HUFFMAN CODES

How does compression work? Lossy (JPG, MPEG, ...) vs Lossless (ZIP) (Also uses Huffman!) (BZIP2)

Huffman, grad student 1952, did this instead of preparing for finals. Improved on his professor's work.

- Normally, computers use 8 bits per symbol (ASCII). What happens if our file only uses ≤ 32 symbols (a-z, space, comma, ...)? We can use only 5 bits per symbol. Fixed length encoding.

- Some symbols are used much more often than others (e, a, t, i vs w, x). Maybe we can encode them using fewer bits?

- $e \mapsto 1$
  $a \mapsto 01$
  $b \mapsto 010$

  What is 0101? "aa" or "be"?

To make sure there's no ambiguity, use:

DEF A prefix code for a set S is a function $c: S \to \{0,1\}^*$ s.t.
$\forall x, y \in S$, $x \neq y$, $c(x)$ is not a prefix of $c(y)$.

Ex
$a \mapsto 11$
$e \mapsto 01$
$k \mapsto 001$
$r \mapsto 10$
$u \mapsto 0000$

$\underbrace{10}_{r}\underbrace{01}_{e}\underbrace{0000}_{u}\underbrace{01}_{k}$

Suppose we have a file of 1 billion symbols with these frequencies:

$$f_a = 0.32 \quad f_e = 0.25 \quad f_k = 0.2 \quad f_r = 0.18 \quad f_u = 0.05$$

The size of the encoded text is:

$$2f_a + 2f_e + 3f_k + 2f_r + 4f_u = 2.23 \text{ bits/symbol}$$

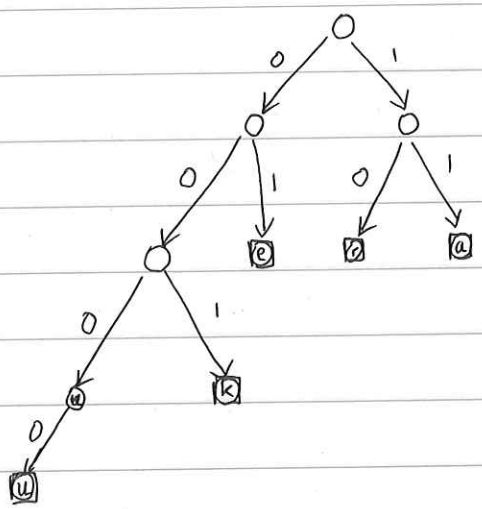
so its 2.23 billion bits.

Generally,

DEF $$ABL(c) = \sum_{x \in S} f_x \cdot |c(x)|,$$

is the average bits per letter of a prefix code c.

It is convenient to model a prefix code as a tree.



Only leaves have labels.

10110010000
"raku"

$$ABL(T) = \sum_{x \in S} f_x \cdot \text{depth}_T(x)$$

Can this tree be made more efficient?

Replace $u \mapsto 0000$ with $u \mapsto 000$. 2.25 bits/symbol.

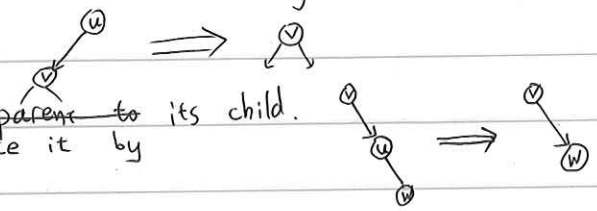
DEF A tree is full if every node that is not a leaf has two children.

Claim The binary tree corresponding to an optimal prefix code is full.

Proof Assume by contradiction T is the binary tree of an optimal prefix code and is not full. Then there is a node u with only one child:

~~Case 1: if u is the root, remove it~~

~~Case 2:~~ Remove u, and ~~connect its parent to~~ replace it by its child.



Since this decreases some depths, and leaves the others unaffected, the ABL got smaller, in contradiction.

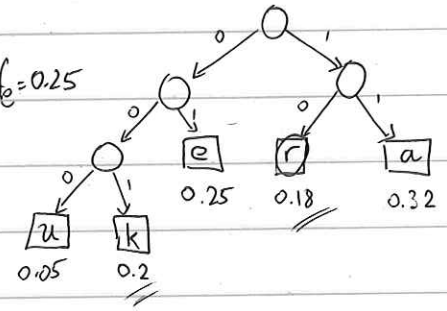
Greedy attempt #1: [Shannon-Fano 1949]

Create tree top-down, splitting S into two sets S_1 and S_2 with almost equal frequencies. Recursively build tree for S_1 and S_2.

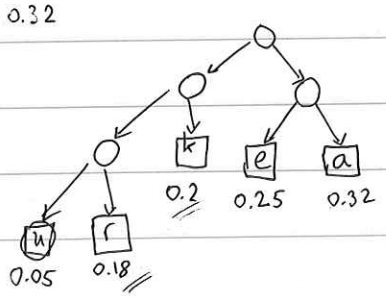
$$f_a = 0.32 \quad f_e = 0.25 \quad f_k = 0.20 \quad f_r = 0.18 \quad f_u = 0.05$$

So Shannon-Fano gives: $f_e + f_k + f_u = 0.5$ and $f_a + f_r = 0.5$

$f_k + f_u = 0.25$, $f_e = 0.25$



2.25 bits/symbol



2.23 bits/symbol

Surprisingly, you can do better!

Huffman encoding

Observation 1: Lowest frequency symbols should be at the lowest level

Observation 2: The lowest level always contains at least two leaves

Observation 3: The order in which items appear in a level does not matter

Claim 1 There is an optimal prefix code with tree T^* where the two lowest-frequency symbols are in sibling leaves.

Huffman's greedy approach [1952]: Create tree bottom up. Make two leaves for two lowest-freq letters y and z. Recursively build tree for rest plus meta-letter yz of combined freq.

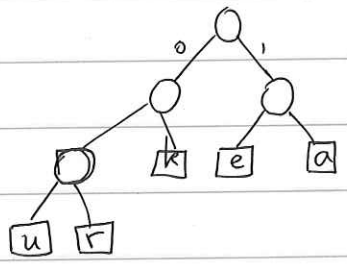
Example:

① $f_a = 0.32$ $f_e = 0.25$ $f_k = 0.20$ $f_r = 0.18$ $f_u = 0.05$

② $f_a = 0.32$ $f_e = 0.25$ $f_{rk} = 0.20$ $f_{ru} = 0.23$

③ $f_a = 0.32$ $f_e = 0.25$ $f_{kru} = 0.43$

④ $f_{ae} = 0.57$ $f_{kru} = 0.43$



HUFFMAN(S):

1. if $|S| = 2$:
2. return tree with root and two leaves
3. Let y and z be lowest freq symbols in S
4. $S' = S$
5. Remove y and z from S'
6. Insert new symbol w in S' with $f_w = f_y + f_z$
7. $T' =$ HUFFMAN(S')
8. $T =$ add two children y and z to leaf w in T'
9. Return T

Time complexity:

- Naive implementation: $T(n) = T(n-1) + O(n)$ (Step 3), so $O(n^2)$

- Use priority queue to store symbols of S with key being freq.
  Then steps 3, 5, & 6 can be implemented in $O(\log n)$ time?
  $T(n) = T(n-1) + O(\log n)$, so $T(n) = O(n \log n)$.

PRIORITY QUEUE:

- INSERT(x)
- EXTRACT-MIN : removes and returns the element with smallest key.

Runtime: $O(\log n)$ using binary heaps.
(can be used to implement sort in $O(n \log n)$ time; "Heap Sort")

Claim 2 $ABL(T') = ABL(T) - f_w$

Proof The difference between T and T' is that instead of the leaf w of T', T has two leaves y and z. Their combined freq is f_w, but they are deeper by 1.

In more detail,

$$ABL(T) = \sum_{x \in S} f_x \cdot depth_T(x)$$

$$= f_y \cdot depth_T(y) + f_z \cdot depth_T(z) + \sum_{\substack{x \in S \\ x \neq y,z}} f_x \, depth_T(x)$$

$$= f_w (1 + depth_{T'}(w)) + \text{[illegible]} \quad \text{---} \; // \; \text{---}$$

$$= f_w + f_w \cdot depth_{T'}(w) + \text{[illegible]} \quad \text{---} \; " \; \text{---}$$

$$= f_w + \sum_{x \in S'} f_x \, depth_{T'}(x)$$

$$= f_w + ABL(T')$$

Claim 3 The Huffman code achieves the minimum ABL of any prefix codes.

Proof By induction on $n = |S|$.

Base: for $n=2$ ~~there is~~ no we get $ABL = 1$ and there is no shorter code.

Hypothesis: ~~A~~ True for up to $n-1$ symbols.

Step: • By hypothesis, Huffman tree T' for S' of size $n-1$ with w instead of y and z is optimal.

• Suppose by contradiction that Huffman tree T for S is not optimal.

• So there is a tree Z such that $ABL(Z) < ABL(T)$. (*)

• By Claim 1, there is also a tree Z for which y and z are sibling leaves.

• Let Z' be Z with y and z deleted, and their former parent (now leaf) labeled w. (Just like T' is derived from T)

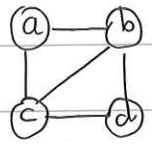
• By Claim 2, $ABL(Z') = ABL(Z) - f_w$

$ABL(T') = ABL(T) - f_w$

• By (*), $ABL(Z') < ABL(T')$ in contradiction.

Graph Algorithms (CLRS §22)

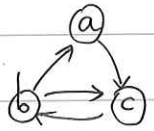
Undirected



$V = \{a, b, c, d\}$

$E = \{\{a,b\}, \{a,c\}, \{b,c\}, \{b,d\}, \{c,d\}\}$

Directed



$V = \{a, b, c\}$

$E = \{(b,a), (a,c), (b,c), (c,b)\}$

Applications:

- Google Pagerank
- Facebook social graphs
- Google Maps
- Internet routing
- Biological network (protein-protein interactions)

Ex Pocket Cube (2x2x2 Rubik cube)

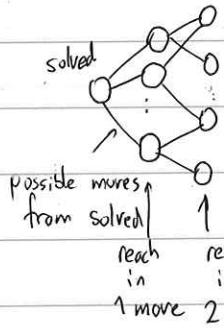
Configuration graph: one vertex for each possible state of the cube

#vertices = $|V| = 8! \cdot 3^8$ (8 cubies, and each has 3 twists) = 264,539,520

/24 (symmetries)

/3 (not all reachable)

- edges: undirected because you can undo every move.



≥ 0

≥ 0

$\uparrow$

11 = God's number moves

$2 \times 2 \times 2 = 11$

$3 \times 3 \times 3 = 20$ [2012].

$4 \times 4 \times 4$ = unknown

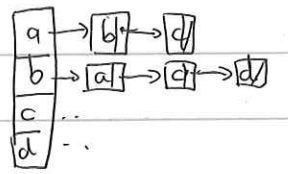
$\vdots$

$n \times n \times n = \Theta\left(\frac{n^2}{\log n}\right)$ [DEMAINE 2012]

($O(n^2)$ easy)

Graph representation:

① Adjacency list: array Adj of |V| linked list.



For each $u \in V$, $Adj[u]$ stores all of u's neighbors,

$Adj[u] = \{v \in V \mid (u,v) \in E\}$.

This is our default representation. (Storing E as an array would be extremely slow)

Space: $\Theta(|V|+|E|)$ (textbook writes $\Theta(V+E)$)

② Adjacency matrix (rarely used):

- A 2-dim array A where $A[u,v]$ = true iff $(u,v) \in E$.

Space $O(|V|^2)$

(might be preferable when $|E| \approx |V|^2$)

(very quick to tell if there's an edge from u to v)

	a	b	c
a	0	0	1
b	1	0	1
c	0	1	0

Breadth-First Search (BFS) (a graph exploration problem)

Input: A graph $G=(V,E)$ (either directed or undirected)

given as an adjacency list, and

a vertex $s \in V$ ("source")

Output: All vertices reachable from s

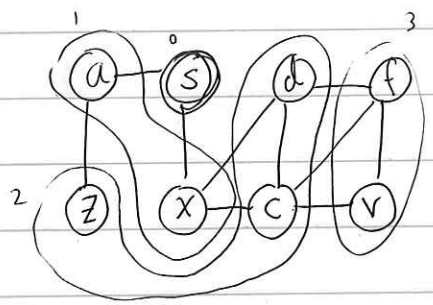
Run time $O(|V|+|E|)$ (linear time — size of input)

Idea: - look at all nodes reachable in 0 moves ($\{s\}$), 1 move ($Adj[s]$), 2 moves, etc.

- Careful to avoid duplicates (otherwise infinite runtime whenever there's a cycle)

BFS (s, Adj) (see CLRS for another implementation)

```
level = {s: 0}
parent = {s: None}
i = 1
frontier = [s]
while frontier:
    next = []
    for u in frontier:
        for v in Adj[u]:
            if v not in level:
                level[v] = i
                parent[v] = u
                next.append(v)
    frontier = next
    i += 1
```



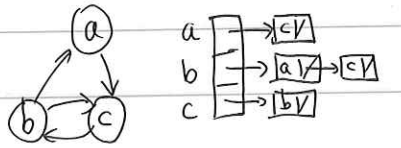
For all v, the path (v, parent[v], parent[parent[v]], ..., s) in reverse, is a shortest path from s to v (of length level[v])

The set of vertices we explored (all those reachable from s) together with the edges (parent[v], v) for each such v, give a BFS tree.

Basic Alg Lecture 19 (CLRS §22.3)

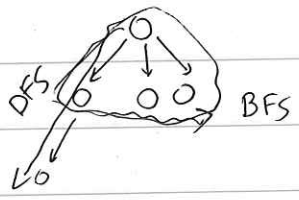
- Depth First Search (DFS)
- Edge classification
- Cycle detection
- Topological sort

Recall: graphs represented as adjacency lists:



$Adj[u] = \{v \in V \mid (u,v) \in E\}$

Remark: Facebook graph, average distance between two random people is 4.57,
(six degrees of separation: all people are ≤ 6 social connections from each other)

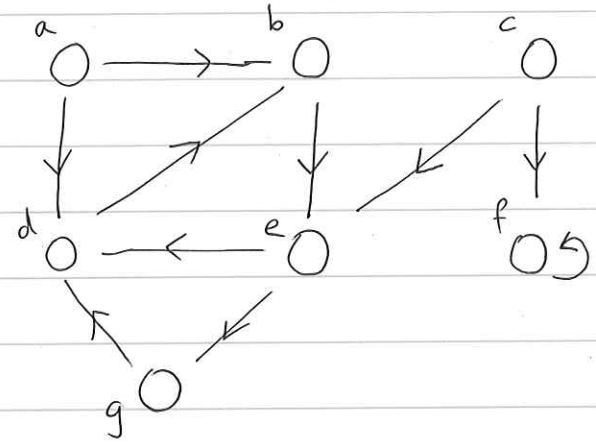


- In BFS we explore layer by layer, giving shortest paths
- In DFS we explore recursively each neighbor,

- Both algs apply to both directed and undirected graphs.

```
BFS(s, Adj)
frontier = [s]
while frontier not empty:
    next = []
    for u in frontier:
        for v in Adj[u]:
            if v.color == WHITE
                v.color = BLACK
                next.append(v)
    frontier = next
```

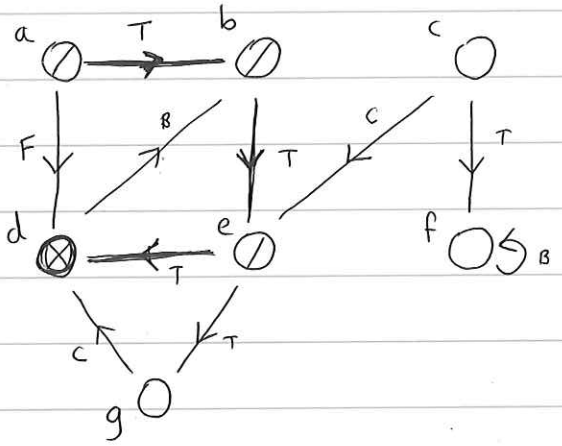
Ex



BFS(a): Level 0: {a}
Level 1: {b, d}
Level 2: {e}
Level 3: {g}

Edges from level i cannot go to level $j \geq i+2$.

DFS(a):



○ WHITE

⊘ GRAY

⊗ BLACK

```
DFS-VISIT(G, u)
LATER  1.  time += 1
       2.  u.d = time
       3.  u.color = GRAY
       4.  FOR v ∈ Adj[u]:
       5.      IF v.color == WHITE:
       6.          v.parent = u
       7.          DFS-VISIT(G, v)
       8.  u.color = BLACK
LATER  9.  time += 1
       10. u.f = time
```

runs once for each edge $(u,v) \in E$

$\Theta(|E|)$

We usually use DFS to learn something about the graph (as opposed to about a vertex as in BFS). We therefore usually use:

```
DFS(G)
LATER  1.  time = 0
       2.  FOR u ∈ V
       3.      u.color = WHITE
       4.      u.parent = NIL
       5.  FOR u ∈ V:
       6.      IF u.color == WHITE:
       7.          DFS-VISIT(G, u)
```

$\Theta(|V|)$

Runtime:

① $\Theta(|V|)$ in DFS (not counting recursive calls)

② We only call DFS-Visit on each vertex v once (when it's white / first discovered), spending $|Adj[v]|$ time. So in total we spend $\Theta(\sum_{v \in V} |Adj[v]|) = \Theta(|E|)$ time (by Handshake lemma; see appendix of CLRS).

$\Rightarrow$ $\Theta(|V|+|E|)$ runtime.

Just like BFS, but provides other properties. (E.g., does <u>not</u> provide shortest paths; see (a,d)).

Edge Classification

- Tree edge: visit new vertex via edge (i.e., vertex was white)
- Forward edge: node → descendant in tree.
- Backward edge: node → ancestor in tree
- Cross edges: between two non-ancestor related vertices.

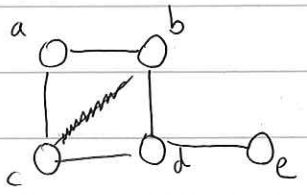
How can we detect type of edge (u,v) is?

- Tree edge: v is white when we explore this edge.
- Forward edge: v is black — "—
- Backward edge: v is gray — "—
- Cross edge: v is black

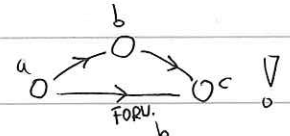
To distinguish forward from cross edge, we add "discovery time" and "finish time" to each vertex. [SHOW EXAMPLE & MODIFY ALG]

- In Forward edge, v is black and v.d > u.d
- In Cross edge, v is black and v.d < u.d (and also v.f < u.d)

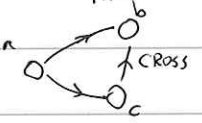
Ex



We can't have



We also can't have



THM 22.10 In DFS of an undirected graph G, every edges is either tree edge or backedge (no forward or cross edges) In other words, when we (first) explore $(u,v) \in E$, (from u,) v cannot be black.

Proof If v is black, it means we finished exploring its neighbors, in particular ~~[illegible]~~ have been black already. we should have explored (u,v) from v already.

Cor An undirected graph is acyclic iff there are no back edges.

Proof If no back edges, we only have tree edges, so acyclic; if there is a backedge, forms a cycle.

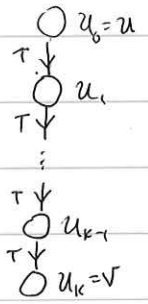
What about directed graphs?

(white-path theorem)

Thm 22.9 ^ v is a descendant of u in the DFS forest iff at time $u.d$ there exists a path from u to v with only white vertices.

Proof idea:

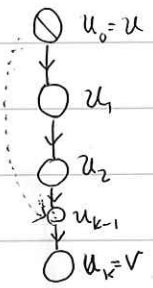
$\Rightarrow$ Let $u_0 = u, u_1, u_2 \ldots u_k = v$ be a path in the DFS forest (i.e., (u_i, u_{i+1}) is a tree edge $\forall i$). Then u_1 was discovered from u_0, u_2 was discovered from u_1, etc. Therefore, $u_k.d > u_{k-1}.d > \ldots > u_1.d > u_0.d$, so $u_1 \ldots u_k$ are all white at time $u_0.d$.



$\Leftarrow$ u_1 is finished before u_0 is, u_2 is finished before u_1 is etc.

$$u_k.f < u_{k-1}.f < \cdots < u_1.f < u_0.f$$

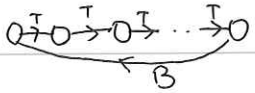
Therefore, v is detected during the visit from u, and must have a path from u in the DFS forest.



Lemma 22.11: A directed graph is acyclic iff DFS finds no back edges.

We will equivalently prove: a directed graph has cycles iff DFS finds back edges.

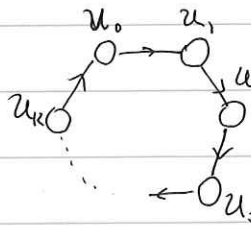
Proof.

$\Leftarrow$ 

Any back edge immediately creates a cycle.

$\Rightarrow$

Assume there is a cycle. Label its vertices so that the first to be discovered by DFS is u_0. Then, by Thm 22.9, u_k will be a descendant of u_0. Then the edge (u_k, u_0) is a back edge.

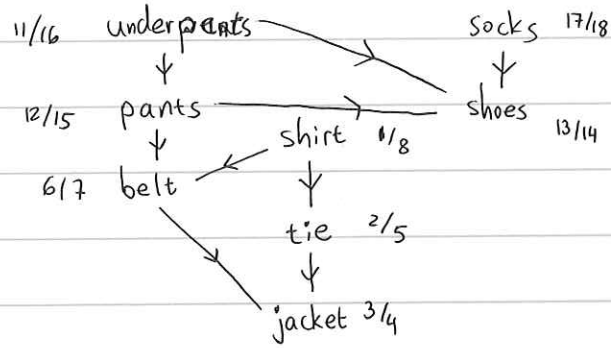


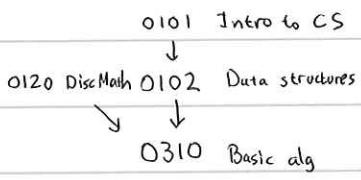
start u_0
start u_k
finish u_k
finish u_0

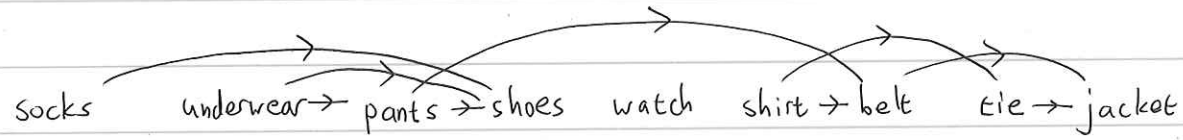
balanced parentheses
$(\ldots(\ldots)\ldots)$
$u_0\ u_k\ u_k\ u_0$

Topological Sort (CLRS §22.4)

In job scheduling, there is a set of tasks to perform where some tasks must be completed before others. E.g., prereqs in courses. The input is represented as a directed acyclic graph (DAG).







Algorithm:

Input: DAG $G=(V,E)$

1. Run DFS(G)
2. Output vertices in decreasing order of finish time

Runtime is $O(|V|+|E|)$. In particular, no need to sort by finish time; we just add vertices to front of linked list as they are finished.

Remark Name comes from graph being a "topology"

Thm Alg outputs a topo. sort

Proof It suffices to show that for all $(u,v)\in E$, $v.f < u.f$. When we explore (u,v), in the DFS alg., v cannot be gray by Lemma 22.11. Therefore, either black or white.

- If black, v.f is already set, but we are still exploring u (it is gray), so u.f is not set yet. We get $v.f < u.f$.
- If white, we will recursively visit v. When recursive call returns, v.f is set, but u.f is not. So again $v.f < u.f$. ∎

If you see DAG think topological sort!

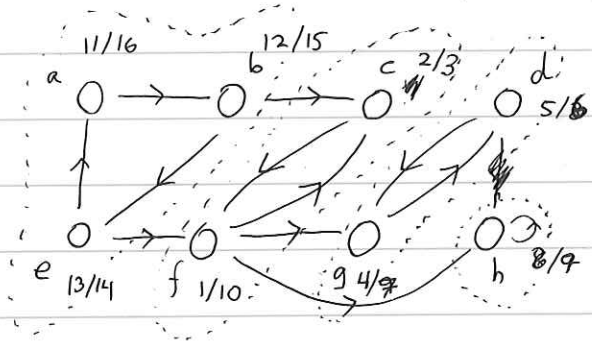
Strongly-connected components (SCC) (CLRS §22.5) Kosaraju / Sharir

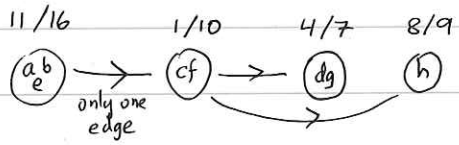
If you see a directed graph, think SCC!

We will show a $O(|V|+|E|)$ time algorithm for finding the SCC decomposition.

[ADD TIMES LATER]

G:

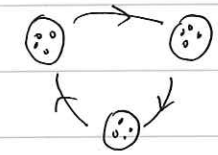




G^{SCC} : Component graph

Claim The component graph is a DAG

Proof A cycle would form a larger SCC, in contradiction:



Consider now DFS on the graph G.

We can pretend that this DFS takes place in G^{SCC} We say that:

- A component is discovered when one of its vertices is discovered
- —— " —— finished when all its vertices are finished.
- A component starts white, then becomes gray when discovered, then black when finished

Key lemma: the "virtual" walk on G^{SCC} induced by DFS on G is a valid DFS exploration of G^{SCC}.

Proof idea: When a ~~vertex~~ component C is first discovered, say by a call to DFSVisit on some $u \in C$, then when that call returns, all vertices in C as well as in all components reachable from C by white path, are finished, i.e., black, just like in a real DFS visit.

In particular, if there is an edge from C to C' in G^{scc} (which is a DAG) then $C.f > C'.f$, just like we saw in the analysis of topological sort. Therefore, the component with largest finish time, must be first in the topological sort of G^{scc}.

Observation: the last vertex to finish is in the left-most component.

We can now identify the entire left-most component by running DFS-Visit from that vertex but on the transpose graph G^T.
Reason: the SCC of G^T is like that of G, but with edges reversed.

SCC(G)

1. Call DFS(G) to compute $u.f$ for each $u \in V$
2. Call DFS(G^T) where in the main outerloop consider vertices in order of decreasing $u.f$.
3. Output the vertices of each tree as a separate SCC.

Run time: $O(|V|+|E|)$