4.2 DIRECTED GRAPHS

- introduction
- digraph API
- depth-first search
- breadth-first search
- topological sort
4.2 Directed Graphs

- introduction
- digraph API
- depth-first search
- breadth-first search
- topological sort

https://algs4.cs.princeton.edu
Road networks

Vertex = intersection; edge = one-way street.
Political blogosphere links

Vertex = political blog; edge = link.

The Political Blogosphere and the 2004 U.S. Election: Divided They Blog, Adamic and Glance, 2005
Russian troll network

Vertex = Russian troll; edge = Twitter mention.
Science clickstreams

http://www.plosone.org/article/info:doi/10.1371/journal.pone.0004803
Overnight interbank loans

Vertex = bank; edge = overnight loan.

The Topology of the Federal Funds Market, Bech and Atalay, 2008
<table>
<thead>
<tr>
<th>digraph</th>
<th>vertex</th>
<th>directed edge</th>
</tr>
</thead>
<tbody>
<tr>
<td>transportation</td>
<td>street intersection</td>
<td>one-way street</td>
</tr>
<tr>
<td>web</td>
<td>web page</td>
<td>hyperlink</td>
</tr>
<tr>
<td>food web</td>
<td>species</td>
<td>predator–prey relationship</td>
</tr>
<tr>
<td>WordNet</td>
<td>synset</td>
<td>hypernym</td>
</tr>
<tr>
<td>scheduling</td>
<td>task</td>
<td>precedence constraint</td>
</tr>
<tr>
<td>financial</td>
<td>bank</td>
<td>transaction</td>
</tr>
<tr>
<td>cell phone</td>
<td>person</td>
<td>placed call</td>
</tr>
<tr>
<td>infectious disease</td>
<td>person</td>
<td>infection</td>
</tr>
<tr>
<td>game</td>
<td>board position</td>
<td>legal move</td>
</tr>
<tr>
<td>citation</td>
<td>journal article</td>
<td>citation</td>
</tr>
<tr>
<td>object graph</td>
<td>object</td>
<td>pointer</td>
</tr>
<tr>
<td>inheritance hierarchy</td>
<td>class</td>
<td>inherits from</td>
</tr>
<tr>
<td>control flow</td>
<td>code block</td>
<td>jump</td>
</tr>
</tbody>
</table>
**Directed graph terminology**

**Digraph.** Set of vertices connected pairwise by *directed* edges.
### Some digraph problems

<table>
<thead>
<tr>
<th>problem</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>s→t path</td>
<td>Is there a path from s to t?</td>
</tr>
<tr>
<td>shortest s→t path</td>
<td>What is the shortest path from s to t?</td>
</tr>
<tr>
<td>directed cycle</td>
<td>Is there a directed cycle in the graph?</td>
</tr>
<tr>
<td>topological sort</td>
<td>Can the digraph be drawn so that all edges point upwards?</td>
</tr>
<tr>
<td>strong connectivity</td>
<td>Is there a directed path between every pairs of vertices?</td>
</tr>
<tr>
<td>transitive closure</td>
<td>For which vertices v and w is there a directed path from v to w?</td>
</tr>
<tr>
<td>PageRank</td>
<td>What is the importance of a web page?</td>
</tr>
</tbody>
</table>
4.2 Directed Graphs

- Introduction
- Digraph API
- Depth-first search
- Breadth-first search
- Topological sort

https://algs4.cs.princeton.edu
## Digraph API

Almost identical to Graph API.

```java
public class Digraph {
    Digraph(int V) create an empty digraph with V vertices
    void addEdge(int v, int w) add a directed edge v→w
    Iterable<Integer> adj(int v) vertices adjacent from v
    int V() number of vertices
    ... ...
}
```

**Note.** Full Digraph API includes additional methods, such as `reverse()`.
Digraph representation: adjacency lists

Maintain vertex-indexed array of lists.
Which is the order of growth of the running time for removing an edge \( v \rightarrow w \) from a digraph using the \textit{adjacency-lists} representation, where \( V \) is the number of vertices and \( E \) is the number of edges?

A. 1  
B. \( \text{outdegree}(v) \)  
C. \( \text{indegree}(w) \)  
D. \( \text{outdegree}(v) + \text{indegree}(w) \)
Which is the order of growth of the running time of the following code fragment if the digraph uses the adjacency-lists representation, where $V$ is the number of vertices and $E$ is the number of edges?

A. $V$
B. $E + V$
C. $V^2$
D. $VE$

```java
for (int v = 0; v < G.V(); v++)
    for (int w : G.adj(v))
        StdOut.println(v + "->" + w);
```

prints each edge exactly once
Digraph representations

In practice. Use adjacency-lists representation.
- Algorithms based on iterating over vertices adjacent from $v$.
- Real-world graphs tend to be **sparse** (not **dense**).

<table>
<thead>
<tr>
<th>representation</th>
<th>space</th>
<th>insert edge from $v$ to $w$</th>
<th>edge from $v$ to $w$?</th>
<th>iterate over vertices adjacent from $v$?</th>
</tr>
</thead>
<tbody>
<tr>
<td>list of edges</td>
<td>$E$</td>
<td>1</td>
<td>$E$</td>
<td>$E$</td>
</tr>
<tr>
<td>adjacency matrix</td>
<td>$V^2$</td>
<td>$1$</td>
<td>$1$</td>
<td>$V$</td>
</tr>
<tr>
<td>adjacency lists</td>
<td>$E + V$</td>
<td>$1$</td>
<td>$\text{outdegree}(v)$</td>
<td>$\text{outdegree}(v)$</td>
</tr>
</tbody>
</table>

† disallows parallel edges
public class Graph
{
    private final int V;
    private Bag<Integer>[] adj;

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

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

    public Iterable<Integer> adj(int v)
    {
        return adj[v];
    }
}
public class Digraph
{
    private final int V;
    private Bag<Integer>[] adj;

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

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

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

- introduction
- digraph API
- depth-first search
- breadth-first search
- topological sort
Reachability

**Problem.** Find all vertices reachable from $s$ along a directed path.
Depth-first search in digraphs

Same method as for undirected graphs.

- Every undirected graph is a digraph (with edges in both directions).
- DFS is a digraph algorithm.

**DFS (to visit a vertex v)**

Mark vertex v.
Recursively visit all unmarked vertices w adjacent from v.
Depth-first search demo

To visit a vertex $v$:

- Mark vertex $v$ as visited.
- Recursively visit all unmarked vertices adjacent from $v$. 

![Directed Graph Diagram]

a directed graph
Depth-first search demo

To visit a vertex \( v \):
- Mark vertex \( v \) as visited.
- Recursively visit all unmarked vertices adjacent from \( v \).

<table>
<thead>
<tr>
<th>( v )</th>
<th>marked[]</th>
<th>edgeTo[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>T</td>
<td>–</td>
</tr>
<tr>
<td>1</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>T</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>T</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>T</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
<td>–</td>
</tr>
<tr>
<td>7</td>
<td>F</td>
<td>–</td>
</tr>
<tr>
<td>8</td>
<td>F</td>
<td>–</td>
</tr>
<tr>
<td>9</td>
<td>F</td>
<td>–</td>
</tr>
<tr>
<td>10</td>
<td>F</td>
<td>–</td>
</tr>
<tr>
<td>11</td>
<td>F</td>
<td>–</td>
</tr>
<tr>
<td>12</td>
<td>F</td>
<td>–</td>
</tr>
</tbody>
</table>

reachable from vertex 0
Depth-first search (in undirected graphs)

Recall code for undirected graphs.

```java
public class DepthFirstSearch {
    private boolean[] marked;

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

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

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

- true if connected to s
- constructor marks vertices connected to s
- recursive DFS does the work
- is vertex v is connected to s?
**Depth-first search (in directed graphs)**

Code for **directed** graphs identical to undirected one.

```java
public class DirectedDFS {
    private boolean[] marked;

    public DirectedDFS(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 visited(int v) {
        return marked[v];
    }
}
```

true if connected to s

constructor marks vertices connected to s

recursive DFS does the work

is vertex v reachable from s?
Reachability application: program control-flow analysis

Every program is a digraph.
- Vertex = basic block of instructions (straight-line program).
- Edge = jump.

Dead-code elimination.
Find (and remove) unreachable code.

Infinite-loop detection.
Determine whether exit is unreachable.
Reachability application: mark–sweep garbage collector

Every data structure is a digraph.
- Vertex = object.
- Edge = reference.

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

Reachable objects. Objects indirectly accessible by program (starting at a root and following a chain of pointers).
Reachability application: mark–sweep garbage collector

Mark–sweep algorithm.  [McCarthy, 1960]

- Mark: mark all reachable objects.
- Sweep: if object is unmarked, it is garbage (so add to free list).

Memory cost.  Uses 1 extra mark bit per object (plus DFS stack).
Depth-first search in digraphs summary

**DFS enables direct solution of simple digraph problems.**
- Reachability.
- Path finding.
  - Topological sort.
  - Directed cycle detection.

**Basis for solving difficult digraph problems.**
- 2-satisfiability.
- Directed Euler path.
- Strongly connected components.

---

**DEEP-First Search AND LINEAR Graph Algorithms**

ROBERT TARJAN†

*Abstract.* The value of depth-first search or “backtracking” as a technique for solving problems is illustrated by two examples. An improved version of an algorithm for finding the strongly connected components of a directed graph and an algorithm for finding the biconnected components of an undirected graph are presented. The space and time requirements of both algorithms are bounded by $k_1 V + k_2 E + k_3$, for some constants $k_1$, $k_2$, and $k_3$, where $V$ is the number of vertices and $E$ is the number of edges of the graph being examined.
4.2 Directed Graphs

- introduction
- digraph API
- depth-first search
- breadth-first search
- topological sort
Shortest directed paths

**Problem.** Find directed path from $s$ to each vertex that uses fewest edges.
Breadth-first search in digraphs

Same method as for undirected graphs.

- Every undirected graph is a digraph (with edges in both directions).
- BFS is a digraph algorithm.

**BFS (from source vertex s)**

Put s onto a FIFO queue, and mark s as visited.
Repeat until the queue is empty:
- remove the least recently added vertex v
- for each unmarked vertex adjacent from v:
  add to queue and mark as visited.

**Proposition.** In worst case, BFS computes directed path with fewest edges from s to each vertex in time proportional to $E + V$. 
Directed breadth-first search demo

Repeat until queue is empty:
- Remove vertex $v$ from queue.
- Add to queue all unmarked vertices adjacent from $v$ and mark them.

![Graph G](image-url)
Directed breadth-first search demo

Repeat until queue is empty:
• Remove vertex \( v \) from queue.
• Add to queue all unmarked vertices adjacent from \( v \) and mark them.

<table>
<thead>
<tr>
<th>( v )</th>
<th>edgeTo[]</th>
<th>marked[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>–</td>
<td>T</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>T</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>T</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>T</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>T</td>
</tr>
</tbody>
</table>

all done
MULTIPLE-SOURCE SHORTEST PATHS

Given a digraph and a set of source vertices, find shortest path from any vertex in the set to every other vertex.

Ex.  \( S = \{ 1, 7, 10 \} \).

- Shortest path to 4 is 7→6→4.
- Shortest path to 5 is 7→6→0→5.
- Shortest path to 12 is 10→12.

Q. How to implement multi-source shortest paths algorithm?
Suppose that you want to design a web crawler. Which graph-search algorithm should you use?

A. depth-first search
B. breadth-first search
C. either A or B
D. neither A nor B
## Web crawler output

### BFS crawl

http://www.princeton.edu  
http://www.w3.org  
http://ogp.me  
http://giving.princeton.edu  
http://www.princetonartmuseum.org  
http://www.goprincetontigers.com  
http://library.princeton.edu  
http://helpdesk.princeton.edu  
http://tigernet.princeton.edu  
http://alumni.princeton.edu  
http://gradschool.princeton.edu  
http://vimeo.com  
http://princetonusg.com  
http://artmuseum.princeton.edu  
http://jobs.princeton.edu  
http://odoc.princeton.edu  
http://blogs.princeton.edu  
http://www.facebook.com  
http://twitter.com  
http://www.youtube.com  
http://deimos.apple.com  
http://qeprize.org  
http://en.wikipedia.org  
...

### DFS crawl

http://www.princeton.edu  
http://deimos.apple.com  
http://www.youtube.com  
http://www.google.com  
http://news.google.com  
http://csi.gstatic.com  
http://googlenewsblog.blogspot.com  
http://labs.google.com  
http://groups.google.com  
http://imgl1.blogspot.com  
http://feeds.feedburner.com  
http://fusion.google.com  
http://insidesearch.blogspot.com  
http://agoogleaday.com  
http://static.googleusercontent.com  
http://searchresearch1.blogspot.com  
http://feedburner.google.com  
http://www.dot.ca.gov  
http://www.TahoeRoads.com  
http://www.LakeTahoeTransit.com  
http://www.laketahoe.com  
http://ethel.tahoeguide.com  
...


Breadth-first search in digraphs application: web crawler


Solution. [BFS with implicit digraph]
- Choose root web page as source $s$.
- Maintain a Queue of websites to explore.
- Maintain a Set of marked websites.
- Dequeue the next website and enqueue any unmarked websites to which it links.

Remark. Industrial-strength web crawlers use more sophisticated algorithms.
Bare-bones web crawler: Java implementation

Queue<String> queue = new Queue<String>();
SET<String> marked = new SET<String>();

String root = "http://www.princeton.edu";
queue.enqueue(root);
marked.add(root);

while (!queue.isEmpty())
{
    String v = queue.dequeue();
    StdOut.println(v);
    In in = new In(v);
    String input = in.readString();

    String regexp = "http://([\w.]+)([\w]+)";
    Pattern pattern = Pattern.compile(regexp);
    Matcher matcher = pattern.matcher(input);

    while (matcher.find())
    {
        String w = matcher.group();
        if (!marked.contains(w))
        {
            marked.add(w);
            q.enqueue(w);
        }
    }
}
4.2 Directed Graphs

- introduction
- digraph API
- depth-first search
- breadth-first search
- topological sort
Combinational circuit

Vertex = logical gate; edge = wire.
WordNet digraph

Vertex = synset; edge = hypernym relationship.

https://wordnet.princeton.edu
Vertex = revision of repository; edge = revision relationship.
Precedence scheduling

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

**Digraph model.** vertex = task; edge = precedence constraint.

0. Algorithms
1. Complexity Theory
2. Machine Learning
3. Intro to CS
4. Cryptography
5. Scientific Computing
6. Discrete Math

---

**tasks**

**precedence constraint graph**

**feasible schedule**
Topological sort

**DAG.** Directed **acyclic** graph.

**Topological sort.** Redraw DAG so all edges point upwards.

Edges in DAG define a "partial order" for vertices

Directed edges:

\[
0 \rightarrow 5 \quad 0 \rightarrow 2 \\
0 \rightarrow 1 \quad 3 \rightarrow 6 \\
3 \rightarrow 5 \quad 3 \rightarrow 4 \\
5 \rightarrow 2 \quad 6 \rightarrow 4 \\
6 \rightarrow 0 \quad 3 \rightarrow 2 \\
1 \rightarrow 4
\]
Suppose that you want to topologically sort the vertices in a DAG. Which graph-search algorithm should you use?

A. depth-first search
B. breadth-first search
C. either A or B
D. neither A nor B
Topological sort demo

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

```
Topological sort demo

- Run depth-first search.
- Return vertices in reverse DFS postorder.
```

```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
Topological sort demo

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

DFS postorder
4 1 2 5 0 6 3

Topological order (reverse DFS postorder)
3 6 0 5 2 1 4

done
Depth-first search: reverse postorder

```java
public class DepthFirstOrder {
    private boolean[] marked;
    private Stack<Integer> reversePostorder;

    public DepthFirstOrder(Digraph G) {
        reversePostorder = new Stack<Integer>();
        marked = new boolean[G.V()];
        for (int v = 0; v < G.V(); v++)
            if (!marked[v]) dfs(G, v);
    }

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

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

returns all vertices in “reverse DFS postorder”
Why is the reverse DFS postorder a topological order?

- First vertex in DFS postorder (last in topological order) has outdegree 0.
- Second vertex in DFS postorder can point only to first vertex.
- ...

Topological sort in a DAG: intuition

DFS postorder
4 1 2 5 0 6 3

topological order
(reverse DFS postorder)
3 6 0 5 2 1 4
Topological sort in a DAG: correctness proof

**Proposition.** Reverse DFS postorder of a DAG is a topological order.

**Pf.** Consider any edge \( v \rightarrow w \). When \( \text{dfs}(v) \) is called:

- **Case 1:** \( \text{dfs}(w) \) has already been called and returned.
  - thus, \( w \) appears before \( v \) in DFS postorder

- **Case 2:** \( \text{dfs}(w) \) has not yet been called.
  - \( \text{dfs}(w) \) will get called directly or indirectly by \( \text{dfs}(v) \)
  - so, \( \text{dfs}(w) \) will return before \( \text{dfs}(v) \) returns
  - thus, \( w \) appears before \( v \) in DFS postorder

- **Case 3:** \( \text{dfs}(w) \) has already been called, but has not yet returned.
  - function-call stack contains directed path from \( w \) to \( v \)
  - edge \( v \rightarrow w \) would complete a directed cycle
  - contradiction (it’s a DAG)
Proposition. A digraph has a topological order iff no directed cycle.

Pf.
- If directed cycle, topological order impossible.
- If no directed cycle, DFS-based algorithm finds a topological order.

Goal. Given a digraph, find a directed cycle.

Solution. DFS. What else? See textbook.
Directed cycle detection application: precedence scheduling

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

![Directed cycle detection application: precedence scheduling](http://xkcd.com/754)

**Remark.** A directed cycle implies scheduling problem is infeasible.
Directed cycle detection application: cyclic inheritance

The Java compiler does directed cycle detection.

```java
public class A extends B {
    ...
}

public class B extends C {
    ...
}

public class C extends A {
    ...
}
```

% javac A.java
A.java:1: cyclic inheritance involving A
public class A extends B { }
^
1 error
Directed cycle detection application: spreadsheet recalculation

Microsoft Excel does directed cycle detection.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>&quot;=B1 + 1&quot;</td>
<td>&quot;=C1 + 1&quot;</td>
<td>&quot;=A1 + 1&quot;</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Microsoft Excel cannot calculate a formula. Cell references in the formula refer to the formula’s result, creating a circular reference. Try one of the following:

- If you accidentally created the circular reference, click OK. This will display the Circular Reference toolbar and help for using it to correct your formula.
- To continue leaving the formula as it is, click Cancel.
### Digraph-processing summary: algorithms of the day

<table>
<thead>
<tr>
<th>Problem</th>
<th>Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single-source reachability in a digraph</td>
<td>DFS/BFS</td>
</tr>
<tr>
<td>Shortest path in a digraph</td>
<td>BFS</td>
</tr>
<tr>
<td>Topological sort in a DAG</td>
<td>DFS</td>
</tr>
</tbody>
</table>