Poll: Attendance

Today is

A) Monday
B) Tuesday
C) Wednesday
D) Thursday
E) Friday

COSC 290 Discrete Structures

Lecture 20: Graph search continued

Prof. Michael Hay
Friday, Mar. 23, 2018
Colgate University

Plan for today

1. Directed acyclic graphs and topological sort
2. Algorithms for topological sort

Directed acyclic graphs and topological sort

Recall: Directed Acyclic Graphs

Definition (Directed Acyclic Graph)

A directed acyclic graph (DAG) is a directed graph that contains no directed cycles.

Example

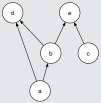

Figure 1: Example DAG

Topological ordering

Definition

Given a DAG, a topological ordering is an ordering of the vertices (a sequence) such that for every directed edge $(u, v) \in E$, vertex u comes before v in the ordering.

Example

Continuing the todo list example, a topological ordering is valid way to order the tasks such that all precedence constraints are obeyed.

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

are both valid topological orderings.

Algorithms for topological ordering

Today, we will look at three algorithms:

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

Algorithms for topological sort

Version 1: repeatedly find minimal

Input: Directed graph $G = (V, E)$.
Output: list of vertices, in some topological order

Initialize *order* to empty list
$S := V$
while S is not empty **do**
  $X := \mathit{findMinimal}(S)$
  choose any x from X
  $S := S - \{x\}$
  append x to *order* ▷ put x at *end* of order
return *order*

Where $\mathit{findMinimal}(S)$ returns a subset of $X \subseteq S$ of vertices that are minimal with respect to S.

x is minimal with respect to S if $x \in S$ and $\forall y \in S - \{ x \} : (y, x) \notin E$.

(Apply to example on board.)

Poll: Reasoning about findMinimal

$\mathit{findMinimal}(S)$ returns a subset of $X \subseteq S$ of vertices that are minimal with respect to S. x is minimal with respect to S if $x \in S$ and $\forall y \in S - \{ x \} : (y, x) \notin E$.

Let's consider how X changes in each iteration of previous algorithm. Let's use $X^{(1)}$ to denote X in first iteration, $X^{(2)}$ to denote X in second iteration, etc. So, for example,

- $X^{(1)} := \mathit{findMinimal}(V)$
- $X^{(2)} := \mathit{findMinimal}(V - \{ x \})$ for some $x \in \mathit{findMinimal}(V)$

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 vertex in $V - \{ x \}$
D) if $y \in X^{(2)} - X^{(1)}$, then y is a neighbor of x
E) More than one of above

Version 2: same idea, more efficient

Input: Directed graph $G = (V, E)$.
Output: list of vertices, in some topological order

Initialize *order* to empty list
Let $\mathit{count}[x] := |\{ y \in V : (y, x) \in E \}|$. ▷ count x's *incoming* edges
Initialize $X := \{ x \in V : \mathit{count}[x] = 0 \}$. ▷ 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 each $y \in V$ such that $(x, y) \in E$ **do** ▷ y is x's neighbor
    $\mathit{count}[y] := \mathit{count}[y] - 1$
    if $\mathit{count}[y] = 0$ **then** ▷ y is now minimal among remaining
      $X := X \cup \{ y \}$ ▷ add y to X
return *order*

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

Poll: Reasoning about version 2

Input: Directed graph $G = (V, E)$.
Output: list of vertices, in some topological order

Initialize *order* to empty list
Let $\mathit{count}[x] := |\{ y \in V : (y, x) \in E \}|$.
Initialize $X := \{ x \in V : \mathit{count}[x] = 0 \}$.
while X is not empty **do**
  remove any x from X
  append x to *order*
  for each $y \in V$ such that $(x, y) \in E$ **do**
    $\mathit{count}[y] := \mathit{count}[y] - 1$
    if $\mathit{count}[y] = 0$ **then**
      $X := X \cup \{ y \}$
return *order*

Initially the count for each vertex x is equal to x's in-degree (number of incoming edges). Vertex x will only be added to the order when its count reaches zero. Suppose the initial count for vertex y is 2 and for vertex y' is 1. Which of the following statements is most accurate about the final order?

A) y appears before y'
B) y' appears before y
C) y may appear before y'
D) y' may appear before y
E) more than one of above

Version 3

Version 3 is based on depth-first search.

It is as efficient as version 2.

Depth-first search

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

What does a single invocation of DFS(x, marked) 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.

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 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 each neighbor y of x do
        if marked[y] = false then
            BadDFS(y, marked, order)
```


Poll: reasoning about BadDFS

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

Suppose *marked*[*x*] := false for all *x*. Then we call BadDFS(*a*, *marked*). Assuming that *d* appears before *b* in *a*'s list of neighbors, what does the algorithm output?

A) a, b, d, e

B) a, b, d, e, c

C) a, d, b, e

D) a, d, b, e, c

E) More than one / None of above

Good vs. bad

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

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

DFS is correct; BadDFS is not. Why does DFS work?

- Suppose *marked* initialized to all false, and we call DFS(x_0, *marked*, *order*) for some x_0. Computation ensues and now, for some *x*, DFS(*x*, *marked*, *order*) is about to be executed.
- Consider *x*'s ancestors: have they been added to *order*?
- Consider *x*'s descendants: have they been added to *order*?