

MA/CSSE 473

Day 19



BFS

Topological Sort

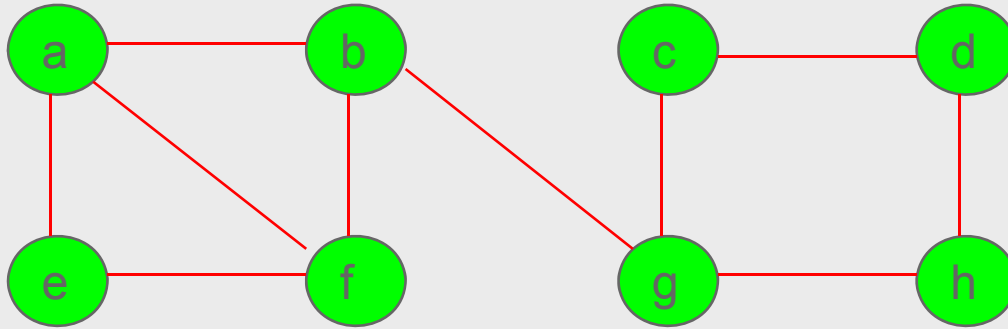
**Combinatorial
Object Generation**

MA/CSSE 473 Day 19

- HW 8 due tomorrow
- Don't forget the quickhull implementation problem, due next Monday
- HW 10 will be due on Wednesday (Oct 15, the day before break) at noon, instead of on Tuesday
- Due to break, there will be no HW due on Tuesday, Oct 21. HW 11 will be due Friday.
- HW 11 will be due Friday
- **Student Questions**
- DFS and BFS
- Topological Sort
- Combinatorial Object Generation



Example: DFS traversal of undirected graph



DFS traversal stack:

DFS tree:



Notes on DFS

- DFS can be implemented with graphs represented as:
 - adjacency matrix: $\Theta(V^2)$
 - adjacency list: $\Theta(|V| + |E|)$
- Yields two distinct ordering of vertices:
 - order in which vertices are first encountered (pushed onto stack)
 - order in which vertices become dead-ends (popped off stack)
- Applications:
 - checking connectivity, finding connected components
 - checking acyclicity
 - finding articulation points
 - searching state-space of problems for solution (AI)



Breadth-first search (BFS)

- Visits graph vertices by moving across to all the neighbors of last visited vertex
- Instead of a stack, BFS uses a queue
- Similar to level-order tree traversal
- “Redraws” graph in tree-like fashion (with tree edges and cross edges for undirected graph)



Pseudocode for BFS

ALGORITHM *BFS*(*G*)

//Implements a breadth-first search traversal of a given graph

//Input: Graph $G = \{V, E\}$

//Output: Graph G with its vertices marked with consecutive integers

//in the order they have been visited by the BFS traversal

mark each vertex in V with 0 as a mark of being “unvisited”

count \leftarrow 0

for each vertex v in V **do**

if v is marked with 0

bfs(v)

bfs(v)

//visits all the unvisited vertices connected to vertex v by a path

//and assigns them the numbers in the order they are visited

//via global variable *count*

count \leftarrow *count* + 1; mark v with *count* and initialize a queue with v

while the queue is not empty **do**

for each vertex w in V adjacent to the front vertex **do**

if w is marked with 0

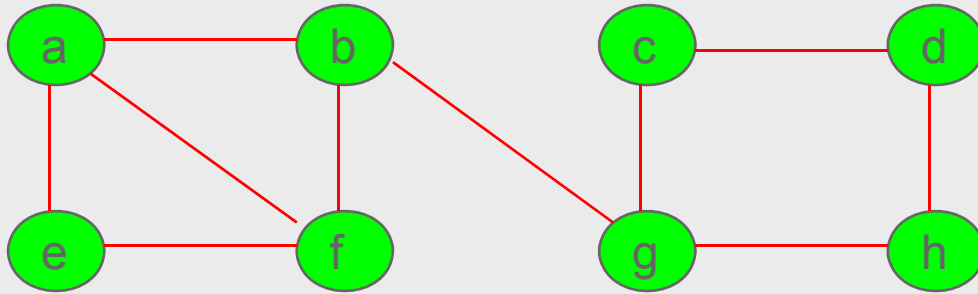
count \leftarrow *count* + 1; mark w with *count*

 add w to the queue

 remove the front vertex from the queue



Example of BFS traversal of undirected graph



BFS traversal queue:

BFS tree:



Notes on BFS

- BFS has same efficiency as DFS and can be implemented with graphs represented as:
 - adjacency matrices: $\Theta(V^2)$
 - adjacency lists: $\Theta(|V| + |E|)$
- Yields single ordering of vertices (order added/deleted from queue is the same)
- Applications: same as DFS, but can also find shortest paths (smallest number of edges) from a vertex to all other vertices



DFS and BFS

TABLE 5.1 Main facts about depth-first search (DFS) and breadth-first search (BFS)

	DFS	BFS
Data structure	stack	queue
No. of vertex orderings	2 orderings	1 ordering
Edge types (undirected graphs)	tree and back edges	tree and cross edges
Applications	connectivity, acyclicity, articulation points	connectivity, acyclicity, minimum-edge paths
Efficiency for adjacent matrix	$\Theta(V ^2)$	$\Theta(V ^2)$
Efficiency for adjacent lists	$\Theta(V + E)$	$\Theta(V + E)$



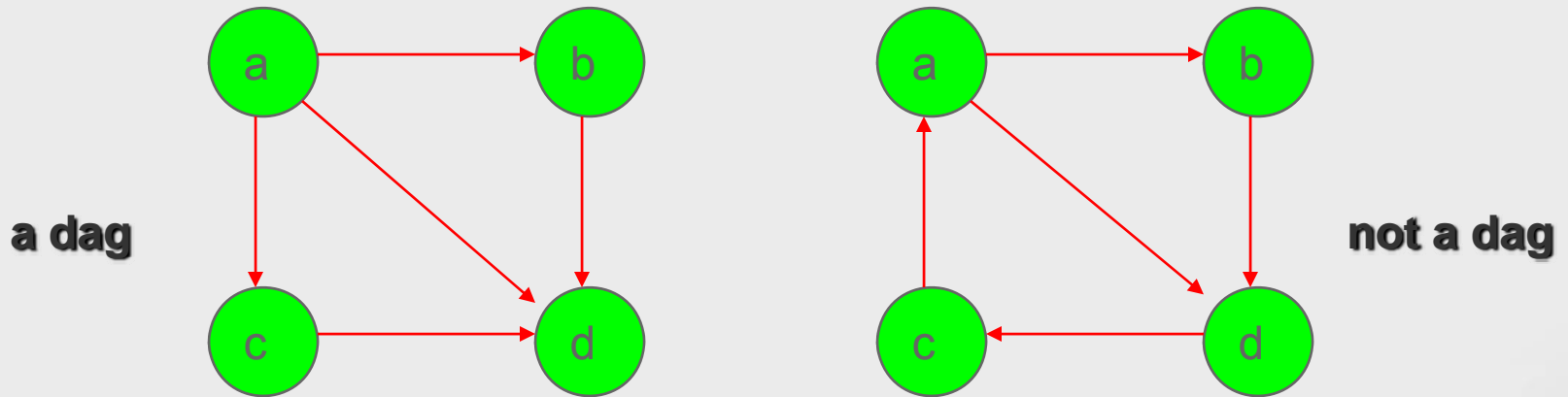
Directed graphs

- In an undirected graph, each edge is a "two-way street".
 - The adjacency matrix is symmetric
- In an directed graph, each edge goes only one way.
 - (a,b) and (b,a) are separate edges.
 - One can be in the graph without the other.



Dags and Topological Sorting

A dag: a directed acyclic graph, i.e. a directed graph with no (directed) cycles



Dags arise in modeling many problems that involve prerequisite constraints (construction projects, document version control, compilers)

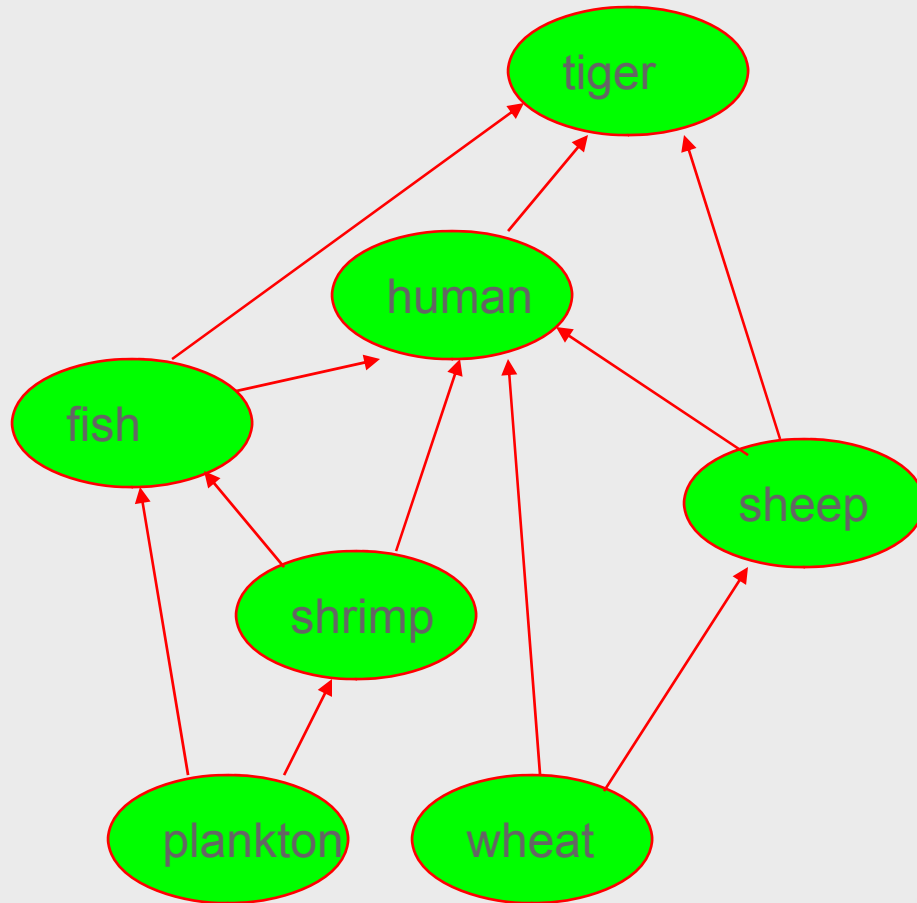
Vertices of a dag can be linearly ordered so that for every edge its starting vertex is listed before its ending vertex (topological sorting).

A graph must be a dag in order for topological sorting to be possible.



Topological Sorting Example

Order the following items in a food chain

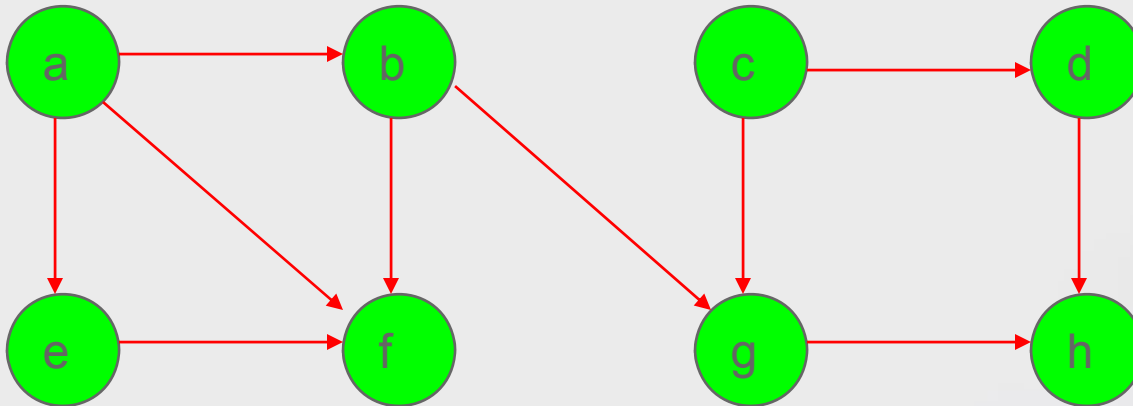


DFS-based Algorithm

DFS-based algorithm for topological sorting

- Perform DFS traversal, noting the order vertices are popped off the traversal stack
- Reverse order solves topological sorting problem
- Back edges encountered? → NOT a dag!

Example:



Efficiency:

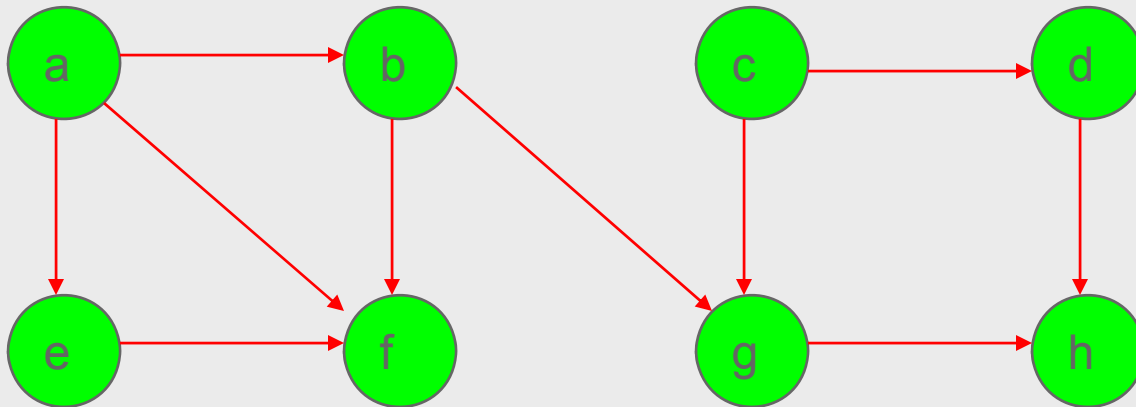


Source Removal Algorithm

Source removal algorithm

Repeatedly identify and remove a *source* (a vertex with no incoming edges) and all the edges incident to it until either no vertex is left (problem is solved) or there is no source among remaining vertices (not a dag)

Example:



Efficiency: same as efficiency of the DFS-based algorithm



Application: Spreadsheet program

- What is an allowable order of computation of the cells' values?

	A	B	C
1	=C4-7	4	=C4+6
2	=A3+A1-C4	=1+B1	=B1-A4
3	7	=A3*C2-B2	=B3+A3
4	=A1*B1*A2	=C2-A4	9



Combinatorial Object Generation

- Generation of permutations, combinations, subsets.
- This is a big topic in CS
- Evidence: The book I received on Saturday
 - *The Art of Computer Programming, Volume 4, Fascicle 2*
 - *Generating All Tuples and Permutations*
 - *127 pages of very small print.*
- *We will just scratch the surface of this subject.*



Permutations

- We generate all permutations of the numbers $1..n$.
 - Permutations of any other collection of n distinct objects can be obtained from these by a simple mapping.
- How would a "decrease by 1" approach work?
 - Find all permutations of $1.. n-1$
 - Insert n into each position of each such permutation
 - We'd like to do it in a way that minimizes the change from one permutation to the next.
 - It turns out we can do it so that we always get the next permutation by swapping two adjacent elements.



First approach we might think of

- for each permutation of $1..n-1$
 - for $i=0..n-1$
 - insert n in position i
- That is, we do the insertion of n into each smaller permutation from left to right each time
- However, to get "minimal change", we alternate:
 - Insert n L-to-R in one permutation of $1..n-1$
 - Insert n R-to-L in the next permutation of $1..n-1$
 - Etc.



Example

- Bottom-up generation of permutations of 123

start	1		
insert 2 into 1 right to left	12	21	
insert 3 into 21 right to left	123	132	312
insert 3 into 21 left to right	321	231	213

FIGURE 5.12 Generating permutations bottom up

- Example: Do the first few permutations for $n=4$



Johnson-Trotter Approach

- integrates the insertion of n with the generation of permutations of $1..n-1$
- Does it by keeping track of which direction each number is currently moving

→ ← → ←
3 2 4 1

The number k is **mobile** if its arrow points to an adjacent element that is smaller than itself.

- In this example, 4 and 3 are mobile
- We exchange the largest mobile number with its neighbor



Johnson-Trotter Driver

```
def main():  
    p = Permutation(4)  
    list = []  
    next = p.next()  
    while next:  
        list += [next]  
        next = p.next()  
    print list
```



Johnson-Trotter background code

```
left = - 1 # equivalent to the left- and  
right = 1  # right-pointing arrows in the book
```

```
def swap(list1, list2, i, j):  
    "Swap positions i and j in both lists"  
    list1[i], list1[j] = list1[j], list1[i]  
    list2[i], list2[j] = list2[j], list2[i]
```

```
class Permutation:  
    "Set current to the unpermuted list, and all directions pointing left"  
    def __init__(self, n):  
        self.current = range(1, n + 1)  
        self.direction = [left] * n  
        self.n = n  
        self.more = True # This is not the last permutation.
```



Johnson-Trotter major methods

```
def isMobile(self, k):
    ''' An element of a permutation is mobile if its direction "arrow"
        points to an element with a smaller value. '''
    return k + self.direction[k] in range(self.n) and \
           self.current[k + self.direction[k]] < self.current[k]

def next(self):
    "return current permutation and calculate next one"
    if not self.more:
        return False
    returnValue = [self.current[i] for i in range(self.n)]

    largestMobile = 0
    for i in range(self.n):
        if self.isMobile(i) and self.current[i] > largestMobile:
            largestMobile = self.current[i]
            largePos = i

    if largestMobile == 0:
        self.more = False # This is the last permutation
    else:
        swap(self.current, self.direction,
             largePos, largePos + self.direction[largePos])
        for i in range(self.n):
            if self.current[i] > largestMobile:
                self.direction[i] *= - 1

    return "".join([str(v) for v in returnValue])
```



Lexicographic Permutation Generation

- Generate the permutations in "natural" order.
- Let's do it recursively.

