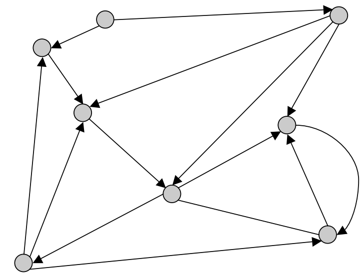
Directed Graphs

Reference: Chapter 19, Algorithms in Java, 3rd Edition, Robert Sedgewick

Directed Graphs

Digraph. Set of objects with oriented pairwise connections.

Ex. One-way street, hyperlink.

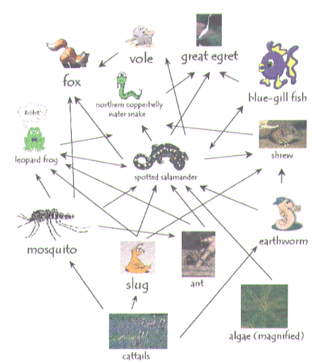


Digraph Applications

Digraph	Vertex	Edge
financial	stock, currency	transaction
transportation	street intersection, airport	highway, airway route
scheduling	task	precedence constraint
WordNet	synset	hypernym
Web	web page	hyperlink
game	board position	legal move
telephone	person	placed call
food web	species	predator-prey relation
infectious disease	person	infection
citation	journal article	citation
object graph	object	pointer
inheritance hierarchy	class	inherits from
control flow	code block	jump

Ecological Food Web

Food web graph.
- Vertex = species.
- Edge = from prey to predator.



http://www.twingroves.district96.k12.il.us/Wetlands/Salamander/SalGraphics/salfoodweb.giff

Some Digraph Problems

Transitive closure. Is there a directed path from `v` to `w`?

Strong connectivity. Are all vertices mutually reachable?

Topological sort. Can you draw the graph so that all edges point from left to right?

PERT/CPM. Given a set of tasks with precedence constraints, what is the earliest that we can complete each task?

Shortest path. Given a weighted graph, find best route from `s` to `t`?

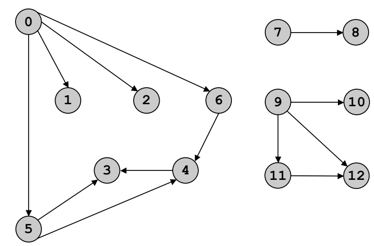
PageRank. What is the importance of a web page?

Digraph Representation

Vertex names.

- This lecture: use integers between `0` and `V-1`.
- Real world: convert between names and integers with symbol table.

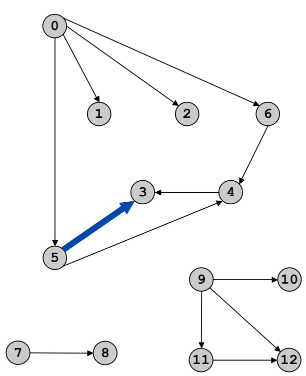
Orientation of edge matters.



Adjacency Matrix Representation

Adjacency matrix representation.

- Two-dimensional `V` × `V` boolean array.
- Edge `v→w` in graph: `adj[v][w] = true`.

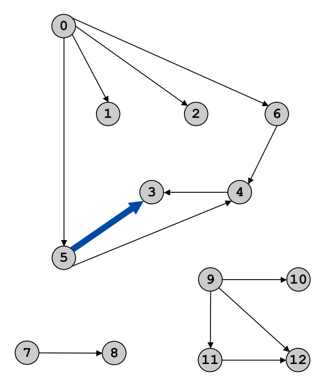


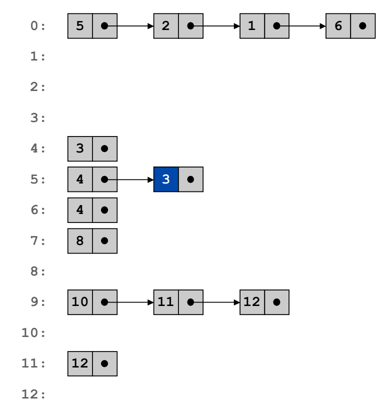
to

from	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	1	1	0	0	1	1	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	1	0	0	0	0	0	0	0	0	0
5	0	0	0	1	1	0	0	0	0	0	0	0	0
6	0	0	0	0	1	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	1	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	1	1	1
10	0	0	0	0	0	0	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0	0	0	1
12	0	0	0	0	0	0	0	0	0	0	0	0	0

Adjacency List Representation

Adjacency list representation. Vertex indexed array of lists.





Adjacency List: Java Implementation

Implementation. Same as `Graph`, but only insert one copy of each edge.

```
public class Digraph {
   private int V;
   private Sequence<Integer>[] adj;

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

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

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


Digraph Representations

Digraphs are abstract mathematical objects.

- ADT implementation requires specific representation.
- Efficiency depends on matching algorithms to representations.

Representation	Space	Edge from v to w?	Iterate over edges leaving v?
List of edges	E + V	E	E
Adjacency matrix	V^2	1	V
Adjacency list	E + V	outdegree(v)	outdegree(v)

Digraphs in practice. ⇒ use adjacency list representation

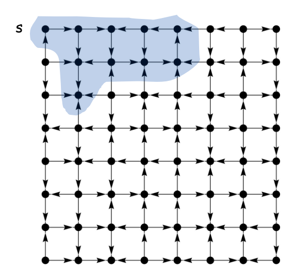
- Bottleneck is iterating over edges leaving `v`.
- Real world digraphs are sparse.

E is proportional to V

Digraph Search

Reachability

Goal. Find all vertices reachable from `s` along a directed path.



Depth First Search

Depth first search. Same as for undirected graphs.

DFS (to visit a vertex v)
Mark v as visited. Visit all unmarked vertices w adjacent to v .

Running time. O(E) since each edge examined at most once.

Depth First Search

Remark. Same as undirected version, except `Digraph` instead of `Graph`.

```
public class DFSearcher {
   private boolean[] marked;

   public DFSearcher(Digraph G, int s) {
      marked = new boolean[G.V()];
      dfs(G, s);
   }

   private void dfs(Digraph G, int v) {
      marked[v] = true;
      for (int w : G.adj(v))
         if (!marked[w]) dfs(G, w);
   }

   public boolean isReachable(int v) { return marked[v]; }

}
```


Control Flow Graph

Control-flow graph.

- Vertex = basic block (straight-line program).
- Edge = jump.

Dead code elimination. Find (and remove) code blocks that are unreachable during execution.

dead code can arise from compiler optimization
(or careless programmer)

Infinite loop detection. Exit block is unreachable from entry block.
Caveat. Not all infinite loops are detectable.

Mark-Sweep Garbage Collector

Roots. Objects known to be accessible by program (e.g., stack).

Live objects. Objects that the program could get to by starting at a root and following a chain of pointers.

easy to identify pointers in type-safe language

Mark-sweep algorithm. [McCarthy, 1960]

- Mark: run DFS from roots to mark live objects.
- Sweep: if object is unmarked, it is garbage, so add to free list.

Extra memory. Uses 1 extra mark bit per object, plus DFS stack.

Depth First Search

DFS enables direct solution of simple digraph problems.

- Reachability.
- Cycle detection.
- Topological sort.
- Transitive closure.
- Find path from `s` to `t`.

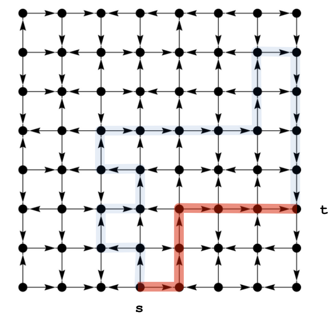
Basis for solving difficult digraph problems.

- Directed Euler path.
- Strong connected components.

Breadth First Search

Shortest path. Find the shortest directed path from `s` to `t`.

BFS. Analogous to BFS in undirected graphs.



Application: Web Crawler

Web graph. Vertex = website, edge = hyperlink.

Goal. Crawl Internet, starting from some root website.
Solution. BFS with implicit graph.

BFS.

- Start at some root website, say `http://www.princeton.edu`.
- Maintain a `Queue` of websites to explore.
- Maintain a `SET` of discovered websites.
- Dequeue the next website, and enqueue websites to which it links (provided you haven't done so before).

Q. Why not use DFS?

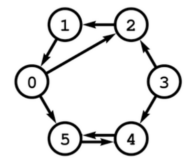
Web Crawler: Java Implementation

```
Queue<String> q = new Queue<String>();        ← queue of sites to crawl
SET<String> visited = new SET<String>();      ← set of visited sites
String s = "http://www.princeton.edu";
q.enqueue(s);                                 ↖ start crawling from s
visited.add(s);
while (!q.isEmpty()) {
   String v = q.dequeue();
   System.out.println(v);                     read in raw html
   In in = new In(v);                       ↙                      http://xxx.yyy.zzz
   String input = in.readAll();                                  ↙
   String regexp = "http://(\\w+\\.)*(\\w+)";
   Pattern pattern = Pattern.compile(regexp);
   Matcher matcher = pattern.matcher(input);
   while (matcher.find()) {                  ← search using regular expression
      String w = matcher.group();
      if (!visited.contains(w)) {
            visited.add(w);                   if unvisited, mark as visited
            q.enqueue(w);                     and put on queue
      }
   }
}
```

Transitive Closure

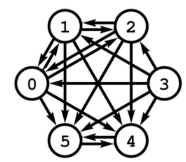
Transitive closure. Is there a directed path from `v` to `w`?

G



	0	1	2	3	4	5
0	1	0	1	0	0	1
1	1	1	0	0	0	0
2	0	1	1	0	0	0
3	0	0	1	1	1	0
4	0	0	0	0	1	1
5	0	0	0	0	1	1

Transitive closure



	0	1	2	3	4	5
0	1	1	1	0	1	1
1	1	1	1	0	1	1
2	1	1	1	0	1	1
3	1	1	1	1	1	1
4	0	0	0	0	1	1
5	0	0	0	0	1	1

tc[v][w] = 1 iff path from v to w

Transitive closure. Is there a directed path from `v` to `w`?

Lazy. Run separate DFS for each query.
Eager. Run DFS from every vertex `v`; store results.

Method	Preprocess	Query	Space
DFS (lazy)	1	E + V	E + V
DFS (eager)	E V	1	V^2

Remark. Directed problem is harder than undirected one.
Open research problem. $O(1)$ query, $O(V^2)$ preprocessing time.

Implementation. Use an array of `DFSearcher` objects.

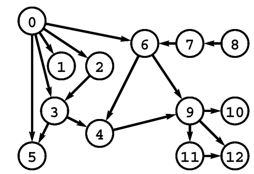
```
public class TransitiveClosure {
   private DFSearcher[] tc;

   public TransitiveClosure(Digraph G) {
      tc = new Reachability[G.V()];
      for (int v = 0; v < G.V(); v++)
         tc[v] = new Reachability(G, v);
   }

   public boolean reachable(int v, int w) {
      return tc[v].isReachable(w);
   }
}
```

is w reachable from v?

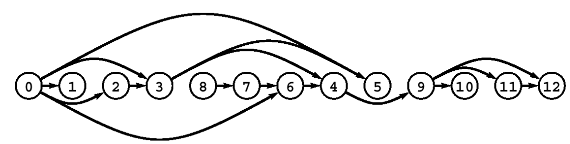
Topological Sort

Topological Sort

DAG. Directed acyclic graph.



Topological sort. Redraw DAG so all edges point left to right.



Observation. Not possible if graph has a directed cycle.

Application: Scheduling

Scheduling. Given a set of tasks to be completed with precedence constraints, in what order should we schedule the tasks?

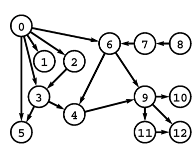
Graph model.

- Create a vertex `v` for each task.
- Create an edge `v`→`w` if task `v` must precede task `w`.
- Schedule tasks in topological order.

0. read programming assignment
1. download files
2. write code

...

12. sleep



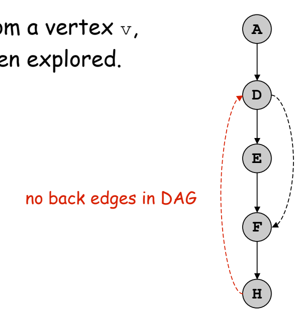
Topological Sort: DFS

Topologically sort a DAG.

- Run DFS.
- Reverse postorder numbering yields a topological sort.

Pf of correctness. When DFS backtracks from a vertex `v`, all vertices reachable from `v` have already been explored.

Running time. O(E + V).



DFS tree

Q. If not a DAG, how would you identify a cycle?

Topological Sort: Java Implementation

```java
public class TopologicalSorter {
   private int count;
   private boolean[] visited;
   private int[] ts;

   public TopologicalSorter(Digraph G) {
      visited = new boolean[G.V()];
      ts = new int[G.V()];
      count = G.V();
      for (int v = 0; v < G.V(); v++)
         if (!visited[v]) tsort(G, v);
   }

   private void tsort(Digraph G, int v) {
      visited[v] = true;
      for (int w : G.adj(v))
         if (!visited[w]) tsort(G, w);
      }
      ts[--count] = v;
   }
}
```

assign numbers in reverse DFS postorder

Topological Sort: Applications

Topological sort applications.

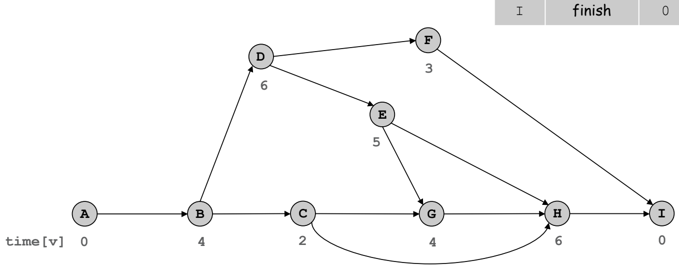
- Causalities.
- Compilation units.
- Class inheritance.
- Course prerequisites.
- Deadlocking detection.
- Temporal dependencies.
- Pipeline of computing jobs.
- Check for symbolic link loop.
- Evaluate formula in spreadsheet.

Program Evaluation and Review Technique / Critical Path Method

PERT/CPM.

- Task `v` takes `time[v]` units of time.
- Can work on jobs in parallel.
- Precedence constraints: must finish task `v` before beginning task `w`.
- What's earliest we can finish each task?

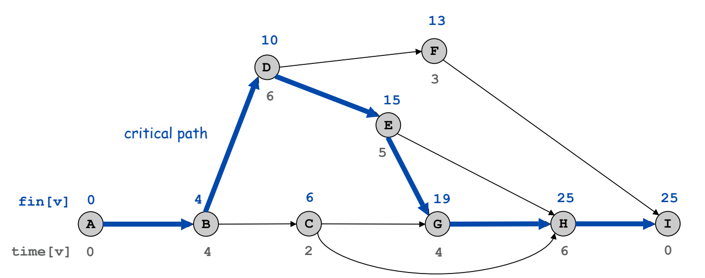
index	task	time	prereq
A	begin	0	-
B	framing	4	A
C	roofing	2	B
D	siding	6	B
E	windows	5	D
F	plumbing	3	D
G	electricity	4	C, E
H	paint	6	C, E
I	finish	0	F, H



Program Evaluation and Review Technique / Critical Path Method

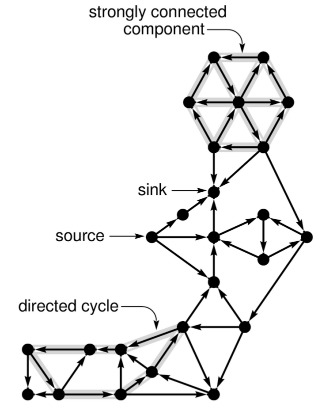
PERT/CPM algorithm.

- Compute topological order of vertices.
- Initialize `fin[v] = 0` for all vertices `v`.
- Consider vertices `v` in topological order.
  - for each edge `v→w`, set `fin[w]= max(fin[w], fin[v] + time[w])`



Strongly Connected Components

Terminology



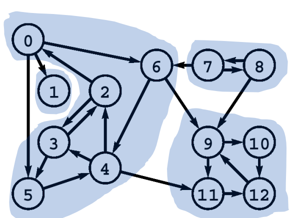
Strong Components

Def. Vertices `v` and `w` are strongly connected if there is a path from `v` to `w` and a path from `w` to `v`.

Properties. Symmetric, transitive, reflexive.

Strong component. Maximal subset of strongly connected vertices.

Brute force. O(EV) time using transitive closure.



	0	1	2	3	4	5	6	7	8	9	10	11	12
sc	2	1	2	2	2	2	2	3	3	0	0	0	0

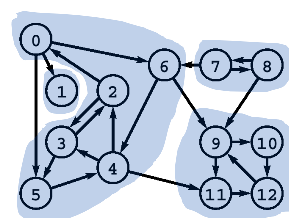
Computing Strongly Connected Components

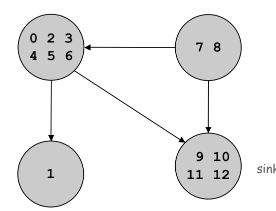
Observation 1. If you run DFS from a vertex in sink strong component, all reachable vertices constitute a strong component.

Observation 2. If you run DFS on G, the node with the highest postorder number is in source strong component.

in order of completion of recursive calls

Observation 3. If you run DFS on G^R, the node with the highest postorder number is in sink strong component.

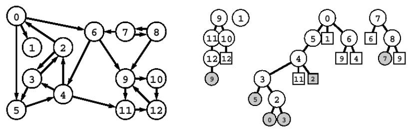




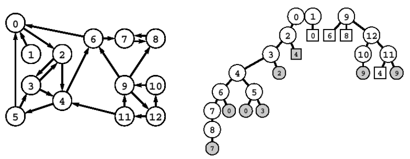
kernel DAG

Kosaraju's algorithm.

- Run DFS on G^R and compute postorder.
- Run DFS on G, considering vertices in reverse postorder.



	0	1	2	3	4	5	6	7	8	9	10	11	12
post	8	7	6	5	4	3	2	0	1	11	10	12	9

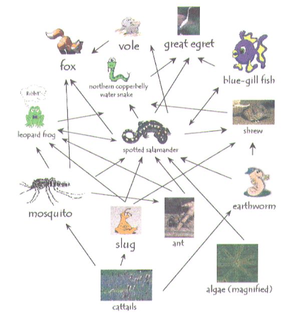


	0	1	2	3	4	5	6	7	8	9	10	11	12
sc	2	1	2	2	2	2	2	3	3	0	0	0	0

Theorem. Trees in second DFS are strong components. (!)

Ecological food web.

- Vertex = species.
- Edge = from producer to consumer.
- Strong component = subset of species for which energy flows from one another and back.



http://www.twingroves.district96.k12.il.us/Wetlands/Salamander/SalGraphics/salfoodweb.giff