# 1 Graph Search And Connectivity

## 1.1 A Few Motivations

1. Check if a network is connected (can get to anywhere from anywhere else)
2. driving directions
3. formulate a plan [e.g. how to fill in a sudoku puzzle]
   (a) nodes = a partially completed puzzle
   (b) arcs = filling in one new square
4. compute the "pieces (or components)" of a graph
   (a) clustering, structure of the web graph, etc.

Goals:

1. find everything findable from the a given start vertex
2. don't explore anything twice $O(m + n)$ time

Generic Algorithm (given graph G, vertex $s$)

```
- initially s explored, all other vertices unexplored
- while possible:
        - choose an edge (u, v) with u explored and v unexplored (if none, halt)
        - mark v explored
```

Claim: at end of the algorithm, v explored $\iff$ has a path from $s$ to $v$.

Proof: induction on number of iterations

by contradiction

Suppose $G$ has a path $P$ from $s$ to $v$:

but $v$ is unexplored at the end of the algorithm. Then $\exists$ edge $(u, w) \in P$ with $u$ explored and $w$ unexplored. But then algorithm would not terminate, contradiction.

### 1.1.1 BFS vs. DFS

Note: how to choose among the possibility many "frontier" edges?

**Breadth First Search (BFS)**

$O(m + n)$ using a queue (FIFO)

explore nodes in "layers"

can compute shortest paths

can compute connected components of an undirected graph

**Depth First Search (DFS)**

$O(m + n)$ time using a stack (LIFO), or via recursion

explore aggresively like a maze, backtrack only when necessary

compute topological ordering of directed acyclic graph

compute connected components in graphs

## 1.2 Breadth-First Search (BFS): The Basics

### 1.2.1 The Code

```
BFS(graph G, start vertex s)
[all nodes intially unexplored]
- mark s as explored
- let Q = queue (FIFO), initialized with s
- while Q != 0
        - remove the first node of Q, call it v
        - for each edge (v, w):
                if w unexplored
                        - mark w as explored
                        - add w to Q (at the end)
```

Claim #1: At the end of BFS, $v$ explored $\iff$ G has a path from $s$ to $v$.

Reason: Special case of the generic algorithm

Claim #2: running time of main while lop $= O(n_s + m_s)$, where:

- $n_s$= number of nodes reachable from $s$
- $m_s$= number of edges reachable from $s$

### 1.2.2 BFS and Shortest Paths

Goal: Compute $dist(v)$, the fewest number of edges on a path from $s$ to $v$.

Extra code:

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

```
when considering edge (v, w):
        -if w unexplored
                - set dist(w)=dist(v)+1
```

Claim: at termination, $dist(v) = i \iff v$ in $i^{th}$ layer (i.e. $\iff$ shortest $s - v$ path has $i$ edges)

Proof idea: every layer $i$ node $w$ is added to $Q$ by a layer $(i - 1)$ node $v$ via the edge (u, w)

### 1.2.3 BFS and Undirected Connectivity

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

Connected components = the "pieces" of $G$.

Formal definition: equivalence classes of the relation $u \sim v \iff \exists u - v$ path in $G$.

Equivalence relation:

satisfy these three properties

- reflexive - everything needs to be related to itself
- symmetric - if $u \sim v$ then $v \sim u$, true since undirected graph
- transitive - if $u \sim v$ and $v \sim w$ then $u \sim w$

Goal: compute all connected components

Why: check if network is disconnected

- graph visualization - clustering

### 1.2.4 Connected Components via BFS

To compute all components (undirected case)

```
-all nodes unexplored [O(n)]
[assume labelled 1 to n]
- for i=1 to n [O(n)]
        - if vertex i not yet explored [O(n), in some previous BFS]
                - BFS(G, i) [discovers percisely i's connected components]
```

Note: Finds every connected component.

Running time: $O(m+n)$, $m \rightarrow O(1)$ per edge in each BFS, $n \rightarrow O(1)$ per node

## 1.3 Depth First Search (DFS): The Basics

### 1.3.1 Overview and Example

Explore aggressively, only backtrack when necessary

Also computes a topological ordering of a directed acyclic graph

and strongly connected components of directed graphs

Running time: $O(m+n)$

### 1.3.2 The Code

Exercise: mimic BFS code, use a stack instead of a queue [+minor other modifications].

Recursive version:

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

### 1.3.3 Basic DFS Properties

Claim #1: at end of the algorithm, v marked as explored $\iff$ $\exists$ path from $s$ to $v$ in $G$.

Reason: particular instantiation of generic search procedure.

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

- $n_s$ =number of nodes reachable from $s$
- $m_s$ =number of edges reachable from $s$

Reason: ?ods at each node in connected component of $s$ at most once, each edge at most twice.

## 1.4 Topological Sort

Definition: A topological ordering of a directed graph $G$ is a labelling $F$ of $G$'s nodes such that:

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

Motivation: sequence tasks while respecting all precedence constraings.

Note: $G$ has directed cycle $\Rightarrow$no topological ordering.

Theorem: no directed cycle $\Rightarrow$can recompute topological ordering in $O(m+n)$ time

### 1.4.1 Straightforward Solution

Every directed, acyclic graph has a sink vertex, a vertex without any outgoing arcs.

Reason: if not, can keep following outgoing arcs to produce directed cycle.

To compute topological ordering:

```
– let v be a sing vertex of G
– set f(v)=n
– recurse on G–{v}
```

Why does it work? When $v$ is assigned to position $i$, all outgoing arcs already deleted $\Rightarrow$all lead to later vertices in ordering.

### 1.4.2 Topological Sort via DFS (Slick)

```
DFS–Loop(graph G)
– mark all nodes unexplored
– current_label=n [ to keep track of ordering]
– for each vertex v in G:
        – if v not yet explored [in some previous DFS call]
        – DFS(G, v)

DFS(graph G, start vertex s)
– mark s explored
– for every edge (s, v):
        – if v not yet explored
                – DFS(G, v)
– set F(s)=current_label
– current_label––
```

[16:05]

Running time: $O(m + n)$

Reason: $O(1)$ time per node, $O(1)$ time per edge

Correctness: need to show that if (u, v) is an edge, then $f(u) < f(v)$.

Case 1: $u$ visited by DFS before $v$ $\Rightarrow$recursive call corresponding to $v$ finishes before that of $u$ (since DFS) $\Rightarrow f(v) > f(u)$.

Case 2: $v$ visted before $u$ $\Rightarrow$$v$'s recursive call finishes before $u$'s even starts $\Rightarrow f(v) > f(u)$

## 1.5 Computing Strong Components: The Algorithm

Formal Definition: The strongly connected components (SCCs) of a directed graph $G$ are the equivalence classes of the relation $u \sim v \iff \exists$path $u \rightsquigarrow v$ and a path $u \rightsquigarrow u$ in G.

### 1.5.1 Why Depth-First Search

Depending on which vertex you start with, you can end up with one component - whole graph or many.

### 1.5.2 Kosaraju's Two-Pass Algorithm

Theorem: Can compute SCCs in $O(+n)$ time

Algorithm: (given directed graph $G$)

1. Let $G^{reverse} = G$ with all arcs reversed
2. run DFS-Loop on $G^{rev}$
   (a) goal: compute "magical ordering" of nodes
   (b) Let $f(v)$="finishing time" of each $v \in V$
3. run DFS-Loop on G
   (a) goal: discover the SCCs one-by-one
   (b) processing nodes in decreasing order of finishing times
   (c) [SCCs=nodes with the same "leader"]

#### 1.5.3 DFS-Loop

```
- global variable t=0 [for finishing times in 1st pass
[#of nodes processed so far
- global variable s=NULL [for leaders in 2nd pass]
[current source vertex]
Assume nodes labeled 1 to n
- for i=n downto 1
        - if i not yet explored
                - s:=i
                DFS(G, i)

DFS(graph G, node i)
-mark i as explored [for rest of DFS loop
- set leader(i):=node s
- for each arc (i, j) in G:
        - if j not yet explored:
                - DFS(G, j)
- t++
- set f(i):=t
[f(i) i's finishing time]
```

For second pass $\Rightarrow$reverse orientation, change names to $f(i)$'s.

Running Time: $2 \cdot DFS = O(m+n)$

### 1.6 Computing Strong Components: The Analysis

#### 1.6.1 Observation

Claim: the SCCs of a directed graph induce an acyclic "meta graph":

mega-nodes = the SCCs $C_1, \dots, C_k$ of $G$

$\exists$ arc $C \to \hat{C} \iff \exists$ arc $(i,j) \in G$ with $i \in C, j \in \hat{C}$

#### 1.6.2 Why acyclic?

A cycle of SCCs would collapse into one.

SCC of the original graph $G$ and its reversal $G^{rev}$ is exactly the same.

### 1.6.3 Key Lemma

Lemma: [7:00] Consider two "adjacent" SCCs in $G$:

Let $F(v)$ = finishing times of DFS-Loop in $G^{rev}$.

Then: $\max\limits_{v \in C_1} F(v) < \max\limits_{v \in C_2} F(v)$

Corollary: maximum F-value of $G$ must lie in a "sink SCC" [10:00]

### 1.6.4 Correctness Intuition

By Corollary: 2nd pass of DFS-Loop begins somewhere in a sink SCC $C*$.

$\Rightarrow$First call to DFS discovers $C*$ and nothing else.

$\Rightarrow$rest of DFS-Loop like recursing on $G$ with $C*$ deleted. [starts in a sink node of $G - C*$]

$\Rightarrow$successive calls to DFS($G_{ij}$) "peel off" SCCs one by one [in reverse topological order of the "meta-graph" of the SCCs]

### 1.6.5 Proof of Key Lemma

in $G^{rev}$: $C_1 \qquad i \leftarrow j \qquad C_2$ [still SCCs (of $G^{rev}$)]

let $v$ = 1st node of $C_1 \cup C_2$ reached by 1st pass of DFS-Loop (on $G^{rev}$)

Case 1 [$v \in C_1$]: all of $C_1$explored before $C_2$ ever reached

Reason: no paths from $C_1$to $C_2$ (since meta-graph is acyclic)

$\Rightarrow$all F-values in $C_1$ are less than all of F-values in $C_2$.

Case 2 [$v \in C_2$]: DFS($G^{rev}$, $v$) won't finish until all of $C_1 \cup C_2$ completely explored $\Rightarrow \forall_{w \in C_1} F(v) > F(w)$ .