Directed Graphs

Digraph = set of vertices connected pairwise by directed edges

edge goes from vertex (y) to vertex (x)

* directed path
* directed cycle
* outdegree
* indegree

Digraph problems

- path: is there a directed path from s to t
- shortest path: what is the shortest directed path from s to t
- topological sort: can you draw a digraph so that all the edges point upwards
- strong connectivity: is there a directed path between all pairs of vertices
- transitive closure: for which vertices v

and w is there a path from v to w?

- page rank : what is the importance of a webpage?

Digraph API

class Digraph

- void addEdge(int v, int w) $\boxed{v \to w}$
- Iterable<Integer> adj(int v)
  - ↳ vertices pointing from v
- int V()
- int E()
- Digraph reverse()

* use adjacency-lists digraph representation

Digraph Search

Problem : find all vertices reachable from s along a directed path

- Depth-first search in digraphs

- same method as for undirected graphs

- every undirected graph is a digraph (with edges in both direction

- DFS is a digraph algorithm

DFS (to visit a vertex v)

- mark v as visited
- recursively visit all unmarked vertices w pointing from v

* program control-flow analysis using digraphs
  - vertex = basic block of instructions (straight-line program)
  - edge = jump
- dead-code elimination
- infinite-loop detection

* mark-sweep garbage collector

→ every data structure is a digraph

- vertex = object
- edge = reference

(Java)

DFS solution for:

- reachability
- path finding
- topological sort
- directed cycle detection

BFS in digraphs

- same method as for undirected graphs
- BFS is a digraph algorithm

BFS (from source vertex s)

Put (s) onto a FIFO queue, and mark s as visited. Repeat until the queue is empty.

- remove the least recently added vertex ✓

- for each unmarked vertex pointing from v: add to queue and mark as visited

* BFS computes shortest paths (fewest number of edges) from s to all other vertices in a digraph in time proportional to $E+V$

Multiple-source shortest paths

- given a digraph and a set of source vertices
- find shortest path from any vertex in the set to each other vertex
  - use BFS but initialize by enqueuing all source vertices

Web crawler using BFS

Topological Sort

- ~~Given a set of tasks to be completed with~~ precedence constraints, in which order should we schedule the tasks?

Model vertex = task

edge = precedence constraint

* draw the graph such as all the edges point upwards

* topological sort works on a **DAG**

= directed acyclic graph (no cycles)

Solution: DFS

- run DFS

- return vertices in reverse postorder ↗ stack

- **postorder** - the order in which we are done with the vertices

Proposition

Reverse DFS postorder of a DAG is a topological order.

* Consider any edge $v \to w$. When dfs(v) is called

case 1) dfs(w) has already been called and returned => w done before v

case 2) dfs(w) has not yet been called

dfs(w) will be called directly or indirectly by dfs(v) => will finish before dfs(v)

=> w done before v

case 3) dfs(w) has already been called but have not yet returned

- can't happen in a DAG: function call stack contains path from $w \to v$

=> $(v \to w)$ would complete a cycle

Proposition

- a digraph has a topological order if no directed cycle

Solution : DFS to find a directed cycle

* Java compiler does cycle detection

Strong Components

Def Vertices v and w are strongly connected if there is a directed path from v to w ($v \longrightarrow w$) and from w to v ($w \longrightarrow v$)

* strong connectivity is an equivalence relation
  - $v \longrightarrow v$
  - $v \longrightarrow w$ then $w \longrightarrow v$
  - $v \longrightarrow w$ and $w \longrightarrow x \Rightarrow v \longrightarrow x$

Def A strong component is a maximal subset of strongly-connected vertices

Software modules

vertex = software module

edge = from module to dependency

Strong component = subset of mutually interacting modules

Reverse graph — reverse the sense of all the edges => strong components in G are the same as in G^R

kernel DAG = contract each strong component into a single vertex (acyclic)

Idea

- compute topological order in kernel DAG
- Run DFS => considering vertices in reverse topological order

kosaraju - Sharir

phase 1) Compute reverse postorder in G^R

phase 2) Run DFS in G, visiting unmarked vertices in reverse postorder of G^R

* connected component code but first compute the Depth First Order of G^R and iterate through the reverse postorder of the vertices