

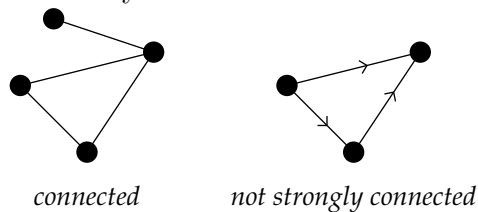
Graphs

21.1 Graphs

A graph has two parts: *vertices* (one vertex) also called *nodes*. An *undirected graph* has undirected *edges*. Two vertices joined by edge are *neighbors*. A *directed graph* has directed *edges/arcs*; each arc goes from *in-neighbor* to *out-neighbor*. Examples include:

- city map
- circuit diagram
- chemical molecule
- family tree

A *path* is sequence of vertices with successive vertices joined by edge/arc. A *cycle* is a sequence of vertices ending up where started such that successive vertices are joined by edge/arc. A graph is *connected* (a directed graph is *strongly connected*) if there is a path from every vertex to every other vertex.



21.2 Graph Representation

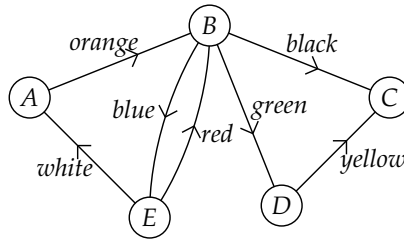
ADJACENCY MATRIX

- 1) container of vertices, numbered
- 2) array where each entry has info about the corresponding edge

ADJACENCY LIST

- 1) container of vertices
- 2) for each vertex a list of out-neighbors

An example directed graph (with labeled vertices and arcs):



Adjacency array:

	A	B	C	D	E
A	—	<i>orange</i>	—	—	—
B	—	—	<i>black</i>	<i>green</i>	<i>blue</i>
C	—	—	—	—	—
D	—	—	<i>yellow</i>	—	—
E	<i>white</i>	<i>red</i>	—	—	—

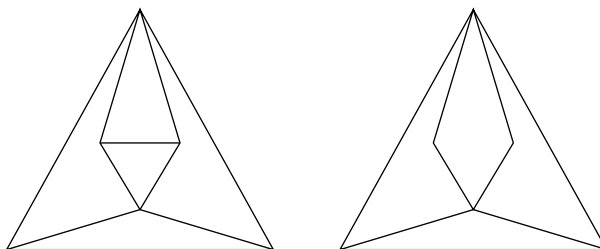
Adjacency list:

A	<i>orange, B</i>		
B	<i>black, C</i>	<i>green, D</i>	<i>blue, E</i>
C			
D	<i>yellow, C</i>		
E	<i>red, B</i>	<i>white, A</i>	

The advantage of the adjacency matrix is that determining $\text{isAdjacent}(u,v)$ is $O(1)$. The disadvantage of adjacency matrix is that it can be space-inefficient, and enumerating outNeighbors etc. can be slow in sparse graphs.

21.3 Aside

Practice. Draw each of the following without lifting your pen.



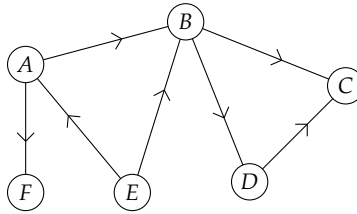
21.4 Topological Sort

A **DAG**, directed acyclic graph, is a directed graph without directed cycles. The classic example is scheduling constraints between tasks of a project.

A **topological ordering** is an ordering of the vertices such that every arc goes from lower number to higher number vertex

Fact: there is a topological ordering iff directed graph is acyclic

EXAMPLE. In the following DAG, one topological ordering is: E A F B D C



Here is an algorithm for topological sort. A **source** is a vertex with no in-arcs.

Algorithm: TopologicalOrdering()
Repeatedly
Find source, output and remove

For efficiency:

1. maintain counter **indegree** at each vertex **v**, and decrement every time the current source has arc to **v** (no deletions).
2. every time a decrement creates a source, add to container of sources.

21.5 Sample Code

Here is an abstract base class **Dag**, an implementation of topological sort for that class, an adjacency-list implementation of the class, and a primitive test program. (The class **IntQueue** provides a queue of ints.)

```
// Dag.h - wdg - 2008
#ifndef DAG_H
#define DAG_H

//DAG is Directed Acyclic Graph
#include<string>
using namespace std;
```

```

class DAG
{
    public:
        virtual void addEdge (int u, int v) = 0;
        virtual void removeEdge(int u, int v) = 0;
        virtual bool isAdjacent (int u, int v) const = 0;
        virtual int outDegree(int u) const = 0;
        virtual int inDegree(int u) const = 0;
        virtual int numberEdges() const = 0;
        virtual int numberVertices() const = 0;
        virtual string toString() const = 0;
};

#endif

```

```

// GraphAlgorithms.cpp - wdg - 2008
// Using Java style!
#include "Dag.h"
#include "IntQueue.h"

#include <iostream>
using namespace std;

class GraphAlgorithms
{
    public:
        static string topologicalSort(const DAG & dag)
        {
            int N = dag.numberVertices();
            int inDeg[N];
            for(int i=0; i<N; i++)
                inDeg[i] = dag.inDegree(i);

            IntQueue Q(N);
            for(int j=0; j<N; j++)
                if( inDeg[j]==0 )
                    Q.enqueue( j );

            string S;
            while ( !Q.isEmpty() ) {

```

```

        int curr = Q.dequeue();
        S += curr+'0';
        S += " ";
        for(int j=0; j<N; j++)
            if( dag.isAdjacent(curr,j) ) {
                inDeg[j]--;
                if( inDeg[j]==0 )
                    Q.enqueue(j);
            }
    }

    return S;
}
};

```

```

// AListDAG.h - wdg - 2008
// Adjacency list implementation of directed graph
#include "Dag.h"
#include <cstdlib>
using namespace std;

class GNode
{
    int neighbor;
    GNode *next;
    GNode( int nb, GNode *nx ) : neighbor(nb), next(nx) {
    }
    friend class AListDAG;
};

class AListDAG : public DAG
{
private:
    int N;
    GNode **outNeigh;    // used as array of pointers

public:
    // Constructor
    AListDAG(int vertices) : N(vertices), outNeigh(new GNode *[vertices])

```

```

{
    for(int i=0; i<N; i++)
        outNeigh[i]=NULL;
}

virtual ~AListDAG( )
{
    for(int i=0; i<N; i++)
        while( outNeigh[i] ) {
            GNode * hold = outNeigh[i]->next;
            delete outNeigh[i];
            outNeigh[i]=hold;
        }
    delete [] outNeigh;
}

void addEdge(int u, int v)
{
    outNeigh[u] = new GNode(v, outNeigh[u]);
}

void removeEdge(int u, int v)
{
    if( outNeigh[u] && v==outNeigh[u]->neighbor )
        outNeigh[u] = outNeigh[u]->next;
    else {
        GNode *curr = outNeigh[u];
        while ( curr->next && v!=curr->next->neighbor )
            curr = curr->next;
        if( curr->next && v==curr->next->neighbor )
            curr->next = curr->next->next;
    }
}

int numberVertices( ) const
{
    return N;
}

```

```

int inDegree(int v) const
{
    int count = 0 ;
    for(int i=0; i<N; i++)
        for( GNode *curr = outNeigh[i]; curr; curr=curr->next )
            if( v==curr->neighbor )
                count++;
    return count;
}

```

```

int outDegree(int v) const
{
    int count = 0 ;
    for( GNode *curr = outNeigh[v]; curr; curr=curr->next )
        count++;
    return count;
}

```

```

bool isAdjacent (int u, int v) const
{
    for( GNode *curr = outNeigh[u]; curr; curr=curr->next )
        if( v==curr->neighbor )
            return true;
    return false;
}

```

```

int numberEdges ( ) const
{
    int count = 0;
    for( int i=0; i<N; i++)
        count += outDegree(i);
    return count;
}

```

```

string toString() const
{
    string S;
    for(int i=0; i<N; i++) {
        S += i+'0';
    }
}

```

```

        S += ": ";
        for( GNode *curr = outNeigh[i]; curr; curr=curr->next) {
            S += curr->neighbor+'0';
            S += "," ;
        }
        S+= "\n";
    }
    return S;
}

};

```

```

// SortTest.cpp - wdg - 2008
// one exercising of topological sort
#include "AListDAG.cpp"
#include "GraphAlgorithms.cpp"
#include<iostream>
using namespace std;

int main( )
{
    AListDAG d(10);
    d.addEdge(4,0); d.addEdge(1,8);
    d.addEdge(4,1); d.addEdge(4,9);
    d.addEdge(2,8); d.addEdge(6,7);
    d.addEdge(6,4); d.addEdge(5,4);
    d.addEdge(3,4); d.addEdge(1,0);
    d.addEdge(0,8); d.addEdge(7,8);
    d.addEdge(4,8); d.addEdge(9,1);
    d.addEdge(9,2); d.addEdge(5,9);
    d.addEdge(3,6); d.addEdge(1,2);
    cout << d.toString() << endl;
    cout << GraphAlgorithms::topologicalSort(d) << endl;
    cout << "Should be: 3 5 6 4 7 9 1 0 2 8" << endl;
    return 0;
}

```