MA/CSSE 473 Day 12

Insertions Sort quick review

DFS, BFS

Topological Sort

MA/CSSE 473 Day 12

- Changes to HW 6 (details: schedule page Day 15)
  - Due date postponed
  - Problems 9-11 added back in
- Questions?
- Depth-first Search
- Breadth-first Search
- Topological Sort
- (Introduce permutation and subset generation)

Some "decrease by one" algorithms

- Insertion sort
- Selection Sort
- Depth-first search of a graph
- Breadth-first search of a graph

Review: Analysis of Insertion Sort

- Time efficiency

  $C_{worst}(n) = n(n\text{-}1)/2 \in \Theta(n^2)$

  $C_{avg}(n) \approx n^2/4 \in \Theta(n^2)$

  $C_{best}(n) = n - 1 \in \Theta(n)$

  (also fast on almost-sorted arrays)
- Space efficiency: in-place

  (constant extra storage)
- Stable: yes
- Binary insertion sort (HW 6)
  - use Binary search, then move elements to make room for inserted element

Graph Traversal

Many problems require processing all graph vertices (and edges) in systematic fashion

Most common Graph traversal algorithms:

- Depth-first search (DFS)
- Breadth-first search (BFS)

Depth-First Search (DFS)

- Visits a graph's vertices by always moving away from last visited vertex to unvisited one, backtracks if no adjacent unvisited vertex is available
- Uses a stack
  - a vertex is pushed onto the stack when it's reached for the first time
  - a vertex is popped off the stack when it becomes a dead end, i.e., when there are no adjacent unvisited vertices
- "Redraws" graph in tree-like fashion (with tree edges and back edges for undirected graph)
  - A back edge is an edge of the graph that goes from the current vertex to a previously visited vertex (other than the current vertex's parent).

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)

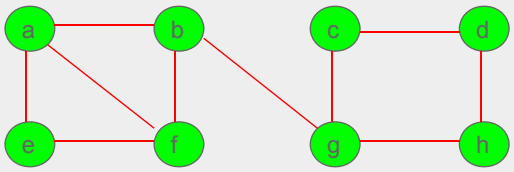
Pseudocode for DFS

ALGORITHM *DFS*(G)

```
//Implements a depth-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've been first encountered by the DFS traversal
mark each vertex in V with 0 as a mark of being "unvisited"
count ← 0
for each vertex v in V do
    if v is marked with 0
        dfs(v)

dfs(v)
//visits recursively all the unvisited vertices connected to vertex v by a path
//and numbers them in the order they are encountered
//via global variable count
count ← count + 1; mark v with count
for each vertex w in V adjacent to v do
    if w is marked with 0
        dfs(w)
```

Example: DFS traversal of undirected graph



DFS traversal stack:

DFS tree:

Breadth-first search (BFS)

- Visits graph vertices in increasing order of length of path from initial vertex.
- Vertices closer to the start are visited early
- Instead of a stack, BFS uses a queue
- Level-order traversal of a rooted tree is a special case of BFS
- “Redraws” graph in tree-like fashion (with tree edges and cross edges for undirected graph)

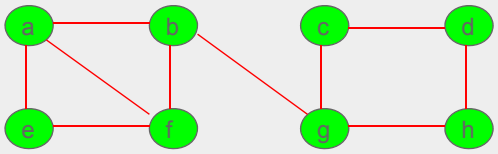
Pseudocode for BFS

```
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 ← 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 ← 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 ← count + 1;  mark w with count
                add w to the queue
        remove the front vertex from the queue
```

Note that this code is like DFS, with the stack replaced by a queue

Example of BFS traversal of undirected graph



BFS traversal queue:

BFS tree:

Q3

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 a single ordering of vertices (order added/deleted from the 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

Q4

DFS and BFS

TABLE 3.1 Main facts about depth-first search (DFS) and breadth-first search (BFS)

	DFS	BFS
Data structure	a stack	a queue
Number of vertex orderings	two orderings	one 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 adjacency matrix	$\Theta(\|V^2\|)$	$\Theta(\|V^2\|)$
Efficiency for adjacency lists	$\Theta(\|V\|+\|E\|)$	$\Theta(\|V\|+\|E\|)$

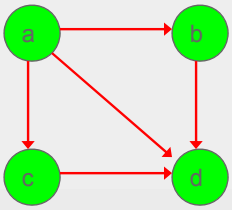
Directed graphs

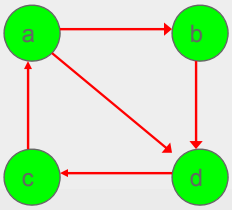
- In an undirected graph, each edge is a "two-way street".
  - The adjacency matrix is symmetric
- In an directed graph (digraph), each edge goes only one way.
  - (a,b) and (b,a) are separate edges.
  - One such edge can be in the graph without the other being there.

Dags and Topological Sorting

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

a dag





not a dag

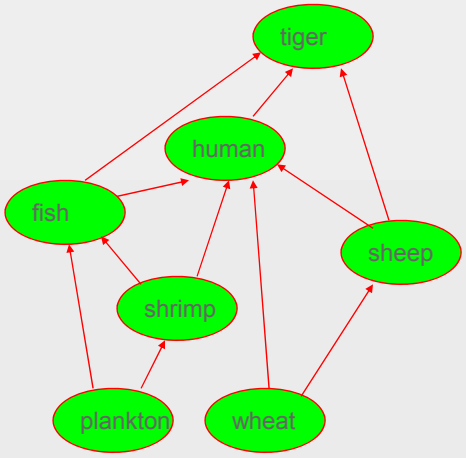
Dags arise in modeling many problems that involve prerequisite constraints (construction projects, document version control, compilers)

The vertices of a dag can be linearly ordered so that every edge's starting vertex is listed before its ending vertex (**topological sort**).

A graph must be a dag in order for a topological sort of its vertices to be possible.

Topological Sort Example

Order the following items in a food chain

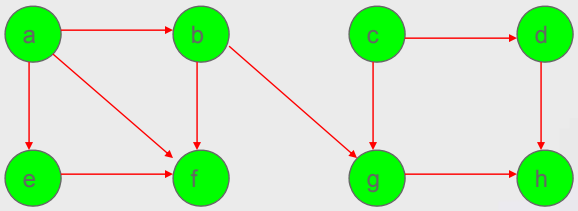


DFS-based Algorithm

DFS-based algorithm for topological sorting

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

Example:

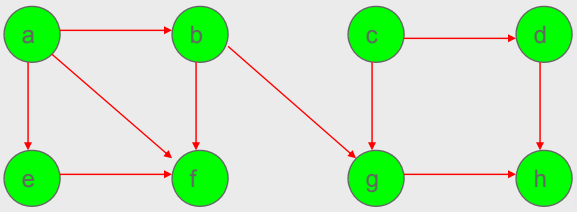


Efficiency:

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

Q8

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

Q9-11

Cycles cause a problem!

	A	B	C
1	=C2-7	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

(We may not get to this today)

Permutations

Subsets

COMBINATORIAL OBJECT GENERATION

Combinatorial Object Generation

- Generation of permutations, combinations, subsets.
- This is a big topic in CS
- We will just scratch the surface of this subject.
  - Permutations of a list of elements (no duplicates)
  - Subsets of a set

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

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