Introduction to Graphs

Fundamental Data Structures and Algorithms

Prof. S.G.Gollagi

Announcements

- HW 6 is about to be released.

Start asap.

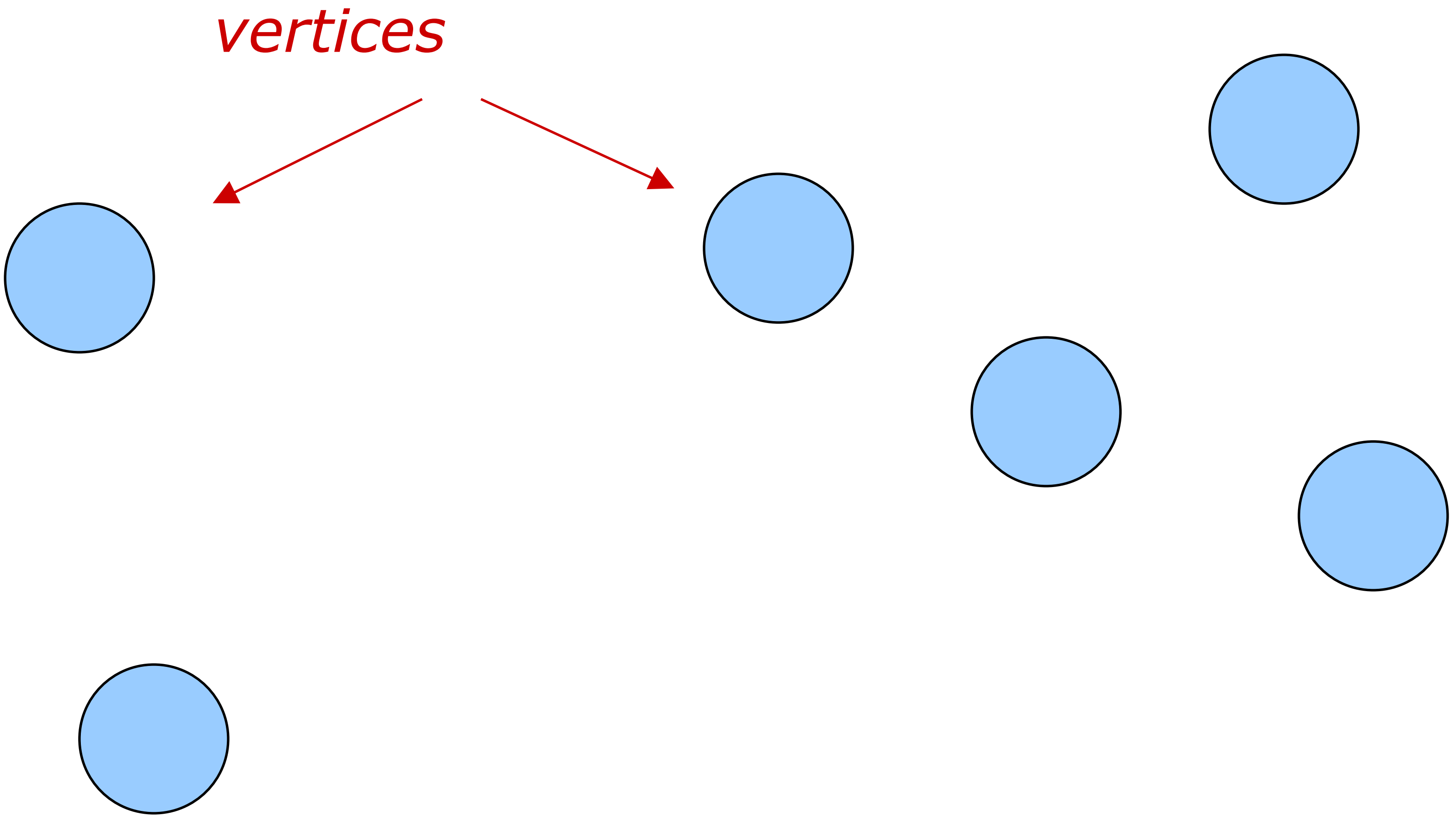
- Reading: Chapter 14 in MAW.

There are quite few definitions there,
make sure you understand the ideas and
concepts.

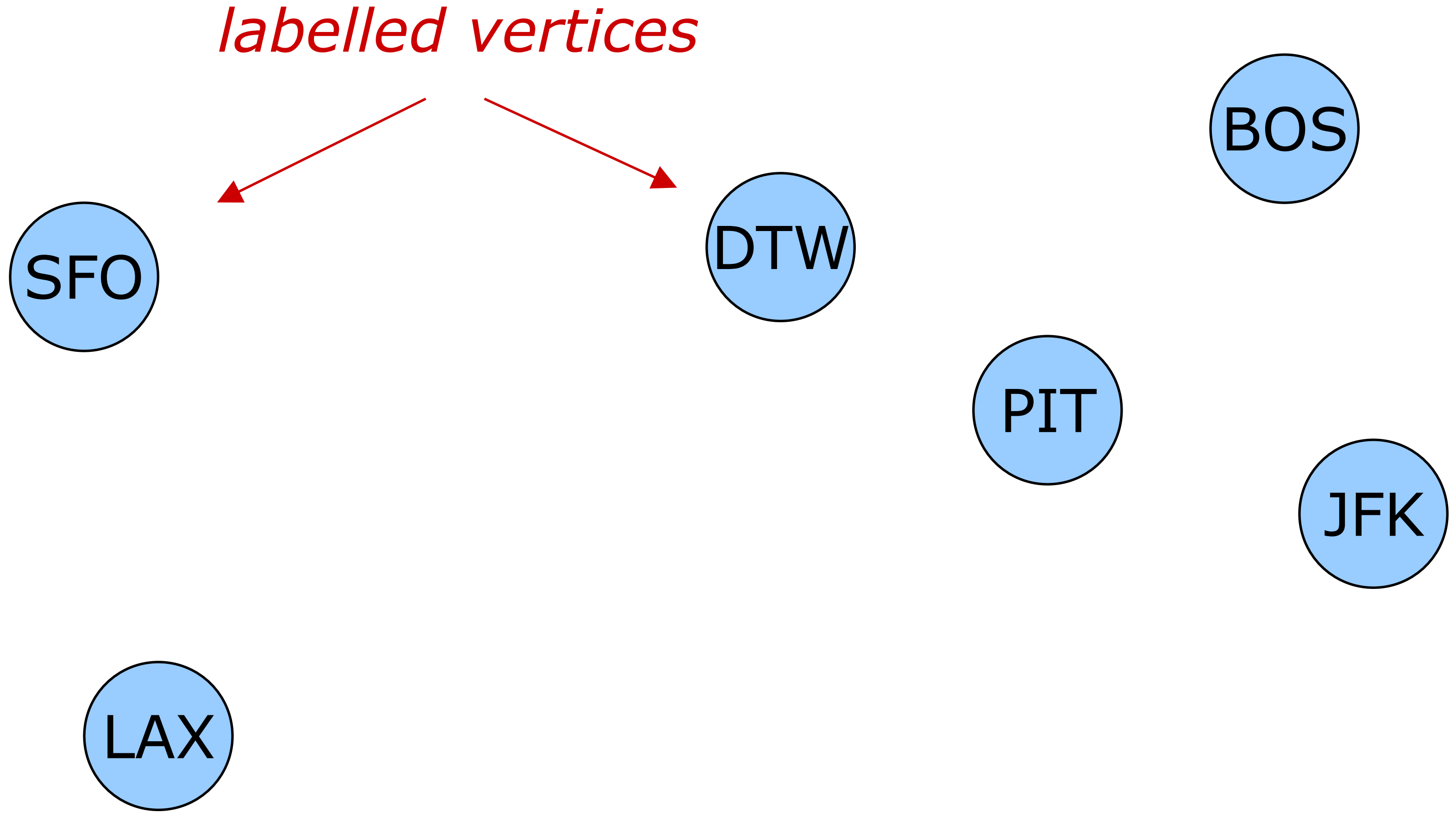
- Extra credit: Write a one-page essay about
  roving eyeballs.

Introduction to Graphs

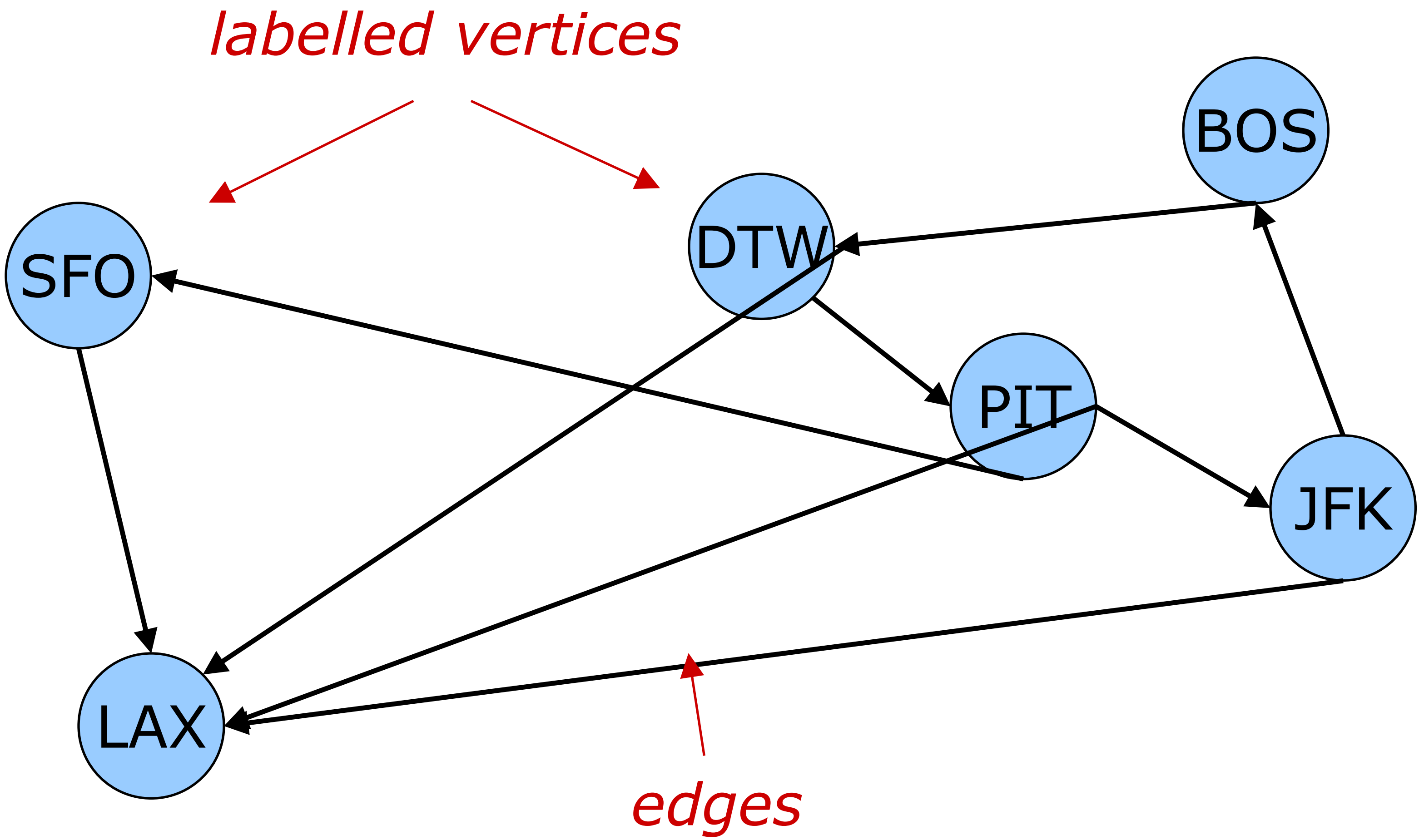
Graphs — an overview



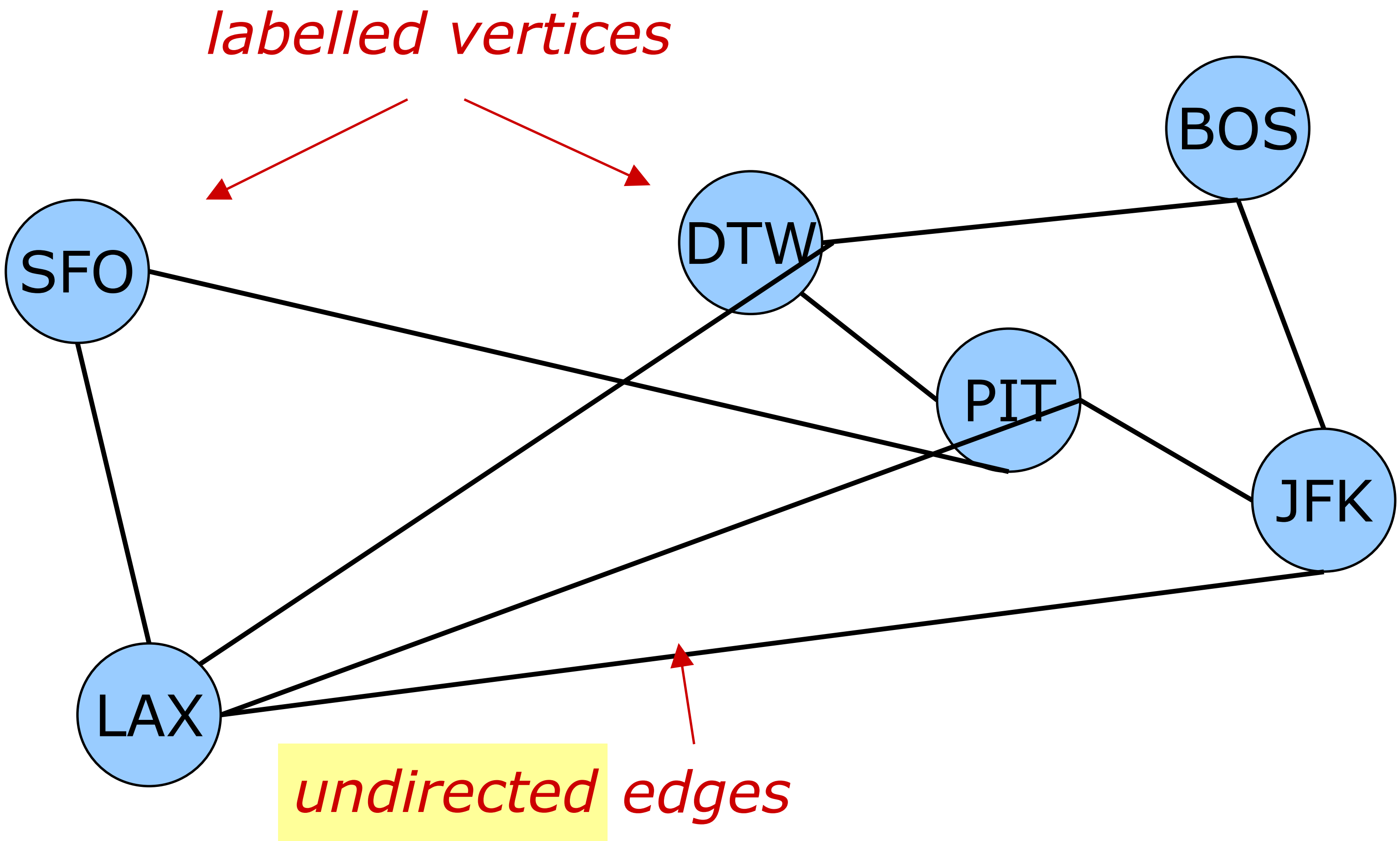
Graphs — an overview



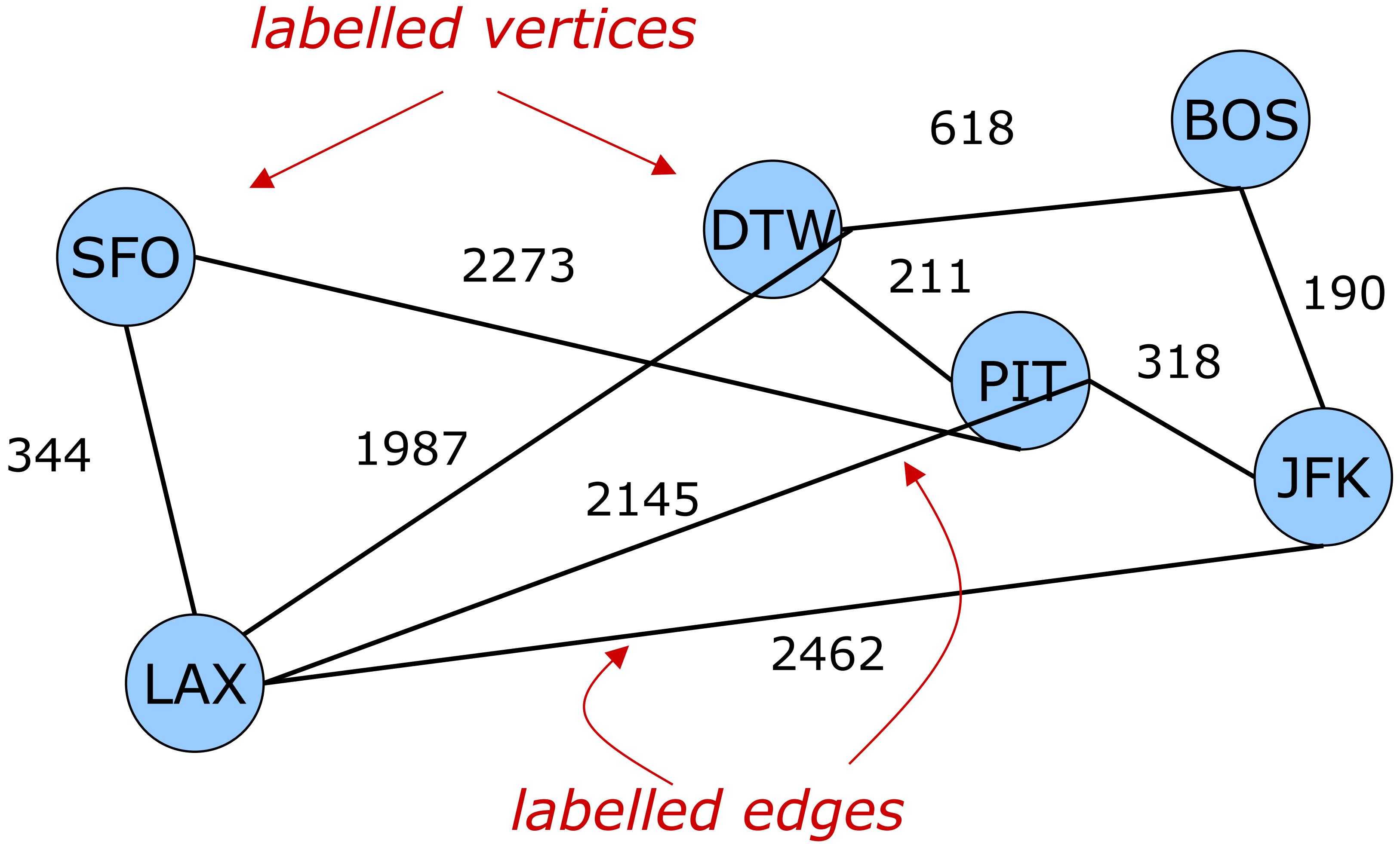
Graphs — an overview



Graphs — an overview



Graphs — an overview



Terminology

- vertices (aka nodes, points)

- edges (aka arcs, lines)

directed or undirected (digraphs and ugraphs)
multiple or single
loops

- vertex labels

- edge labels

G = (V,E) or G = (V,E,lab)

Edges

- directed (x,y) or just xy

x is the source and y the target of the edge

- undirected $\{x,y\}$ or just xy

Note that $\{x,x\}$ means: undirected loop at x.

- edge is incident upon vertex x and y

Degrees

- directed

    out-degree of x: the number of edges (x,y)

    in-degree of x: the number of edges (y,x)

degree: sum of in-degree and out-degree

- undirected

    degree of x: the number of edges {x,y}

Degrees are often important in determining the running time of an algorithm.

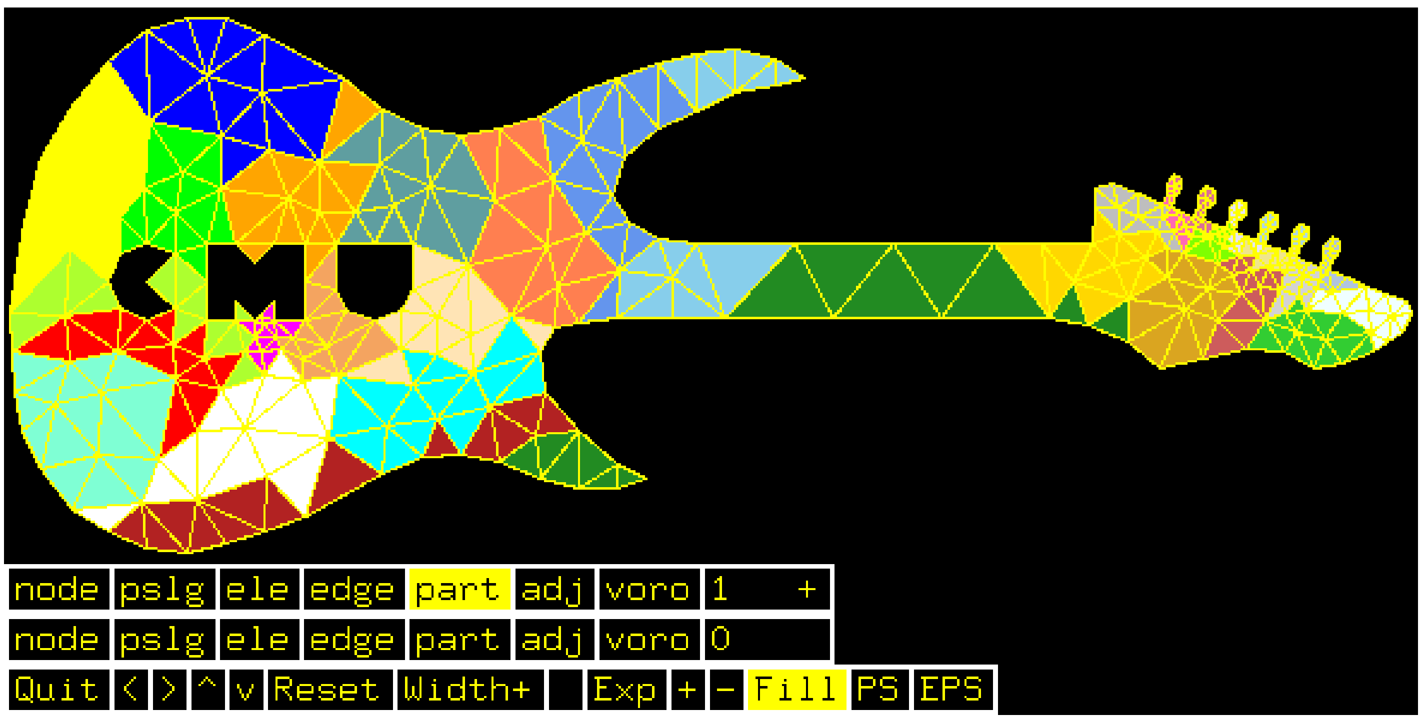
Graphs are Everywhere

Examples

- Roadmaps
- Communication networks
- WWW
- Electrical circuits
- Task schedules

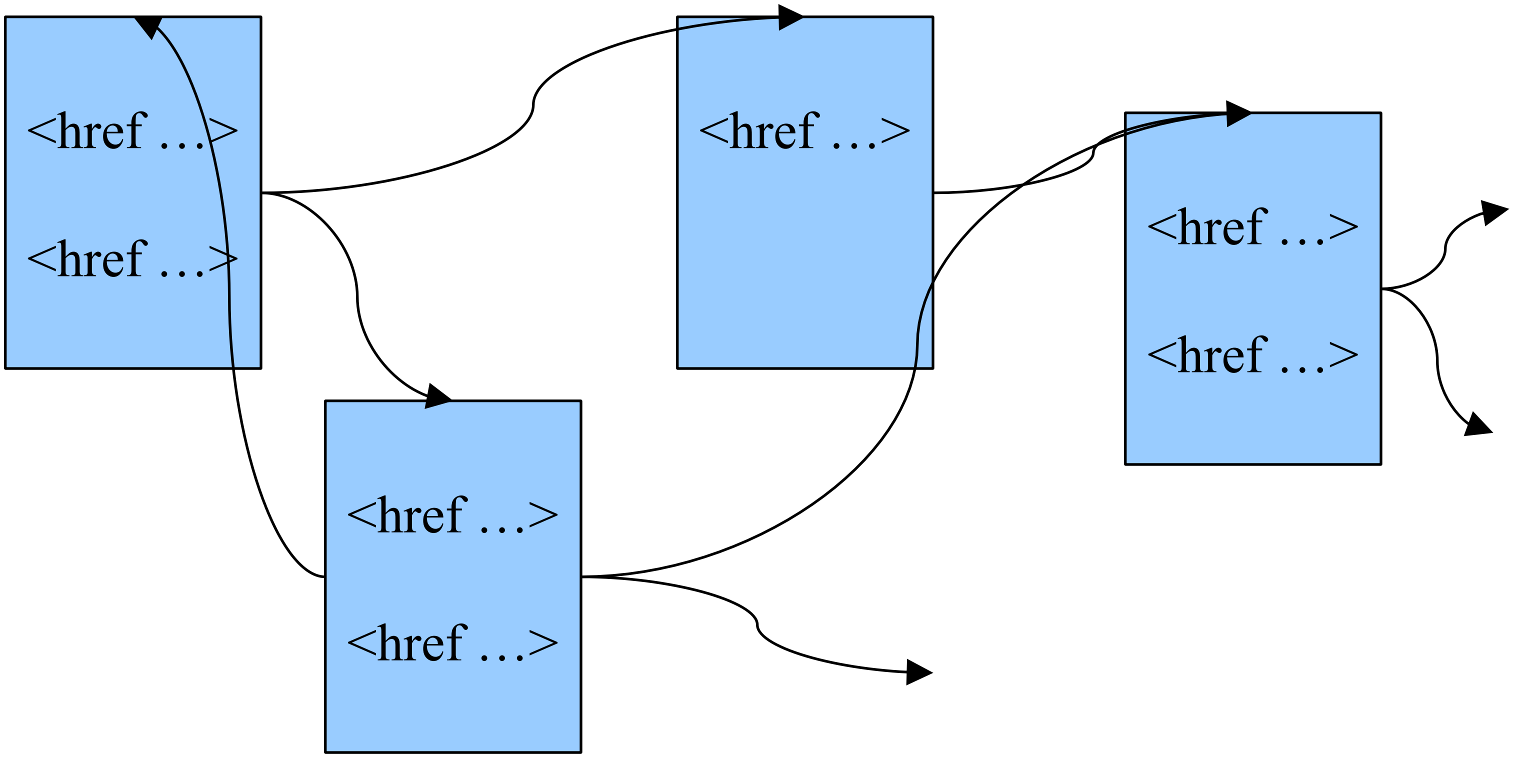
Graphs as models

- Physical objects are often modeled by meshes, which are a particular kind of graph structure.



By Jonathan Shewchuk

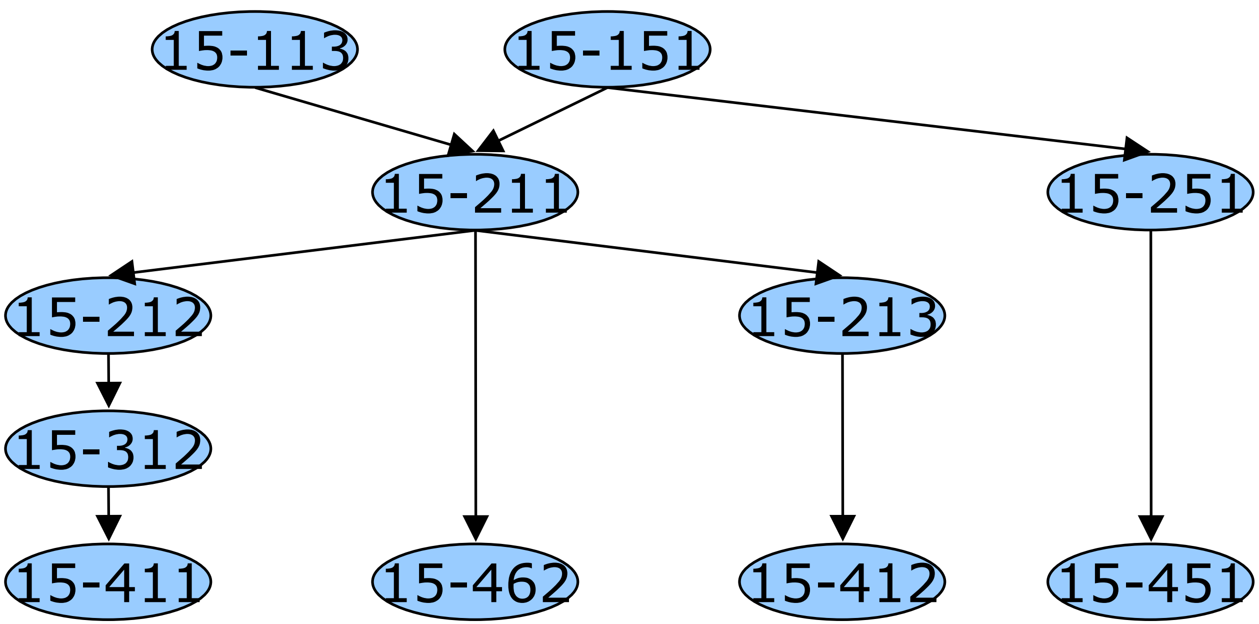
Web Graph



- Web Pages are nodes (vertices)
- HTML references are links (edges)

Relationship graphs

- Graphs are also used to model *relationships* among entities.
  - Scheduling and resource constraints.
  - Inheritance hierarchies.



More Generally

Suppose we have a system with a collection of possible configurations. Suppose further that a configuration can change into a next configuration (transition, non-deterministic).

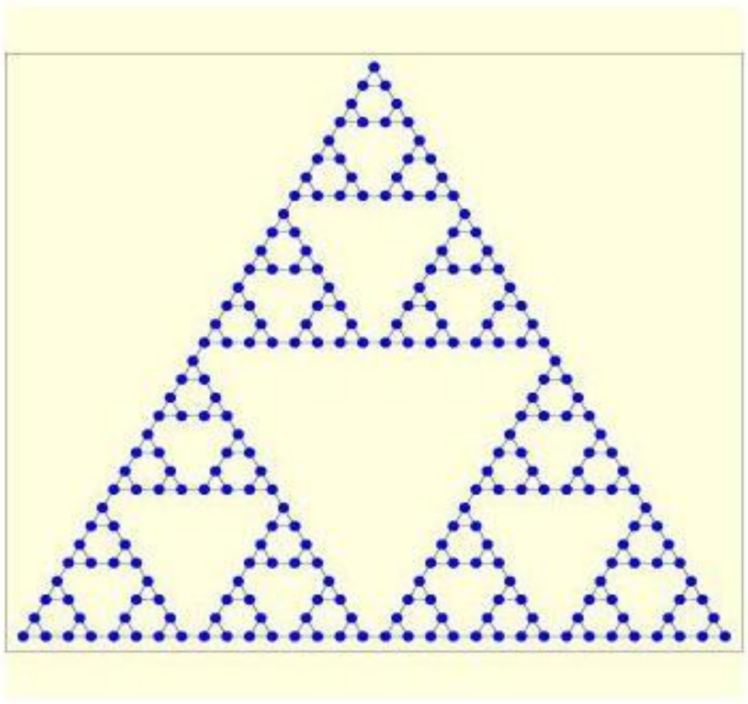
Model by a graph

G = (configurations, transitions)

Evolution of the system corresponds to a path in the graph.

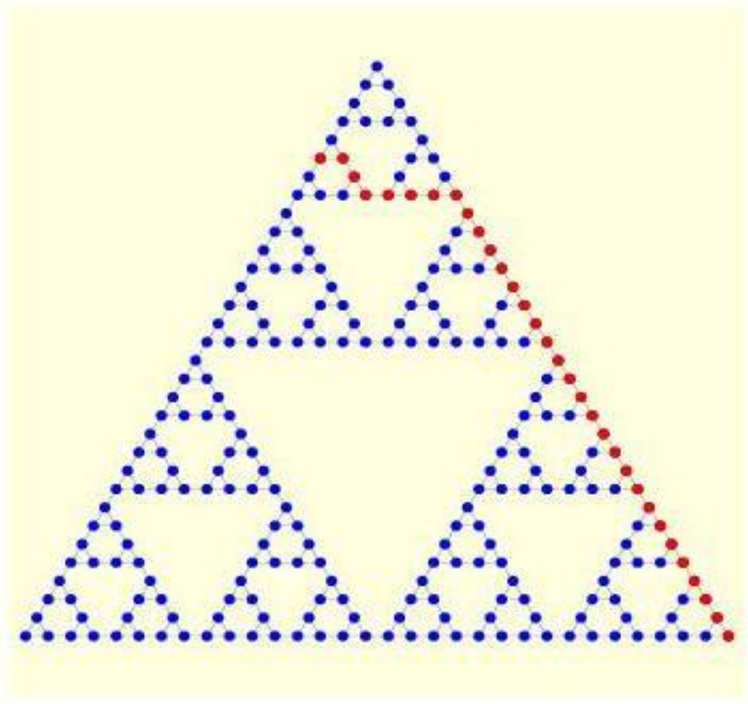
Example: Games

The game of Hanoi with 5 disks corresponds to the following graph:



Solving a Game

A solution is just a path in the graph:



Discrete Math View

Can think of a graph $G = (V,E)$ as a binary relation E on V.

E.g.

G undirected: relation symmetric

G loop-free: relation irreflexive

But this does not address additional labeling and layout information.

Path Problems

A path from vertex a to vertex b in a graph G is a sequence of vertices

$$a = x_0, x_1, x_2, \ldots, x_k = b$$

such that (x_i, x_{i+1}) is an edge in G for $i = 0,\ldots,k-1$.
k is the length of the path.

Vertex b is reachable from a if there is a path from a to b.

$R(a)$ is the set of all vertices reachable from a.

Distance

A distance from vertex a to vertex b is the length of the shortest path from a to b (infinity if there is no such path).

If the edges are labeled by a cost (a real number) the length of a path

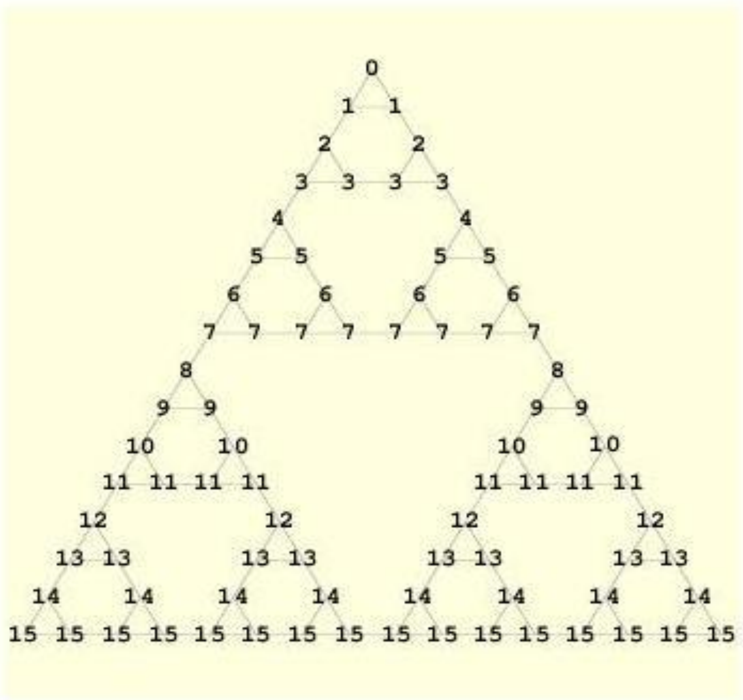
$$a = x_0, x_1, x_2 ,\ldots, x_k = b$$

is defined to be the sum of the edge-costs $cost(x_i, x_{i+1})$.

So in the unlabeled case each edge is assumed to have cost 1.

Exercise

Find a good way to compute the distance of any two Hanoi configurations.



Connectivity

A graph G is connected if $R(a) = V$ for all vertices a.

For an undirected graph this is equivalent to $R(a) = V$ for some vertex a.

A connected component of a ugraph G is a set C that is connected (meaning $R(a) = C$ for all a in C) and that is a maximal such.

For digraphs the situation is more complicated, postpone.

Typical Graph Problems

Connectivity

Given a graph G, check if G is connected.

Connected Components

Given a ugraph G, compute its connected components.

Typical Graph Problems

Shortest Path

Given a graph G and vertices a and b, find a shortest path from a to b.

Distance

Given a graph G, compute the distance between any pair of vertices.

Representing Graphs

Representing Graphs

We need a data structure to represent graphs.

Crucial parameters:

> n = number of vertices
> e = number of edges

Note that e may be quadratic in n.

Size of a graph is $n + e$.

Representing Graphs

Ignoring labels, we may assume that $V = \{1,2,\ldots,n\}$.

Need to represent E.

- Edge lists

- Adjacency lists

- Adjacency matrices

- Succinct representation

Supporting Operations

We need to be able to perform operations such as the following:

- insert/delete a vertex
- insert/delete an edge
- check whether (x,y) is an edge
- given x, enumerate its neighbors y

Enumerating neighbors is crucial in many graph algorithms.

Example: Compute degrees.

Edge Lists

A list of pairs (x,y) of vertices.

May be implemented by an array of pairs.

Size: $\Theta(e)$

Running time:

edge query ?

neighbor enumeration ?

Adjacency Lists

An array A of size n of lists of vertices:

A[x] = list of all neighbors of x.

Size: $\Theta(n+e)$

Running time:

edge query ?

neighbor enumeration ?

Adjacency Matrices

An n by n boolean array A:

A[x,y] = true iff (x,y) is an edge.

Size: $\Theta(n^2)$

Running time:

edge query ?

neighbor enumeration ?

Adjacency Matrices

Size alone often rules out the use of adjacency matrices.

But for small graphs very important alternative implementation.

Can exploit bit-parallelism or even special purpose parallel hardware (matrix multiplication).

Also a very nice conceptual tool.

Succinct Representation

For large n one often cannot afford to keep an explicit representation of the adjacencies.

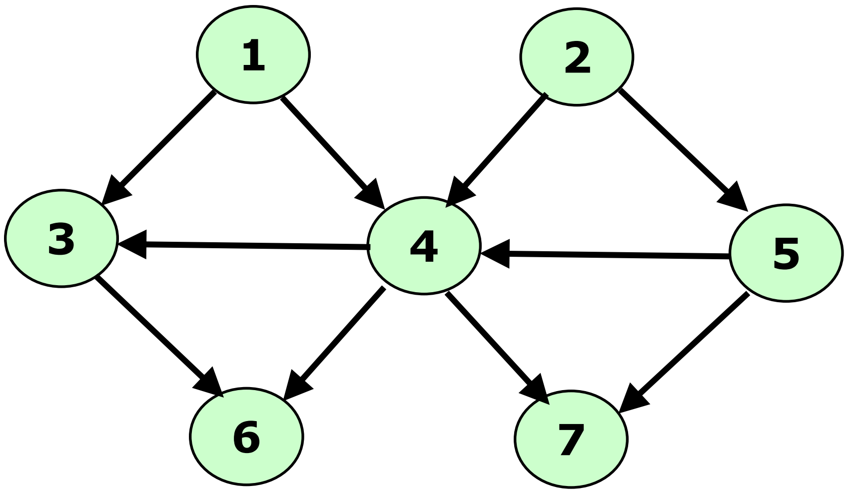
But one may be able to get by with functions:

```
boolean  edgeQ( vertex x, vertex y )

VertexList   neighbors( vertex x )
```

Typical example: the web graph.

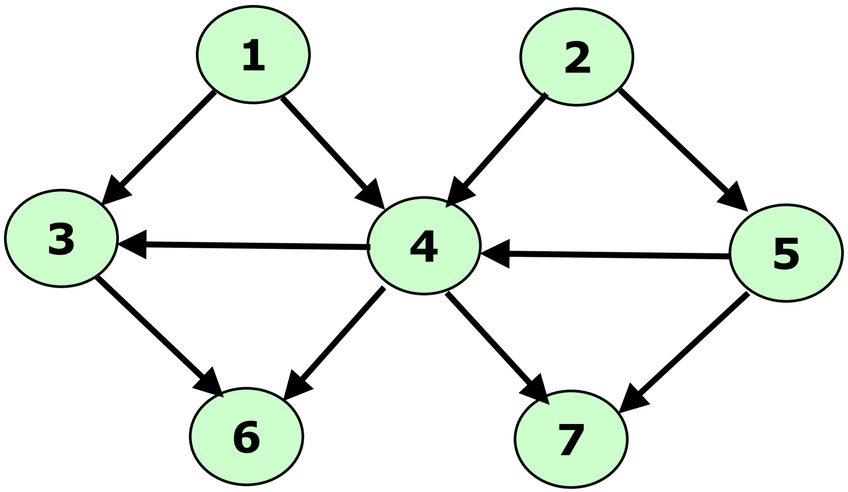
Example: Edge List

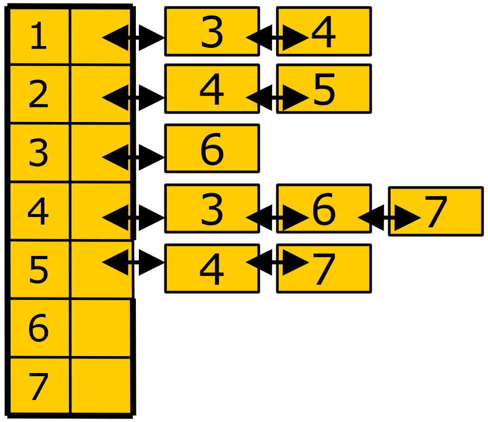


(1,3) (1,4) (2,4) (2,5)
(2,4)
(3,6) (4,6) (4,7) (5,4)
(5,7)

natural order, but could be arbitrarily permuted

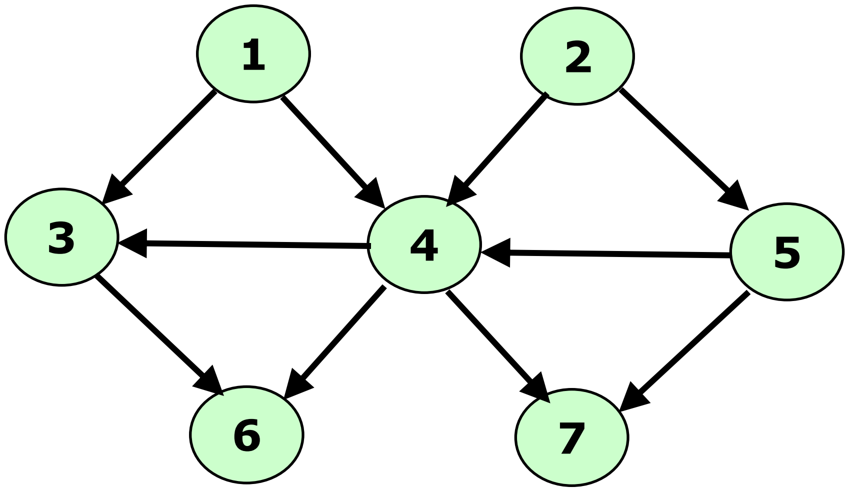
Example: Adjacency List





natural order, but lists could be arbitrarily permuted

Example: Adjacency Matrix



	1	2	3	4	5	6	7
1			x	x			
2				x	x		
3						x	
4			x			x	x
5				x			x
6							
7							

Choosing a representation

- Size of V relative to size of E is a primary factor.
  - *Dense*: e/n is large
  - *Sparse*: e/n is small
  - *Adjacency matrix is expensive if the graph is sparse.*
  - *Adjacency list is expensive if the graph is dense.*
- Dynamic changes to V.
  - *Adjacency matrix is expensive to copy/extend if V is extended.*

A Connectivity Algorithm

How do we test whether a given graph G is connected?

For an undirected graph we can

- pick any vertex v
- compute $R = R(v)$
- check if $|R| = n$

In the directed case we can repeat for all v (there are better algorithms).

Either way, the key problem is to compute R(v).

Inductive Attack

Note that

- v is in R(v)
- x in R(v) and (x,y) an edge implies y in R(v)

This can be used to construct R(v) in stages.

Edge (x,y) requires attention if x is in R but y is not.

An edge (that requires attention) is relaxed (or receives attention) when the missing endpoint is placed into R.

A Reachability Algorithm

```
R = {v};
while( some edge (x,y) requires attention )
      add y to R;    // relax the edge
```

Claim: The algorithm always terminates.

Proof: An edge can receive attention at most once.

So the loop executes at most e times.

Correctness

```
R = {v};
while( some edge (x,y) requires attention )
      add y to R;
```

Claim: Upon completion of the algorithm R = R(v).

Proof:
"R is a subset of R(v)" is a loop-invariant.

Suppose x is in R(v) but not in R. Choose one such x with minimal distance d from v. Then there is some vertex y that is in R and such that (y,x) is an edge. But then this edge requires attention, contradiction.

Efficiency

```
R = {v};
while( some edge (x,y) requires attention )
      add y to R;
```

We have to specify a way to pick edges that require attention.

Place new vertices into a container (stack or queue) and then check all incident edges.

Breadth First Search

```
bfs( vertex s )
{
  Q.enqueue( s );
  mark  s;          // put x into R
    while( !Q.empty() ) {
      x = Q.dequeue();
      forall (x,y) in E  do
        if( y not marked ) { // relax edges
            Q.enqueue(y);
            mark y;      // put y into R
      }
    }
}
```

BFS

Correctness is already taken care of.

Efficiency:

Using adjacency lists the forall loop is linear in the number of edges starting at vertex x.

So total running time is O(n+e).

How about edge lists?
How about adjacency matrices?

BFS and Distance

BFS uses a queue.

As a consequence, vertices are traversed in order of non-decreasing distance from the starting point and we can easily modify the algorithm to compute distance:

```
dist[v] = 0;
...

dist[y] = dist[x] + 1;
```

Exercise: Prove that this modification really works.

Depth First Search

```
dfs( vertex x )
{
   mark x;

   forall  (x,y) in E  do
     if( y not marked )
        dfs( y );  // explore edge
}
```

Stack is hidden via recursion.

DFS

Again: correctness taken care of.

Running time is $O(n+e)$ given adjacency lists.

DFS is a real workhorse: has many variants that solve a number of computational graph theory problems.

Application: Closure

Given a binary relation S on $\{1,2,...,n\}$, the transitive reflexive closure $trc(S)$ is the least relation R such that

- $x\ S\ y$ implies $x\ R\ y$
- $x\ R\ x$ for all x
- $x\ R\ y$ and $y\ R\ z$ implies $x\ R\ z$

If we model the relation by a graph, $trc(S)$ can be computed by repeated calls to DFS (or BFS).

Good solution if the graph is sparse.

Warshall's Algorithm

But when S is dense one might as well bite the bullet and use a cubic (in n) algorithm that has good constants.

Compute a n by n by n boolean matrix B whose first slice B[.,.,0] is the adjacency matrix of S plus diagonal:

```
for( k = 1; i <= n; k++ )
 for( x = 1; x <= n; i++ )
 for( y = 1; y <= n; j++ )
   B[x,y,k] = B[x,y,k-1] ||
                ( B[x,k,k-1] && B[k,y,k-1] );
```

Upon completion, B[.,.,n] is the adjacency matrix for

Warshall's Algorithm

```
for( k = 1; i <= n; k++ )
for( x = 1; x <= n; i++ )
for( y = 1; y <= n; j++ )
  B[x,y,k] = B[x,y,k-1] ||
             ( B[x,k,k-1] && B[k,y,k-1] );
```

Upon completion, B[.,.,n] is the adjacency matrix for the transitive reflexive closure of S.

What is the space complexity of this method?

Warshall's Algorithm

Code is beautifully simple, but correctness is far from obvious.

Claim: B[x,y,k] = 1 iff there is a path from x to y using only intermediate vertices in {1,2,...,k}.

Proof:
By induction on k.

Effectively we erase vertices higher than k and then put them back in.

Example of dynamic programming, more later.