

Note: these are just my own notes that I used in class. The material is mostly drawn from the Dasgupta book, chapters 6.1, 6.2, 6.4, 6.6 –jfh

At the end of last class, I posed the *all pairs shortest path problem*, and we got a few ideas. We'll return to this shortly.

Let's start by remembering TOPOLOGICAL SORT:

Observation: for a directed graph,

- acyclicity
- linearizability
- no back edges during BFS

are all equivalent.

Reason: do BFS on a graph. If there are back edges, you've got a cycle. If NOT, the decreasing "post" numbers \rightarrow linearization of graph. So acyclic \rightarrow linearizable, and no back edges \rightarrow acyclic. But linearizable surely means no back edges, so we're done.

Problem: how do you order things in decreasing "post" numbers? You computed the post numbers, but now do you have to sort them? That'd make the algorithm $|V| \log |V|$, rather than linear. Observation: "post" numbers get assigned in increasing order!

```
procedure topologicalSort(G)
  clock = 0
  s = new empty stack
  for all v in V :
    visited(v) = false
  for all v in V :
    if not visited(v): explore(v)
  return s;
```

```
procedure explore(G; v)
```

Input: $G = (V; E)$ is a graph; v in V

Output: visited(u) is set to true for all nodes u reachable from v

```
  visited(v) = true
  pre(v) = clock
  clock = clock + 1;
  for each edge (v; u) in E:
    if not visited(u):
      explore(u)
  post(v) = clock
  s.push(v);
  clock = clock + 1;
```

Alternative version: Observe that

- in a DAG, there are sources and sinks, and
- the node with lowest "post" number is a sink, and the one with highest "post" number is a source

```
procedure topologicalSort2(G)
```

```
using BFS, build a queue Q of all sources (and order)
```

```
L <- empty output list
```

```
while Q is not empty
```

```
  v = deq(Q)
```

```
  L.insert(v);
```

```
  foreach edge (v, u) in E
```

```
    E.remove(v, u)
```

```
    if u.indegree = 0
```

```
      Q.enq(u)
```

```
if E nonempty
```

```
  return "Graph has a cycle"
```

```
else
```

```
  return L;
```

Implications for shortest path in a DAG:

- (1) linearize (i.e., perform topological sort).
- (2) any antecedent of v has distance infity
- (3) descendents have distance that's a min.

```
proc DAGShortestPath(G, v)
```

```
L = topologicalSort(G);
```

```
forall u in V(G): dist(u) = infity
```

```
dist(v) = 0;
```

```
while (L not empty)
```

```
  u = extractFirst(L)
```

```
  for all (w, u) in E // all in-edges for u
```

```
    dist(u) = min[dist(u), dist(w) + length(w, u)]
```

Observations:

- It was easy to update dist(u) because we knew distances of all predecessors!
- We could have computed max-dist in the same way
- Finding distance for u depended on finding distance for predecessors ... i.e., we were solving subproblems that had the same structure as the original

Let's apply these ideas again: all pairs shortest path.

Insight: if you knew the shortest distance from any p to any q using *only* vertex v1 as an interior vertex (although

paths without an interior vertex are OK too), you could find the shortest distance from r to s using only v1 and v2 as interior vertices:

```
dist(r, s, 2) = min[dist(r, s, 1),
                    dist(r, v2, 1) + dist(v2, s, 1)]
```

This works in general!

We took the all-pairs distances problem and created SUBproblems, which are ORDERED. I.e., we created an implicit DAG. This comes under the heading of *dynamic programming*.

Hand out 7 node graph and ask for all pairs shortest distances.

Another dynamic programming example: Knapsack.

Room is on fire. You have a knapsack that can hold 10 lbs.

item	value	weight
T&G	\$40	5
Abba	\$90	4
Ortiz bobb	\$30	3
Socks	\$3	1

Idea: solve subproblems!

Given capacity $w = 0, 1 \dots 10$, and items $0, 1, 2, \dots, k$, which ones would you take? Fill in grid.

Another dynamic programming example: seam carving

Another example: longest increasing subsequence

4 1 7 5 2 5 8 6

Solution: draw all permissible arrows. It's a DAG. Path-in-dag = increasing subsequence. So just find longest path in dag.

```
for j = 1...n :
  L(j) = 1 + max[L(i) : (i, j) in E]
return max_j L(j)
```

[for the empty set, max is zero!]

Again, there are SUBPROBLEMS, they're ORDERED, and a relation that says how to solve a subproblem, given answers to "smaller" or "previous" subproblems.

Final example: Recursive function like "Fibonacci". As long as we have a way to INDEX the subproblems, we can make a memo of the result that we got in solving the subproblem.

```
Proc fib(n)
  If (n = 0 or 1) return 1
  Else return fib(n-1) + fib(n-2)
```

```
Proc fib(n)
  If (n = 0 or 1) return 1
  Else
    A = fib(n-1)
    B = fib(n-2)
    Return A + B
```

```
Proc fib(n)
  Set value[0...n] to Null

  If (n = 0 or 1) return 1
  Else
    A = fibH(n-1)
    B = fibH(n-2)
    Return A + B
```

```
Proc fibH(k)
  If value(k) is null
    Value(k) = fib(k)
  Return value(k)
```

This reduces $O(2^n)$ to $O(n)$.
Called "memorization"