CS 4150

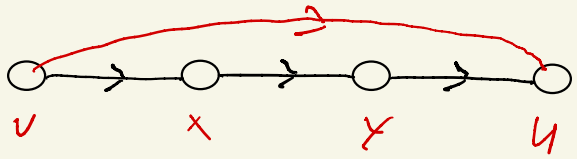
Wednesday, October 21, 2020

Outline

- Directed acyclic graphs
- Topological Sorting
- Longest Path

Directed Acyclic Graphs

- A directed acyclic graph (DAG) is a directed graph with no directed cycles.



- Consequence: if $u \in \text{reach}(v)$, then $v \notin \text{reach}(u)$

  $\uparrow$

  set of vertices x

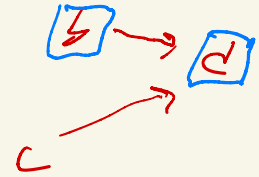
  s.t. $\exists$ directed vx-path

Detecting Cycles Ideas

Let's suppose we have a directed graph G.
How to tell if it's acyclic.

- DFS + revisiting a visited vertex ⇒ cycle

"active"



- track how many times visited, if there's a cycle this count is unbounded

Detecting Cycles Recursive

```
def isAcyclic(G):
    for v in V(G):
        v.status = new

    for v in V(G):
        if v.status = new:
            if not isAcyclicDFS(G, v):
                return false

    return true
```

```
def isAcyclicDFS(G, v):
    v.status = active
    for v→u in E(G):
        if u.status = active:
            return false
        if u.status = new:
            if not isAcyclicDFS(G, u):
                return false
    v.status = finished
    return true
```

Detecting Cycles Iterative

```
def isAcyclic(G):
    visited = set()
    for v in V(G):
        if v in visited:
            continue
        if not isAcyclicDFS(G, v, visited):
            return false

    return true
```

```
def isAcyclicDFS(G, v, visited):
    active = set()
    stack = stack()
    stack.push(v)
    while not stack.empty():
        curr = stack.pop()
        if curr in active:
            return false
        if curr in visited:
            continue
        active.add(curr)
        visited.add(curr)
        for curr → next in E(G):
            stack.push(next)

    return true
```

Review: Relations and Partial Orders

- A relation is a set $R \subseteq A \times B$ of pairs (a, b) which relate to eachother.

  $\leq, \geq, =, <, >$ mainly interested in the case where $A = B$

- reflexive, irreflexive, symmetric, anti-symmetric, asymmetric, transitive

  $\forall x \in A$ $x R x$

  $\forall x$ not $x R x$

  $aRb \Rightarrow bRa$

  aRb and $bRa \Rightarrow a = b$

  $aRb \Rightarrow$ not bRa

  aRb and $bRc \Rightarrow aRc$

$R \subseteq A \times A$

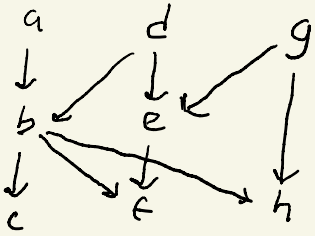
- A partial order of S is a relation $R \subseteq S \times S$ which is reflexive, anti-symmetric, transitive "like $\leq$ or $\geq$"

Claim: reach(u) in a DAG is a partial order.

uRv iff $v \in$ reach(u)

Topological Sorting

- Goal: given a relation $R \subseteq S \times S$, sort the elements of S ("order") such that $index(u) > index(v) \Rightarrow$ not uRv
- In a DAG, sort the vertices so that edges point from left to right



a d g b e c f h

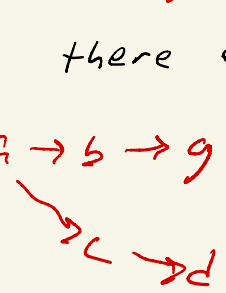
a d b c g h e f

Claim: DAGs can be topologically sorted

true because DAGs are a partial orders

Topological Sorting

Claim: $u.post < v.post$ (for any edge $u \to v$) implies there exists a directed vu-path in G.

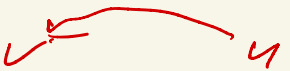


Corollary: for any edge $u \to v$ in a DAG G, $u.post > v.post$.

$u.post < v.post \Rightarrow$ directed vu-path by claim

$\Rightarrow$ directed cycle

cannot be true



In a postorder of G, every edge points right to left.

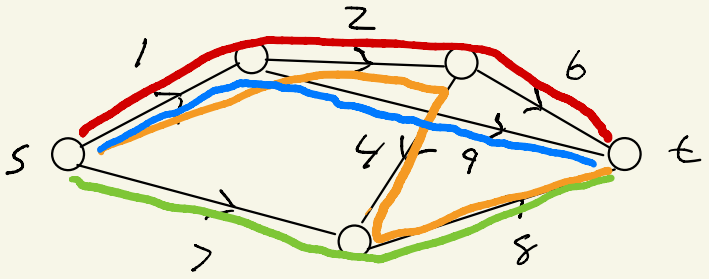
If we take a reversed postorder, all edges point left to right.

Dynamic Programming in DAGs

- Compute the longest st-path in a DAG with edge weights $w(e)$.

  The length of a path P is $\sum_{e \in P} w(e)$.

Given s and t



Red path: 9

Green path: 15] both optimal

Orange path: 15]

Blue path: 10

$$\text{longest path}(s, t, G) = \begin{cases} 0 & \text{if } s = t \\ \max_{s \to v} \text{longest path}(v, t, G) + w(sv) & \text{otherwise} \end{cases}$$

Longest Path

```
def LongestPath(G, s, t):
    for v in post(G):
        if v = t:
            v.longest = 0
        best = -∞
        for v → u in E(G):
            best = max(best, u.longest + w(vu))
        v.longest = best
    return s.longest
```

takes $O(V+E)$

Time complexity: $O(V+E)$

naive → $O(EV)$

✗ wrong, too bfg

computes max

All DP algorithms are implicitly DP on DAGs

- Let V be the set of possible arguments.
- Add edge from $a \to b$ if computing $f(a)$ requires computing $f(b)$.

vertex set

edge set

Called dependency graph.