Werkcollege Datastructuren, April 3, 2007 (Twee niet behandelde opgaven)

Clemens Grabmayer (mailto:`clemens@phil.uu.nl`), 10 april 2007.

Extra Vraag. *Zij G een gerichte graaf met n knopen en m takken. Toon aan dat G een oneindig pad heeft dan en slechts dan als er een cykel in G is. Geef een $O(n+m)$ algoritme in pseudo-code (in termen van het Graph ADT) voor het vinden van een pad van maximale lengte in G (dit kan oneindig lang zijn) vanuit een knoop v. Geef als uitvoer de knopen en takken van dat pad (in het geval het pad oneindig is, geef een pad dat eindigt in een cykel).*

Als G een cykel heeft, dan bevat G natuurlijk ook oneindige paden. Als $\pi = e_1e_2e_3\ldots$ een oneindig pad in G is dat via de takken $e_1, e_2, e_3, \ldots$ (in deze volgorder) loopt, dan moeten, omdat G maar eindig vele takken heeft, i en j met $i < j$ bestaan zó dat $e_i = e_j$. Laat i_0 minimaal zó dat er een $j > i_0$ bestaat met $e_{i_0} = e_j$; en laat vervolgens $j_0 > i_0$ minimaal zó dat $e_{i_0} = e_{j_0}$. Dan vormen de takken $e_{i_0}, e_{i_0+1}, \ldots, e_{j_0-1}$ tezamen een cykel in G.

We geven hier maar één stap tot het volledige algoritme: een pseudo-code formalisatie van een recursief algoritme MLP dat, gegeven een gerichte graaf G en een knoop v van G, de maximale lengte van een pad in G vanuit v berekent, als die bestaat, en anders herkent dat er cyklische paden vanuit v in G bestaan. Het algoritme MLP gebruikt een depth-first search vanuit v, het houdt bij welke knopen al zijn bezocht, welke knopen op het actieve back-tracking pad liggen, en voor bezochte knopen u die niet op het actieve back-tracking pad liggen houdt het in $mlp(u)$ bij de al berekente maximale lengte van een pad in G vanuit u.

Algorithm MLP(G, v):
  Input: A graph G and a vertex v of G
  Output: A tuple $\langle b, M\rangle$, where b is a boolean, and M an integer such that $b =$ **false** if there exists a cyclic path starting in v, or $b =$ **true** if there are no cyclic paths starting in v, and M is the length of a maximal path starting in v.

```
Label v as "visited" and "on active path"
mlp(v) ← 0
for all edge e in G.incidentEdges(v) do
  if edge e is not labelled "visited" then
    Label e as "visited"
    w ← G.opposite(v, e)
    if w is not labelled "visited" then     {a new vertex is discovered}
      ⟨b, M⟩ ← MLP(G, w)      {recursively call MLP on w}
      if b = false then
        return ⟨false, −1⟩
      mlp(v) ← max{mlp(v), M + w(e)}
    else if w is labelled "visited", but not "on active path" then
          {the max. length of a path from w has already been computed}
      mlp(v) ← max{mlp(v), mlp(w) + w(e)}
    else    {a loop is detected}
      return ⟨false, −1⟩
Remove label "on active path" from v
return ⟨true, mlp(v)⟩
```

Het in de opgave gevraagde algoritme dat als uitvoer geeft een maximale pad, is een verfijning van het algoritme MLP.

C-13.21 *Computer networks should avoid single points of failure, that is, network nodes that can disconnect if they fail. We say that a connected graph G is* biconnected *if it contains no vertex whose removal would divide G into two or more connected components. Give an $O(n+m)$-time algorithm for adding at most n edges a connected graph, with $n \geq 3$ vertices and $m \geq n-1$ edges, to guarantee that G is biconnected.*

Het idee is om een iterator voor de knopen van G te zien als de definitie van een aftelling $V = \{v_1, \ldots, v_n\}$ van de knopen van G, die gebruikt kan worden om n nieuwe takken aan G toe te voegen die respectievelijk v_1 met v_2, v_2 met v_3, $\ldots$, v_{n-1} met v_n, en v_n met v_1 verbinden. De graaf G' die op deze manier uit G ontstaat, is biconnected omdat na de verwijdering van een knoop de resterende knopen blijven verbonden alleen maar via de toegevoegde takken.

Dit idee is hieronder uitgewerkt tot een pseudo-code formalisatie van een algoritme dat runtime $O(n+m)$ heeft:

Algorithm Biconnect(G):
  Input: A graph G with n vertices and m edges
  Output: A biconnected graph G' that results from G by adding $\leq n$ edges

```
G1 ← G.clone()                      {make a working copy G1 of the graph G}
vertit ← G1.vertices()              {vertit is an iterator of the vertices of G1}
v0 ← vertit.next()                  {initialize the current node to the first node}
vstart ← v0                         {keep the reference to the first node}
while (vertit.hasNext()) do           {loop through the vertices of G1}
  v1 ← vertit.next()
  G1.insertEdge(v0, v1, null)         {add a new edge between consecutive vertices}
  v0 ← v1
G1.insertEdge(v0, vstart, null)       {add an edge between last and first vertex}
G' ← G1
return G'
```