Week 4

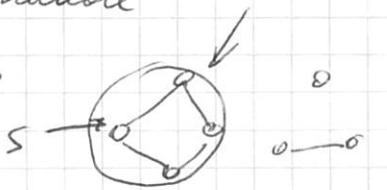
Graph Search

motivations

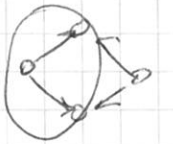
1. check if a network is connected (get from A to B)
2. driving directions (path through network, shortest, etc)
3. (sudoku?) - formulate a plan
4. compute "pieces" (components) of a graph clustering, structure of web graph, etc

Generic Graph Search

① Find everything findable from a given vertex



② don't explore everything twice



Goal $O(m+n)$ time

starting vertex

Generic Algorithm (graph G, vertex s)

- initially only s is explored
- while possible
  - choose an edge (u, v) with u explored and v unexplored
  - mark v explored

/

so it doesn't explore twice and found everything findable

Claim

at the end of the algorithm, v explored

$\Leftrightarrow$

G has a path from s to v

BFS vs DFS

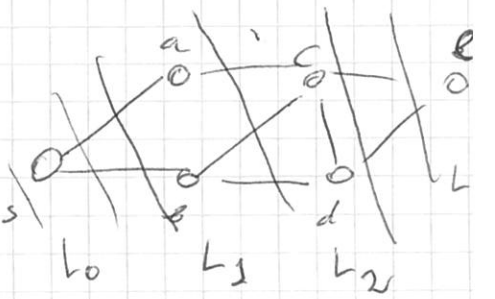
how to select a node to explore next?

both $O(m+n)$

- Breadth-First Search BFS
  - explores in "layers" FIFO
  - can compute shortest path
  - can compute connected components of an undirected graph
- Depth-First Search DFS LIFO/recursion
  - explore aggressively, backtrack only when necessary
  - compute topological ordering of a directed acyclic graph
  - compute connected components

BFS

- explore in "layers"
- shortest path
- connected components



BFS (graph G, start vertex s)

[all nodes are initially unexplored]

- mark s as explored
- let Q = queue, initialized with s
- while $Q \neq \emptyset$

  remove v - first from Q

  for each edge (v, w)

    if w - unexplored

      mark w as explored

      add w to Q

Basic BFS properties

Claim #1 at the end of BFS, v explored ⇒ G has a path from s to v

(see generic algorithm)

Claim #2

$O(m_s + n_s)$ running time

(m_s = # reached from s, n_s = # reached from s)

by code ~~exp~~ inspection

Application: Shortest Path

Goal: compute dist(v) = the fewest number of edges on the path from s to v

extra code

$$dist(v) = \begin{cases} 0 & \text{if } v = s \\ \infty & \text{if } v \neq s \end{cases}$$

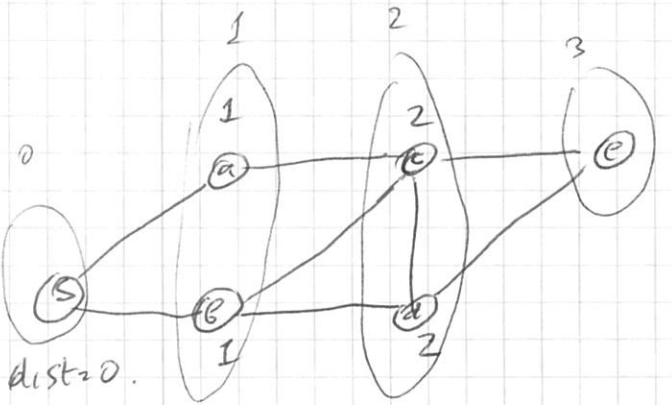
↑ initialization

when considering edge (v, w):

if w unexplored

set $dist(w) = dist(v) + 1$

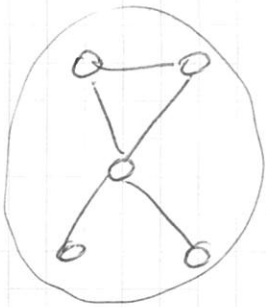
Claim: at termination $dist(v) = i \iff$ v is in the i^{th} layer



Special property of BFS

Application: Undirected Connectivity

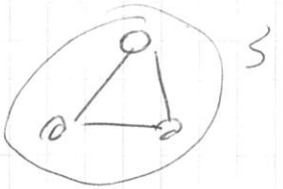
Let $G = (V, E)$ be an undirected graph



connected components - the pieces of G

Formal Definition: equivalence classes of the relation

$u \leftrightarrow v \Rightarrow$ there is a u-v path in G



connected components

Goal: compute all connected components

Why?

- is network disconnected?
- graph visualization
- clustering (quick and dirty)

Connected Components

[assume ~~nodes~~ labelled 1 to n]

for i = 1 to n:

if i not explored

BFS(G, i)

Running time: $O(n+m)$

$O(1)$ per node

$O(1)$ per edge in BFS

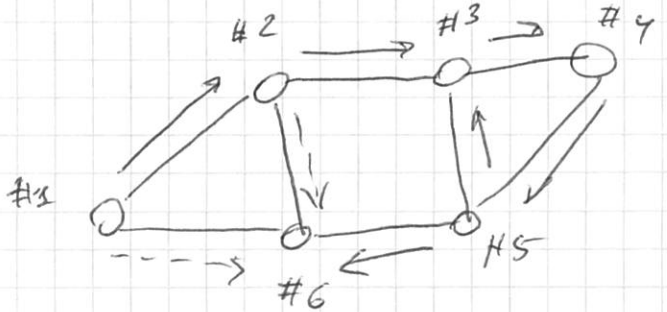
~~Depth~~ Depth-First Search

DFS:

explore aggressively

backtrack when necessary

Run time $O(n+m)$



- computes a topological ordering of a directed acyclic graph
- and strongly connected components of a directed graph

DFS: same as BFS, but using stack

Recursive version

```
DFS(graph G, start vertex s)
    mark s as explored
    for every edge (s, v)
        if (v) is unexplored
            DFS(G, v)
```

Claim 1 at the end of the algorithm, v marked as explored

$\Leftrightarrow$

there exists a path from s to v in G

Claim 2: Running time is $O(n_s + m_s)$

Reachable from s

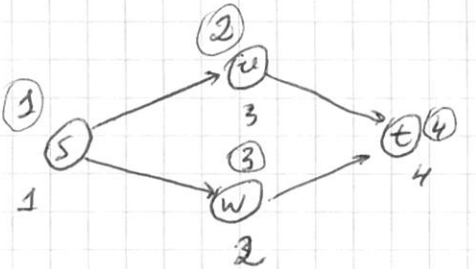
Topological sort

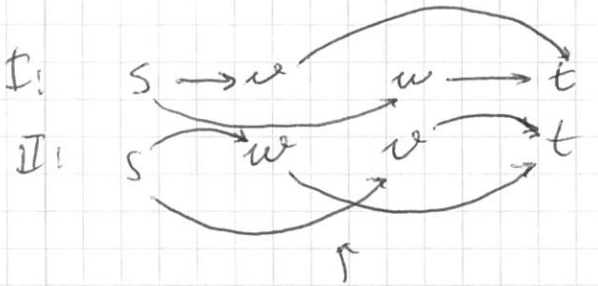
A topological ordering of a directed graph G is a labelling f of G's nodes such that

1. the $f(u)$'s are the set $\{1, \ldots, n\}$
2. $(u, v) \in G \Rightarrow f(u) < f(v)$

① first case

② second case





all edges go forward!

Motivation

sequence tasks while respecting all precedence constraints

(courses at uni, etc)

If (directed) G has a cycle, it has no topological ~~cycle~~ ordering

Theorem: no directed cycle ⇒ can compute topological ordering in $O(m+n)$

Straightforward Solution

every directed acyclic graph has a ~~sink~~ sink vertex

To compute topological ordering

↑ let v be a sink vertex of G

set $f(v) = n$

↓ recurse on $G - \{v\}$

(computing backwards, finding a sink on each iteration)

Why does it work?

when v is assigned to position i
all outgoing ~~arcs~~ arcs already deleted $\Rightarrow$

all lead to later vertices in ordering

But ~~it~~ can be computed using DFS very quickly

~~DFS (graph G, vertex s)~~

~~Mark s explored~~

DFS (graph G, vertex s)

so we find a sink and then backward to the beginning

- for every edge (s, v)
  - if v not visited
    - mark v explored
    - DFS(G, v)

DFS

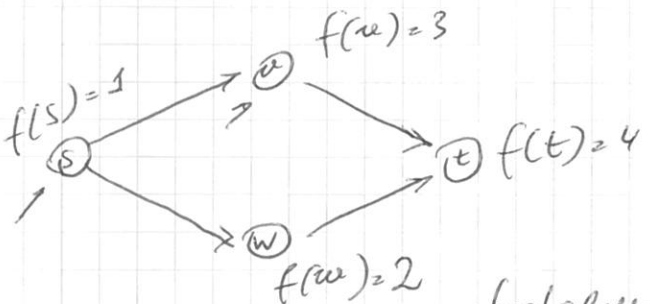
set f(s) = current_label

current_label = current_label - 1

this will find a sink

and here in sink a label will be assigned

2



for labelling

1

DFS-loop(graph G, ~~vertex s~~)

- mark all nodes unexplored
- current_label = n ← to keep track of ordering
  - ↖ global
- for each vertex
  - if v not explored
    - DFS(G, v)

Correctness:

if (u, v) is an edge, $f(u) < f(v)$

Case 1 u visited by DFS before v,
recursive call to v finishes
before the call to u (DFS!)
$\Rightarrow f(v) > f(u)$

Case 2 v is visited before u,
v's call finishes before
u's even starts
$f(v) > f(u)$

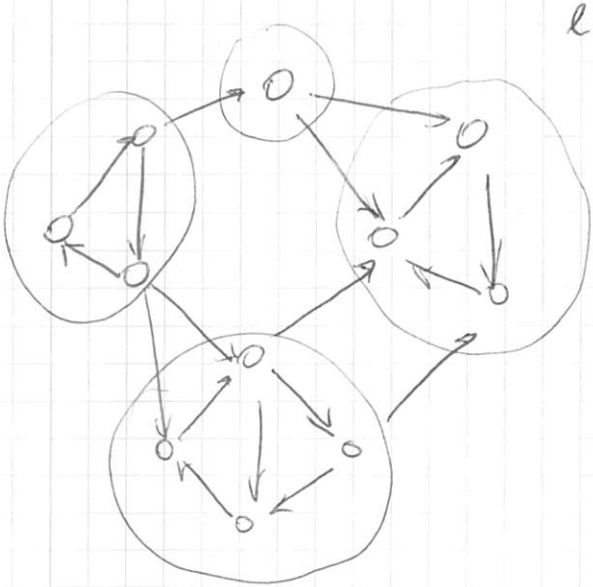
Computing Strong Components

Strongly connected — you can get to any point from any point

Formal Definition (SCCs) — of a directed graph G are the equivalence classes of the relation

$u \leftrightarrow v \iff$ there exists a path $u \to v$ and a path $v \to u$ in G

e.g.



Based of DFS?

- Where to start?
  - It depends on the starting point
    with good s.p. we may discover a SCC
  - with bad, the whole graph

Kosaraju's Two-Pass Algorithm

Kosaraju's

1. let G_{rev} = G with all arcs reversed
2. Run DFS-Loop on G_{rev}
   let f(v) = "finishing time" ← compute "magical ordering" of nodes
3. Run DFS-Loop on G
   ↖ discover the SCCs one-by-one

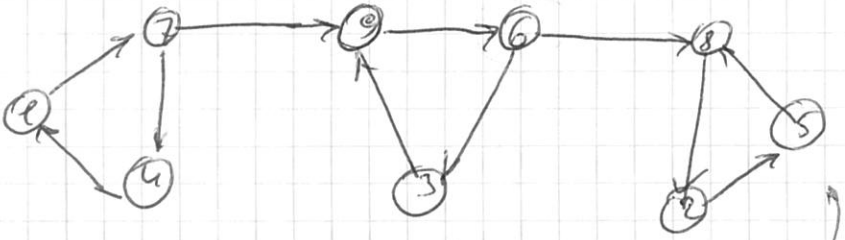
processing nodes in decreasing order of finishing time

[SCCs – nodes with the same "leader"]

```
DFS-Loop (graph G)
    global var t = 0
        // number of nodes processed so far
    global s = NULL
        // most recent node from which DFS was initiated
    Assume nodes labelled 1 to n
    for i = n down to 1
        if i not yet explored
            s = i
            DFS(G, i)

DFS (graph G, node i)
    mark i as explored
    leader(i) = node s
    for each arc (i, j) in G:
        if j not yet explored
            DFS(G, j)
    t++; set f(i) = t        // finishing time
```

Example



What is a possible set of finishing times for the nodes 1...9

when the DFS-Loop is executed on the graph

leader (i)

1	2	3	4	5	6	7	8	9
		9		9	9		9	9

explored

1	2	3	4	5	6	7	8	9

$t = 0$.

$S = \emptyset$

for 9 to 1:

9 not yet explored

$S = 9$

DFS(G, 9)

12

DFS(G, 9)

mark 9

leader(9) = 9

and call 6.

$t = 6$

$f(9) = 6$

2

11

DFS(G, 6)

mark 6

leader(6) = 9

$t = 5$

$f(6) = 5$

7

4

3

10

DFS(G, 2)

mark 2

leader(2) = 9

5

DFS(G, 8)

mark 8

leader(8) = 9

DFS(G, 3)

mark 3

leader(3) = 9

$t = 2$

$f(2) = 2$

6

$t = 1$

$f(6) = 1$

DFS(G, 5)

mark 5

leader(5) = 9.

$t = 3$

$f(8) = 3$

$t = 4$

$f(3) = 4$

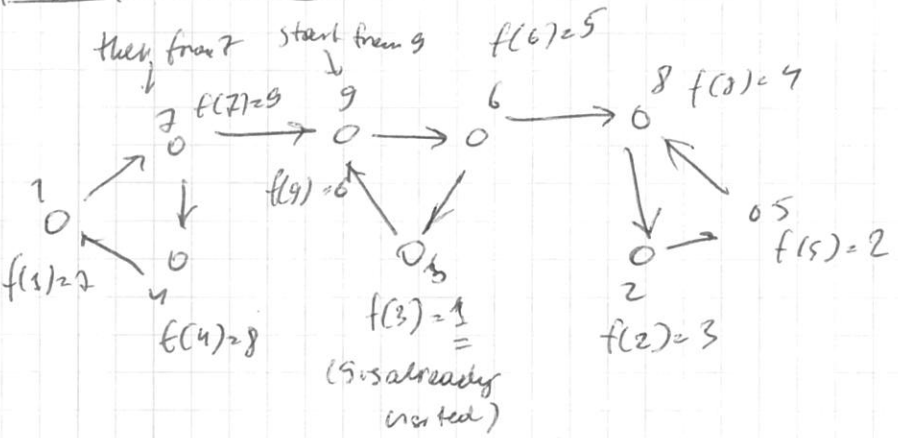
1

or

if from 6 go to 3, not to 8

| 7 | 3 | 1 | 8 | 2 | 5 | 9 | 4 | 6 |

= t

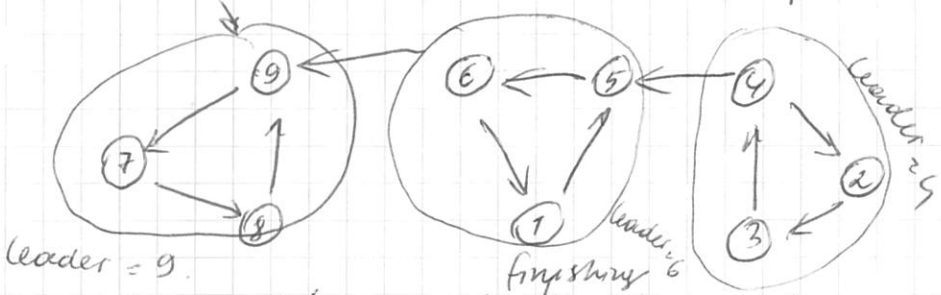


Example : Second Pass

the point of the first pass is to compute a magical ordering (these finishing times)

now we will replace original node names with finishing times

and it's run on the original graph, not reversed (as the ~~te~~ 1st pass)



now we need to compute times in second pass (from 9 down to 1)

nine is our leader. So let's start from 9

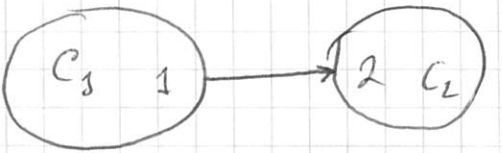
next is 6

next is 4.

Running Time: 2-BFS = $O(m+n)$

The correctness

Key Lemma two adjacent SCCs:

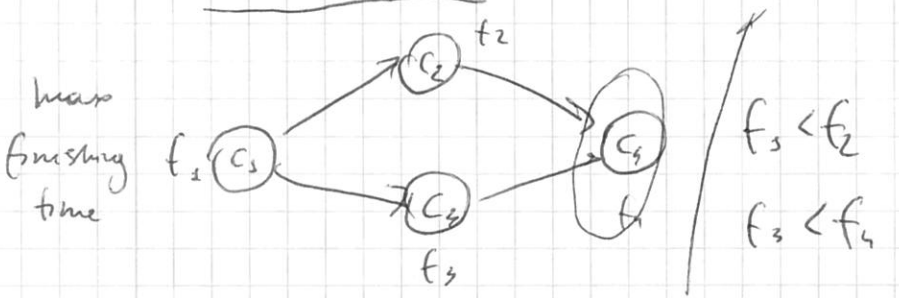


let $f(v)$ = finishing times of DFS-loop in G^{rev}

then $\max_{v \in C_1} f(v) < \max_{v \in C_2} f(v)$

Corollary

maximum finishing time is in a "sink SCC"



by corollary: 2nd pass of DFS-loop begins somewhere in a sink SCC C^*

- First call to DFS discovers C^* and nothing else!

- Second pass:

successive calls to DFS(G, i) "peels off" the SCCs one-by-one

↗

in reverse topological order

Web Graph

vertices - web pages edges - hyperlinks