

## Poll: Attendance

Today is

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

1

## 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

2

## 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

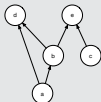


Figure 1: Example DAG

3

## 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.

4

## 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)

5

## Algorithms for topological sort

## Version 1: repeatedly find minimal

**Input:** Directed graph  $G = (V, E)$ .

**Output:** list of vertices, in some topological order

- 1: Initialize *order* to empty list
- 2:  $S := V$
- 3: **while**  $S$  is not empty **do**
- 4:    $X := \text{findMinimal}(S)$
- 5:   choose any  $x$  from  $X$
- 6:    $S := S - \{x\}$
- 7:   append  $x$  to *order* ▷ put  $x$  at end of order
- 8: **return order**

Where  $\text{findMinimal}(S)$  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.)

6

## Poll: Reasoning about findMinimal

$\text{findMinimal}(S)$  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,

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

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

7

## Version 2: same idea, more efficient

**Input:** Directed graph  $G = (V, E)$ .

**Output:** list of vertices, in some topological order

- 1: Initialize *order* to empty list
- 2: Let  $\text{count}[x] := |\{y \in V : (y, x) \in E\}|$ . ▷ count  $x$ 's incoming edges
- 3: Initialize  $X := \{x \in V : \text{count}[x] = 0\}$ . ▷  $x$  is minimal
- 4: **while**  $X$  is not empty **do**
- 5:   remove any  $x$  from  $X$
- 6:   append  $x$  to *order* ▷ put  $x$  at end of order
- 7:   **for each**  $y \in V$  such that  $(x, y) \in E$  **do** ▷  $y$  is  $x$ 's neighbor
- 8:      $\text{count}[y] := \text{count}[y] - 1$
- 9:     **if**  $\text{count}[y] = 0$  **then** ▷  $y$  is now minimal among remaining
- 10:        $X := X \cup \{y\}$  ▷ add  $y$  to  $X$
- 11: **return order**

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

8

## Poll: Reasoning about version 2

**Input:** Directed graph  $G = (V, E)$ .

**Output:** list of vertices, in some topological order

- 1: Initialize *order* to empty list
- 2: Let  $\text{count}[x] := |\{y \in V : (y, x) \in E\}|$ .
- 3: Initialize  $X := \{x \in V : \text{count}[x] = 0\}$ .
- 4: **while**  $X$  is not empty **do**
- 5:   remove any  $x$  from  $X$
- 6:   append  $x$  to *order*
- 7:   **for each**  $y \in V$  such that  $(x, y) \in E$
- 8:      $\text{count}[y] := \text{count}[y] - 1$
- 9:     **if**  $\text{count}[y] = 0$  **then**
- 10:        $X := X \cup \{y\}$
- 11: **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

9

## Version 3

Version 3 is based on **depth-first search**.

It is as efficient as version 2.

10

## Depth-first search

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

Apply to example:

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

11

## Poll: DFS behavior

```
1: function DFS(x, marked)
2:   marked[x] := true
3:   for each neighbor y of x do
4:     if marked[y] = false then
5:       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

12

## 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*?

13

## Keeping track of topological order

```
1: function DFS(x, marked, order)           ▷ new parameter order
2:   marked[x] := true
3:   for each neighbor y of x do
4:     if marked[y] = false then
5:       DFS(y, marked, order)
6:   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?

14

## Incorrect version

This version does **not** work:

```
1: function BADDFS(x, marked, order)
2:   marked[x] := true
3:   append x to order                       ▷ put x at end of order
4:   for each neighbor y of x do
5:     if marked[y] = false then
6:       BADDFS(y, marked, order)
```

15

## 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

16

## 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*<sub>0</sub>, *marked*, *order*) for some *x*<sub>0</sub>. 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*?

17

## Wrapping up

Version 3, which uses DFS to help with topological sort, is wrapped up at the beginning of the next class.