Graph Processing

István Albert

Bioinformatics Consulting Center
Huck Institute for Life Sciences
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
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 - more than $O(N^k)$ $k = constant$

Traveling salesman: $O(N!)$
$$15!/5! = 10 \text{ billion}$$

Many others are unknown

Luckily the 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

Real representations are often more complicated
### 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>0</td>
<td>1</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
ga 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><strong>matrix</strong></td>
<td></td>
<td></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
<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>6</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>5</td>
</tr>
</tbody>
</table>
Properties of DFS trees

for a link going from $v$ to $w$

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

- edges connecting a vertex with an ancestor that is not a parent
  - **back link** if $\text{pre}[w] < \text{pre}[v]$
  - **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
Application Example

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

- **brute force** - check every pair ~ V^2
- **clever methods** - linear time ~ 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 $O(VlgV)$
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
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
Books

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

**LEDA**
**A Platform for Combinatorial and Geometric Computing**
by Kurt Mehlhorn and Stefan Näher
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/
Part 2

Software Tools
Classification

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

- **Input:** mouse-clicks or certain file formats
- **Output:** image files or files in certain formats

- **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 version="1.0" encoding="UTF-8"?>
<graphml>
<graph id="G" edgedefault="undirected">
  <node id="n0"/>
  <node id="n1"/>
  <node id="n2"/>
  <node id="n3"/>
  <edge source="n0" target="n2"/>
  <edge source="n1" target="n2"/>
  <edge source="n2" target="n3"/>
</graph>
</graphml>
Graphviz

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

Graphviz - Graph Visualization Software

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.
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
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
LEDA example

#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);
}
NetworkX- Python graph library

https://networkx.lanl.gov/

```python
from networkx import Graph

G = Graph()
G.add_edge(1, 2)
G.add_edge(2, 3)
...

print G.nodes()
# [1, 2, 3, 4, 5, 6]

for v in G.nodes():
    print v, G.degree(v)

#1 1
#2 5
#4 2
...```
**base:** Base classes for graphs and digraphs.
**centrality:** Centrality measures.
**cliques:** Cliques - Find and manipulate cliques of graphs
**cluster:** Compute clustering coefficients and transitivity of graphs.
**cores:** Find and manipulate the k-cores of a graph
**drawing**
  ◦ **layout:** Layout (positioning) algorithms for graph drawing.
  ◦ **nx_pydot:** Import and export networkx networks to dot format using pydot.
  ◦ **nx_vtk:** Draw networks in 3d with vtk.
**generators:** A package for generating various graphs in networkx.
  ◦ **atlas:** Generators for the small graph atlas.
  ◦ **classic:** Generators for some classic graphs.
  ◦ **degree_seq:** Generate graphs with a given degree sequence.
  ◦ **geometric:** Generators for geometric graphs.
  ◦ **random_graphs:** Generators for random graphs
  ◦ **small:** Various small and named graphs, together with some compact generators.
**hybrid:** Hybrid
**io:** Read and write graphs and networks.
**isomorph:** Fast checking to see if graphs are not isomorphic.
**judybase:** Use pyjudy interface to Judy as underlying data structure.
**operators:** Operations on graphs; including union, complement, subgraph.
**paths:** Shortest paths, diameter, radius, eccentricity, and related methods.
**queues:** Helper queues for use in graph searching.
**release:** Release data for NetworkX.
**search:** Search algorithms, shortest path, spanning trees, etc.
**search_class:** Graph search classes
**spectrum:** Laplacian, adjacency matrix, and spectrum of graphs.