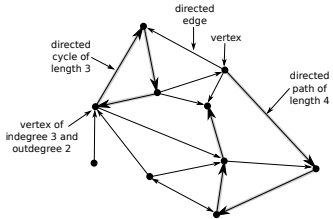
Directed Graphs

Outline

Directed Graphs

Directed Graphs

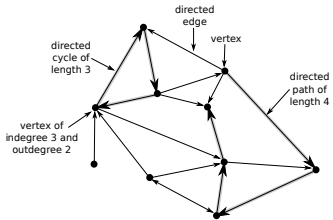
A directed graph (digraph) is a set of vertices and a collection of directed edges, each connecting an ordered pair of vertices



Directed Graphs

A directed graph (digraph) is a set of vertices and a collection of directed edges, each connecting an ordered pair of vertices

The outdegree of a vertex in a digraph is the number of edges going from it; the indegree of a vertex is the number of edges going to it

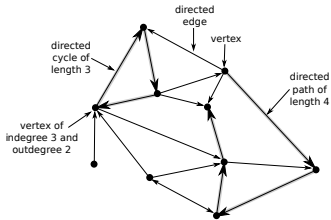


Directed Graphs

A directed graph (digraph) is a set of vertices and a collection of directed edges, each connecting an ordered pair of vertices

The outdegree of a vertex in a digraph is the number of edges going from it; the indegree of a vertex is the number of edges going to it

A directed path in a digraph is a sequence of vertices in which there is a (directed) edge pointing from each vertex in the sequence to its successor in the sequence



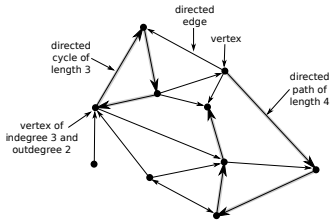
Directed Graphs

A directed graph (digraph) is a set of vertices and a collection of directed edges, each connecting an ordered pair of vertices

The outdegree of a vertex in a digraph is the number of edges going from it; the indegree of a vertex is the number of edges going to it

A directed path in a digraph is a sequence of vertices in which there is a (directed) edge pointing from each vertex in the sequence to its successor in the sequence

A directed cycle is a directed path with at least one edge whose first and last vertices are the same



Directed Graphs

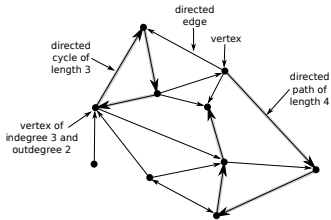
A directed graph (digraph) is a set of vertices and a collection of directed edges, each connecting an ordered pair of vertices

The outdegree of a vertex in a digraph is the number of edges going from it; the indegree of a vertex is the number of edges going to it

A directed path in a digraph is a sequence of vertices in which there is a (directed) edge pointing from each vertex in the sequence to its successor in the sequence

A directed cycle is a directed path with at least one edge whose first and last vertices are the same

The length of a path or a cycle is its number of edges



Directed Graphs

Digraph applications

Digraph	Vertex	Edge
transportation	street intersection	one-way street
web	web page	hyperlink
food web	species	predator-prey relationship
WordNet	synset	hypernym
scheduling	task	precedence constraint
financial	bank	transaction
cell phone	person	placed call
infectious disease	person	infection
game	board position	legal move
citation	journal article	citation
object graph	object	pointer
inheritance hierarchy	class	inherits from
control flow	code block	jump

Directed Graphs

Directed Graphs

Some digraph problems

Problem	Description
$s \rightarrow t$ path	is there a path from s to t?
shortest $s \rightarrow t$ path	what is the shortest path from s to t?
directed cycle	is there a directed cycle in the graph?
topological sort	can the digraph be drawn so that all edges point in a single direction?

Directed Graphs

Directed Graphs

DiGraph API

Method	Description
`DiGraph(int V)`	create an empty digraph with V vertices
`DiGraph(In in)`	create a digraph from input stream
`void addEdge(int v, int w)`	add a directed edge $v \rightarrow w$
`Iterable<Integer> adj(int v)`	vertices pointing from v
`int V()`	number of vertices
`int E()`	number of edges
`Digraph reverse()`	reverse of this digraph

Directed Graphs

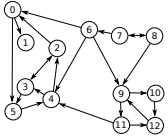
DiGraph API

Method	Description
`DiGraph(int V)`	create an empty digraph with V vertices
`DiGraph(In in)`	create a digraph from input stream
`void addEdge(int v, int w)`	add a directed edge $v \rightarrow w$
`Iterable<Integer> adj(int v)`	vertices pointing from v
`int V()`	number of vertices
`int E()`	number of edges
`Digraph reverse()`	reverse of this digraph

Graph input format

```
>_ ~/workspace/dsa/programs

$ more ../data/tinyDG.txt
13 22
4 2 2 3 3 2 6 0 0 1 2 0 11 12 12 9
9 10 9 11 8 9 10 12 11 4 4 3 3 5
7 8 8 7 5 4 0 5 6 4 6 9 7 6
```



Directed Graphs

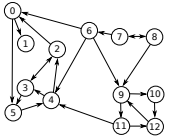
DiGraph API

Method	Description
`DiGraph(int V)`	create an empty digraph with V vertices
`DiGraph(In in)`	create a digraph from input stream
`void addEdge(int v, int w)`	add a directed edge $v \rightarrow w$
`Iterable<Integer> adj(int v)`	vertices pointing from v
`int V()`	number of vertices
`int E()`	number of edges
`Digraph reverse()`	reverse of this digraph

Graph input format

```
>_ ~/workspace/dsa/programs

$ more ../data/tinyDG.txt
13 22
4 2 2 3 3 2 6 0 0 1 2 0 11 12 12 9
9 10 9 11 8 9 10 12 11 4 4 3 3 5
7 8 8 7 5 4 0 5 6 4 6 9 7 6
```



Typical graph-processing code

```
In in = new In(args[0]);
DiGraph G = new DiGraph(in);
for (int v = 0; v < G.V(); v++) {
    for (int w : G.adj(v)) {
        StdOut.println(v + "->" + w);
    }
}
```

Directed Graphs

Directed Graphs

DiGraph.java

```
package dsa;

import stdlib.In;
import stdlib.StdOut;

public class DiGraph {
    private LinkedBag<Integer>[] adj;
    private int V;
    private int E;

    public DiGraph(int V) {
        adj = (LinkedBag<Integer>[]) new LinkedBag[V];
        for (int v = 0; v < V; v++) {
            adj[v] = new LinkedBag<Integer>();
        }
        this.V = V;
        this.E = 0;
    }

    public DiGraph(In in) {
        this(in.readInt());
        int E = in.readInt();
        for (int i = 0; i < E; i++) {
            int v = in.readInt();
            int w = in.readInt();
            addEdge(v, w);
        }
    }

    public int V() {
        return V;
    }

    public int E() {
        return E;
```

Directed Graphs

DiGraph.java

```
    }

    public void addEdge(int v, int w) {
        adj[v].add(w);
        E++;
    }

    public Iterable<Integer> adj(int v) {
        return adj[v];
    }

    public int outDegree(int v) {
        return adj[v].size();
    }

    public int inDegree(int v) {
        int inDegree = 0;
        for (LinkedBag<Integer> bag : adj) {
            for (int u : bag) {
                inDegree += u == v ? 1 : 0;
            }
        }
        return inDegree;
    }

    public String toString() {
        StringBuilder s = new StringBuilder();
        s.append(V + " vertices, " + E + " edges\n");
        for (int v = 0; v < V; v++) {
            s.append(String.format("%d: ", v));
            for (int w : adj[v]) {
                s.append(String.format("%d ", w));
            }
            s.append("\n");
        }
```

Directed Graphs

DiGraph.java

```
        return s.toString().strip();
    }

    public static void main(String[] args) {
        String filename = args[0];
        In in = new In(filename);
        DiGraph G = new DiGraph(in);
        StdOut.println(G);
    }
}
```

Depth-First Search (DFS)

Depth-First Search (DFS)

Same method as for undirected graphs, ie, to visit a vertex v

- Mark a vertex v as visited
- Recursively visit all unmarked vertices pointing from v

Depth-First Search (DFS)

Same method as for undirected graphs, ie, to visit a vertex v

- Mark a vertex v as visited
- Recursively visit all unmarked vertices pointing from v

Reachability problem

- Single-source reachability: given a digraph and a source vertex s, support queries of the form *is there a directed path from s to a given target vertex v?*
- Multi-source reachability: given a digraph and a set of source vertices, support queries of the form *is there a directed path from any vertex in the set to a given target vertex v?*

Depth-First Search (DFS)

Same method as for undirected graphs, ie, to visit a vertex v

- Mark a vertex v as visited
- Recursively visit all unmarked vertices pointing from v

Reachability problem

- Single-source reachability: given a digraph and a source vertex s, support queries of the form *is there a directed path from s to a given target vertex v?*
- Multi-source reachability: given a digraph and a set of source vertices, support queries of the form *is there a directed path from any vertex in the set to a given target vertex v?*

Applications

- Program control-flow analysis such as dead-code elimination and infinite-loop detection
- Mark and sweep garbage collector

Breadth-First Search (BFS)

Breadth-First Search (BFS)

Same method as for undirected graphs, ie, repeat until queue is empty

- Remove vertex v from queue
- Add to queue all unmarked vertices pointing from v and mark them

Breadth-First Search (BFS)

Same method as for undirected graphs, ie, repeat until queue is empty

- Remove vertex v from queue
- Add to queue all unmarked vertices pointing from v and mark them

BFS computes shortest paths (fewest number of edges) from source vertex s to all other vertices in a digraph in time proportional to $E + V$

Breadth-First Search (BFS)

Same method as for undirected graphs, ie, repeat until queue is empty

- Remove vertex v from queue
- Add to queue all unmarked vertices pointing from v and mark them

BFS computes shortest paths (fewest number of edges) from source vertex s to all other vertices in a digraph in time proportional to $E + V$

Multiple-source shortest paths: given a digraph and a set of source vertices, find shortest path from any vertex in the set to each other vertex; solution: use BFS, but initialize by enqueuing all source vertices

Topological Sort

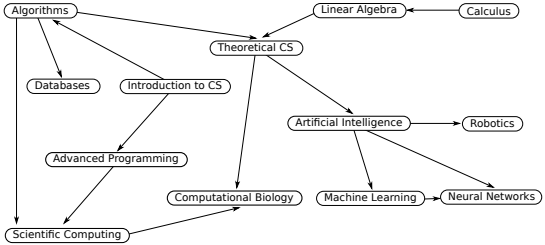
Topological Sort

Precedence-constrained scheduling: given a set of jobs to be completed, with precedence constraints that specify that certain jobs have to be completed before certain other jobs are begun, how can we schedule the jobs such that they are all completed while still respecting the constraints?

Topological Sort

Precedence-constrained scheduling: given a set of jobs to be completed, with precedence constraints that specify that certain jobs have to be completed before certain other jobs are begun, how can we schedule the jobs such that they are all completed while still respecting the constraints?

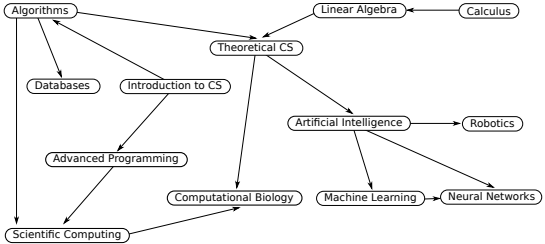
Example (precedence-constrained course scheduling problem)



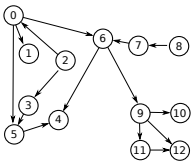
Topological Sort

Precedence-constrained scheduling: given a set of jobs to be completed, with precedence constraints that specify that certain jobs have to be completed before certain other jobs are begun, how can we schedule the jobs such that they are all completed while still respecting the constraints?

Example (precedence-constrained course scheduling problem)



A digraph model for the problem



Topological Sort

Topological Sort

Topological sort: given a directed acyclic graph (DAG), put the vertices in order such that all its edges point from a vertex earlier in the order to a vertex later in the order

Topological Sort

Topological sort: given a directed acyclic graph (DAG), put the vertices in order such that all its edges point from a vertex earlier in the order to a vertex later in the order

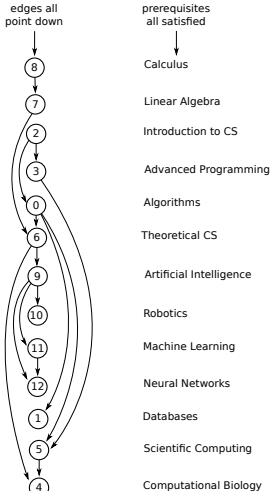
A digraph has a topological order if and only if it is a DAG

Topological Sort

Topological sort: given a directed acyclic graph (DAG), put the vertices in order such that all its edges point from a vertex earlier in the order to a vertex later in the order

A digraph has a topological order if and only if it is a DAG

Topological order for the precedence-constrained course scheduling problem



Topological Sort

Topological Sort

DiCycle.java

```
package dsa;

import stdlib.In;
import stdlib.StdOut;

public class DiCycle {
    private boolean[] marked;
    private int[] edgeTo;
    private boolean[] onStack;
    private LinkedStack<Integer> cycle;

    public DiCycle(DiGraph G) {
        marked = new boolean[G.V()];
        edgeTo = new int[G.V()];
        onStack = new boolean[G.V()];
        for (int v = 0; v < G.V(); v++) {
            if (!marked[v] && cycle == null) {
                dfs(G, v);
            }
        }
    }

    public boolean hasCycle() {
        return cycle != null;
    }

    public Iterable<Integer> cycle() {
        return cycle;
    }

    private void dfs(DiGraph G, int v) {
        marked[v] = true;
        onStack[v] = true;
        for (int w : G.adj(v)) {
            if (cycle != null) {
```

Topological Sort

DiCycle.java

```
                return;
            } else if (!marked[w]) {
                edgeTo[w] = v;
                dfs(G, w);
            } else if (onStack[w]) {
                cycle = new LinkedStack<Integer>();
                for (int x = v; x != w; x = edgeTo[x]) {
                    cycle.push(x);
                }
                cycle.push(w);
                cycle.push(v);
            }
        }
        onStack[v] = false;
    }

    public static void main(String[] args) {
        In in = new In(args[0]);
        DiGraph G = new DiGraph(in);
        DiCycle finder = new DiCycle(G);
        if (finder.hasCycle()) {
            StdOut.print("Directed cycle: ");
            for (int v : finder.cycle()) {
                StdOut.print(v + " ");
            }
            StdOut.println();
        } else {
            StdOut.println("No directed cycle");
        }
    }
}
```

Topological Sort

Topological Sort

```
>_ ~/workspace/dsa/programs

$ java dsa.DiCycle ../data/tinyDG.txt
Directed cycle: 3 5 4 3
```

Topological Sort

```
>_ ~/workspace/dsa/programs

$ java dsa.DiCycle ../data/tinyDG.txt
Directed cycle: 3 5 4 3
```

Directed cycle detection applications

- Cyclic inheritance
- Circular references in spreadsheet calculations

Topological Sort

Topological Sort

DFS orders

- Preorder: order in which `dfs()` is called
- Postorder: order in which `dfs()` returns
- Reverse postorder: reverse order in which `dfs()` returns

Topological Sort

Topological Sort

DFSOrders.java

```
package dsa;

import stdlib.In;
import stdlib.StdOut;

public class DFSOrders {
    private boolean[] marked;
    private int[] pre;
    private int[] post;
    private LinkedQueue<Integer> preorder;
    private LinkedQueue<Integer> postorder;
    private int preCounter;
    private int postCounter;

    public DFSOrders(DiGraph G) {
        marked = new boolean[G.V()];
        pre = new int[G.V()];
        post = new int[G.V()];
        preorder = new LinkedQueue<Integer>();
        postorder = new LinkedQueue<Integer>();
        for (int v = 0; v < G.V(); v++) {
            if (!marked[v]) {
                dfs(G, v);
            }
        }
    }

    public int pre(int v) {
        return pre[v];
    }

    public int post(int v) {
        return post[v];
    }
```

Topological Sort

DFSOrders.java

```
    public Iterable<Integer> pre() {
        return preorder;
    }

    public Iterable<Integer> post() {
        return postorder;
    }

    public Iterable<Integer> reversePost() {
        LinkedStack<Integer> reverse = new LinkedStack<Integer>();
        for (int v : postorder) {
            reverse.push(v);
        }
        return reverse;
    }

    private void dfs(DiGraph G, int v) {
        marked[v] = true;
        pre[v] = preCounter++;
        preorder.enqueue(v);
        for (int w : G.adj(v)) {
            if (!marked[w]) {
                dfs(G, w);
            }
        }
        postorder.enqueue(v);
        post[v] = postCounter++;
    }

    public static void main(String[] args) {
        In in = new In(args[0]);
        DiGraph G = new DiGraph(in);
        DFSOrders dfsOrders = new DFSOrders(G);
        StdOut.println("   v  pre post");
        StdOut.println("--------------");
```

Topological Sort

DFSOrders.java

```
        for (int v = 0; v < G.V(); v++) {
            StdOut.printf("%4d %4d %4d\n", v, dfsOrders.pre(v), dfsOrders.post(v));
        }
        StdOut.print("Pre-order:  ");
        for (int v : dfsOrders.pre()) {
            StdOut.print(v + " ");
        }
        StdOut.println();
        StdOut.print("Post-order: ");
        for (int v : dfsOrders.post()) {
            StdOut.print(v + " ");
        }
        StdOut.println();
        StdOut.print("Reverse post-order: ");
        for (int v : dfsOrders.reversePost()) {
            StdOut.print(v + " ");
        }
        StdOut.println();
    }
}
```

Topological Sort

Topological Sort

```
>_ ~/workspace/dsa/programs

$ java dsa.DFSOrders ../data/tinyDG.txt
   v  pre post
--------------
   0    0    5
   1    5    4
   2    4    0
   3    3    1
   4    2    2
   5    1    3
   6    6   11
   7   12   12
   8   11   10
   9    7    9
  10   10    8
  11    8    7
  12    9    6
Preorder:  0 5 4 3 2 1 6 9 11 12 10 8 7
Postorder: 2 3 4 5 1 0 12 11 10 9 8 6 7
Reverse postorder: 7 6 8 9 10 11 12 0 1 5 4 3 2
```

Topological Sort

Topological Sort

Topological sort (solution)

- Run depth-first search
- Return vertices in reverse postorder

Topological Sort

Topological Sort

Topological.java

```
package dsa;

import stdlib.In;
import stdlib.StdOut;

public class Topological {
    private Iterable<Integer> order;

    public Topological(DiGraph G) {
        DiCycle finder = new DiCycle(G);
        if (finder.hasCycle()) {
            order = null;
        } else {
            DFSOrders dfs = new DFSOrders(G);
            order = dfs.reversePost();
        }
    }

    public boolean hasOrder() {
        return order != null;
    }

    public Iterable<Integer> order() {
        return order;
    }

    public static void main(String[] args) {
        In in = new In(args[0]);
        String delim = args[1];
        SymbolDiGraph sg = new SymbolDiGraph(in, delim);
        Topological topological = new Topological(sg.diGraph());
        if (topological.hasOrder()) {
            for (int v : topological.order()) {
                StdOut.println(sg.nameOf(v));
            }
```

Topological Sort

Topological.java

```
        } else {
            StdOut.println("Topological order does not exist");
        }
    }
}
```

Topological Sort

Topological Sort

```
>_ ~/workspace/dsa/programs
```

```
$ more ../data/jobs.txt
Algorithms/Theoretical CS/Databases/Scientific Computing
Introduction to CS/Advanced Programming/Algorithms
Advanced Programming/Scientific Computing
Scientific Computing/Computational Biology
Theoretical CS/Computational Biology/Artificial Intelligence
Linear Algebra/Theoretical CS
Calculus/Linear Algebra
Artificial Intelligence/Neural Networks/Robotics/Machine Learning
Machine Learning/Neural Networks
```

Topological Sort

```
>_ ~/workspace/dsa/programs

$ more ../data/jobs.txt
Algorithms/Theoretical CS/Databases/Scientific Computing
Introduction to CS/Advanced Programming/Algorithms
Advanced Programming/Scientific Computing
Scientific Computing/Computational Biology
Theoretical CS/Computational Biology/Artificial Intelligence
Linear Algebra/Theoretical CS
Calculus/Linear Algebra
Artificial Intelligence/Neural Networks/Robotics/Machine Learning
Machine Learning/Neural Networks
```

```
>_ ~/workspace/dsa/programs

$ java dsa.Topological ../data/jobs.txt "/"
Calculus
Linear Algebra
Introduction to CS
Advanced Programming
Algorithms
Theoretical CS
Artificial Intelligence
Robotics
Machine Learning
Neural Networks
Databases
Scientific Computing
Computational Biology
```