DP in DAGs and Strongly Connected Components

Lecture 17

March 27, 2025

Part I

Dynamic Programming and DAGs

Recall LIS

Longest Increasing Subsequence problem: given $A[1..n]$, find the longest subsequence of A where each element is larger than the last.

Subproblem: $LIS(i)$ is the length of the longest increasing subsequence of A whose first element is $A[i]$.

Recurrence: $LIS(i) = 1 + \max\{LIS(j) \mid j > i \text{ and } A[j] > A[i]\}$
(Define $\max \emptyset = 0$)

Consider the *subproblem dependency graph*:

- Vertex i corresponds to subproblem $LIS(i)$
- Include edge (i, j) if computing $LIS(i)$ uses the value of $LIS(j)$
  ("Subproblem i depends on subproblem j")

Subproblem Dependency Graph Example

Suppose $A = [3, 1, 4, 1, 5]$.

(Recall $LIS(i) = 1 + \max\{LIS(j) \mid j > i \text{ and } A[j] > A[i]\}$)

DP is DAGs

For *every* DP algorithm, the subproblem dependency graph is a DAG.

An evaluation order is valid *if and only if* it is a reverse topological order of the dependency graph.

DAGs Support DP

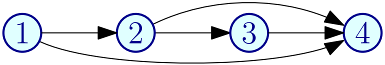
This connection works both ways, allowing us to apply DP to interesting problems on DAGs!

General outline:

- For each vertex v, define some subproblem corresponding to v.
- Write a recurrence for your subproblems. (Most often v will refer to the subproblems of either its parents or its children.)
- Evaluate the subproblems in (reverse) topological order.

Example of DAG DP

Given an (unweighted) DAG G in topological order, what is the length of the longest path in G?



Subproblem definition?

Recurrence?

Evaluation order?

Longest Path in a DAG

```
DAG-LongestPath(G):
    Initialize array LLP
    for vertices v in reverse topological order:
        if v is a sink:
            LLP[v] = 0
        else
            LLP[v] = 1 + max{LLP(u) | (v, u) ∈ E}
    return max_v LLP(v)
```

Efficiency? $O(V + E)$

Exercise: generalize this algorithm to

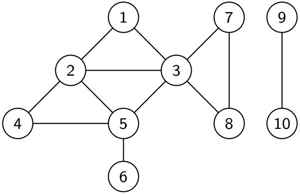
- Weighted edges
- Paths from s to t
- *Shortest* path from s to t
- ...

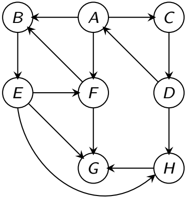
Part II

Strongly Connected Components

Recall Connected Components

In an *undirected* graph, we defined the connected component containing u as the set of all vertices it can reach.



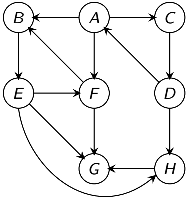


In a *directed* graph this relation isn't symmetric, so we can't talk about the "connected components of G".

Strongly Connected Components

Definition

The ***strongly connected component*** (SCC) of a vertex v is the set of all vertices u such that v can reach u and u can reach v.

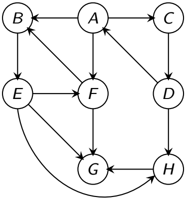


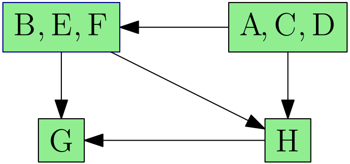
This definition doesn't depend on what vertex we choose within the SCC, so we can talk about the SCCs of G!

The Meta-Graph

Once we have the SCCs of G, we can construct a "meta graph":

- A vertex for each SCC.
- An edge (C_1, C_2) iff there is an edge in G between *some* vertex in C_1 and *some* vertex in C_2.

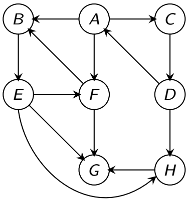




Meta graph will always be a DAG! (Also called the "DAG of SCCs".)

Warm-Up: Compute One SCC

How would we compute the SCC of a particular vertex? (Say B)



Naïve SCC Algorithm

```
FindSCC(G, v):
    Construct ‘‘reverse graph’’ G^R (reverse all edges)
    Compute reach(v) = WFS(G, v)
    Compute reachable(v) = WFS(G^R, v)
    Return reach(v) ∩ reachable(v)
```

```
NaïveSCC(G):
    while G is non-empty:
        Pick an arbitrary vertex v from G
        Compute FindSCC(G, v), removing it from G
    return list of SCCs found
```

Efficiency? $O(V+E)$ for `FindSCC`, $O(V \cdot (V+E))$ for `NaïveSCC`

Can we do better?

Making Improvements

Inefficiency: may have to explore the whole graph, even if v's connected component is small.

If v is in a sink SCC, only have to explore that!

```
GoalSCC(G):
    while G is non-empty:
        Pick a vertex v in a sink SCC
        Compute v's SCC from WFS(G, v), removing it from G
    return list of SCCs found
```

Efficiency? $O(V + E)$

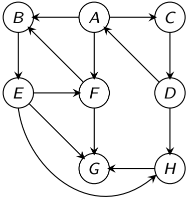
How do we find a vertex in a sink SCC though?

First Attempt: Mimic DAG Strategy

In a DAG, the first vertex in the `post`-order is always a sink.

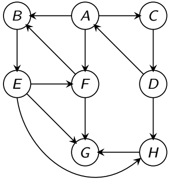
First attempt: is the first vertex in the `post`-order of an arbitrary graph guaranteed to be in a sink SCC?

No :(



Second Attempt: Large `post` Values

What if we look at the *last* vertex in the `post`-order?



Claim: in this graph, *must* be ***A***, ***C***, or ***D***.

Vertices ***A***, ***C***, and ***D*** aren't reachable from anywhere else, so `DFSAll` must (at some point) start a DFS run from one of them.

But *everything* is reachable from ***A***, ***C***, and ***D***, so that DFS run finds all remaining vertices, making it the last one.

Finding Sources

Claim

The last vertex in any `post`-order must be in a source SCC.

`DFSAll` must (at some point) start a run of DFS from *each* source SCC, as they are unreachable from anywhere else.

Every vertex is reachable from some source SCC, so once we've explored from them all we're done.

So the *last* run of DFS must start from a source SCC!

Sources to Sinks

We want to find a sink, but only know how to find a source—how can we bridge this gap?

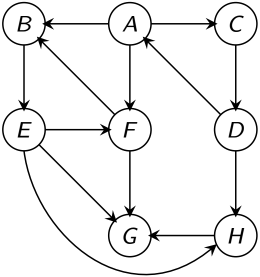
If we reverse the edges of G, sinks become sources!

In particular, a vertex is in a sink SCC of G if and only if it is in a source SCC of G^R.

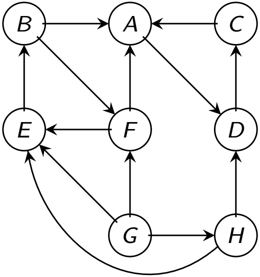
```
KosarajuSharirSCC(G):
    Construct G^R by reversing all edges
    Run DFSAll(G^R)
    for vertices v in decreasing order of post:
        if v is still in G:
            Find all vertices reachable from v with WFS
            Mark these as v's SCC and remove them from G
    return list of SCCs found
```

Efficiency?

Kosaraju-Sharir Example



Graph G



Graph G^R

Vertex	A	B	C	D	E	F	G	H
pre	1	7	3	2	9	8	13	14
post	6	12	4	5	10	11	16	15

Problems and Algorithms From Today

- Longest path in a DAG (weighted or unweighted)
  - Use DP, $O(V + E)$ time
  - Can also find shortest $s - t$ path

- Find the SCC of a single vertex
  - WFS on G and G^R, $O(V + E)$

- Find *all* SCCs of G
  - Kosaraju-Sharir, $O(V + E)$
  - Can also output the DAG of SCCs