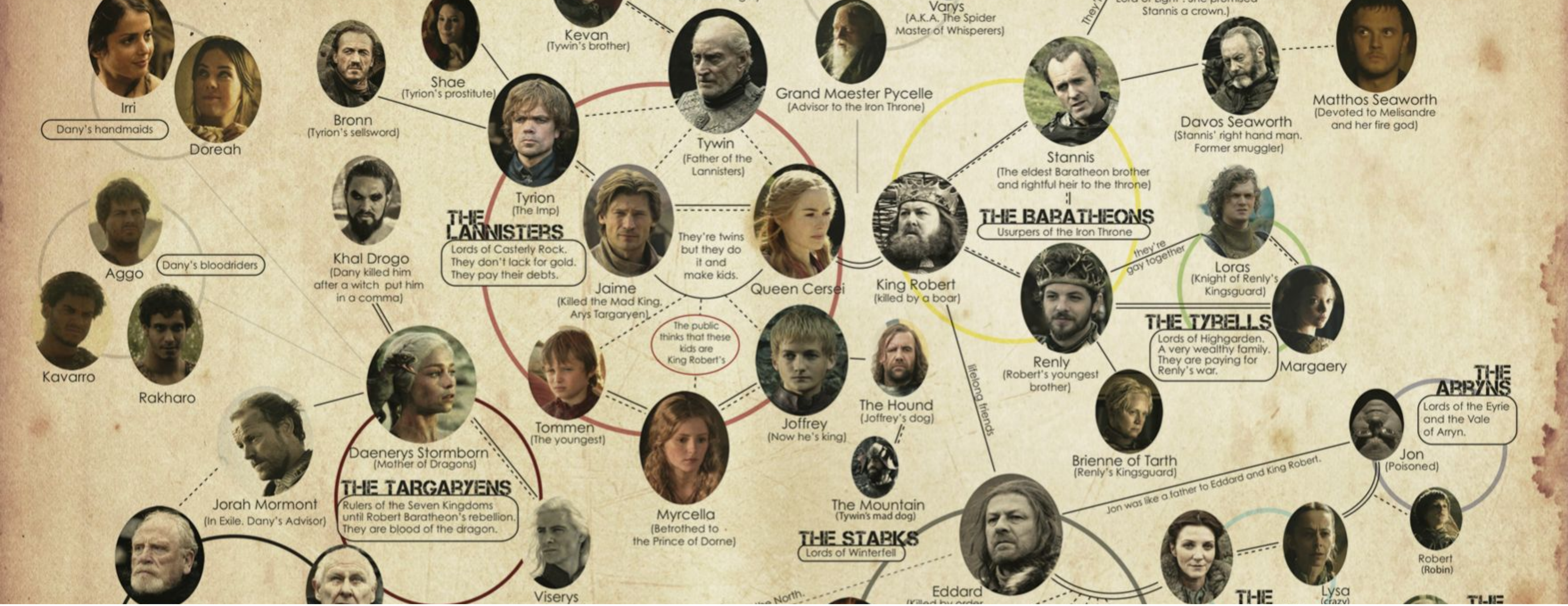


Lecture 20: Implementing Graphs

CSE 373: Data Structures and Algorithms

https://visualdatahub.wordpress.com/2013/04/02/a-web-of-thrones-character-relationships/

Administrivia

HW 5 pt 1 due tomorrow

HW 5 pt 2 out today

More Kasey office hours!

Introduction to Graphs

Inter-data Relationships

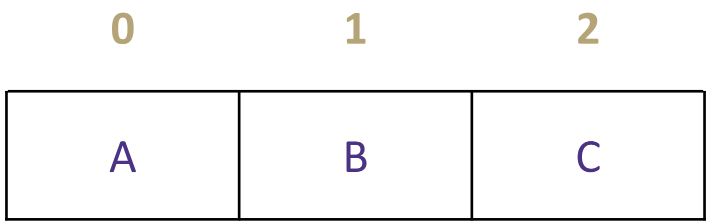
Arrays

Categorically associated

Sometimes ordered

Typically independent

Elements only store pure data, no connection info



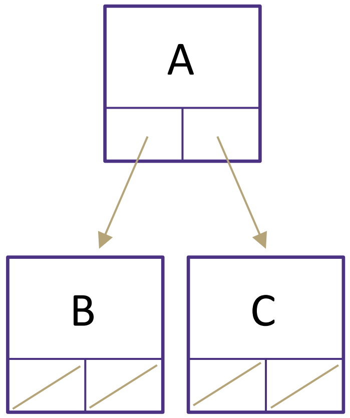
Trees

Directional Relationships

Ordered for easy access

Limited connections

Elements store data and connection info



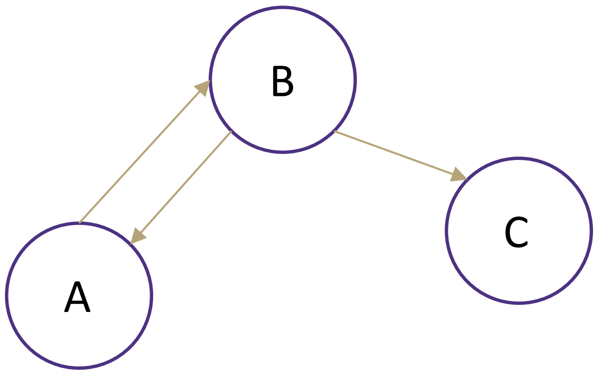
Graphs

Multiple relationship connections

Relationships dictate structure

Connection freedom!

Both elements and connections can store data



Graph: Formal Definition

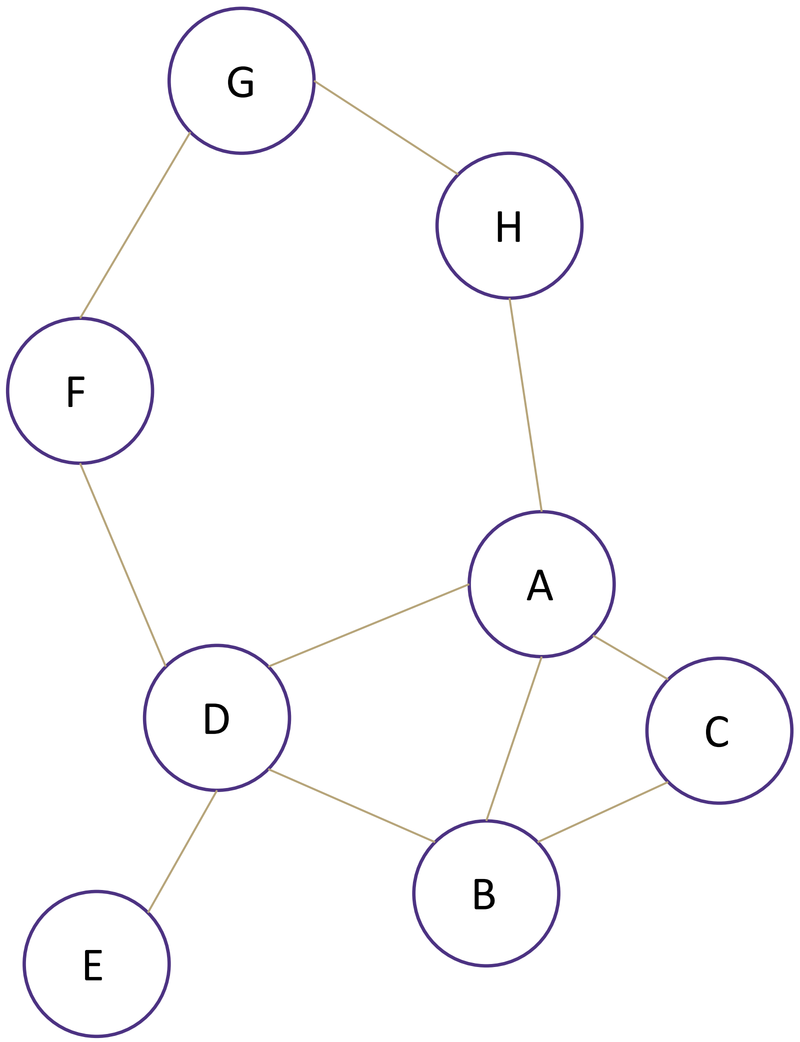
A **graph** is defined by a pair of sets G = (V, E) where…

- V is a set of **vertices**
  - A vertex or “node” is a data entity

  V = { A, B, C, D, E, F, G, H }

- E is a set of **edges**
  - An edge is a connection between two vertices

  E = { (A, B), (A, C), (A, D), (A, H),
  (C, B), (B, D), (D, E), (D, F),
  (F, G), (G, H)}



Applications

Physical Maps
- Airline maps
  - Vertices are airports, edges are flight paths
- Traffic
  - Vertices are addresses, edges are streets

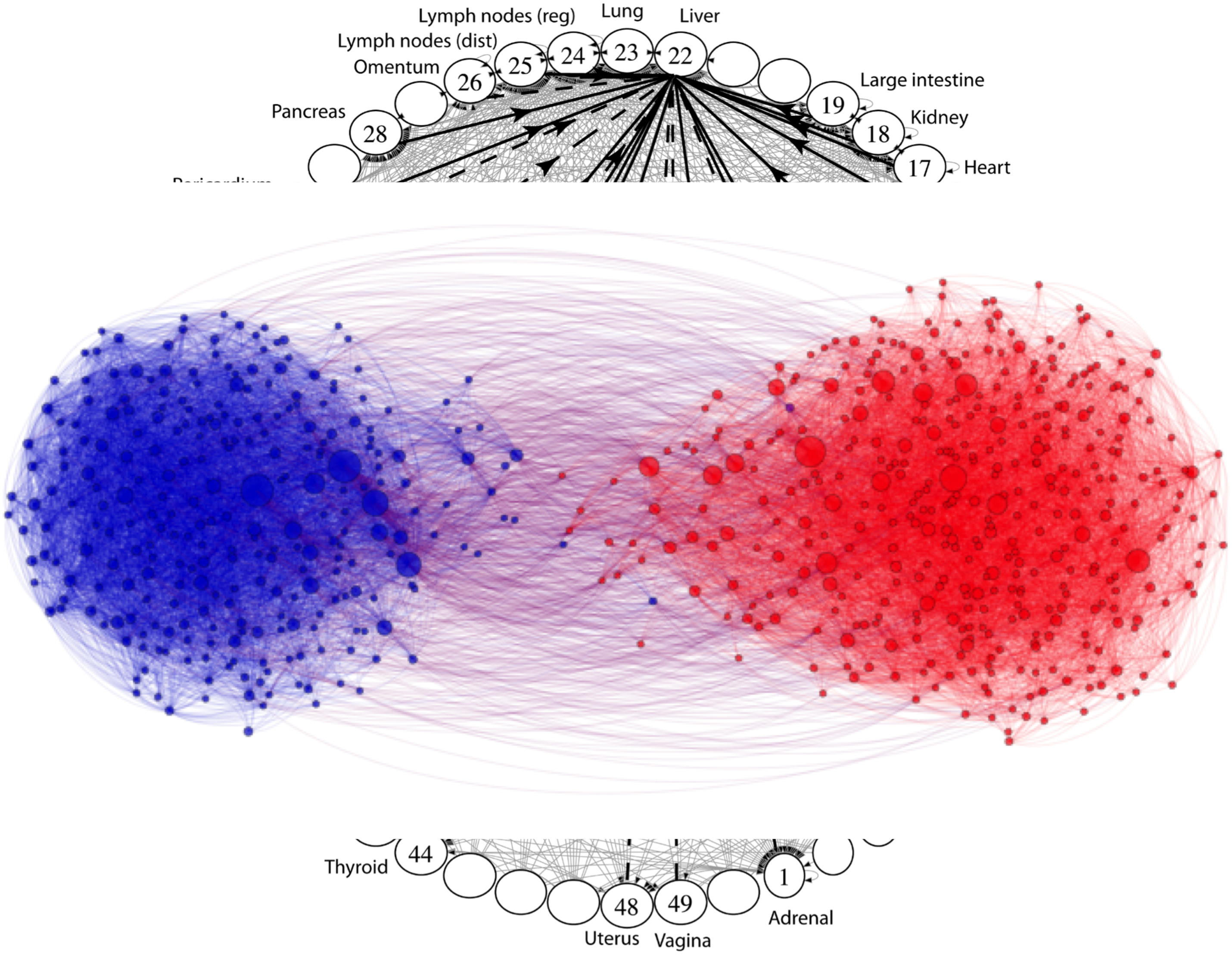
Relationships
- Social media graphs
  - Vertices are accounts, edges are follower relationships
- Code bases
  - Vertices are classes, edges are usage

Influence
- Biology
  - Vertices are cancer cell destinations, edges are migration paths

Related topics
- Web Page Ranking
  - Vertices are web pages, edges are hyperlinks
- Wikipedia
  - Vertices are articles, edges are links

SO MANY MORREEEE

www.allthingsgraphed.com



Graph Vocabulary

Graph Direction

- **Undirected graph** – edges have no direction and are two-way

  V = { Dany, Drogo, Jon }

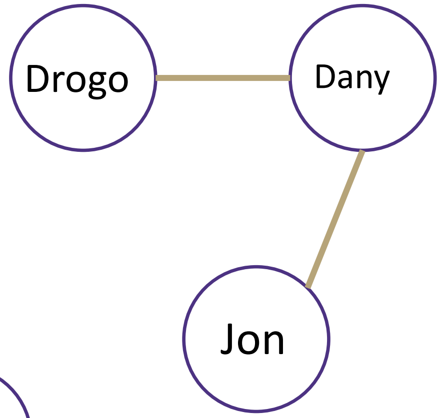
  E = { (Dany, Drogo), (Dany, Jon) } *inferred (Drogo, Dany) and (Jon, Dany)*

- **Directed graphs** – edges have direction and are thus one-way

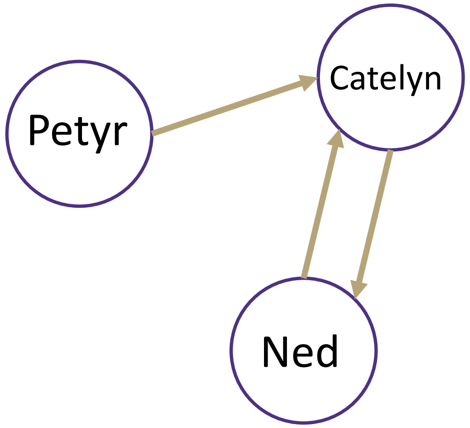
  V = { Petyr, Catelyn, Ned }

  E = { (Petyr, Catelyn), (Catelyn, Ned), (Ned, Catelyn) }

Undirected Graph:



Directed Graph:



Degree of a Vertex

- **Degree** – the number of edges connected to that vertex

  Drogo : 1, Danny : 1, Jon : 1

- **In-degree** – the number of directed edges that point to a vertex

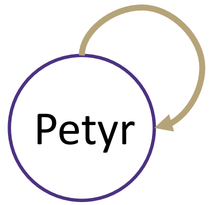
  Petyr : 0, Catelyn : 2, Ned : 1

- **Out-degree** – the number of directed edges that start at a vertex

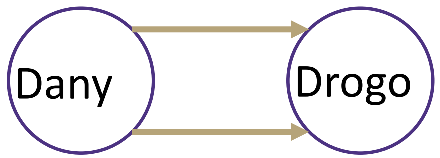
  Petyr : 1, Catelyn : 1, Ned : 1

Graph Vocabulary

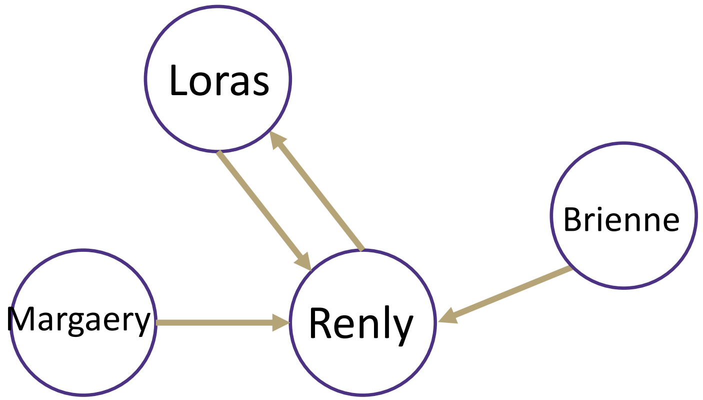
Self loop – an edge that starts and ends at the same vertex



Parallel edges – two edges with the same start and end vertices



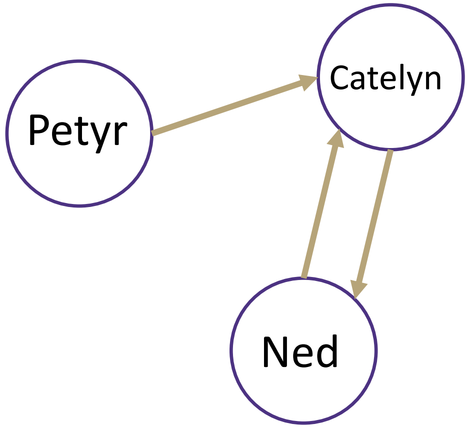
Simple graph – a graph with no self-loops and no parallel edges



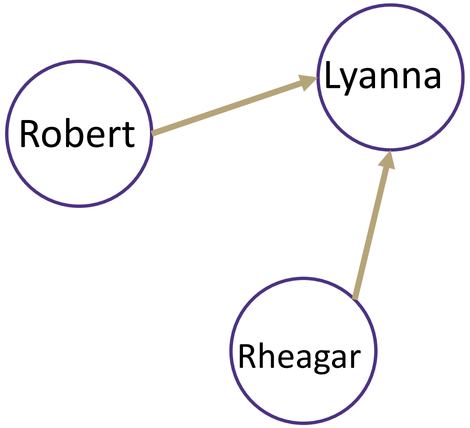
Food for thought

Is a graph valid if there exists a vertex with a degree of 0?

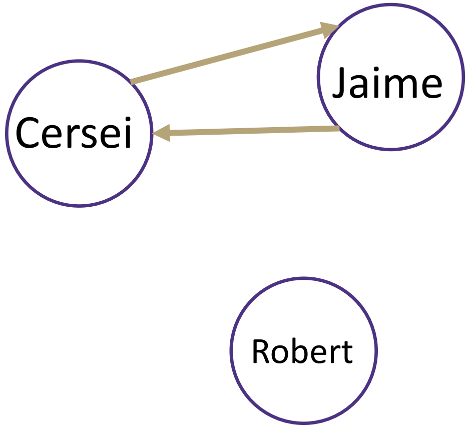
Yes



Petyr has an “in degree” of 0

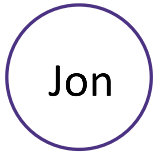


Lyanna has an “out degree” of 0



Robert has both an “in degree” and an “out degree” of 0

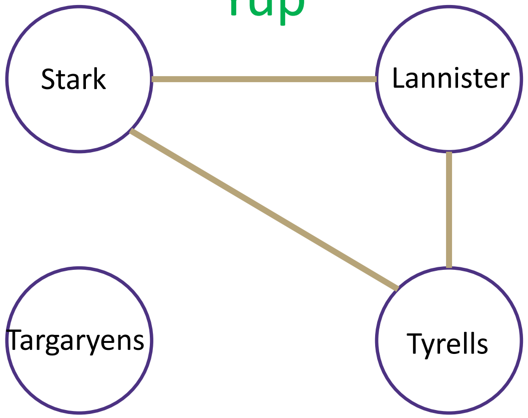
Is this a valid graph?

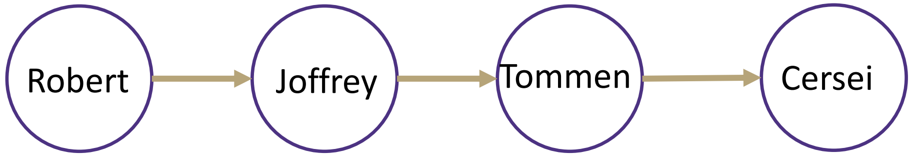


Yes!

Are these valid?

Yup





Sure

Implementing a Graph

Implement with nodes…

Implementation gets super messy

What if you wanted a vertex without an edge?

How can we implement without requiring edges to access nodes?

Implement using some of our existing data structures!

Adjacency Matrix

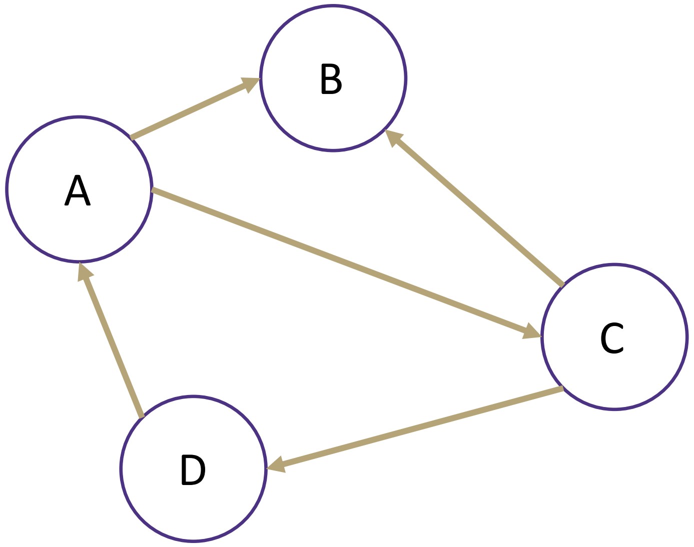
Assign each vertex a number from 0 to V – 1

Create a V x V array of Booleans

If (x,y) ∈ E then arr[x][y] = true

	A	B	C	D
A		T	T	
B				
C		T		T
D	T			

Runtime (in terms of V and E)
- get out - edges for a vertex O(v)
- get in – edges for a vertex O(v)
- decide if an edge exists O(1)
- insert an edge O(1)
- delete an edge O(1)
- delete a vertex (subject to implementation)
- add a vertex (subject to implementation)

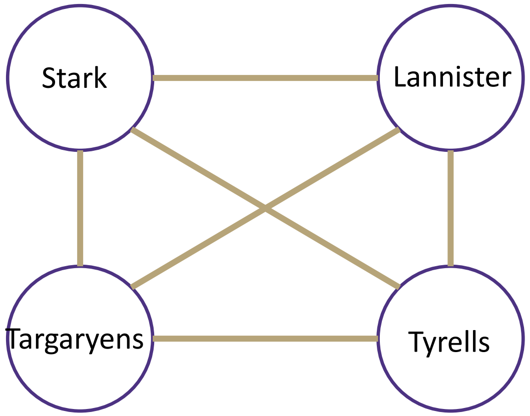


How much space is used?
V^2

Graph Vocabulary

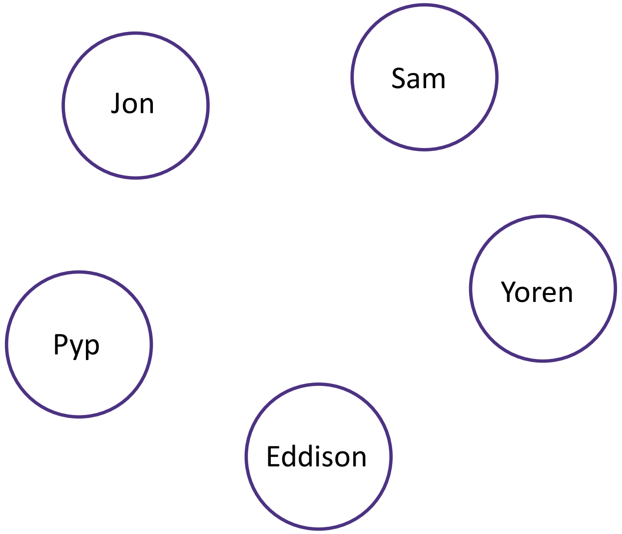
Dense Graph – a graph with a lot of edges

$E \in \Theta(V^2)$



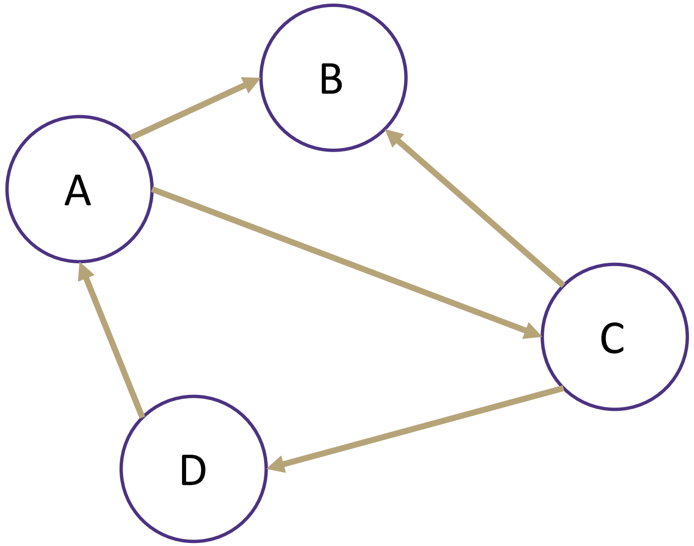
Sparse Graph – a graph with “few” edges

$E \in \Theta(V)$



An Adjacency Matrix seems a waste for a sparse graph…

Adjacency List



Create a Dictionary of size V from type V to Collection of E

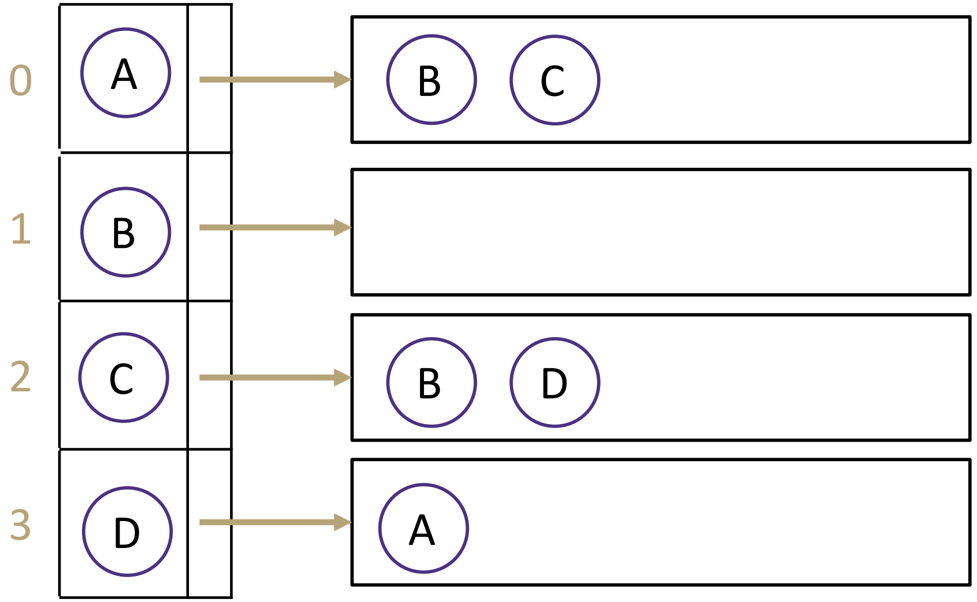
If (x,y) ∈ E then add y to the set associated with the key x

Runtime (in terms of V and E)

- get out - edges for a vertex O(1)
- get in - edges for a vertex O(V + E)
- decide if an edge exists O(1)
- insert an edge O(1)
- delete an edge O(1)
- delete a vertex (subject to implementation)
- add a vertex (subject to implementation)

How much space is used?

V + E

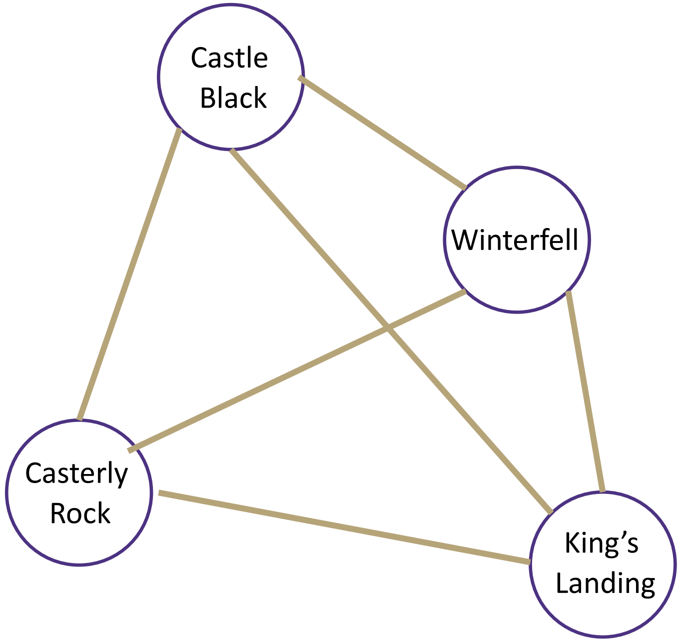


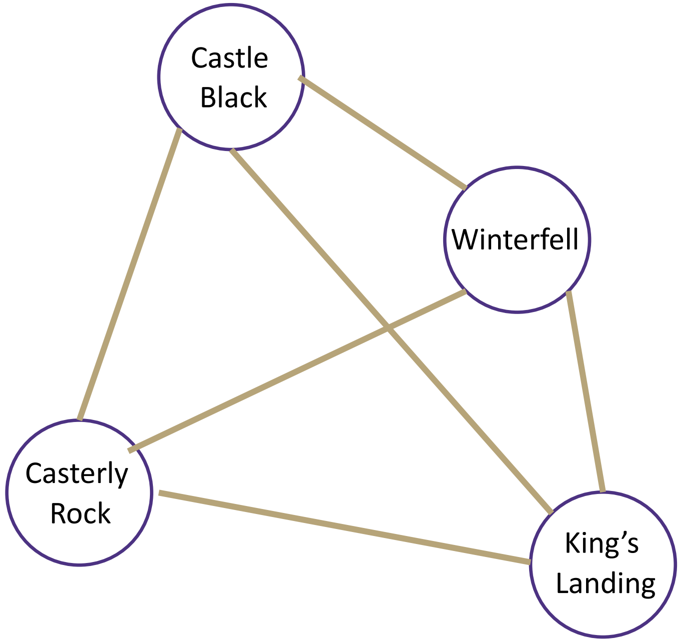
Walks and Paths

Walk – continuous set of edges leading from vertex to vertex

A list of vertices where if I is some int where 0 < 1 < Vn every pair (Vi, Vi+1) in E is true

Path – a walk that never visits the same vertex twice





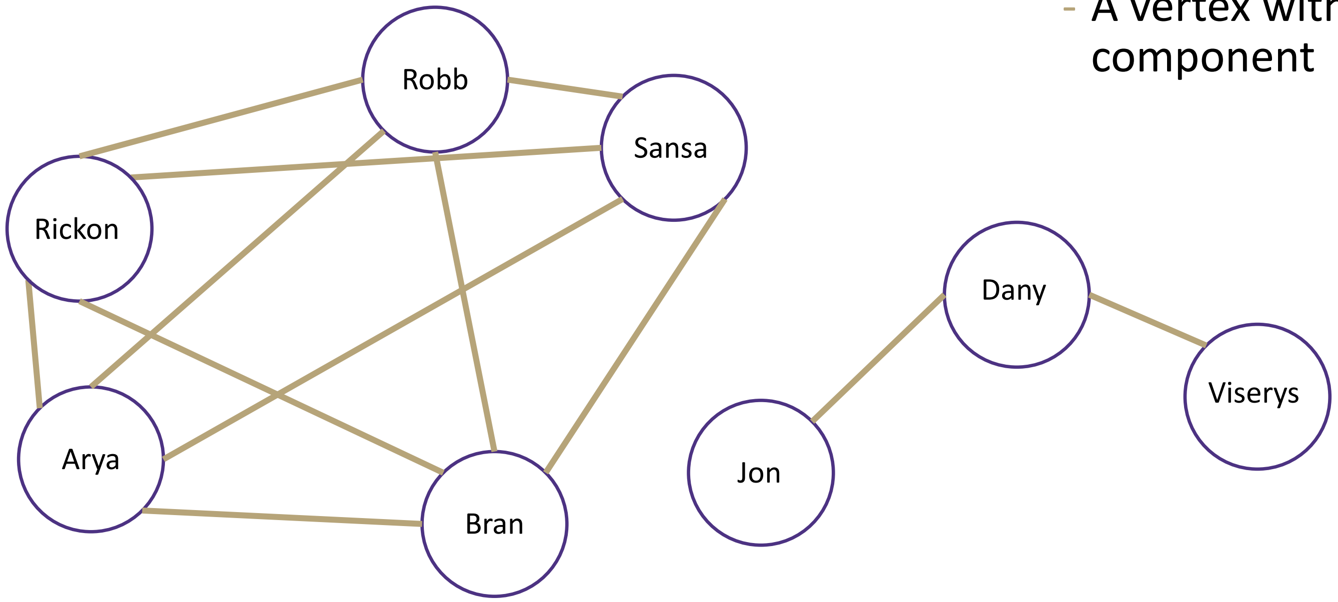
Connected Graphs

Connected graph – a graph where every vertex is connected to every other vertex via some path. It is not required for every vertex to have an edge to every other vertex

There exists some way to get from each vertex to every other vertex

Connected Component – a *subgraph* in which any two vertices are connected via some path, but is connected to no additional vertices in the *supergraph*

- There exists some way to get from each vertex within the connected component to every other vertex in the connected component
- A vertex with no edges is itself a connected component



Graph Algorithms

Traversing a Graph

In all previous data structures:

1. Start at first element
2. Move to next element
3. Repeat until end of elements

For graphs – Where do we start? How do we decide where to go next? When do we end?

1. Pick any vertex to start, mark it “visited”
2. Put all neighbors of first vertex in a “to be visited” collection
3. Move onto next vertex in “to be visited” collection
4. Mark vertex “visited”
5. Put all unvisited neighbors in “to be visited”
6. Move onto next vertex in “to be visited” collection
7. Repeat…

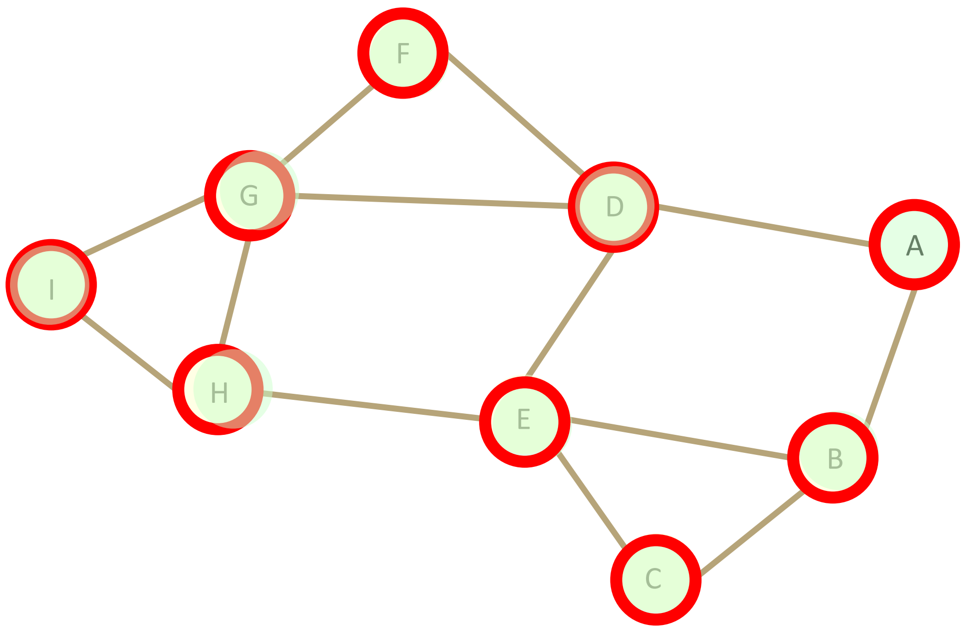
Breadth First Search

```
search(graph)
   toVisit.enqueue(first vertex)
   while(toVisit is not empty)
      current = toVisit.dequeue()
      for (v : current.neighbors())
         if (v is not in visited)
            toVisit.enqueue(v)
      visited.add(current)
```

Current node: I

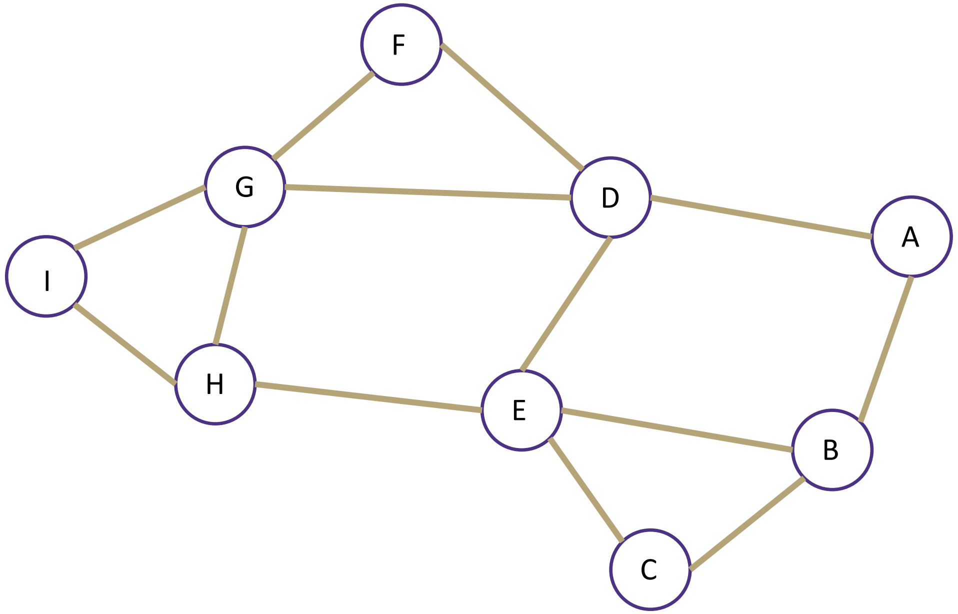
Queue: B D E C F G H I

Visited: A B D E C F G H I



Breadth First Search Analysis

```
search(graph)
   toVisit.enqueue(first vertex)
   while(toVisit is not empty)
      current = toVisit.dequeue()
      for (v : current.neighbors())
         if (v is not in visited)
            toVisit.enqueue(v)
      visited.add(current)
```



Visited: A B D E C F G H I

How many times do you visit each node? 1 time each

How many times do you traverse each edge? Max 2 times each
- Putting them into toVisit
- Checking if they're in toVisit

Runtime? O(V + 2E) = O(V + E) "graph linear"

Depth First Search (DFS)

BFS uses a queue to order which vertex we move to next

Gives us a growing “frontier” movement across graph

Can you move in a different pattern? Can you use a different data structure?

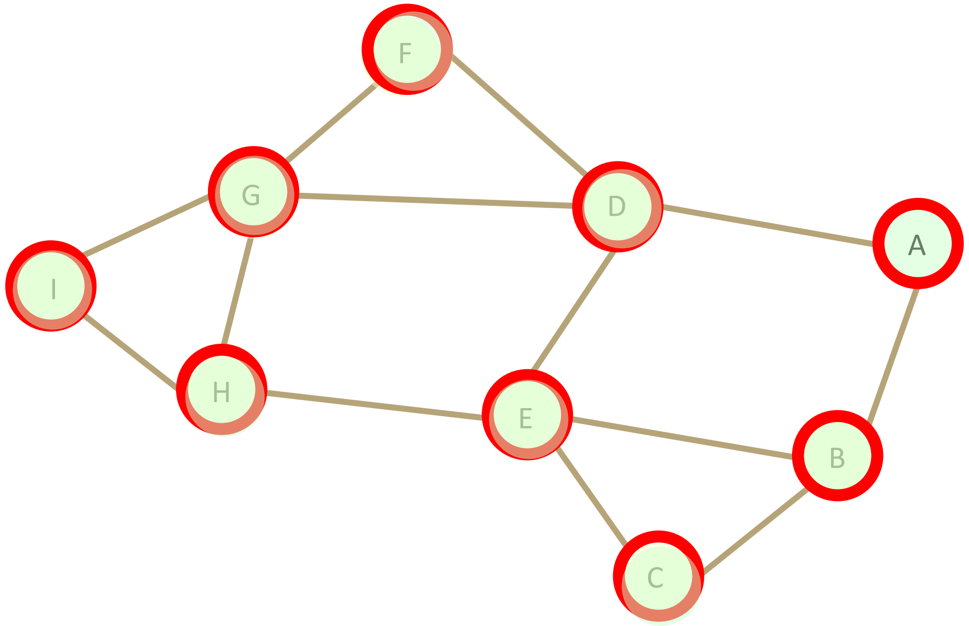
What if you used a stack instead?

```
bfs(graph)
   toVisit.enqueue(first vertex)
   while(toVisit is not empty)
      current = toVisit.dequeue()
      for (V : current.neighbors())
         if (V is not in visited)
            toVisit.enqueue(v)
      visited.add(current)
```

```
dfs(graph)
   toVisit.push(first vertex)
   while(toVisit is not empty)
      current = toVisit.pop()
      for (V : current.neighbors())
         if (V is not in visited)
            toVisit.push(v)
      visited.add(current)
```

Depth First Search

```
dfs(graph)
   toVisit.push(first vertex)
   while(toVisit is not empty)
      current = toVisit.pop()
      for (V : current.neighbors())
         if (V is not in stack)
            toVisit.push(v)
      visited.add(current)
```



Current node: D

Stack: D CB EI HGF

Visited: A B E H G F I C D

How many times do you visit each node? 1 time each

How many times do you traverse each edge? Max 2 times each

- Putting them into toVisit
- Checking if they're in toVisit

Runtime? O(V + 2E) = O(V + E) "graph linear"