Graph

- Graph
  - Introduction
  - Adjacency list
  - DFS, BFS
  - **Challenges**
- **Digraph – Directed Graphs**
  - digraph – DFS, BFS
  - Applications – crawl web, topological sort
- Minimum Spanning Tree(MST)

Major references:
1. Fundamentals of Data Structures by Horowitz, Sahni, Anderson-Freed,
2. Algorithms 4th edition - Part 1 & Part 2 by Robert Sedgewick and Kevin Wayne
3. Wikipedia and many resources available from internet

Prof. Youngsup Kim, idebtor@gmail.com, Data Structures, CSEE Dept., Handong Global University

Graph-processing challenge 1

Problem: Is a graph bipartite (or bigraph)?

How difficult?

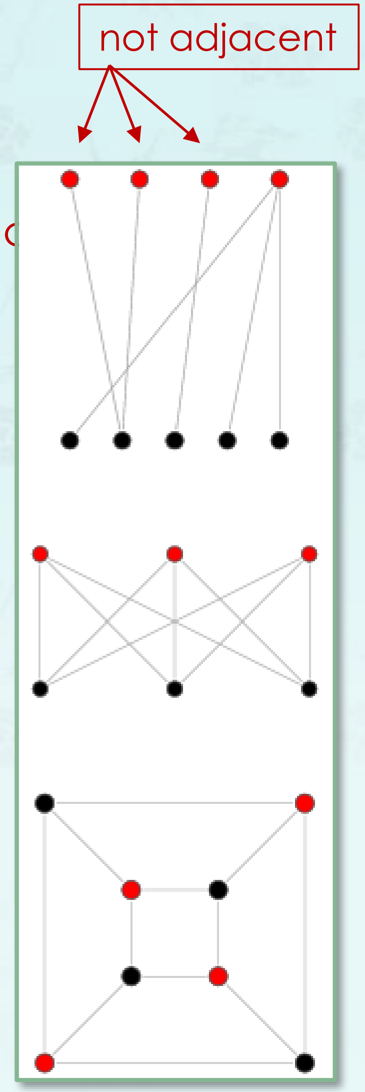
- Any programmer could do it.
- Typical diligent algorithms student could do it.
- Hire an expert.
- Intractable.
- No one knows.
- Impossible.

Graph-processing challenge 1

Problem: Is a graph bipartite (or bigraph)?

a set of graph vertices decomposed into two disjoint sets such that no two graph vertices within the same set are adjac

A bigraph can be split into two groups of vertices such that no two vertices in the same group share an edge.



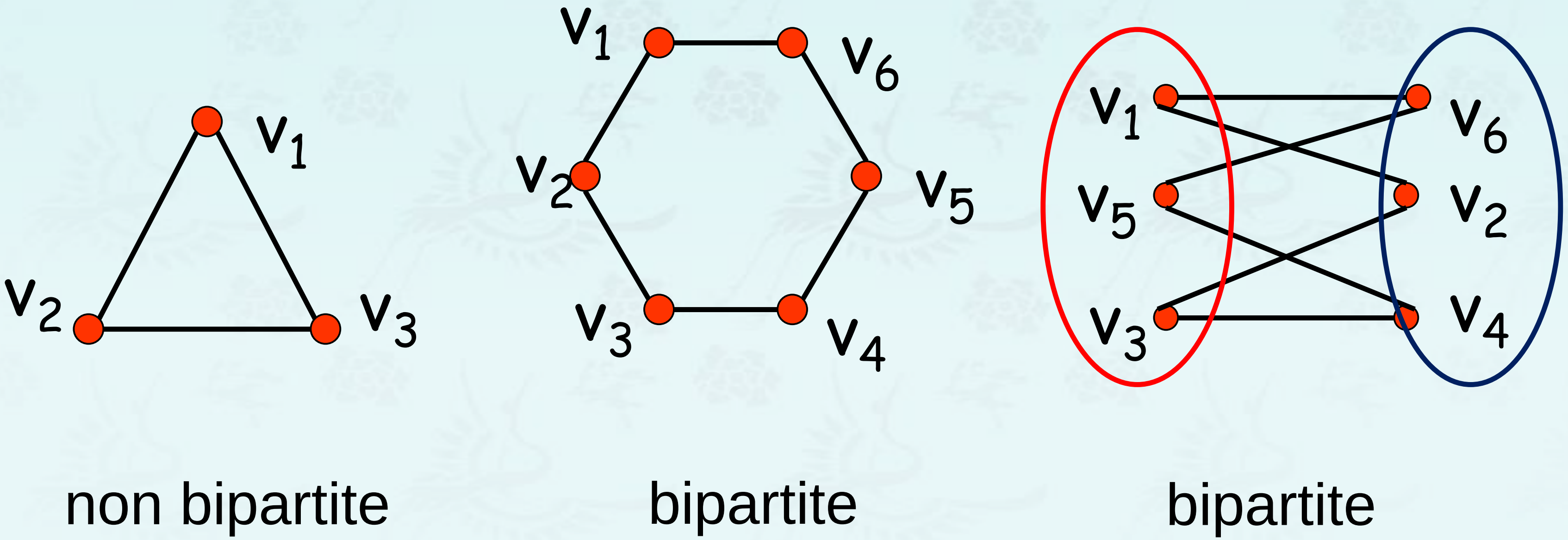
How difficult?

- Any programmer could do it.
- Typical diligent algorithms student could do it.
- Hire an expert.
- Intractable.
- No one knows.
- Impossible.

Graph-processing challenge 1

Problem: Is a graph bipartite (or bigraph)?

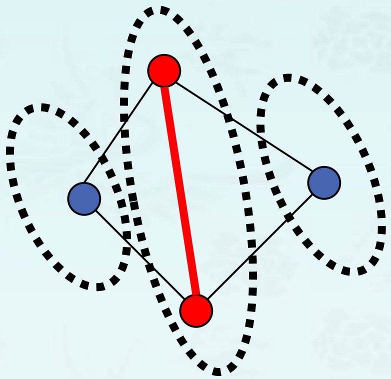
a set of graph vertices decomposed into two disjoint sets
such that no two graph vertices within the same set are adjacent.



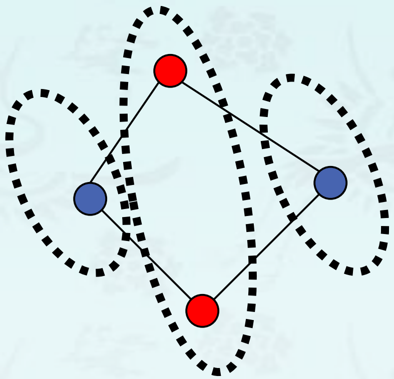
Graph-processing challenge 1

Problem: Is a graph bipartite (or bigraph)?

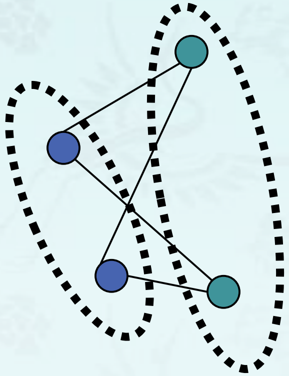
a set of graph vertices decomposed into two disjoint sets
such that no two graph vertices within the same set are adjacent.



non bipartite



bipartite



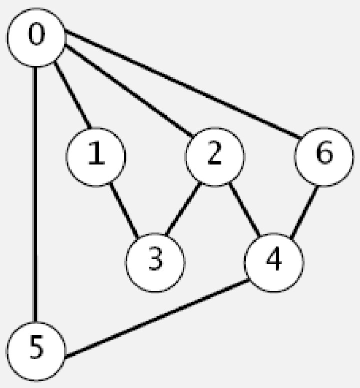
bipartite

Graph-processing challenge 1

Problem: Is a graph bipartite (or bigraph)?

a set of graph vertices decomposed into two disjoint sets
such that no two graph vertices within the same set are adja

a bigraph ?



How difficult?

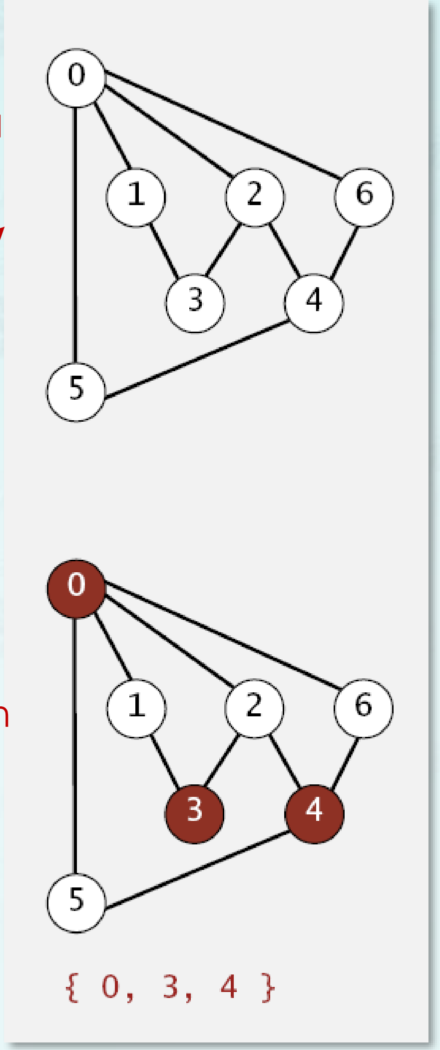
- Any programmer could do it.
- Typical diligent algorithms student could do it.
- Hire an expert.
- Intractable.
- No one knows.
- Impossible.

Graph-processing challenge 1

Problem: Is a graph bipartite (or bigraph)?

a set of graph vertices decomposed into two disjoint sets
such that no two graph vertices within the same set are adja

a bigraph ?



How difficult?

- Any programmer could do it.
- Typical diligent algorithms student could do it.
- Hire an expert.
- Intractable.
- No one knows.
- Impossible.

simple DFS or BFS-based solution

Graph-processing challenge 2

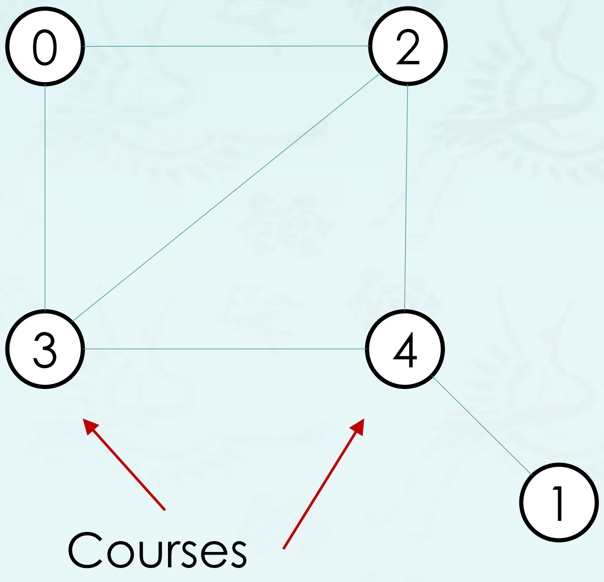
Problem: Graph Coloring

- Given a graph G and K colors, assign a color to each node so adjacent nodes get different colors.
- The minimum value of color K which such a coloring exists is the Chromatic Number of G, $\chi(G)$

Graph-processing challenge 2

Problem: Graph Coloring

- Given a graph G and K colors, assign a color to each node so adjacent nodes get different colors.
- The minum value of color K which such a coloring exists is the **Chromatic Number** of G, $\chi(G)$

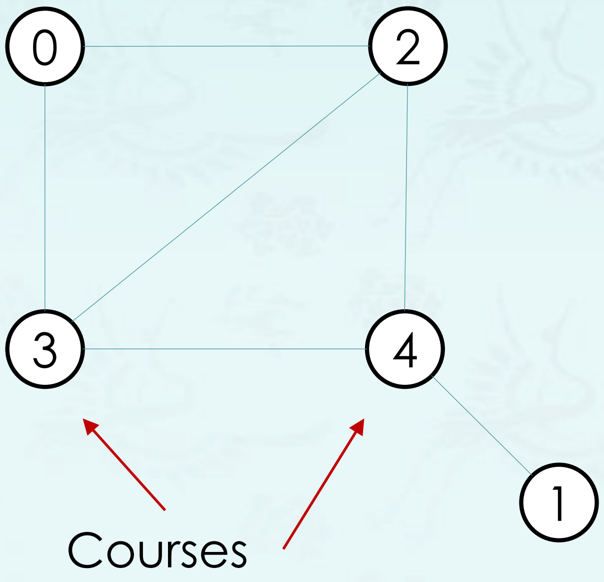


Source: https://www.youtube.com/watch?v=h9wxtqoa1jY

Graph-processing challenge 2

Problem: Graph Coloring

- Given a graph G and K colors, assign a color to each node so adjacent nodes get different colors.
- The minum value of color K which such a coloring exists is the **Chromatic Number** of G, $\chi(G)$



What is the Chromatic Number of the following G?

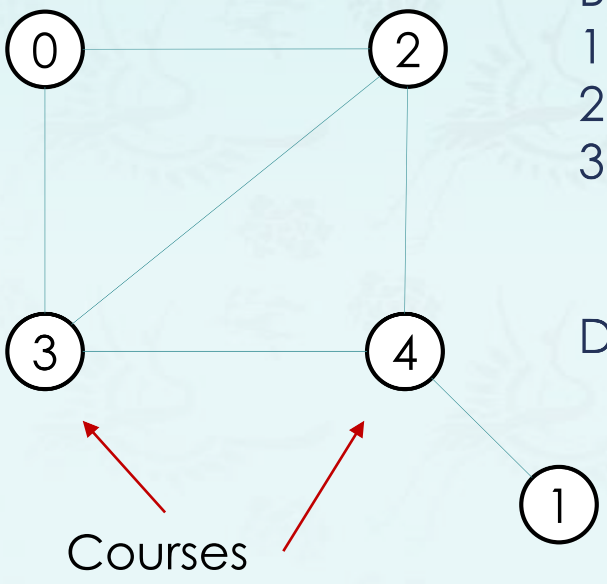
Final Exam time slots:

A: 1-3 pm
B: 4-6 pm
C: 7-9 pm
D: 10-12 pm
E: 1 – 3 pm

Source: https://www.youtube.com/watch?v=h9wxtqoa1jY

Problem: Graph Coloring

- Given a graph G and K colors, assign a color to each node so adjacent nodes get different colors.
- The minum value of color K which such a coloring exists is the **Chromatic Number** of G, $\chi(G)$



Basic Coloring Algorithm for G(V, E)

1. Order the nodes v1, v2, v3, …
2. Order the colors c1, c2, …
3. For i = 1, 2, …, n
   Assign the lowest legal color

Different ordering -> Different results

Source: https://www.youtube.com/watch?v=h9wxtqoa1jY

By Prof. Tom Leighton

Graph Coloring Case Study

Akamai runs a network of thousands of servers and the servers are used to distribute content on Internet. They install a new software or update existing software's pretty much every week. The update cannot be deployed on every server at the same time, because the server may have to be taken down for the install. Also, the update should not be done one at a time, because it will take a lot of time. There are sets of servers that cannot be taken down together, because they have certain critical functions.

This is a typical **scheduling application of graph coloring problem**. It turned out that 8 colors were good enough to color the graph of **75000** nodes.

So they could install updates in 8 passes.

Source: https://www.youtube.com/watch?v=h9wxtqoa1jY By Prof. Tom Leighton

Graph Coloring Case Study

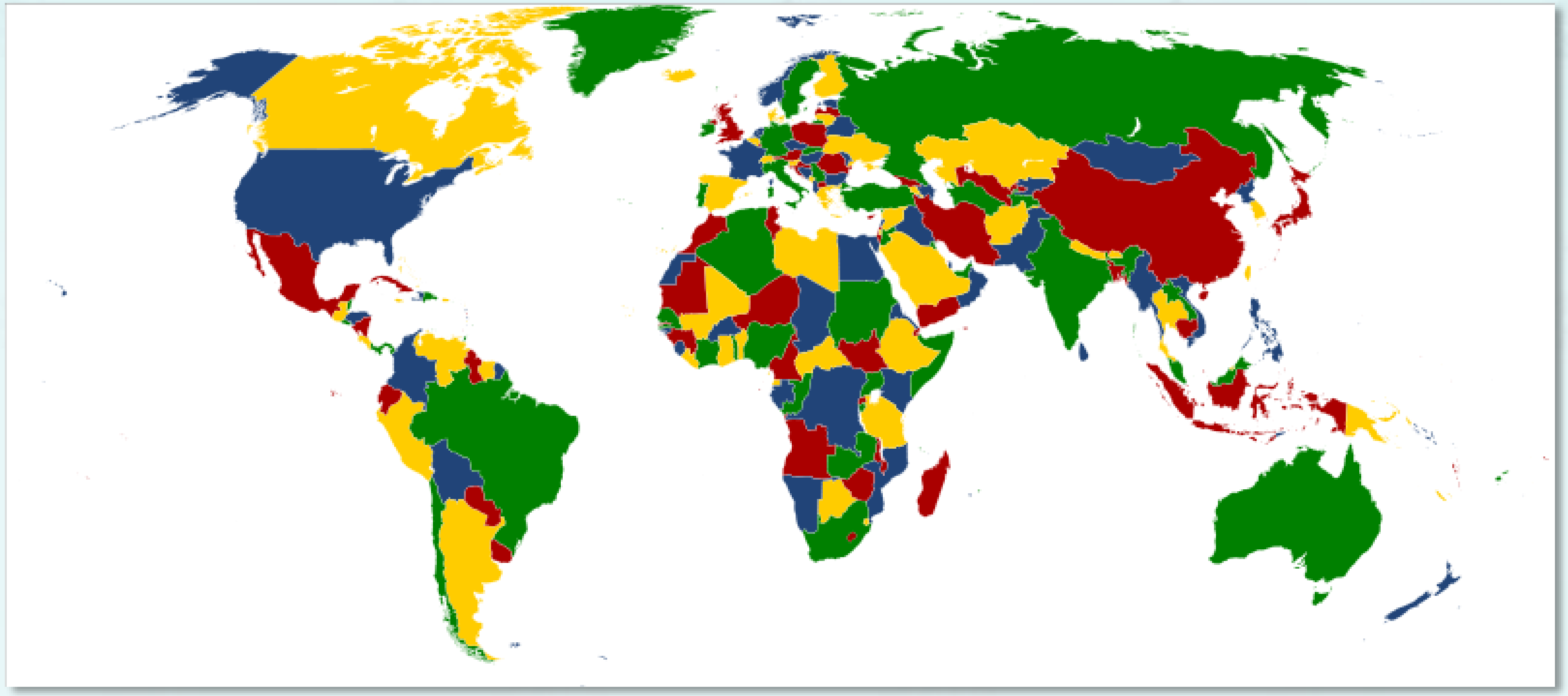
Given any separation of a plane into contiguous regions, producing a figure called map, no more than _______ colors are required to color the regions of the map so that no two adjacent regions have the same color.

Source: https://www.youtube.com/watch?v=h9wxtqoa1jY

By Prof. Tom Leighton

Graph Coloring Case Study

Given any separation of a plane into contiguous regions, producing a figure called map, no more than _______ colors are required to color the regions of the map so that no two adjacent regions have the same color.



Graph-processing challenge 2

Problem: Graph Coloring

- Given a graph G and K colors, assign a color to each node so adjacent nodes get different colors.
- The minimum value of color K which such a coloring exists is the Chromatic Number of G, $\chi(G)$

How difficult?

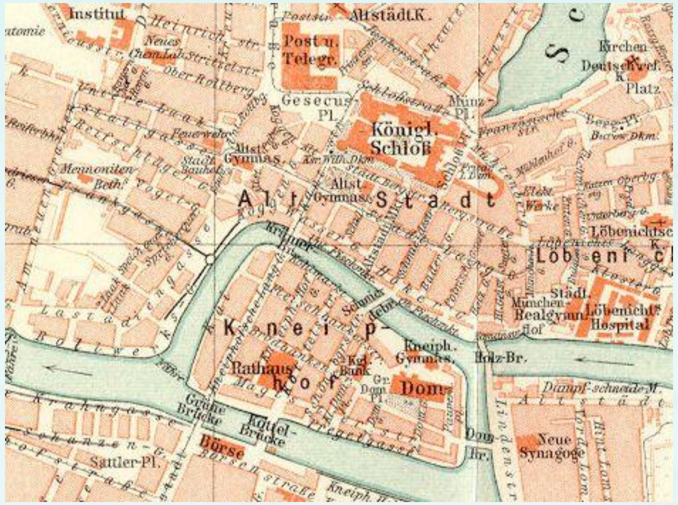
- Any programmer could do it.
- Typical diligent algorithms student could do it.
- Hire an expert.
- Intractable.
- No one knows.
- Impossible.

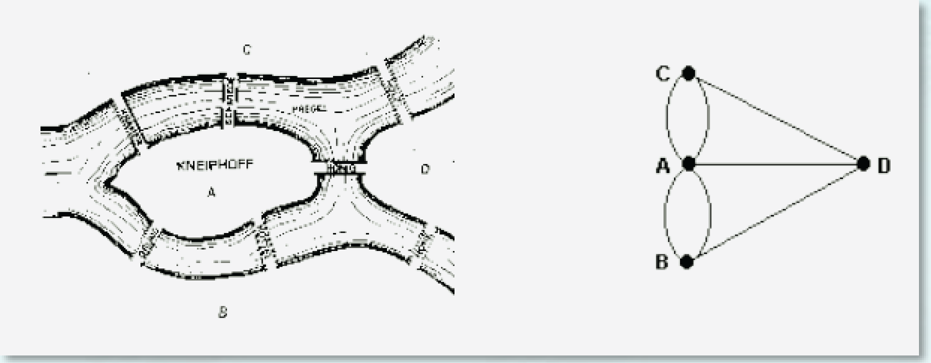
A NP complete problem

Graph-processing challenge 3

Problem: The Seven Bridge of Kőnigsberg. [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."*

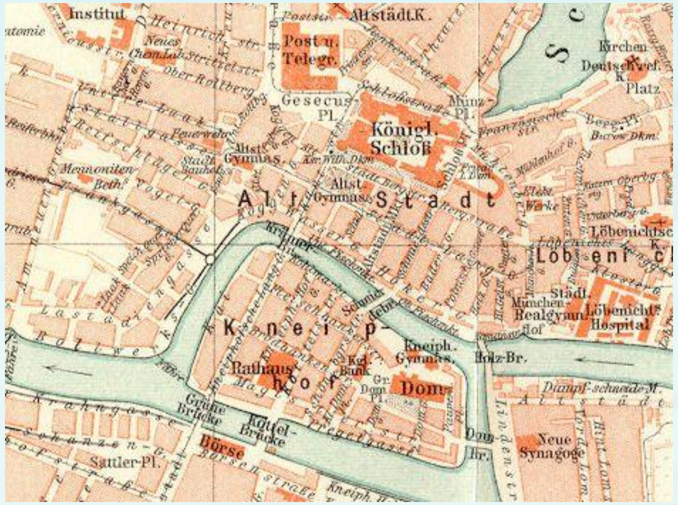


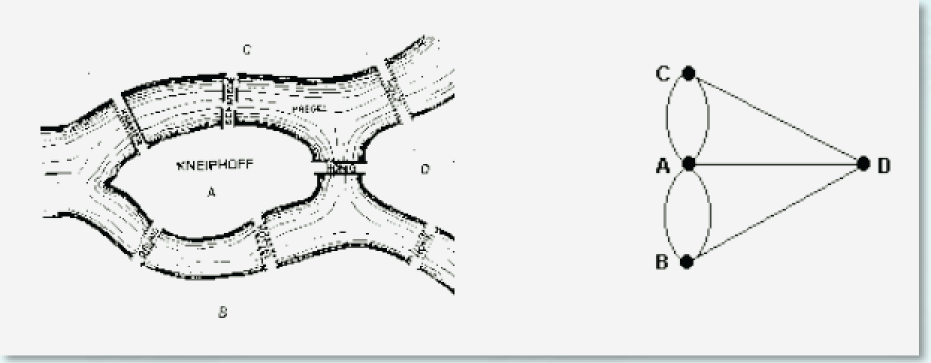


Graph-processing challenge 3

Problem: The Seven Bridge of Kőnigsberg. [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."*



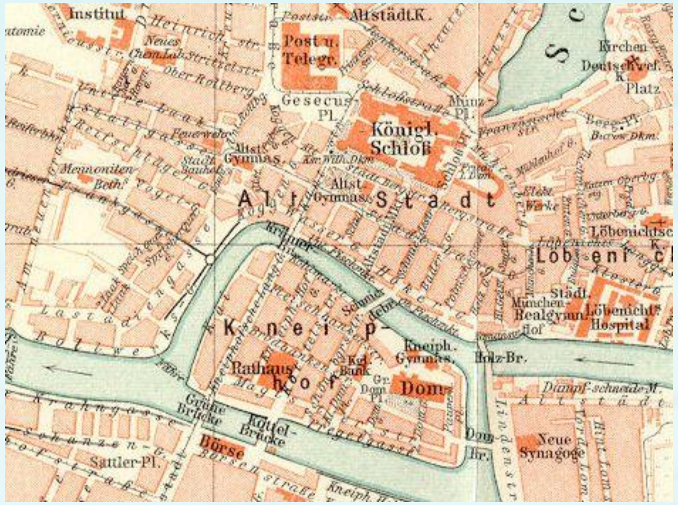


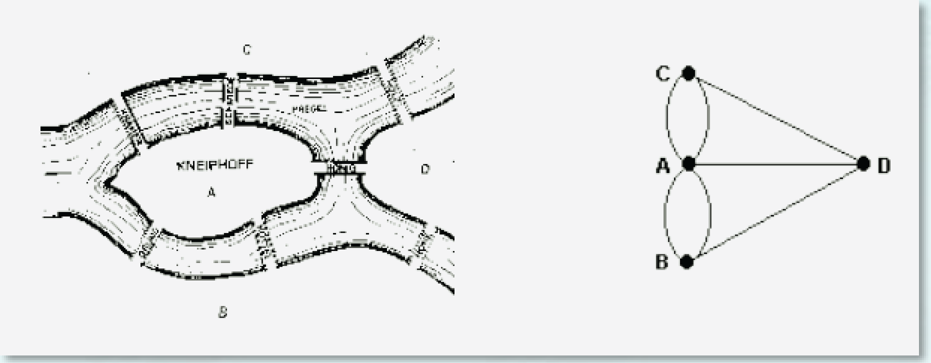
Euler tour: Is there a (general) cycle that uses each **edge** exactly once?

Graph-processing challenge 3

Problem: The Seven Bridge of Kőnigsberg. [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. "

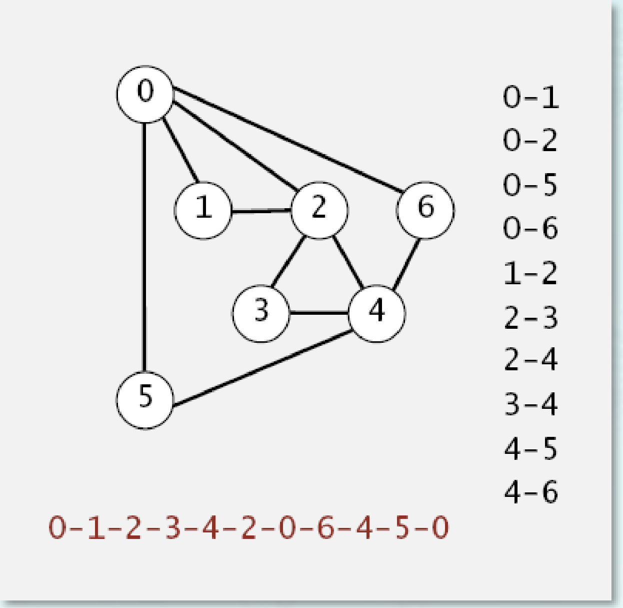




Euler tour: Is there a (general) cycle that uses each **edge** exactly once?
Answer: A connected graph is Eulerian iff all vertices have **even** degree.

Graph-processing challenge 3

Problem: Find a (general) cycle that uses every **edge exactly once**.



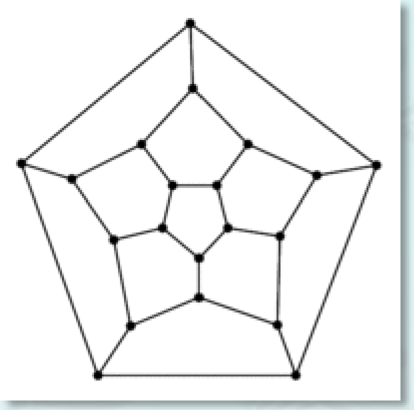
How difficult? Euler tour:

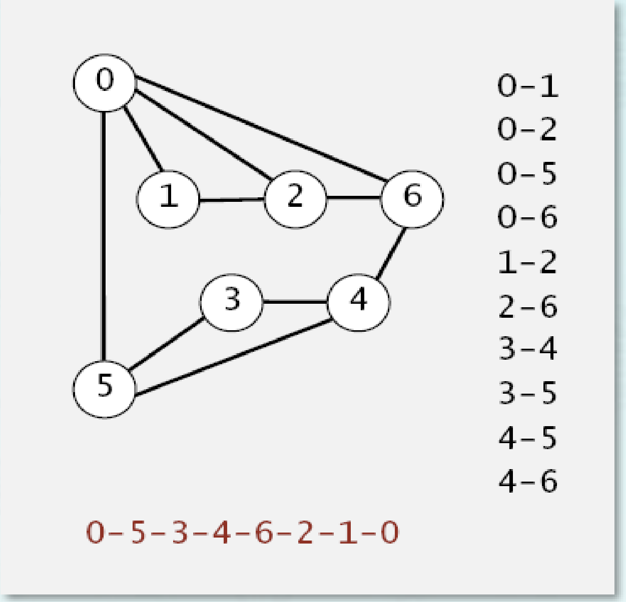
- Any programmer could do it.
- Typical diligent algorithms student could do it.
- Hire an expert.
- Intractable.
- No one knows.
- Impossible.

Eulerian tour
(classic graph-processing problem)

Graph-processing challenge 4

Problem: Find a cycle that visits every **vertex exactly once**.



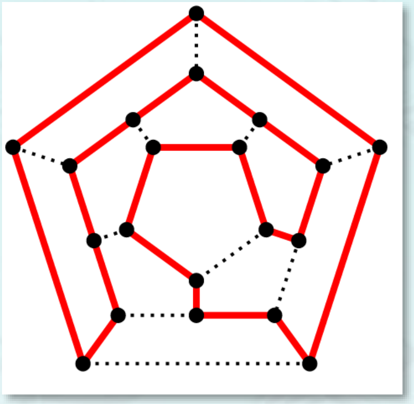


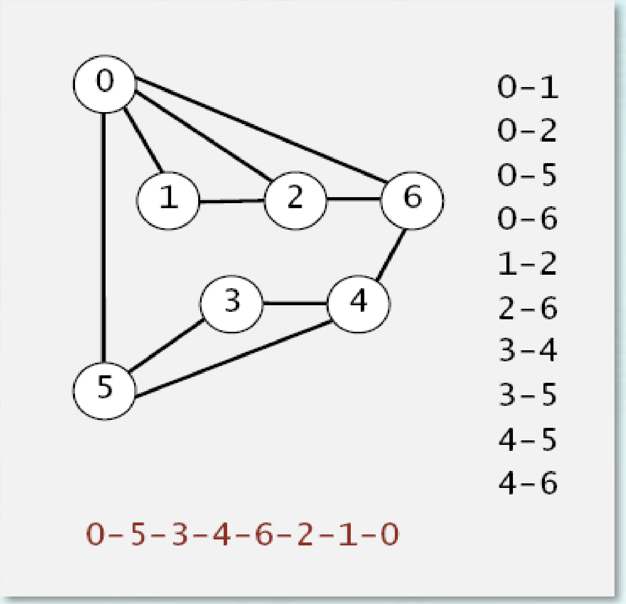
How difficult?

- Any programmer could do it.
- Typical diligent algorithms student could do i
- Hire an expert.
- Intractable.
- No one knows.
- Impossible.

Graph-processing challenge 4

Problem: Find a cycle that visits every **vertex exactly once.**





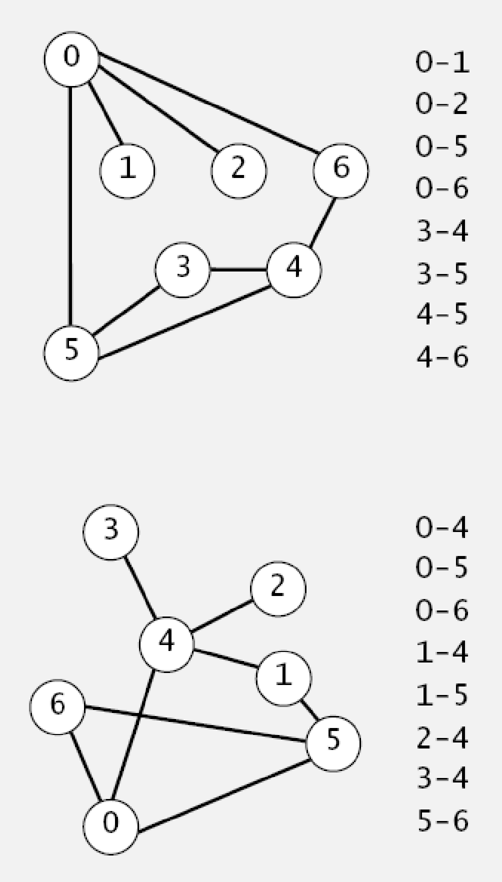
How difficult? **Hamilton tour**:

- Any programmer could do it.
- Typical diligent algorithms student could do i
- Hire an expert.
- Intractable.
- No one knows.
- Impossible.

Hamilton cycle
(classic NP-complete
problem)

Graph-processing challenge 5

Problem: Are **two graphs identical** except for vertex names?

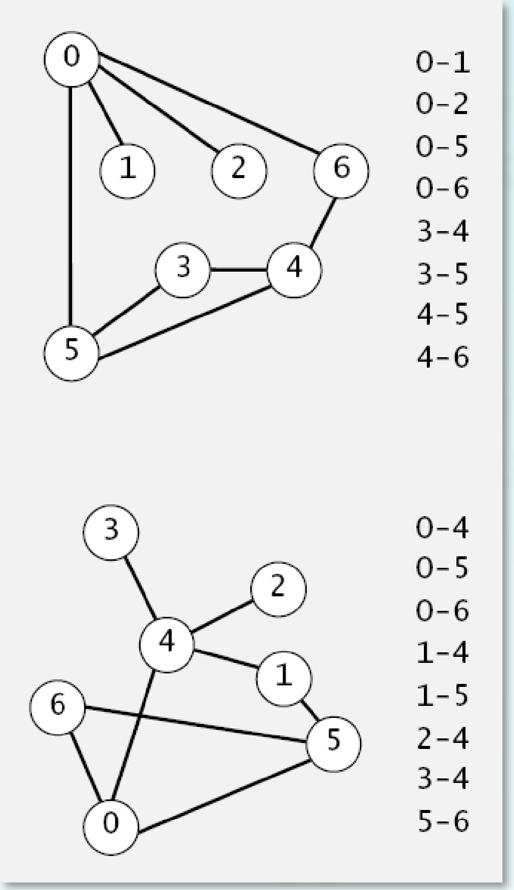


0↔4, 1↔3, 2↔2, 3↔6, 4↔5, 5↔0, 6↔1

How difficult?
- Any programmer could do it.
- Typical diligent algorithms student could do it.
- Hire an expert.
- Intractable.
- No one knows.
- Impossible.

Graph-processing challenge 5

Problem: Are **two graphs identical** except for vertex names?



0↔4, 1↔3, 2↔2, 3↔6, 4↔5, 5↔0, 6↔1

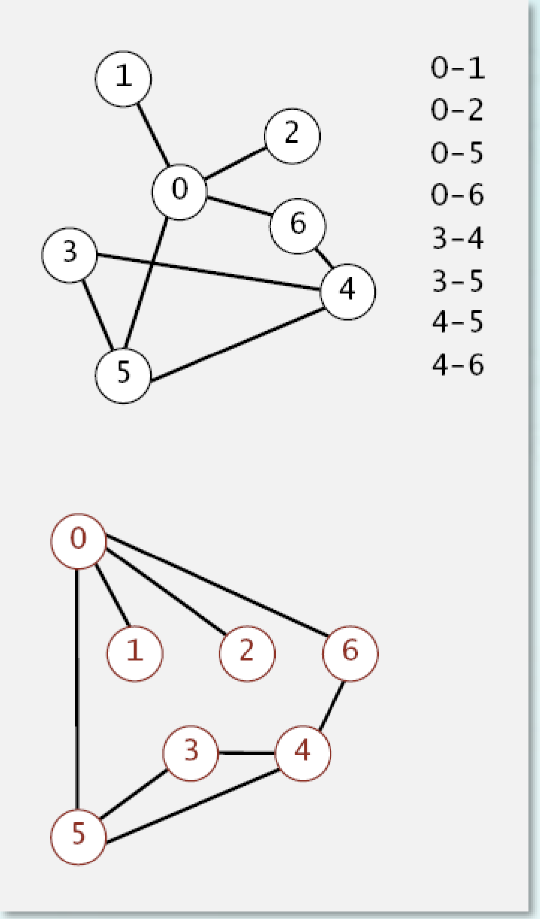
How difficult?

- Any programmer could do it.
- Typical diligent algorithms student could do it.
- Hire an expert.
- Intractable.
- No one knows.
- Impossible.

graph **isomorphism** is longstanding open problem

Graph-processing challenge 6

Problem: Lay out a graph in the plane **without crossing edges**?

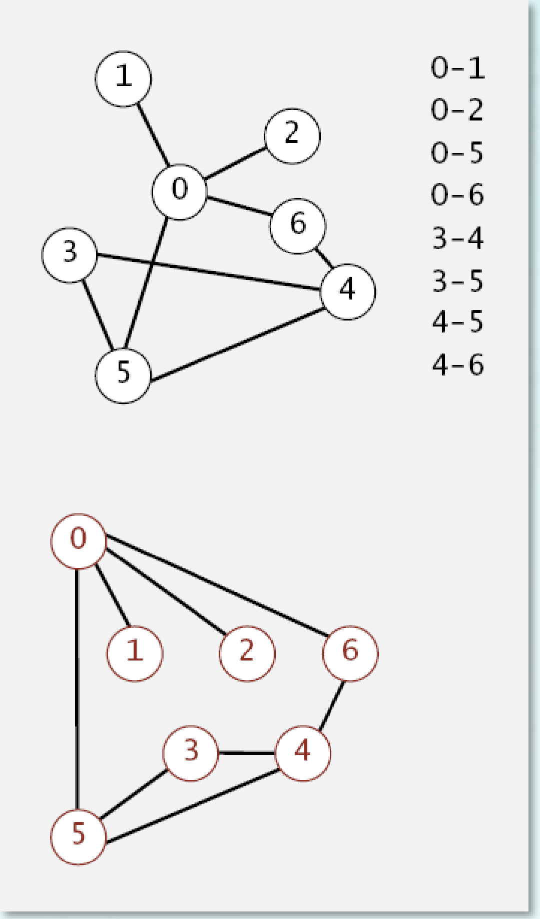


How difficult?

- Any programmer could do it.
- Typical diligent algorithms student could do it.
- Hire an expert.
- Intractable.
- No one knows.
- Impossible.

Graph-processing challenge 6

Problem: Lay out a graph in the plane **without crossing edges**?



How difficult?

- Any programmer could do it.
- Typical diligent algorithms student could do it.
- Hire an expert.
- Intractable.
- No one knows.
- Impossible.

linear-time DFS-based planarity algorithm
discovered by Tarjan in 1970s
(too complicated for most practitioners)

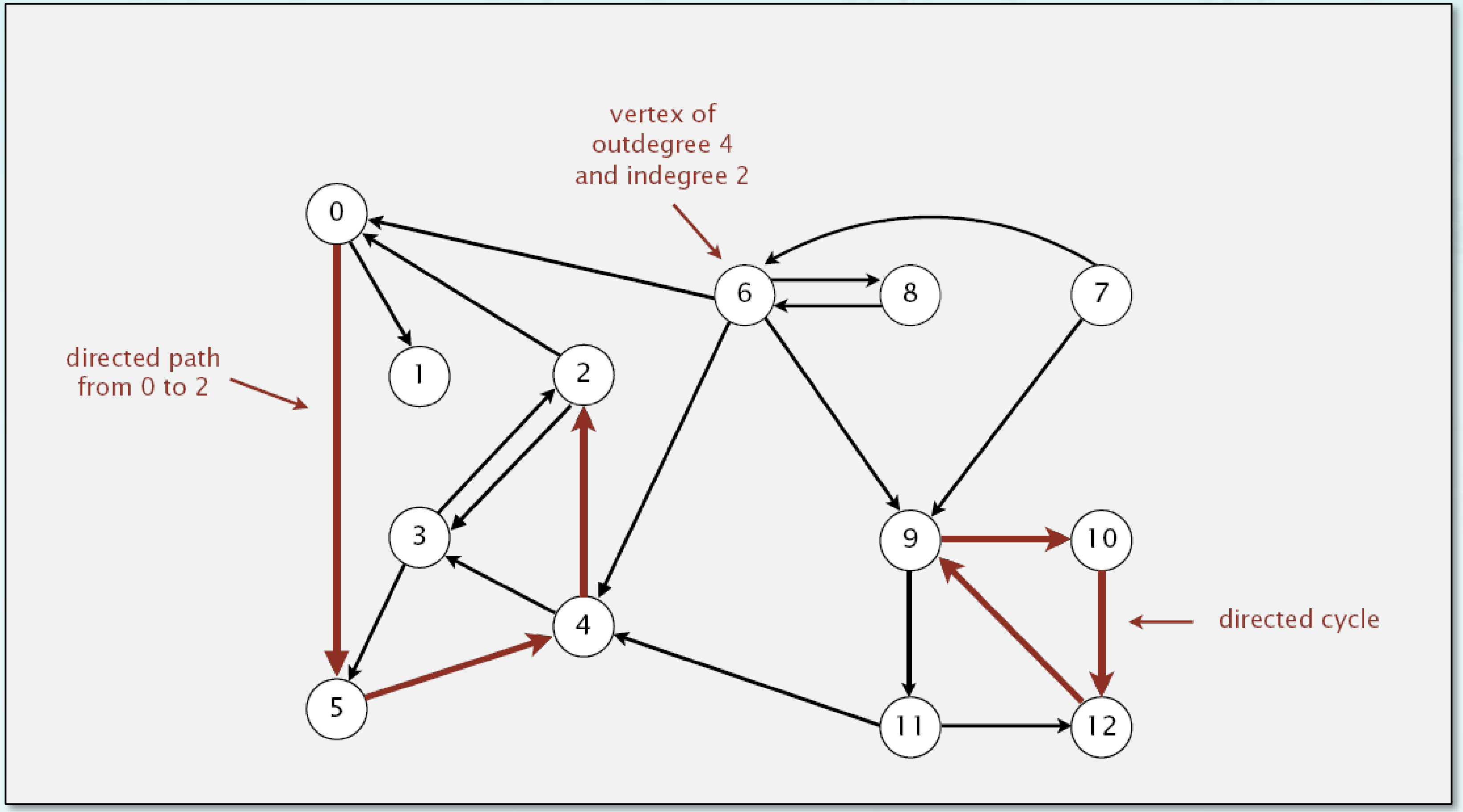
Graph

- *Graph*
  - *Introduction*
  - *Adjacency list*
  - *DFS, BFS*
  - *Challenges*
- ***Digraph – Directed Graphs***
  - *digraph – DFS, BFS*
  - *Applications – crawl web, topological sort*
- *Minimum Spanning Tree(MST)*

Major references:
1. Fundamentals of Data Structures by Horowitz, Sahni, Anderson-Freed,
2. Algorithms 4th edition - Part 1 & Part 2 by Robert Sedgewick and Kevin Wayne
3. Wikipedia and many resources available from internet

Directed graphs

Digraph: Set of vertices connected pairwise by directed edges.

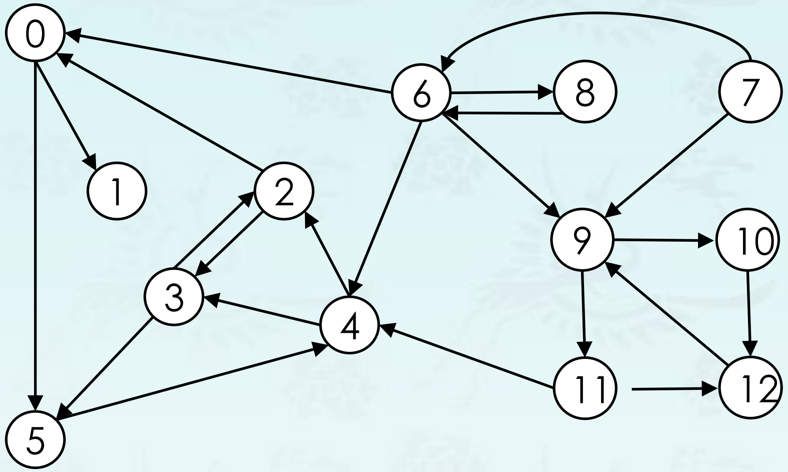


Digraph API

public class Digraph		
	Digraph(int V)	*create an empty digraph with V vertices*
	Digraph(In in)	*create a digraph from input stream*
void	addEdge(int v, int w)	*add a directed edge v→w*
Iterable<Integer>	adj(int v)	*vertices pointing from v*
int	V()	*number of vertices*
int	E()	*number of edges*
Digraph	reverse()	*reverse of this digraph*
String	toString()	*string representation*

Digraph API

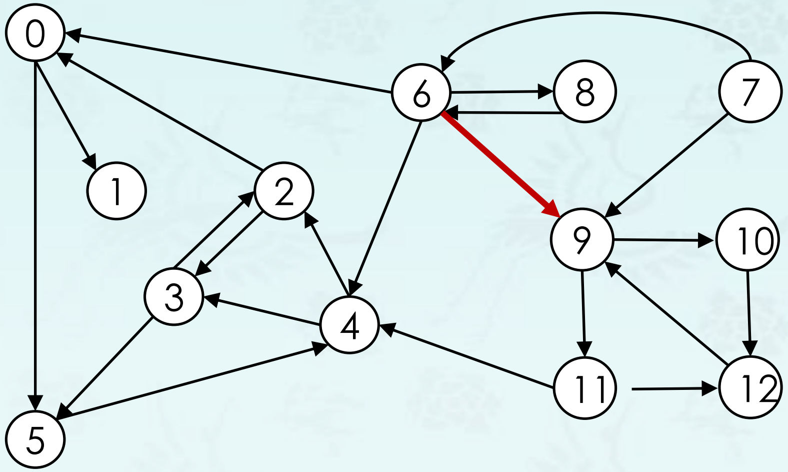
```
myG.txt
13   ← V
22   ← E
4 2
2 3
3 2
6 0
0 1
2 0
11 12
12 9
9 10
9 11
7 9
10 12
11 4
4 3
3 5
6 8
8 6
5 4
0 5
6 4
6 9
7 6
```

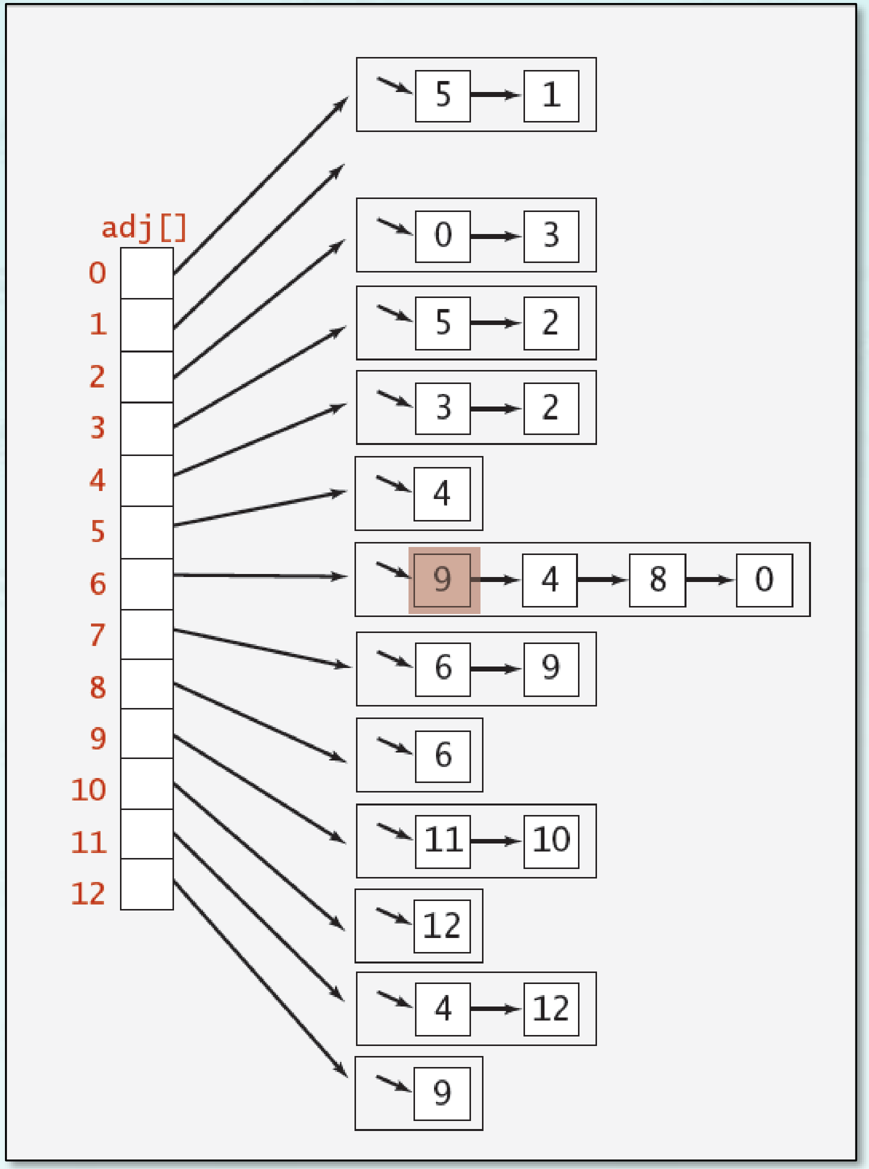


Adjacency-lists digraph representation

Maintain vertex-indexed array of lists.

```
myG.txt
13  ← V
22  ← E
4 2
2 3
3 2
6 0
0 1
2 0
11 12
12 9
9 10
9 11
7 9
10 12
11 4
4 3
3 5
6 8
8 6
5 4
0 5
6 4
6 9
7 6
```





Adjacency-lists graph representation (review) in Java

```
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) {
    add[v].add(w);
    add[w].add(v);
  }

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

adjacency lists (using Bag data type)

create empty graph with V vertices

add edge v-w (parallel edges and self-loops allowed)

iterator for vertices adjacent to v

Adjacency-lists digraph representation in Java

```java
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) {
    add[v].add(w);

  }

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

adjacency lists
(using Bag data type)

create empty graph
with V vertices

add edge v -> w

iterator for vertices
pointing from v

Digraph representations

In practice: Use adjacency-lists representation.

- Algorithms based on iterating over vertices pointing from v.
- Real-world digraphs tend to be **sparse.**

huge number of vertices,
small average vertex degree

representation	space	add edge	edge between v and w?	iterate over vertices adjacent to v?
list of edges	E	1	E	E
adjacency matrix	V^2	1	1	V
adjacency lists	$E + V$	1	$outdegree(v)$	$outdegree(v)$

Graph

- *Digraph – Directed Graphs*
  - *Introduction*
  - *digraph API*
  - ***digraph search***

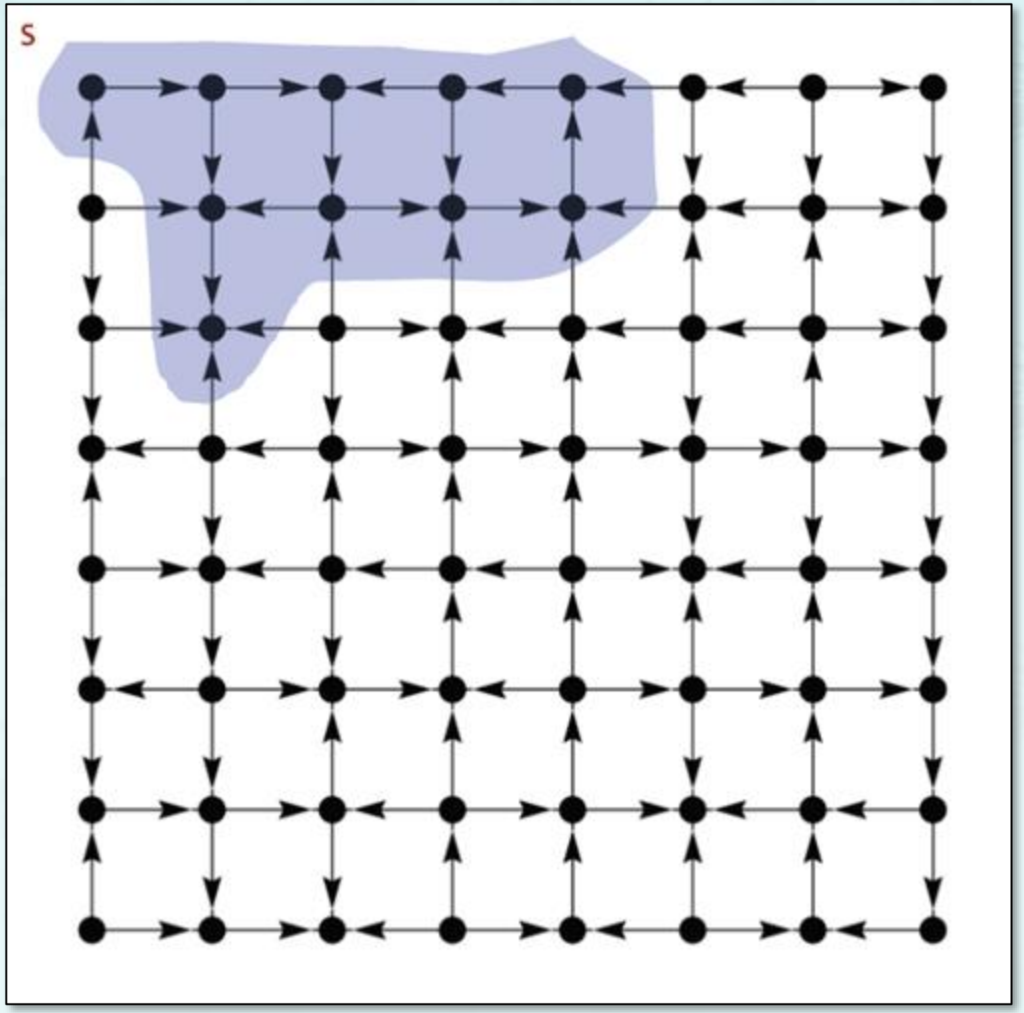
Major references:

1. Fundamentals of Data Structures by Horowitz, Sahni, Anderson-Freed,
2. Algorithms 4th edition - Part 1 & Part 2 by Robert Sedgewick and Kevin Wayne
3. Wikipedia and many resources available from internet

Prof. Youngsup Kim, idebtor@handong.edu, 2014 Data Structures, CSEE Dept., Handong Global University

Reachability

Problem: Find all vertices reachable from **s** along a directed path



Depth-first search in digraphs

Same methods as for undirected graphs:

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

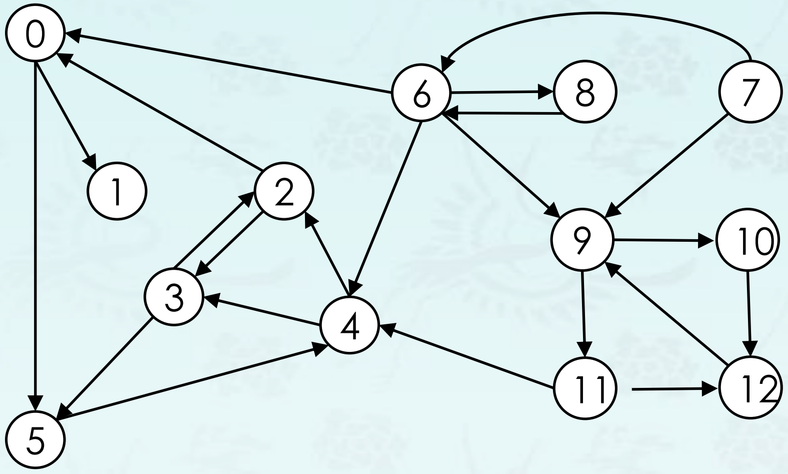
DFS (to visit a vertex v)

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

Depth-first search in digraphs

To visit a vertex v:

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



a directed graph

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

Digraph API - Quiz

To visit a vertex v:

- Suppose that a digraph G is represented using the adjacency-lists representation. What is the order of growth of the running time to find all vertices that **point to** a given vertex v or indegree of v?

___ indgree(v)
___ outdegree(v)
___ V
___ E
___ V * E
___ V + E

Solution: You must scan through each of the V adjacency lists and each of the E edges. If this were a common operation in digraph-processing problems, you could associate two adjacency lists with each vertex—one containing all of the vertices pointing from v (as usual) and one containing all of the vertices pointing to v.

Digraph API - Quiz

To visit a vertex v:

- Suppose that a digraph G is represented using the adjacency-lists representation. What is the order of growth of the running time to find all vertices that **point to** a given vertex v or indegree of v?

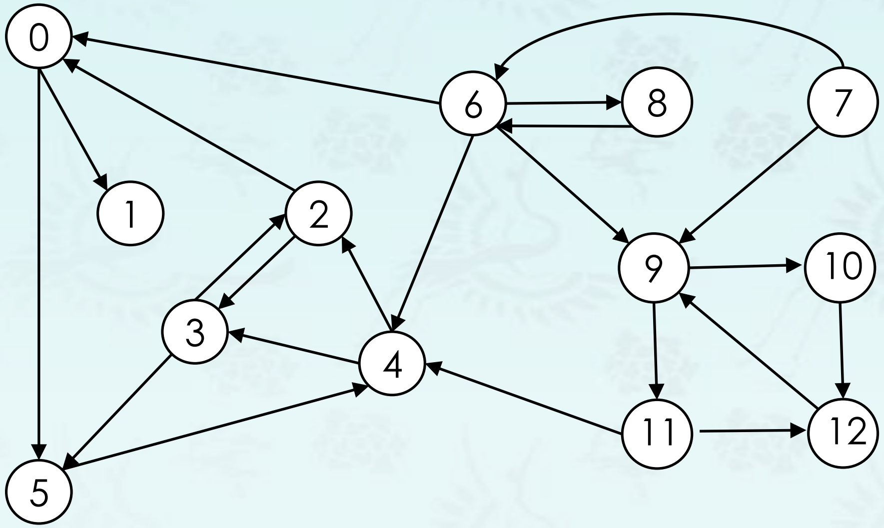
___ indgree(v)
___ outdegree(v)
___ V
___ E
___ V * E
___ **V + E**

Solution: You must scan through each of the V adjacency lists and each of the E edges. If this were a common operation in digraph-processing problems, you could associate two adjacency lists with each vertex—one containing all of the vertices pointing from v (as usual) and one containing all of the vertices pointing to v.

Depth-first search demo

To visit a vertex v:

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



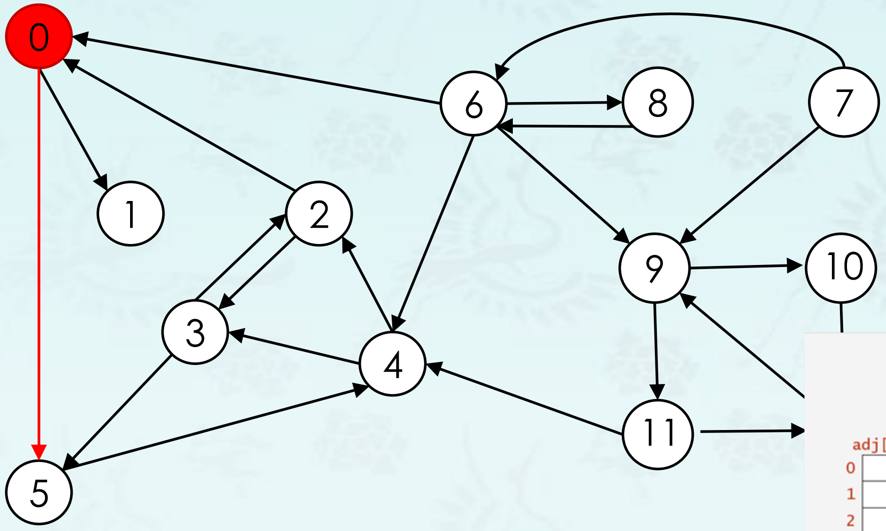
a directed graph

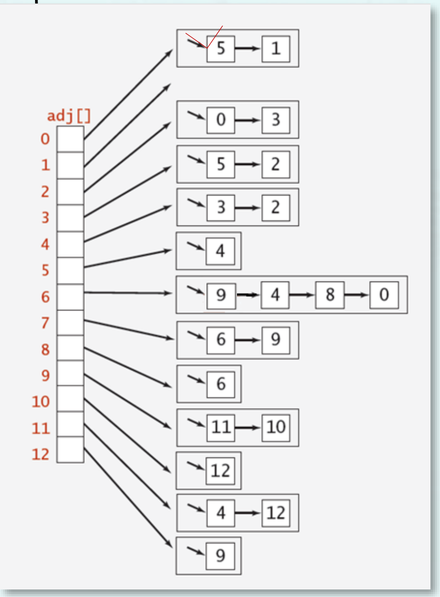
4->2
2->3
3->2
6->0
0->1
2->0
11->12
12->9
9->10
9->11
8->9
10->12
11->4
4->3
3->5
6->8
8->6
5->4
0->5
6->4
6->9
7->6

Depth-first search demo

To visit a vertex v:

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





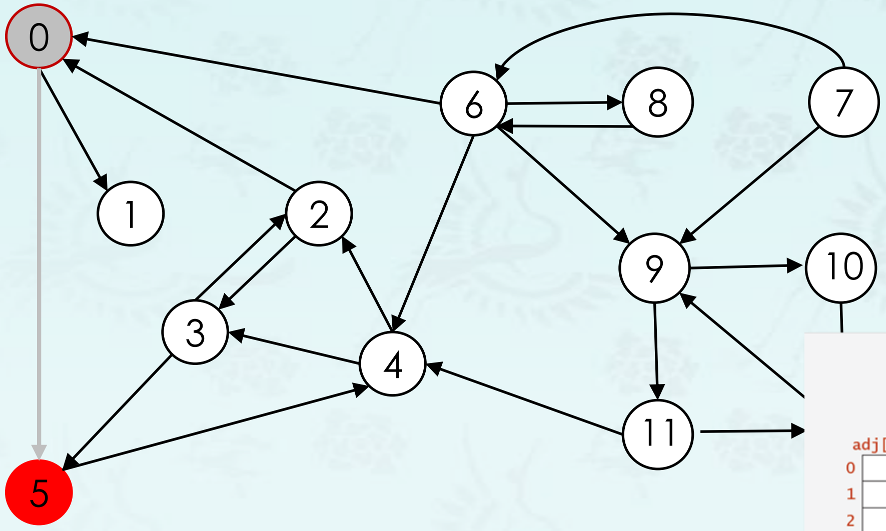
v	marked[]	parent[v]
0	T	–
1	F	–
2	F	–
3	F	–
4	F	–
5	F	–
6	F	–
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

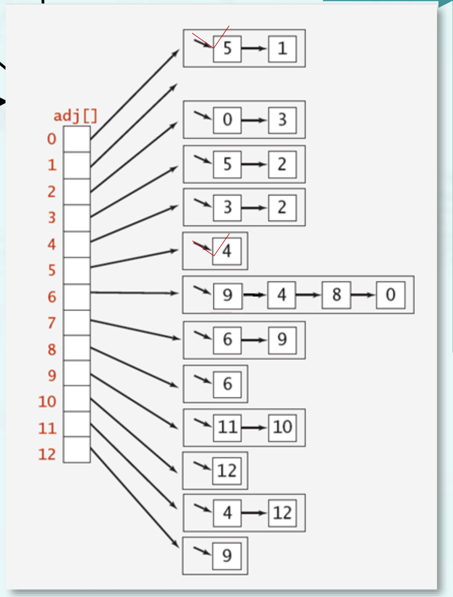
visit 0: check 5 and check 1

Depth-first search demo

To visit a vertex v:

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





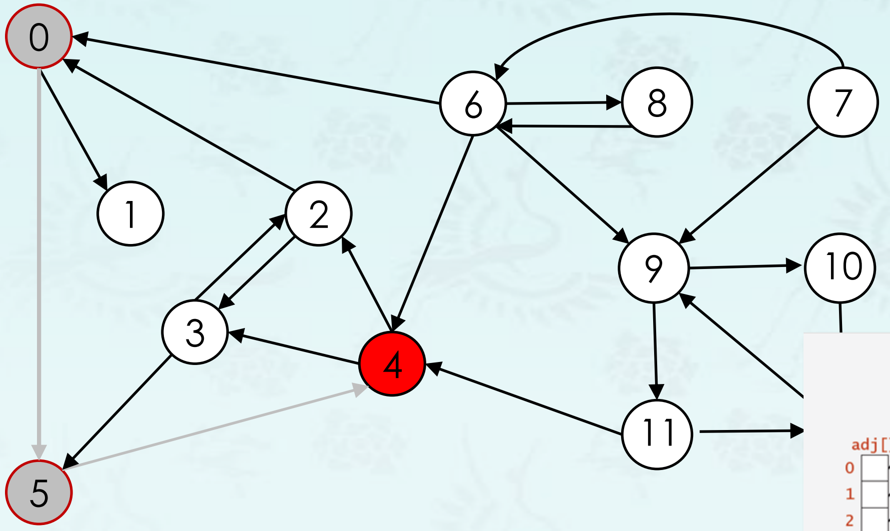
v	marked[]	parent[v]
0	T	–
1	F	–
2	F	–
3	F	–
4	F	–
5	T	0
6	F	–
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

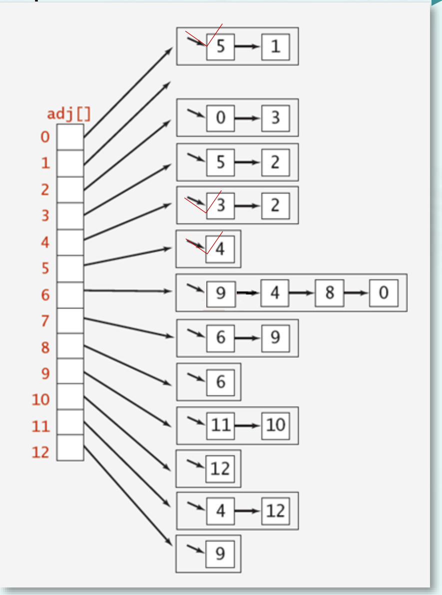
visit 5: check 4

Depth-first search demo

To visit a vertex v:

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





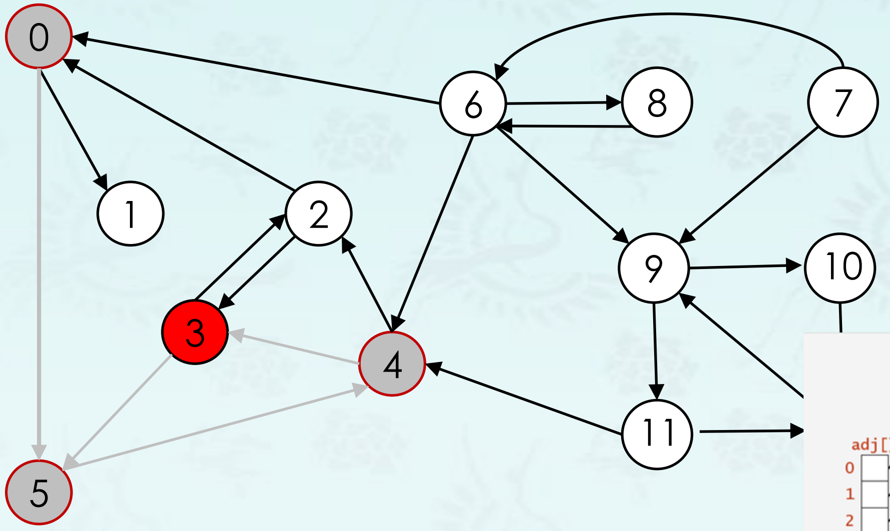
v	marked[]	parent[v]
0	T	–
1	F	–
2	F	–
3	F	–
4	T	5
5	T	0
6	F	–
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

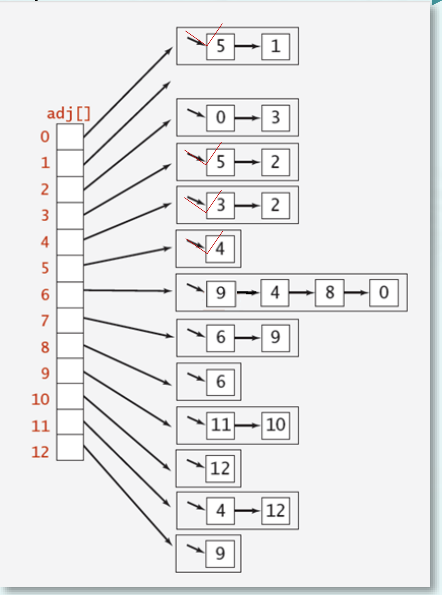
visit 4: check 3 and check 2

Depth-first search demo

To visit a vertex v:

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





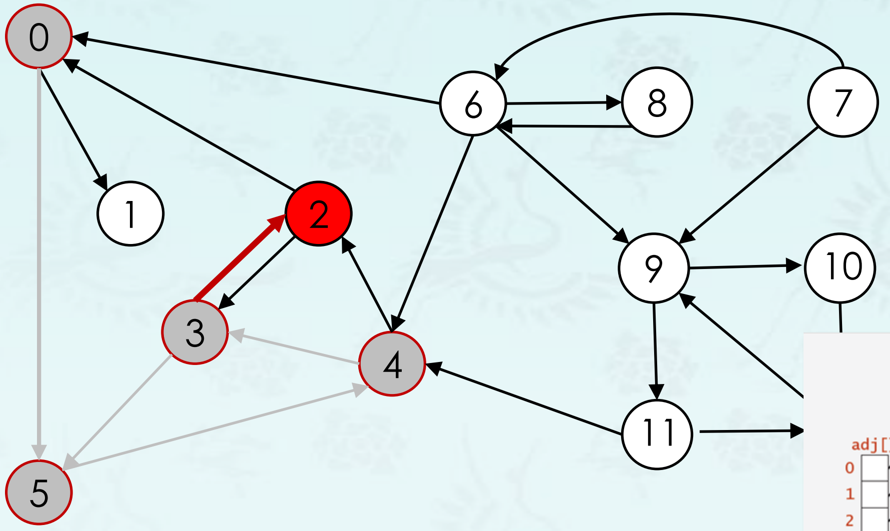
v	marked[]	parent[v]
0	T	–
1	F	–
2	T	–
3	T	4
4	T	5
5	T	0
6	F	–
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

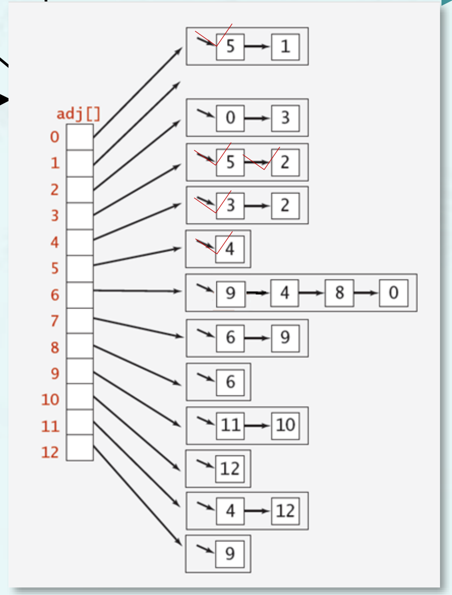
visit 3: check 5 and check 2

Depth-first search demo

To visit a vertex v:

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





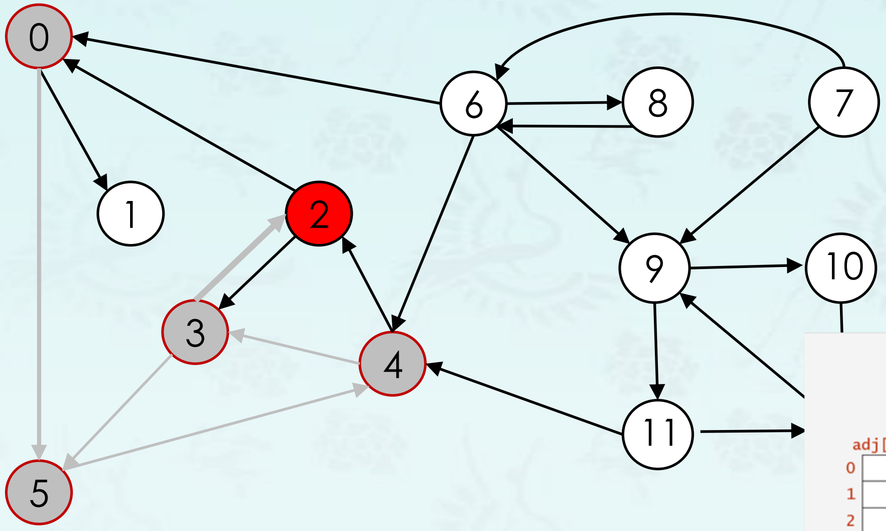
v	marked[]	parent[v]
0	T	–
1	F	–
2	T	–
3	T	4
4	T	5
5	T	0
6	F	–
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

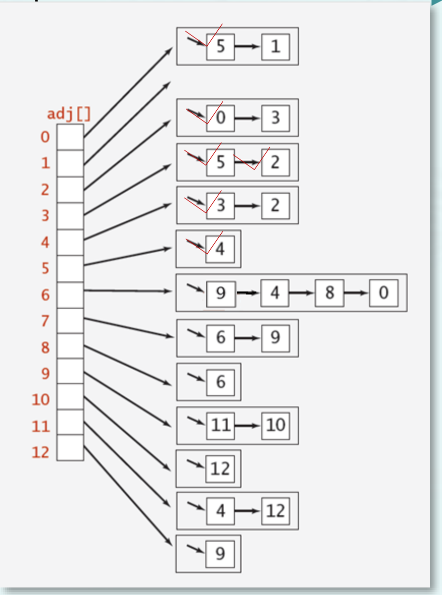
visit 3: check 5 **and check 2**

Depth-first search demo

To visit a vertex v:

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





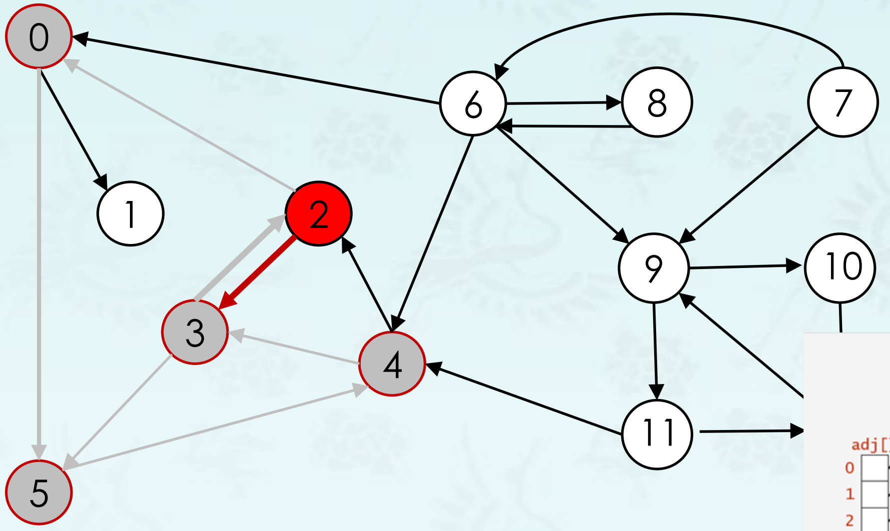
v	marked[]	parent[v]
0	T	–
1	F	–
2	T	3
3	T	4
4	T	5
5	T	0
6	F	–
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

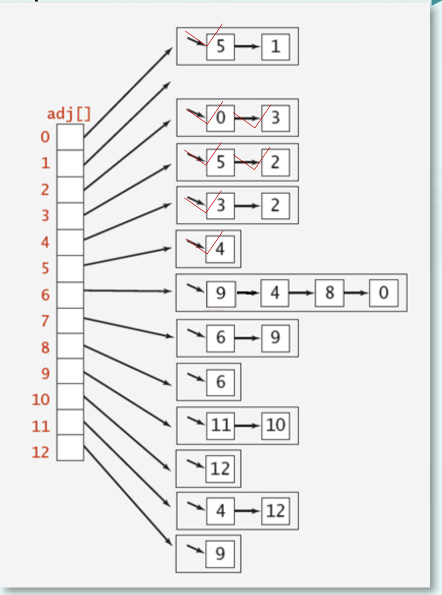
visit 2: **check 0** and **check 3**

Depth-first search demo

To visit a vertex v:

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





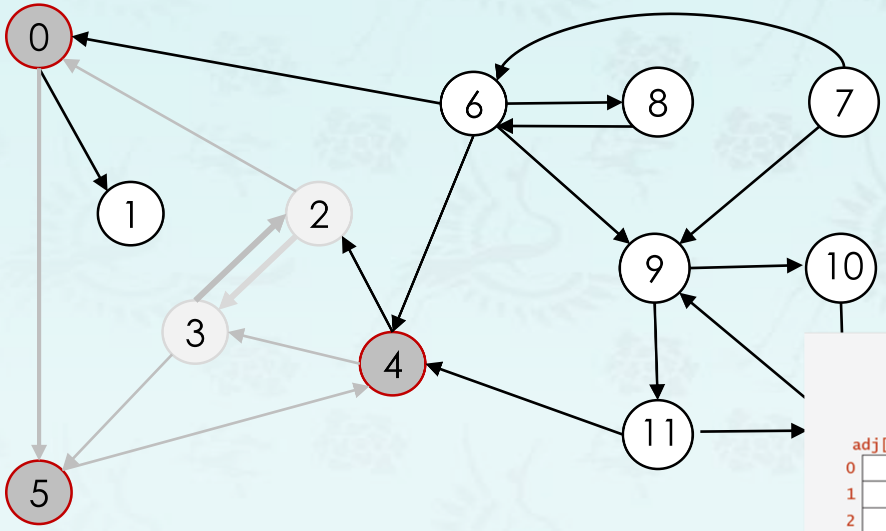
v	marked[]	parent[v]
0	T	–
1	F	–
2	T	3
3	T	4
4	T	5
5	T	0
6	F	–
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

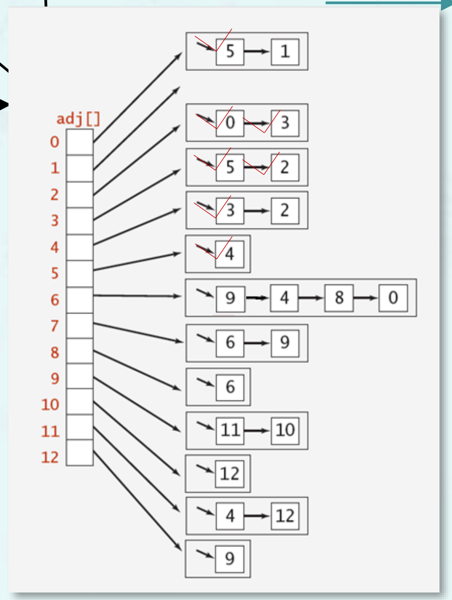
visit 2: check 0 and **check 3**

Depth-first search demo

To visit a vertex v:

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





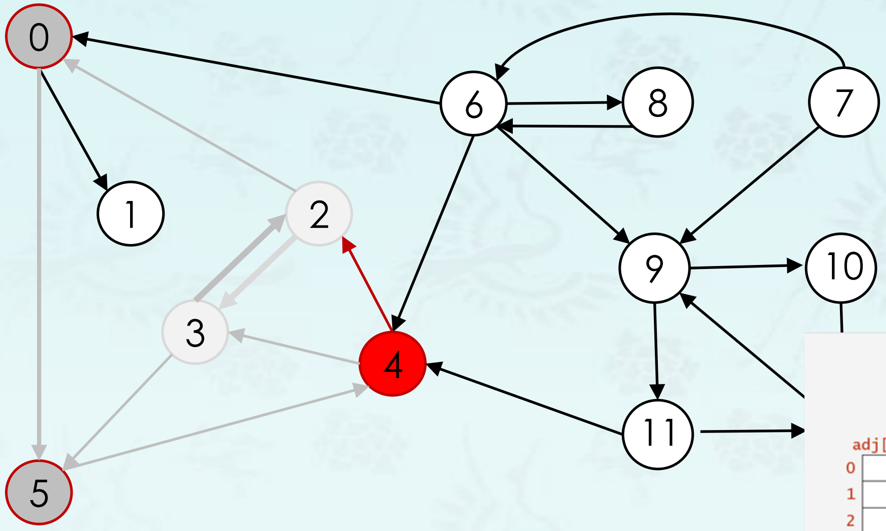
v	marked[]	parent[v]
0	T	–
1	F	–
2	T	3
3	T	4
4	T	5
5	T	0
6	F	–
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

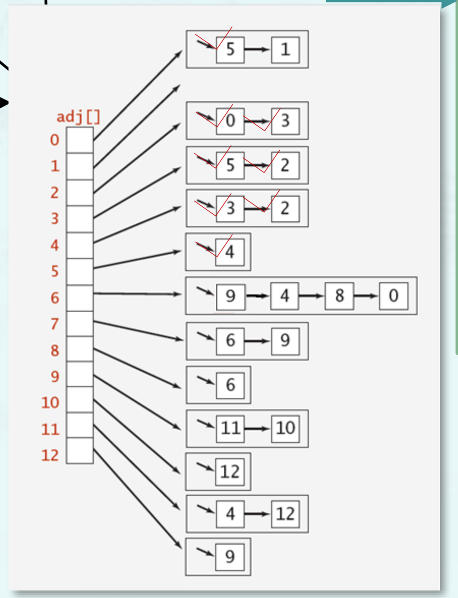
done **3**

Depth-first search demo

To visit a vertex v:

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





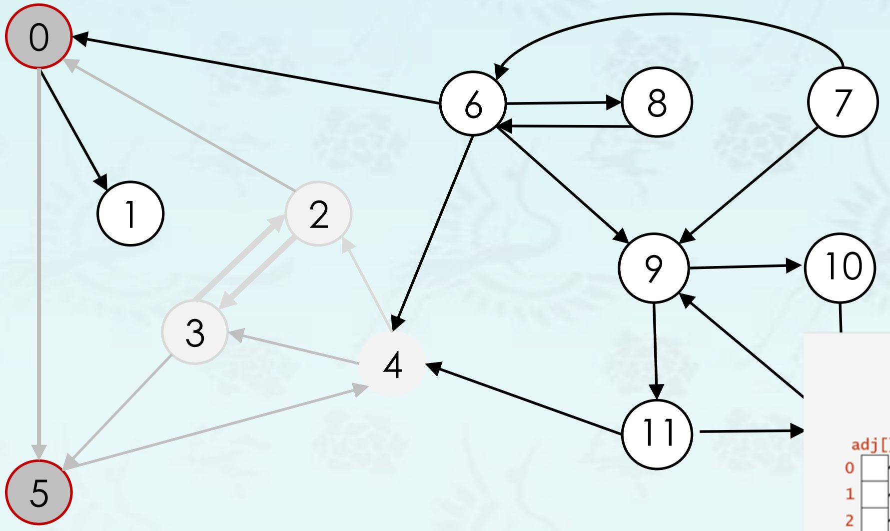
v	marked[]	parent[v]
0	T	–
1	F	–
2	T	3
3	T	4
4	T	5
5	T	0
6	F	–
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

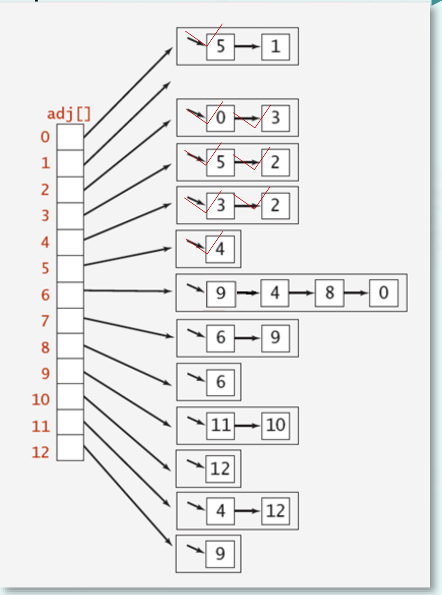
visit 4: **check 3 and check 2**

Depth-first search demo

To visit a vertex v:

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





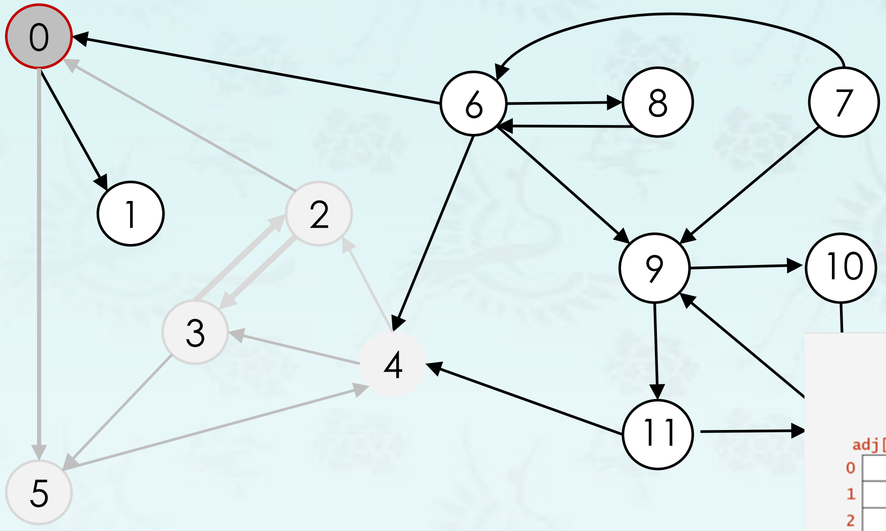
v	marked[]	parent[v]
0	T	–
1	F	–
2	T	3
3	T	4
4	T	5
5	T	0
6	F	–
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

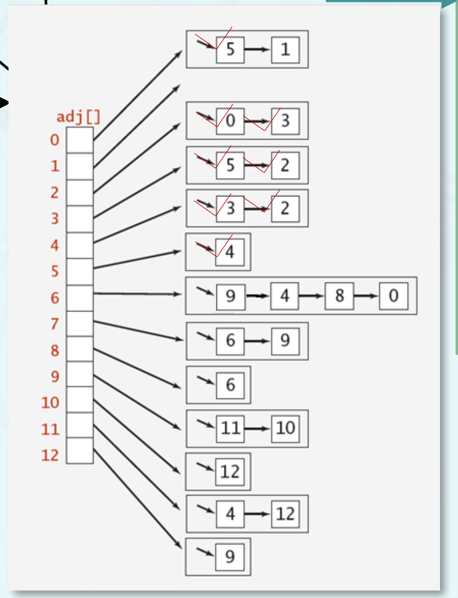
Done 4

Depth-first search demo

To visit a vertex v:

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





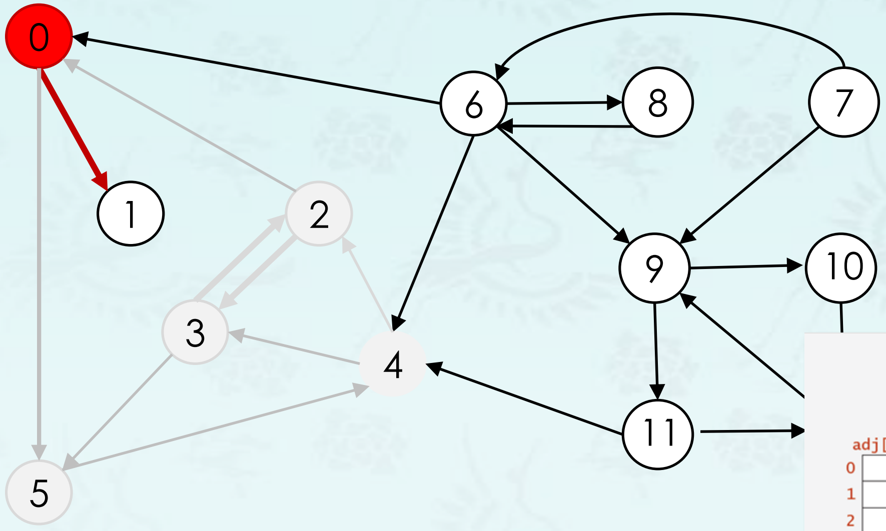
v	marked[]	parent[v]
0	T	–
1	F	–
2	T	3
3	T	4
4	T	5
5	T	0
6	F	–
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

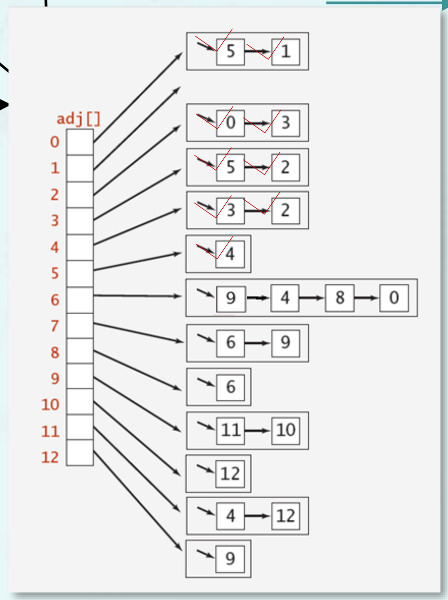
Done 5

Depth-first search demo

To visit a vertex v:

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





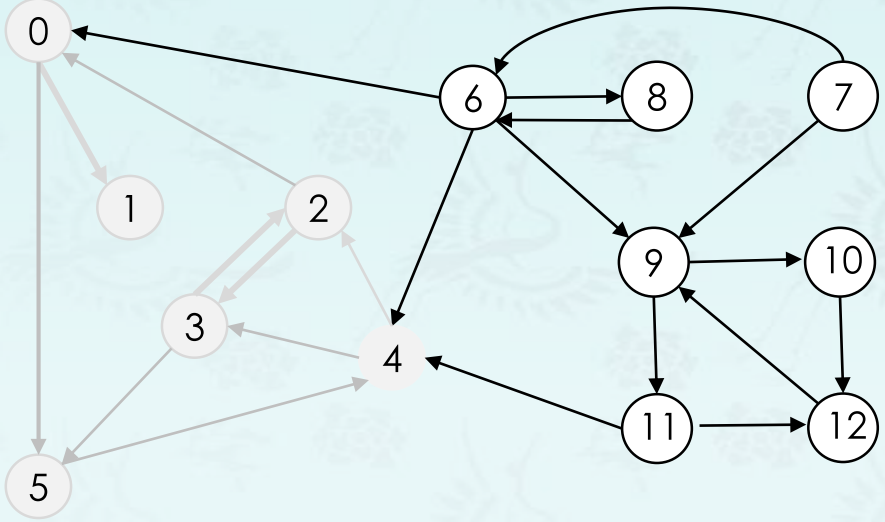
v	marked[]	parent[v]
0	T	–
1	T	0
2	T	3
3	T	4
4	T	5
5	T	0
6	F	–
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

visit 0: check 5 and **check 1**

Depth-first search demo

To visit a vertex v:

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



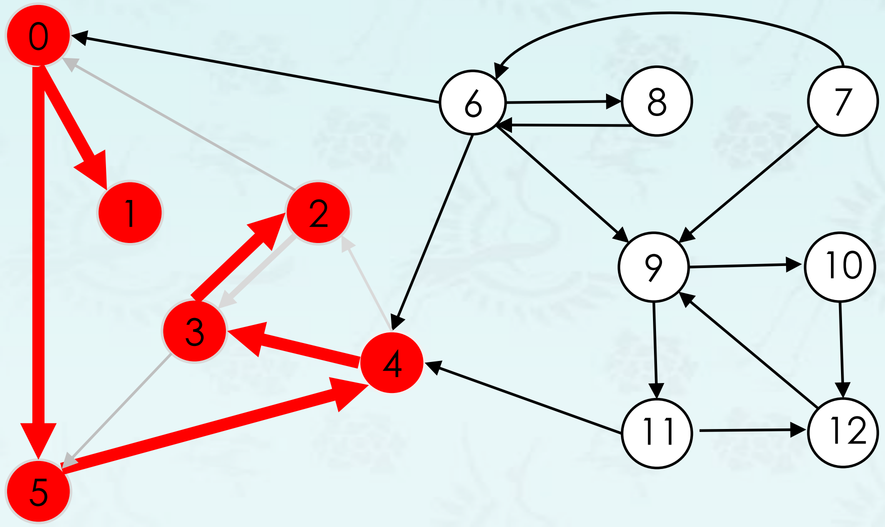
v	marked[]	parent[v]
0	T	–
1	T	0
2	T	3
3	T	4
4	T	5
5	T	0
6	F	–
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

1 done

Depth-first search demo

To visit a vertex v:

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



v	marked[]	parent[v]
0	T	-
1	T	0
2	T	3
3	T	4
4	T	5
5	T	0
6	F	-
7	F	-
8	F	-
9	F	-
10	F	-
11	F	-
12	F	-

reachable from vertex 0

reachable from 0

Depth-first search (in undirected graph) in 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 path to s

constructor marks vertices connected to s

recursive DFS does the work

client can ask whether any vertex connected to s

Depth-first search (in undirected graphs) in Java

Code for **directed** graphs identical to undirected one.
[Substitute Digraph for Graph.]

```
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 path to s

constructor marks vertices connected to s

recursive DFS does the work

client can ask whether any vertex is **reachable from s**

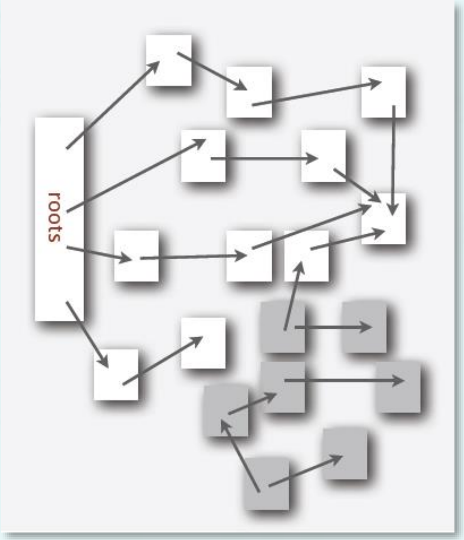
Reachability application: mark-sweep garbage collector

Every data structure (in java) 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).

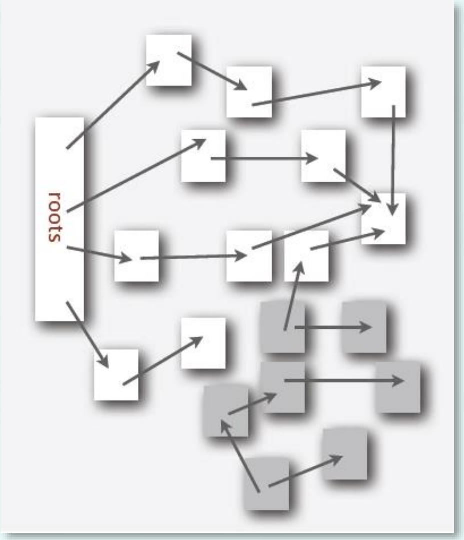


Reachability application: mark-sweep garbage collector

Mark-sweep algorithm (McCathy, 1960)

1. Mark data objects in a program that cannot be accessed in the future.
2. 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).



Graph

- *Challenges*
- *Digraph – Directed Graphs*
  - *Introduction*
  - *digraph API*
  - *digraph search – DFS*
  - ***digraph search – BFS***

Major references:
1. Fundamentals of Data Structures by Horowitz, Sahni, Anderson-Freed,
2. Algorithms 4th edition - Part 1 & Part 2 by Robert Sedgewick and Kevin Wayne
3. Wikipedia and many resources available from internet

Breadth-first search in digraph

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:

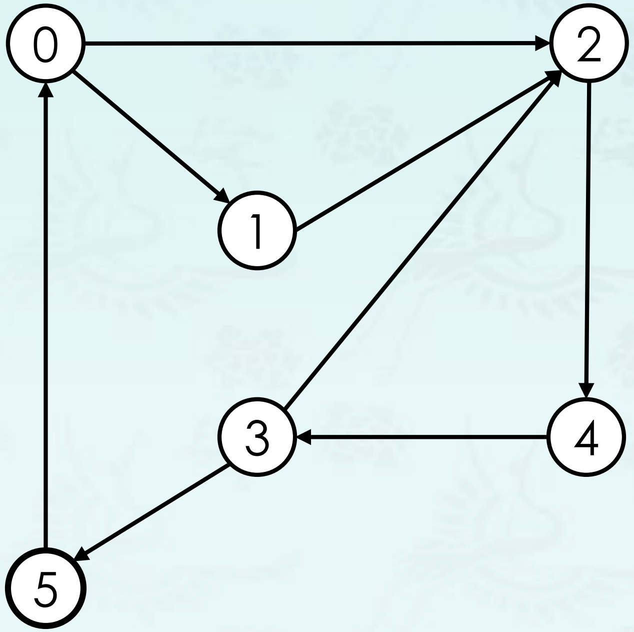
- **remove the least recently added vertex v**
- **for each unmarked vertex pointing from v:**
  add to queue and mark as visited.

Proposition: BFS computes shortest paths (fewest number of edges) from s to all other vertices in a digraph 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 pointing from v and mark them.



myDG.txt

```
6    ← V
8    ← E
5 0
2 4
3 2
1 2
0 1
4 3
3 5
0 2
```

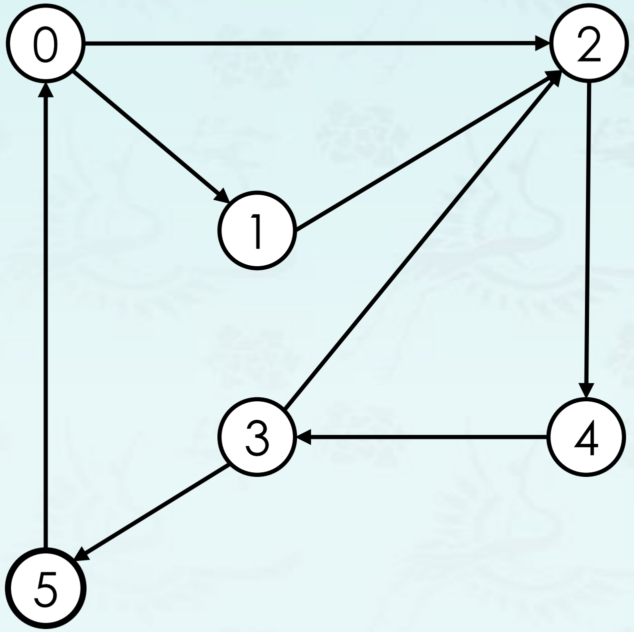
Challenge: build adjacency lists

Graph g:

Directed breadth-first search demo

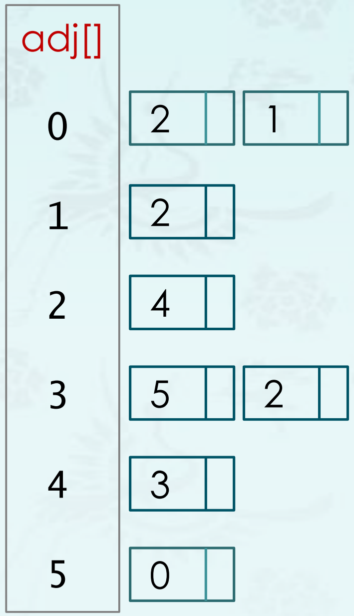
Repeat until queue is empty.

- Remove vertex v from queue.
- Add to queue all unmarked vertices pointing from v and mark them.



Graph g:

Adjacency lists



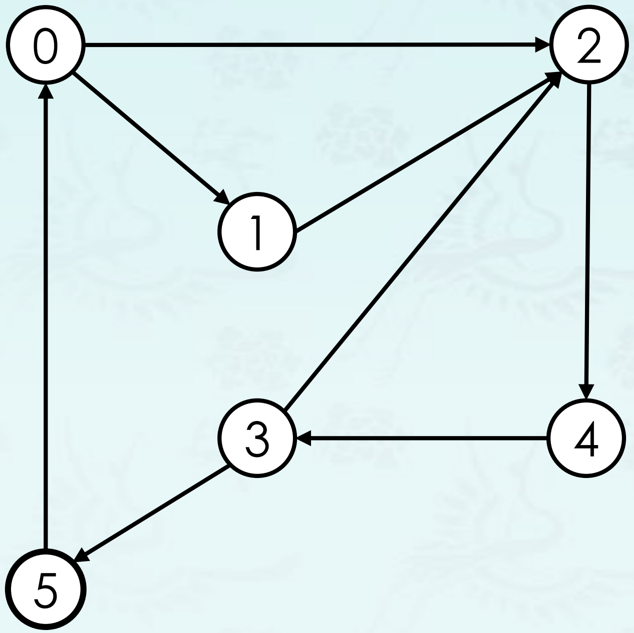

```
myDG.txt
6     V
8     E
5 0
2 4
3 2
1 2
0 1
4 3
3 5
0 2
```

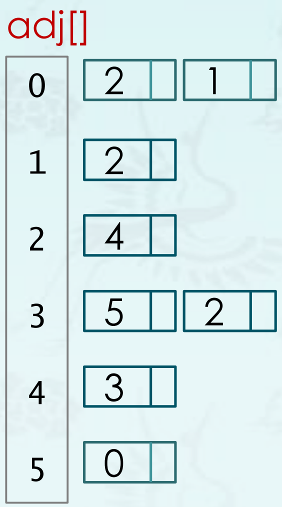
Challenge: build adjacency lists – Job done

Directed breadth-first search demo

Repeat until queue is empty.

- Remove vertex v from queue.
- Add to queue all unmarked vertices pointing from v and mark them.





queue: 0

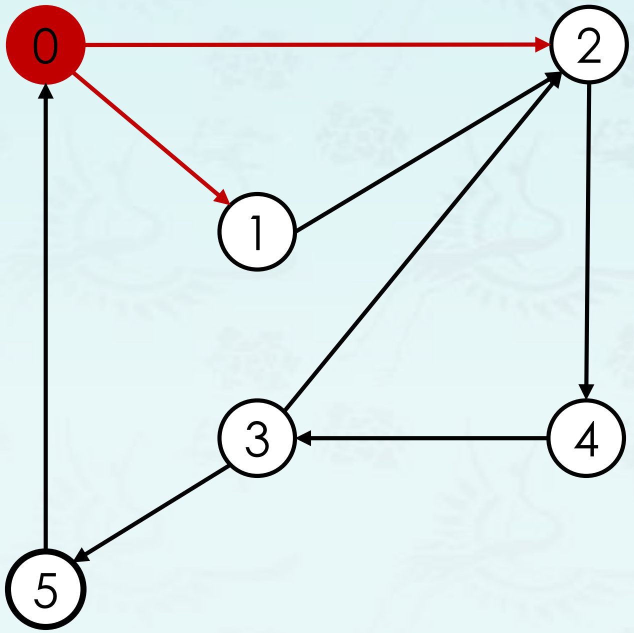
v	parent[]	distTo[]
0	–	0
1	–	–
2	–	–
3	–	–
4	–	–
5	–	–

Graph g:

Directed breadth-first search demo

Repeat until queue is empty.

- Remove vertex v from queue.
- Add to queue all unmarked vertices pointing from v and mark them.



adj[]

0	2 → 1
1	2
2	4
3	5 → 2
4	3
5	0

queue

2

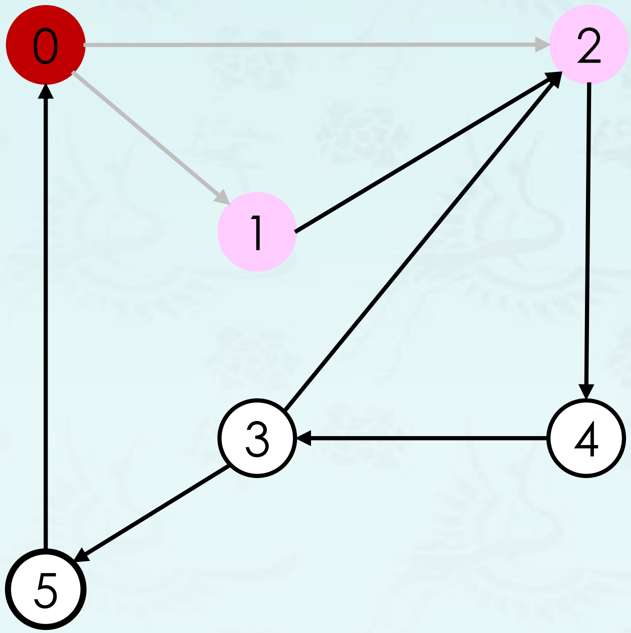
v	parent[]	distTo[]
0	–	0
1	–	–
2	0	1
3	–	–
4	–	–
5	–	–

dequeue 0: check 2 and check 1

Directed breadth-first search demo

Repeat until queue is empty.

- Remove vertex v from queue.
- Add to queue all unmarked vertices pointing from v and mark them.



adj[]

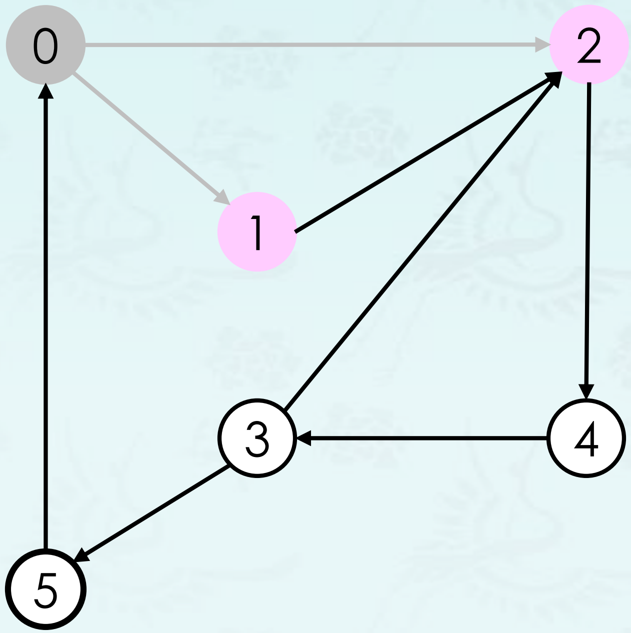
0	2	1
1	2	
2	4	
3	5	2
4	3	
5	0	

queue: 1, 2

v	parent[]	distTo[]
0	–	0
1	0	1
2	0	1
3	–	–
4	–	–
5	–	–

dequeue 0: check 2 and check 1

Directed breadth-first search demo



adj[]

0	2	1
1	2	
2	4	
3	5	2
4	3	
5	0	

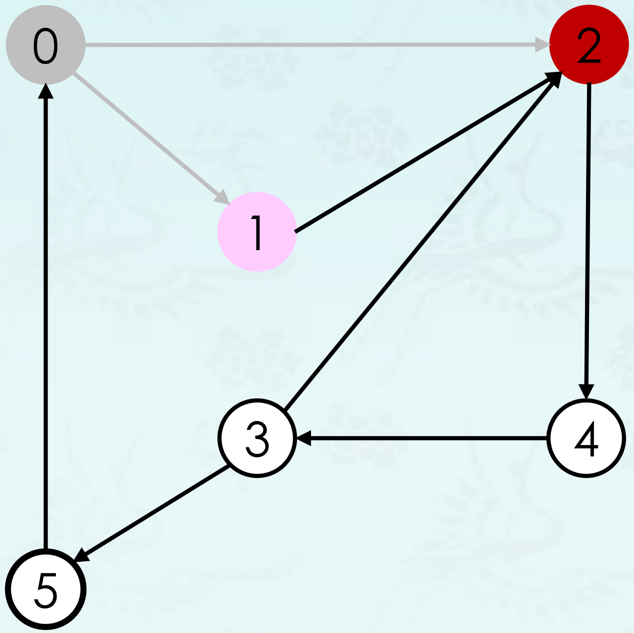
queue

1
2

v	parent[]	distTo[]
0	–	0
1	0	1
2	0	1
3	–	–
4	–	–
5	–	–

0 done

Directed breadth-first search demo



adj[]

0	2	1
1	2	
2	4	
3	5	2
4	3	
5	0	

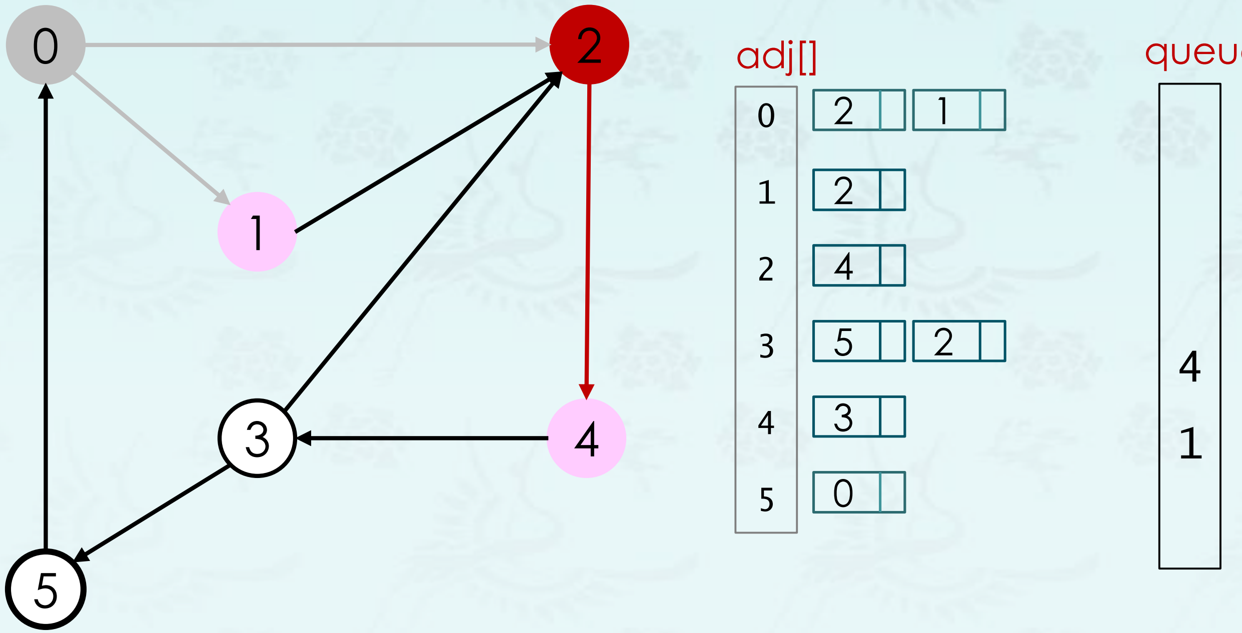
queue

1

v	parent[]	distTo[]
0	–	0
1	0	1
2	0	1
3	–	–
4	–	–
5	–	–

dequeue 2

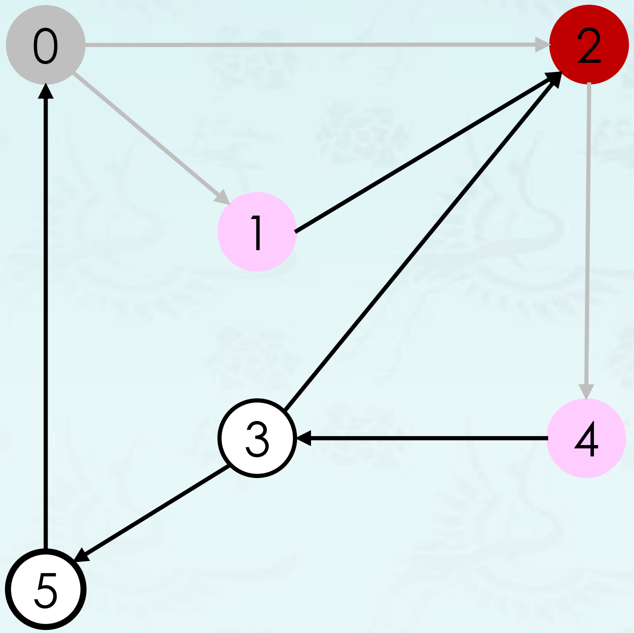
Directed breadth-first search demo



v	parent[]	distTo[]
0	–	0
1	0	1
2	0	1
3	–	–
4	2	2
5	–	–

dequeue 2 : check 4

Directed breadth-first search demo



adj[]

0	2	1
1	2	
2	4	
3	5	2
4	3	
5	0	

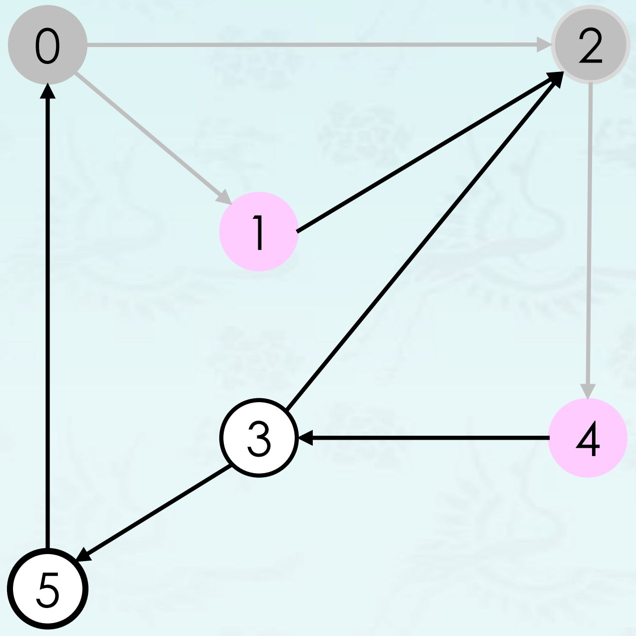
queue

4
1

v	parent[]	distTo[]
0	–	0
1	0	1
2	0	1
3	–	–
4	2	2
5	–	–

dequeue 2 : check 4

Directed breadth-first search demo



adj[]

0	2	1
1	2	
2	4	
3	5	2
4	3	
5	0	

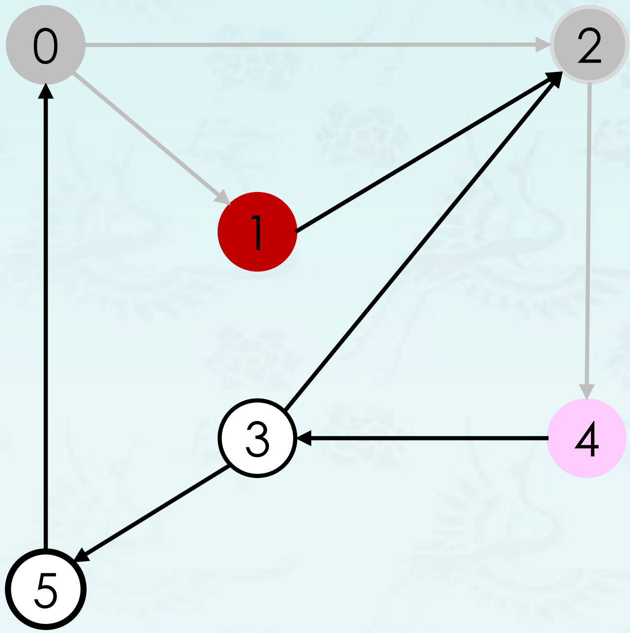
queue

4
1

v	parent[]	distTo[]
0	–	0
1	0	1
2	0	1
3	–	–
4	2	2
5	–	–

2 done

Directed breadth-first search demo



adj[]

0	2 → 1
1	2
2	4
3	5 → 2
4	3
5	0

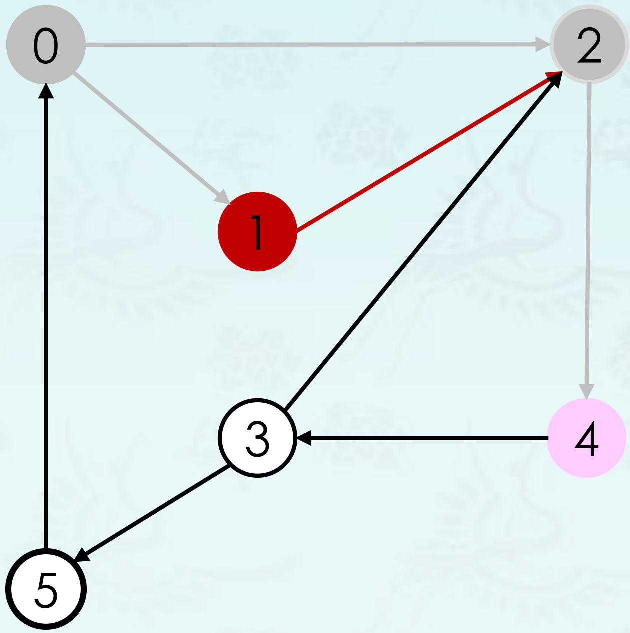
queue

4

v	parent[]	distTo[]
0	–	0
1	0	1
2	0	1
3	–	–
4	2	2
5	–	–

dequeue 1

Directed breadth-first search demo



adj[]

0	2	1
1	2	
2	4	
3	5	2
4	3	
5	0	

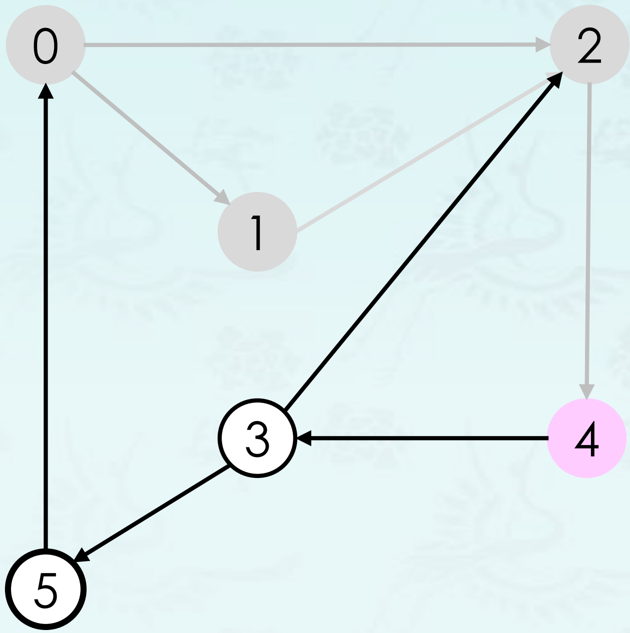
queue

4

v	parent[]	distTo[]
0	–	0
1	0	1
2	0	1
3	–	–
4	2	2
5	–	–

dequeue 1 : check 2

Directed breadth-first search demo



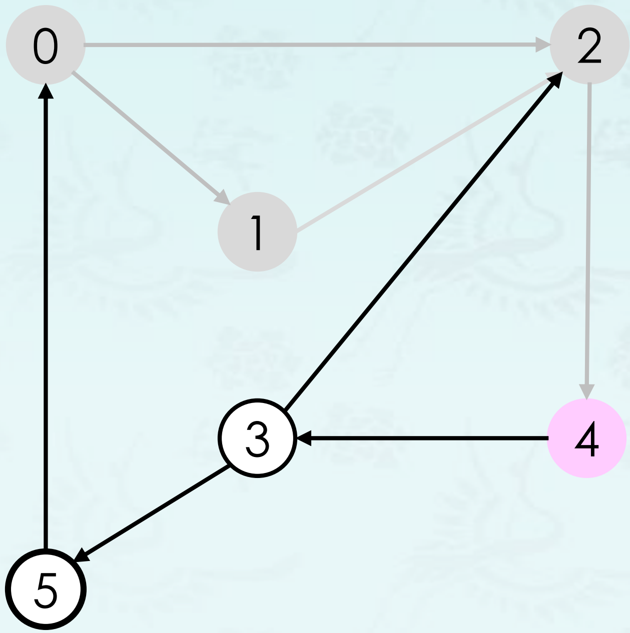
adj[]

0	2	1
1	2	
2	4	
3	5	2
4	3	
5	0	

queue	v	parent[]	distTo[]
4	0	–	0
	1	0	1
	2	0	1
	3	–	–
	4	2	2
	5	–	–

1 done

Directed breadth-first search demo



adj[]

0	2 → 1
1	2
2	4
3	5 → 2
4	3
5	0

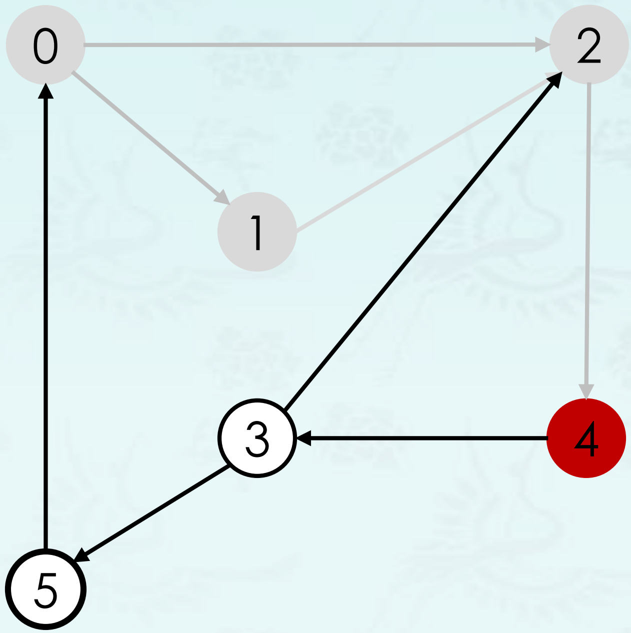
queue

4

v	parent[]	distTo[]
0	–	0
1	0	1
2	0	1
3	–	–
4	2	2
5	–	–

dequeue 4

Directed breadth-first search demo



adj[]

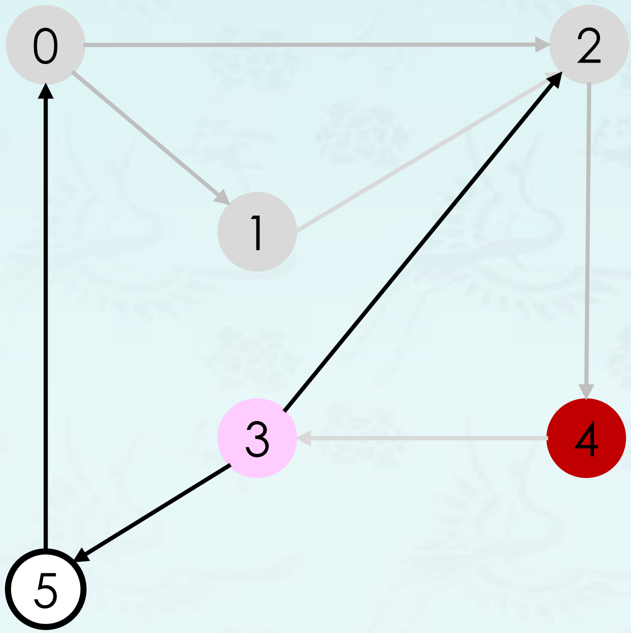
0	2	1
1	2	
2	4	
3	5	2
4	3	
5	0	

queue

v	parent[]	distTo[]
0	–	0
1	0	1
2	0	1
3	–	–
4	2	2
5	–	–

dequeue 4

Directed breadth-first search demo



adj[]

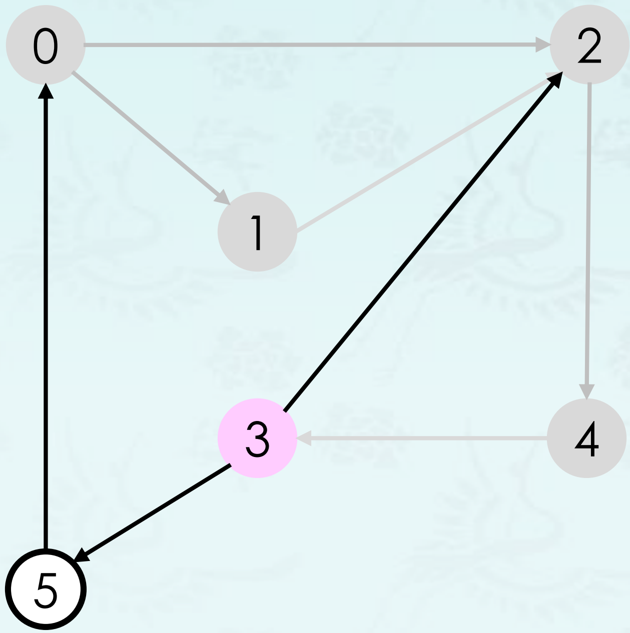
0	2 → 1
1	2
2	4
3	5 → 2
4	3
5	0

queue: 3

v	parent[]	distTo[]
0	–	0
1	0	1
2	0	1
3	4	3
4	2	2
5	–	–

dequeue 4 : check 3

Directed breadth-first search demo



adj[]

0	2	1
1	2	
2	4	
3	5	2
4	3	
5	0	

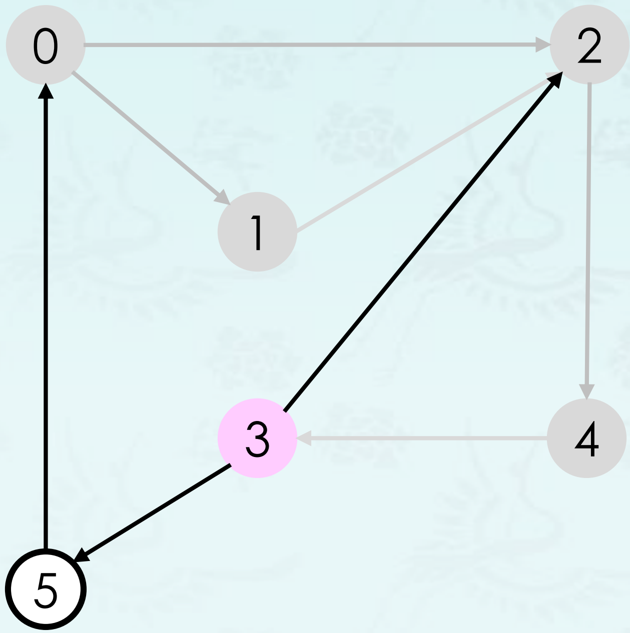
queue

3

v	parent[]	distTo[]
0	-	0
1	0	1
2	0	1
3	4	3
4	2	2
5	-	-

4 done

Directed breadth-first search demo



adj[]

0	2	1
1	2	
2	4	
3	5	2
4	3	
5	0	

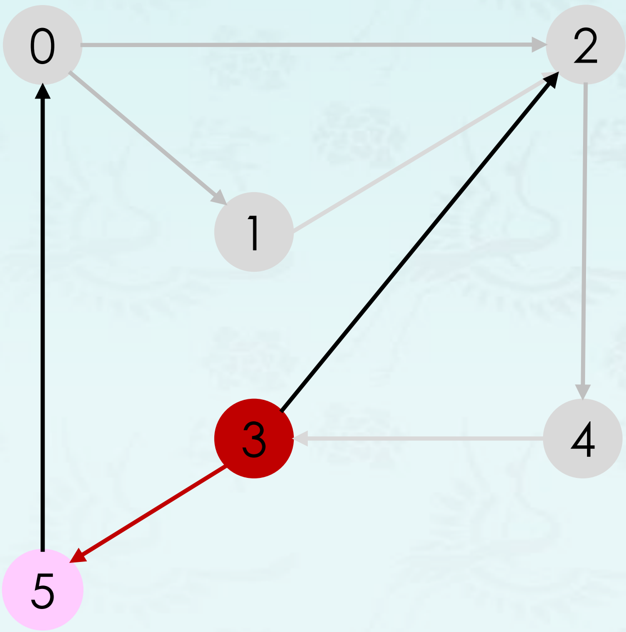
queue

3

v	parent[]	distTo[]
0	–	0
1	0	1
2	0	1
3	4	3
4	2	2
5	–	–

dequeue 3

Directed breadth-first search demo



adj[]

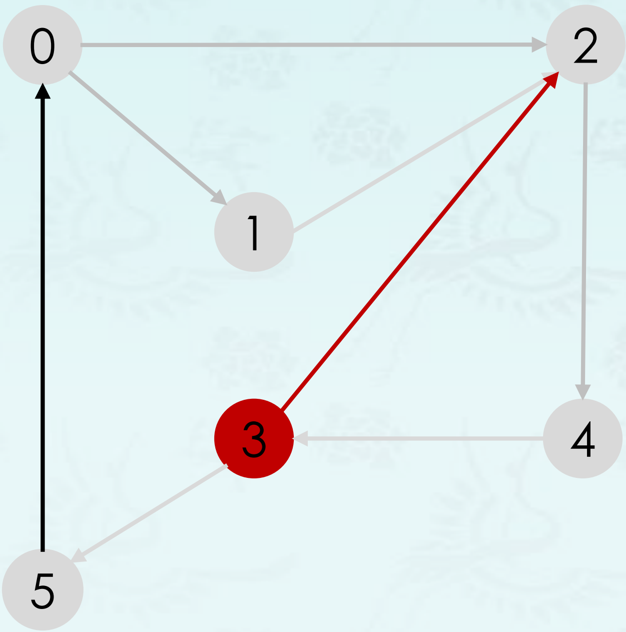
0	2	1
1	2	
2	4	
3	5	2
4	3	
5	0	

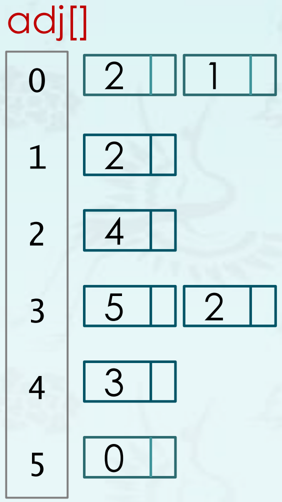
queue: 5

v	parent[]	distTo[]
0	-	0
1	0	1
2	0	1
3	4	3
4	2	2
5	3	4

dequeue 3 : check 5 and check 2

Directed breadth-first search demo





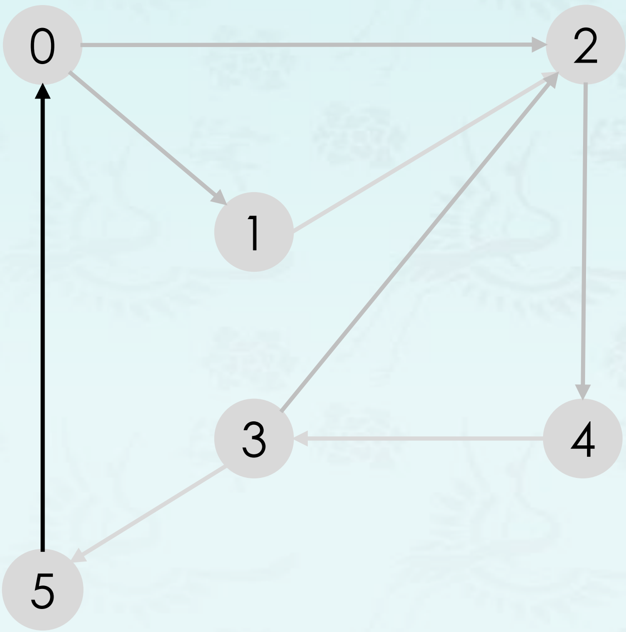
queue

5		

v	parent[]	distTo[]
0	-	0
1	0	1
2	0	1
3	4	3
4	2	2
5	3	4

dequeue 3 : check 5 and check 2

Directed breadth-first search demo



adj[]

0	2	1
1	2	
2	4	
3	5	2
4	3	
5	0	

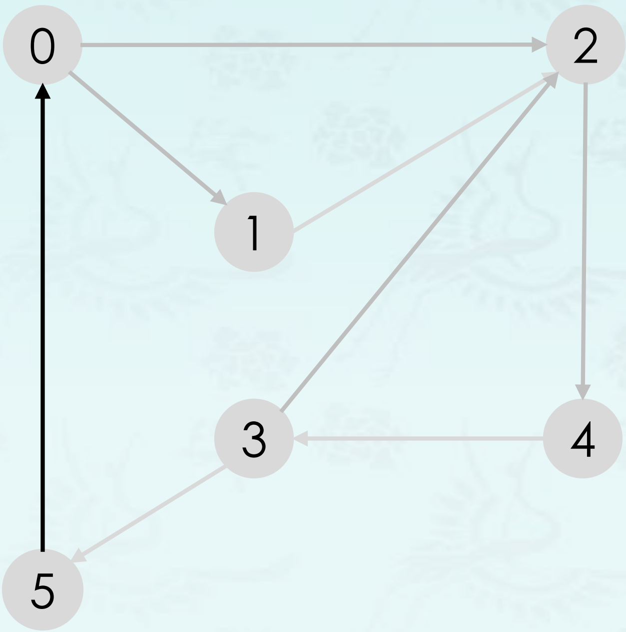
queue

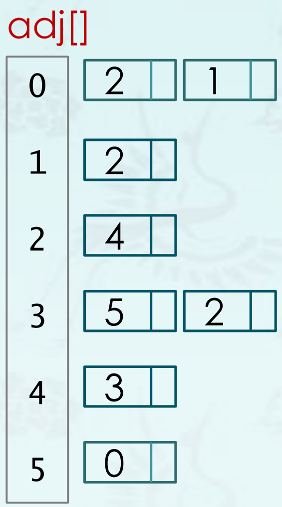
5

v	parent[]	distTo[]
0	-	0
1	0	1
2	0	1
3	4	3
4	2	2
5	3	4

dequeue 3 : check 5 and check 2

Directed breadth-first search demo



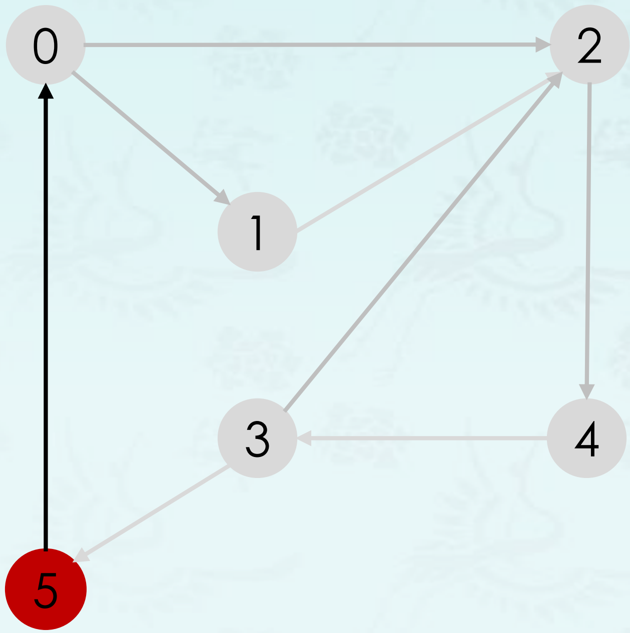


queue: 5

v	parent[]	distTo[]
0	-	0
1	0	1
2	0	1
3	4	3
4	2	2
5	3	4

dequeue 5 :

Directed breadth-first search demo



adj[]

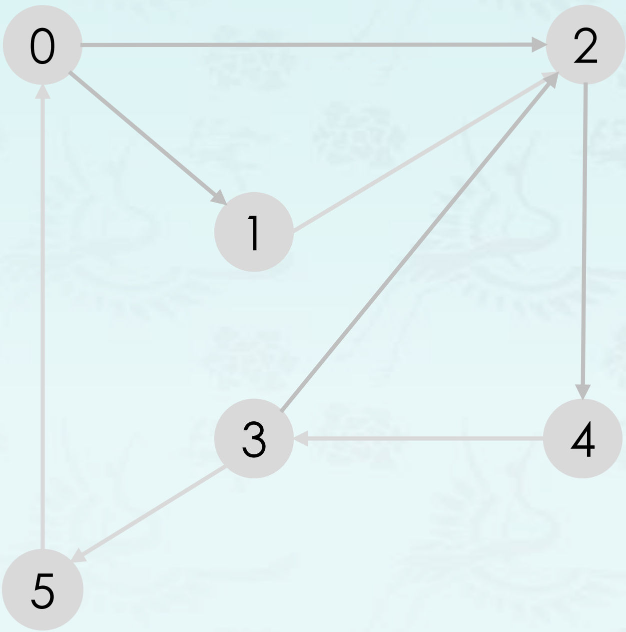
0	2	1
1	2	
2	4	
3	5	2
4	3	
5	0	

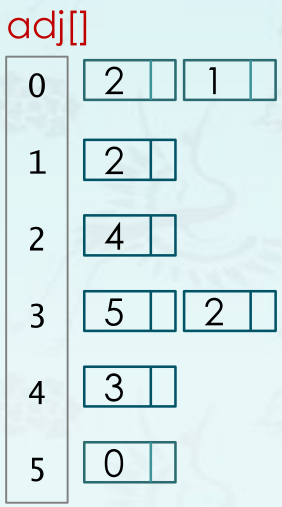
queue

v	parent[]	distTo[]
0	-	0
1	0	1
2	0	1
3	4	3
4	2	2
5	3	4

dequeue 5 : check 0

Directed breadth-first search demo



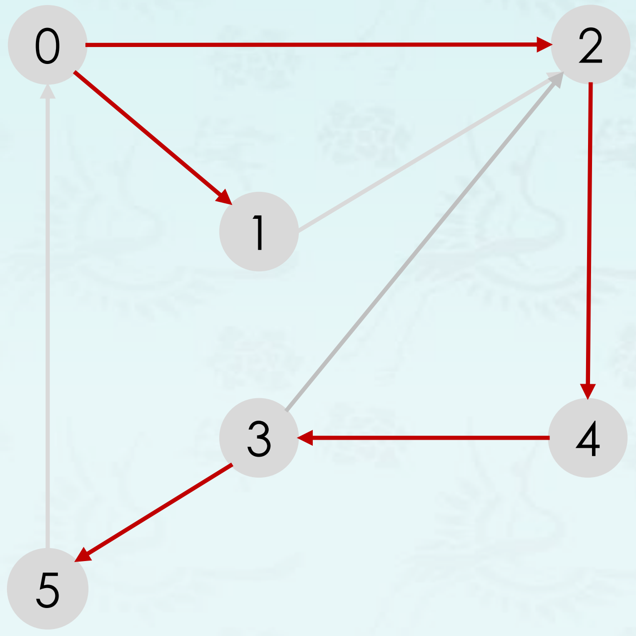


queue

v	parent[]	distTo[]
0	-	0
1	0	1
2	0	1
3	4	3
4	2	2
5	3	4

5 done

Directed breadth-first search demo



queue	v	parent[]	distTo[]
	0	–	0
	1	0	1
	2	0	1
	3	4	3
	4	2	2
	5	3	4

done

Graph

- *Graph*
  - *Introduction*
  - *Adjacency list*
  - *DFS, BFS*
  - *Challenges*
- ***Digraph – Directed Graphs***
  - *digraph – DFS, BFS*
  - *Applications – crawl web, topological sort*
- *Minimum Spanning Tree(MST)*

Major references:

1. Fundamentals of Data Structures by Horowitz, Sahni, Anderson-Freed,
2. Algorithms 4th edition - Part 1 & Part 2 by Robert Sedgewick and Kevin Wayne
3. Wikipedia and many resources available from internet