Software Science Seminar

Week 7 - Graph Search

Claus Aranha
caranha@cs.tsukuba.ac.jp

College of Information Sciences

2015-06-08

Last updatet June 6, 2015

Weeks 7 and 8

Chapter 7 – Graph Transversal

- Characteristics;
- Representation;
- Transversal;

Chapter 8 – Graph Algorithms

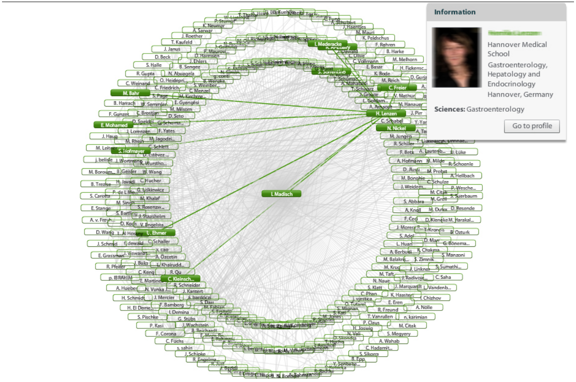
- Network Flow;
- Bipartite Graphs;
- Spanning Trees;

Graphs Everywhere!

Graphs are one of the unifying themes of computer sciences: Many theoretical and application problems can be described or thought of as some sort of graph.

- Communication Networks
- Decision Trees
- Program Execution
- Human Relationships
- Geometric shapes
- Language Grammar
- Transportation Networks
- Scheduling Restrictions
- Module Dependencies
- File System Structure
- Recurrence Relations
- Finite State Automata

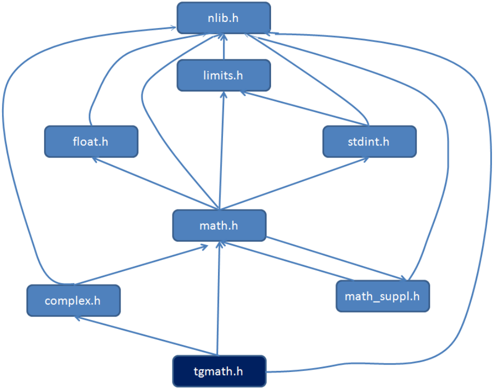
Graph Examples: Network Graph



A Network Graph can represent the connections between people in a network. It shows how a minority of people are central nodes that connect many, while the majority only have a few connections.

Image from researchgate.net

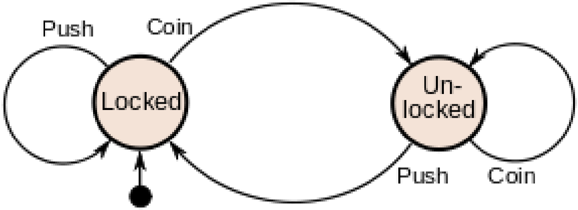
Graph Examples: Dependency Graph



A Dependency Graph links libraries and software packages by their direct dependencies. It can detect circular dependencies, or orphaned packages.

Image from Wikipedia

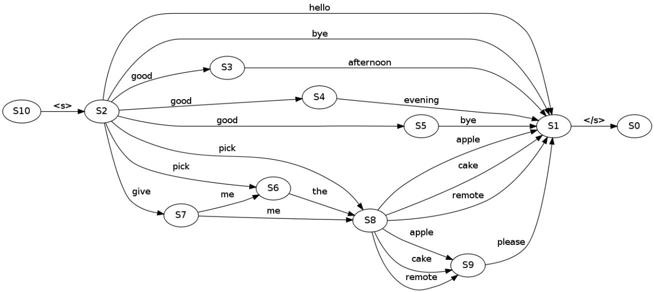
Graph Examples: State Graph



The usual way to represent State machines is a state graph. You can see at a glance all the states and transitions of the state machine, and have an idea of what it does.

Image from Wikipedia

Graph Examples: Grammar Graph



Gramma graphs show all possible sentences that can be generated from a given grammar. You can generate any sequences by walking the graph using different patterns.

Image from openhri.net

Basic Definition

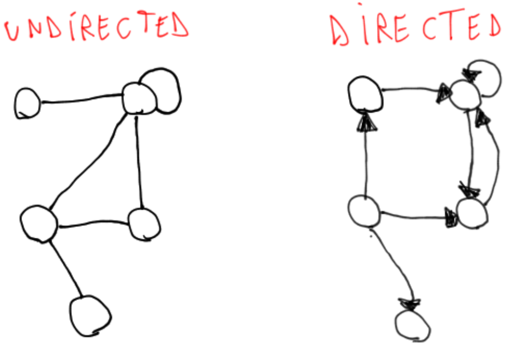
Definition

- A Graph $G = (V, E)$ is defined by a set V of Vertices, and a set E of Edges.
- An edge consists of an ordered or unordered pair of vertices from V.

Graphs can be described by their properties, which influence how they can be represented and analyzed.

Undirected x Directed (1)

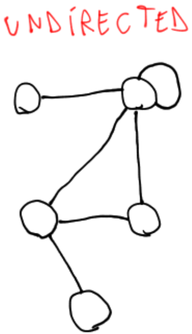
- Undirected Graphs: Imply that if an edge (x, y) exists, then an edge (x, y) also exists.
- Directed Graphs: Edges can connect Vertices in a single way.

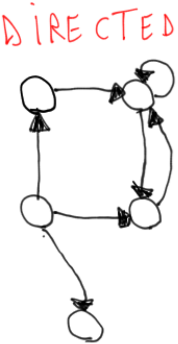


Undirected x Directed (2)

Why is this important?

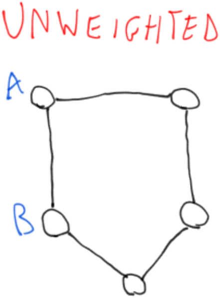
- In undirected graphs, the path from A to B is the same as the path from B to A.
- In directed graphs, this is not necessarily true.

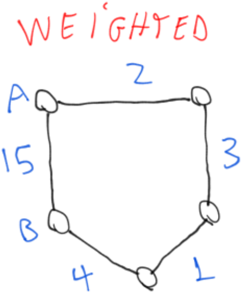




Weighted x Unweighted (1)

- Weighted Graphs: Each edge in E has an associated numerical value w.
- Unweighted Graphs: All edges can be considered to have the same value (or no value).

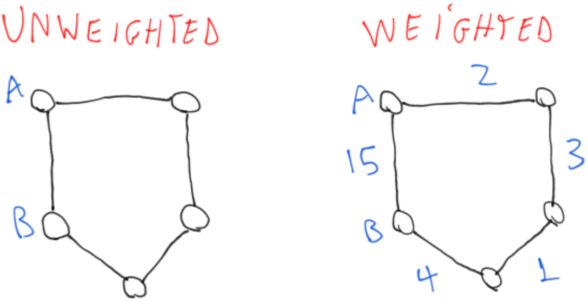




Weighted x Unweighted (2)

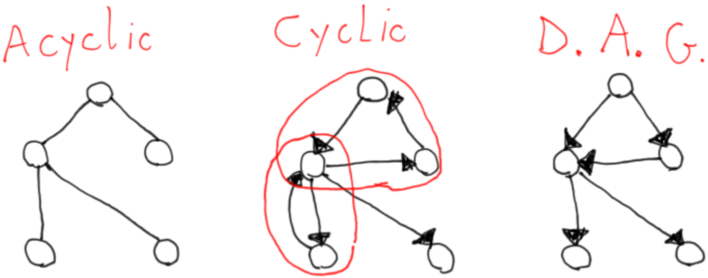
Consequences of Weights

- Weights change how we define the shortest path;
- We can generalize unweighted graphs to graphs with weight 1;
- Negative weights may imply in infinite loops!



Cyclic x Acyclic (1)

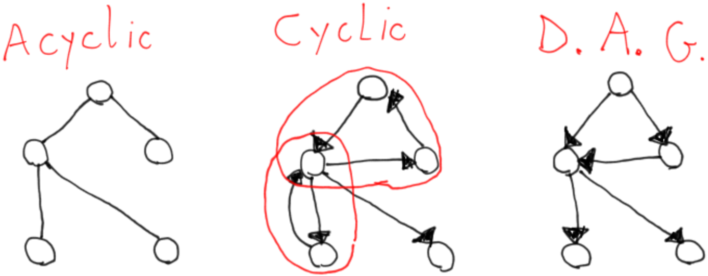
- Cyclic: A subset of Vertices in the graph are connected as a cycle.
- Acyclic: Does not contain any cycles;



Cyclic x Acyclic (2)

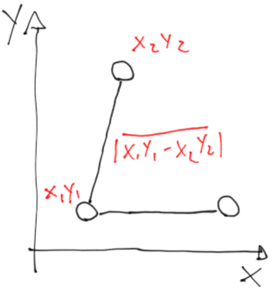
Directed Acyclic Graphs

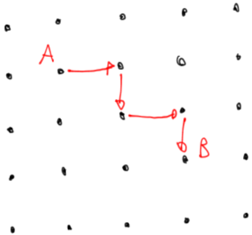
If a graph is both Directed and Acyclic at the same time, it can be Sorted Topologically. Topological sorting is useful for a wide variety of algorithms.



Embedded x Topological (1)

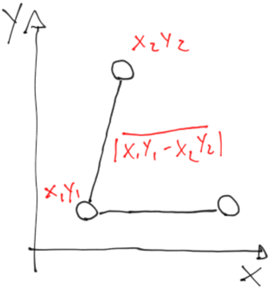
- Embedded Graphs: Vertices and edges have been assigned some sort of coordinates;
  (embedding may be relevant or not!)
- Topological Graphs: They are completely defined by their embedding (edge weight equals vertice distance, etc);

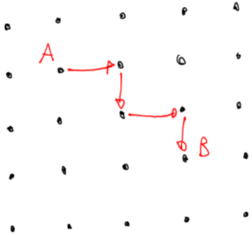




Embedded x Topological (2)

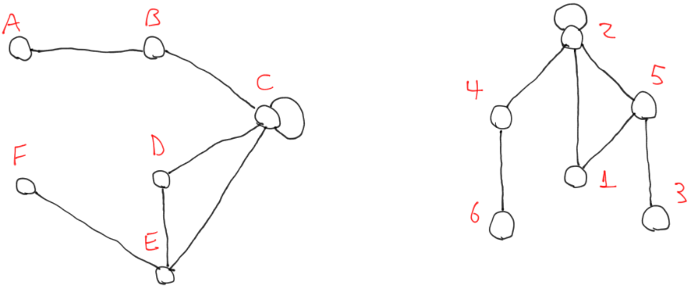
The idea of topological graphs is to help visualize or organize things that are not naturally seen as graphs. For example, maps and paths in a map.





Labelled x Unlabelled (1)

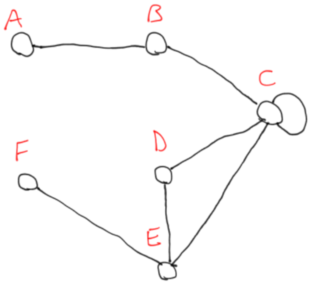
- Labelled: Each different vertex is assigned an unique name;
- Unlabelled graphs don't have such assignments.

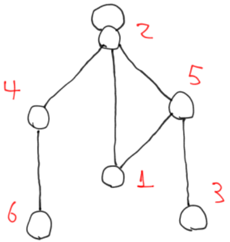


Labelled x Unlabelled (2)

Isomorphism

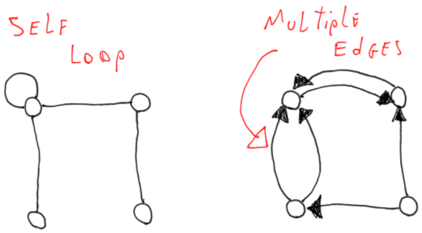
- In “real world” graphs (maps, processes), there is a natural labelling;
- Two graphs with the same structure but different labelling are called Isomorphic
- Detecting isomorphic graphs is an important problem;





Others

- Self Loop: A graph with an edge (x, x), involving only one vertex;
- Multi Edge: The edge (x, y) happens more than once on the graph;
- Implicit: The graph is built as the algorithm progresses, it is not completely defined at the beginning.



Quiz!

How can we represent graphs?

- The characteristics we presented influence in the representation?
- Is there a representation which is good for one type of graph, but not another?

Representing Graphs

Adjacency Matrix

An $n \times n$ matrix where M[i,j] says whether (i,j) is an edge of G or not. Fast access to edges, but has some problems on sparse Graphs.

Adjacency List in Lists

Array of Vertices, with linked lists of neighbors.

Representing Graphs

Adjacency List in Matrices

Simple to program:

- A degree array that lists the number of out-edges connected to each vertex;
- An edge matrix listing the adjacent vertices to each vertex.

```
#define MAXV = 100       // maximum number of Vertices
#define MAXDEGREE = 50 // maximum vertex outdegree

typedef struct{
   int edges[MAXV+1][MAXDEGREE];
   int degree[MAXV+1];
   int nvertices;
   int nedges;
} graph;
```

Representing Graphs

Adjacency List in Matrices

Simple to program:

- A degree array that lists the number of out-edges connected to each vertex;
- An edge matrix listing the adjacent vertices to each vertex.

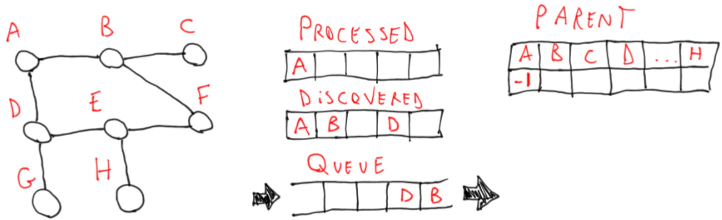
```
insert_edge(graph *g, int x, int y, bool directed)
{
  g->edges[x][g->degree[x]] = y;
  g->degree[x]++
  if (directed == FALSE)
    insert_edge(g,y,x,TRUE)
  else
    g->nedges++;
}
```

Graph Traversal

BFS vs DFS

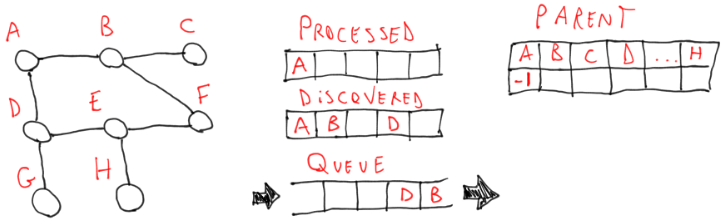
- Breadth First Search: Order of nodes is not important, but we want to find the shortest paths.
- Depth First Search: Easier to find cycles in the graph.
- When is each of these better?

Breadth First Search Algorithm




```
Put "Start" vertex on the "queue" and "discovered";
While queue is not empty:
  v = queue.pop, put v on "processed";
  for each edge in "v", get "k" neighbor to "v":
    if k is not in "discovered":
      put k on the "queue" and "discovered";
      parent of k is v;
```

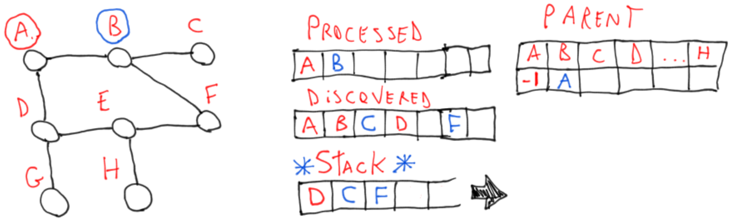
BFS



Finding Paths

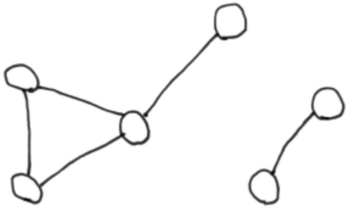
- By following the “parent” array, we can construct the smallest path from any node to the root of the BFS.

Depth First Search



- Implementation: Stack instead of Queue
- Detecting Cycles: Finding a “visited” node while processing a new edge.

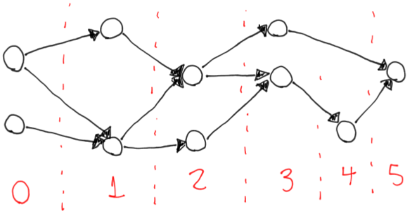
Connected Components



- Connected Components are parts of a graph where there is a path between every vertice of the graph.
- Can be used for detecting invalid positions (in a decision tree graph).
- Can be found by DFS or BFS - any nodes not visited are not connected.

Topological Sorting

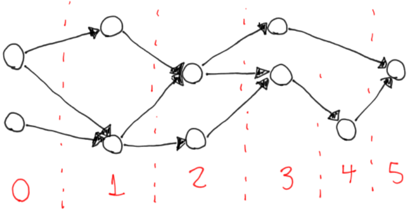
- Requires a Directed Acyclic Graph;
- Can speed up path searches by determining nodes “unrelated” to the search;



Topological Sorting

Topological Sorting Algorithm

```
DAG topsort:
  i = 0
  while there are vertices:
    select all vertices with in-degree 0;
    give them rank "i"; i++;
    remove these vertices and their edges;
```



This Week's Problems

- Bicoloring
- Tourist Guide
- Slash Maze
- From Dusk until Dawn