COSC 290 Discrete Structures

Lecture 28: Relations, V: Topological Sort and DFS

Prof. Michael Hay

Wednesday, Nov. 8, 2017

Colgate University

Plan for today

1. Review: Hasse diagram
2. Data Representation
3. Algorithms for topological sort

Review: Hasse diagram

Recall: Hasse diagram

A partial order $\preceq$ on A can be drawn using a Hasse diagram.

- Draw nodes: one node for each A
- Draw edges: edge from a to b if $a \preceq b$, except...
- ... *omit* edges that can be inferred by reflexivity
- ... *omit* edges that can be inferred by transitivity
- ... and *layout* nodes "by level" if $a \preceq b$ for $a \neq b$, then a is placed *lower* than b

Example

A to do list,

$$[attendClass, sleep, borrowBook, eat, brushTeeth, study]$$

with constraints:

- $borrowBook \preceq study$
- $study \preceq attendClass$
- $sleep \preceq attendClass$
- $eat \preceq brushTeeth$
- $brushTeeth \preceq sleep$

(Hasse diagram shown on board.)

Topological ordering

Given a partial order $\preceq$, a topological ordering is a total order $\preceq_{total}$ that is *consistent* with $\preceq$.

Examples:

- borrowBook, study, eat, brushTeeth, sleep, attendClass
- eat, brushTeeth, sleep, borrowBook, study, attendClass
- eat, borrowBook, brushTeeth, study, sleep, attendClass
- ~~eat, brushTeeth, sleep, attendClass, borrowBook, study~~

Algorithms for topological ordering

Today, we will look at three algorithms:

- version 1: repeatedly find a minimal element
- version 2: same idea, more efficient than v1
- version 3: based on depth-first search (as efficient as v2)

Data Representation

Graph

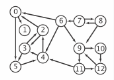

Figure 1: Simple graph (relation on $\{0, 1, \ldots, 12\}$).

Figure taken from Sedgewick and Wayne, Algorithms, 4th edition

Adjacency matrix

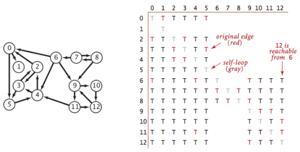

Figure 2: Adjacency matrix (on right) for the *transitive closure* of graph (on left).

In Java: `boolean[][] adjMatrix`

Figure taken from Sedgewick and Wayne, Algorithms, 4th edition

Adjacency list

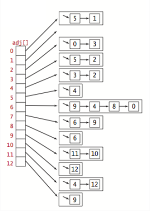

Figure 3: Adjacency list for previous graph.

In Java:

Using a Map:
`Map<Integer, List<Integer>> adjList`

or, using a *jagged array*:
`int[][] adjList`

- `adjList[i].length` equals number of neighbors of i
- `adjList[i][k]` gives the k^{th} neighbor of i

Figure taken from Sedgewick and Wayne, Algorithms, 4th edition

Algorithms for topological sort

Example

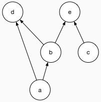

Figure 4: Example Hasse diagram

Version 1: repeatedly find minimal

Input: Partially ordered set P with partial order $\preceq$
Output: list of elements, representing total order consistent with $\preceq$

```
Initialize order to empty list
S := P
repeat n times
    X := findMinimal(S)
    choose any x from X
    S := S − {x}
    append x to order                ▷ put x at end of order
return order
```

Where *findMinimal*(S) returns a subset of $X \subseteq S$ of elements that are minimal with respect to S.

x is minimal with respect to S if $x \in S$ and $\forall y \in S - \{\, x \,\} : y \not\preceq x$.

(Apply to example on board.)

Poll: findMinimal

findMinimal(S) returns a subset of $X \subseteq S$ of elements that are minimal with respect to S.

x is minimal with respect to S if $x \in S$ and $\forall y \in S - \{\, x \,\} : y \not\preceq x$.

Consider the following sets...

- $X_1 := \mathit{findMinimal}(S)$
- $X_2 := \mathit{findMinimal}(S - \{\, x \,\})$ for some $x \in \mathit{findMinimal}(S)$

What must be true about X_1 and X_2? Choose the *best* answer:

A) $X_1 \subseteq X_2$
B) $X_2 \subseteq X_1$
C) $X_2 - X_1$ could contain any element in $S - \{\, x \,\}$
D) if $y \in X_2 - X_1$, then y is a neighbor of x

Version 2: same idea, more efficient

Input: Partially ordered set P with partial order $\preceq$
Output: list of elements, representing total order consistent with $\preceq$

```
Initialize order to empty list
Initialize count[x] to be number of incoming edges to x
Initialize X to be set of x where count[x] = 0        ▷ x is minimal
while X is not empty do
    remove any x from X
    append x to order                ▷ put x at end of order
    for for each neighbor y of x do
        count[y] := count[y] − 1
        if count[y] = 0 then  ▷ y is now minimal among remaining
            add y to X
return order
```

(Apply to example on board. *Why more efficient than version 1?*)

Depth-first search

```
function DFS(x, marked)
    marked[x] := true
    for for each neighbor y of x do
        if marked[y] = false then
            DFS(y, marked)
```

Apply to example:

- Initialize *marked*[x] := false for all x.
- DFS(e, *marked*)
- DFS(a, *marked*)

Poll: DFS behavior

```
function DFS(x, marked)
    marked[x] := true
    for for each neighbor y of x do
        if marked[y] = false then
            DFS(y, marked)
```

What does DFS do?

A) It marks x

B) It marks x and x's neighbors

C) It marks x and x's previously unmarked neighbors

D) It marks x and x's descendants

E) It marks x and x's previously unmarked descendants

Key property

Key property After the recursive calls on neighbors, *all* of x's descendants have been marked.

(Subtle point: some might have already been marked. How can this happen?)

Suppose we modify procedure to build up a total order as it goes. Thus, recursive calls on neighbors on x's neighbors add them to the order. Given above property, when/where should we add x?

Keeping track of topological order

```
function DFS(x, marked, order)              ▷ new parameter order
    marked[x] := true
    for for each neighbor y of x do
        if marked[y] = false then
            DFS(y, marked, order)
    prepend x to order                      ▷ put x at front of order
```

Why *prepend* to *order after* for loop? Why not *append* to order *before* for loop?

Incorrect version

This version does **not** work:

function BadDFS(x, *marked*, *order*)
    marked[x] $:=$ true
    append x to *order* ▷ put x at *end* of order
    for for each neighbor y of x **do**
        if *marked*[y] $=$ false **then**
            BadDFS(y, *marked*, *order*)

Apply to example:

- Initialize *marked*[x] $:=$ false for all x.
- BadDFS(a, *marked*) and suppose d appears before b in a's list of neighbors

Why this doesn't work: Before the recursive calls on neighbors (e.g., neighbors of b), it is not necessarily true that *no descendants* appear in *order*. Some might already be there (e.g., d already there). Thus, it is not safe to add to end of *order*.

Putting it all together

Input: Partially ordered set P with partial order $\preceq$
Output: list of elements, representing total order consistent with $\preceq$

Initialized *marked* to all false.
for each $x \in P$ **do**
    if *marked*[y] $=$ false **then**
        DFS(x, *marked*, *order*)
return *order*
function DFS(x, *marked*, *order*)
    marked[x] $:=$ true
    for for each neighbor y of x **do**
        if *marked*[y] $=$ false **then**
            DFS(y, *marked*, *order*)
    prepend x to *order* ▷ put x at *front* of order