Graphs: Directed Graphs

Neerja Mhaskar

Dept. of Computing and Software, McMaster University, Canada

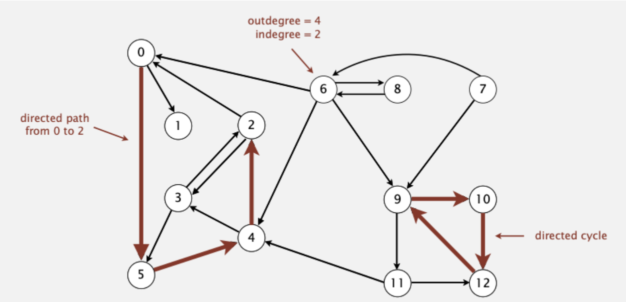
Acknowledgments: Material mainly based on the textbook Algorithms by Robert Sedgewick and Kevin Wayne (Chapters 4.2) and Prof. Janicki's course slides

Directed Graphs

In directed graphs (or digraphs), edges are one-way: the pair of vertices that defines each edge is an ordered pair that specifies a one-way adjacency.

The indegree of a vertex in a digraph is the number of directed edges that point to that vertex.

The outdegree of a vertex in a digraph is the number of directed edges that emanate from that vertex.



Directed Graph API

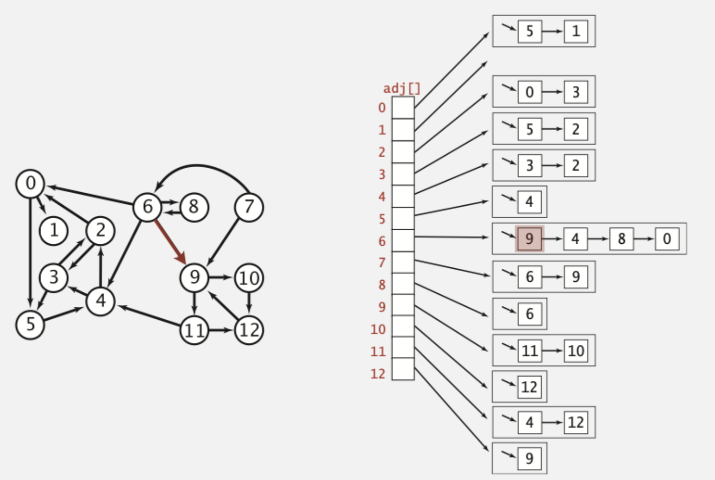
Almost identical to Graph API [has only an addition method 'reverse']

`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*

Directed representation: adjacency lists

Maintain vertex-indexed array of lists.



Adjacency-list graph representation: Java implementation for Digraph

```java
public class Graph
{
   private final int V;
   private final Bag<Integer>[] adj;          // adjacency lists

   public Graph(int V)                        // create empty graph with V vertices
   {
      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 edge v–w
   {
      adj[v].add(w);
      adj[w].add(v);
   }

   public Iterable<Integer> adj(int v)        // iterator for vertices adjacent to v
   {  return adj[v];  }
}
```

```java
public class Digraph
{
   private final int V;
   private final Bag<Integer>[] adj;          // adjacency lists

   public Digraph(int V)                      // create empty digraph with V vertices
   {
      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 edge v→w
   {
      adj[v].add(w);

   }

   public Iterable<Integer> adj(int v)        // iterator for vertices pointing from v
   {  return adj[v];  }
}
```

Searching in Digraph

Depth-first search in digraphs: Same method as for undirected graphs.

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

Code:

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

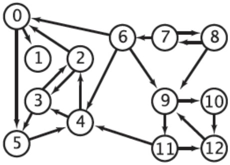
Figure 1: DFS for undirected graphs

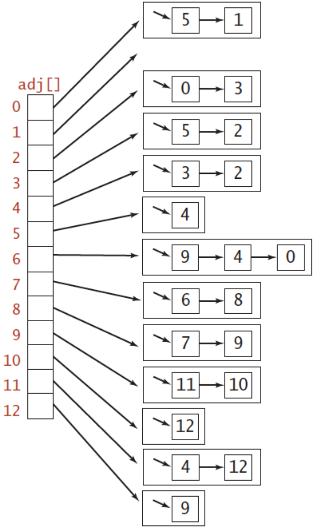
```
private void dfs(Digraph G, int v)
{
   marked[v] = true;
   for (int w : G.adj(v))
      if (!marked[w]) dfs(G, w);
}
```

Figure 2: DFS for directed graphs

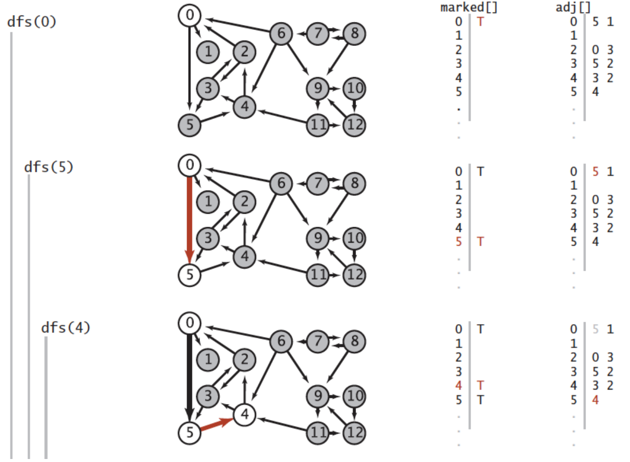
See Demo: `https://algs4.cs.princeton.edu/lectures/`

Searching in Digraph: Example

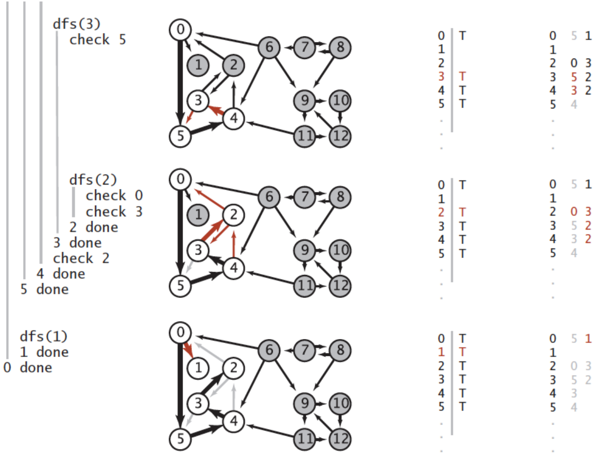




Searching in Digraph: DFS I



Searching in Digraph: DFS II



Searching in Digraph - BFS

Breadth-first search in digraphs: Same method as for undirected graphs.

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

Proposition. BFS computes shortest paths (fewest number of edges) from s to all other vertices in a digraph in time proportional to $|V| + |E|$.

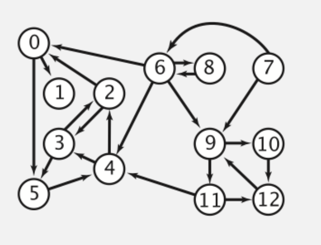
See Demo: `https://algs4.cs.princeton.edu/lectures/`

Multiple-source shortest paths

Multiple-source shortest paths. Given a digraph and a set of source vertices S, find shortest path from any vertex in the set to each other vertex.

Example. $S = \{1, 7, 10\}$

- Shortest path to 4 is $7 \to 6 \to 4$
- Shortest path to 5 is $7 \to 6 \to 0 \to 5$
- Shortest path to 12 is $10 \to 12$



How to implement multi-source shortest paths algorithm?

- Use BFS, but initialize by enqueuing all source vertices.

Precedence-constrained scheduling

Precedence-constrained scheduling problem. Given a set of jobs to be completed, with precedence constraints that specify that certain jobs have to be completed before certain other jobs are begun, how can we schedule the jobs such that they are all completed while still respecting the constraints?

For any such problem, a digraph model is useful, where

vertex = task and edge = precedence constraint.

However, if job x must be completed before job y, job y before job z, and job z before job x, clearly these constraints cannot be satisfied; that is, a digraph with cycles will not have a feasible solution.

A directed acyclic graph (DAG) is a digraph with no directed cycles.

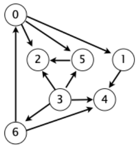
Topological Sort

Topological sort. Given a digraph, put the vertices in order such that all its directed edges point from a vertex earlier in the order to a vertex later in the order (or report that doing so is not possible).

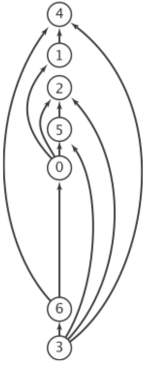
Alternatively, topological sort is a linear ordering of its vertices such that for every directed edge u, v from vertex u to vertex v, u comes before v in the ordering.

0→5	0→2
0→1	3→6
3→5	3→4
5→2	6→4
6→0	3→2
1→4	

directed edges



DAG



topological order

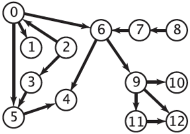
Topological Sort: using DFS

Idea: Depth-first search visits each vertex exactly once. If we save the vertex given as argument to the recursive dfs() in a data structure, then iterate through that data structure (as queue or stack), we see all the graph vertices, in an order determined by the nature of the data structure and by whether we do the save before or after the recursive calls.

Three vertex orderings are of interest in typical applications:

- Preorder: Put the vertex on a queue before the recursive calls.
- Postorder: Put the vertex on a queue after the recursive calls.
- Reverse postorder: Put the vertex on a stack after the recursive calls.

Vertex ordering partial example



preorder is order of `dfs()` *calls*

postorder is order in which vertices are `done`

	pre (*queue*)	post (*queue*)	reversePost (*stack*)
`dfs(0)`	0		
`dfs(5)`	0 5		
`dfs(4)`	0 5 4		
`4 done`		4	4
`5 done`		4 5	5 4
`dfs(1)`	0 5 4 1		
`1 done`		4 5 1	1 5 4
`dfs(6)`	0 5 4 1 6		
`dfs(9)`	0 5 4 1 6 9		
`dfs(11)`	0 5 4 1 6 9 11		
`dfs(12)`	0 5 4 1 6 9 11 12		
`12 done`		4 5 1 12	12 1 5 4
`11 done`		4 5 1 12 11	11 12 1 5 4
`dfs(10)`	0 5 4 1 6 9 11 12 10		
`10 done`		4 5 1 12 11 10	10 11 12 1 5 4

Depth First Search Order: Reverse Postorder

```
public class DepthFirstOrder
{
   private boolean[] marked;
   private Stack<Integer> reversePostorder;

   public DepthFirstOrder(Digraph G)
   {
      reversePostorder = new Stack<Integer>();
      marked = new boolean[G.V()];
      for (int v = 0; v < G.V(); v++)
         if (!marked[v]) dfs(G, v);
   }

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

   public Iterable<Integer> reversePostorder()
   {  return reversePostorder;  }
}
```

returns all vertices in "reverse DFS postorder"

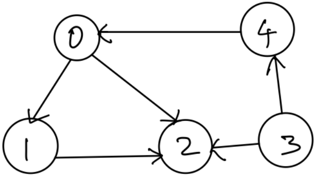
Topological Sort: using DFS

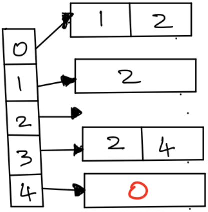
Reverse postorder in a DAG is a topological sort; that is, reverse postorder gives the order of vertices such that all its directed edges point from a vertex earlier in the order to a vertex later in the order.

```
public Topological(Digraph G)
{
   DirectedCycle cyclefinder = new DirectedCycle(G);
   if (!cyclefinder.hasCycle())
   {
      DepthFirstOrder dfs = new DepthFirstOrder(G);
      order = dfs.reversePost();
   }
}
```

See Demo: `https://algs4.cs.princeton.edu/lectures/demo/42DemoTopologicalSort.mov`

Topological Sort example - I



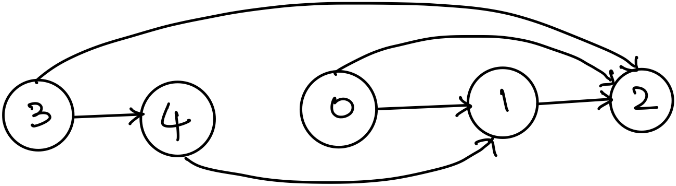


Topological Sort example - II

	Preorder	Post order	Rev. Post order
dfs 0	0		
dfs 1	0 1		
dfs 2	0 1 2		
done 2		2	2
done 1		2 1	1 2
check 2			
done 0		2, 1 0	0 1 2
dfs 3	0 1 2 3		
check 2			
dfs 4	0 1 2 3 4		
check 0	—	—	—
done 4		2 1 0 4	4 0 1 2
done 3		2 1 0 4 3	3 4 0 1 2

Topological Sort example - III

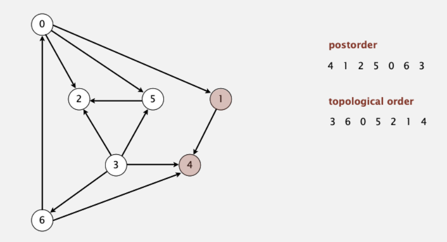
Topological Sort:



Topological sort in a DAG: intuition

Why does topological sort (T-sort) algorithm work?

- Last vertex in reverse postorder (i.e., first vertex in post order) has outdegree 0.
- Second last vertex in reverse postorder (i.e., send vertex in post order) can only point to first vertex.
- ....



Proposition. A digraph has a topological order if and only if it is a DAG.

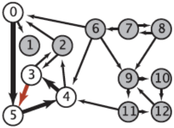
Topological sort in a DAG: running time

Proposition. A digraph has a topological order if and only if it is a DAG.

Proposition. With DFS, we can T-sort a DAG in time proportional to $|V| + |E|$.

Directed cycle detection

- The recursive call stack maintained by the system during $dfs()$ represents the "current" directed path under consideration.
- By maintaining a vertex indexed boolean array $onStack$, we can keep track of the elements in this call stack.
- If we ever find a directed edge $v \to w$ to a vertex w that is on that stack, we have found a cycle, since the stack is evidence of a directed path from w to v, and the edge $v \to w$ completes the cycle.
- The absence of any such back edges implies that the graph is acyclic.



	marked[]	edgeTo[]	onStack[]
	0 1 2 3 4 5 ...	0 1 2 3 4 5 ...	0 1 2 3 4 5 ...
dfs(0)			
dfs(5)	1 0 0 0 0 0	- - - - - 0	1 0 0 0 0 0
dfs(4)	1 0 0 0 0 1	- - - - 5 0	1 0 0 0 0 1
dfs(3)	1 0 0 0 1 1	- - - 4 5 0	1 0 0 0 1 1
check 5	1 0 0 1 1 1	- - - 4 5 0	1 0 0 1 1 1

Finding a directed cycle in a digraph

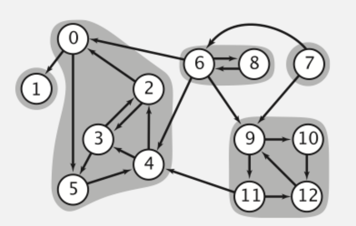
Strongly-connected components

Def. Vertices v and w are strongly connected if there is both a directed path from v to w and a directed path from w to v.

Key property. Strong connectivity is an equivalence relation:

- v is strongly connected to v.
- If v is strongly connected to w, then w is strongly connected to v.
- If v is strongly connected to w and w to x, then v is s.c to x.

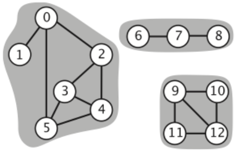
Def. A strong component is a maximal subset of strongly-connected vertices.



5 strongly-connected components

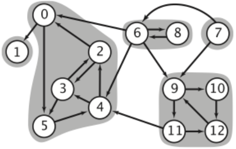
Connected components vs. strongly-connected components

v and w are connected if there is a path between v and w



3 connected components

v and w are strongly connected if there is both a directed path from v to w and a directed path from w to v



5 strongly-connected components

connected component id (easy to compute with DFS)

	0	1	2	3	4	5	6	7	8	9	10	11	12
id[]	0	0	0	0	0	0	1	1	1	2	2	2	2

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

constant-time client connectivity query

strongly-connected component id (how to compute?)

	0	1	2	3	4	5	6	7	8	9	10	11	12
id[]	1	0	1	1	1	1	3	4	3	2	2	2	2

```
public boolean stronglyConnected(int v, int w)
{  return id[v] == id[w];  }
```

constant-time client strong-connectivity query

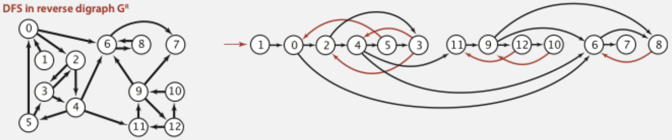
Kosaraju-Sharir algorithm

Simple (but mysterious) algorithm for computing strong components in a graph G

- Run DFS on G^R (reverse graph of G with edges reversed) and compute the reverse postorder.
- Run DFS on G, considering vertices in reverse postorder.

Kosaraju-Sharir algorithm I

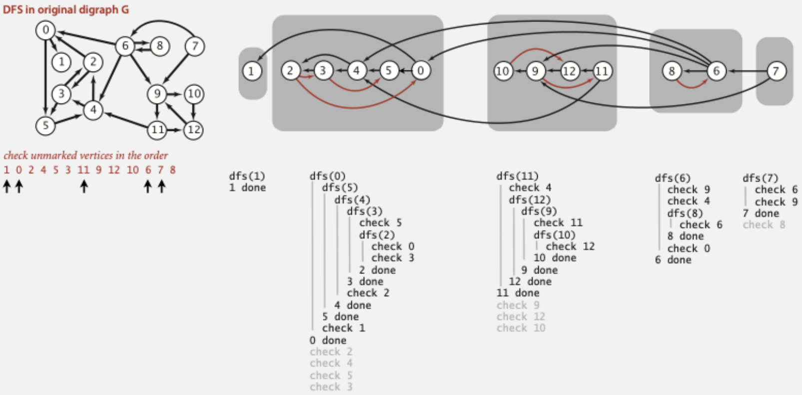
- **Phase 1: run DFS on G^R to compute reverse postorder.**
- Phase 2: run DFS on G, considering vertices in order given by first DFS.




```
dfs(0)
| dfs(6)
| | dfs(8)
| | | check 6
| | 8 done
| | dfs(7)
| | 7 done
| 6 done
| dfs(2)
| | dfs(4)
| | | dfs(11)
| | | | dfs(9)
| | | | | dfs(12)
| | | | | | check 11
| | | | | | dfs(10)
| | | | | | | check 9
| | | | | | 10 done
| | | | | 12 done
| | | | | check 7
| | | | | check 6
...
```

Kosaraju-Sharir algorithm II

- Phase 1: run DFS on G^R to compute reverse postorder.
- **Phase 2: run DFS on G, considering vertices in order given by first DFS.**



Connected components in an undirected graph (with DFS) and strong components in a digraph (with two DFSs)

```
public class CC
{
   private boolean marked[];
   private int[] id;
   private int count;

   public CC(Graph G)
   {
      marked = new boolean[G.V()];
      id = new int[G.V()];

      for (int v = 0; v < G.V(); v++)
      {
         if (!marked[v])
         {
            dfs(G, v);
            count++;
         }
      }
   }

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

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

```
public class KosarajuSharirSCC
{
   private boolean marked[];
   private int[] id;
   private int count;

   public KosarajuSharirSCC(Digraph G)
   {
      marked = new boolean[G.V()];
      id = new int[G.V()];
      DepthFirstOrder dfs = new DepthFirstOrder(G.reverse());
      for (int v : dfs.reversePostorder())
      {
         if (!marked[v])
         {
            dfs(G, v);
            count++;
         }
      }
   }

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

   public boolean stronglyConnected(int v, int w)
   {  return id[v] == id[w];  }
}
```

Kosaraju-Sharir algorithm: running time complexity

Proposition. Kosaraju-Sharir algorithm computes the strong components of digraph in time proportional to $|V| + |E|$.