**4.2.1 What is the maximum number of edges in a digraph with V vertices and no parallel edges? What is the minimum number of edges in a digraph with V vertices, none of which are isolated?**

Maximum number of edges: V(V-1)

Minimum number of edges: V - 1

**4.2.2 Draw, in the style of the figure in the text (page 524), the adjacency lists built by Digraph’s input stream constructor for the file**

|  |  |
| --- | --- |
| 0 | 6 |
| 1 | 11 |
| 2 | 3, 0 |
| 3 | 6, 10 |
| 4 | 1 |
| 5 | 2, 10 |
| 6 | 2 |
| 7 | 11, 8 |
| 8 | 4, 1 |
| 9 |  |
| 10 | 3 |
| 11 | 8 |

**4.2.3 Create a copy constructor for Digraph that takes as input a digraph G and creates and initializes a new copy of the digraph. Any changes a client makes to G should not affect the newly created digraph.**

[Same as Undirected Graph]

**4.2.4 Add a method hasEdge() to Digraph which takes two int arguments v and w and returns true if the graph has an edge v->w, false otherwise.**

[Same as Undirected Graph]

**4.2.5 Modify Digraph to disallow parallel edges and self-loops.**

[Same as Undirected Graph]

**4.2.6 Develop a test client for Digraph.**

**4.2.7 The indegree of a vertex in a digraph is the number of directed edges that point to that vertex. The outdegree of a vertex in a digraph is the number of directed edges that emanate from that vertex. No vertex is reachable from a vertex of outdegree 0, which is called a sink; a vertex of indegree 0, which is called a source, is not reachable from any other vertex. A digraph where self-loops are allowed and every vertex has outdegree 1 is called a map (a function from the set of integers from 0 to V–1 onto itself). Write a program Degrees.java that implements the** **following API:**

**public class Degrees**

**Degrees(Digraph G) constructor**

**int indegree(int v) indegree of v**

**int outdegree(int v) outdegree of v**

**Iterable<Integer> sources() sources**

**Iterable<Integer> sinks() sinks**

**boolean isMap() is G a map?**

**public** Degrees(DiGraph g)

{

**for**(**int** v = 0 ; v < g.verticeCount() ; v++)

{

**int** out = 0;

**for**(**int** u : g.adj(v))

{

indegree[u]++;

out++;

}

outdegree[v] = out;

}

**for**(**int** v = 0 ; v < g.verticeCount() ; v++)

{

**if**(indegree[v] == 0) sources.add(v);

**if**(outdegree[v] == 0) sinks.add(v);

**if**(outdegree[v] != 1) isMap = **false**;

}

}

**4.2.8 Draw all the nonisomorphic DAGs with two, three, four, and five vertices (see Exercise 4.1.28).**

**4.2.9 Write a method that checks whether or not a given permutation of a DAG’s vertices is a topological order of that DAG.**

**public** **static** **boolean** isTopological(DiGraph g, **int**[] vSeq)

{

**for**(**int** i = vSeq.length - 1 ; i > 0 ; i--)

{

DirectedDfsPath ddp = **new** DirectedDfsPath(g, vSeq[i]);

**for**(**int** j = i - 1 ; j >= 0 ; j--)

**if**(ddp.pathTo(vSeq[j]) != **null**)

**return** **false**;

}

**return** **true**;

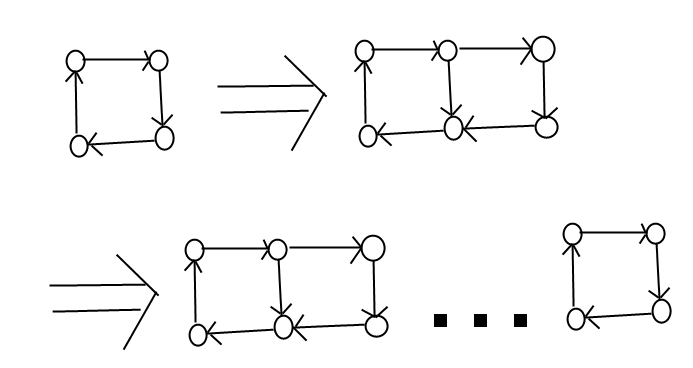
}

**4.2.10 Given a DAG, does there exist a topological order that cannot result from applying a DFS-based algorithm, no matter in what order the vertices adjacent to each vertex are chosen? Prove your answer.**

Yes, this is possible.

Consider a graph contains u, v, w, where u, v are precedent (not directly) of w, but there’s no path from u to v. Then u, v … w would be acceptable topological order but this cannot be achieved by DFS.

**4.2.11 Describe a family of sparse digraphs whose number of directed cycles grows exponentially in the number of vertices.**

****

Let V denotes the vertices of the graph, C(V) denotes the number of directed cycles of the graph.

C(V) = 2 \* C(V - 2) + 2

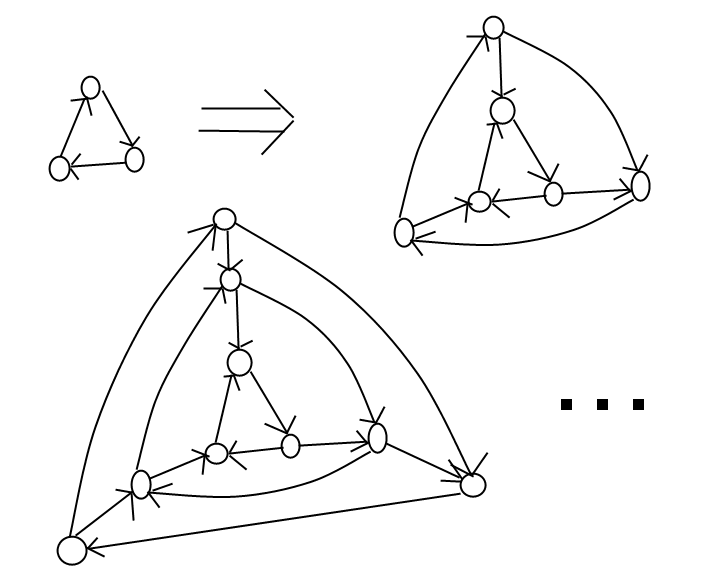
= 4 \* C(V - 4) + 2 + 4

=..

= 2^((V -4 ) / 2) \* C(4) + SUM(i from 1 to (V-4)/2, 2^i)

= 6 \* 2^((V -4 ) / 2) -2

**Analysis above is WRONG.**



Let C denotes the directed cycle in the graph and V is the number of vertices in the graph.

Then C(V) >= 2 \* C(V - 3) ~ 2^(V)

**4.2.12 How many edges are there in the transitive closure of a digraph that is a simple directed path with V vertices and V–1 edges?**

V + V-1 +… + 2 + 1 = V(V+1)/2

**4.2.13 Give the transitive closure of the digraph with ten vertices and these edges: 3->7 1->4 7->8 0->5 5->2 3->8 2->9 0->6 4->9 2->6 6->4**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 0 | T |  | T |  | T | T | T |  |  | T |
| 1 |  | T |  |  | T |  |  |  |  | T |
| 2 |  |  | T |  |  |  | T |  |  | T |
| 3 |  |  |  | T |  |  |  | T | T |  |
| 4 |  |  |  |  | T |  |  |  |  | T |
| 5 |  |  | T |  | T | T | T |  |  | T |
| 6 |  |  |  |  | T |  | T |  |  | T |
| 7 |  |  |  |  |  |  |  | T | T |  |
| 8 |  |  |  |  |  |  |  |  | T |  |
| 9 |  |  |  |  |  |  |  |  |  | T |

**4.2.14 Prove that the strong components in GR are the same as in G.**

For any 2 vertices u, v in Gr, if u, v is in the same SCC, then there’s a path u ~ v and there’s a path v ~ u. The first path is path v ~ u in G and the second is u ~ v in G, meaning that they’re in the same SCC in G too.

If there’re not at the same SCC then either no path from u ~ v or v ~ u or both and in G this property would not change.

**4.2.15 What are the strong components of a DAG?**

A DAG with V vertices has V strong components, i.e. every vertex is a single strong components.

**4.2.16 What happens if you run Kosaraju’s algorithm on a DAG?**

DFS will be called exactly once for every vertex.

A simple prove:

The reverse post order of the GR is the “reversed” topological sort of G, so for every vertex v, its adjacent list should either be empty or all vertices in it have been marked.

**4.2.17 True or false: The reverse postorder of a graph’s reverse is the same as the postorder of the graph.**

False

Consider graph G =

0 2

1 2

GR

2 0

2 1

Reverse post order of GR = 2 1 0

Post order of G = 2 0 1

**4.2.18 Compute the memory usage of a Digraph with V vertices and E edges, under the memory cost model of Section 1.4.**

**4.2.19 *Topological sort and BFS.* Explain why the following algorithm does not necessarily produce a topological order: Run BFS, and label the vertices by increasing distance to their respective source.**

Directly apply BFS for topological sort may not generate desired result because for an edge v->w, BFS may visit w first then when v is visited, w will not be considered for it has been marked. This means w may have smaller distance than v and generates incorrect topological order.

However, if we run BFS for every vertex regardless if the adjacent vertices are marked and label its adjacent vertices with larger distance then sort all vertices by its distance then we get the correct topological order because…

For any edge v->w, w will have larger distance value than v.

This algorithm runs at V(V+E) +VlgV at worst case.

**4.2.20 *Directed Eulerian cycle.* An Eulerian cycle is a directed cycle that contains each edge exactly once. Write a graph client Euler that finds an Eulerian cycle or reports that no such tour exists. *Hint* : Prove that a digraph *G* has a directed Eulerian cycle if and only if *G* is connected and each vertex has its indegree equal to its outdegree.**

**4.2.21 *LCA of a DAG.* Given a DAG and two vertices v and w, find the *lowest common ancestor* (LCA) of v and w. The LCA of v and w is an ancestor of v and w that has no descendants that are also ancestors of v and w. Computing the LCA is useful in multiple inheritance in programming languages, analysis of genealogical data (find degree of inbreeding in a pedigree graph), and other applications. *Hint* : Define the height of a vertex v in a DAG to be the length of the longest path from a root to v. Among vertices that are ancestors of both v and w, the one with the greatest height is an LCA of v and w.**

**4.2.22 *Shortest ancestral path.* Given a DAG and two vertices v and w, find the *shortest ancestral path* between v and w. An ancestral path between v and w is a common ancestor x along with a shortest path from v to x and a shortest path from w to x. The shortest ancestral path is the ancestral path whose total length is minimized. *Warmup*: Find a DAG where the shortest ancestral path goes to a common ancestor x that is not an LCA.**

***Hint*: Run BFS twice, once from v and once from w.**

**4.2.23 *Strong component.* Describe a linear-time algorithm for computing the strong connected component containing a given vertex v. On the basis of that algorithm, describe a simple quadratic algorithm for computing the strong components of a digraph.**

**4.2.24 *Hamiltonian path in DAGs.* Given a DAG, design a linear-time algorithm to determine whether there is a directed path that visits each vertex exactly once.**

***Answer* : Compute a topological sort and check if there is an edge between each consecutive**

**pair of vertices in the topological order.**

**4.2.25 *Unique topological ordering.* Design an algorithm to determine whether a digraph has a unique topological ordering. *Hint* : A digraph has a unique topological ordering if and only if there is a directed edge between each pair of consecutive vertices in the topological order (i.e., the digraph has a Hamiltonian path). If the digraph has multiple topological orderings, then a second topological order can be obtained by swapping a pair of consecutive vertices.**