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

Max = V \* (V-1) i.e. every vertex has an edge to every other vertex

Min = V – 1 i.e. it’s a line.

**4.1.2** Draw, in the style of the figure in the text (page 524), the adjacency lists built by Graph’s input stream constructor for the file tinyGex2.txt depicted at left.

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

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

**public** Graph(Graph g)

{

V = g.V;

E = g.E;

adj = (Bag<Integer>[]) **new** Bag[V];

**for**(**int** i = 0 ; i < V ; i++)

**for**(Integer e : g.adj[i])

adj[i].add(e);

}

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

public boolean hasEdge(int u, int v){return adj[u].contains(v);}

**4.1.5** Modify Graph to disallow parallel edges and self-loops.

**public** **void** addEdge(**int** u, **int** v)

{

**if**(adj[u].contains(v) || u == v) **return**;

adj[u].add(v);

adj[v].add(u);

E++;

}

**4.1.6** Consider the four-vertex graph with edges 0-1, 1-2, 2-3, and 3-0. Draw an array of adjacency-lists that could *not* have been built calling addEdge() for these edges *no matter what order*.

|  |  |
| --- | --- |
| 0 | 1 3 |
| 1 | 2 0 |
| 2 | 3 1 |
| 3 | 0 2 |

**4.1.7** Develop a test client for Graph that reads a graph from the input stream named as command-line argument and then prints it, relying on toString().

**4.1.8** Develop an implementation for the Search API on page 528 that uses UF, as described in the text.

**public** **class** UnionFindGraphSearch {

**private** PathCompressedWeightedQuickUnion unionFind;

**public** UnionFindGraphSearch(Graph g)

{

unionFind = **new** PathCompressedWeightedQuickUnion(g.verticeCount());

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

**for**(**int** e : g.adj(i))

unionFind.union(i, e);

}

**public** **boolean** isConnected(**int** u, **int** v)

{

**return** unionFind.connected(u, v);

}

**public** **int** connectedCount(**int** v)

{

//do not count v itself

**return** unionFind.size(v) - 1;

}

/\*\*

\* **@param** args

\*/

**public** **static** **void** main(String[] args) {

Graph g = GraphFileBuilder.*BuildGraph*("tinyG.txt");

UnionFindGraphSearch ufgs = **new** UnionFindGraphSearch(g);

System.*out*.println(ufgs.isConnected(0, 4));

System.*out*.println(ufgs.isConnected(0, 12));

System.*out*.println(ufgs.connectedCount(0));

System.*out*.println(ufgs.connectedCount(12));

}

}

**4.1.9** Show, in the style of the figure on page 533, a detailed trace of the call dfs(0) for the graph built by Graph’s input stream constructor for the file tinyGex2.txt (see Exercise 4.1.2). Also, draw the tree represented by edgeTo[].

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **T** |  |  |  |  |  |  |  |  |  |  |  |

dfs(0)

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  | **0** |  |  |  |  |  |

dfs(6)

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| T |  |  |  |  |  | **T** |  |  |  |  |  |

check 0

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  | **6** |  |  | 0 |  |  |  |  |  |

dfs(3)

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| T |  |  | **T** |  |  | T |  |  |  |  |  |

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | **3** | 6 |  |  | 0 |  |  |  |  |  |

dfs(2)

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| T |  | **T** | T |  |  | T |  |  |  |  |  |

check3

check0

check6

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | 3 | 6 |  | **2** | 0 |  |  |  |  |  |

dfs(5)

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| T |  | T | T |  | **T** | T |  |  |  |  |  |

check2

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | 3 | 6 |  | 2 | 0 |  |  |  | **5** |  |

dfs(10)

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| T |  | T | T |  | T | T |  |  |  | **T** |  |

check3

check5

10 done

check0

5 done

2 done

check6

check10

3 done

check2

6 done

check2

check5

0 done

ALL DONE

10 🡪 5 🡪 2 🡪 3 🡪 6 🡪 0

**4.1.10** Prove that every connected graph has a vertex whose removal (including all adjacent edges) will not disconnect the graph, and write a DFS method that finds such a vertex. *Hint*: Consider a vertex whose adjacent vertices are all marked.

Consider a vertex whose all adjacent vertices are all marked during a search on graph (dfs or bfs). This vertex can be removed from graph without disconnecting the graph because its adjacent vertex can still be reached by other paths.

There must be such at least on such vertex because any search method successfully visits all vertexes in connected graph and terminates. The last visited vertex would be one, any other with degree 1 would also fulfill the condition.

**public** **static** **int** getRemovableVertex(Graph g)

{

**boolean**[] marked = **new** **boolean**[g.verticeCount()];

**return** *getRemovableVertex*(g, 0, marked);

}

**public** **static** **int** getRemovableVertex(Graph g, **int** s, **boolean**[] marked)

{

marked[s] = **true**;

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

**if**(!marked[v])

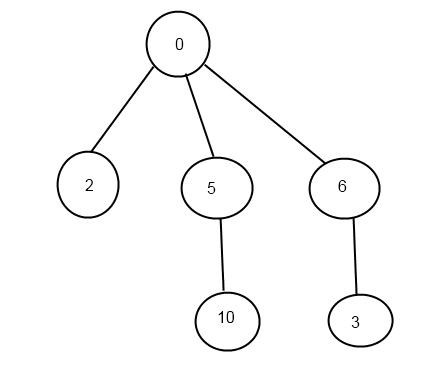
**return** *getRemovableVertex*(g, v, marked);

**return** s;

}

**4.1.11** Draw the tree represented by edgeTo[] after the call bfs(G, 0) in Algorithm 4.2 for the graph built by Graph’s input stream constructor for the file tinyGex2.txt (see Exercise 4.1.2).

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | 0 | 6 |  | 0 | 0 |  |  |  | 5 |  |  |



**4.1.12** What does the BFS tree tell us about the distance from v to w when neither is at the root?

The distance of v to w is in BFS tree is the shortest distance of those vertex in graph. Because otherwise root to v then to w is not the shortest path from root to w, a contradiction of BFS generates the shortest paths from root to every reachable vertex.

**4.1.13** Add a distTo() method to the BreadthFirstPaths API and implementation, which returns the number of edges on the shortest path from the source to a given vertex. A distTo() query should run in constant time.

**private** **void** bfs(**int** s)

{

marked[s] = **true**;

ArrayQueue<Integer> q = **new** ArrayQueue<Integer>();

ArrayQueue<Integer> disQ = **new** ArrayQueue<Integer>();

q.enqueue(s);

disQ.enqueue(0);

**while**(!q.isEmpty())

{

**int** u = q.dequeue();

distTo[u] = disQ.dequeue();

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

{

**if**(!marked[v])

{

marked[v]=**true**;

edgeTo[v] = u;

q.enqueue(v);

disQ.enqueue(distTo[u] + 1);

}

}

}

}

**public** **int** distTo(**int** v)

{

**return** distTo[v];

}

**4.1.14** Suppose you use a stack instead of a queue when running breadth-first search. Does it still compute shortest paths?

No. This is actually is the simulation of DFS.

**4.1.15** Modify the input stream constructor for Graph to also allow adjacency lists from standard input (in a manner similar to SymbolGraph), as in the example tinyGadj.txt shown at right. After the number of vertices and edges, each line contains a vertex and its list of adjacent vertices.

**4.1.16** The *eccentricity* of a vertex v is the the length of the shortest path from that vertex to the furthest vertex from v. The *diameter* of a graph is the maximum eccentricity of any vertex. The *radius* of a graph is the smallest eccentricity of any vertex. A *center* is a vertex whose eccentricity is the radius. Implement the following API:

public class GraphProperties

GraphProperties(Graph G) *constructor* (*exception if* G *not connected*)

int eccentricity(int v) *eccentricity of* v

int diameter() *diameter of* G

int radius() *radius of* G

int center() *a center of* G

**private** **void** bfs(Graph g, **int** s)

{

**boolean**[] marked = **new** **boolean**[g.verticeCount()];

ArrayQueue<Integer> q = **new** ArrayQueue<Integer>();

ArrayQueue<Integer> disQ = **new** ArrayQueue<Integer>();

marked[s] = **true**;

q.enqueue(s);

disQ.enqueue(0);

**while**(!q.isEmpty())

{

**int** u = q.dequeue();

**int** dis = disQ.dequeue();

**if**(dis > eccentricity[s])

eccentricity[s] = dis;

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

{

**if**(!marked[v])

{

marked[v] = **true**;

q.enqueue(v);

disQ.enqueue(dis + 1);

}

}

}

**if**(eccentricity[s] > diameter)

diameter = eccentricity[s];

**if**(eccentricity[s] < radius)

{

radius = eccentricity[s];

center = s;

}

}

**4.1.18** The *girth* of a graph is the length of its shortest cycle. If a graph is acyclic, then its girth is infinite. Add a method girth() to GraphProperties that returns the girth of the graph. *Hint* : Run BFS from each vertex. The shortest cycle containing s is a shortest path from s to some vertex v, plus the edge from v back to s.

**public** **int** girth()

{

**int** n = g.verticeCount();

**for**(**int** s = 0 ; s < n; s++)

{

**int** circleLen = bfsFindShortestCircle(s);

**if**(circleLen < **this**.girth)

**this**.girth = circleLen;

}

**return** **this**.girth;

}

**private** **int** bfsFindShortestCircle(**int** s) {

**int** n = g.verticeCount();

ArrayQueue<Integer> q = **new** ArrayQueue<Integer>();

**int**[] edgeTo = **new** **int**[n];

**boolean**[] marked = **new** **boolean**[n];

marked[s] = **true**;

edgeTo[s] = s;

q.enqueue(s);

**while**(!q.isEmpty())

{

**int** u = q.dequeue();

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

{

**if**(!marked[v])

{

marked[v] = **true**;

edgeTo[v] = u;

q.enqueue(v);

}

//marked and it's not the source vertex, this is the shorted circle

**else** **if**(v != edgeTo[u])

{

**return** calcCircleLen(u, v, edgeTo);

}

}

}

**return** Integer.*MAX\_VALUE*;

}

**private** **int** calcCircleLen(**int** u, **int** v, **int**[] edgeTo)

{

**int** ret = 1;

**while**(u != v)

{

**if**(u != edgeTo[u])

{

u = edgeTo[u];

ret++;

}

**if**(v != edgeTo[v])

{

v = edgeTo[v];

ret++;

}

}

**return** ret;

}

**4.1.19** Show, in the style of the figure on page 545, a detailed trace of CC for finding the connected components in the graph built by Graph’s input stream constructor for the file tinyGex2.txt (see Exercise 4.1.2).

dfs(0)

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **T** |  |  |  |  |  |  |  |  |  |  |  |

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** |  |  |  |  |  |  |  |  |  |  |  |

dfs(6)

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| T |  |  |  |  |  | **T** |  |  |  |  |  |

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 |  |  |  |  |  | **0** |  |  |  |  |  |

check0

dfs(3)

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| T |  |  | **T** |  |  | T |  |  |  |  |  |

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 |  |  | **0** |  |  | 0 |  |  |  |  |  |

dfs(2)

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| T |  | **T** | T |  |  | T |  |  |  |  |  |

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 |  | 0 | 0 |  |  | 0 |  |  |  |  |  |

check3

check0

check6

dfs(5)

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| T |  | T | T |  | **T** | T |  |  |  |  |  |

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 |  | 0 | 0 |  | **0** | 0 |  |  |  |  |  |

check2

dfs(10)

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| T |  | T | T |  | T | T |  |  |  | **T** |  |

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 |  | 0 | 0 |  | 0 | 0 |  |  |  | **0** |  |

check3

check5

10 done

check0

5 done

2 done

check6

check10

3 done

check2

6 done

check2

check5

0 done

**count++**

dfs(1)

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| T | **T** | T | T |  | T | T |  |  |  | **T** |  |

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | **1** | 0 | 0 |  | 0 | 0 |  |  |  | **0** |  |

dfs(11)

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| T | T | T | T |  | T | T |  |  |  | T | **T** |

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 0 | 0 |  | 0 | 0 |  |  |  | 0 | 1 |

check1

dfs(7)

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| T | T | T | T |  | T | T | **T** |  |  | T | T |

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 0 | 0 |  | 0 | 0 | **1** |  |  | 0 | 1 |

dfs(8)

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| T | T | T | T |  | T | T | T | **T** |  | T | T |

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 0 | 0 |  | 0 | 0 | 1 | **1** |  | 0 | 1 |

11 done

dfs(4)

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| T | T | T | T | **T** | T | T | T | T |  | T | T |

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 0 | 0 | **1** | 0 | 0 | 1 | 1 |  | 0 | 1 |

check8

check1

4 done

1 done

**count++**

dfs(9)

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| T | T | T | T | T | T | T | T | T | **T** | T | T |

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | **2** | 0 | 1 |

ALL DONE.

**4.1.20** Show, in the style of the figures in this section, a detailed trace of Cycle for finding a cycle in the graph built by Graph’s input stream constructor for the file tinyGex2.txt (see Exercise 4.1.2). What is the order of growth of the running time of the Cycle constructor, in the worst case?

Worst case: ~(V + E)

**4.1.21** Show, in the style of the figures in this section, a detailed trace of TwoColor for finding a two-coloring of the graph built by Graph’s input stream constructor for the file tinyGex2.txt (see Exercise 4.1.2). What is the order of growth of the running time of the TwoColor constructor, in the worst case?

Worst case: ~ (V + E)

**4.1.22** Run SymbolGraph with movies.txt to find the Kevin Bacon number of this year’s Oscar nominees.

Bacon, Kevin

Wild Things (1998)

Dillon, Matt (I)

Beautiful Girls (1996)

Portman, Natalie

===================================

Bacon, Kevin

Where the Truth Lies (2005)

Firth, Colin

===================================

Bacon, Kevin

Trapped (2002)

Fanning, Dakota

Hide and Seek (2005)

Leo, Melissa

===================================

Bacon, Kevin

Where the Truth Lies (2005)

Wateridge, Sarah

Batman Begins (2005)

Bale, Christian

===================================

**4.1.23** Write a program BaconHistogram that prints a histogram of Kevin Bacon numbers, indicating how many performers from movies.txt have a Bacon number of 0, 1, 2, 3, ... . Include a category for those who have an infinite number (not connected to Kevin Bacon).

**4.1.24** Compute the number of connected components in movies.txt, the size of the largest component, and the number of components of size less than 10. Find the eccentricity, diameter, radius, a center, and the girth of the largest component in the graph. Does it contain Kevin Bacon?

**4.1.25** Modify DegreesOfSeparation to take an int value y as a command-line argument

and ignore movies that are more than y years old.

**4.1.26** Write a SymbolGraph client like DegreesOfSeparation that uses *depth-first*

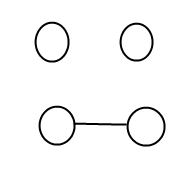
search instead of breadth-first search to find paths connecting two performers, producing

output like that shown on the facing page.

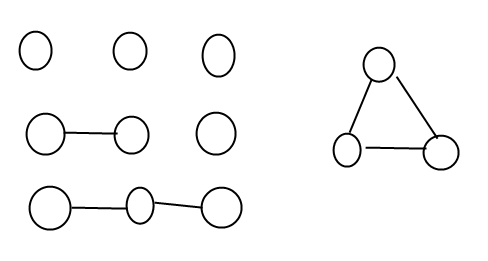
**4.1.27** Determine the amount of memory used by Graph to represent a graph with *V* vertices and *E* edges, using the memory-cost model of Section 1.4.

**4.1.28** Two graphs are *isomorphic* if there is a way to rename the vertices of one to make it identical to the other. Draw all the nonisomorphic graphs with two, three, four, and five vertices.

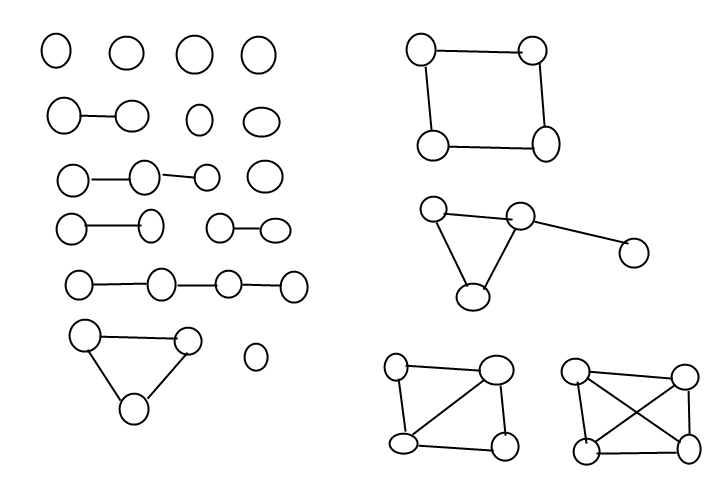
2



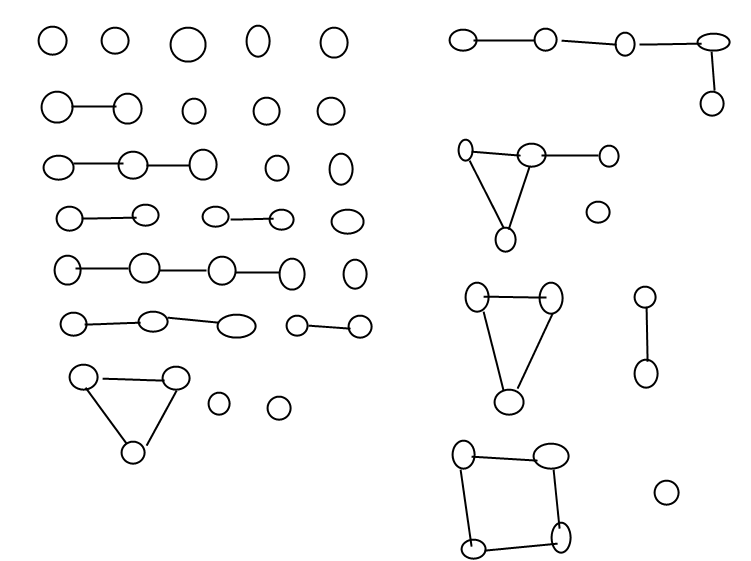
3

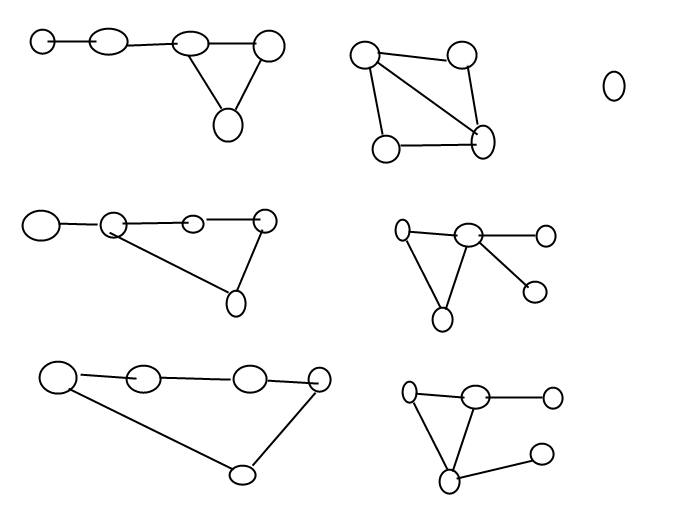


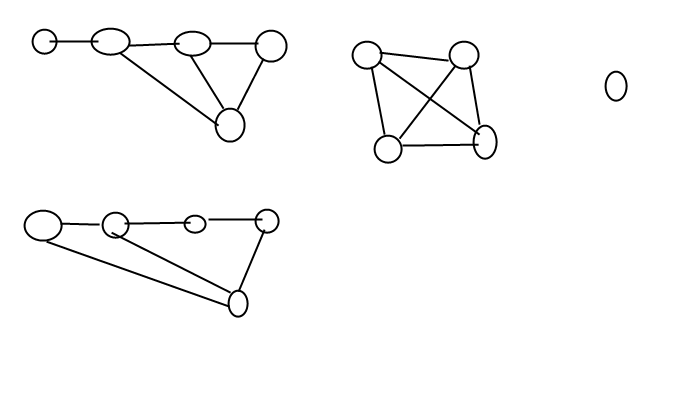
4

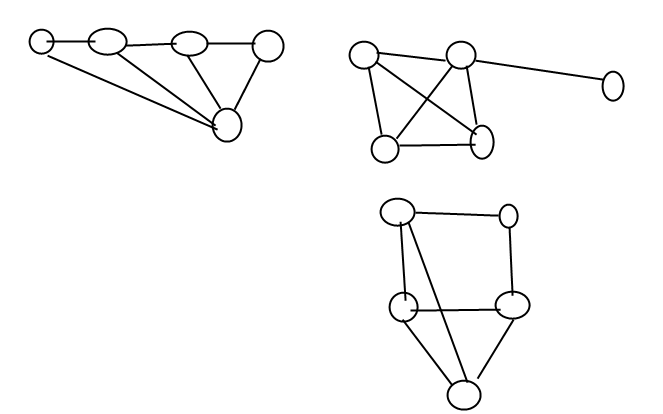


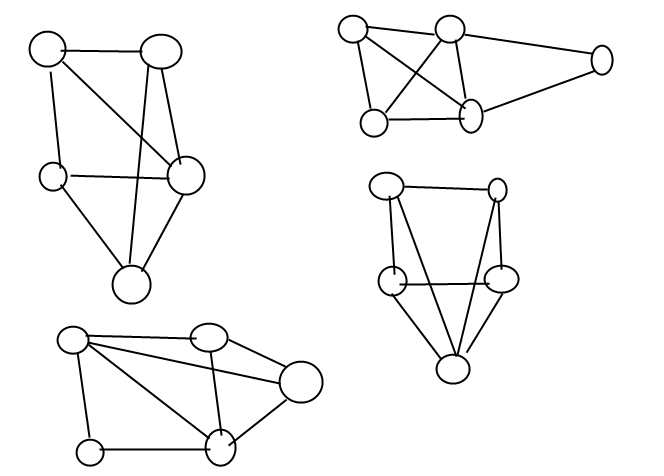
5

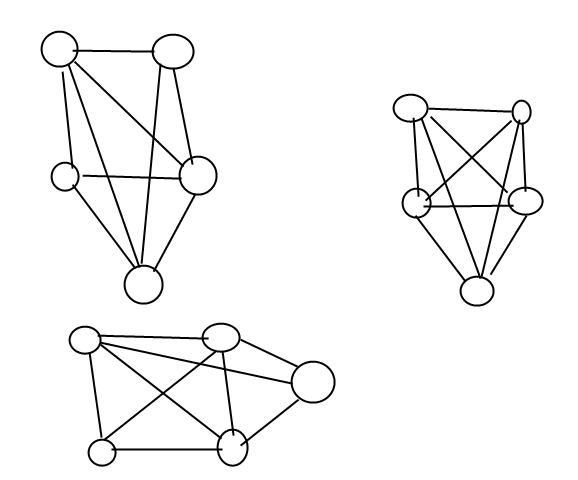












**4.1.29** Modify Cycle so that it works even if the graph contains self-loops and parallel edges.

**private** **void** gfs(**int** v, **int** from)

{

marked[v] = **true**;

**boolean** fromEdgeSeen = **false**;

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

{

**if**(u == from && !fromEdgeSeen)

fromEdgeSeen = **true**;

**else** **if** (!marked[u])

gfs(u, v);

//visiting marked vertex and if it is the from vertex,

//it is not the first time we've seen it, so it must fall in

//1 of these 3 cases:

// a. a simple cycle.

// b. a parallel edge.

// c. a self-loop.

// either way, it is a cycle.

**else**

hasCycle = **true**;

}

}

**4.1.30** *Eulerian and Hamiltonian cycles.* Consider the graphs defined by the following

four sets of edges:

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

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

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

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

Which of these graphs have Euler cycles (cycles that visit each edge exactly once)?

Which of them have Hamilton cycles (cycles that visit each vertex exactly once)?

1. has no Euler cycles but has Hamilton cycles = 0, 3, 6, 9, 2, 5, 8, 7, 4, 1
2. has
   1. Euler cycles 8-8, 8-5, 5-2, 2-0, 0-3, 3-0, 0-1, 1-3, 3-6, 6-5, 5-9, 9-6, 6-7, 7-4, 4-8
   2. Hamilton cycles: 2, 0, 1, 3, 6, 7, 4, 8, 5, 9
3. has no Euler cycles but has Hamilton cycles = 0, 3, 6 ,7, 4, 8, 5, 9, 2, 1
4. has no Euler cycles but has Hamilton cycles = 1, 6, 2, 0, 5, 4, 7, 3, 9, 8

**4.1.31** *Graph enumeration.* How many different undirected graphs are there with *V* vertices and *E* edges (and no parallel edges)?

**4.1.32** *Parallel edge detection.* Devise a linear-time algorithm to count the parallel edges in a graph.

**public** ParallelEdgeDetector(Graph g)

{

**this**.g = g;

marked = **new** **boolean**[g.verticeCount()];

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

**if**(!marked[i])

dfsToFindParallel(i, -1);

}

**private** **void** dfsToFindParallel(**int** s, **int** from)

{

marked[s] = **true**;

**boolean** hasSeenSource = **false**;

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

{

**if**(v == from)

{

**if**(hasSeenSource) parallelCount++;

**else** hasSeenSource = **true**;

}

**else** **if**(!marked[v])

dfsToFindParallel(v, s);

}

}

**public** **int** parallelCount()

{

**return** parallelCount;

}

**4.1.33** *Odd cycles.* Prove that a graph is two-colorable (bipartite) if and only if it contains no odd-length cycle.

🡺

When graph is 2-colorable then for every vertex in the cycle (if there’s a one), the vertex must be in color A then color B alternatively. This indicates the length of cycle must be even.

🡸

Let “s” be an arbitrary vertex in the graph. We partition the vertices of the graph into 2 sets:

X = {x| the shortest distance between x and s is odd};

Y = {y| the shortest distance between y and s is even};

Obviously X, Y is a partition of the graph for it partitioned all vertices of graph and join set of X and Y is empty.

Now we show X, Y is a bipartite partition, i.e. for any different x1, x2, in X, there’s no edge connecting them.

Prove by contradiction, suppose for a, b in X and there’s an edge e directly connecting them.

Then consider the shortest path s….a and s….b.

Let c be the last identical vertex between path s…a and s…b then c…a and c…b must be either odd or even (by the set definition of X).

Then the cycle c…a.b…c is an odd cycle, a contradiction.

**4.1.34** *Symbol graph.* Implement a one-pass SymbolGraph (it need not be a Graphclient). Your implementation may pay an extra log *V* factor for graph operations, for symbol-table lookups.

**4.1.35** *Biconnectedness.* A graph is *biconnected* if every pair of vertices is connected by two disjoint paths. An *articulation point* in a connected graph is a vertex that would disconnect the graph if it (and its adjacent edges) were removed. Prove that any graph with no articulation points is biconnected. *Hint* : Given a pair of vertices s and t and a path connecting them, use the fact that none of the vertices on the path are articulation points to construct two disjoint paths connecting s and t.

**4.1.36** *Edge connectivity.* A *bridge* in a graph is an edge that, if removed, would separate a connected graph into two disjoint subgraphs. A graph that has no bridges is said to be *edge connected*. Develop a DFS-based data type for determing whether a given graph is edge connected.

A straight forward algorithm:

Boolean IsEdgeConnected(Graph g)

{

Foreach(Edge e in graph.AllEdges)

{

Graph g-e = graph – e;

ConnectedComponent cc = new ConnectedComponent(g-e);

if(cc.ccCount > 1)

return false;

}

return true;

}

The time complexity of this algorithm is ~ E \* (V+E)

**4.1.37** *Euclidean graphs.* Design and implement an API EuclideanGraph for graphs whose vertices are points in the plane that include coordinates. Include a method show() that uses StdDraw to draw the graph.