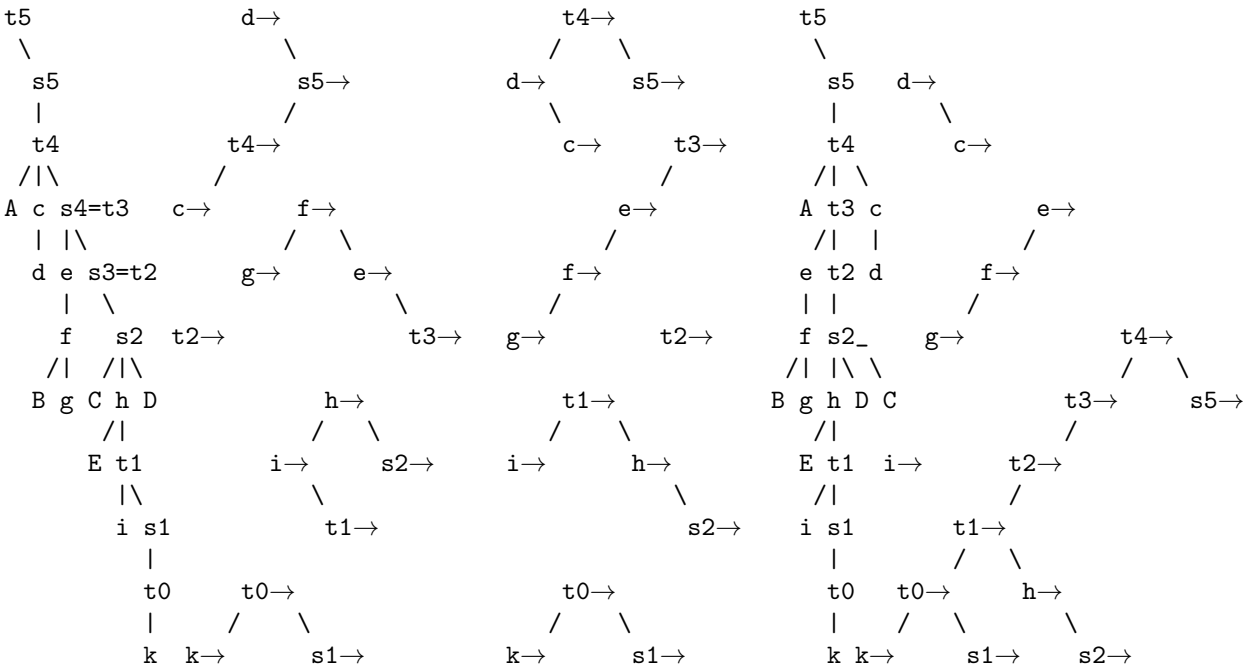
V.1 poids des sommets et des arcs et potentiel V.

Les poids des sommets et des arcs sont définis selon [3]. Le poids $P(\mathtt{s})$ d'un sommet `s` est la taille du sous-arbre de racine `s`, en nombre de sommets. Le poids d'un arc est $p(\mathtt{s}\rightarrow) = P(\mathtt{s}) - P(\mathtt{fils\_préféré(s)})$. Par convention $P(\mathtt{fils\_préféré(s)})$ est nul si `s` n'a pas de fils préféré. Autrement dit $p(\mathtt{s}\rightarrow) = P(\mathtt{s})$ si `s` est le point de départ d'un chemin actif. $P(\mathtt{s}\rightarrow)$ est la somme des poids des noeuds du sous-abr ayant pour racine `s`→. Quand un chemin actif complet se terminant par l'arc `y`→`z` est représenté par un abr ayant pour racine `s`→, alors $P(\mathtt{s}\rightarrow) = P(\mathtt{y})$. Soit V la somme des $\lg P(\mathtt{s}\rightarrow)$ sur tous les arcs courants `s`→. Le coût réel d'un appel `splay(s`→`)` compté en nombre de rotations simples effectuées, est la profondeur initiale de `s`→ dans son abr. Mais ce nombre est majoré par $1 + 3d \lg P(\mathtt{s}\rightarrow) - dV$. On peut donc dire que son coût amorti est $1 + 3d \lg P(\mathtt{s}\rightarrow)$, un plus le triple de la variation du logarithme en base 2 de $P(\mathtt{s}\rightarrow)$. Comme $P(\mathtt{s}\rightarrow)$ est toujours un entier entre 1 et n, on en déduit que le coût amorti de tout `splay()` est au plus $1 + 3 \lg n$.

V.2 coût amorti de `extend()`

Dans l'exemple suivant, les arcs préférés sont verticaux. On prolonge le chemin passant par `t0`→. Les 4 arcs qui deviennent préférés sont `s1`→`t1`, `s2`→`t2`, `s3`→`t3` et `s4`→`t4`. Les lettres majuscules `A`, `B`, `C`, `D` et `E` représentent des sous-arbres dont seuls les poids importent. Dans `extend()` on a une boucle qui contient un `splay(`$\mathtt{t}_i$`→)` puis un changement de fils gauche de $\mathtt{t}_i$→. Mais on va faire comme si tous les `splay()` avaient lieu avant tous les changements de fils gauches. Les 5 chemins actifs modifiés par `extend(t0`→`)` sont : `d`→`c`→`t4`→`s5`→`t5`, `g`→`f`→`e`→`t3`→`t4`, `t2`→`t3`, `i`→`t1`→`h`→`s2`→`t2` et `k`→`t0`→`s1`→`t1`. Les abr correspondants sont dessinés avant et après les 4 `splay(`$\mathtt{t}_i$`→)`. Après `extend(t0`→`)` il y a 4 chemins : 3 débuts d'anciens chemins `d`→`c`→`t4`, `g`→`f`→`e`→`t3` et `i`→`t1` et les fins concaténées `k`→`t0`→`s1`→`t1`→`h`→`s2`→`t2`→`t3`→`t4`→`s5`→`t5`.

Les abr correspondants sont les 3 fils gauches reniés par les $\mathtt{t}_i$→ et l'abr construit :



Les poids sont (en notant p et P les anciens poids et p' et P' les nouveaux):

s	t5	s5	t4	c	d	t3	e	f	g	t2	s2	h	t1	i	s1	t0	k	A	B	C	D	E
$P(\mathtt{s})$	38	37	36	2	1	27	7	6	1	19	18	10	5	1	3	2	1	6	4	2	5	4
$p(\mathtt{s}\rightarrow)$		1	34	1	1	20	1	5	1	19	8	5	4	1	1	1	1					
$P(\mathtt{s}\rightarrow)$		36	35	1	37	20	21	27	1	19	8	18	4	5	1	3	1					
$P'(\mathtt{s}\rightarrow)$		1	37	1	2	27	7	6	1	19	8	13	18	1	1	3	1					
$p'(\mathtt{s}\rightarrow)$		1	9	1	1	8	1	5	1	1	8	5	2	1	1	1	1					

Les $P(\mathtt{s})$ ne changent pas. Lors des `splay()`, les $p(\mathtt{s}\rightarrow)$ ne changent pas mais les $P(\mathtt{s}\rightarrow)$ changent. Le potentiel V change, mais sa variation amortit le coût des `splay()`. Par contre lors des changements d'arcs préférés, par reniement du fils gauche et adoption d'un autre, les $P(\mathtt{s}\rightarrow)$ ne changent pas, bien que les $p(\mathtt{s}\rightarrow)$ changent. Donc V est inchangé. On remarque que

$37 = P(\mathtt{s5}) = P(\mathtt{d}\rightarrow) = P'(\mathtt{t4}\rightarrow) > P(\mathtt{t4}\rightarrow) = 36 >$
$27 = P(\mathtt{s4}) = P(\mathtt{f}\rightarrow) = P'(\mathtt{t3}\rightarrow) > P(\mathtt{t3}\rightarrow) = 20 >$
$19 = P(\mathtt{s3}) = \quad P'(\mathtt{t2}\rightarrow) = P(\mathtt{t2}\rightarrow) = 19 >$
$18 = P(\mathtt{s2}) = P(\mathtt{h}\rightarrow) = P'(\mathtt{t1}\rightarrow) > P(\mathtt{t1}\rightarrow) = \ 4 >$
$3 = P(\mathtt{s1}) = P(\mathtt{t0}\rightarrow) = P'(\mathtt{t0}\rightarrow)$ et donc

$$\sum_{i=1}^{4} d\lg P(\mathtt{t}_i\rightarrow) = \lg\frac{37}{36} + \lg\frac{27}{20} + \lg\frac{19}{19} + \lg\frac{18}{4} < \lg\frac{37}{27} + \lg\frac{27}{19} + \lg\frac{19}{18} + \lg\frac{18}{3} = \lg\frac{37}{3} < \lg 37 < \lg n$$

Donc les 4 `splay()` sont payés par $4 + 3\lg(n) - dV$. C'est vrai en général :

Dans `extend(t0)` on a une boucle, effectuée k fois. Soient $\mathtt{s}_i\rightarrow\mathtt{t}_i$ le dernier arc du chemin actif allongé au début de la ième itération et $\mathtt{s}_{k+1}\rightarrow\mathtt{t}_{k+1}$ le dernier arc du chemin construit. Les sommets $\mathtt{s}_1$, $\mathtt{s}_2$, $\mathtt{s}_3$, etc apparaissent dans cet ordre dans le chemin final. Donc les $P(\mathtt{s}_i)$ croissent. Avant `splay(`$\mathtt{t}_i\rightarrow$`)` l'arc $s_i\rightarrow t_i$ n'est pas préféré donc $P(\mathtt{t}_i\rightarrow) \geq p(\mathtt{t}_i\rightarrow) \geq P(\mathtt{s}_i)$. D'autre part après le `splay()`, $P'(\mathtt{t}_i\rightarrow) = P(\mathtt{s}_{i+1})$. Donc $d\lg P(t_i\rightarrow) \leq \lg(P(s_{i+1})/P(s_i))$. Donc la somme des $d\lg P(\mathtt{t}_i\rightarrow)$ est majorée par $\lg(P(\mathtt{s}_{k+1})/P(\mathtt{s}_1)) \leq \lg n$. Donc le coût amorti de `extend(`$\mathtt{s}\rightarrow$`)` est majoré par $k + 3\lg n$ si la boucle est itérée k fois.

Si juste avant de faire `extend(`$\mathtt{t0}\rightarrow$`)` on avait fait un `splay(t0)` il aurait coûté au plus $1 + 3\lg P(\mathtt{s}_1)$. Donc le coût total des deux opérations serait majoré par $k + 1 + 3\lg n$ qui est donc le coût d'un `access()`.

V.3 arcs courants préférés, arcs courants légers, potentiel W

Il reste à démontrer que le nombre d'itérations amorti de cette boucle est $O(\lg n)$. Pour cela on va suivre [3] et classer les arcs courants en préférés ou non préférés et en lourds ou légers. L'arc courant $\mathtt{s}\rightarrow\mathtt{t}$ est l'arc préféré de `t` ssi il n'est pas le dernier d'un chemin actif. Donc le sommet `t` n'a qu'un seul arc préféré, c'est celui qui précède $\mathtt{t}\rightarrow$ dans son chemin actif. Si `t` est le point de départ d'un chemin actif, il n'a pas d'arc préféré. L'arc $\mathtt{s}\rightarrow\mathtt{t}$ est lourd ssi $2P(\mathtt{s}) \geq P(\mathtt{t})$. Puisque $P(\mathtt{t}) = 1 + \sum_{\mathtt{cour[s]}\,=\,\mathtt{t}} P(\mathtt{s})$, il y a donc au plus un arc lourd qui arrive sur `t`. On va utiliser une nouvelle fonction potentielle W égale au nombre d'arcs courants préférés légers augmenté du nombre de sommets sans arc préféré. Disons que le coût réel d'une itération est 1. Lors d'une itération, quand $\mathtt{t}\rightarrow$ renie son fils gauche, pour adopter le chemin qu'on allonge, le sommet `t` change de préféré. Si le nouveau préféré est lourd, alors l'ancien était léger ou inexistant et donc `dW=-1` donc le coût amorti est nul. Si le nouveau préféré est léger, le coût amorti est majoré par 2. Les nouveaux arcs préférés se retrouvent tous à la fin sur le chemin actif allongé. Mais un chemin actif ne peut pas contenir plus de $\lg n$ arcs légers car le poids des sommets augmente le long du chemin et double au moins à chaque arc léger. Et le poids d'un sommet est un entier entre 1 et n. Donc le nombre d'itérations ayant un coût amorti de 1 ou 2 est inférieur à $\lg n$. Le coût amorti total est majoré par $2\lg n$. On a montré que le nombre amorti d'itérations de la boucle de `extend()` est au plus $2\lg n$.

V.4 coût amorti de `cut()`

Lors d'un `cut(`$\mathtt{s}\rightarrow\mathtt{t}$`)` les poids des sommets sur le chemin de `t` à la racine `r` de l'arbre diminuent de $P(\mathtt{s})$. Les poids des autres sommets sont inchangés. Donc certains $p(\mathtt{a}\rightarrow)$ diminuent de $P(\mathtt{s})$. Les autres sont inchangés. Donc V diminue. Tous les arcs qui deviennent légers sont sur le chemin de `t` à `r`. Il n'y en pas plus de $\lg n$. De plus seul le sommet `t` peut perdre son arc préféré. Donc $dW \leq 1 + \lg n$. Donc quand $\mathtt{s}\rightarrow\mathtt{t}$ est déjà la racine d'un abr, alors `cut(`$\mathtt{s}\rightarrow\mathtt{t}$`)` a un coût réel de $O(1)$ et un coût amorti de $O(1) + \lg n$. Quand $\mathtt{s}\rightarrow\mathtt{t}$ n'est pas encore la racine d'un abr, il faut d'abord exécuter un `splay(`$\mathtt{s}\rightarrow\mathtt{t}$`)` et le coût amorti du `cut()` devient $O(1) + 4\lg n$.

V.5 coût amorti de `link()`, arcs endormis

Lors d'un `link(s→t)`, W peut varier et V augmente. On peut limiter leur variation si le chemin actif passant par `t` va jusqu'à la racine de l'arbre. Alors tous les arcs préférés dont la légèreté change sont sur ce chemin et ils deviennent lourds. Donc $dW \leq 0$. De même seul $p(\texttt{t}\rightarrow)$ augmente et si de plus `t`→ est la racine d'un abr, alors seul $P(\texttt{t}\rightarrow)$ augmente et donc $dV \leq \lg n$. Il semble donc nécessaire de faire `extend(t)` puis `splay(t→)` avant de faire `link(s→t)` pour s'assurer un coût amorti en $O(\lg n)$. Mais c'est inutile. En effet il suffit d'attendre que l'arc nouvellement créé `s→t` soit utilisé dans un `extend()`. Jusqu'à la fin de cet `extend()`, on le maintient en sommeil, en faisant comme si l'arc était `s→t'`, où `t'` est un nouveau sommet n'intervenant dans aucun autre arc pour le calcul des poids $p()$ et $P()$. Juste après un `extend()`, on réveille tous les arcs endormis situés sur le chemin allongé. Cela ne change pas V et la variation totale de W pour l'allongement d'un chemin et le réveil de ses arcs endormis ne dépasse pas $\lg n$. Cependant pendant l'allongement, chaque arc endormi traversé peut provoquer une diminution de $\lg P(\texttt{s}_i\rightarrow)$ qui coûte $3\lg n$. Puisqu'on ne calcule jamais explicitement les poids pendant l'exécution de l'algorithme, on n'a donc pas non plus besoin de savoir quels arcs sont endormis. Il n'y a donc rien de spécial à faire lors d'un `link()` et il coûte $4\lg n$ répartis en $3\lg n$ pour le réveil de l'arc plus tard et $\lg n > \lg P(\texttt{s}\rightarrow) = \lg p(\texttt{s}\rightarrow)$. Pour l'algorithme de Dinic, un arc endormi est immédiatement réveillé, et il est utilisé pendant la première itération de la boucle dans `extend()`. Il ne provoque donc pas de diminution de $\lg P(\texttt{s}_i\rightarrow)$ entre deux itérations. De plus il est fait juste àprès un `cut(s→)`, donc $P(\texttt{s}\rightarrow)$ reprend sa valeur initiale. Donc le `link()` ne coûte rien.

Chaque arc endormi rajoute virtuellement un nouveau sommet dans le graphe. On pourait croire qu'il faut remplacer partout $\lg n$ par $\lg 2n$. Mais ce n'est pas le cas. En effet $\lg n$ est un majorant de $\lg P(\texttt{s}\rightarrow)$ et $P(\texttt{s}\rightarrow)$ est maximal quand `s`→ est la racine d'un abr et alors il est égal à $P(\texttt{a})$ où `a` est l'avant dernier sommet d'un chemin actif. Donc $P(\texttt{a}) < n$. En fait on pourait plutôt remplacer partout $\lg n$ par $\lg(n-1)$.

VI Description détaillée des procédures.

Les tableaux `lson[]`, `rson[]`, `father[]`, `dcap[]`, `capmin[]`, `cour[]`, `cap[]` sont tous globaux et de taille $n+1$. Leur cases d'indice n peut être modifiée à volonté. Quand `s` n'a plus d'arc courant alors `cour[s]=n` et la case d'indice `s` des 6 autres tableaux est indéfinie. Sinon l'arc courant sortant de `s` est `s→cour[s]`. Sa capacité résiduelle est `cap[s]`. C'est un noeud d'un abr qui a pour fils gauche `lson[s]`, pour fils droit `rson[s]` et pour père `father[s]`. Le sous-abr dont `s`→ est la racine, contient des arcs dont la capacité minimum est `capmin[s]`. La capacité de tous ces arcs est virtuellement augmentée de `dcap[s]`. `cour[s]` et `cap[s]` ne sont modifiés que par `next(s,fl)` qui augmente le flux dans `s`→ de `fl`, puis avance l'arc courant dans la liste des arcs sortants de `s`. Si `a`→ est la racine d'un abr représentant un chemin actif se terminant en `b`, alors `father[a]=b-n`. Donc `a`→ est une racine ssi `father[a]<0`. n est une constante globales. Toutes les variables `s`, `t`, `x`, `y`, `z`, `a`, `b`, `c` et `D` sont locales. Les arguments des procédures et fonctions sont passés par valeur, comme en C.

```
procédure set_capmin(a)     // recalcule capmin[a] en supposant que les fils de a→ sont à jour.
  capmin[n]=∞               // pour ignorer les fils vides dans la ligne suivante
  capmin[a]=dcap[a]+min(capmin[lson[a]],capmin[rson[a]],cap[a])
procédure addcap(a,D)       // ajoute D à la capacité de chaque arc du sous-abr de racine a→
  capmin[a]+=D; dcap[a]+=D // a peut valoir n
procédure adoptel(a,b)      // a adopte b comme fils gauche.
  lson[a]=b; father[b]=a;   // b peut valoir n
procédure adopter(a,b)
  rson[a]=b; father[b]=a;
procédure zag(a)                            //     a             b
  b=rson[a]; y=lson[b]; d=dcap[b]           //    / \           / \
  adopter(a,y); adoptel(b,a)                //   x   b  -->    a   z
  capmin[b]=capmin[a]; dcap[b]=d+dcap[a] //      / \       / \
  addcap(y,d); dcap[a]=-d; set_capmin(a) //     y   z     x   y
procédure zig(a)                           // rotaton simple à droite
  b=lson[a]; y=rson[b]; d=dcap[b]          // b→ est un noeud, y un sous-abr et d un flux.
  adoptel(a,y); adopter(b,a)               // mise à jour des filiations
  capmin[b]=capmin[a]; dcap[b]=d+dcap[a]// mise à jour des capacités pour les sous-abr, de racine b→,
  addcap(y,d); dcap[a]=-d; set_capmin(a)// y, et de racine a→
```

```
procédure splay(x)
  z=x; a=father[x] // z,a,b,c sont 4 éléments consécutifs dans la suite des pères itérés à partir de x.
  // z est un fils de a, qui a déjà été remplacé par x mais father[x] et fx[a] n'ont pas encore été changés.
  s=faux // s indique qu'on a fait un nombre impair de rotations, donc un début de zigzag ou de zagzig.
  tant que a≥0 faire                          // tant que z n'était pas la racine
    b=father[a]
    si z==rson[a]                             // zag
      rson[a]=x
      si s ou b<0 ou lson[b]==a               // fin d'un zigzag ou zag final ou début d'un zagzig
        zag(a); z=a; a=b; s=non s             // zag: rotation simple, (z,a) avance d'un cran.
      sinon
        c=father[b]; zag(b); zag(a); z=b; a=c// zagzag: rotation double, (z,a) avance de deux crans.
    sinon                                     // zig
      lson[a]=x
      si s ou b<0 ou rson[b]==a               // fin d'un zagzig ou zig final ou début d'un zigzag
        zig(a); z=a; a=b; s=non s             // zig
      sinon
        c=father[b]; zig(b); zig(a); z=b; a=c// zigzig
  father[x]=a;                                // bout du chemin
fonction extend(s)  // s→ est la racine d'un abr représentant un chemin que l'on prolonge jusqu'à la
  répéter           //                                                racine de l'arbre.
    t=father[s]+n   // bout du chemin
    si cour[t]==n   // t est la racine de l'arbre. Le chemin ne peut plus être prolongé.
      return s
    splay(t)        // t→ devient la racine de son abr
    b=lson[t]
    si b!=n         // t renie son fils gauche. Ce test est inutile car
      addcap(b,dcap[t]); father[b]-=n           // cette ligne tolère b==n.
    adoptel(t,s); addcap(s,-dcap[t]); set_capmin(t) // t adopte s comme fils gauche
    s=t
fonction relink(s)
  splay(s)    // s→ devient la racine de l'abr
  b=rson[s]   // On coupe le chemin après cet arc en détachant le fils droit
  si b!=n     //  s'il existe. Mais ce test est inutile, car la ligne suivante tolère b==n.
    rson[s]=n; addcap(b,dcap[s]); father[b]=father[s]
  a=lson[s]
  addcap(a,dcap[s])      // On détache virtuellement le fils gauche
  next(s,-dcap[s])       // cour[s] change. On fait passer un flux -dcap[s] dans l'ancien arc s→
  si cour[s]!=n          // On a un nouvel arc
    dcap[s]=0            // de flux nul.
    set_capmin(s)
    father[s]=cour[s]-n // bout du chemin.
    return s
  sinon                  // Il n'y a plus d'arc sortant de s
    father[a]=s-n        // On raccourcit le chemin en enlevant son dernier arc.
    return a
procédure realize(s,D) // procédure récursive qui réalise les variations de flux rangées dans le sous-abr
  tant que cour[s]!=n  // si le sous-abr n'est pas vide                                de racine s→
    D+=dcap[s]         // D est une variation de capacité à ajouter à chacun des arcs du sous-abr.
    next(s,-D)         // Le flux de s→ augmente de -D quand sa capacité résiduelle augmente de D
    realize(rson[s],D) // traitement récursif du fils droit
    s=lson[s]          // traitement itératif du fils gauche
```

```
procédure blocking_flow
  pour tout sommet s                 // Au début
    si cour[s]!=n                    // chaque arc courant constitue un abr à lui tout seul.
      lson[s]=rson[s]=n              // s→ n'a pas de fils.
      dcap[s]=0                      // Sa capacité n'a pas encore été modifiée.
      capmin[s]=cap[s]               // min{cap[s]}=cap[s]
      father[s]=cour[s]-n            // cour[s] est le bout du chemin s→cour[s]
  s=source
  tant que s!=n                      // Tant qu'il existe un chemin partant de la source,
    s=extend(s)                      // on le prolonge au maximum.
    si father[s]==puit               // On a trouvé un chemin augmentant de la source au puit.
      dcap[s]-=capmin[s]             // On le sature.
      capmin[s]=0                    // On va descendre dans l'abr jusqu'à l'arc saturé.
      D=dcap[s]                      // D est la capacité à ajouter au sous-abr courant.
      capmin[n]=∞                    // Pour éviter de prendre un fils vide.
      tant que D!=-cap(s)            // Tant que s→ n'est pas saturé,
        si capmin[rson[s]]==-D       // si le sous-abr droit contient un arc saturé
          s=rson[s]                  // on descend à droite
        sinon                        // sinon
          s=lson[s]                  // on descend à gauche.
        D+=dcap[s]                   // On met à jour D pour le nouveau sous-abr.
    sinon                            // Le chemin est une impasse
      tant que rson[s]!=n            // On va sur son dernier arc.
        s=rson[s]
    s=relink(s)                      // On élimine un arc saturé ou qui ne mène pas au puit.
  pour tout sommet s                 // On réalise toutes les variations de flux stockées dans les abr.
    si cour[s]!=n et father[s]<0     // en commençant à la racine de chaque abr
      realize(s,0)                   // un parcours en profondeur d'abord.
```

VII programme en C

L'algorithme précédent a été écrit en C99 gnu, dans un programme qui calcule le flux maximal dans des réseaux de transport tirés au hasard. Pour chaque réseau, le calcul est fait deux fois par l'algorithme de Dinic, une fois sans arbre dynamique en $O(n^2m)$ et une fois avec des arbres dynamiques en $O(nm \lg n)$. Les temps d'exécution sont mesurés avec la fonction `clock()` et affichés. Sous unix l'algorithme est toujours plus rapide avec les arbres dynamiques, d'un facteur 4/5 environ. Sous windows la fonction `clock()` ne donne pas le temps cpu utilisé par le programme, mais le temps réel écoulé. Le résultat est nettement moins probant. Ce programme se trouve sur `http://mapage.noos.fr/laurent.pierre/graphe.c` Dans ce programme, la liste des arcs `s`→`t` sortant du sommet `s` est une liste chainée avec des tableaux, qui peut être parcourue par: `for(int k=tete[s];k!=-1;k=suite[k]) { int t=g[k]; ... }`

L'arc numéro `k` est `s`→`t` avec `t=g[k]` et `s=g[k^1]` où `k^1` est `k+1` si `k` est pair et `k-1` sinon. Sa capacité résiduelle est `flux[k]`. Son flux est donc `flux[k^1]` et sa capacité est `flux[k]+flux[k^1]`. Par exemple si `g[4]=3` et `g[5]=6` alors l'arc numéro 4 est 6→3 et l'arc numéro 5 est 3→6. Les tableaux `cour[]`, `capa[]` et `cap[]` n'existent pas. On remplace `cour[s]` par `g[tete[s]]` et `cap[s]` par `flux[tete[s]]`. Donc `cour[s]==n` devient `tete[s]==-1` et `next` devient :

```
procédure next(s,fl)
  k=tete[s]; flux[k]-=fl; flux[k^1]+=fl; tete[s]=suite[k]
```

VIII arbres dynamiques généraux

Les simplifications apportées aux arbres dynamiques dans le cas de l'algorithme de Dinic, peuvent s'appliquer en partie à des arbres dynamiques plus généraux. Il faut rajouter la procédure `cut(s→)` qui se comporte exactement comme `relink(s→)` dans le cas où `s→` serait le dernier arc sortant de `s`. Il faut aussi ajouter `link()` qui crée un nouvel arc endormi, tout seul dans son chemin actif et `access(s)` qui coupe le chemin actif contenant `s→` juste avant `s` et le prolonge jusqu'à la racine.

```
procédure link(s,t,capacité)
  cour[s]=t; pere[s]=t-n; lson[s]=rson[s]=n; cap[s]=capmin[s]=capacité; dcap[s]=0;
fonction access(s)
  si cour[s]!=n
    splay(s); b=lson[s]                                  // On amène s→ à la racine de son abr.
    addcap(b,dcap[s]); father[b]-=n; lson[s]=n; set_capmin(s) // s→ renie son fils gauche.
    return extend(s)
  sinon
    return s
```

Chacune de ces procédures et fonctions prend un temps amorti en $O(\lg n)$, même `link()` qui prend un temps réel en $O(1)$ et ne contient aucun appel à `splay`, `access` ou `extend`. Le temps de calcul des procédures, en ne comptant que les rotations simples dans les abr est:

opération	temps réel	temps amorti par V	temps amorti par $V+W$	temps amorti par $V+W'$
`splay(x)`	profondeur de `x`$< n$	$1+3d\lg P(\mathtt{x}\rightarrow) < 1+3\lg n$	$1+3\lg n$	$1+3\lg n$
`extend`	$\sum_{i=1}^{k}$ profondeur de $\mathtt{t}_i\rightarrow < n$	$k+3\lg\frac{P(\mathtt{s}_{k+1})}{P(\mathtt{s}_1)} < k+3\lg n$	$5\lg n$	$4\lg n$
`access`	n	$k+1+3\lg P(\mathtt{s}_{k+1}) < k+1+3\lg n$	$1+5\lg n$	$1+4\lg n$
`cut` ou `relink`	n	$1+3\lg n$	$2+4\lg n$	$2+3.5\lg n$
`link`	0	$4\lg n$	$4\lg n$	$4\lg n$

Les majorations des temps de calcul peuvent être améliorées, sans changer l'algorithme, si on remplace le potentiel W par un potentiel W', comme expliqué plus loin dans la section IX.

La gestion statique des sommets peut être remplacée par une gestion dynamique. Un sommet `s` ne sera plus un entier compris entre 0 et $n-1$ mais un pointeur sur une structure ayant des champs de type pointeur: `cour, lson, rson, father`, des champs numériques: cap, dcap, capmin et un champ booléen: `isroot`. Quand `s`$\rightarrow$ est la racine d'un abr on remplacera `father[s]=bout-n` par `s->father=bout; s->isroot=vrai` et quand `s`$\rightarrow$ n'est pas une racine on remplacera `father[s]=a` par `s->father=a; s->isroot=faux`. Donc `s->isroot` remplacera `father[s]<0`. Enfin le sommet fictif n sera remplacé par un sommet `vide`.

IX analyse plus fine du temps de calcul : variante du potentiel W

Le potentiel W utilisé jusqu'ici, compte les arcs préférés légers, comme le faisaient Sleator et Tarjan. Mais on peut le remplacer par un potentiel plus efficace W', qui est la somme sur tous les sommets `t` des $w'(\mathtt{t}) = \min(1, \frac{1}{2}\lg\frac{P(\mathtt{t})}{P(\mathtt{s})}) = \frac{1}{2}\lg\frac{P(\mathtt{t})}{\max(P(\mathtt{s}), P(\mathtt{t})/4)}$ où `s`$\rightarrow$`t` est l'arc préféré de `t`. Si `t` n'a pas d'arc préféré, on considèrera que $P(\mathtt{s})$ est très petit, donc $w'(\mathtt{t}) = 1$. Quand l'arc préféré de `t` change de `a`$\rightarrow$`t` en `s`$\rightarrow$`t` alors en notant $A = \max(P(\mathtt{a}), P(\mathtt{t})/4)$ et $S = \max(P(\mathtt{s}), P(\mathtt{t})/4)$ on a $dw'(t) = \frac{1}{2}\lg\frac{A}{S}$ et donc $\lg\frac{P(\mathtt{t})}{P(\mathtt{s})} - dw'(s) \geq \lg\frac{P(\mathtt{t})}{S} - dw'(s) = \lg(P(\mathtt{t})) - \frac{\lg A}{2} - \frac{\lg S}{2} > 1$ car
ou bien $P(\mathtt{a}) > P(\mathtt{t})/4$ et $P(\mathtt{s}) > P(\mathtt{t})/4$ et alors $A + S = P(\mathtt{a}) + P(\mathtt{s}) < P(\mathtt{t})$,
ou bien $P(\mathtt{a}) > P(\mathtt{t})/4$ et $P(\mathtt{s}) = P(\mathtt{t})/4$ et alors $\lg A < \lg P(\mathtt{t})$ et $\lg S = \lg(P(\mathtt{t})) - 2$,
ou bien $P(\mathtt{a}) = P(\mathtt{t})/4$ et alors $\lg A = \lg(P(\mathtt{t})) - 2$ et $\lg S < \lg P(\mathtt{t})$.
Le coût réel de ce changement est 1, mais le coût amorti par $w'(s)$ est (majoré par) $\lg\frac{P(\mathtt{t})}{P(\mathtt{s})}$.

Lors d'un `extend(`$\mathtt{s}_0$`)` qui active le chemin $\ldots\mathtt{s}_0\rightarrow\mathtt{s}_1\rightarrow\mathtt{s}_2\ldots\mathtt{s}_{k+1}$, en considérant que le coût réel du traitement de l'arc $\mathtt{s}_i\rightarrow\mathtt{s}_{i+1}$ de ce chemin est 1 s'il n'était pas encore préféré et 0 s'il l'était déjà, alors le coût amorti par $w'(\mathtt{s}_{i+1})$ est $\lg\frac{P(\mathtt{s}_{i+1})}{P(\mathtt{s}_i)}$. Le coût total amorti par W' est donc (majoré par) $\lg\frac{P(\mathtt{s}_k)}{P(\mathtt{s}_0)} \leq \lg n$. Autrement dit, le nombre de nouveaux arcs préférés amorti par W' est majoré par $\lg n$. Le majorant était $2\lg n$ quand on amortissait par W. En fait on peut considérer que W est la somme sur tous les sommets `t` des $w(\mathtt{t}) = \min(1, \lfloor 2w'(\mathtt{t})\rfloor)$ et donc W est en gros le double de W'.

Ainsi lors d'un `cut(s`$\rightarrow$`t)` les sommets `a` autres que `s` pour lesquels $w'(\mathtt{a})$ varie, sont tous sur le chemin de `t` à la racine `r` de l'arbre contenant `t`. La somme de ces $w'(\mathtt{a})$ est majorée par

$\frac{1}{2}\lg\frac{P(\mathtt{r})}{P(\mathtt{s})} \leq \frac{\lg n}{2}$. Donc $dW' \leq 1 + \frac{\lg n}{2}$ alors que $dW \leq 1 + \lg n$.

De même lors d'un `link(s→t)` si le chemin actif passant par `t` va jusqu'à la racine, alors les $w'(\mathtt{a})$ qui varient, diminuent tous, car les `a` correspondants sont tous sur ce chemin actif. Donc $dW' \leq 0$, de même que $dW \leq 0$.

X variante de l'algorithme de Dinic

Dans l'algorithme décrit jusqu'ici, quand on sature un chemin augmentant, on peut saturer plusieurs arcs de ce chemin, mais on en n'enlève qu'un seul, a priori n'importe lequel. Les autres arcs saturés restent dans le graphe et seront éventuellement enlevés quand ils apparaîtront dans un autre chemin augmentant, qui sera donc déjà saturé. Si la probabilité que cela arrive est assez grande, on va peut-être accélérer l'algorithme en arrêtant d'allonger un chemin dès qu'il est saturé. Alors tous les arcs à réveiller sur ce chemin se trouvent avant le premier arc saturé. On peut donc repousser le réveil de ces arcs juste après la suppression de l'arc saturé choisi. Ainsi l'allongement du chemin a le même coût qu'un allongement jusqu'à la racine. Après un `relink`, ou bien le chemin n'est pas saturé, et alors l'arc endormi créé est immédiatement réveillé sans surcoût de $3\lg n$. Ou bien le chemin est saturé, et alors l'arc endormi créé sera réveillé plus tard avec un surcoût de $3\lg n$, mais l'`extend` suivant le `relink` ne modifie pas le chemin et a un coût nul. Cela économise $5\lg n$ et compense donc le surcoût de $3\lg n$. Globalement on y gagne.

XI Références

[1] E. A. DINITS,An algorithm for the solution of a problem of maximal flow in a network with power estimation, Soviet Math. Dokl. I 1 (1970), 1277-1280.

[2] Z. GALIL and A. NAAMAD, An O(EVlog^2 V) algorithm for the maximal flow problem, J. Comput. System Sci. 21 (1980), 203-217.

[3] D. D. Sleator, R. E. Tarjan, A Data Structure for Dynamic Trees, Journal. Comput. Syst. Sci., 28(3):362-391, 1983.

[4] D. D. SLEATOR and R. E. TARJAN, Self-adjusting binary trees, in "Proc. Fifteenth Annual ACM Symp. on Theory of Computing," pp. 235-245, 1983.

[5] Link-Cut trees in: lecture notes in advanced data structures, Spring 2012, lecture 19. Prof. Erik Demaine, Scribes: Justin Holmgren (2012), Jing Jian (2012), Maksim Stepanenko (2012), Mashhood Ishaque (2007).

[6] Dinitz Y. (2006) Dinitz' Algorithm: The Original Version and Even's Version. In: Goldreich O., Rosenberg A.L., Selman A.L. (eds) Theoretical Computer Science. Lecture Notes in Computer Science, vol 3895. Springer, Berlin, Heidelberg pp 218-240