Part 1

Representations and Algorithms
Graph Processing

- Compute some static value of the graph degree distribution, clustering coefficient

- Identify some subset of edges or vertices shortest path, spanning tree, connected components
From the Graph Atlas:

### Tables of graph numbers

**Graphs with \( n \) vertices, for \( n = 1, 2, \ldots, 25 \)**

<table>
<thead>
<tr>
<th>( n )</th>
<th>graphs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>11</td>
</tr>
<tr>
<td>5</td>
<td>34</td>
</tr>
<tr>
<td>6</td>
<td>156</td>
</tr>
<tr>
<td>7</td>
<td>1044</td>
</tr>
<tr>
<td>8</td>
<td>12346</td>
</tr>
<tr>
<td>9</td>
<td>274668</td>
</tr>
<tr>
<td>10</td>
<td>12005168</td>
</tr>
<tr>
<td>11</td>
<td>10189 97864</td>
</tr>
<tr>
<td>12</td>
<td>1650911 72592</td>
</tr>
<tr>
<td>13</td>
<td>5050 20313 67952</td>
</tr>
<tr>
<td>14</td>
<td>2905415 56572 35488</td>
</tr>
<tr>
<td>15</td>
<td>31426 48596 98043 08768</td>
</tr>
<tr>
<td>16</td>
<td>640 01015 70452 75578 94928</td>
</tr>
<tr>
<td>17</td>
<td>2459358 64153 53293 26837 19776</td>
</tr>
<tr>
<td>18</td>
<td>178757 77251 45611 70054 78781 90848</td>
</tr>
<tr>
<td>19</td>
<td>24637 80925 31250 04524 38300 74914 32768</td>
</tr>
<tr>
<td>20</td>
<td>6454 90122 79579 98418 56164 63849 07427 49440</td>
</tr>
<tr>
<td>21</td>
<td>3222 02728 99808 98343 35022 44253 75528 36160 97664</td>
</tr>
<tr>
<td>22</td>
<td>3070 84648 30941 44300 63756 85171 87105 41058 66578 14272</td>
</tr>
<tr>
<td>23</td>
<td>5599 24939 69979 20805 97976 38081 94621 79812 27634 84589 81632</td>
</tr>
<tr>
<td>24</td>
<td>19570 49063 02078 44792 21748 62416 72625 60041 22075 26706 33657 54368</td>
</tr>
<tr>
<td>25</td>
<td>131331 39356 98955 19432 16154 84058 16890 14638 92147 06146 48338 04585 76384</td>
</tr>
</tbody>
</table>
Not easy to write

• The Challenge of Binary Search - it took more than 10 years to publish the first correct binary search

• Graph algorithms are difficult to test for correctness
The Big Oh

Represents the complexity of a worst case analysis.

Common complexities:

\[ O(1) \quad O(\log N) \quad O(N) \]
\[ N \quad O(\log N) \quad O(N^2) \]

It is about growth not efficiency!
Graph algorithms can be hard

Some are NP Hard - proved to have no efficient solutions - better than $O(N^k)$ $k = constant$

Traveling salesman: $O(N!)$
$15!/5! = 10$ billion

Many others are unknown

Luckily problems can often be greatly simplified!
- a bridge is an edge that if removed ...

- an articulation point is a vertex that if removed ...

... would separate a connected graph into at least two disjoint sub-graphs

no bridges = edge connected biconnected => no articulation points
Representations

V - number of vertices (nodes), E - number of edges

Edge array  -  size: E

Adjacency matrix -  size: V * V

Adjacency lists -  size: V + E

“pure” representations,
## Adjacency matrix

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
<th>h</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>c</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>d</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>e</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>f</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>g</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>h</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Adjacency List

- **a**: b, e, g
- **b**: c, h
- **c**: b, g
- **d**: c, e
- **e**: a, d
- **f**: c, g
- **g**: a, c, f
- **h**: b
## Representation properties

<table>
<thead>
<tr>
<th></th>
<th>V - number of vertices</th>
<th>E - number of edges</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>matrix</strong></td>
<td><strong>list</strong></td>
</tr>
<tr>
<td>space</td>
<td>$V^2$</td>
<td>$V+E$</td>
</tr>
<tr>
<td>find edge</td>
<td>1</td>
<td>$V$</td>
</tr>
<tr>
<td>insert edge</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>path $v$ to $w$?</td>
<td>$V^2$</td>
<td>$V+E$</td>
</tr>
</tbody>
</table>
Important details

- Directional or undirectional
- Cyclic or acyclic
- Self loops
- Parallel edges
- Node or edge removal (hiding)

Optimize the data structure for the task
Graph Processing Problems

• Traversal, shortest paths, longest paths
• Simple and strong connectivity
• Subgraphs, cliques
• Planarity (no edge crossing)
• Isomorphism
• Graph drawing
Which two graphs are identical?
<table>
<thead>
<tr>
<th>Vertex</th>
<th>Pre</th>
<th>Post</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>c</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>d</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>e</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>f</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>g</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>h</td>
<td>8</td>
<td>6</td>
</tr>
</tbody>
</table>
Properties of DFS trees

for a link going from \( v \) to \( w \)

- edges corresponding to recursive calls (tree edges)
  - \textbf{tree link} if \( w \) has not been visited
  - \textbf{parent link} if during the traversal we went from \( v \) to \( w \)

- edges connecting a vertex with an ancestor that is not a parent
  - \textbf{back link} if \( \text{pre}[w] < \text{pre}[v] \)
  - \textbf{down link} if \( \text{pre}[w] > \text{pre}[v] \)
<table>
<thead>
<tr>
<th>Vertex</th>
<th>Pre</th>
<th>Post</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>e</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>g</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>c</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>h</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>d</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>f</td>
<td>8</td>
<td>1</td>
</tr>
</tbody>
</table>
Properties of BFS traversal

Corresponds to shortest path (hops) if the edges have equal weight

- DFS wends it way through the graph
- BFS sweeps through the graph

but the only difference between them is in the data structure that holds the nodes, Stack vs Queue
Algorithm Example

**Strong components**: find the largest group of mutually reachable nodes

- **brute force** - check every pair $\sim V^2$
- **clever methods** - linear time $\sim V$

**Kosaraju algorithm**: Run the depth first search on its reverse and compute the post-order numbering. Then run the search again but this time visit the highest post ordered vertices first.
Shortest Paths

Weight for every edge (cost, distance, time)

- non-negative and negative weights
- single source
- all-pairs shortest paths

Concept - relaxation - at each step check whether the path is shorter than the paths seen so far

Dijkstra’s algorithm - $O(ElgV)$ or $O(VlglgV)$
In class exercise

1. List the nodes in a bfs order starting from Dallas node (hop distance), ...

2. Find the cheapest route from Dallas to Chicago
Graph processing with the python **networkx** library

```
import networkx as nx

def read_data(fname):
    "Creates a directed graph from an edgelist file"
    graph = nx.DiGraph()
    for line in file(fname):
        head, tail, cost = line.split(',
        graph.add_edge( head, tail, float(cost) )
    return graph

graph = read_data( 'cities.dat' )
print nx.path.bfs(graph, 'Dallas')
print nx.path.bidirectional_dijkstra(graph, 'Dallas', 'Los Angeles')
```

```
[ 'Dallas', 'Miami', 'Chicago', 'New York', 'Detroit', 'San Francisco', 'Los Angeles' ]
(450.0, [ 'Dallas', 'Miami', 'Detroit', 'Chicago', 'Los Angeles' ])
```
Mutating Graph methods

- `G.add_node(n)`, `G.add_nodes_from(nbunch)`
- `G.delete_node(n)`, `G.delete_nodes_from(nbunch)`
- `G.add_edge(n1,n2)`, `G.add_edge(e)`, where `e=(u,v)`
- `G.add_edges_from(nbunch)`
- `G.delete_edge(n1,n2)`, `G.delete_edge(e)`, where `e=(u,v)`
- `G.delete_edges_from(nbunch)`
- `G.add_path(nlist)`
- `G.add_cycle(nlist)`
- `G.clear()`
- `G.subgraph(nbunch,inplace=True)`

Non-mutating Graph methods

- `len(G)`
- `G.has_node(n)`
- `n in G` (equivalent to `G.has_node(n)`)
- `for n in G`: (iterate through the nodes of `G`)
- `G.nodes()`
- `G.nodes_iter()`
- `G.has_edge(n1,n2)`, `G.has_neighboor(n1,n2)`, `G.get_edge(n1,n2)`
- `G.edges()`, `G.edges(n)`, `G.edges(nbunch)`
- `G.edges_iter()`, `G.edges_iter(n)`, `G.edges_iter(nbunch)`
- `G.neighbors(n)`
- `G[n]` (equivalent to `G.neighbors(n)`)
- `G.neighbors_iter(n)` # iterator over neighbors
- `G.number_of_nodes()`, `G.order()`
- `G.number_of_edges()`, `G.size()`
- `G.edge_boundary(nbunch1)`, `G.node_boundary(nbunch1)`
- `G.degree(n)`, `G.degree(nbunch)`
- `G.degree_iter(n)`, `G.degree_iter(nbunch)`
- `G.is_directed()`
- `G.info()` # print various info about a graph
- `G.prepare_nbunch(nbunch)` # return list of nodes in `G` and `nbunch`

Methods returning a new graph

- `G.subgraph(nbunch)`
- `G.subgraph(nbunch)`
- `G.copy()`
- `G.to_undirected()`
- `G.to_directed()`

Create an empty graph structure (a "null graph") with zero nodes and zero edges.

```
>>> from networkx import *
>>> G=Graph()
```

`G` can be grown in several ways. By adding one node at a time:

```
>>> G.add_node(1)
```

by adding a list of nodes:

```
>>> G.add_nodes_from([2,3])
```

by using an iterator:

```
>>> G.add_edges_from(xrange(100,110))
```

or by adding any `nbunch` of nodes (see above definition of an `nbunch`):

```
>>> H=path_graph(10)
>>> G.add_nodes_from(H)
```

`H` can be another graph, or dict, or set, or even a file. Any hashable object (except `None`) can be used.

```
>>> G.add_node(8)
```

`G` can also be grown by adding one edge at a time:

```
>>> G.add_edge(1,2)
```

by adding a list of edges:

```
>>> G.add_edges_from([(1,2),(1,3)])
```
import networkx as nx
import networkx.generators.random_graphs as rgen

# generate a few different types of random graphs
er = rgen.erdos_renyi_graph( 100, 0.1 )
ws = rgen.watts_strogatz_graph ( 100, 10, 0.2 )
ba = rgen.barabasi_albert_graph( 100, 2 )

# collect them in a list
all = [ er, ws, ba ]

# print various network measures
# diameter = the diameter is the maximum of all pairs shortest path
# betweenness_centrality = fraction of number of shortests paths that go through each node
for graph in all:
    avg = nx.cluster.average_clustering( graph )
    diam = nx.distance.diameter( graph )
    cent = nx.centrality.betweenness_centrality(graph)
    print 'Avg=%s, Diameter=%s, BCentral=%s' % (avg, diam, cent)

print er.nodes()[:10]
print er.edges()[:10]
Good documentation helps you find what you need

**Table of Contents**

- Everything
- Modules
  - networkx
  - networkx.central
  - networkx.clustering
  - networkx.component
  - networkx.convert
  - networkx.core
  - networkx.dag
  - networkx.digraph
  - networkx.distance
  - networkxdrawing
  - networkxdrawing.layout
  - networkxdrawing.nx_agraph
  - networkxdrawing.nx_pydot
  - networkxdrawing.nx_pygraph

---

**Module generators**

[show_private]
Other graph concepts

- **Biconnectivity**: every pair of edges is connected by two disjoint paths

- **Spanning trees**: the tree that can reach all vertices

- **Topological sorting (DAG)**: re-label vertices so that every edge points from a lower number to a higher one.

- **Strong components**: each pair of nodes is reachable from both ends
Part 2

Software Tools
Categories

- **GUI programs:** Pajek, SpectraNET, yED
- **Command line programs:** graphviz, R
- **Libraries:** LEDA, boost, pygraphlib
Input - Output

- **Input**: user input (mouse-clicks) or files
- **Output**: image files or files

- Focus: *visualization, display or graph algorithms*

- Most file formats are (should be) text!
- GraphML, Pajek, DOT, LEDA
Potential standard

The GraphML File Format

What is GraphML?

GraphML is a comprehensive and easy-to-use file format for graphs. It consists of a language core to describe the structural properties of a graph and a flexible extension mechanism to add application-specific data. Its main features include support of:

- directed, undirected, and mixed graphs,
- hypergraphs,
- hierarchical graphs,
- graphical representations,
- references to external data,
- application-specific attribute data, and
- light-weight parsers.

Unlike many other file formats for graphs, GraphML does not use a custom syntax. Instead, it is based on XML and hence ideally suited as a common denominator for all kinds of services generating, archiving, or processing graphs.

Getting Started

An easy introduction to GraphML is the GraphML Primer. GraphML Primer is a non-normative document intended to
GraphML example

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<graphml>
  <graph id="G" edgedefault="undirected">
    <node id="Detroit" />
    <node id="Chicago" />
    <node id="New York" />
    <node id="Miami" />
    <edge source="Detroit" target="Chicago" />
    <edge source="Detroit" target="Miami" />
    <edge source="Miami" target="New York" />
  </graph>
</graphml>
```

GraphML an XML is a markup language for standardization at the cost of making everything a whole lot more complicated.
Graphviz

command line graph visualizer - the language that it understands is called dot

Graph Visualization

Graph visualization is a way of representing structural information as diagrams of abstract graphs and networks. Automatic graph drawing has many important applications in software engineering, database and web design, networking, and in visual interfaces for many other domains.

Graphviz is open source graph visualization software. It has several main graph layout programs. See the gallery for some sample layouts. It also has web and interactive graphical interfaces, and auxiliary tools, libraries, and language bindings.

The Mac OS X edition of Graphviz, by Glen Low, won two 2004 Apple Design Awards.
digraph world {
size="7,7";
{rank=same: S8 S24 S1 S35 S30;}
{rank=same: T8 T24 T1 T35 T30;}
{rank=same: S5 S7 S6 10 2;}
{rank=same: S25 9 38 40 13 17 12 18;}
{rank=same: 26 42 11 3 33 19 39 14 16;}
{rank=same: 4 31 34 21 41 28 20;}
{rank=same: 27 5 22 32 29 15;}
{rank=same: 6 25;}

S8 -> a;
S24 -> 25;
S24 -> 27;
S1 -> 2;
S1 -> 10;
S35 -> 43;
S35 -> 36;
S30 -> 31;
S30 -> 33;
9 -> 42;
9 -> T1;
25 -> T1;
25 -> 26;
27 -> T24;
2 -> {3; 16; 17; T1; 18}
10 -> {11; 14; T1; 13; 12;}
31 -> T1;
31 -> 32;
33 -> T30;
33 -> 34;
42 -> 4;
26 -> 4;
3 -> 4;
16 -> 15;
17 -> 19;
16 -> 29;
11 -> 4;
14 -> 15;
37 -> {39; 41; 38; 40;}
13 -> 19;
12 -> 25;
Pajek

windows program,

http://vlado.fmf.uni-lj.si/pub/networks/pajek/
yED - yFiles

yED (editor) is free but yFiles costs $$$

the best graph layout you’ll ever see

What is yFiles?

yFiles is an extensive Java™ class library that provides algorithms and components for analyzing, viewing and drawing graphs, diagrams, and networks.

The current yFiles version is 2.3. Refer to the release notes to learn more about technical requirements and feature enhancements.

Benefits

yFiles provides essential building blocks for Java applications that need to analyze, visualize, edit, or automatically draw graphs, diagrams, or networks. Anyone who needs such components as part of his/her own application should consider using yFiles.

Versatility is one of the main features of yFiles. Our customers come from diverse application areas such as:

- biochemical network analysis and visualization
- business process modeling
- data mining (e.g. log file analysis)
- database management and modeling
- network management
- social networks
- software engineering (e.g. UML diagramming)
- workflow management
- WWW visualization
LEDA

Library of efficient data types - written in C++
$150 for the academic license

You can buy the Leda Guide from Amazon
(also available free on the web)

High performance, professional product,
covers much more than just graphs.

So abstract that it feels like a
high level programming language
#include <LEDA/graph.h>
#include <LEDA/basic_graph_alg.h>
using namespace leda;

int main() {
    graph G;
    list<node> dfs_res ;
    
    node n0 = G.new_node ();
    ... 
    G.new_edge (n0,n1);
    ...
    dfs_res = DFS (G, n0, n5);

    forall (node, dfs_res)
        G.print_node(node);
}
A Free Resource

Data Structures and Algorithms with Object-Oriented Design Patterns (Python Java, C# and C++)

by Bruno R. Preiss

web-book with source code:
http://www.brpreiss.com/books/opus7/
Books

- Graph Algorithms
  by Robert Sedgewick
  many code examples C, C++, Java

LEDA
A Platform for Combinatorial and Geometric Computing
by Kurt Mehlhorn and Stefan Näher