Graph Traversal

visit all nodes and edges in a graph systematically

gathering global information

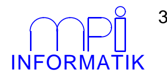
1 Breadth First Search

- find all nodes reachable from some source node s
- Prove this by giving a minimum length path

Applications: Minimize number of train changes; Mr. Euler decides that he hates bridges; low radius spanning tree; subroutines; . . .

Lemma 1.

If $p' = \langle s, \ldots, v, v' \rangle$ *is a shortest path to* v' *then* $p = \langle s, \ldots, v \rangle$ *is a shortest path to* v



```
// Find a BFS tree rooted at s
parent=[⊥,...,⊥] : NodeArray of Node
depth=[n,...,n] : NodeArray of {0..n}              // n ≡ ∞
parent[s]:= s                                      // selfloop signals root
depth[s]:= 0
q=⟨s⟩ : FIFOQueue of Node
while q ≠ ⟨⟩ do
    u:= q.popFront()
    foreach (u,v) ∈ E do
        if parent(v) = ⊥ then
            q.pushBack(v)
            parent(v):= u
            depth[v]:= depth[u] + 1
// Note that the parent pointers are reverse to the corresponding edges in E
```

Correctness

The execution of BFS can be subdivided in phases such that in phase k, q contains only nodes with depth k and (after it) $k+1$.

At the beginning of phase k, q contains all nodes of depth k.

Proof. Induction over k. True for $k = 0$.

$k-1 \leadsto k$:

When phase $k-1$ ends, only nodes of depth k are in q.

Suppose some depth k node v is missing in q.

Let $p = \langle s, \ldots, v', v \rangle$ denote a length k path to v.

v' was in q at the beginning of phase $k-1$.

We have explored the edges leaving v'.

Contradiction □

Analysis

Theorem 2. *BFS runs in time $\mathcal{O}(m+n)$*

- Use (e.g.) adjacency arrays
- We look at each of m edges once (navigation)
- $\leq n$ queue insertions and removals
- all operations are constant time

1.1 Generalization

```
// Find a Q-tree rooted at s
parent=[⊥,...,⊥] : NodeArray of Node
parent[s]:= s                                   // selfloop signals root
q=⟨s⟩ : Q of Node                               // any queue type Q
while q ≠ ⟨⟩ do
    u:= q.removeSomeElement()
    foreach (u,v) ∈ E do
        if parent(v) = ⊥ then
            q.pushBack(v)
            parent(v):= u
        else do sth with (u,v)
```

Possibly restart at any nonvisited node

Preview

What different queue types give us:

FIFO: BFS

Stack: perhaps fastest way (sequential) to get some spanning trees of the CCs of an undirected graph (not quite DFS)

Priority Queue: depends on key type:

- edge weights: Minimum spanning trees
- paths: Shortest paths

2 Depth First Search

Idea: Nodes on stack; descend without looking left and right

Presentation is close to Ulrik Brandes' Dagstuhl 2003 talk

2.1 DFS Template

S : *Stack*

foreach $s \in V$ **do**

  if s is not marked **then** // root of a DFS tree

    mark s; S.*push*(s)

    root(s)

    while $S \neq \langle\rangle$ **do**

      v:= S.*top*$()$

      if $\exists e = (v, w) \in E : e$ is unmarked **then**

        traverse(v, w)

        mark *e* // does NOT mark reverse edge

        if w is not marked **then**

          mark *w*; S.*push*(w)

      else w:= S.*pop*$()$; *backtrack*$(S.top(), w)$

2.2 DFS Tree

parent : *NodeArray* **of** *Node*

// node marks could be implemented using *parent*

Procedure *root*(s) *parent*(s):= s

Procedure *backtrack*(v, w) *parent*(v):= w

2.3 DFS Numbering

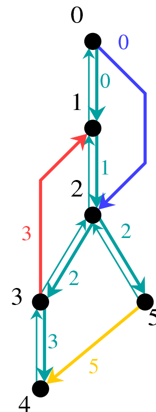
Procedure *root*(s)

  dfsNum$[s]$:= 0

  d:= 1

Procedure *traverse*(v, w)

  if w is not marked **then** *dfsNum*$[w]$:= d++



Def: $u \preceq v \Leftrightarrow dfsNum[u] \leq dfsNum[v]$

Classification of edges:

tree edge	w not marked
forward edge	w marked, $v \prec w$
back edge	w marked, $w \preceq v$, $w \in S$
cross edge	w marked, $w \prec v$, $w \notin S$

2.4 Topological Sorting

Sort nodes such that all edges go from left to right

topOrder : *Sequence* **of** *Node*

Procedure *traverse*(v, w) **if** (u, w) is a back edge **then**

  throw exception“the graph contains a cycle”

  // One cycle consists of (v, w) and

  // a path of tree edges from w to v.

Procedure *backtrack*(v, w) *topOrder.pushFront*(w)

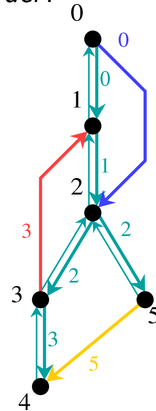
Correctness

A node w is finished, when $backtrack(u,w)$ is called.

To prove correcntess it suffices to show that when w is inserted into $topOrder$, there are no edges (u,w) with u already in $topOrder$.

Assume the contrary. What kind of edge could (u,w) be?

tree edge	u would not be finished
forward edge	u would not be finished
back edge	alg. would signalled a cycle
cross edge	w would have finisheded before



2.5 Strongly Connected Components (SCC)

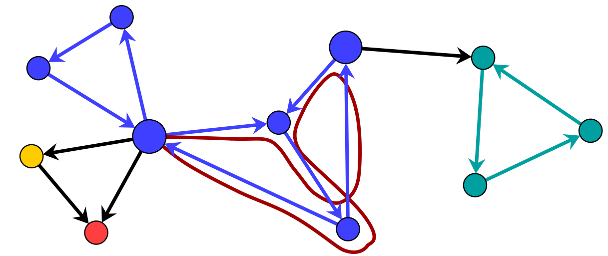
Lemma 3. *two vertices are in the same strongly connected component, if and only if they lie on a cycle.*

output: $\forall u$ *component*$[u]$ is a unique representative node.

Every graph is a DAG of SCCs

Applications: comm. network, equivalence classes, lifeness, 2SAT ...

SCC solution + DAG solution = general solution?

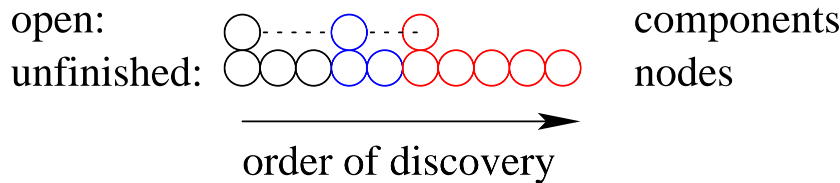


Idea for a single pass algorithm

Maintain SCCs of of nodes and edges seen so far.

components are open / closed, if dfs has visited some of / backtracked over all its inducing elements.

Maintain two additional stacks:



Algorithm

Procedure *root*(s) *open.push*(s); *oNodes.push*(s)

Procedure *traverse*(v,w)

  if (v,w) is a tree edge **then** *open.push*(w); *oNodes.push*(w)

  elsif $w \in$ *oNodes* **then** // collapse components on cycle

    while $w \prec$ *open.top*() **do** *open.pop*()

Procedure *backtrack*(v,w)

  if $w =$ *open.top*() **then**

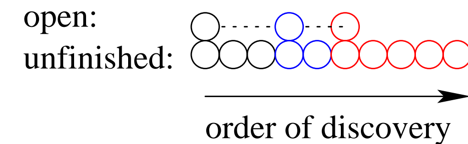
    open.pop()

    repeat

      u:= *oNodes.pop*()

      component$[u]$:= w

    until $u = w$



2.6 Biconnected and 2-Edge-Connected Components

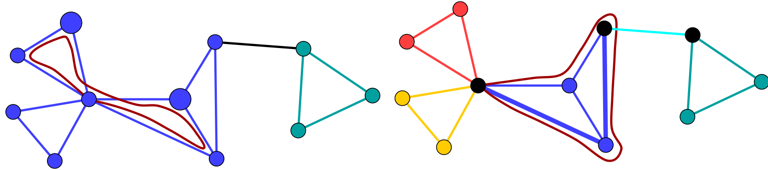
nodes / edges

of an undirected graph are in the same

2-edge connected / biconnected

component if they lie on a

cycle / simple cycle



Undirected graphs are trees of such components.

An application: networks that tolerate link / node failures

Generic Algorithm: Components Based on Cycles

Procedure *root*(s) new component

Procedure *traverse*(v, w)

  if (v, w) is a tree edge **then** create new open component

  elsif (v, w) closes a cycle **then** merge components

Procedure *backtrack*(v, w)

  if leaving component **then** close component

2-Edge-Connected Components

Procedure *root*(s) $\mathit{open}.\mathit{push}(s)$; $\mathit{oNodes}.\mathit{push}(s)$

Procedure *traverse*(v, w)

  if (v, w) is a tree edge **then** $\mathit{open}.\mathit{push}(w)$; $\mathit{oNodes}.\mathit{push}(w)$

  elsif $w \in \mathit{oNodes}$ **then** // collapse components on cycle

    while $w \prec \mathit{open}.\mathit{top}()$ **do** $\mathit{open}.\mathit{pop}()$

Procedure *backtrack*(v, w)

  if $w = \mathit{open}.\mathit{top}()$ **then**

    open.pop()

    repeat

      u:= $\mathit{oNodes}.\mathit{pop}()$

      component$[u]$:= w

    until $u = w$

Biconnected Components

Procedure *traverse*(v, w)

  $\mathit{oEdges}.\mathit{push}((v, w))$

  if $\{v, w\}$ is a tree edge **then** $\mathit{open}.\mathit{push}(\{v, w\})$

  elsif $\{v, w\}$ is a back edge **then** // no cross edges!

    while $\mathit{dfsNum}(w) < \min \mathit{dfsNum}(\mathit{open}.\mathit{top}())$ **do** $\mathit{open}.\mathit{pop}()$

Procedure *backtrack*(v, w)

  if $\{v, w\} = \mathit{open}.\mathit{top}()$ **then**

    open.pop()

    repeat

      e:= $\mathit{oEdges}.\mathit{pop}()$

      component$[e]$:= $\{v, w\}$

    until $e = \{v, w\}$