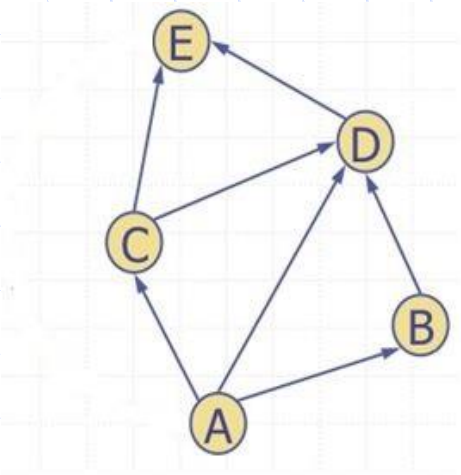
Lesson 12
Digraphs
Knowledge Has Organizing Power

Wholeness of the Lesson

Directed graphs are built on top of undirected graphs, making the underlying relationships in an undirected graph exhibit direction, flow, and a kind of dynamism, and consequently are more versatile tools for modeling the real world. Facts about directed graphs, and applications of them, make use of their underlying undirected structure as well as the dynamic features resulting from the presence of directed edges. Every undirected graph can be turned into a directed graph in many different ways; the undirected starting point represents pure relationship, pure knowledge, with a huge range of possibilities for dynamic expression, each of which potentially gives expression to dynamics of the manifest world. This phenomenon is an application of the principle that Knowledge Has Organizing Power – just the presence of underlying relationships, forming the content of knowledge, has the potential for a great variety of dynamic expression.

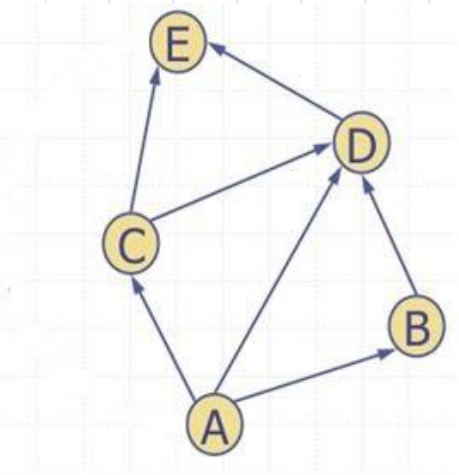
Overview of Digraphs

- A **digraph** is a graph whose edges are all directed
  - Represent x $\rightarrow$ y by the ordered pair (x,y)
  - y is called the out vertex for x
  - and x is called the in vertex for y
- Applications
  - one-way streets
  - flights
  - task scheduling
- Still use notation G = (V, E) but now
  - Each edge goes in one direction:
    - Edge (a,b) goes from a to b, but not b to a.
- If G is simple, m $\leq$ n (n - 1).
- *Implementation Adjustment:* We keep in-vertices and out-vertices in separate adjacency lists.



Overview of Digraphs - Continued

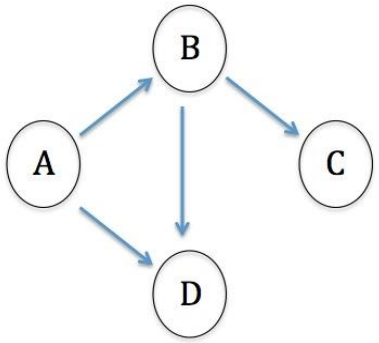
- In a directed graph, for any $v \in V$,
  - indeg(v) = $|\{w \mid (w,v) \in E\}|$
  - outdeg(v) = $|\{w \mid (v,w) \in E\}|$
  - $\sum_v$ indeg(v) = $\sum_v$ outdeg(v) = |E|
- In a digraph, a *directed path* from vertex v to vertex w is a path from v to w in which all edges are directed forward.
  Example:
  C-D-E is a directed path from C to E
- Vertex w is *reachable* from vertex v if there is a directed path from v to w. Example: E is reachable from C but A is not reachable from C.
  Note: Any vertex is reachable from itself (via the empty path).



DFS for Digraphs

- The DFS algorithm works the same as in the undirected case, except that the task of examining unvisited adjacent vertices for a vertex v is replaced by examining unvisited *out vertices* for a vertex v.

- Also, DFS in the directed case marks all vertices that are *reachable* from the starting vertex. Example:



Here, if starting vertex is A, DFS marks all vertices.
If starting vertex is B, DFS marks only B, D, C (an outer loop would need to search for next unvisited vertex in order to mark the vertex A)

DFS for digraphs

Algorithm: Depth First Search (DFS) (for digraphs)
Input: A simple *directed* graph G = (V,E), start vertex *s*
Output: G, with all vertices reachable from *s* marked as visited.

```
Initialize a stack S   //supports backtracking
Mark s as visited
S.push(s)
while S ≠ ∅ do
    v ← S.peek()
    if some out vertex of v not yet visited then
        w ← next out vertex of v (i.e. (v,w) in E)
        mark w
        push w onto S
    else //if can't find such a w, backtrack
        S.pop()
```

Running time

- We show the running time is O(n+m). Every vertex eventually is marked and pushed onto the stack, and then is eventually popped from the stack, and each of these occurs only once. Therefore, each vertex undergoes O(1) steps of processing.
- In addition, each vertex v will experience a peek operation, at which time the algorithm will search for an unvisited out vertex of v. This peek step, together with the search, will take place repeatedly until every out vertex of v has been visited – in other words, outdeg(v) times.
- It is conceivable that repeatedly searching the entire list of vertices adjacent to v (obtained from the adjacency list) could be costly, but, as in the undirected case, we arrange it so that each vertex in the list of vertices adjacent to v is accessed exactly one time during the entire component loop.

  Therefore, for each v, O(1) + O(outdeg(v)) steps are executed. The sum over all v in V is

  $$\mathbf{O(\sum_v (1 + outdeg(v))) = O(n + m)}$$

Applications of DFS for Digraphs

- Using DFS for digraphs, it is possible to compute the following in O(n + m) time:
  - All vertices reachable from a given starting vertex
  - Whether vertex v can be reached from vertex w, for any v, w.
  - A *topological ordering* of the graph, *provided G has no directed cycles.* (A directed cycle is a directed path with no repeated edges for which the first and last vertex in the path are the same.)

Some Techniques

- Algorithms for using DFS to solve the above problems:

Problem: Find all vertices reachable from a given starting vertex s.

Solution: Perform DFS with starting vertex s, and make sure the implementation of DFS stores the list of vertices discovered during its traversal. Then return this list.

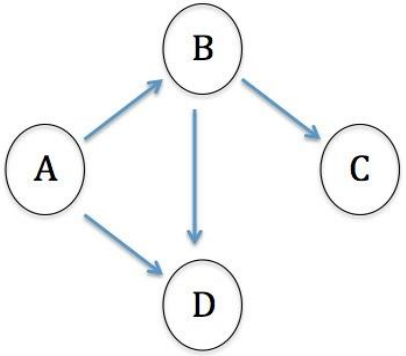
Problem: Given vertices v, w, determine whether v can be reached from w with a directed path.

Solution: Perform DFS with starting vertex w, get the list of visited vertices, and check whether v is in the list.

Topological Orderings

- **Idea.** A topological ordering is a way of organizing the vertices of G into a list in such a way that whenever (v,w) is a directed edge of G, v precedes w in the list.
- *Applications:*
  - Devise a schedule of classes subject to the constraint that certain classes have pre-requisites. Classes are represented as vertices and $v \rightarrow w$ if and only if class v is a pre-requisite for class w.
  - Organize a sequence of tasks so that whenever $task_1$ needs to be done before $task_2$, the output sequence ensures that this requirement is met. Tasks are represented as vertices and $v \rightarrow w$ only if task v must be executed before task w.
- In these applications, the pre-requisite requirements can be represented in a directed graph. The objective is to organize the vertices of the graph in a linear arrangement so that all the requirements are satisfied.

Topological Orderings



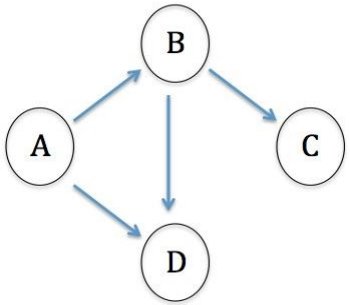
One solution: A, B, D, C

Another solution: A, B, C, D

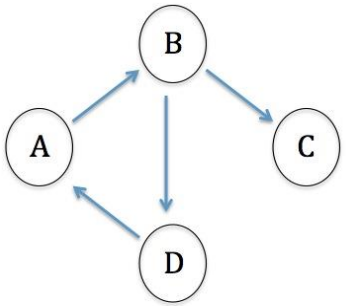
Not a solution: A, D, B, C

Topological Orderings

- **Definition**. Suppose G = (V,E) is a directed graph. A topological ordering of G is a function f: V -> N satisfying: for all (v,w) in E, f(v) < f(w).



1. A topological ordering:
   f(A) = 1, f(B) = 2, f(D) = 3, f(C) = 4
   - gives the ordering A, B, D, C
2. An assignment that is not a topological ordering:
   f(A) = 1, f(D) = 2, f(B) = 3, f(C) = 4
   - now, (B,D) is an edge, but f(B) > f(D)



No topological ordering is possible here because of the directed cycle A $\rightarrow$B $\rightarrow$ D $\rightarrow$ A: We must have f(A) < f(B) < f(D) < f(A), so f(A) < f(A).

Topological Sorting with DFS

- **Theorem.** A digraph admits a topological ordering if and only if it contains no directed cycle (iff it is a *directed acyclic graph (DAG)*).

Topological Sorting with DFS

Algorithm: TopSort

Input: A DAG G = (V, E), starting vertex s (having no in-vertices)

Output: An arrangement of the elements of V reachable from s, in topological order (may be less than n)

-- run DFS till a vertex v is encountered that has no unvisited out vertices (this is precisely where the stack is popped) and label v with n

-- continue running DFS till another v is encountered, with no unvisited out vertices, and label this vertex with n-1

-- continue until all vertices reachable from start vertex have been labeled

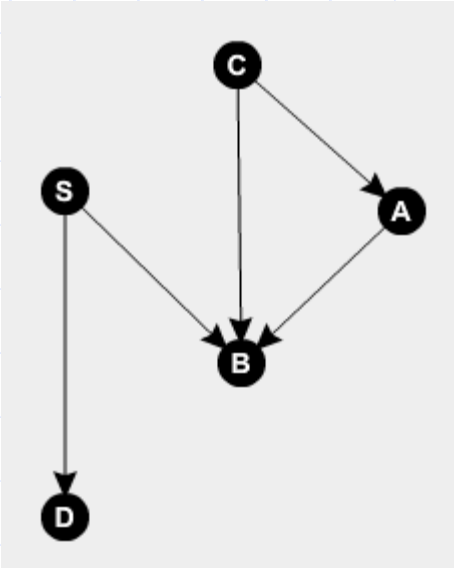
NOTE: In case a starting vertex is not provided as input, a starting vertex can be found by scanning in the in-list of the graph for a vertex that has zero in-vertices. Any such vertex can serve as the starting vertex. (This requires only O(n) extra work.)

Topological Sorting with DFS

- *Running Time of TopSort:* The only change from the usual DFS algorithm is that we assign numeric values to each vertex. Therefore, running time is O(n + m).

Reaching All Vertices with Topological Sorting

- The approach in previous slides will not topologically sort all vertices in a DAG, in general. (An example DAG on the right)
- Here, after the sort S, D, B is done (or it could be S, B, D), the vertices C, A remain unsorted. To handle these, we can seek another starting vertex having no in-vertices – in this case, the vertex C works; then we repeat the basic topological sort algorithm. Now we have two portions of the graph sorted: S, D, B and C, A.



Reaching All Vertices with Topological Sorting

- ***Efficient Discovery of Start Vertices.*** We do not want to repeatedly search for vertices with no in-vertices, since this could increase the running time by as much as n^2. Instead, we locate all the vertices with no in-vertices at the beginning of the algorithm, and place them in a stack.

Algorithm: GeneralTopSort

Input: A DAG G = (V, E)

Output: An arrangement of the elements of V in topological order

- create new stack S and scan the in-list for vertices having no in-vertices; for each one found, push onto S.
- while S is not empty
  - k <- number of unlabeled vertices remaining (starting value is n)
  - s <- S.pop()
  - run TopSort with start vertex s and start value k

Reaching All Vertices with Topological Sorting

- **Running Time**: The running time is the cost of initialization (finding list of vertices with no in-vertices), cost of each round of TopSort and the cost of finding the next start vertex. This is O(n) + O(n+m) + O(n)= O(n+m).

Main Point

Many of the theoretical results about directed graphs are more complicated analogues to results about undirected graphs. For instance, algorithms for marking all vertices *reachable* from a starting vertex and for discovering a directed path from one vertex to another are only slightly more complicated than their analogues for undirected graphs. Beyond these, there are many additional results about directed graphs that could not be anticipated from a study of undirected graphs alone. An example is the application of the Depth First Search algorithm to produce a topological ordering for a DAG; in the absence of directed edges, the concept of "ordering" on a graph doesn't even make sense, so this discovery could never have been made by studying undirected graphs alone. This illustrates the basic principle that, in the manifestation of the world, different laws of nature arise, governing very different dynamics. Laws governing the surface level of life are very different from those governing subtle realms of existence and thought, and these are different still from the laws that govern the first sprouting of manifestation.

Connecting the Parts of Knowledge With the Wholeness of Knowledge

TOPOLOGICAL SORT

1. The DFS algorithm for directed graphs provides an efficient procedure for traversing all reachable vertices in a directed graph.
2. By paying attention to the exact moments at which DFS changes direction and begins to backtrack, and assigning integers in reverse order each time there is such a transition, we can put the vertices of the digraph in topologically sorted order.
3. *Transcendental Consciousness*, the field of pure consciousness, is experienced at the junction point between the inward movement of thought and outward movement of thought.
4. *Impulses within the Transcendental field*: In the gap between the settling of a thought to unboundedness and the emergence of a new thought, the unmanifest activity of the transcendental field spontaneously computes the direction, energy, and content of the new thought.
5. *Wholeness moving within itself*: In Unity Consciousness, each expression of the universe is seen as the effortless creation of one's own unbounded nature.