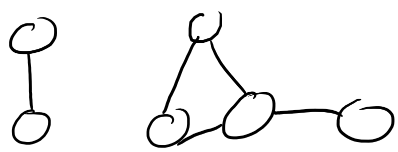
2018-04-05 Graphs Pt. 1

Thursday, April 5, 2018 8:54 AM

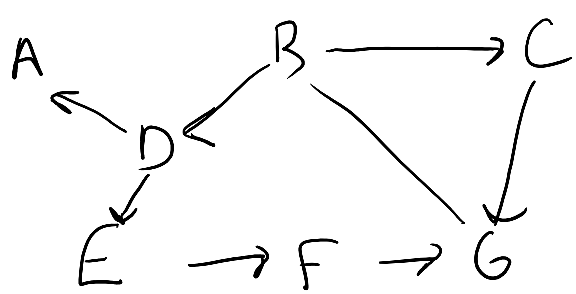
Preliminaries

- Graphs are a "capstone" data structure that often employ several data structures discussed previously in class. Depending on the problem, they may use:
  - Hash tables,
  - Priority queues,
  - Vectors
  - Linked Lists
  - Queues
  - Stacks
- Graphs are just like trees ***except*** that they can have multiple paths between any two nodes (vertices)
- Graphs can contain disconnected segments (not every node is reachable from any other node)
- Example graph:



- Unlike trees, graphs vertices can have from 0, 1, …, |V| paths to each vertex
- All trees are graphs, but not all graphs are trees
- Typically in CS, graph edges are one-way (directional). Graphs with one directional edges are called directed graphs (digraph).
  - This is because pointers can only point to one thing!
- How would we represent a bi-directional edge in CS?
  - With two pointers. A->B; B->A
- By convention, edges without arrows are considered bi-directional

Example Graph



- Unlike a tree, there is no "root" of a graph
- Thus, we need to store the graph as whole (allow access to any vertex in graph immediately)

Vector-based graph implementation (Adjacency Matrix)

- A value of 1 represents connectivity
- Read using row-major order. Rows tell us what the vertex is connected to

	A	B	C	D	E	F	G
A	0	0	0	0	0	0	0
B	0	0	1	1	0	0	1
C	0	0	0	0	0	0	1
D	1	0	0	0	1	0	0
E	0	0	0	0	0	1	0
F	0	0	0	0	0	0	1
G	0	1	0	0	0	0	0

Size = $|V^2|$

- PA2 used an adjacency matrix
- Pros
  - Very nice visual representation
  - Can be a bit more straight forward to work with
- Cons
  - Takes up a lot of space. The only important things to know in the graph are the 1s. The 0s are wasted space.

Linked List Implementation (Edge List)

- Each vertex maintains a list of connected vertices

Vertex	LinkedList<Vertex*> connected_vertices
A	{}
B	D->C->G
C	G
D	A->E
E	F
F	G
G	B

Size = $|E|$

for all graphs $|E| \leq |V^2|$

Choice of which to use based on this

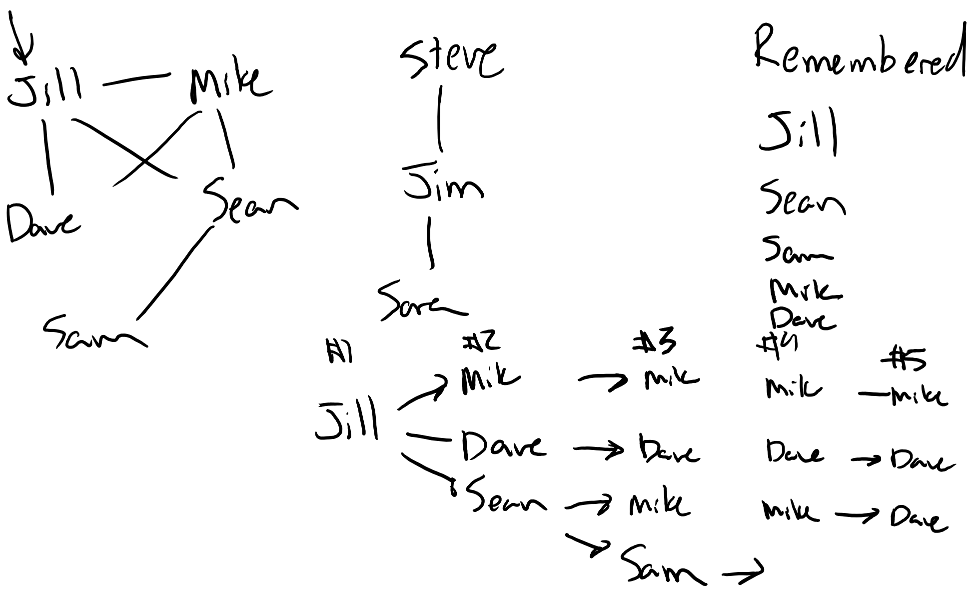
- Pros
  - Takes up less space when the graph is sparse (not a lot of edges in the graph)
  - Can be nicer in recursive situations
- Cons
  - Overall picture of graph is less clear

For PA #5, our graph is represented in two parts. For the actors:
unordered_map<string, Actor*>
For movies:
unordered_map<string, Movie*>

Graph Traversals

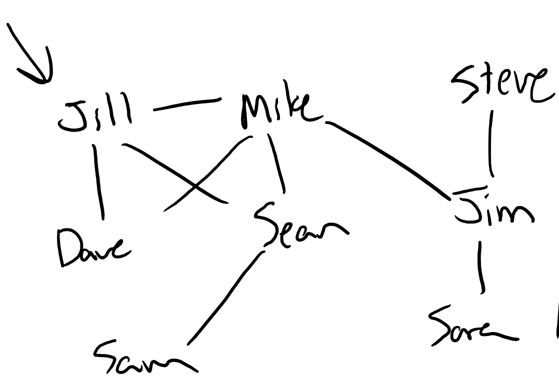
- Given this social graph:

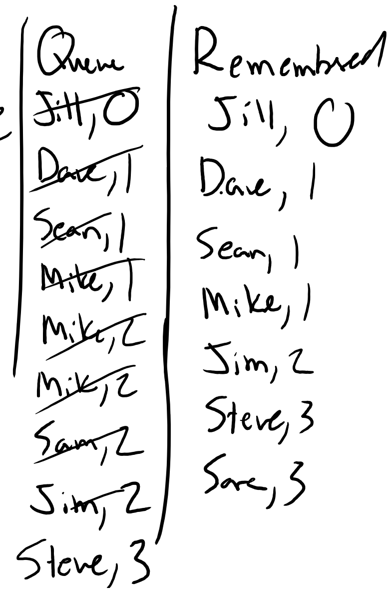
- We might want to ask whether or not Jill and Sara are in the same social circle.
- When answering this question, we must keep track of visited vertices
- Algorithm:
    1. Pick some starting location (e.g. Jill)
    2. Put starting vertex into a list of vertices called ***to_visit***
    3. While ***to_visit*** is not empty:
        i. Remove ***item*** from ***to_visit***. For each vertex (***v***) in ***item***.
            1) If ***v*** not seen before, add to ***to_visit***
        ii. Remember that we've seen ***item***
    4. If target is not in our list of seen items, they are not connected



Degrees of separation

- How many degrees of separation are there between Jill and Sara?
    - I.e. how many edges are there between Jill and Sara
- Uses BFS, Requires an additional counter. Instead of remembering just a name, we must remember a distance as well





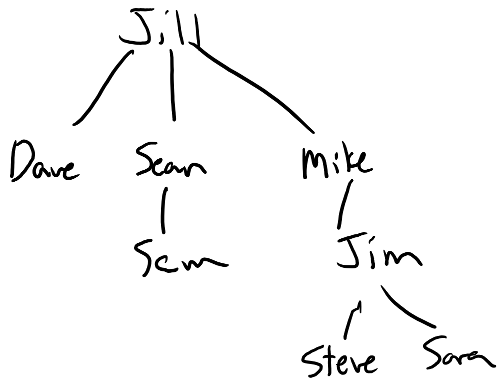
- This process creates a tree, which can

Jim, 2

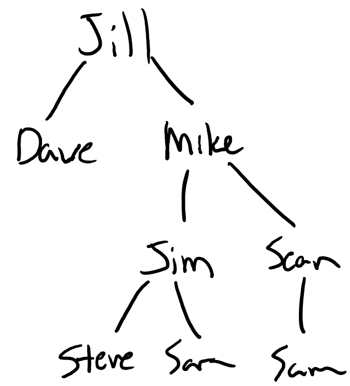
Steve, 3

Sara, 3

- This process creates a tree, which can then be traversed. Not required for PA#5, but it might be helpful (Adam did not do this).



- Trees can also be built from a DFS



DFS trees allow us to find weak points in a graph (articulation points).

- In social graph, if one person were to go away (die), who would no longer be friends?