Cours sur les graphes : les algorithmes

F. Madelaine
C. Simon

1 L'algorithme de Bellman-Ford et ses variantes

Dans le cas classique, il s'agit de calculer les distances des plus courts chemins depuis une source s à chaque autre sommet v d'un graphe orienté valué $\overrightarrow{G}$. Les distances sont non pas en termes de nombres d'arcs d'un chemin mais, en termes de *somme des valeurs de chaque arc qui le compose*. En effet, on travaille sur un graphe orienté valué, c'est-à-dire qu'on dispose d'une fonction sur les arcs de $\overrightarrow{G}$ à valeur dans $\mathbb{R}$. En termes plus informatique, on a à notre disposition la fonction suivante :

– `length`$(u, v, \overrightarrow{G})$: renvoie la valeur de l'arc (u, v) dans le graphe orienté valué $\overrightarrow{G}$ (si l'arc existe).

Ici, nous détaillons l'algorithme dans le cas où l'entrée a un plus court chemin. Autrement dit, si le graphe n'a pas de circuit de longueur strictement négative.

Algorithme 1 : Bellman-Ford

Données
$\overrightarrow{G}$ *un graphe orienté valué sans circuit de longueur strictement négative*
s *un sommet de* $\overrightarrow{G}$
Variables locales
L *tableau des distances depuis* s `/* indexé par les sommets */`
(u, v) *un arc*
début
  initialisation
  $L[s] := 0$
  pour tous les *sommet* $v \neq s$ **faire** $L[v] := +\infty$
  tant que L *change* **faire**
    `/* relaxation de l'arc` (u, v) `*/`
    pour chaque *arc* (u, v) *de* $\overrightarrow{G}$ **faire**
      si *L(v)>L(u)+*`length(`$(u, v, \overrightarrow{G})$`)` **alors**
        L(v) :=L(u)+`length(`$(u, v, \overrightarrow{G})$`)`
      finsi
    finprch
  fintq
fin
Sorties : L, le tableau des distances des plus court chemins de s

Mémoriser les chemins.

En plus de calculer la longueur d'un plus court chemin depuis s, on peut aussi *stocker* un plus court chemin. En effet, à chaque fois qu'on relâche un arc (u, v), cela signifie que le chemin de s à v passe dorénavant par u, autrement dit le prédécesseur de v est maintenant le sommet u. On peut par exemple stocker ces prédécesseurs dans un tableaux indexé par les sommets du graphe. Voir l'algorithme 2 ci-contre pour une implémentation (les lignes correspondantes sont en bleu).

Détecter si un graphe quelconque a un circuit négatif.

On peut montrer que l'algorithme de Bellman-Ford (Algorithme 1) s'arrête après au plus $n-1$ executions de la boucle **Tant que**, dans le cas d'un graphe sans circuit de poids négatif (ici n est le nombres de sommets du graphe). La preuve se fait par récurrence sur le nombre d'itérations et l'hypothèse de récurrence est la suivante. *Au bout de x itérations de la boucle **Tant Que**, pour tout sommet v, $L[v]$ est la longueur du plus court chemin de s à v parmi ceux qui ont au plus x arcs.*

Si un graphe connexe a un circuit négatif, alors il n'y a pas de plus court chemin depuis s vers certains sommets (en particuliers ceux qui sont sur ce circuit négatif). Donc, l'algorithme 1 sur un tel graphe executerait indéfiniment dans la boucle **Tant Que**. En particulier, cette dernière s'executerait au moins n fois. On peut donc executer la boucle **Tant Que** n fois sur un graphe quelconque, et si à la dernière itération, la valeur de L change, alors on sait qu'il y a un circuit négatif.

Voir l'algorithme 2 ci-contre pour une implémentation (les lignes correspondantes sont en vert).

Algorithme 2 : Bellman-Ford (amélioré)

Données
$\overrightarrow{G}$ *un graphe orienté valué connexe*
s *un sommet de* $\overrightarrow{G}$
Variables locales
L *tableau des distances depuis* s
Pred tableau des prédécesseurs sur un plus court chemin depuis s
Circuit? Booléen vrai ssi il y a un circuit négatif
(u, v) *un arc*
début
  initialisation
  $L[s] := 0$
  pour tous les *sommet* $v \neq s$ **faire** $L[v] := +\infty$
  Circuit? := faux.
  pour $x = 0$ *à* $n-1$ **faire**
    pour chaque *arc* (u, v) *de* $\overrightarrow{G}$ **faire**
      si $L(v) > L(u) +$ `length`$((u, v, \overrightarrow{G}))$ **alors**
        L(v) :=L(u)+`length`$((u, v, \overrightarrow{G}))$
        Pred[v] :=u
      finsi
    finprch
  finpour
  pour chaque *arc* (u, v) *de* $\overrightarrow{G}$ **faire**
    si $L(v) > L(u) +$ `length`$((u, v, \overrightarrow{G}))$ **alors**
      Circuit? := vrai
    finsi
  finprch
fin
Sorties : L, *Pred*, *Circuit?*.

Variantes

Le cas dual : distance vers un sommet Au lieu de calculer des plus courts chemins depuis s vers tous les sommets, on peut calculer la distance des plus courts chemins depuis n'importe quel sommet à un sommet t (voir algorithme 3).

Algorithme 3 : Variante de Bellman-Ford (le cas dual)

Données
$\overrightarrow{G}$ *un graphe orienté valué sans circuit de longueur strictement négative*
t *un sommet de* $\overrightarrow{G}$
Variables locales
L *tableau des distances vers un sommet à* t
Succ tableau des successeurs sur un plus court chemin à t
(u, v) *un arc*
début
  initialisation
  $L[t] := 0$
  pour tous les *sommet* $v \neq t$ **faire** $L[v] := +\infty$
  tant que L *change* **faire**
    pour chaque *arc* (u, v) *de* $\overrightarrow{G}$ **faire**
      si $L(u) > L(v) + \texttt{length}((u, v, \overrightarrow{G}))$ **alors**
        L(u) :=L(v)+`length`$((u, v, \overrightarrow{G}))$
        Succ[u] :=v
      finsi
    finprch
  fintq
fin
Sorties : L, succ

Plus long chemin vers un sommet Au lieu de calculer des plus courts chemins depuis s vers tous les sommets, on peut calculer la distance des plus longs chemins. Bien evidemment, l'algorithme a maintenant des limitations différentes : il n'a de sens que dans un graphe sans circuit strictement positif (voir algorithme 4).

Algorithme 4 : Autre variante de Bellman-Ford (plus long chemin)

Données
$\overrightarrow{G}$ *un graphe orienté valué sans circuit de longueur strictement positive*
s *un sommet de* $\overrightarrow{G}$
Variables locales
L *tableau des distances depuis un sommet* s
Pred tableau des successeurs sur un plus court chemin à t
(u, v) *un arc*
début
  initialisation
  $L[s] := 0$
  pour tous les *sommet* $v \neq s$ **faire** $L[v] := -\infty$
  tant que L *change* **faire**
    pour chaque *arc* (u, v) *de* $\overrightarrow{G}$ **faire**
      si $L(v) < L(u) + \texttt{length}((u, v, \overrightarrow{G}))$ **alors**
        L(v) :=L(u)+`length`$((u, v, \overrightarrow{G}))$
        Pred[v] :=u
      finsi
    finprch
  fintq
fin
Sorties : L, Pred

2 Parcours

Dans un graphe non-orienté G à partir d'un sommet s, on découvre les sommets de la *composante connexe* de s dans G (c'est-à-dire l'ensemble des sommets accessibles par un chemin depuis s dans G). Lors du calcul, on utilise un ensemble Z pour stocker les sommets déjà visités, et une variable F pour stocker les sommets de la frontière, c'est-à-dire ceux ayant au moins un voisin non visité. Selon la structure de donnée qu'on utilise pour F on obtient des parcours différents.

Structures de données

L'ensemble

On dispose des fonctions suivantes sur un ensemble S.
- `choose`(S) : renvoie un sommet x de S (si S n'est pas vide).
- `empty`?(S) : renvoie vrai ssi S est l'ensemble vide.
- `add`(x, S) : ajoute x à l'ensemble S.
- `remove`(x, S) : enlève x de l'ensemble S (si x appartient à S).

La file

On dispose des fonctions suivantes sur une file Q.
- `checkFront`(Q) : renvoie *le premier* sommet de F (si Q n'est pas vide).
- `empty`?(Q) : renvoie vrai ssi Q est la file vide.
- `pushBack`(x, Q) : ajoute x *à l'arrière* de la file Q.
- `popFront`(Q) : enlève le premier sommet de la file Q (si celle-ci n'est pas vide).

La pile

On dispose des fonctions suivantes sur une pile P.
- `checkFront`(P) : renvoie le premier sommet de P (si P n'est pas vide).
- `empty`?(P) : renvoie vrai ssi P est la pile vide.
- `pushFront`(x, P) : ajoute x *au début* de la pile P.
- `popFront`(P) : enlève le premier sommet de la pile P (si celle-ci n'est pas vide).

Algorithmes de parcours

Algorithme 5 : Parcours général

```
Données
G un graphe
s un sommet de G
Variables locales
Z un ensemble                                   /* zone connue */
F un ensemble                                   /* la frontière */
u, v deux sommets
début
    initialisation
    add(s,Z)
    add(s,F)
    répéter
        v :=choose (F)
        si il existe u ∉ Z adjacent à v alors
            add(u,F)                            /* première visite de u */
            add(u,Z)
        sinon
            remove(v,F)                         /* dernière visite de v */
        finsi
    jusqu'à empty ?(F)
fin
```

Algorithme 6 : Parcours en largeur

```
Données
G un graphe
s un sommet de G
Variables locales
Z un ensemble                                   /* zone connue */
F une file                                      /* la frontière */
u, v deux sommets
début
    initialisation
    add(s,Z)
    pushBack(s,F)
    répéter
        v :=checkFront (F)
        si il existe u ∉ Z adjacent à v alors
            add(u,Z)                            /* première visite de u */
            pushBack(u,F)
        sinon
            popFront(F)                         /* dernière visite de v */
        finsi
    jusqu'à empty ?(F)
fin
```

Algorithme 7 : Parcours en profondeur

```
Données
G un graphe
s un sommet de G
Variables locales
Z un ensemble                                   /* zone connue */
F une pile                                      /* la frontière */
u, v deux sommets
début
    initialisation
    add(s,Z)
    pushFront(s,F)
    répéter
        v :=checkFront (F)
        si il existe u ∉ Z adjacent à v alors
            add(u,Z)                            /* première visite de u */
            pushFront(u,F)
        sinon
            popFront(F)                         /* dernière visite de v */
        finsi
    jusqu'à empty ?(F)
fin
```

Application du parcours en largeur

Distance à la source On peut calculer la distance depuis la source s à chaque sommet v du graphe (si G est connexe). En effet, on peut facilement observer que les niveaux de l'arbre (= distance à la racine) correspondent à la distance à la source.

Algorithme 8 : Parcours en largeur + distance à la source

Données
G un graphe connexe et s un sommet de G
Variables locales
Z un ensemble, F une file, u, v deux sommets
L tableau des distances à la source `/* indexés par les sommets */`
début
  initialisation
  `add`(s,Z)
  `pushBack`(s,F)
  $L[s] := 0$
  répéter
    $v :=$`checkFront` (F)
    si *il existe $u \notin Z$ adjacent à v* **alors**
      `add`(u,Z) `/* première visite de` u `*/`
      `pushBack`(u,F)
      $L[u] := L[v] + 1$
    sinon
      `popFront`(F) `/* dernière visite de` v `*/`
    finsi
  jusqu'à `empty?`(F)
fin
Sorties : L, tableau des distances à la source s

Applications du parcours en profondeur

On s'intéresse au moment où on visite un sommet pour la première fois et celui où on le visite pour la dernière fois. On a deux tableaux *Début* et *Fin* dans lesquels on stocke les temps correspondant. On mesure le temps en termes du nombre x d'executions de la boucle **répéter**.

Algorithme 9 : Parcours en profondeur + début et fin

Données
G un graphe connexe et s un sommet de G
Variables locales
Z un ensemble, F une pile et u, v deux sommets
Debut et Fin deux tableaux `/* indexés par les sommets */`
début
  initialisation
  $x=0$
  `add`(s,Z)
  `pushFront`(s,F)
  $Début[s] := x$
  répéter
    $x := x + 1$
    $v :=$`checkFront` (F)
    si *il existe $u \notin Z$ adjacent à v* **alors**
      `add`(u,Z) `/* première visite de` u `*/`
      `pushFront`(u,F)
      $Début[s] := x$
    sinon
      `popFront`(F) `/* dernière visite de` v `*/`
      $Fin[s] := x$
    finsi
  jusqu'à `empty?`(F)
fin

Le Tri topologique. On dispose d'un graphe orienté sans cycle. Un tel graphe représente typiquement *un problème d'ordonnancement de tâches* : un sommet correspond à une tâche et un arc (u, v) indique la contrainte de précédence "*la tâche u doit être terminée avant que la tâche v ne puisse commencer*". Le problème du tri topologique consiste à ordonner les sommets du graphe de telle sorte que si il y a un arc (u, v) alors u vient avant v dans l'ordre calculé.

Bien que le graphe soit orienté, on peut parfaitement y appliquer l'algorithme de parcours en profondeur. On peut observer que la dernière visite d'un sommet a forcément lieu *avant* la dernière visite d'un sommet qui le précède. Ainsi, on peut voir les valeurs calculées dans le tableau *Fin* comme des valeurs de priorité : plus la valeur est haute, plus la tâche est importante. Autrement dit, l'ordre inverse de celui donné par *Fin* est un ordre topologique.

Algorithme 10 : Parcours en profondeur + Tri Topologique

Données
$\overrightarrow{G}$ *un graphe orienté sans cycle*
s *un sommet de* G
Variables locales
Z *un ensemble,* F *une pile et* u, v *deux sommets*
Priorité un tableau `/* indexé par les sommets */`
début
  initialisation
  x=0
  `add(`s`,`Z`)`
  `pushFront(`s`,`F`)`
  répéter
    $x := x + 1$
    v :=`checkFront` (F)
    si *il existe* $u \notin Z$ *successeur de* v **alors**
      `add(`u`,`Z`)` `/* première visite de` u `*/`
      `pushFront(`u`,`F`)`
    sinon
      `popFront(`F`)` `/* dernière visite de` v `*/`
      Priorité$[s] := x$
    finsi
  jusqu'à `empty?(`F`)`
fin
Sorties : *Priorité*