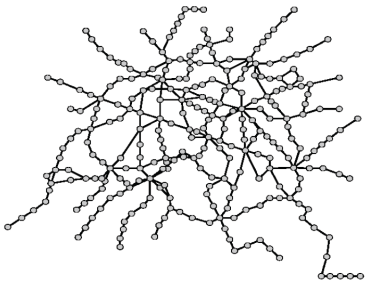
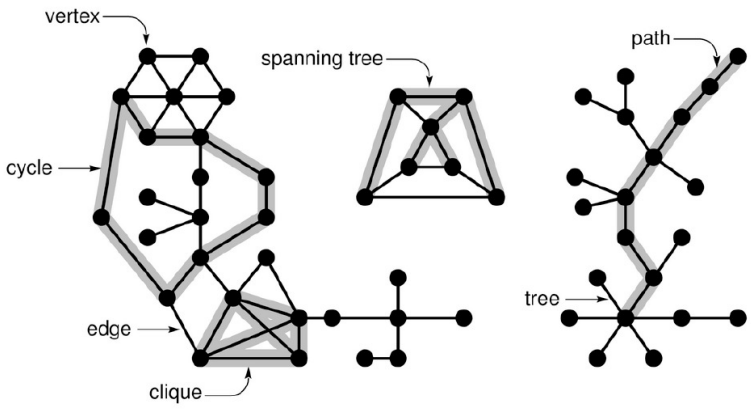
COS 460 – Algorithms (lecture 11) GRAPHS
(Undirected) Graph:
Set of objects with pairwise connections.
Why study graph algorithms?

- Interesting and broadly useful abstraction.
- Challenging branch of computer science and discrete math.
- Hundreds of graph algorithms known.
- Thousands of practical applications.



Graph	Vertices	Edges
communication	telephones, computers	fiber optic cables
circuits	gates, registers, processors	wires
mechanical	joints	rods, beams, springs
hydraulic	reservoirs, pumping stations	pipelines
financial	stocks, currency	transactions
transportation	street intersections, airports	highways, airway routes
scheduling	tasks	precedence constraints
software systems	functions	function calls
internet	web pages	hyperlinks
games	board positions	legal moves
social relationship	people, actors	friendships, movie casts
neural networks	neurons	synapses
protein networks	proteins	protein-protein interactions
chemical compounds	molecules	bonds

Graph Terminology

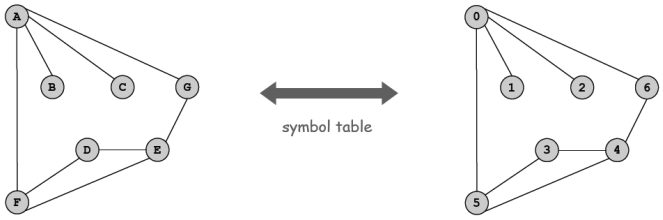


Some Graph Problems
Path. Is there a path between s to t?
Shortest path. What is the shortest path between s and t?
Longest path. What is the longest simple path between s and t?
Cycle. Is there a cycle in the graph?
Euler tour. Is there a cycle that uses each edge exactly once?
Hamilton tour. Is there a cycle that uses each vertex exactly once?
Connectivity. Is there a way to connect all of the vertices?
MST. What is the best way to connect all of the vertices?
Biconnectivity. Is there a vertex whose removal disconnects the graph?
Planarity. Can you draw the graph in the plane with no crossing edges?
Isomorphism. Do two adjacency matrices represent the same graph?

Graph Representation
Vertex representation.

- We use integers between 0 and V–1.
- Real world: convert between names and integers with symbol table.

Other issues. Parallel edges, self-loops.



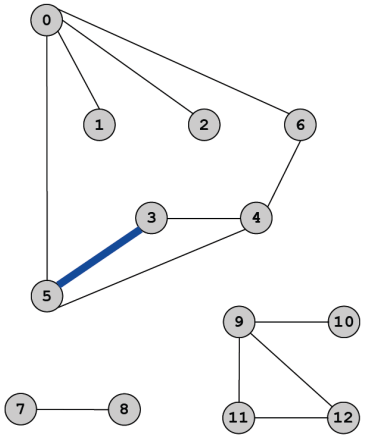
Graph API
`class Graph`
graph data type

`Graph(int V)` create an empty graph with V vertices
`Graph(int V, int E)` create a random graph with V vertices, E edges
`int V()` return number of vertices
`void insert(int v, int w)` add an edge v-w
`int* adj(int v)` return the starting value of an iterator over the neighbors of v
`int* nadj(int v)` return the upper barrier of an iterator over the neighbors of v

Example: iterate through all edges:

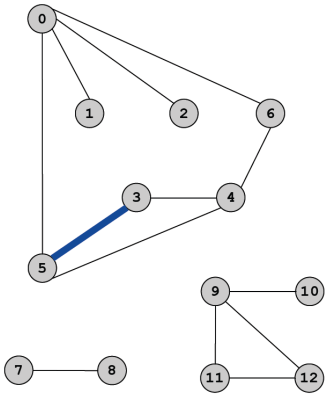
```
Graph* G = new Graph(5, 4);
for (int v = 0; v < G->V(); v++)
{ cout << v << " :";
  for (int* pw = G->adj(v); pw != G->nadj(v); pw++)
   cout << " " << *pw;
  cout << endl;
}
```

Set of edge representation.
Store list of edges.



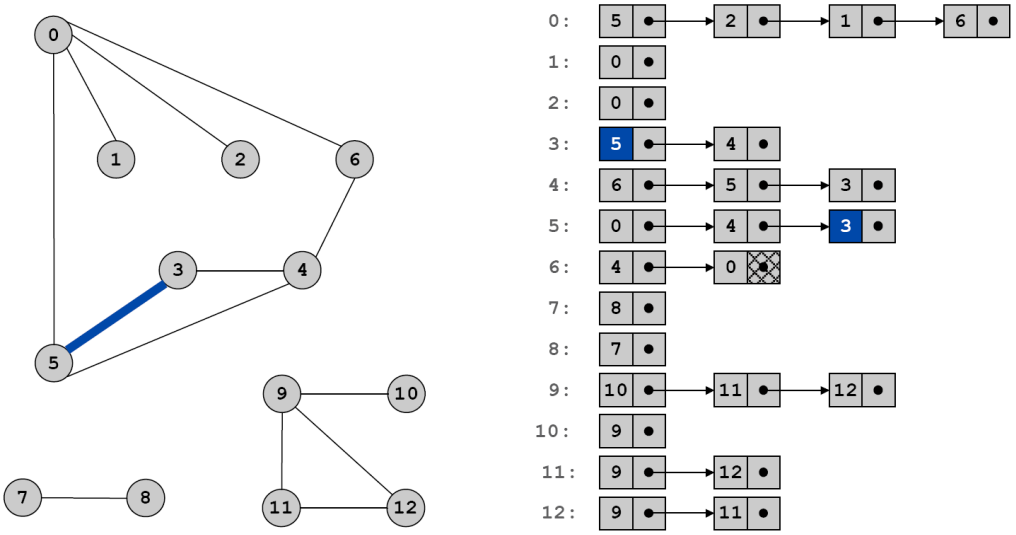

```
0-1
0-6
0-2
11-12
9-12
9-11
9-10
4-3
5-3
7-8
5-4
0-5
6-4
```

Adjacency matrix representation.
Two-dimensional V by V boolean array.
Edge v-w in graph ↔ adj[v][w] = adj[w][v] = true.



	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	1	0	0	0	0	1	1	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	1	1	0	0	0	0	0	0	0
4	0	0	0	1	0	1	1	0	0	0	0	0	0
5	1	1	0	1	1	0	0	0	0	0	0	0	0
6	1	1	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	1	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	1	0	0	0
11	0	0	0	0	0	0	0	0	0	1	0	0	1
12	0	0	0	0	0	0	0	0	0	1	0	1	0

Adjacency List Representation
Vertex indexed array of lists.
Two representations of each undirected edge.



Graph Representations
Graphs are abstract mathematical objects.

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

Representation	Space	Edge between v and w?	Iterate over edges incident to v?
List of edges	E	E	E
Adjacency matrix	V^2	1	V
Adjacency list	E + V	degree(v)	degree(v)

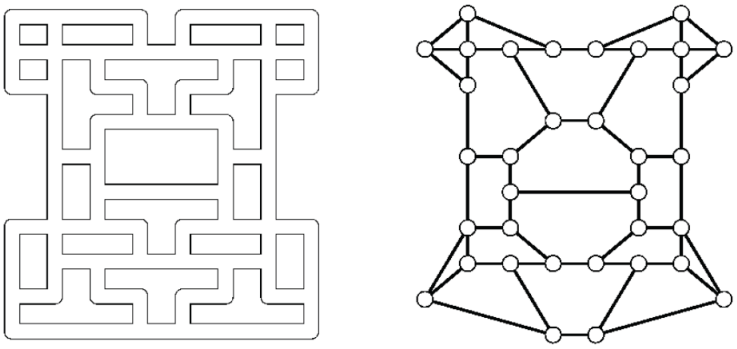
Graphs in practice. [use adjacency list representation]

- Real world graphs are sparse.
- Bottleneck is iterating over edges incident to v.

Maze graphs.

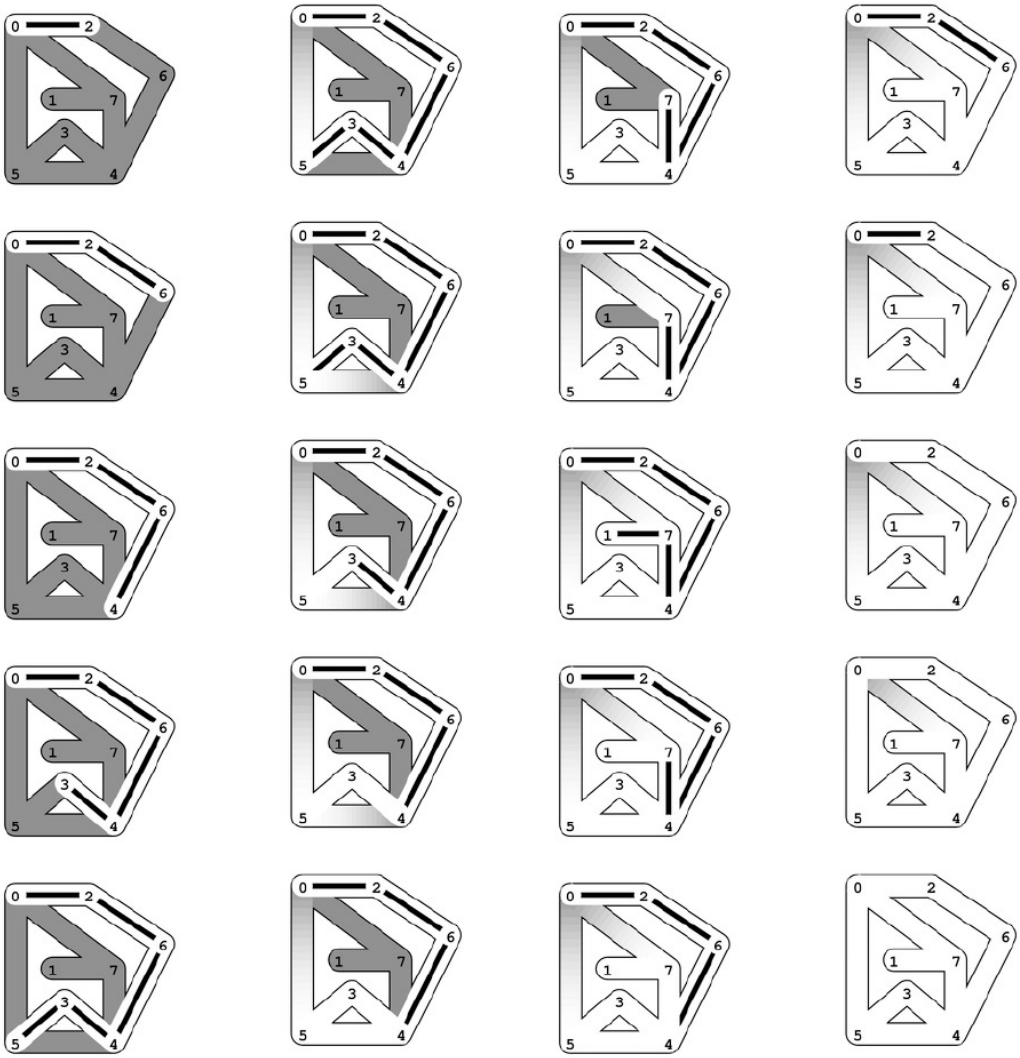
- Vertex = intersections.
- Edge = passage.

Goal. Explore every passage in the maze.



History. Theseus entered labyrinth to kill the monstrous Minotaur; Ariadne held ball of string.

- Unroll a ball of string behind you.
- Mark each visited intersection by turning on a light.
- Mark each visited passage by opening a door.



Depth First Search. Goal. Find all vertices connected to s.

DFS (to visit a vertex v)

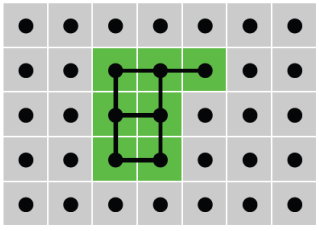
- Mark v as visited.
- Visit all unmarked vertices w adjacent to v.

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

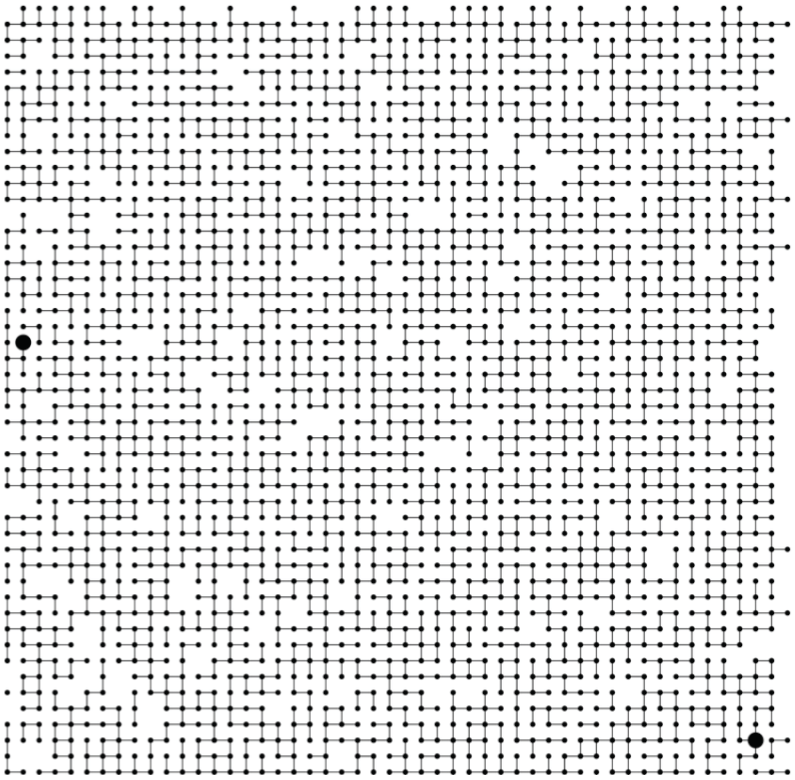
Reachability Application: **Flood Fill**

Given lime green pixel in an image, change color of entire blob of neighboring lime pixels to blue.

- Vertex: pixel.
- Edge: between two adjacent lime pixels.
- Blob: all pixels reachable from chosen lime pixel.



Path. Is there a path from s to t? If so, find one.



Union-Find Abstraction

What are critical operations we need to support?

- Objects.

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

grid points

- Disjoint sets of objects.

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

subsets of connected grid points

- Find: are objects 2 and 9 in the same set?

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

are two grid points connected?

- Union: merge sets containing 3 and 8.

```
0  1  2-3-4-8-9  7
```

add a connection between two grid points

Quick-Find [eager approach]
Data structure.

- Integer array id[] of size N.
- Interpretation: p and q are connected if they have the same id.

i	0	1	2	3	4	5	6	7	8	9
id[i]	0	1	9	9	9	6	6	7	8	9

5 and 6 are connected
2, 3, 4, and 9 are connected

Find. Check if p and q have the same id. (id[3] = 9; id[6] = 6; 3 and 6 not connected)

Union. To merge components containing p and q, change all entries with id[p] to id[q].

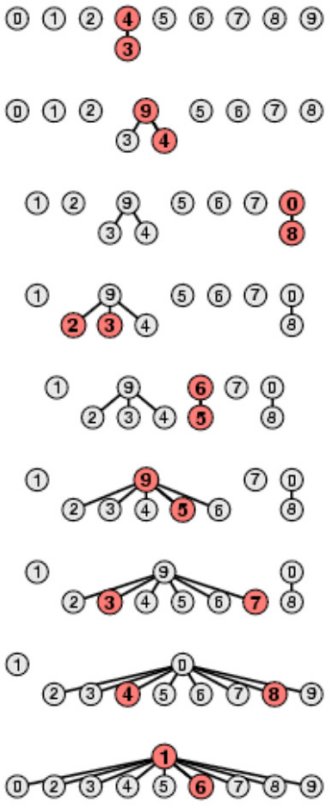
i	0	1	2	3	4	5	6	7	8	9
id[i]	0	1	6	6	6	6	6	7	8	6

many values can change

union of 3 and 6
2, 3, 4, 5, 6, and 9 are connected

Quick-Find: Example

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

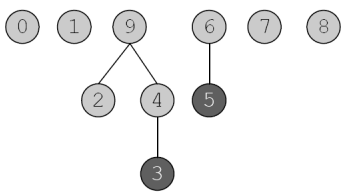


Quick-Union [lazy approach]

Data structure.

- Integer array id[] of size N.
- Interpretation: id[i] is parent of i.
- Root of i is id[id[id[...id[i]...]]].

i	0	1	2	3	4	5	6	7	8	9
id[i]	0	1	9	4	9	6	6	7	8	9



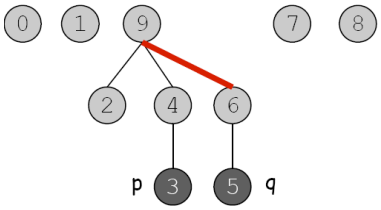
3's root is 9; 5's root is 6
3 and 5 are not connected

Find. Check if p and q have the same root.

Union. Set the id of q's root to the id of p's root.

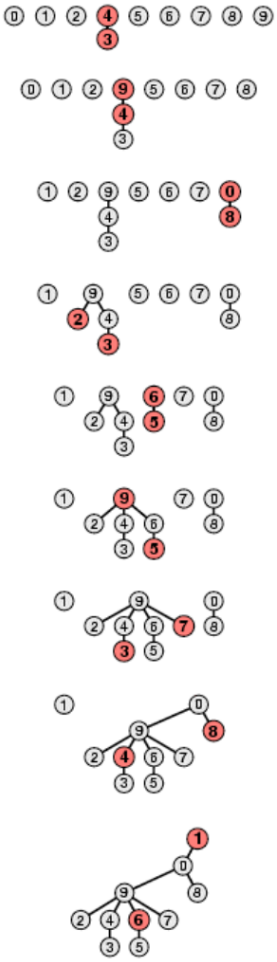
i	0	1	2	3	4	5	6	7	8	9
id[i]	0	1	9	4	9	6	9	7	8	9

only one value changes



Quick-Union: Example

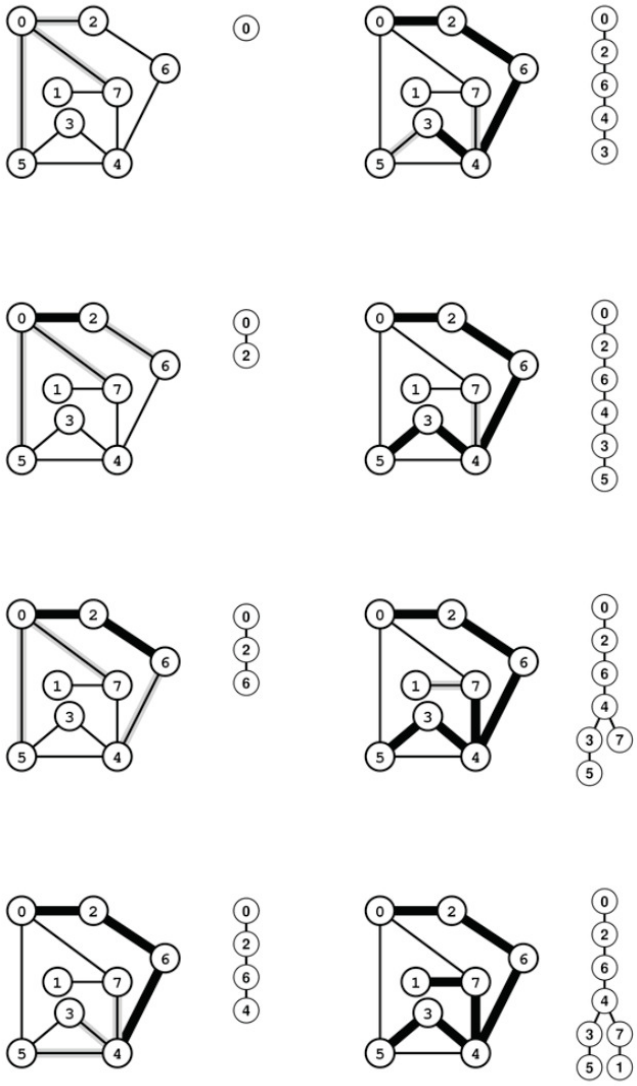
3-4	0	1	2	4	4	5	6	7	8	9
4-9	0	1	2	4	9	5	6	7	8	9
8-0	0	1	2	4	9	5	6	7	0	9
2-3	0	1	9	4	9	5	6	7	0	9
5-6	0	1	9	4	9	6	6	7	0	9
5-9	0	1	9	4	9	6	9	7	0	9
7-3	0	1	9	4	9	6	9	9	0	9
4-8	0	1	9	4	9	6	9	9	0	0
6-1	1	1	9	4	9	6	9	9	0	0



UF advantage. Can intermix query and edge insertion.
DFS advantage. Can recover path itself in same running time.

Keep Track of Path

DFS tree. Upon visiting a vertex v for the first time, remember from where you came pred[v].
Retrace path. To find path between s and v, follow pred[] values back from v.



DFS Summary

Enables direct solution of simple graph problems.

- Find path between s to t.
- Connected components.
- Euler tour.
- Cycle detection.
- Bipartiteness checking.

Basis for solving more difficult graph problems.

- Biconnected components.
- Planarity testing.

Breadth First Search

Depth-first search. Put unvisited vertices on a stack.
Breadth-first search. Put unvisited vertices on a queue.
Shortest path. Find path from s to t that uses fewest number of edges.

BFS (from source vertex s)

Put s onto a FIFO queue.
Repeat until the queue is empty:

- remove the least recently added vertex v
- add each of v's unvisited neighbors to the queue,

and mark them as visited.

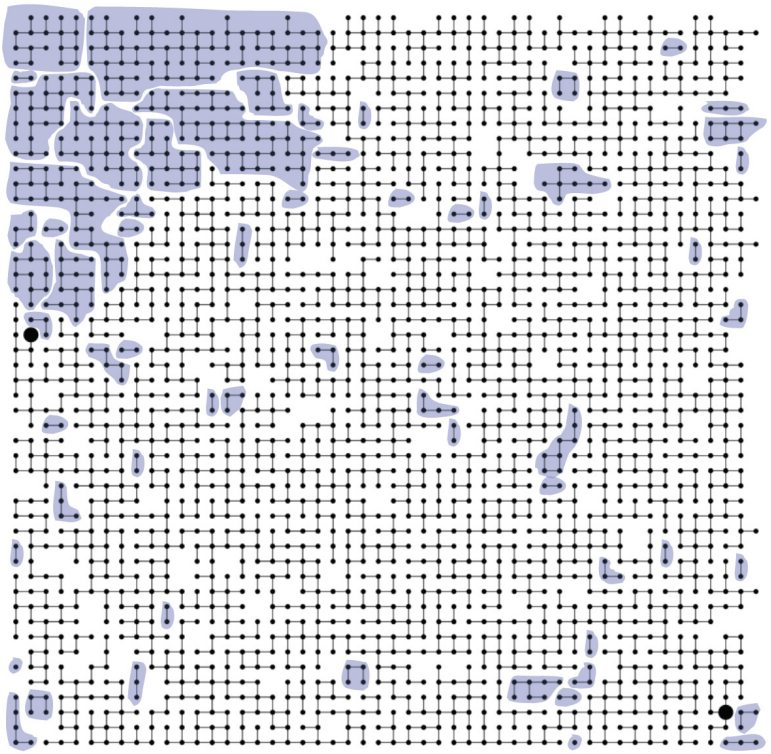
Property. BFS examines vertices in increasing distance from s.

Connectivity Queries
Def. Vertices v and w are connected if there is a path between them.
Property. Symmetric and transitive.
Goal. Preprocess graph to answer queries: is v connected to w?
Connected component. Maximal set of mutually connected vertices.

Brute force. Run DFS from each vertex v: quadratic time and space.

Connected components
Initialize all vertices v as unmarked.
For each unmarked vertex v, run DFS and identify all vertices discovered as part of the same connected component.

Preprocess Time	Query Time	Space
E + V	1	V

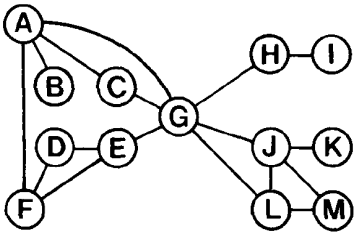


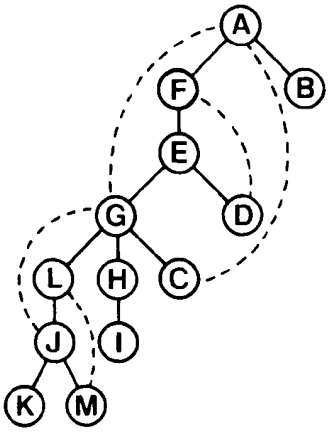
63 connected components

A **cut vertex** (**articulation point**) is a vertex of a graph such that removal of the vertex causes an increase in the number of connected components. If the graph was connected before the removal of the vertex, it will be disconnected afterwards.
A **bridge** is an edge analogous to a cut vertex; that is, the removal of a bridge increases the number of connected components of the graph.
In a tree, every vertex with degree greater than 1 is a cut vertex.
A **biconnected** graph is a connected graph with no articulation vertices. In other words, a **biconnected** graph is **nonseparable**, meaning if any edge were to be removed, the graph will remain connected.
This property is especially useful in maintaining a graph with a two-fold redundancy, to prevent disconnection upon the removal of a single edge (or connection).

The use of **biconnected** graphs is very important in the field of networking, because of this property of redundancy.

Example: The articulation points of this graph are A (because it connects B to the rest of the graph), H (because it connects I to the rest of the graph), J (because it connects K to the rest of the graph), and G (because the graph would fall into three pieces if G were deleted). There are six biconnected components: ACGDEF, GJLM, and the individual nodes B, H, I, and K.





Determining the articulation points turns out to be a simple extension of depth-first search. To see this, consider the depth-first search tree for this graph:

Deleting node E will not disconnect the graph because G and D both have dotted links that point above E, giving alternate paths from them to F (E's father in the tree). On the other hand, deleting G will disconnect the graph because there are no such alternate paths from L or H to E (L's father).

A vertex x is not an articulation point if every son y has some node lower in the tree connected (via a dotted link) to a node higher in the tree than x, thus providing an alternate connection from x to y. This test doesn't quite work for the root of the depth-first search tree, since there are no nodes "higher in the tree." The root is an articulation point if it has two or more sons, since the only path connecting sons of the root goes through the root. These tests are easily incorporated into depth-first search by changing the node-visit procedure into a function which returns the highest point in the tree (lowest val value) seen during the search, as follows:

```
Graph* G; int* p; int c;
int mdfs(int v)
{ int min=c;
  p[v]=c; c++;
  for (int* pw = G->adj(v); pw != G->nadj(v); pw++)
   if(p[*pw]==-1)
    {int m=mdfs(*pw); if(m<min)min=m;
     if(m>=p[v]) cout << v << endl;}
   else if(p[*pw]<min) min=p[*pw];
  return min;
}
int main()
{ G = new Graph();
  int size=G->V(); p = new int[size]; memset(p,-1,size*sizeof(int));
  p[0]=0;c=0;mdfs(0);
}
```

This function recursively determines the highest point in the tree reachable (via a dotted link) from any descendant of vertex *v* and uses this information to determine if *v* is an articulation point (and prints it). Normally this calculation simply involves testing whether the minimum value reachable from a son is higher up in the tree, but we need an extra test to determine whether *v* is the root of a depth-first search tree (or, equivalently, whether this is the first call to visit for the connected component containing *v*. This test is properly performed outside the recursive function, so it does not appear in the code above. Since it is a depth-first search function, the running time is proportional to $V + E$.