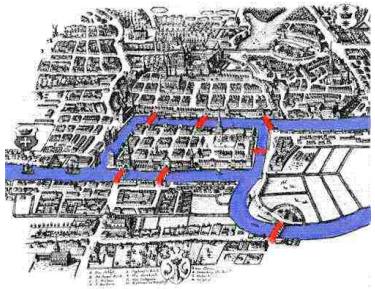
Undirected Graphs

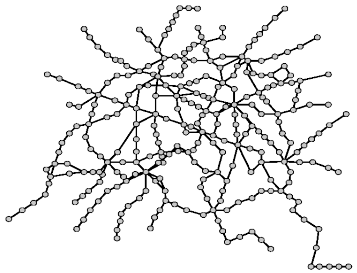


Reference: Chapter 17-18, Algorithms in Java, 3rd Edition, Robert Sedgewick.

Graph. Set of objects with pairwise connections.

Why study graph algorithms?

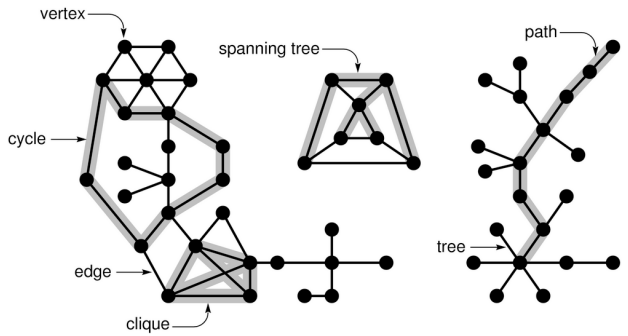
- Interesting and broadly useful abstraction.
- Challenging branch of computer science and discrete math.
- Hundreds of graph algorithms known.
- Thousands of practical applications.



Graph 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
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 two vertices?
Longest path. What is the longest path between two vertices?

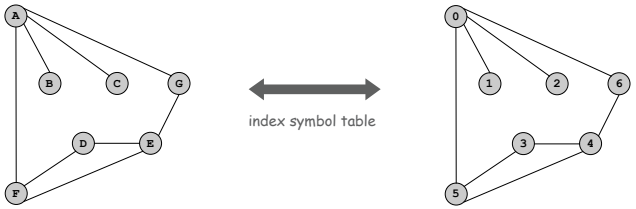
Cycle. Is there a cycle in the graph?
Euler tour. Is there a cyclic path that uses each edge exactly once?
Hamilton tour. Is there a cycle that uses each vertex exactly once?

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



Other issues. Parallel edges, self-loops.

Graph Interface

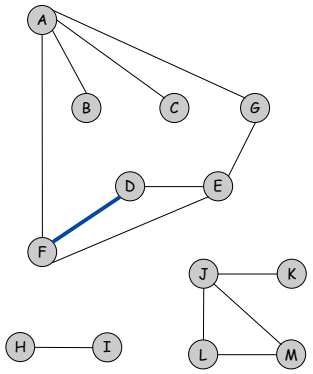
Return Type	Method	Action
	`Graph(int V)`	create empty graph
	`Graph(int V, int E)`	create random graph
`void`	`insert(int v, int w)`	add edge `v-w`
`Iterable<Integer>`	`adj(int v)`	return iterator over neighbors of `v`
`int`	`V()`	return number of vertices
`String`	`toString()`	return string representation

```
Graph G = new Graph(V, E);
System.out.println(G);
for (int v = 0; v < G.V(); v++)
   for (int w : G.adj(v))
      // edge v-w
```

iterate through all edges (in each direction)

Set of Edge Representation

Set of edge representation. Store list of edges.

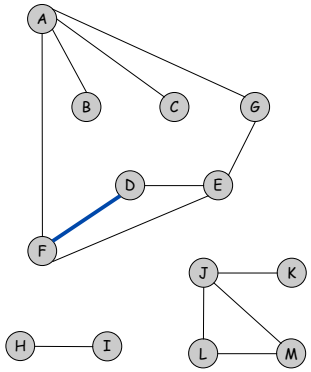



```
A-B
A-G
A-C
L-M
J-M
J-L
J-K
E-D
F-D
H-I
F-E
A-F
G-E
```

Adjacency Matrix Representation

Adjacency matrix representation.

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



	A	B	C	D	E	F	G	H	I	J	K	L	M
0 A	0	1	1	0	0	1	1	0	0	0	0	0	0
1 B	1	0	0	0	0	1	1	0	0	0	0	0	0
2 C	1	0	0	0	0	0	0	0	0	0	0	0	0
3 D	0	0	0	0	1	1	0	0	0	0	0	0	0
4 E	0	0	0	1	0	1	1	0	0	0	0	0	0
5 F	1	1	0	1	1	0	0	0	0	0	0	0	0
6 G	1	1	0	0	1	0	0	0	0	0	0	0	0
7 H	0	0	0	0	0	0	0	0	1	0	0	0	0
8 I	0	0	0	0	0	0	0	1	0	0	0	0	0
9 J	0	0	0	0	0	0	0	0	0	0	1	1	1
10 K	0	0	0	0	0	0	0	0	0	1	0	0	0
11 L	0	0	0	0	0	0	0	0	0	1	0	0	1
12 M	0	0	0	0	0	0	0	0	0	1	0	1	0

Adjacency Matrix Representation: Java Implementation

```
public class Graph {
   private int V;            // number of vertices
   private boolean[][] adj;  // adjacency matrix

   // empty graph with V vertices
   public Graph(int V) {
      this.V = V;
      this.adj = new boolean[V][V];
   }

   // insert edge v-w if it doesn't already exist
   public void insert(int v, int w) {
      adj[v][w] = true;
      adj[w][v] = true;
   }

   // return iterator for neighbors of v
   public Iterable<Integer> adj(int v) {
      return new AdjIterator(v);
   }
}
```


Adjacency Matrix Iterator

```
private class AdjIterator implements Iterator<Integer>,
                                     Iterable<Integer> {
   int v, w = 0;
   AdjIterator(int v) { this.v = v; }

   public boolean hasNext() {
      while (w < V) {
         if (adj[v][w]) return true;
         w++;
      }
      return false;
   }

   public int next() {
      if (hasNext()) return w++;
      else           return -1;
   }
}
```

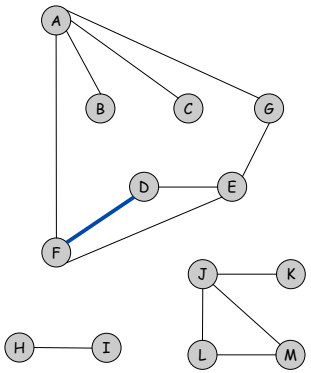
does v have another neighbor w?

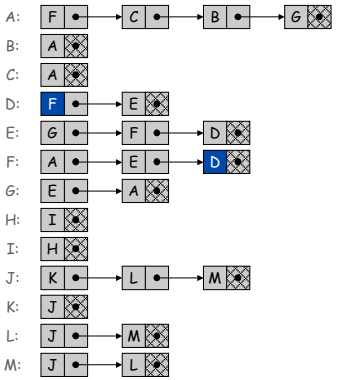
return next neighbor w of v

Adjacency List Representation

Vertex indexed array of lists.

- Space proportional to number of edges.
- Two representations of each undirected edge.





Adjacency List Representation: Java Implementation

```
public class Graph {
   private int V;                    // # vertices
   private Sequence<Integer>[] adj; // adjacency lists

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

   // insert v-w, parallel edges allowed
   public void insert(int v, int w) {
      adj[v].add(w);
      adj[w].add(v);
   }

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


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?	Enumerate edges incident to v?
List of edges	$\Theta(E)$	$O(E)$	$\Theta(E)$
Adjacency matrix	$\Theta(V^2)$	$\Theta(1)$	$\Theta(V)$
Adjacency list	$\Theta(E + V)$	$O(\text{degree}(v))$	$\Theta(\text{degree}(v))$

Graphs in practice. [use adjacency list representation]

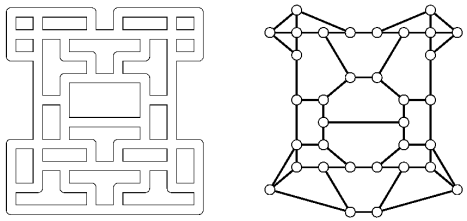
- Real world graphs are sparse.
- Bottleneck is iterating over edges incident to v.

Maze Exploration

Maze Exploration

Maze graphs.

- Vertex = intersections.
- Edge = passage.



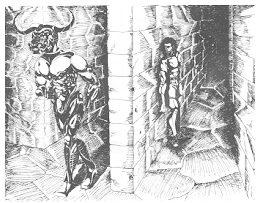
Goal. Explore every passage in the maze.

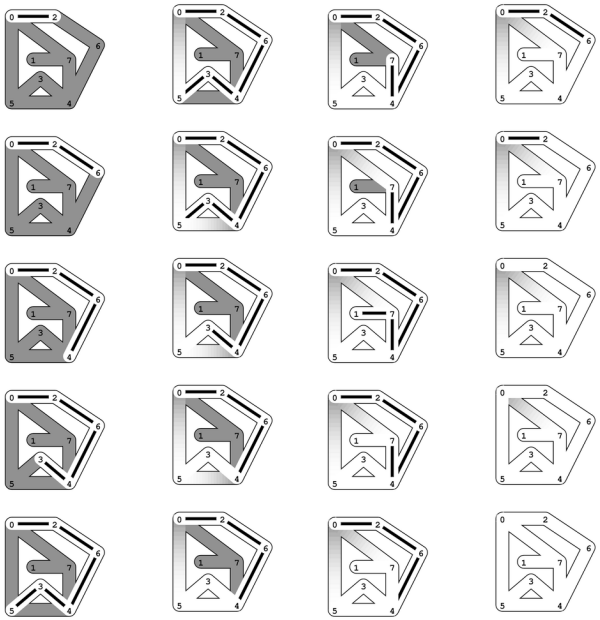
Trémaux Maze Exploration

Trémaux maze exploration.

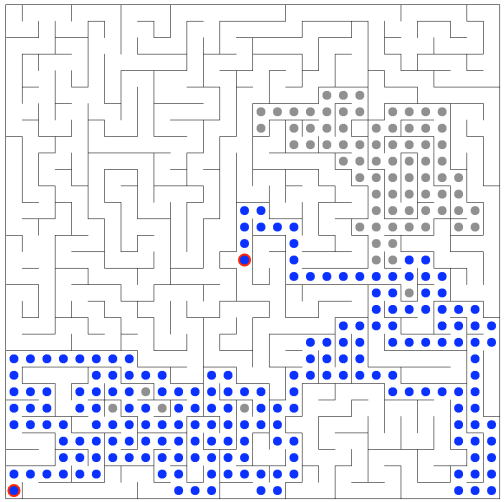
- Unroll a ball of string behind us.
- Mark each visited intersection by turning on a light.
- Mark each visited passage by opening a door.

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





Maze Exploration



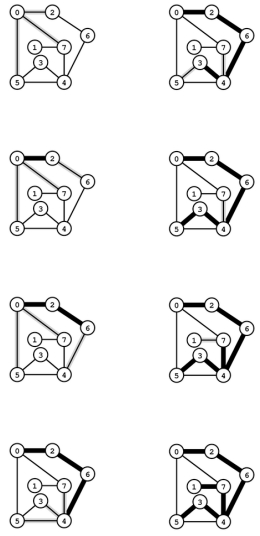
Depth First Search

Goal. Find all vertices connected to s.

Depth first search. To visit a vertex v:

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

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



Graph Processing Client

Typical client program.

- Create a `Graph`.
- Pass the `Graph` to a graph processing routine, e.g., `DFSearcher`.
- Query the graph processing routine for information.
- Design pattern: separate graph from graph algorithms.

```
public static void main(String args[]) {
   int V = Integer.parseInt(args[0]);
   int E = Integer.parseInt(args[1]);
   int s = 0;
   Graph G = new Graph(V, E);
   DFSearcher dfs = new DFSearcher(G, s);
   for (int v = 0; v < G.V(); v++)
      if (dfs.isReachable(v))
         System.out.println(v);
}
```

find and print vertices reachable from s

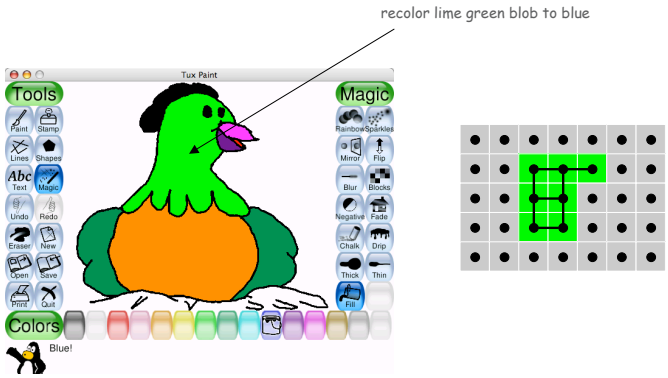
Depth First Search

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

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

   // depth first search from v
   private void dfs(Graph 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]; }

}
```


Reachability Application: Flood Fill

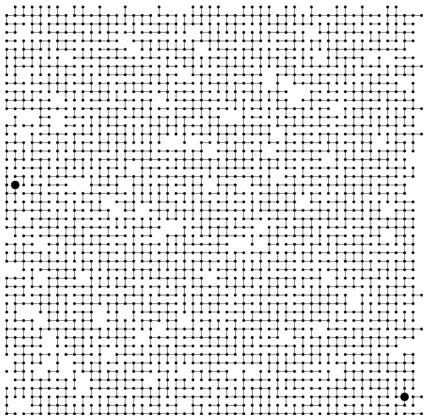
Flood fill. Given lime green pixel in an image, change color of entire blob of neighboring lime pixels to blue.

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



Paths

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



Paths

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

Method	Preprocess Time	Query Time	Space
Union Find	$O(E \log^* V)$ †	$O(\log^* V)$ †	$\Theta(V)$
DFS	$\Theta(E + V)$	$\Theta(1)$	$\Theta(V + E)$

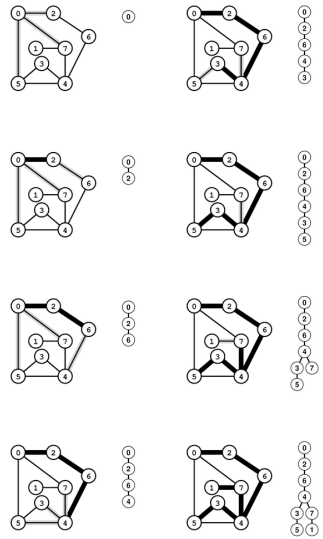
† amortized

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`.



Find Path

```
public class DFSearcher {
   // ininitalize pred[v] to -1 for all v

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

   // return path from s to v
   public Iterable<Integer> path(int v) {
      LinkedList<Integer> list = new LinkedList<Integer>();
      while (v != -1 && marked[v]) {
         list.addFirst(v);
         v = prev[v];
      }
      return list;
   }
}
```


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 difficulty 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.

Breadth first search.

- Initialize `dist[v] = ∞, dist[s] = 0`.
- When considering edge `v-w`:
  - if `w` is marked, then ignore
  - otherwise set `dist[w] = dist[v] + 1` and add `w` to the queue

Property. BFS examines vertices in increasing distance from s.

Breadth First Search

```
public class BFSearcher {
   private static int INFINITY = Integer.MAX_VALUE;

   private int[] dist;

   public BFSearcher(Graph G, int s) {
      dist = new int[G.V()];
      for (int v = 0; v < G.V(); v++) dist[v] = INFINITY;
      dist[s] = 0;
      bfs(G, s);
   }

   public int distance(int v)        { return dist[v]; }
   private void bfs(Graph G, int s) { // NEXT SLIDE   }
}
```

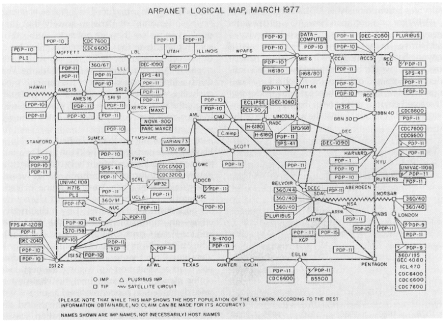

Breadth First Search

```
// breadth-first search from s
private void bfs(Graph G, int s) {
   Queue<Integer> q = new Queue<Integer>();
   q.enqueue(s);
   while (!q.isEmpty()) {
      int v = q.dequeue();
      for (int w : G.adj(v)) {
         if (dist[w] == INFINITY) {
            q.enqueue(w);
            dist[w] = dist[v] + 1;
         }
      }
   }
}
```


BFS Application

BFS applications.

- Facebook.
- Kevin Bacon numbers.
- Fewest number of hops in a communication network.



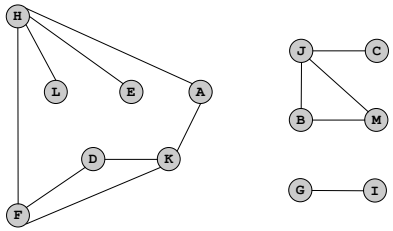
ARPANET

Connected Components

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



Vertex	Component
A	0
B	1
C	1
D	0
E	0
F	0
G	2
H	0
I	2
J	1
K	0
L	0
M	1

Connected component. Maximal set of mutually connected vertices.

Connected Components

Depth-first search.
- To traverse a graph G:
  - initialize all vertices as unmarked
  - visit each unmarked vertex v
- To visit a vertex v:
  - mark v as visited
  - recursively visit all unmarked vertices w adjacent to v

Result.
- Preprocessing: O(V + E) time, O(V) extra space.
- Connectivity query: O(1) time.

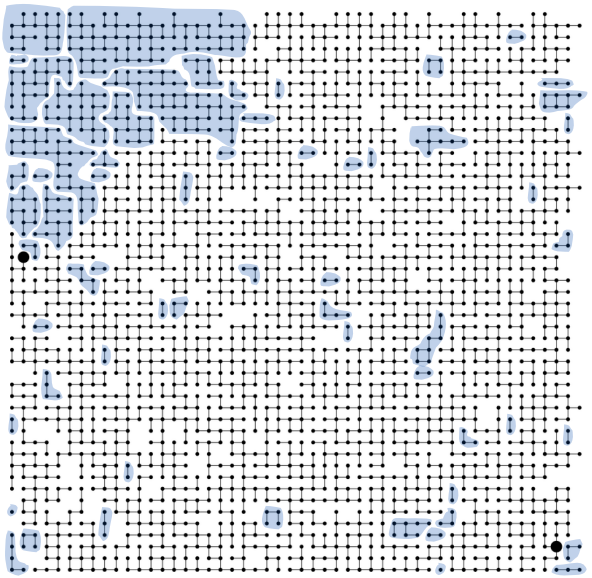
Depth First Search: Connected Components

```
public class CCFinder {
   private int components;
   private int[] cc;

   public CCFinder(Graph G) {
      this.cc = new int[G.V()];
      for (int v = 0; v < G.V(); v++) cc[v] = -1;
      for (int v = 0; v < G.V(); v++)
         if (cc[v] == -1) { dfs(G, v); components++; }
   }

   // depth first search from v
   private void dfs(Graph G, int v) {
      cc[v] = components;
      for (int w : G.adj(v))
         if (cc[w] == -1) dfs(G, w);
   }

   public int connected(int v, int w) { return cc[v] == cc[w]; }
}
```

unmarked

are v and w in same connected component?

Connected Components



63 components

Connected Components Application: Image Processing

Goal. Read in a 2D color image and find regions of connected pixels that have the same color.

original

labeled

Connected Components Application: Image Processing

Goal. Read in a 2D color image and find regions of connected pixels that have the same color.

Efficient algorithm.

- Connect each pixel to neighboring pixel if same color.
- Find connected components in resulting graph.

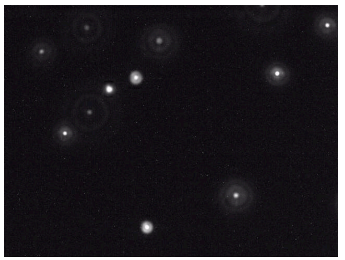
0	1	1	1	1	1	6	6	8	9	9	11
0	0	0	1	6	6	6	8	8	11	9	11
3	0	0	1	6	6	4	8	11	11	11	11
3	0	0	1	1	6	2	11	11	11	11	11
10	10	10	10	1	1	2	11	11	11	11	11
7	7	2	2	2	2	2	11	11	11	11	11
7	7	5	5	5	2	2	11	11	11	11	11

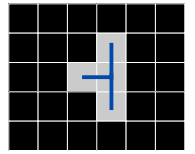
Connected Components Application: Particle Detection

Particle detection. Given grayscale image of particles, identify "blobs."

- Vertex: pixel.
- Edge: between two adjacent pixels with grayscale value ≥ 70.
- Blob: connected component of 20-30 pixels.

black = 0
white = 255





Particle tracking. Track moving particles over time.

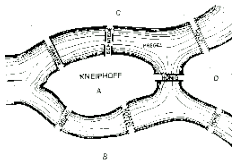
Euler Tour

Bridges of Königsberg

earliest application of graph theory or topology

The Seven Bridges of Königsberg. [Leonhard Euler 1736]

> "..... in Königsberg in Prussia, there is an island A, called the Kneiphof; the river which surrounds it is divided into two branches ... and these branches are crossed by seven bridges. Concerning these bridges, it was asked whether anyone could arrange a route in such a way that he could cross each bridge once and only once....."

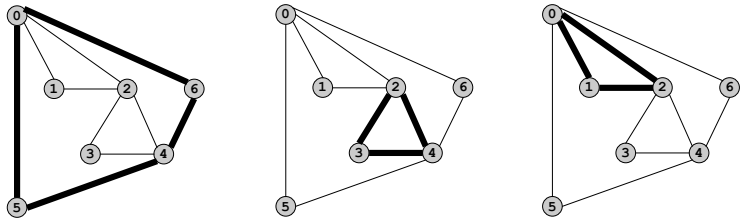


Euler tour. Is there a cyclic path that uses each edge exactly once?
Answer. Yes iff connected and all vertices have even degree.

Euler Tour

How to find an Euler tour. [assuming graph is Eulerian]

- Start at vertex v and follow unused edges until you return to v. (always possible since all vertices have even degree)
- Find additional cyclic paths using remaining edges and splice back into original cyclic path.

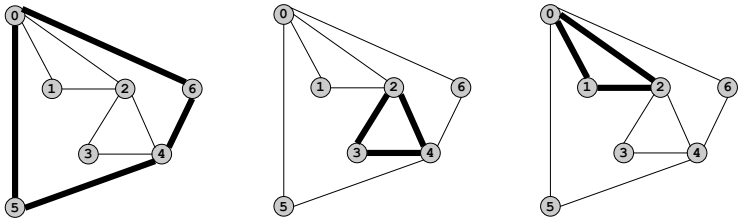


0 – 6 – 4 – 2 – 3 – 4 – 5 – 0 – 2 – 1 – 0

Euler Tour: Implementation

Q. How to efficiently keep track of unused edges?
A. Quick + dirty: delete edge from graph once you use it.

Q. How to efficiently find and splice additional cyclic paths?
A. Push each visited vertex onto a stack.



0 – 6 – 4 – 2 – 3 – 4 – 5 – 0 – 2 – 1 – 0

Two Related Problems

Euler tour. Is there a cyclic path that uses each edge exactly once?
Linear time solution. DFS.

Hamilton tour. Is there a cycle that uses each vertex exactly once?
Polynomial time solution??? NP-complete.