Fin des Graphes et Analyse lexicographique

Eric Goubault

CEA/Saclay

Eric.Goubault@cea.fr

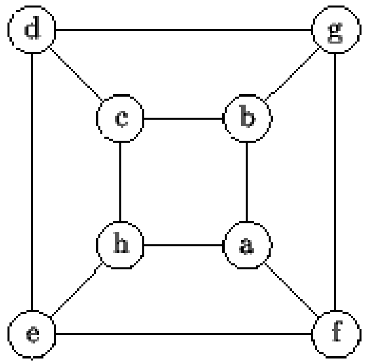
http://www.di.ens.fr/˜goubault

Points d'articulation

- Soit un graphe connexe et non-orienté (X, A), sans arcs d'un sommet à lui même.
- Un point d'articulation a est un sommet tel qu'il existe deux autres sommets distincts s et t connectés et que a appartient à tous les chemins de s à t.
- Un graphe qui ne contient pas de point d'articulation est biconnexe.
- Deux arcs sont équivalents quand il sont égaux ou si existe un cycle les contenant tous deux.
- Un graphe est biconnexe quand tous ses arcs sont en relation.

Les graphes suivants sont-ils biconnexes ?

Les graphes suivants sont-ils biconnexes ?



Question

Montrer l’équivalence des deux définitions de graphe biconnexe.

Programme de Tarjan

```
// renvoie le numéro du point d'attache
private static int doFindArticulation (int s, int pere) {
int lowpt ; boolean found = false ;
++numOrder ; lowpt = num[s] = numOrder ;
for (List p = succ[s] ; p != null ; p = p.next) {
  int n = p.val ;
  if (num[n] == 0) {
    int lowN = doFindArticulation(n, s) ;
    lowpt = Math.min(lowN,lowpt) ;
    if (lowN >= num[s]) {
      System.out.println("Articulation : " + s) ; }
  } else if (num[n] < num[s] && n != pere) {
      lowpt = Math.min(num[n],lowpt) ; } }
  return lowpt ; }
```

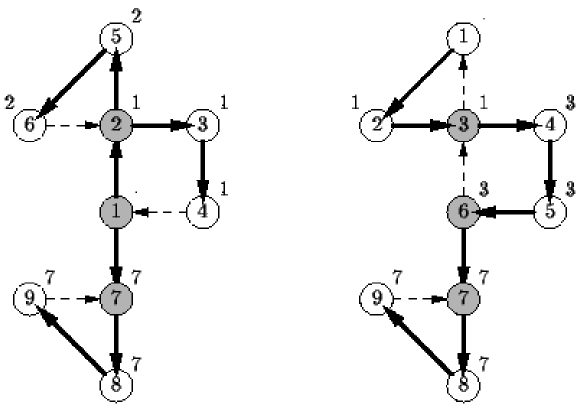

Programme de Tarjan

```
static void findArticulations(int s) {
numOrder = 0 ; num = new int[succ.length] ;
int nfils = 0 ;
numOrder++ ; num[s] = numOrder ;
for (List p = succ[s] ; p != null ; p = p.next) {
  int n = p.val ;
  if (num[n] == 0) {
    doFindArticulation(n, s) ;
    nfils++ ; } }
if (nfils > 1)
  System.out.println("La racine " + s + "
                est un point d'articulation") ; }
```

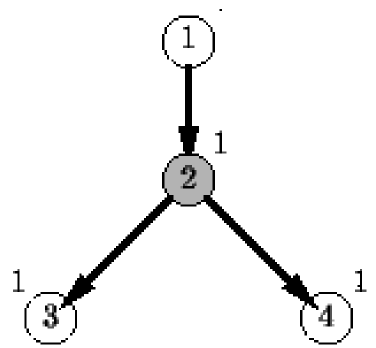

Questions

- Appliquer le programme au graphe de l’exercice précédent.
- Un sommet peut-il être affiché deux fois ? Si oui donner un exemple.
- Justifier la correction du programme. Le comparer à celui du poly.

Solution...



Solution...



Composantes biconnexes

- Les composantes biconnexes sont intuitivement les composantes connexes du graphe une fois supprimés les point d'articulation.
- Plus précisément, la relation d'équivalence sur les arcs définit les classes d'équivalence A_1, A_2, ..., A_k des arcs du graphe (X, A).
- On peut alors définir les ensembles X_1, X_2, ..., X_k de sommets participant respectivement aux arcs de A_1, A_2, ..., A_k.

Il est évident que les graphes (X_i, A_i) sont biconnectés (et en un sens maximaux). Il semble logique que les points d'articulations soient les éventuels points communs des X_i (on ne tentera aucune preuve).

Questions

- Trouver les composantes biconnexes des graphes non-biconnexes déjà vus.
- Concevoir, en utilisant une pile d'arcs, un programme qui affiche les (arcs des) composantes biconnexes, à raison d'une ligne par composante biconnexe.

Calcul naïf des composantes fortement connexes

- Rappeler la définition des composantes fortement connexes dans un graphe orienté.
- Caractériser la composante fortement connexe d'un sommet à l'aide du graphe des prédécesseurs (ou graphe inverse).
- Concevoir un algorithme simple pour afficher les composantes fortement connexes. Indication : on utilisera le graphe des prédécesseurs, et deux méthodes d'exploration partageant un tableau de marques.
- Trouver un exemple d'exécution qui montre que l'algorithme est au moins quadratique (en le nombre de sommet).
- Programmer.

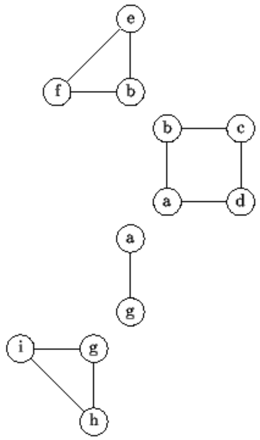
Mots, langages etc.

Notions de:

- Alphabet
- Mots (concaténation de lettres dans l'alphabet)
- Langages (ensembles de mots)
- Automates/langages réguliers, grammaires etc. (description de langages)

Solution...

Solution...



Expressions régulières

- A partir d'un alphabet A
- Plus petit langage formé du mot vide, de l'alphabet A (vu comme mots) et stable par les opérations:
  - produit: $L.L' = \{w.w' \mid w \in L, w' \in L'\}$
  - somme: $L + L' = L \cup L'$
  - étoile: $L^* = \sum_{i \in N} L^i$ où $L^0 = \{\epsilon\}$ (le mot vide), $L^{n+1} = L.L^n$

Notations

- $(a+b)^*c = \{c, ac, bc, aac, abc, bac, bbc, \ldots\}$
- utilisation sous Unix (filtres pour fichiers): `ls *.java` etc.
- notion d'algèbre de Kleene
- permet en général de définir la notion de lexème (séparant le flot continu des lettres)

Application typique

- `lettre=a|b|...|z|A|B|...|Z`
- `chiffre=0|1|...|9`
- `identification=lettre(lettre|chiffre)*` (autre notation pour + et *)
- on oublie tout ce qui est tabulation, espaces etc.

Exercice

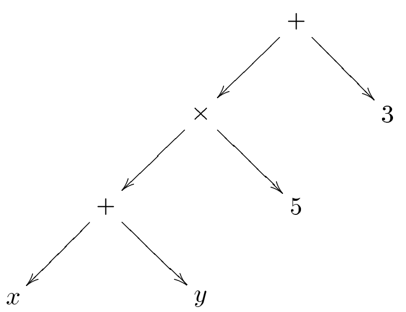
En utilisant une pile, simuler une petite calculatrice HP (notation polonaise inverse), pour évaluer des expressions arithmétiques simples

Arbre de syntaxe abstraite - ex. expressions arithmétiques

Les termes sont transformés en arbres, dont les noeuds sont les opérations ($+$, $\times$ etc.), et dont les feuilles sont les constantes et les variables

Exemple

$(x + y) * 5 + 3$ est représenté par:



Grammaires

Composées de :

- A: alphabet terminal (tous les mots construits par la grammaires sont constitués de lettres de A)
- V_N: alphabet non-terminal; $S \in V_N$ joue un rôle particulier "axiome"
- des règles de la forme: $S \to u$ où $S \in V_N$ et u mot de $A \cup V_N$

Exemple

- $A = \{a, b\}$
- $V_N = \{S, T, U\}$, l'axiome est S
- règles (de "production"):
  - $S \to aTbS$
  - $S \to bUaS$
  - $S \to \epsilon$
  - $T \to aTbT$
  - $T \to \epsilon$
  - $U \to bUaU$
  - $U \to \epsilon$

Utilisation des rg̀les de production

On forme des “dérivations”:

- On part de S, à partir de $S \rightarrow u$
- Si u contient des non-terminaux, on écrit $u = u_1 T u_2$
- On choisit ensuite $T \rightarrow v$ et on obtient $u_1 v u_2$
- etc.

Exemple

- Par $S \rightarrow aTbS$ puis $T \rightarrow \epsilon$:

$$S \rightarrow aTbS \rightarrow abS$$

- Puis $S \rightarrow bUaS$:

$$S \rightarrow aTbS \rightarrow abS \rightarrow abbUaS \rightarrow abbbUaUaS \ldots$$

En fait, ici, le langage engendré par cette grammaire est constitué des mots avec autant de a que de b

Exercices

- Donner une grammaire pour les expressions régulières

Analyse syntaxique

- But: déterminer si un mot f appartient au langage d’une grammaire
- Pour ce faire, trouver une dérivation amenant à f
- S’il existe plusieurs dérivation pour le même mot, la grammaire est ambigüe

Exemple de grammaire ambigüe

- $A = \{+, \times, (,), id, nombre\}$, $V_N = \{E\}$, l'axiome est E, et les règles sont:
  - $E \to E + E$
  - $E \to E \times E$
  - $E \to id$
  - $E \to nombre$
  - $E \to (E)$

Exercice

- Trouver un mot f qui comporte plusieurs dérivations dans cette grammaire
- Y a t-il une grammaire équivalente (reconnaissant le même langage) qui soit non-ambigüe?

Grammaire désambiguée

Par exemple, priorité de $\times$ sur $+$:

- $A = \{+, \times, (,), id, nombre\}$, $V_N = \{F, P, E\}$, l'axiome est E, les règles sont:
  - $E \to P$
  - $E \to P + E$
  - $P \to F$
  - $P \to F \times P$
  - $F \to id$
  - $F \to nombre$
  - $F \to (E)$

Analyses

Etant donné f et la grammaire, trouver une dérivation finie (si elle existe):

$$S \to u_1 \to u_2 \ldots \to u_n = f$$

- analyse descendante: on essaie de trouver u_1 puis u_2 etc.
- analyse ascendante: on essaie de trouver u_{n-1} puis u_{n-2} etc.

Analyse descendante récursive

- Possible dans des certains cas: toute lettre détermine de façon unique, pour un non-terminal donné, la règle à appliquer
- Impossible dans les cas de récursivité à gauche
- On écrit une fonction par non-terminal
- On traduit “mot pour mot” la grammaire

Analyse récursive descendante avec retours en arrière

- On construit les mots u dont on espère qu'ils vont se dériver en f
- On écrit $u = gyv$, $f = gxh$, $x \neq y$
  - si $y \in A$ (alphabet terminal): on ne peut atteindre f et on repart en arrière dans les arborescences des choix
  - si $y \in V_N$ (non-terminal): on recherche toutes les règles dont y est le membre gauche et on applique à u successivement toutes ces règles

Ne marche pas mieux avec les grammaires récursives à gauche, et est bien plus coûteuse!

Analyse LL

Pour éviter les retours en arrière:

- Deviner la règle à appliquer en regardant les premières lettres de u
- On écrit $u = gSv$, $f = gh$ et les premières lettres de h doivent permettre de retrouver la règle à appliquer à S
- Exemple: $S \to (+SS)$, $S \to (*SS)$ et $S \to a$ est une grammaire $LL(2)$
- L'analyse syntaxique d'une grammaire $LL(k)$ se fait en O(longueur du mot)

Exemple

- Prouver que la grammaire des expressions arithmétiques (infixe) n'est pas $LL(k)$ (quelque soit k)
- Construire une grammaire $LL(1)$ équivalente

Réponse

- Les règles $E \to P$ et $E \to P + E$ posent un problème (P peut être aussi long que l'on veut, avant de voir $+$ ou pas)
- On peut "factoriser à gauche", donnant ainsi une grammaire $LL(1)$:
  - $E \to PE'$
  - $E' \to \epsilon, E' \to +PE'$
  - $P \to FP'$
  - $P' \to \epsilon, P' \to \times FP'$
  - $F \to a, F \to (E)$

Analyse ascendante

- Plus général (voir `yacc`)
- On fait des "réductions": on remplace un membre droit d'une règle par le membre gauche correspondant

Exemple

- $S \to aSbS, S \to aSb, S \to abS, S \to ab$
- Réduire $F = ababab$
- En déduire un algorithme $LR(1)$

Exemple d'utilisation

- Ecrire un programme, qui, à partir de l'arbre de syntaxe abstrait d'une expression arithmétique, calcule sa dérivée par rapport à un des identificateurs
- Proposer des règles de simplification