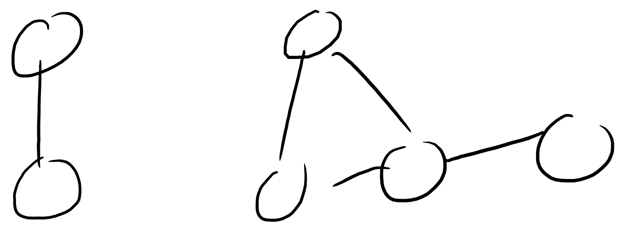
2018-10-30 Graphs 1

Tuesday, October 30, 2018 2:58 PM

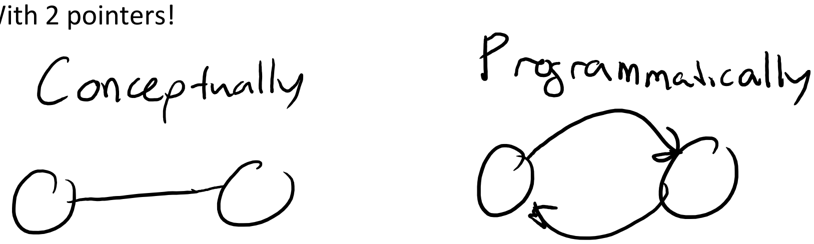
Graph Preliminaries

- Graphs are a "capstone" data structure in that employ many of the data structures discussed in class. Depending on the problem, we may use:
    - Hash tables
    - Priority Queues
    - Vectors
    - Linked Lists
    - Queues
    - Stacks
- Graphs are trees that allow for multiple paths to the same node
    - A node in a graph is also called a vertex
- Unlike trees, graphs can also contain disconnected segments (not every node is reachable from every other node)
- Example that exemplifies both of these properties:

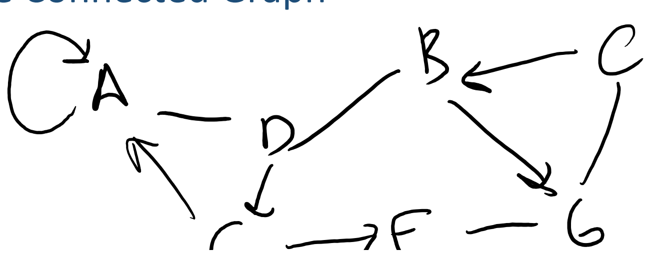


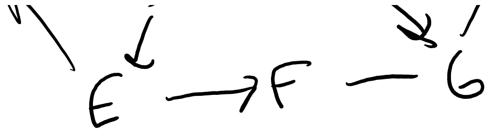
- All trees are graphs, but not all graphs are trees
- In computer, graph edges are unidirectional (go only one way)
    - Graphs with unidirectional edges are called directed graphs

- We use directional edges in CS because pointers can only point one direction
- How would we represent a bidirectional edge in code?
    - With 2 pointers!



Example Connected Graph





- Unlike a tree, a graph does not a clear "root"
- Instead, we use structures that give us immediate access to the graph as a whole

Adjacency Matrix: Vector-based implementation

- Row-major order matrix (2D array). Rows indicate connectivity to other graph nodes.
- A value of 1 in a cell represents connectivity

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

- A lot of online examples use adjacency matrix
  - I'm not sure why. Maybe because they require less up-front design (e.g. custom classes)
- Adjacency matrices tend to take up more memory on sparse graphs (lots of zeros)
  - Requires V^2 memory (V = vertices)

Edge List: Linked-List representation

- Adam tends not to use a LL, but instead HTs

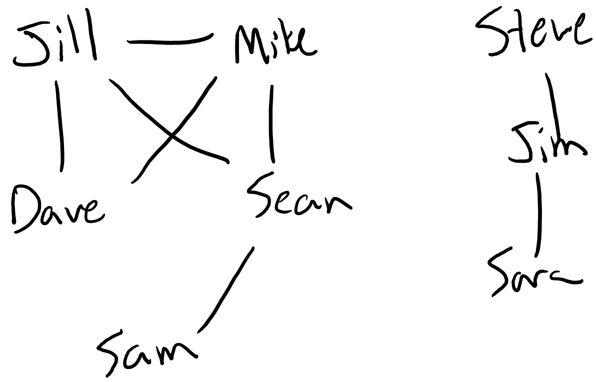
Vertex	Collection<Vertex*>
A	{A, D}
B	{D, G}
C	{B, G}
D	{A, B, E}
E	{A, F}
F	{G}
G	{C, F}

- Tends to take up less space when the graph is sparse
- In HW, we will probably use Edge List representation

Graph Traversals

- Move through the graph trying to answer a question

- Example: Given the following social graph:



  - Might we expect Jill and Sara to go to the same party?

Jill	{Dave, Mike, Sean}
Dave	{Jill, Mike}
Mike	{Jill, Dave, Sean}
Sean	{Mike, Jill, Sam}
Steve	{Jim}
Jim	{Steve, Sara}
Sara	{Jim}
Sam	{Sean}

```
function search(person, target):
For each of person's friends:
    If target is friend, return found
    Else, return search(friend, target)
```

Search(Jill, Sara)
Search(Dave, Sara)
Search(Jill, Sara)
Search(Dave, Sara)

Unlike tree search, graph search ***requires*** you maintain a list of visited nodes! Otherwise, you end up in an infinite loop.

Version 2 search:

```
function search(person, target, visited_list):
For each of person's friends:
    If target is friend, return found
    Else, if visited_list[friend] == null
        Mark friend as visited
        return search(friend, target)
```

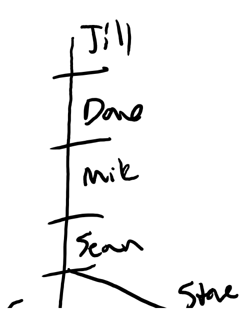
O(V)

```
Search(Jill, Sara)
    Search(Dave, Sara)
        Search(Mike, Sara)
            Search(Sean, Sara)
                Search(Sam, Sara)
                {none}
            {none}
        {none}
    {none}
Is Sara visited?  NO!
```

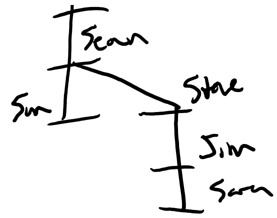
What if Jill and Sara are connected?

Jill	{Dave, Mike, Sean}
Dave	{Jill, Mike}
Mike	{Jill, Dave, Sean}
Sean	{Mike, Jill, Sam, Steve}
Steve	{Jim, Sean}
Jim	{Steve, Sara}
Sara	{Jim}
Sam	{Sean}

```
Search(Jill, Sara)
    Search(Dave, Sara)
        Search(Mike, Sara)
            Search(Sean, Sara)
                Search(Sam, Sara)
                    {none}
                Search(Steve, Sara)
                    Search(Jim, Sara)
                        Search(Sara, Sara)
                            Found!
```



Sam	{Sean}



Found!

- How many degrees of separation exist between Jill and Jim?

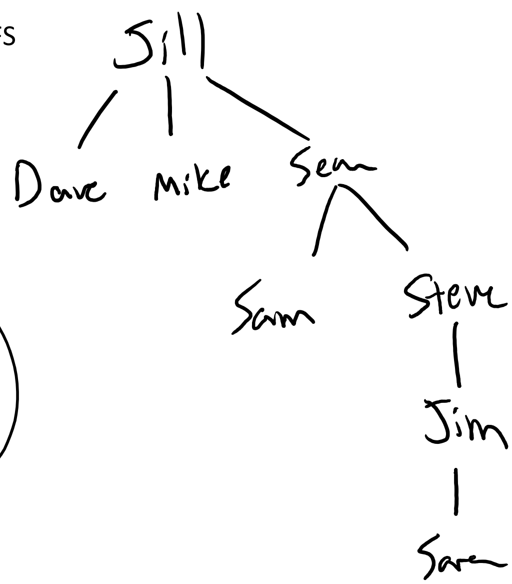
Jill	{Dave, Mike, Sean}
Dave	{Jill, Mike}
Mike	{Jill, Dave, Sean}
Sean	{Mike, Jill, Sam, Steve}
Steve	{Jim, Sean}
Jim	{Steve, Sara}
Sara	{Jim}
Sam	{Sean}

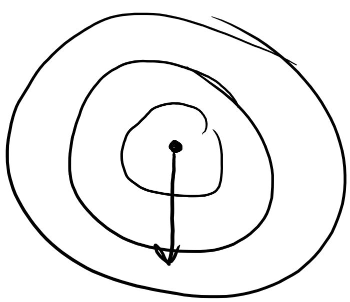
```
function search(origin, target):
    Queue to_visit{<0, origin>};
    HT visited{};
    While to_visit is not empty:
        Person = to_visit.pop();
        For each of person's friends:
            If target is friend, return found
            Else, if visited_list[friend] == null
                To_visit.push({person.first + 1, friend})
            Mark friend as visited
```

O(V)

<0,Jill>	<1, Dave>	<1, Mike>	<1, Sean>	<2, Sam>	<2, Steve>	<3, Jim>
3	0	0	2	0	1	1

- Visualizing BFS





Depth-first vs Breadth-first search

- Both have the same runtime of O(V)
- Choice is wholly dependent on the algorithm that you're wanting to employ
- If the algorithm doesn't care, instead consider the structure of the graph and the nature of the problem that you're trying to solve.
    - Question to ask: Where is my answer likely to exist?
        - If the answer is likely nearby, breadth-first may be a better choice
        - If the answer is likely far away, depth-first search may be a better choice
- Consider a maze

- ○ DFS might be faster on average