Working with Graphs

Graphs

- vertices and edges
- adjacency matrices define graphs

A GUI to display Graphs

- components of the GUI
- code for drawing graphs

Searching Graphs

- which vertices are best connected
- find a path between two vertices

MCS 275 Lecture 40
Programming Tools and File Management
Jan Verschelde, 23 April 2008

Working with Graphs

Graphs

vertices and edges

A graph G is defined by two sets: $G = (V, E)$

- V vertices or nodes,
- E edges or links.

Adjacency matrix A:

$$A[i][j] = 0 \Leftrightarrow \text{no edge between } i \text{ and } j.$$

Directed graphs:

- $A[i][j] > 0$: edge runs from i to j
- $A[i][j] < 0$: edge runs from j to i

$|A[i][j]|$ could be the length of the edge between i and j.

Undirected graphs: A is boolean symmetric matrix.

Graphs

vertices and edges

A graph G is defined by two sets: $G = (V, E)$

- V vertices or nodes,
- E edges or links.

Adjacency matrix A:

$$A[i][j] = 0 \Leftrightarrow \text{no edge between } i \text{ and } j.$$

Directed graphs:

- $A[i][j] > 0$: edge runs from i to j
- $A[i][j] < 0$: edge runs from j to i

$|A[i][j]|$ could be the length of the edge between i and j.

Undirected graphs: A is boolean symmetric matrix.

Graphs

vertices and edges

A graph G is defined by two sets: $G = (V, E)$

- V vertices or nodes,
- E edges or links.

Adjacency matrix A:

$$A[i][j] = 0 \Leftrightarrow \text{no edge between } i \text{ and } j.$$

Directed graphs:

- $A[i][j] > 0$: edge runs from i to j
- $A[i][j] < 0$: edge runs from j to i

$|A[i][j]|$ could be the length of the edge between i and j.

Undirected graphs: A is boolean symmetric matrix.

Graphs

vertices and edges

A graph G is defined by two sets: $G = (V, E)$

- V vertices or nodes,
- E edges or links.

Adjacency matrix A:

$$A[i][j] = 0 \Leftrightarrow \text{no edge between } i \text{ and } j.$$

Directed graphs:

- $A[i][j] > 0$: edge runs from i to j
- $A[i][j] < 0$: edge runs from j to i

$|A[i][j]|$ could be the length of the edge between i and j.

Undirected graphs: A is boolean symmetric matrix.

Generating Adjacency Matrices

```
>>> from numpy import *
>>> A = random.randint(0,2,(3,3))
>>> A
array([[1, 0, 0],
       [0, 0, 1],
       [1, 1, 0]])
>>> A.transpose()
array([[1, 0, 1],
       [0, 0, 1],
       [0, 1, 0]])
>>> B = A + A.transpose()
>>> B
array([[2, 0, 1],
       [0, 0, 2],
       [1, 2, 0]])
>>> B % 2
array([[0, 0, 1],
       [0, 0, 0],
       [1, 0, 0]])
```

Generating Adjacency Matrices

```
>>> from numpy import *
>>> A = random.randint(0,2,(3,3))
>>> A
array([[1, 0, 0],
       [0, 0, 1],
       [1, 1, 0]])
>>> A.transpose()
array([[1, 0, 1],
       [0, 0, 1],
       [0, 1, 0]])
>>> B = A + A.transpose()
>>> B
array([[2, 0, 1],
       [0, 0, 2],
       [1, 2, 0]])
>>> B % 2
array([[0, 0, 1],
       [0, 0, 0],
       [1, 0, 0]])
```

Generating Adjacency Matrices

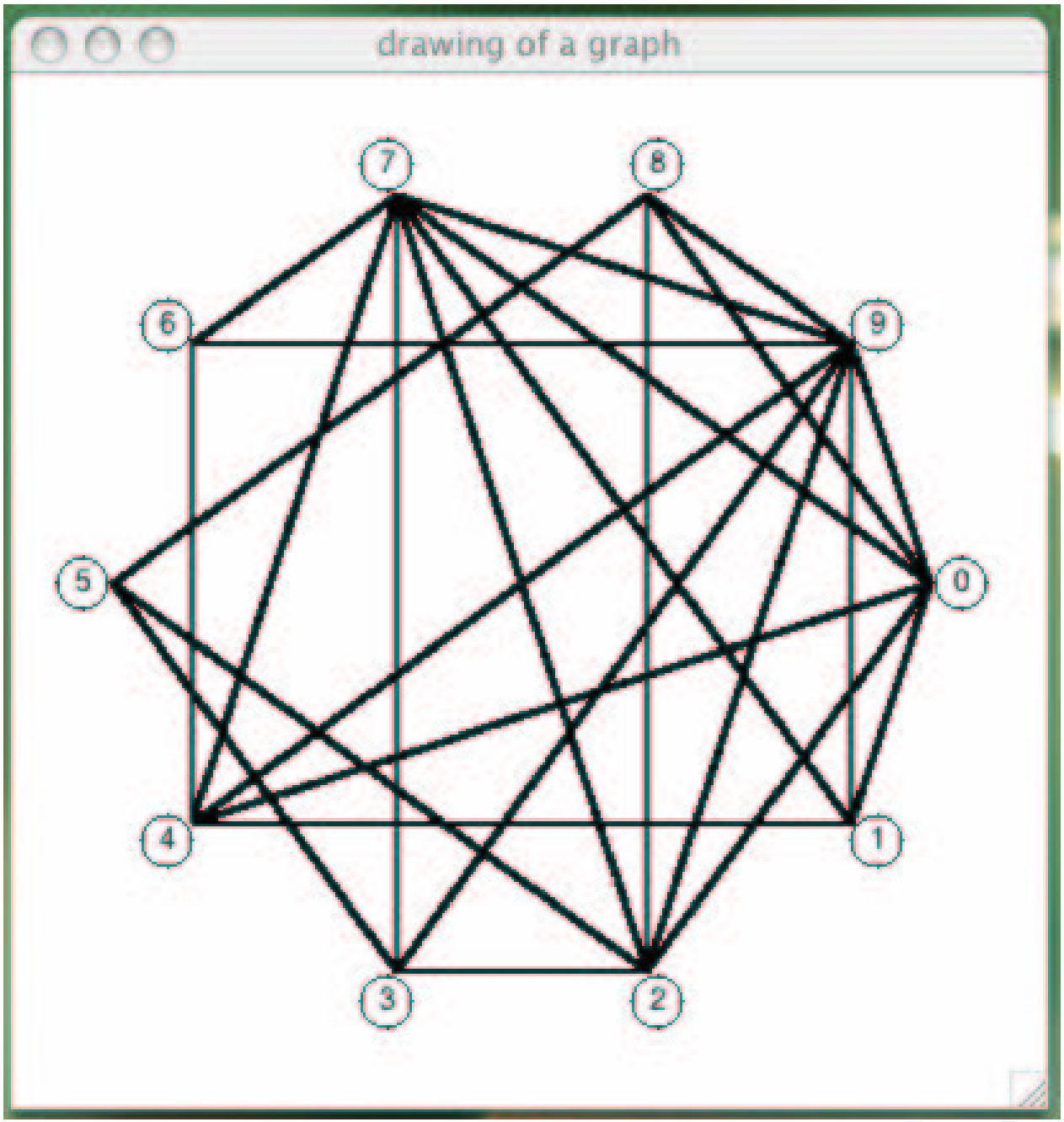
```
>>> from numpy import *
>>> A = random.randint(0,2,(3,3))
>>> A
array([[1, 0, 0],
       [0, 0, 1],
       [1, 1, 0]])
>>> A.transpose()
array([[1, 0, 1],
       [0, 0, 1],
       [0, 1, 0]])
>>> B = A + A.transpose()
>>> B
array([[2, 0, 1],
       [0, 0, 2],
       [1, 2, 0]])
>>> B % 2
array([[0, 0, 1],
       [0, 0, 0],
       [1, 0, 0]])
```

Generating Adjacency Matrices

```
>>> from numpy import *
>>> A = random.randint(0,2,(3,3))
>>> A
array([[1, 0, 0],
       [0, 0, 1],
       [1, 1, 0]])
>>> A.transpose()
array([[1, 0, 1],
       [0, 0, 1],
       [0, 1, 0]])
>>> B = A + A.transpose()
>>> B
array([[2, 0, 1],
       [0, 0, 2],
       [1, 2, 0]])
>>> B % 2
array([[0, 0, 1],
       [0, 0, 0],
       [1, 0, 0]])
```

Working with Graphs

Looking at Graphs



Adjacency Matrices define Graphs

```
$ python drawgraph.py
Give #vertices : 10
[[0 1 1 0 1 0 0 1 1 1]
 [1 0 0 0 1 0 0 1 0 1]
 [1 0 0 1 0 1 0 1 1 1]
 [0 0 1 0 0 1 0 1 0 1]
 [1 1 0 0 0 0 1 1 0 1]
 [0 0 1 1 0 0 0 0 1 0]
 [0 0 0 0 1 0 0 1 0 1]
 [1 1 1 1 1 0 1 0 0 1]
 [1 0 1 0 0 1 0 0 0 1]
 [1 1 1 1 1 0 1 1 1 0]]
```

Adjacency Matrices define Graphs

```
$ python drawgraph.py
Give #vertices : 10
[[0 1 1 0 1 0 0 1 1 1]
 [1 0 0 0 1 0 0 1 0 1]
 [1 0 0 1 0 1 0 1 1 1]
 [0 0 1 0 0 1 0 1 0 1]
 [1 1 0 0 0 0 1 1 0 1]
 [0 0 1 1 0 0 0 0 1 0]
 [0 0 0 0 1 0 0 1 0 1]
 [1 1 1 1 1 0 1 0 0 1]
 [1 0 1 0 0 1 0 0 0 1]
 [1 1 1 1 1 0 1 1 1 0]]
```

Working with Graphs

A GUI to display Graphs

To visualize a graph with modest #nodes:

1. prompt user for number of nodes,
2. generate a random adjacency matrix,
3. draw vertices and edges on canvas.

A GUI with one widget: a canvas.

Drawing the graph:

1. nodes are labeled ovals,
2. distribute nodes evenly on a circle,
3. draw a line for every edge.

A GUI to display Graphs

To visualize a graph with modest #nodes:

1. prompt user for number of nodes,
2. generate a random adjacency matrix,
3. draw vertices and edges on canvas.

A GUI with one widget: a canvas.

Drawing the graph:

1. nodes are labeled ovals,
2. distribute nodes evenly on a circle,
3. draw a line for every edge.

A GUI to display Graphs

To visualize a graph with modest #nodes:

1. prompt user for number of nodes,
2. generate a random adjacency matrix,
3. draw vertices and edges on canvas.

A GUI with one widget: a canvas.

Drawing the graph:

1. nodes are labeled ovals,
2. distribute nodes evenly on a circle,
3. draw a line for every edge.

A GUI to display Graphs

To visualize a graph with modest #nodes:

1. prompt user for number of nodes,
2. generate a random adjacency matrix,
3. draw vertices and edges on canvas.

A GUI with one widget: a canvas.

Drawing the graph:

1. nodes are labeled ovals,
2. distribute nodes evenly on a circle,
3. draw a line for every edge.

A GUI to display Graphs

To visualize a graph with modest #nodes:

1. prompt user for number of nodes,
2. generate a random adjacency matrix,
3. draw vertices and edges on canvas.

A GUI with one widget: a canvas.

Drawing the graph:

1. nodes are labeled ovals,
2. distribute nodes evenly on a circle,
3. draw a line for every edge.

A GUI to display Graphs

To visualize a graph with modest #nodes:

1. prompt user for number of nodes,
2. generate a random adjacency matrix,
3. draw vertices and edges on canvas.

A GUI with one widget: a canvas.

Drawing the graph:

1. nodes are labeled ovals,
2. distribute nodes evenly on a circle,
3. draw a line for every edge.

A GUI to display Graphs

To visualize a graph with modest #nodes:

1. prompt user for number of nodes,
2. generate a random adjacency matrix,
3. draw vertices and edges on canvas.

A GUI with one widget: a canvas.

Drawing the graph:

1. nodes are labeled ovals,
2. distribute nodes evenly on a circle,
3. draw a line for every edge.

Working with Graphs

the function main( )

```
from Tkinter import *
from numpy import *
import math

def main():
    """
    Prompts the user for the number of vertices
    and generates a random adjacency matrix.
    """
    n = input('Give #vertices : ')
    B = random.randint(0,2,(n,n))
    A = (B + B.transpose()) % 2
    print A
    top = Tk()
    show = DrawGraph(top,n,A)
    show.Draw()
    top.mainloop()
```

the function main( )

```
from Tkinter import *
from numpy import *
import math

def main():
    """
    Prompts the user for the number of vertices
    and generates a random adjacency matrix.
    """
    n = input('Give #vertices : ')
    B = random.randint(0,2,(n,n))
    A = (B + B.transpose()) % 2
    print A
    top = Tk()
    show = DrawGraph(top,n,A)
    show.Draw()
    top.mainloop()
```

the function main( )

```
from Tkinter import *
from numpy import *
import math

def main():
    """
    Prompts the user for the number of vertices
    and generates a random adjacency matrix.
    """
    n = input('Give #vertices : ')
    B = random.randint(0,2,(n,n))
    A = (B + B.transpose()) % 2
    print A
    top = Tk()
    show = DrawGraph(top,n,A)
    show.Draw()
    top.mainloop()
```

the function main( )

```
from Tkinter import *
from numpy import *
import math

def main():
    """
    Prompts the user for the number of vertices
    and generates a random adjacency matrix.
    """
    n = input('Give #vertices : ')
    B = random.randint(0,2,(n,n))
    A = (B + B.transpose()) % 2
    print A
    top = Tk()
    show = DrawGraph(top,n,A)
    show.Draw()
    top.mainloop()
```

Constructor of class DrawGraph

```
class DrawGraph():
    """
    GUI to draw a graph on canvas.
    """
    def __init__(self,wdw,n,A):
        """
        Canvas for a graph of n vertices.
        The adjacency matrix is in A.
        """
        wdw.title('drawing of a graph')
        self.d = 400
        self.nv = n
        self.Ae = A
        self.c = Canvas(wdw,width=self.d,\
          height=self.d,bg ='white')
        self.c.grid(row=0,column=0)
```

Constructor of class DrawGraph

```
class DrawGraph():
    """
    GUI to draw a graph on canvas.
    """
    def __init__(self,wdw,n,A):
        """
        Canvas for a graph of n vertices.
        The adjacency matrix is in A.
        """
        wdw.title('drawing of a graph')
        self.d = 400
        self.nv = n
        self.Ae = A
        self.c = Canvas(wdw,width=self.d,\
          height=self.d,bg ='white')
        self.c.grid(row=0,column=0)
```

Constructor of class DrawGraph

```
class DrawGraph():
    """
    GUI to draw a graph on canvas.
    """
    def __init__(self,wdw,n,A):
        """
        Canvas for a graph of n vertices.
        The adjacency matrix is in A.
        """
        wdw.title('drawing of a graph')
        self.d = 400
        self.nv = n
        self.Ae = A
        self.c = Canvas(wdw,width=self.d,\
          height=self.d,bg ='white')
        self.c.grid(row=0,column=0)
```

Drawing a Graph

```
def Draw(self):
    """
    Draws the graph defined by the
    adjacency matrix on n vertices.
    """
    n = self.nv
    for i in range(0,n):
        self.DrawVertex(i)
        for j in range(0,n):
            if self.Ae[i,j] == 1:
                self.DrawEdge(i,j)
```

Drawing a Graph

```
def Draw(self):
    """
    Draws the graph defined by the
    adjacency matrix on n vertices.
    """
    n = self.nv
    for i in range(0,n):
        self.DrawVertex(i)
        for j in range(0,n):
            if self.Ae[i,j] == 1:
                self.DrawEdge(i,j)
```

Drawing a Graph

```
def Draw(self):
    """
    Draws the graph defined by the
    adjacency matrix on n vertices.
    """
    n = self.nv
    for i in range(0,n):
        self.DrawVertex(i)
        for j in range(0,n):
            if self.Ae[i,j] == 1:
                self.DrawEdge(i,j)
```

Drawing Vertices

```
def DrawVertex(self,i):
    """
    Draws the i-th vertex on canvas.
    """
    n = self.nv
    cx = self.d/2
    cy = self.d/2
    radius = 0.43*self.d
    vx = cx + radius*math.cos(2*i*math.pi/n)
    vy = cy + radius*math.sin(2*i*math.pi/n)
    self.c.create_oval(vx-10,vy-10,vx+10,vy+10,\
        width=1,outline='black',fill='white')
    v = str(i)
    self.c.create_text(vx,vy,text=v)
```

Drawing Vertices

```
def DrawVertex(self,i):
    """
    Draws the i-th vertex on canvas.
    """
    n = self.nv
    cx = self.d/2
    cy = self.d/2
    radius = 0.43*self.d
    vx = cx + radius*math.cos(2*i*math.pi/n)
    vy = cy + radius*math.sin(2*i*math.pi/n)
    self.c.create_oval(vx-10,vy-10,vx+10,vy+10,\
        width=1,outline='black',fill='white')
    v = str(i)
    self.c.create_text(vx,vy,text=v)
```

Drawing Vertices

```
def DrawVertex(self,i):
    """
    Draws the i-th vertex on canvas.
    """
    n = self.nv
    cx = self.d/2
    cy = self.d/2
    radius = 0.43*self.d
    vx = cx + radius*math.cos(2*i*math.pi/n)
    vy = cy + radius*math.sin(2*i*math.pi/n)
    self.c.create_oval(vx-10,vy-10,vx+10,vy+10,\
        width=1,outline='black',fill='white')
    v = str(i)
    self.c.create_text(vx,vy,text=v)
```

Drawing Edges

```
def DrawEdge(self,i,j):
    """
    Draws the edge between vertices i and j.
    """
    n = self.nv
    cx = self.d/2
    cy = self.d/2
    radius = 0.4*self.d
    ax = cx + radius*math.cos(2*i*math.pi/n)
    ay = cy + radius*math.sin(2*i*math.pi/n)
    bx = cx + radius*math.cos(2*j*math.pi/n)
    by = cy + radius*math.sin(2*j*math.pi/n)
    self.c.create_line(ax,ay,bx,by,width=2)
```

Drawing Edges

```
def DrawEdge(self,i,j):
    """
    Draws the edge between vertices i and j.
    """
    n = self.nv
    cx = self.d/2
    cy = self.d/2
    radius = 0.4*self.d
    ax = cx + radius*math.cos(2*i*math.pi/n)
    ay = cy + radius*math.sin(2*i*math.pi/n)
    bx = cx + radius*math.cos(2*j*math.pi/n)
    by = cy + radius*math.sin(2*j*math.pi/n)
    self.c.create_line(ax,ay,bx,by,width=2)
```

Drawing Edges

```
def DrawEdge(self,i,j):
    """
    Draws the edge between vertices i and j.
    """
    n = self.nv
    cx = self.d/2
    cy = self.d/2
    radius = 0.4*self.d
    ax = cx + radius*math.cos(2*i*math.pi/n)
    ay = cy + radius*math.sin(2*i*math.pi/n)
    bx = cx + radius*math.cos(2*j*math.pi/n)
    by = cy + radius*math.sin(2*j*math.pi/n)
    self.c.create_line(ax,ay,bx,by,width=2)
```

Working with Graphs

The Degree of a Vertex

Find out which vertices are best connected.

The degree of a vertex v counts #edges
that have v as one of the end points.

The higher the degree of a vertex,
the better the vertex is connected.

Computing the degree of a vertex i,
given the adjacency matrix of the graph:

- compute the sum of the elements on row i, or
- compute the sum of the elements on column i.

The Degree of a Vertex

Find out which vertices are best connected.

The degree of a vertex v counts #edges
that have v as one of the end points.

The higher the degree of a vertex,
the better the vertex is connected.

Computing the degree of a vertex i,
given the adjacency matrix of the graph:

- compute the sum of the elements on row i, or
- compute the sum of the elements on column i.

The Degree of a Vertex

Find out which vertices are best connected.

The degree of a vertex v counts #edges
that have v as one of the end points.

The higher the degree of a vertex,
the better the vertex is connected.

Computing the degree of a vertex i,
given the adjacency matrix of the graph:

- compute the sum of the elements on row i, or
- compute the sum of the elements on column i.

The Degree of a Vertex

Find out which vertices are best connected.

The degree of a vertex v counts #edges
that have v as one of the end points.

The higher the degree of a vertex,
the better the vertex is connected.

Computing the degree of a vertex i,
given the adjacency matrix of the graph:

- compute the sum of the elements on row i, or
- compute the sum of the elements on column i.

The Degree of a Vertex

Find out which vertices are best connected.

The degree of a vertex v counts #edges
that have v as one of the end points.

The higher the degree of a vertex,
the better the vertex is connected.

Computing the degree of a vertex i,
given the adjacency matrix of the graph:

- compute the sum of the elements on row i, or
- compute the sum of the elements on column i.

Computing the Degree of a Vertex

```
>>> from numpy import *
>>> A = random.randint(0,2,(10,10))
>>> A
array([[1, 0, 0, 1, 1, 1, 0, 0, 0, 0],
       [1, 0, 1, 1, 1, 0, 0, 0, 1, 1],
       [1, 1, 0, 0, 0, 1, 0, 0, 0, 1],
       [0, 1, 1, 0, 1, 0, 1, 1, 0, 0],
       [0, 1, 0, 1, 1, 1, 1, 0, 0, 0],
       [0, 0, 0, 1, 1, 1, 0, 1, 1, 0],
       [0, 0, 0, 1, 1, 1, 0, 0, 0, 0],
       [1, 1, 1, 0, 0, 0, 1, 1, 1, 0],
       [0, 1, 1, 0, 1, 0, 0, 0, 1, 1],
       [1, 0, 1, 1, 0, 0, 0, 1, 0, 0]])
>>> [A[i][1] for i in range(0,10)]
[0, 0, 1, 1, 1, 0, 0, 1, 1, 0]
>>> sum(_)
5
```

Computing the Degree of a Vertex

```
>>> from numpy import *
>>> A = random.randint(0,2,(10,10))
>>> A
array([[1, 0, 0, 1, 1, 1, 0, 0, 0, 0],
       [1, 0, 1, 1, 1, 0, 0, 0, 1, 1],
       [1, 1, 0, 0, 0, 1, 0, 0, 0, 1],
       [0, 1, 1, 0, 1, 0, 1, 1, 0, 0],
       [0, 1, 0, 1, 1, 1, 1, 0, 0, 0],
       [0, 0, 0, 1, 1, 1, 0, 1, 1, 0],
       [0, 0, 0, 1, 1, 1, 0, 0, 0, 0],
       [1, 1, 1, 0, 0, 0, 1, 1, 1, 0],
       [0, 1, 1, 0, 1, 0, 0, 0, 1, 1],
       [1, 0, 1, 1, 0, 0, 0, 1, 0, 0]])
>>> [A[i][1] for i in range(0,10)]
[0, 0, 1, 1, 1, 0, 0, 1, 1, 0]
>>> sum(_)
5
```

Computing the Degree of a Vertex

```
>>> from numpy import *
>>> A = random.randint(0,2,(10,10))
>>> A
array([[1, 0, 0, 1, 1, 1, 0, 0, 0, 0],
       [1, 0, 1, 1, 1, 0, 0, 0, 1, 1],
       [1, 1, 0, 0, 0, 1, 0, 0, 0, 1],
       [0, 1, 1, 0, 1, 0, 1, 1, 0, 0],
       [0, 1, 0, 1, 1, 1, 1, 0, 0, 0],
       [0, 0, 0, 1, 1, 1, 0, 1, 1, 0],
       [0, 0, 0, 1, 1, 1, 0, 0, 0, 0],
       [1, 1, 1, 0, 0, 0, 1, 1, 1, 0],
       [0, 1, 1, 0, 1, 0, 0, 0, 1, 1],
       [1, 0, 1, 1, 0, 0, 0, 1, 0, 0]])
>>> [A[i][1] for i in range(0,10)]
[0, 0, 1, 1, 1, 0, 0, 1, 1, 0]
>>> sum(_)
5
```

Computing all Degrees

```
>>> from numpy import *
>>> A = random.randint(0,2,(10,10))
>>> A
array([[1, 0, 0, 1, 1, 1, 0, 0, 0, 0],
       [1, 0, 1, 1, 1, 0, 0, 0, 1, 1],
       [1, 1, 0, 0, 0, 1, 0, 0, 0, 1],
       [0, 1, 1, 0, 1, 0, 1, 1, 0, 0],
       [0, 1, 0, 1, 1, 1, 1, 0, 0, 0],
       [0, 0, 0, 1, 1, 1, 0, 1, 1, 0],
       [0, 0, 0, 1, 1, 1, 0, 0, 0, 0],
       [1, 1, 1, 0, 0, 0, 1, 1, 1, 0],
       [0, 1, 1, 0, 1, 0, 0, 0, 1, 1],
       [1, 0, 1, 1, 0, 0, 0, 1, 0, 0]])
>>> [sum([A[i][j] for i in range(0,10)]
... ) for j in range(0,10)]
[5, 5, 5, 6, 7, 5, 3, 4, 4, 3]
```

Computing all Degrees

```
>>> from numpy import *
>>> A = random.randint(0,2,(10,10))
>>> A
array([[1, 0, 0, 1, 1, 1, 0, 0, 0, 0],
       [1, 0, 1, 1, 1, 0, 0, 0, 1, 1],
       [1, 1, 0, 0, 0, 1, 0, 0, 0, 1],
       [0, 1, 1, 0, 1, 0, 1, 1, 0, 0],
       [0, 1, 0, 1, 1, 1, 1, 0, 0, 0],
       [0, 0, 0, 1, 1, 1, 0, 1, 1, 0],
       [0, 0, 0, 1, 1, 1, 0, 0, 0, 0],
       [1, 1, 1, 0, 0, 0, 1, 1, 1, 0],
       [0, 1, 1, 0, 1, 0, 0, 0, 1, 1],
       [1, 0, 1, 1, 0, 0, 0, 1, 0, 0]])
>>> [sum([A[i][j] for i in range(0,10)]
... ) for j in range(0,10)]
[5, 5, 5, 6, 7, 5, 3, 4, 4, 3]
```

All Vertices with same Degree

more list comprehensions...

```
>>> d = [5, 5, 5, 6, 7, 5, 3, 4, 4, 3]
>>> d.index(5)
0
```

We want *all* indices for which entry in d is 5.

```
>>> L = [(i,d[i]) for i in range(0,len(d))]
>>> L
[(0, 5), (1, 5), (2, 5), (3, 6), (4, 7), ...
 (5, 5), (6, 3), (7, 4), (8, 4), (9, 3)]
>>> F = filter(lambda (x,y): y==5, L)
>>> F
[(0, 5), (1, 5), (2, 5), (5, 5)]
>>> [ x for (x,y) in F ]
[0, 1, 2, 5]
```

All Vertices with same Degree

more list comprehensions...

```
>>> d = [5, 5, 5, 6, 7, 5, 3, 4, 4, 3]
>>> d.index(5)
0
```

We want *all* indices for which entry in d is 5.

```
>>> L = [(i,d[i]) for i in range(0,len(d))]
>>> L
[(0, 5), (1, 5), (2, 5), (3, 6), (4, 7), ...
 (5, 5), (6, 3), (7, 4), (8, 4), (9, 3)]
>>> F = filter(lambda (x,y): y==5, L)
>>> F
[(0, 5), (1, 5), (2, 5), (5, 5)]
>>> [ x for (x,y) in F ]
[0, 1, 2, 5]
```

All Vertices with same Degree

more list comprehensions...

```
>>> d = [5, 5, 5, 6, 7, 5, 3, 4, 4, 3]
>>> d.index(5)
0
```

We want *all* indices for which entry in d is 5.

```
>>> L = [(i,d[i]) for i in range(0,len(d))]
>>> L
[(0, 5), (1, 5), (2, 5), (3, 6), (4, 7), ...
 (5, 5), (6, 3), (7, 4), (8, 4), (9, 3)]
>>> F = filter(lambda (x,y): y==5, L)
>>> F
[(0, 5), (1, 5), (2, 5), (5, 5)]
>>> [ x for (x,y) in F ]
[0, 1, 2, 5]
```

All Vertices with same Degree

more list comprehensions...

```
>>> d = [5, 5, 5, 6, 7, 5, 3, 4, 4, 3]
>>> d.index(5)
0
```

We want *all* indices for which entry in d is 5.

```
>>> L = [(i,d[i]) for i in range(0,len(d))]
>>> L
[(0, 5), (1, 5), (2, 5), (3, 6), (4, 7), ...
 (5, 5), (6, 3), (7, 4), (8, 4), (9, 3)]
>>> F = filter(lambda (x,y): y==5, L)
>>> F
[(0, 5), (1, 5), (2, 5), (5, 5)]
>>> [ x for (x,y) in F ]
[0, 1, 2, 5]
```

Best Connected Vertices

```
$ python searchgraph.py
Give #vertices : 10
[[0 1 1 0 1 1 0 1 1 0]
 [1 0 0 0 1 0 0 0 0 0]
 [1 0 0 0 0 0 1 1 0 1]
 [0 0 0 0 1 1 0 1 0 1]
 [1 1 0 1 0 0 1 1 0 1]
 [1 0 0 1 0 0 1 1 1 0]
 [0 0 1 0 1 1 0 0 1 0]
 [1 0 1 1 1 1 0 0 1 0]
 [1 0 0 0 0 1 1 1 0 1]
 [0 0 1 1 1 0 0 0 1 0]]
best connected vertices :  [0, 4, 7]
with degree  6
```

Best Connected Vertices

```
$ python searchgraph.py
Give #vertices : 10
[[0 1 1 0 1 1 0 1 1 0]
 [1 0 0 0 1 0 0 0 0 0]
 [1 0 0 0 0 0 1 1 0 1]
 [0 0 0 0 1 1 0 1 0 1]
 [1 1 0 1 0 0 1 1 0 1]
 [1 0 0 1 0 0 1 1 1 0]
 [0 0 1 0 1 1 0 0 1 0]
 [1 0 1 1 1 1 0 0 1 0]
 [1 0 0 0 0 1 1 1 0 1]
 [0 0 1 1 1 0 0 0 1 0]]
best connected vertices :  [0, 4, 7]
with degree  6
```

A Python Function

to compute best connected vertices

```
def MaxDegrees(A):
    """
    The degree of a vertex v counts #edges
    that have v as an end point.
    Returns a tuple: first element is a list
    whose degree equals the second element.
    """
    n = len(A)
    d = [sum([A[i][j] for i in range(0,n)]) \
                      for j in range(0,n)]
    m = max(d)
    L = [(i,d[i]) for i in range(0,len(d))]
    F = filter(lambda (x,y): y == m, L)
    r = [ x for (x,y) in F ]
    return (r,m)
```

A Python Function

to compute best connected vertices

```
def MaxDegrees(A):
    """
    The degree of a vertex v counts #edges
    that have v as an end point.
    Returns a tuple: first element is a list
    whose degree equals the second element.
    """
    n = len(A)
    d = [sum([A[i][j] for i in range(0,n)]) \
                      for j in range(0,n)]
    m = max(d)
    L = [(i,d[i]) for i in range(0,len(d))]
    F = filter(lambda (x,y): y == m, L)
    r = [ x for (x,y) in F ]
    return (r,m)
```

A Python Function

to compute best connected vertices

```
def MaxDegrees(A):
    """
    The degree of a vertex v counts #edges
    that have v as an end point.
    Returns a tuple: first element is a list
    whose degree equals the second element.
    """
    n = len(A)
    d = [sum([A[i][j] for i in range(0,n)]) \
                      for j in range(0,n)]
    m = max(d)
    L = [(i,d[i]) for i in range(0,len(d))]
    F = filter(lambda (x,y): y == m, L)
    r = [ x for (x,y) in F ]
    return (r,m)
```

A Python Function

to compute best connected vertices

```
def MaxDegrees(A):
    """
    The degree of a vertex v counts #edges
    that have v as an end point.
    Returns a tuple: first element is a list
    whose degree equals the second element.
    """
    n = len(A)
    d = [sum([A[i][j] for i in range(0,n)]) \
                      for j in range(0,n)]
    m = max(d)
    L = [(i,d[i]) for i in range(0,len(d))]
    F = filter(lambda (x,y): y == m, L)
    r = [ x for (x,y) in F ]
    return (r,m)
```

A Python Function

to compute best connected vertices

```
def MaxDegrees(A):
    """
    The degree of a vertex v counts #edges
    that have v as an end point.
    Returns a tuple: first element is a list
    whose degree equals the second element.
    """
    n = len(A)
    d = [sum([A[i][j] for i in range(0,n)]) \
                      for j in range(0,n)]
    m = max(d)
    L = [(i,d[i]) for i in range(0,len(d))]
    F = filter(lambda (x,y): y == m, L)
    r = [ x for (x,y) in F ]
    return (r,m)
```

Working with Graphs

Graphs
- vertices and edges
- adjacency matrices define graphs

A GUI to display Graphs
- components of the GUI
- code for drawing graphs

Searching Graphs
- which vertices are best connected
- find a path between two vertices

Finding a Path

```
$ python searchgraph.py
Give #vertices : 10
[[0 0 1 1 1 0 1 0 0 1]
 [0 0 0 0 0 0 0 1 0 0]
 [1 0 0 1 0 0 0 1 0 0]
 [1 0 1 0 1 1 0 1 1 0]
 [1 0 0 1 0 1 0 0 1 1]
 [0 0 0 1 1 0 1 1 0 1]
 [1 0 0 0 0 1 0 1 0 0]
 [0 1 1 1 0 1 1 0 0 0]
 [0 0 0 1 1 0 0 0 0 0]
 [1 0 0 0 1 1 0 0 0 0]]
best connected vertices :  [3]
with degree  6
finding a path
  give start vertex : 0
  give target vertex : 8
path :  [0, 2, 3, 8]
```

Finding a Path

```
$ python searchgraph.py
Give #vertices : 10
[[0 0 1 1 1 0 1 0 0 1]
 [0 0 0 0 0 0 0 1 0 0]
 [1 0 0 1 0 0 0 1 0 0]
 [1 0 1 0 1 1 0 1 1 0]
 [1 0 0 1 0 1 0 0 1 1]
 [0 0 0 1 1 0 1 1 0 1]
 [1 0 0 0 0 1 0 1 0 0]
 [0 1 1 1 0 1 1 0 0 0]
 [0 0 0 1 1 0 0 0 0 0]
 [1 0 0 0 1 1 0 0 0 0]]
best connected vertices :  [3]
with degree  6
finding a path
  give start vertex : 0
  give target vertex : 8
path :  [0, 2, 3, 8]
```

Finding a Path

```
$ python searchgraph.py
Give #vertices : 10
[[0 0 1 1 1 0 1 0 0 1]
 [0 0 0 0 0 0 0 1 0 0]
 [1 0 0 1 0 0 0 1 0 0]
 [1 0 1 0 1 1 0 1 1 0]
 [1 0 0 1 0 1 0 0 1 1]
 [0 0 0 1 1 0 1 1 0 1]
 [1 0 0 0 0 1 0 1 0 0]
 [0 1 1 1 0 1 1 0 0 0]
 [0 0 0 1 1 0 0 0 0 0]
 [1 0 0 0 1 1 0 0 0 0]]
best connected vertices :  [3]
with degree  6
finding a path
  give start vertex : 0
  give target vertex : 8
path :  [0, 2, 3, 8]
```

A backtracking Solution

to find a path between two vertices

To find a path between vertices i and j,
using at most d intermediate vertices...

Let A be the adjacency matrix.
Recursive function:

1. base case: check if $A[i][j] == 1$,
2. if $d > 0$: enumerate intermediate vertices.

Restrictions on intermediate vertices:

1. there must be an edge from current vertex,
2. intermediate vertex not visited earlier.

Main data structure: list of visited vertices.

A backtracking Solution

to find a path between two vertices

To find a path between vertices i and j,
using at most d intermediate vertices...

Let A be the adjacency matrix.
Recursive function:

1. base case: check if $A[i][j] == 1$,
2. if $d > 0$: enumerate intermediate vertices.

Restrictions on intermediate vertices:

1. there must be an edge from current vertex,
2. intermediate vertex not visited earlier.

Main data structure: list of visited vertices.

A backtracking Solution

to find a path between two vertices

To find a path between vertices i and j,
using at most d intermediate vertices...

Let A be the adjacency matrix.
Recursive function:

1. base case: check if $A[i][j] == 1$,
2. if $d > 0$: enumerate intermediate vertices.

Restrictions on intermediate vertices:

1. there must be an edge from current vertex,
2. intermediate vertex not visited earlier.

Main data structure: list of visited vertices.

A backtracking Solution

to find a path between two vertices

To find a path between vertices i and j,
using at most d intermediate vertices...

Let A be the adjacency matrix.
Recursive function:

1. base case: check if $A[i][j] == 1$,
2. if $d > 0$: enumerate intermediate vertices.

Restrictions on intermediate vertices:

1. there must be an edge from current vertex,
2. intermediate vertex not visited earlier.

Main data structure: list of visited vertices.

A backtracking Solution

to find a path between two vertices

To find a path between vertices i and j,
using at most d intermediate vertices...

Let A be the adjacency matrix.
Recursive function:

1. base case: check if $A[i][j] == 1$,
2. if $d > 0$: enumerate intermediate vertices.

Restrictions on intermediate vertices:

1. there must be an edge from current vertex,
2. intermediate vertex not visited earlier.

Main data structure: list of visited vertices.

A backtracking Solution

to find a path between two vertices

To find a path between vertices i and j,
using at most d intermediate vertices...

Let A be the adjacency matrix.
Recursive function:

1. base case: check if $A[i][j] == 1$,
2. if $d > 0$: enumerate intermediate vertices.

Restrictions on intermediate vertices:

1. there must be an edge from current vertex,
2. intermediate vertex not visited earlier.

Main data structure: list of visited vertices.

A backtracking Solution

to find a path between two vertices

To find a path between vertices i and j,
using at most d intermediate vertices...

Let A be the adjacency matrix.
Recursive function:

1. base case: check if $A[i][j] == 1$,
2. if $d > 0$: enumerate intermediate vertices.

Restrictions on intermediate vertices:

1. there must be an edge from current vertex,
2. intermediate vertex not visited earlier.

Main data structure: list of visited vertices.

the Function FindPath()

```
def FindPath(A,i,j,d,P):
    """
    Searches for a path between vertices i
    and j in a graph with adjacency matrix A,
    using at most d intermediary nodes.
    P contains the list of visited nodes.
    """
    if A[i][j] == 1:
        return P + [j]
    else:
        if d > 0:
            for k in range(0,len(A)):
                if A[i][k] == 1 and not k in P:
                    KP = FindPath(A,k,j,d-1,P + [k])
                    if KP[len(KP)-1] == j: return KP
        return P
```

the Function FindPath()

```
def FindPath(A,i,j,d,P):
    """
    Searches for a path between vertices i
    and j in a graph with adjacency matrix A,
    using at most d intermediary nodes.
    P contains the list of visited nodes.
    """
    if A[i][j] == 1:
        return P + [j]
    else:
        if d > 0:
            for k in range(0,len(A)):
                if A[i][k] == 1 and not k in P:
                    KP = FindPath(A,k,j,d-1,P + [k])
                    if KP[len(KP)-1] == j: return KP
        return P
```

the Function FindPath()

```
def FindPath(A,i,j,d,P):
    """
    Searches for a path between vertices i
    and j in a graph with adjacency matrix A,
    using at most d intermediary nodes.
    P contains the list of visited nodes.
    """
    if A[i][j] == 1:
        return P + [j]
    else:
        if d > 0:
            for k in range(0,len(A)):
                if A[i][k] == 1 and not k in P:
                    KP = FindPath(A,k,j,d-1,P + [k])
                    if KP[len(KP)-1] == j: return KP
        return P
```

the Function FindPath()

```
def FindPath(A,i,j,d,P):
    """
    Searches for a path between vertices i
    and j in a graph with adjacency matrix A,
    using at most d intermediary nodes.
    P contains the list of visited nodes.
    """
    if A[i][j] == 1:
        return P + [j]
    else:
        if d > 0:
            for k in range(0,len(A)):
                if A[i][k] == 1 and not k in P:
                    KP = FindPath(A,k,j,d-1,P + [k])
                    if KP[len(KP)-1] == j: return KP
        return P
```

the Function FindPath()

```
def FindPath(A,i,j,d,P):
    """
    Searches for a path between vertices i
    and j in a graph with adjacency matrix A,
    using at most d intermediary nodes.
    P contains the list of visited nodes.
    """
    if A[i][j] == 1:
        return P + [j]
    else:
        if d > 0:
            for k in range(0,len(A)):
                if A[i][k] == 1 and not k in P:
                    KP = FindPath(A,k,j,d-1,P + [k])
                    if KP[len(KP)-1] == j: return KP
        return P
```

the Function FindPath()

```
def FindPath(A,i,j,d,P):
    """
    Searches for a path between vertices i
    and j in a graph with adjacency matrix A,
    using at most d intermediary nodes.
    P contains the list of visited nodes.
    """
    if A[i][j] == 1:
        return P + [j]
    else:
        if d > 0:
            for k in range(0,len(A)):
                if A[i][k] == 1 and not k in P:
                    KP = FindPath(A,k,j,d-1,P + [k])
                    if KP[len(KP)-1] == j: return KP
        return P
```

Summary + Assignments

Graphs lead to an interesting class of matrices.
Searching through graps often requires backtracking.
Assignments:

1. Change `drawgraph.py` so that after prompting the user for the number of vertices, the program should either generate a random matrix, or allow the user to enter a specific matrix.
2. Extend the `drawgraph.py` with an entry field and a draw button. Each time the draw button is pressed a new random graph is generated for the number of vertices equal to the value in the entry field.
3. Modify `FindPath()` so that it shows the trace of all function calls.
4. Extend `FindPath()` so that it enumerate *all* paths between two vertices.

Last Homework Collection

Homework collected this Friday 25 April at 11AM:

- assignments 1, and 4 of Lecture 38,
- assignment 1 of Lecture 39,
- assignments 1 and 3 of Lecture 40.