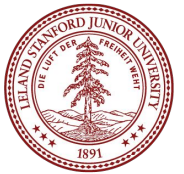
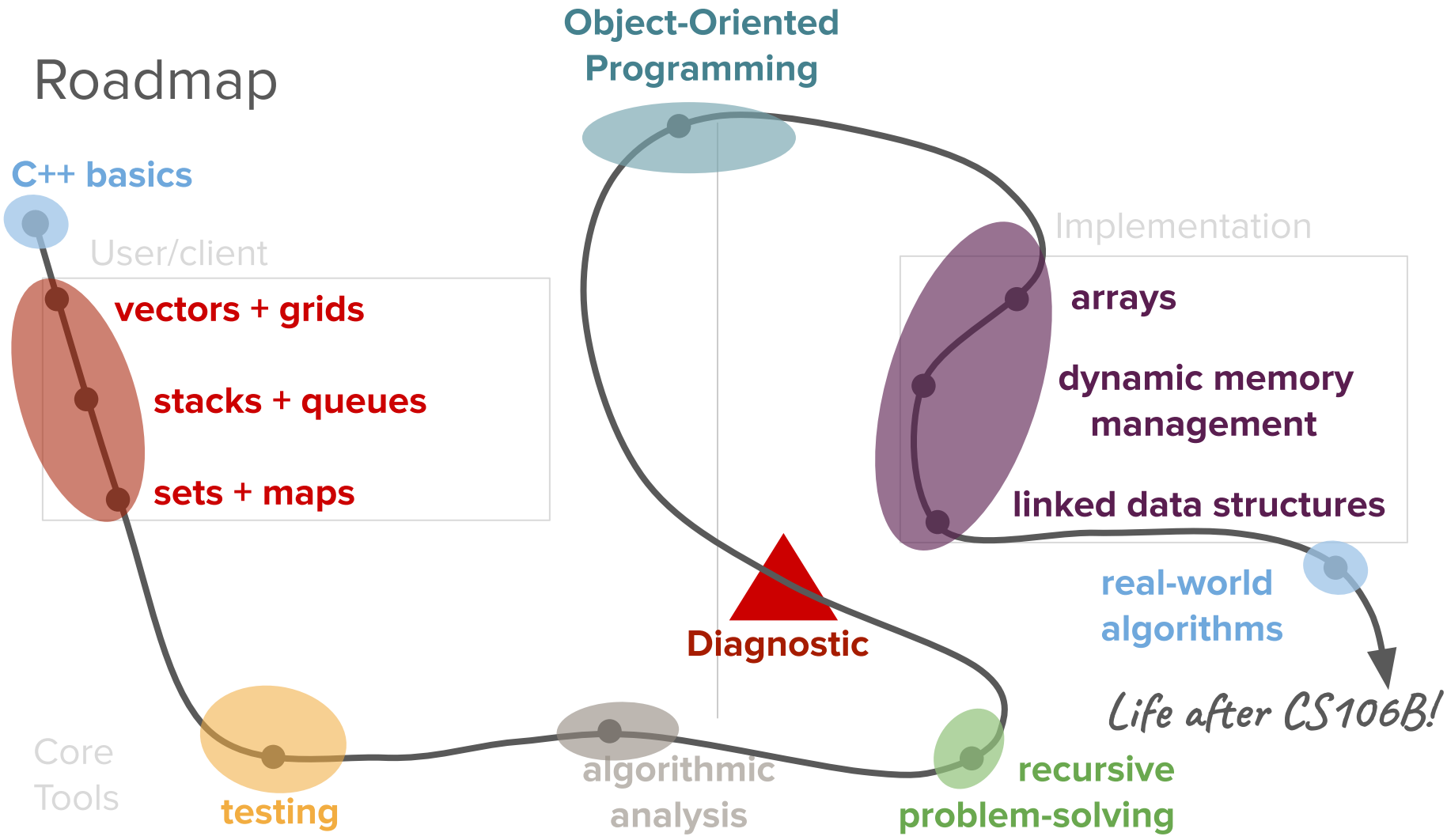
Graphs and Graph Algorithms

What was your favorite part about working on your final project?

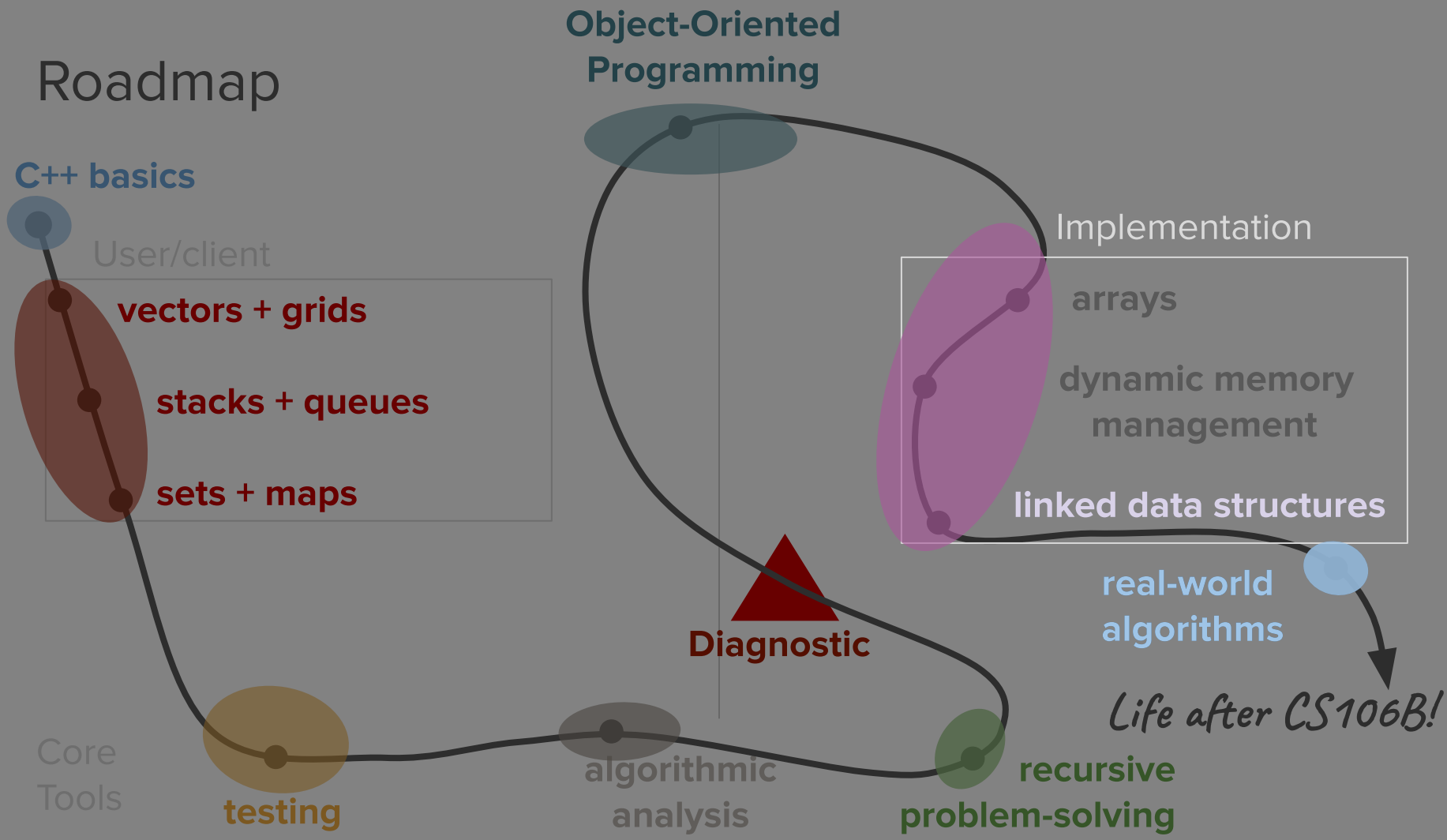
(put your answers the chat)



Roadmap


Object-Oriented Programming
C++ basics
User/client
vectors + grids
stacks + queues
sets + maps
Implementation
arrays
dynamic memory management
linked data structures
real-world algorithms
Diagnostic
Life after CS106B!
Core Tools
testing
algorithmic analysis
recursive problem-solving

Roadmap


Object-Oriented Programming
C++ basics
User/client
vectors + grids
stacks + queues
sets + maps
Implementation
arrays
dynamic memory management
linked data structures
real-world algorithms
Life after CS106B!
Diagnostic
Core Tools
testing
algorithmic analysis
recursive problem-solving

Today's question

How can we represent and organize complex systems of interconnected components?

Today's topics

1. Motivation for Graphs
2. Graph Definition and Terminology
3. Graph Algorithms (BFS, Dijkstra and A*)

Week 8 overview

Week 8

- There is no section this week!
- Today and Tomorrow: Lectures on "fun" topics to prepare you for the real world
  - Today: Graphs and Graph Algorithms
  - Tomorrow: Multithreading and Parallel Computing (Trip)
- Wednesday: Class Wrap-up and "Life after CS106B" Lecture
  - We'll be having an "Ask Us Anything" component. Submit your questions in advance here!
- Thursday: No class! Use the time to prep final project presentations.
- Thursday-Sunday: Final project presentations. Make sure to sign up for a slot!

Lecture Tomorrow

- Trip will be guest lecturing on a topic near and dear to his heart tomorrow (multithreading and parallel computing). It should be an awesome lecture!

- Unfortunately, due to university restrictions, we cannot have minors (< 18 years old) join the Zoom meeting for tomorrow's lecture.

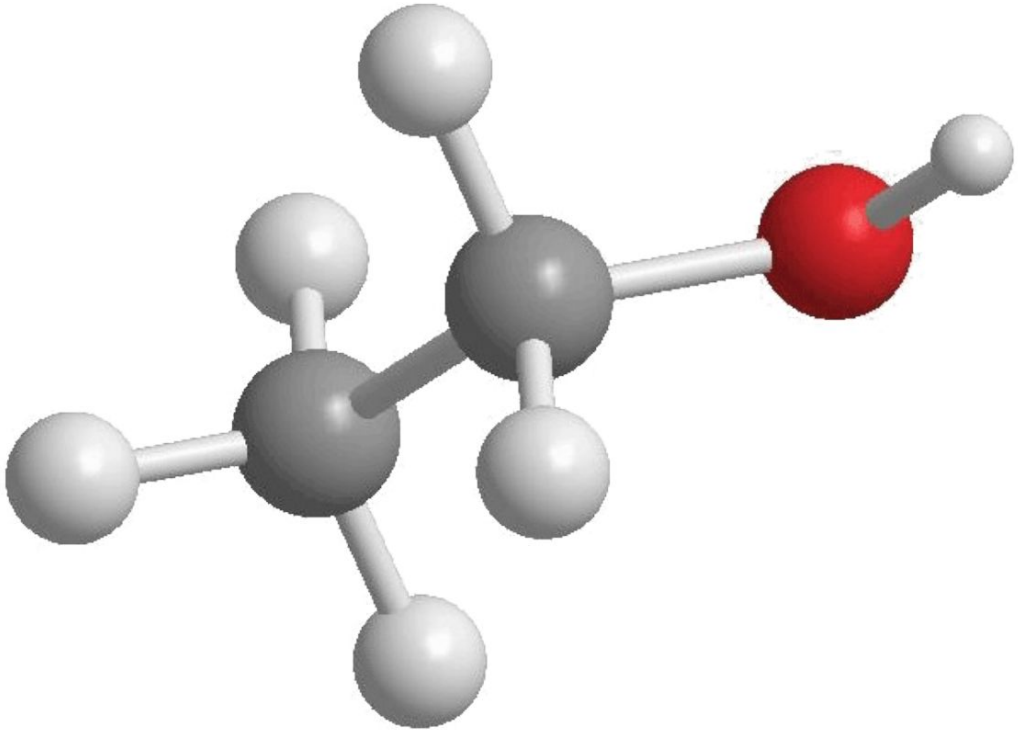
- However, we still want you all to be able to watch and participate live!
  - We will be live-streaming the lecture on YouTube Live.
  - There will be a pinned Ed post that can be used to ask live questions that Kylie/Nick will moderate and deliver to Trip.
  - Links and more information will be posted tomorrow morning.
  - As always, the lecture will also be recorded for later viewing.

How can we represent and organize complex systems of interconnected components?

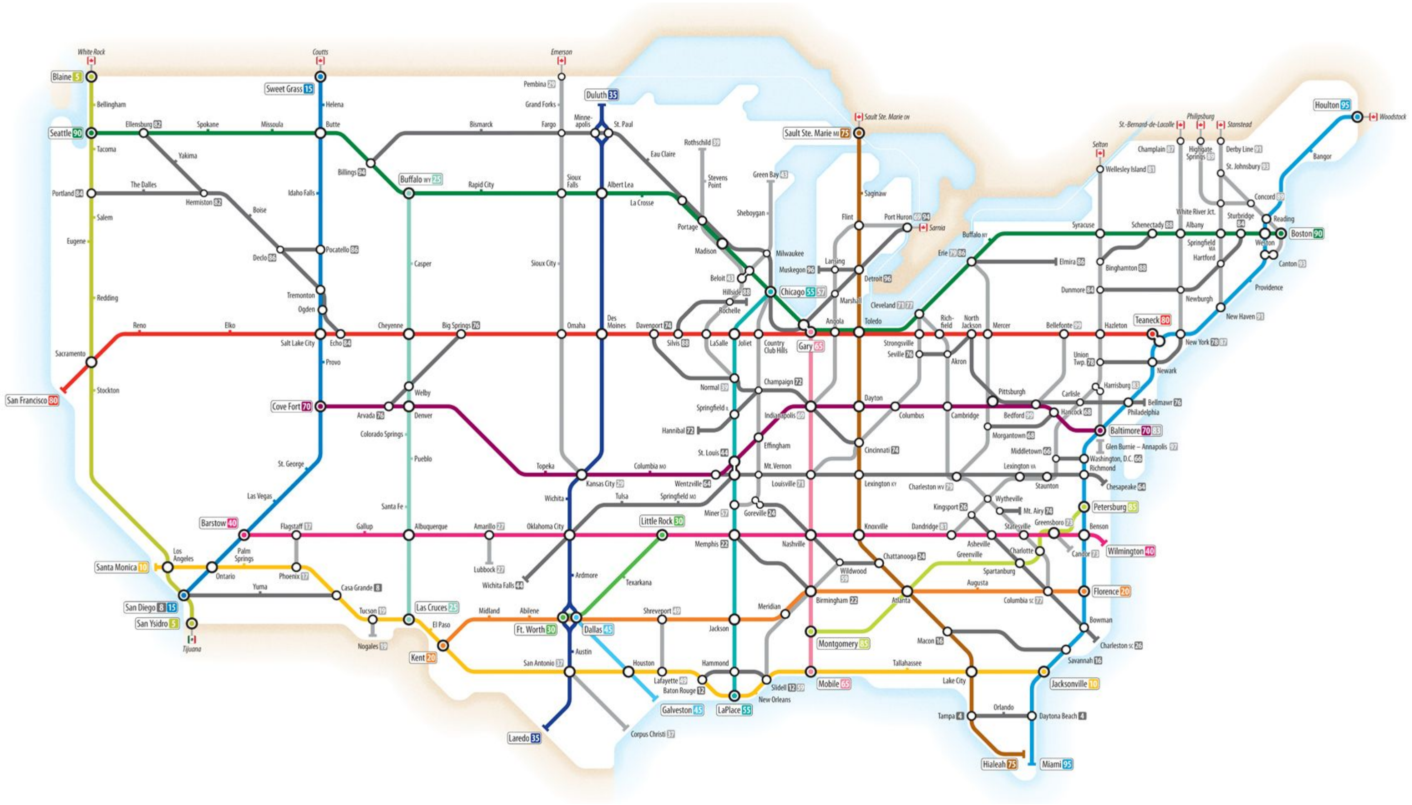
Graphs

Social Networks

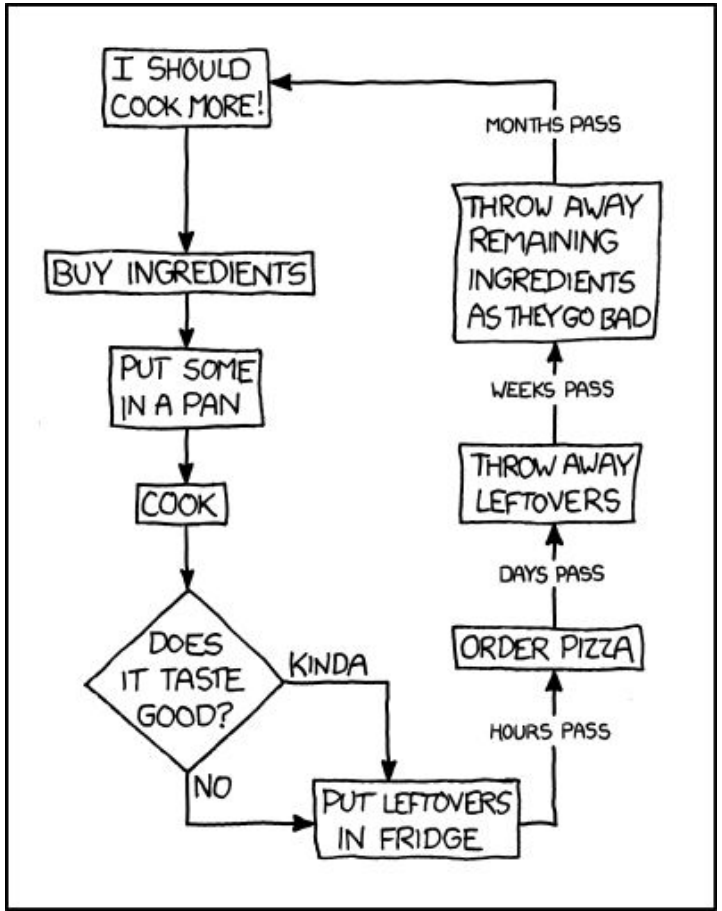
Chemical Bonds



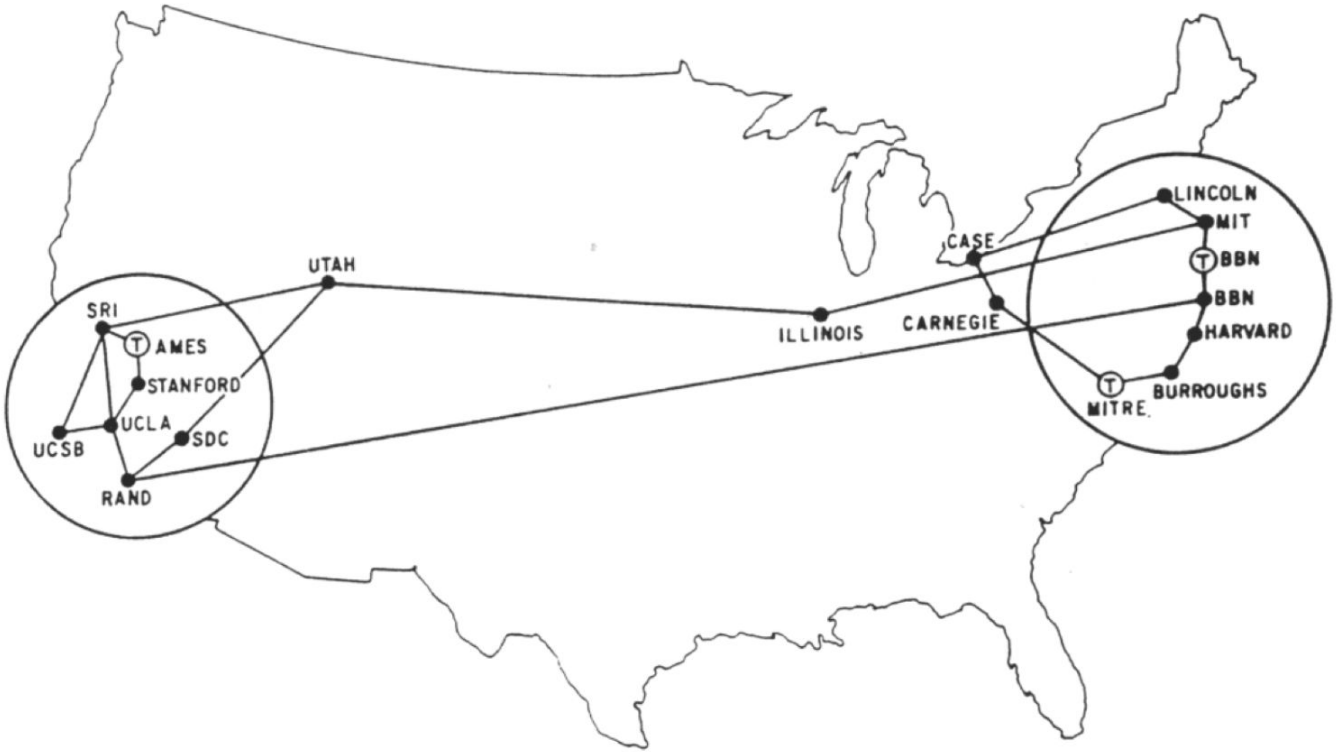
The Interstate Highway System



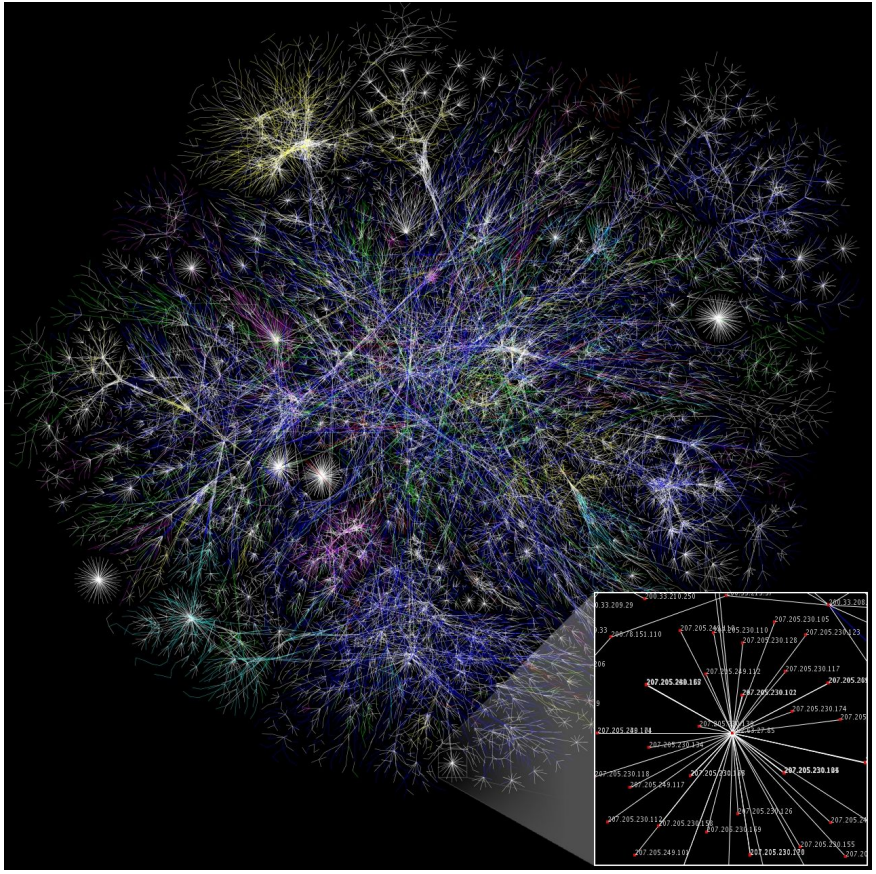
Flowcharts


I SHOULD COOK MORE!
BUY INGREDIENTS
PUT SOME IN A PAN
COOK
DOES IT TASTE GOOD?
KINDA
NO
PUT LEFTOVERS IN FRIDGE
HOURS PASS
ORDER PIZZA
DAYS PASS
THROW AWAY LEFTOVERS
WEEKS PASS
THROW AWAY REMAINING INGREDIENTS AS THEY GO BAD
MONTHS PASS

The Internet!



The Internet!



What is a graph?

Definition

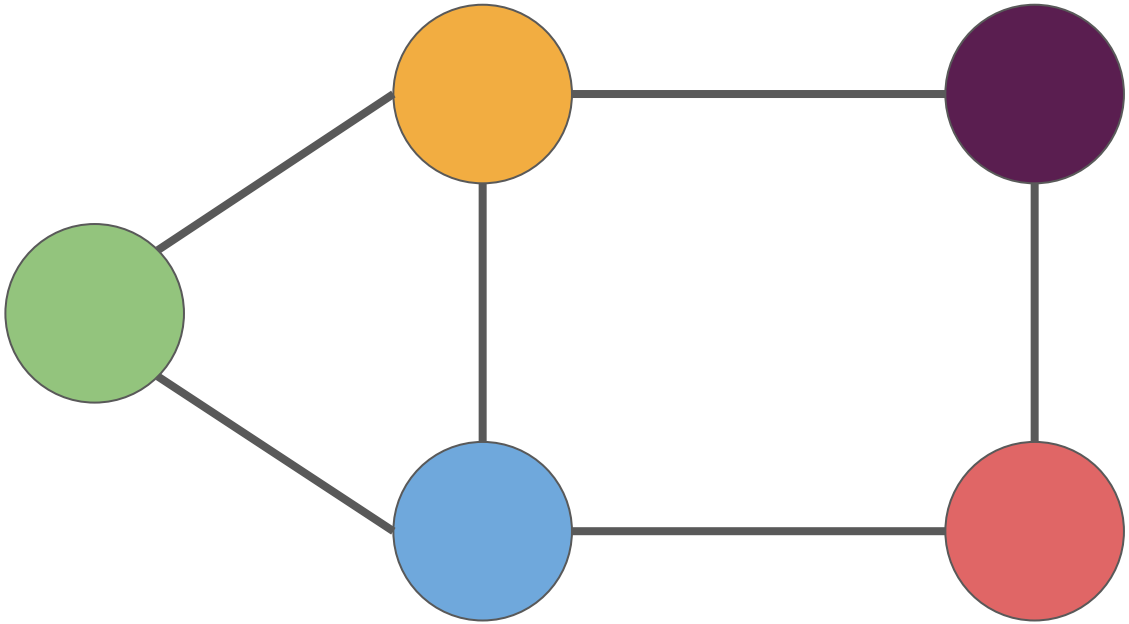
graph
A structured way to represent relationships between different entities.

Our first graph!

- A structured way to represent relationships between different entities.

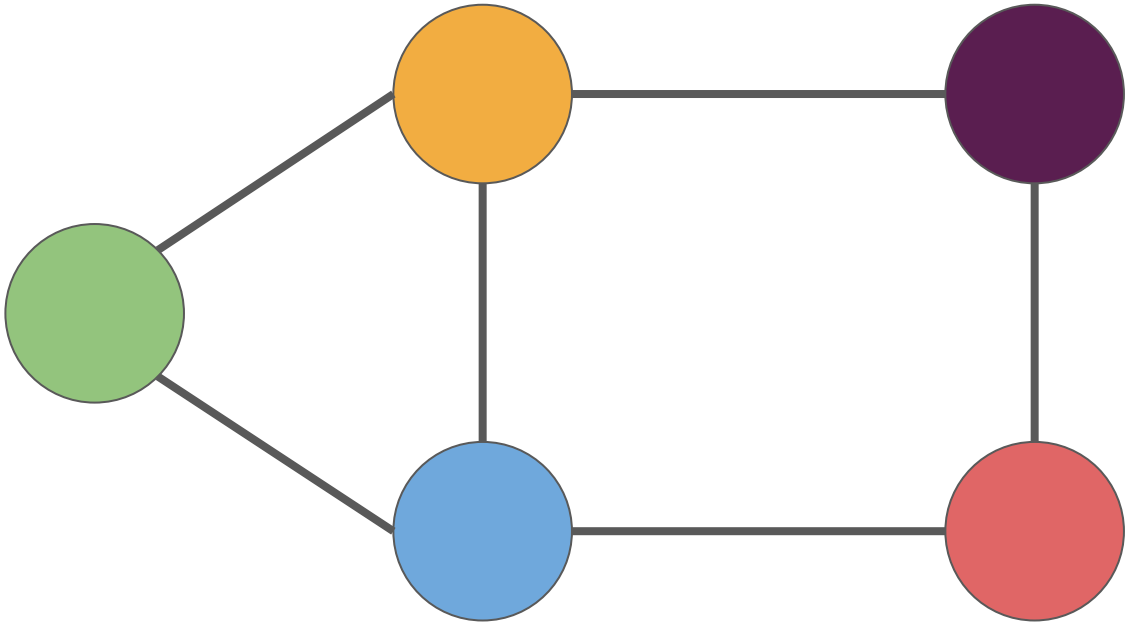
Our first graph!

- A structured way to represent relationships between different entities.



Our first graph!

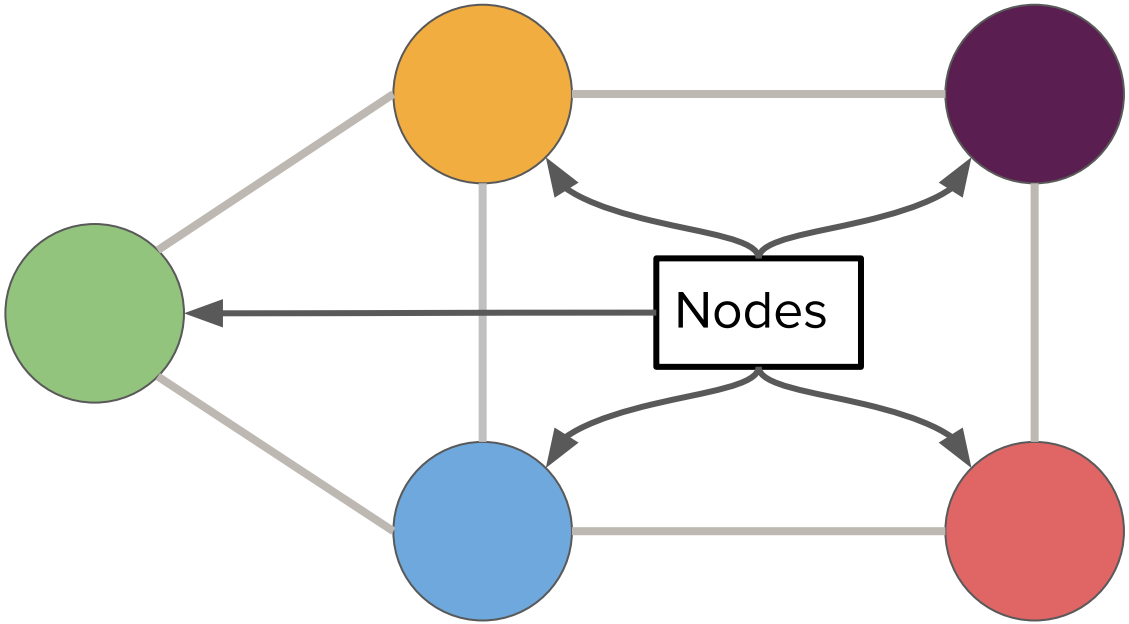
- A structured way to represent relationships between different entities.



A graph consists of a set of **nodes** connected by **edges**.

Our first graph!

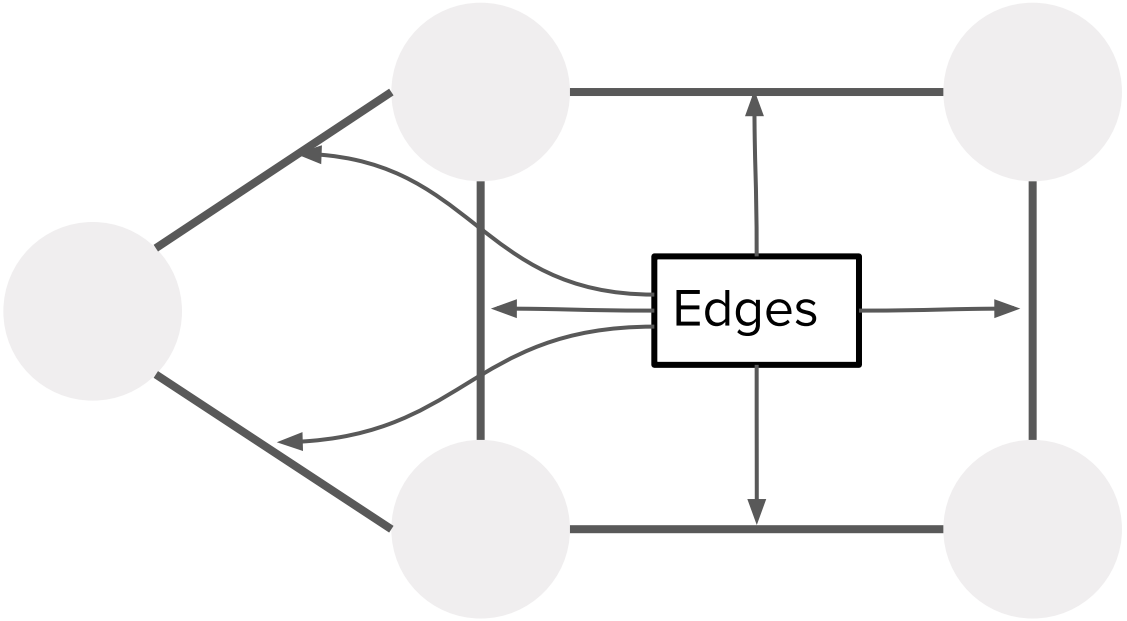
- A structured way to represent relationships between different **entities**.



A graph consists of a set of **nodes** connected by **edges**.

Our first graph!

- A structured way to represent **relationships** between different entities.



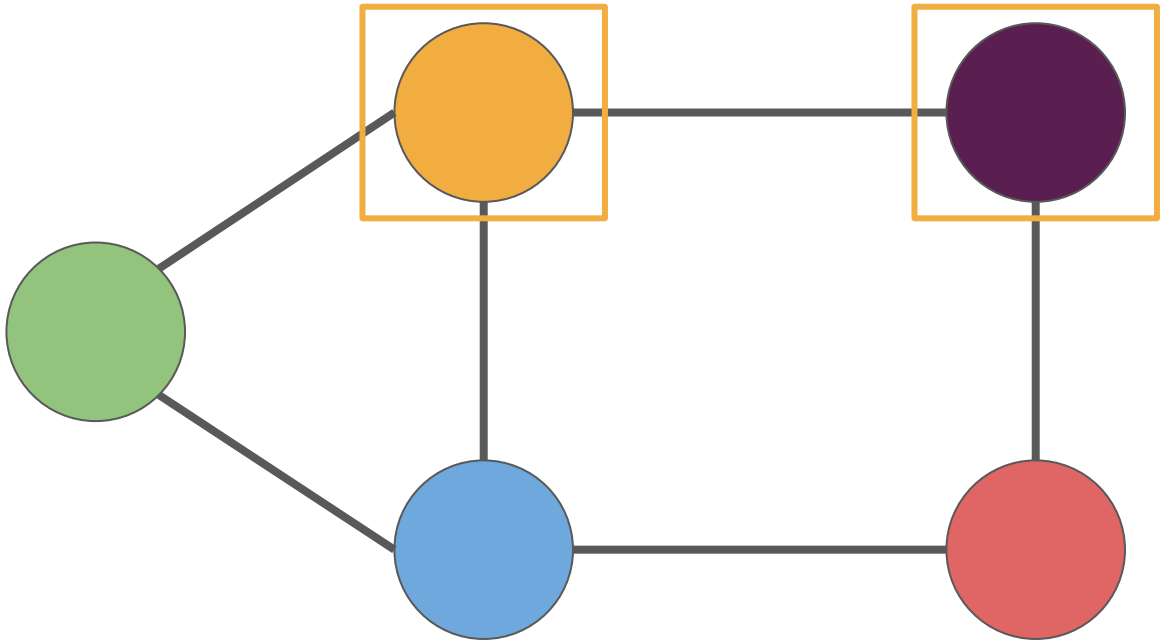
A graph consists of a set of **nodes** connected by **edges**.

Graph Terminology

Graph Terminology

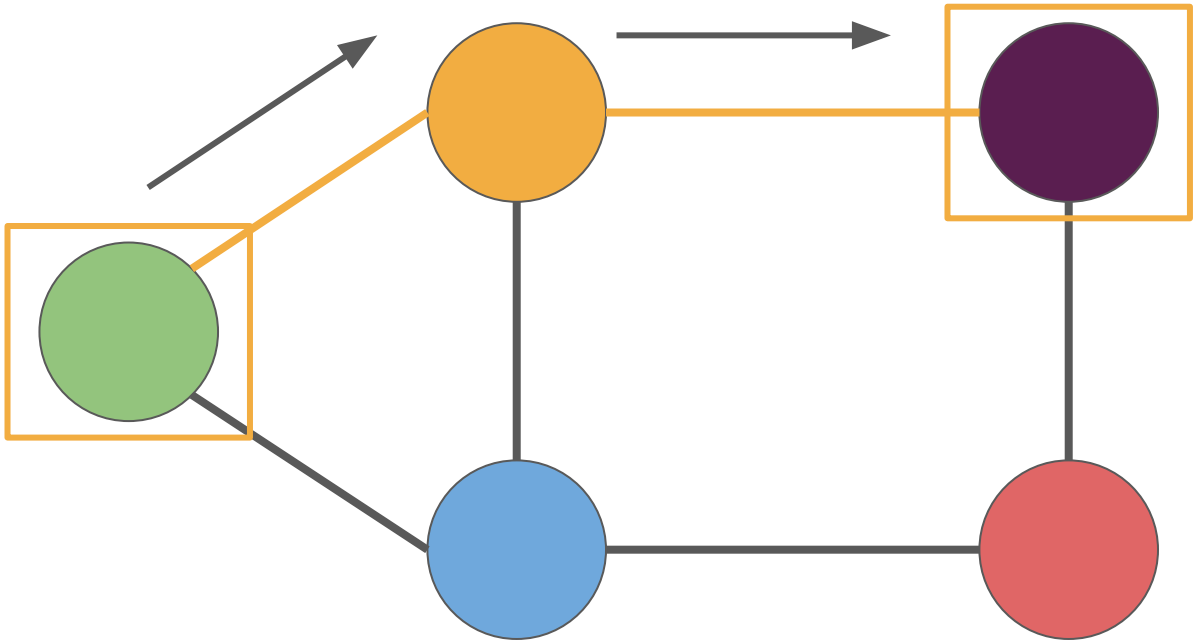
- There are lots of different terms used when talking about graphs and their properties. Let's explore some of them!

Graph Terminology



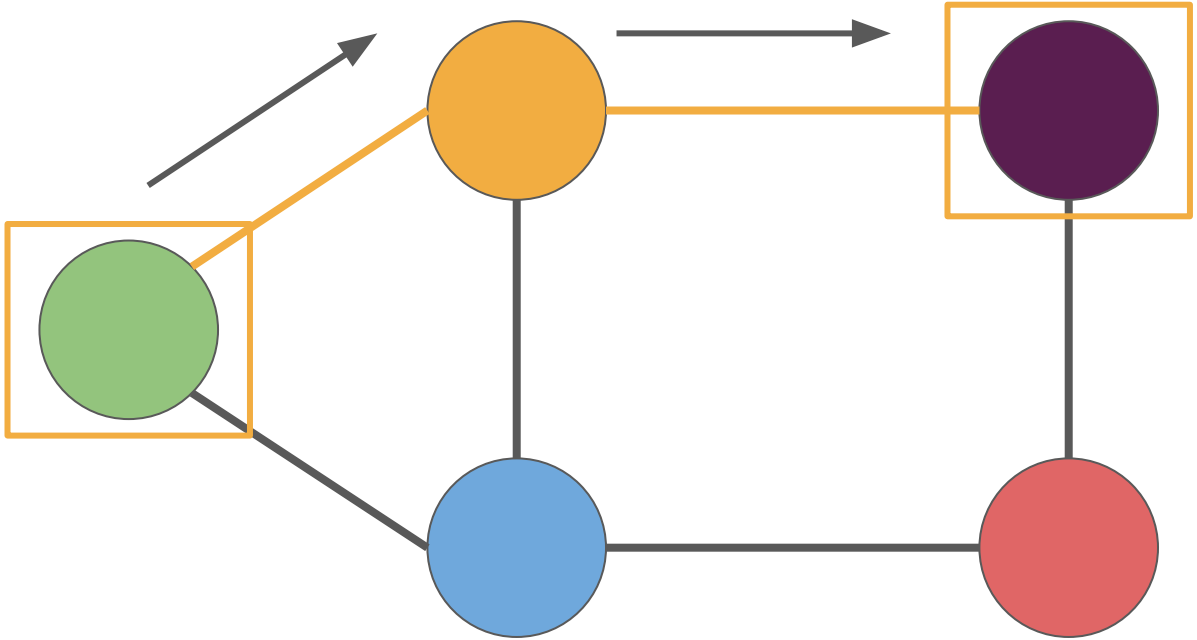
Two nodes are **neighbors** if they are directly connected by an edge.

Graph Terminology



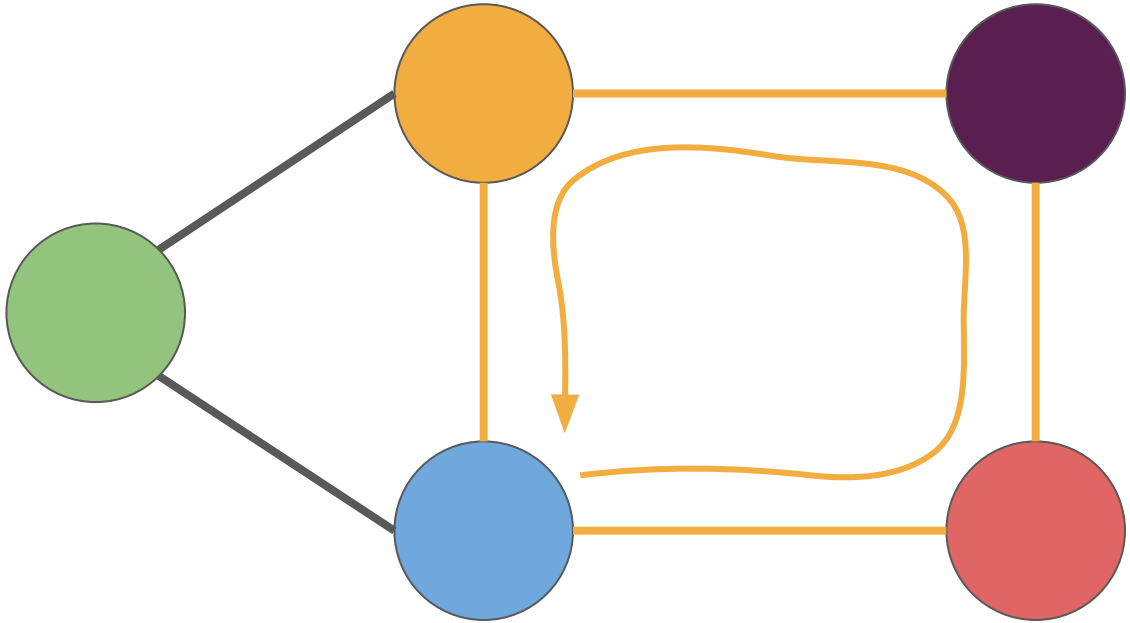
A **path** between two nodes is defined be a sequence of edges that can be followed to traverse between the two nodes.

Graph Terminology



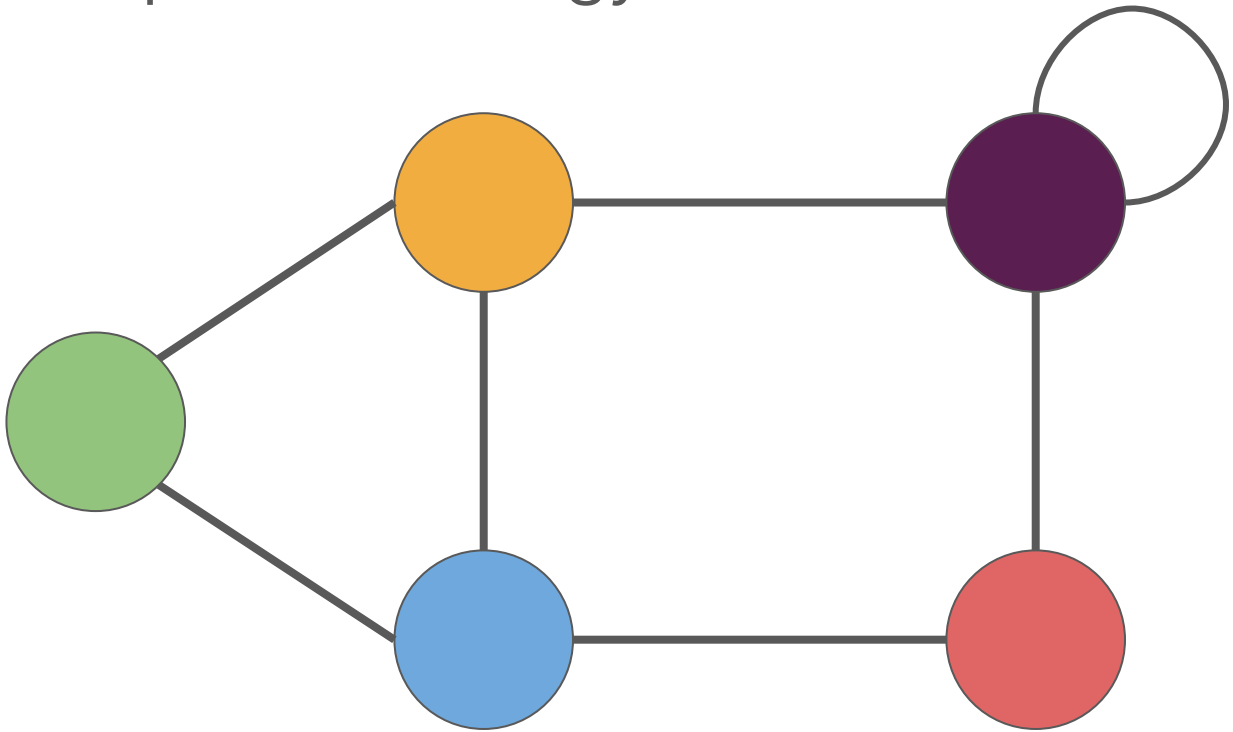
The **length** of a path is the number of edges that make up the path. This path has length 2.

Graph Terminology

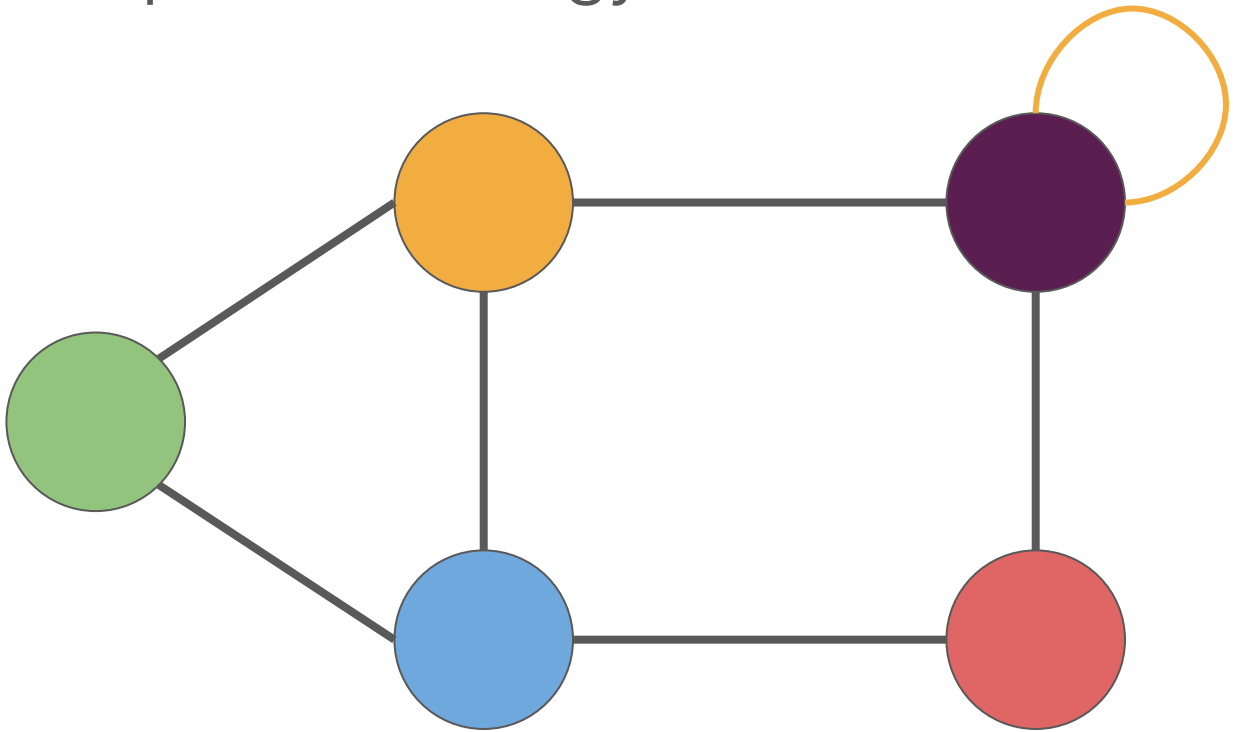


A **cycle** is a path that begins and ends at the same node.

Graph Terminology

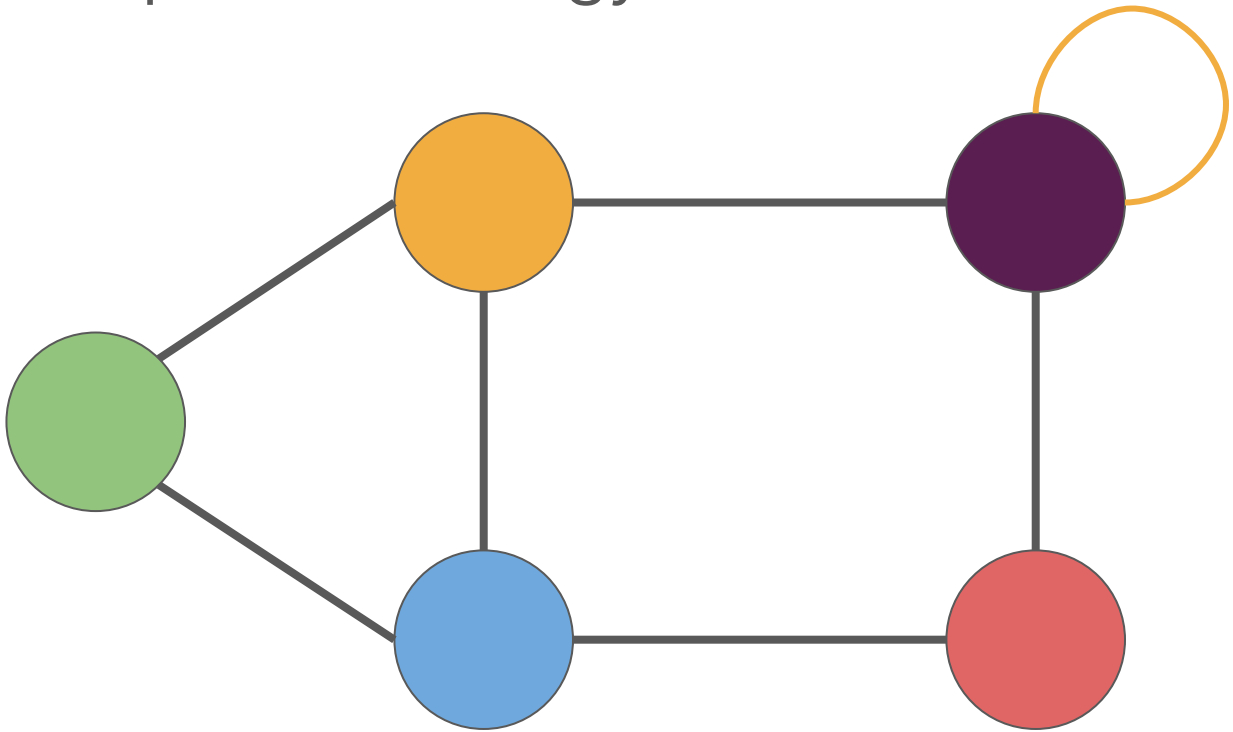


Graph Terminology



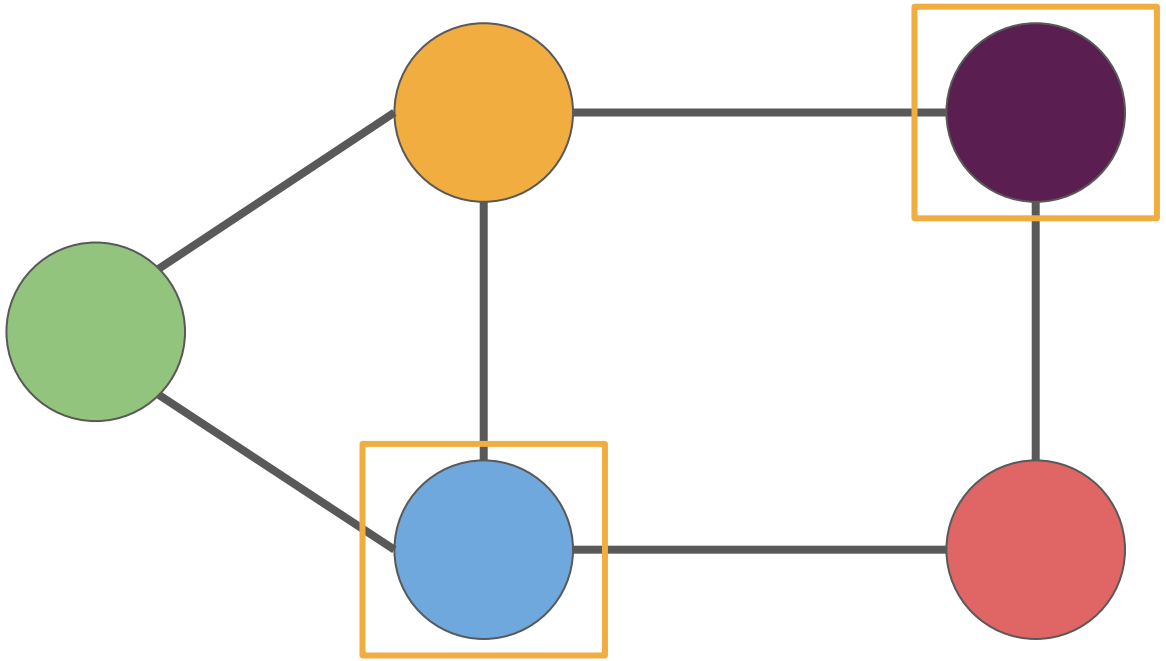
Are we allowed to have edges that look like this?

Graph Terminology



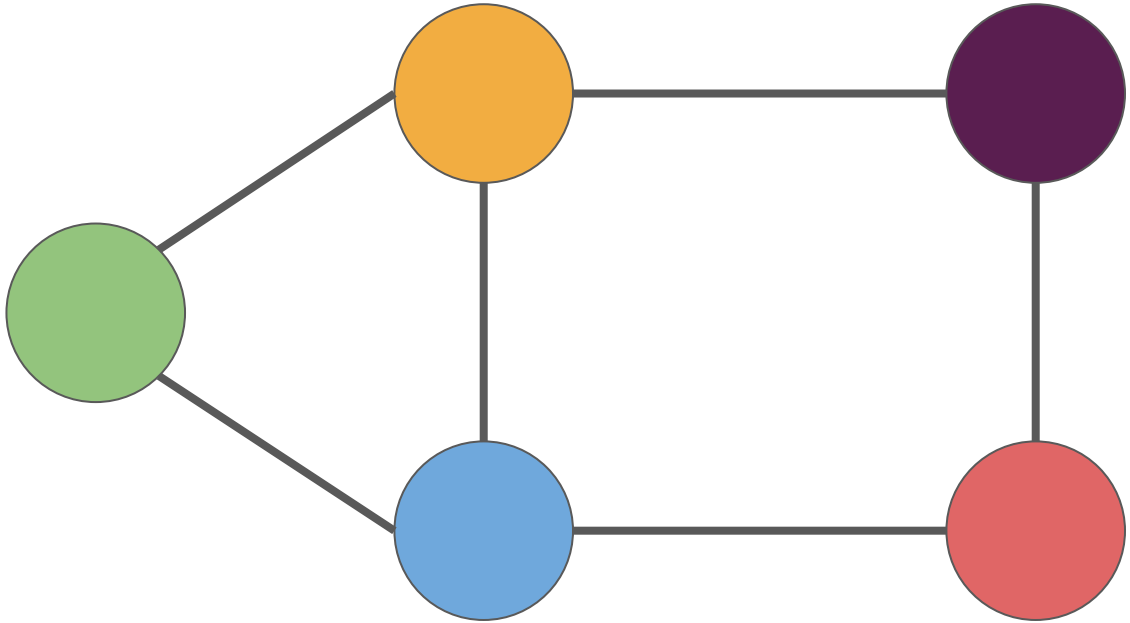
A **loop** is an edge directly from a node back to itself. Some graphs allow loops and some graphs don't!

Graph Terminology



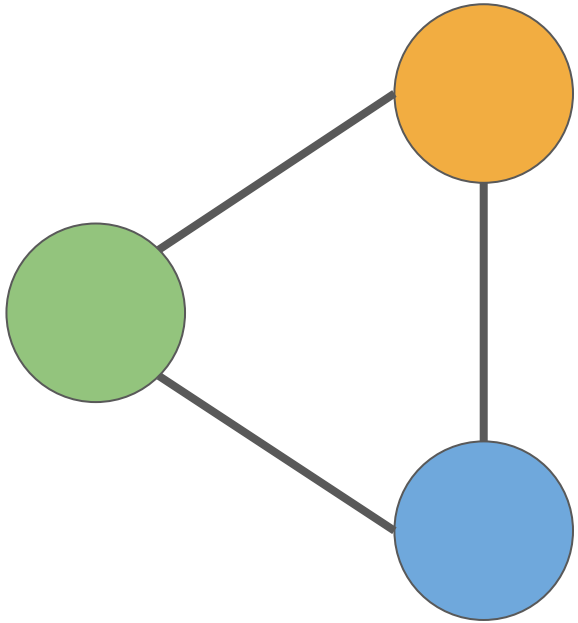
A node is **reachable** from another node if a path exists between the two nodes.

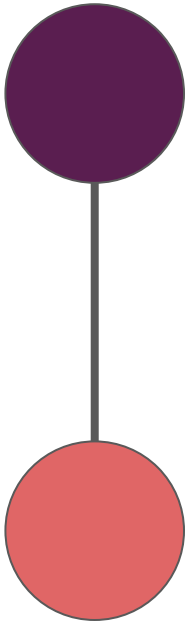
Graph Terminology



A graph is **connected** if all nodes are reachable from all other nodes. This graph is connected!

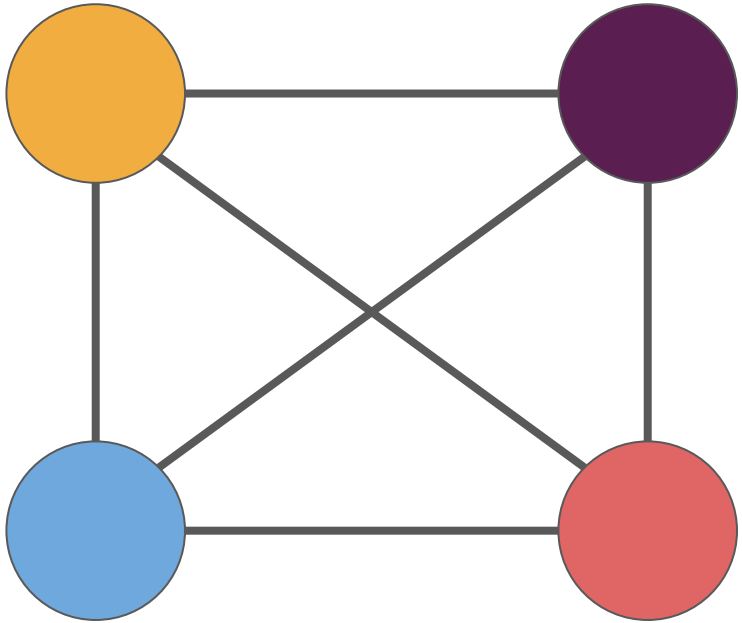
Graph Terminology





A graph is **connected** if all nodes are reachable from all other nodes. This graph is ***not*** connected!

Graph Terminology



A graph is **complete** if every node has an edge connecting it to every other node!

Graph Terminology Summary

- Graph structures
  - Two nodes are **neighbors** if they are directly connected by an edge.
  - A **path** between two nodes is a sequence of edges connecting them. The **length** of a path is defined by the number of edges in the path.
  - A **cycle** is a path that starts and ends at the same node.
  - A **loop** is an edge that connects a node to itself.
- Graph properties
  - A node is **reachable** from another node if a path between the two nodes in the graph exists.
  - A graph is **connected** if all nodes are reachable from all other nodes.
  - A graph is **complete** if edges exist between all pairs of nodes in the graph.

Types of graphs

Different types of graphs

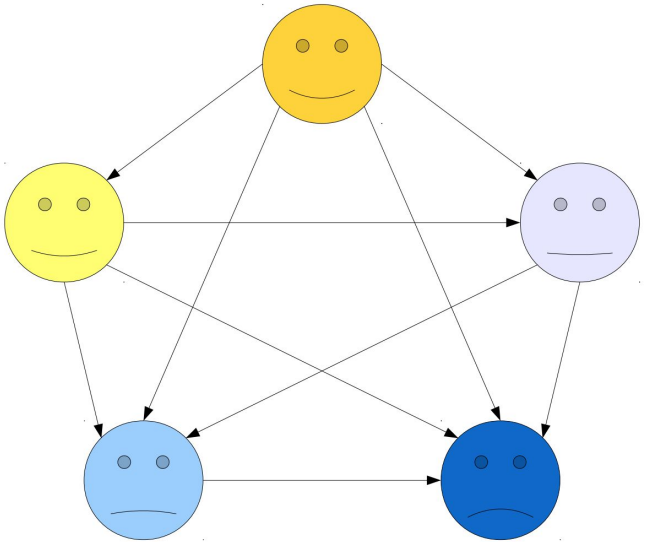
- Some graphs are **directed**. These represent situations where relationships are unidirectional (an action/verb that explicitly implies only one direction).

Different types of graphs

- Some graphs are **directed**. These represent situations where relationships are unidirectional (an action/verb that explicitly implies only one direction).
    - Ex: I follow Dwayne "The Rock" Johnson on Instagram, but he doesn't follow me back.

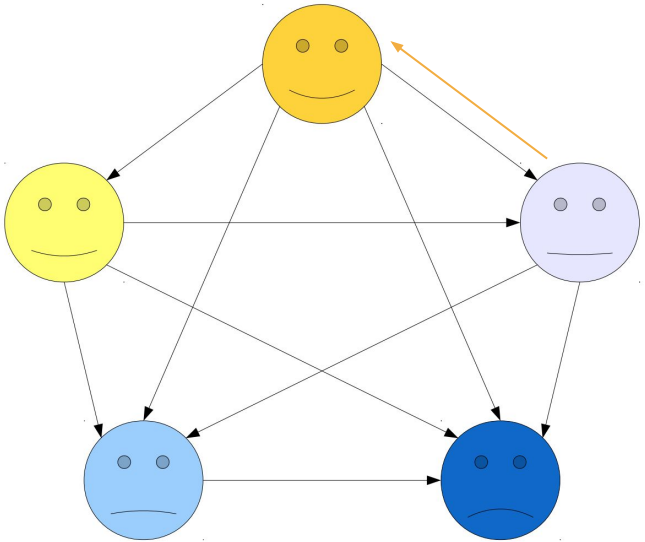
Different types of graphs

- Some graphs are **directed**. These represent situations where relationships are unidirectional (an action/verb that explicitly implies only one direction).
  - Ex: I follow Dwayne "The Rock" Johnson on Instagram, but he doesn't follow me back.



Different types of graphs

- Some graphs are **directed**. These represent situations where relationships are unidirectional (an action/verb that explicitly implies only one direction).
  - Ex: I follow Dwayne "The Rock" Johnson on Instagram, but he doesn't follow me back.



Note: It is possible for a relationship in a directed graph to go both ways between two nodes, but it would need to be explicitly stated.

Different types of graphs

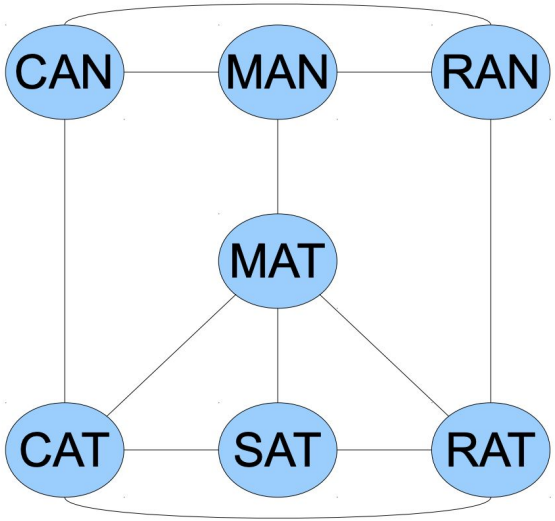
- Some graphs are **undirected**. These represent situations where relationships are bidirectional (the action/verb inherently applies to both entities).

Different types of graphs

- Some graphs are **undirected**. These represent situations where relationships are bidirectional (the action/verb inherently applies to both entities).
  - Ex: I am related to my brother, and he is related to me. The relationship applies to both of us.

Different types of graphs

- Some graphs are **undirected**. These represent situations where relationships are bidirectional (the action/verb inherently applies to both entities).
  - Ex: I am related to my brother, and he is related to me. The relationship applies to both of us.



Different types of graphs

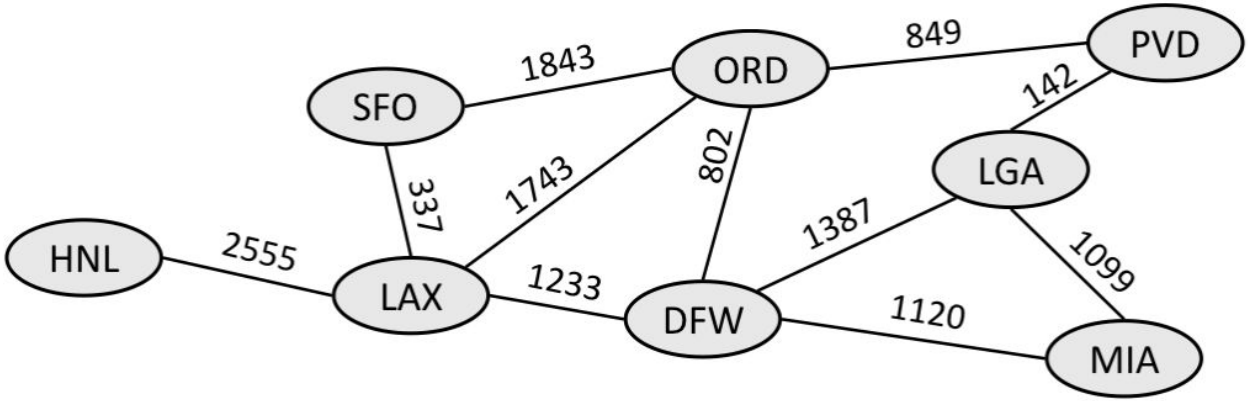
- Some graphs are **weighted**. These represent situations where not all relationships between entities are equal.

Different types of graphs

- Some graphs are **weighted**. These represent situations where not all relationships between entities are equal.
  - Ex: The different bonds between atoms in a single molecule all have different bond energies and strengths.

Different types of graphs

- Some graphs are **weighted**. These represent situations where not all relationships between entities are equal.
  - Ex: The different bonds between atoms in a single molecule all have different bond energies and strengths.

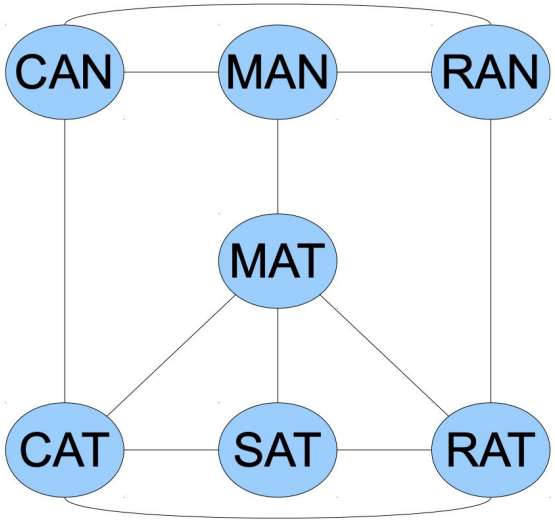


Different types of graphs

- Some graphs are **unweighted**. These represent situations where all relationships between entities have equal importance.

Different types of graphs

- Some graphs are **unweighted**. These represent situations where all relationships between entities have equal importance.
  - Ex: All connected words in a word ladder are one letter apart from one another.



Types of Graphs Summary

- **Directed**: Unidirectional relationships between nodes, represented with a pointed arrow.
- **Undirected**: Bidirectional relationships between nodes, represented with an arrow-less line.
- **Weighted**: Each edge is assigned a numerical "weight" representing its relative significance/strength.
- **Unweighted**: Each edge has equal significance, no labels assigned.

Revisiting Graph Examples

Revisiting Graph Examples: Social Network

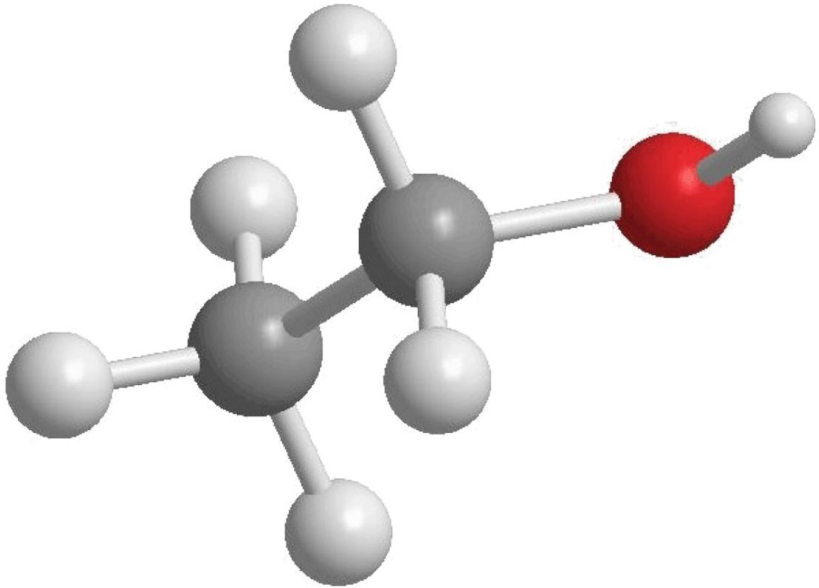
Properties

- Nodes: People
- Edges: "Friendship" or "Following"
- Undirected (Facebook) or Directed (Instagram)
- Unweighted

Revisiting Graph Examples: Chemical Bonds

Properties

- Nodes: Atoms
- Edges: Bonds (covalent or ionic)
- Undirected
- Weighted



Revisiting Graph Examples: Interstate Highways

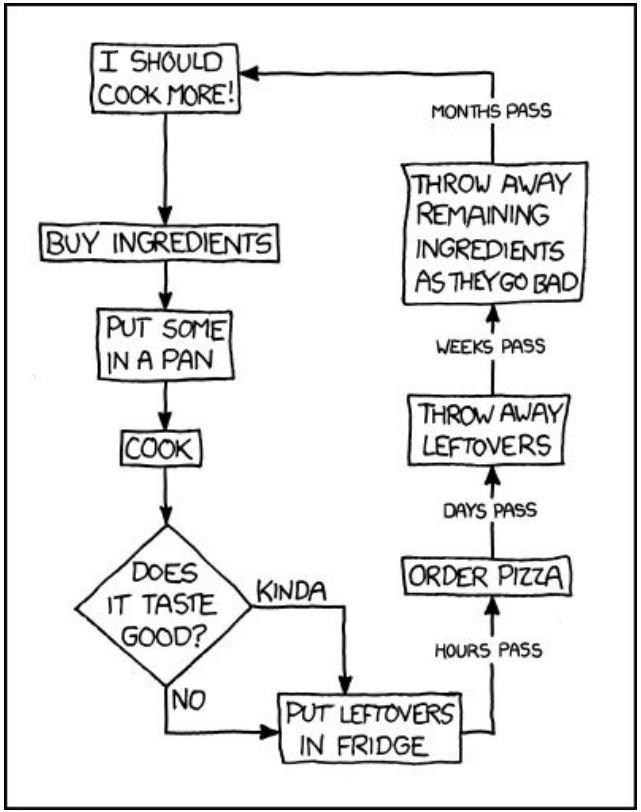
Properties

- Nodes: Cities
- Edges: Highways/roads
- Undirected
- Weighted

Revisiting Graph Examples: Flowcharts

Properties

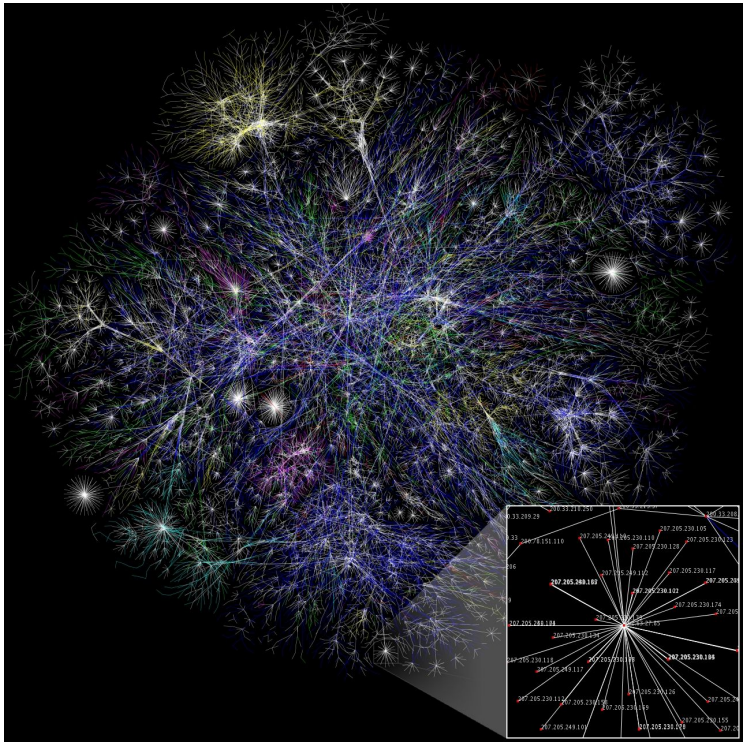
- Nodes: Events/Actions
- Edges: Transitions
- Directed
- Unweighted



Revisiting Graph Examples: The Internet

Properties

- Nodes: Devices (phones, computers, etc.)
- Edges: Connection pathways (Bluetooth, WiFi, Ethernet, cables)
- Undirected
- Can be weighted or unweighted



Graphs as Linked Data Structures

Putting it All Together

- We've seen nodes connected by edges (links) before when discussing linked lists and trees. These, along with graphs, are all **linked data structures**!

Putting it All Together

- We've seen nodes connected by edges (links) before when discussing linked lists and trees. These, along with graphs, are all **linked data structures**!

- What differentiates each of these linked data structures?

Putting it All Together

- We've seen nodes connected by edges (links) before when discussing linked lists and trees. These, along with graphs, are all **linked data structures**!

- What differentiates each of these linked data structures?
  - **Linked lists:** Linear structure, each node connected to at most one other node.

Putting it All Together

- We've seen nodes connected by edges (links) before when discussing linked lists and trees. These, along with graphs, are all **linked data structures**!

- What differentiates each of these linked data structures?
  - **Linked lists:** Linear structure, each node connected to at most one other node.
  - **Trees:** Nodes can connect to multiple other nodes, no cycles, parent/child relationship and a single, special root node.

Putting it All Together

- We've seen nodes connected by edges (links) before when discussing linked lists and trees. These, along with graphs, are all **linked data structures**!

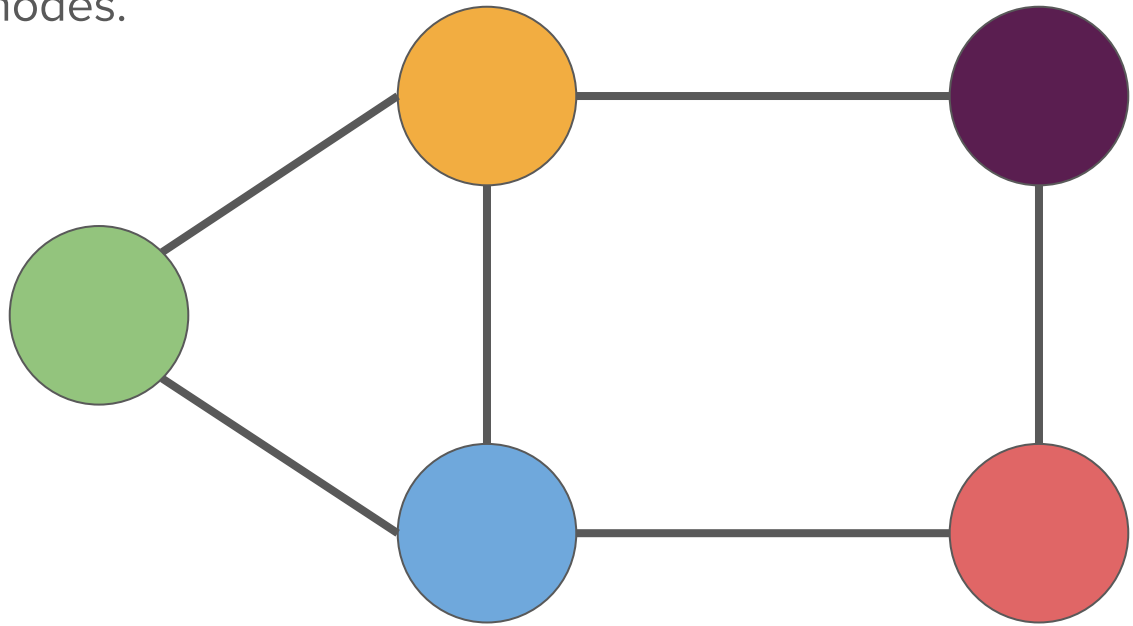
- What differentiates each of these linked data structures?
  - **Linked lists:** Linear structure, each node connected to at most one other node.
  - **Trees:** Nodes can connect to multiple other nodes, no cycles, parent/child relationship and a single, special root node.
  - **Graphs:** No restrictions. It's the wild, wild west of the node-based world!

The Wild World of Graphs

- Graphs can have cycles, and there is no notion of a parent-child relationship between nodes.

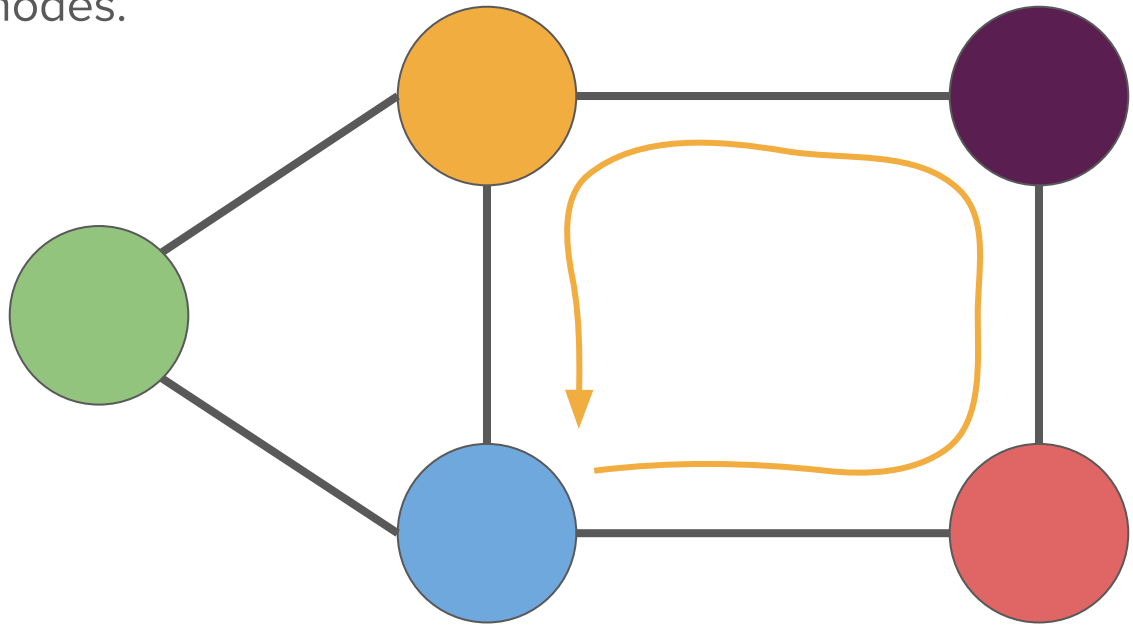
The Wild World of Graphs

- Graphs can have cycles, and there is no notion of a parent-child relationship between nodes.



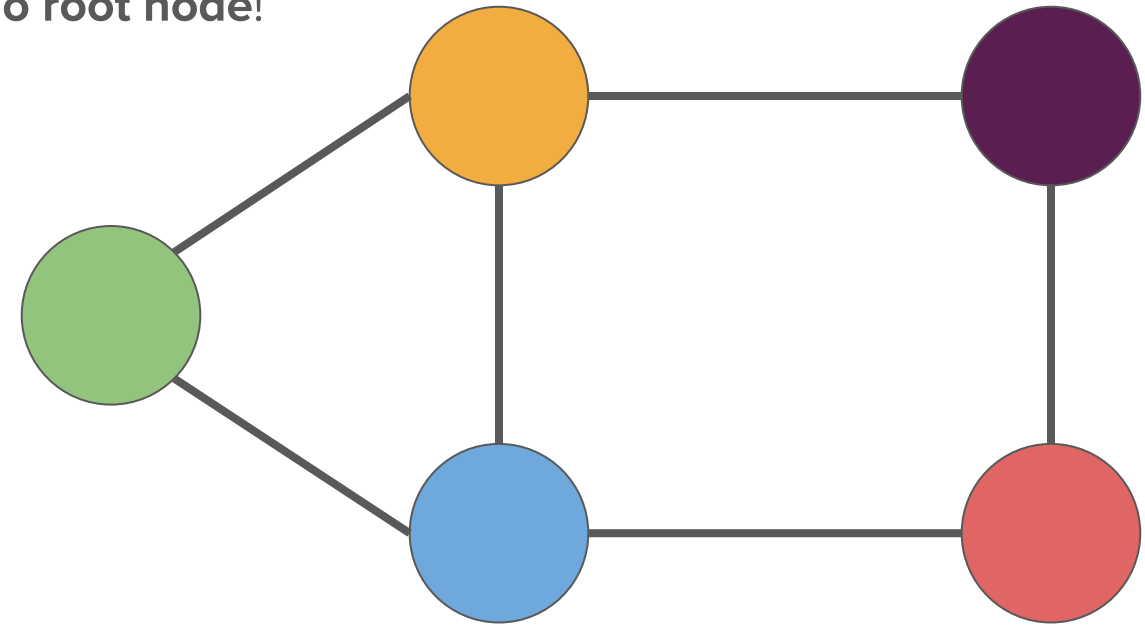
The Wild World of Graphs

- Graphs can have **cycles**, and there is no notion of a parent-child relationship between nodes.



The Wild World of Graphs

- Graphs have no nodes that are more important than other nodes. In particular, **there is no root node**!



Graphs are the most powerful, flexible, and expressive abstraction that we can use to model relationships between different distributed entities. You will find graphs everywhere you look!

Representing Graphs

How do we store and represent graphs in code?

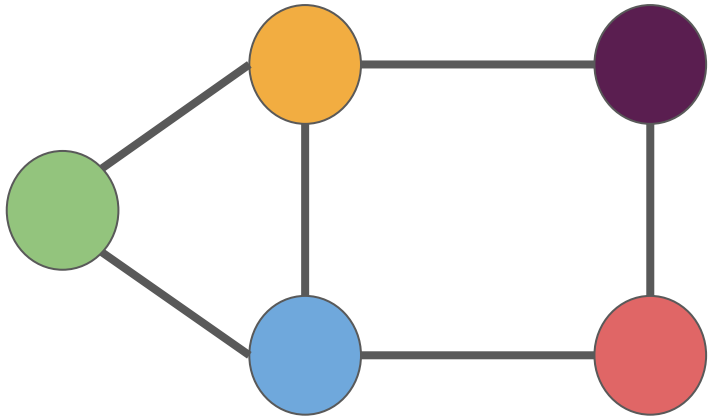
Take 1: Adjacency List

Take 1: Adjacency List

- We can represent a graph as a map from nodes to the collection of nodes that each node is adjacent to.

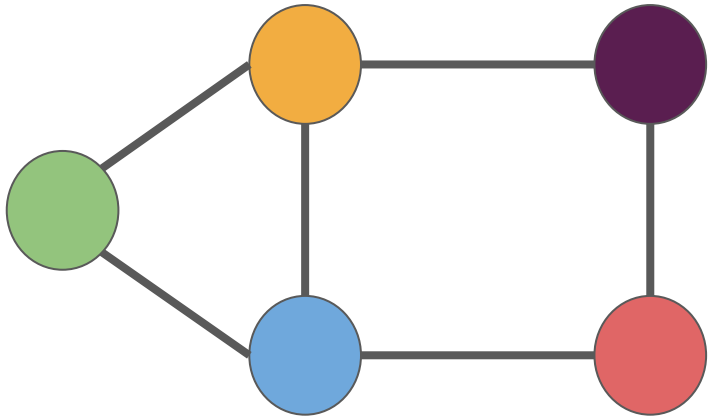
Take 1: Adjacency List

- We can represent a graph as a map from nodes to the collection of nodes that each node is adjacent to.



Take 1: Adjacency List

- We can represent a graph as a map from nodes to the collection of nodes that each node is adjacent to.

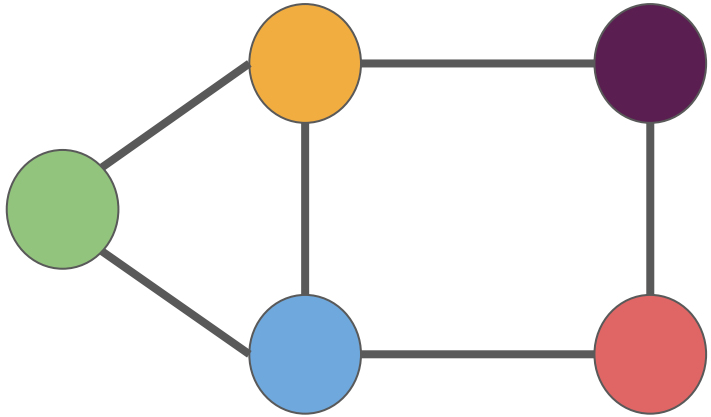


`Map<Node, Set<Node>>`

Node	Set<Node>
Node	Adjacent to

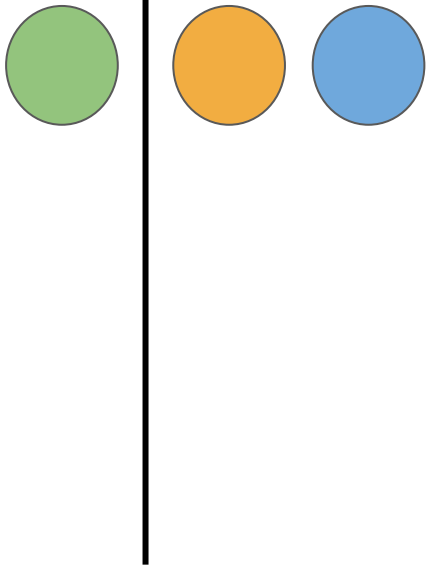
Take 1: Adjacency List

- We can represent a graph as a map from nodes to the collection of nodes that each node is adjacent to.



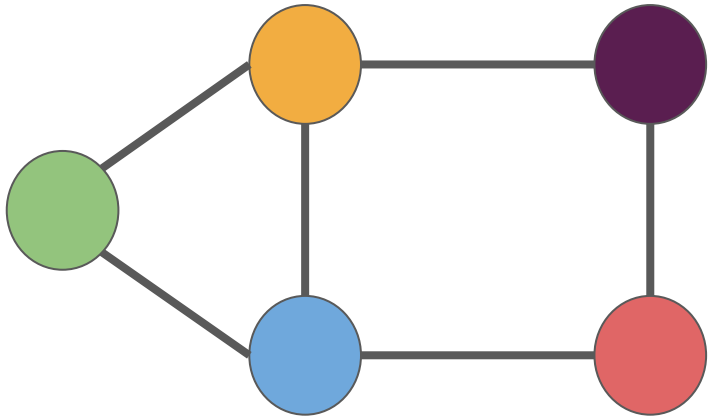
`Map<Node, Set<Node>>`

Node	Set<Node>
Node	Adjacent to



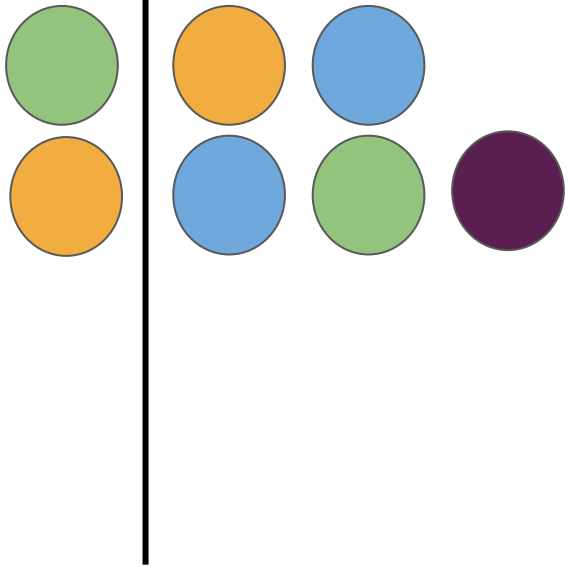
Take 1: Adjacency List

- We can represent a graph as a map from nodes to the collection of nodes that each node is adjacent to.



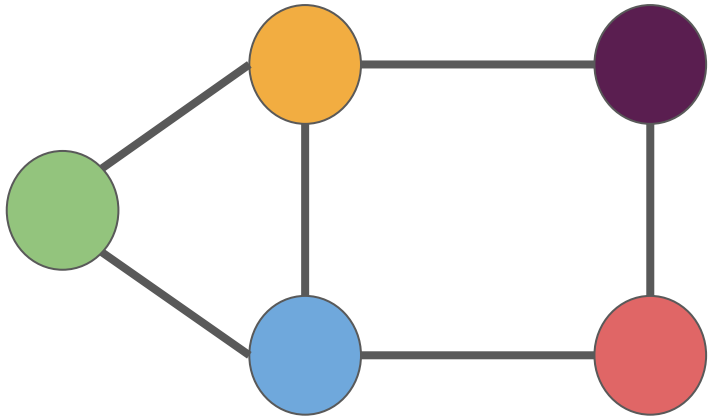
`Map<Node, Set<Node>>`

Node	Set<Node>
Node	Adjacent to

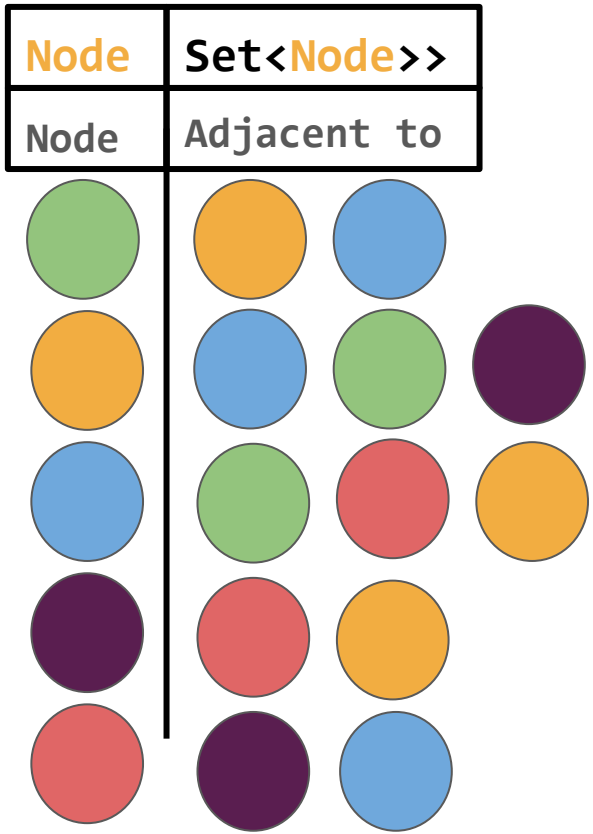


Take 1: Adjacency List

- We can represent a graph as a map from nodes to the collection of nodes that each node is adjacent to.



`Map<Node, Set<Node>>`



Take 1: Adjacency List

- The approach we just saw is called an adjacency list in comes in a number of different forms:
  - **`Map<Node, Vector<Node>>`**
  - **`HashMap<Node, HashSet<Node>>`**
  - **`Map<Node, Set<Node>>`**
  - **`Vector<Node>`** `<- in this case, the Node struct holds collection of its adjacent neighbors`

Take 1: Adjacency List

- The approach we just saw is called an adjacency list in comes in a number of different forms:
  - `Map<Node, Vector<Node>>`
  - `HashMap<Node, HashSet<Node>>`
  - `Map<Node, Set<Node>>`
  - `Vector<Node>` `<- in this case, the Node struct holds collection of its adjacent neighbors`

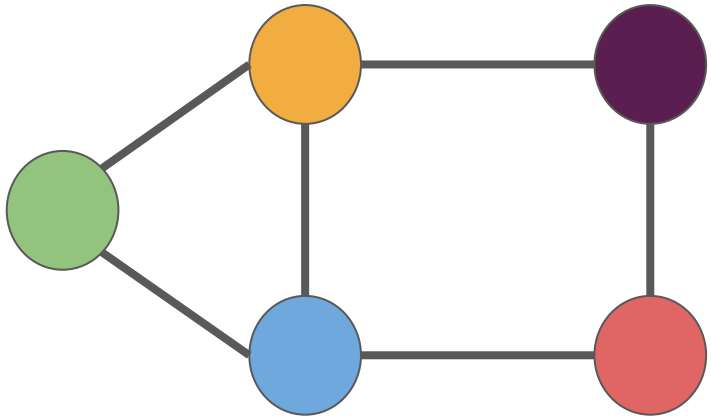
- The core idea is that we have some kind of mapping associating each node with its outgoing edges (or neighboring nodes).

Take 2: Adjacency Matrix

- We can also use a two-dimensional matrix to represent the relationships in a graph.

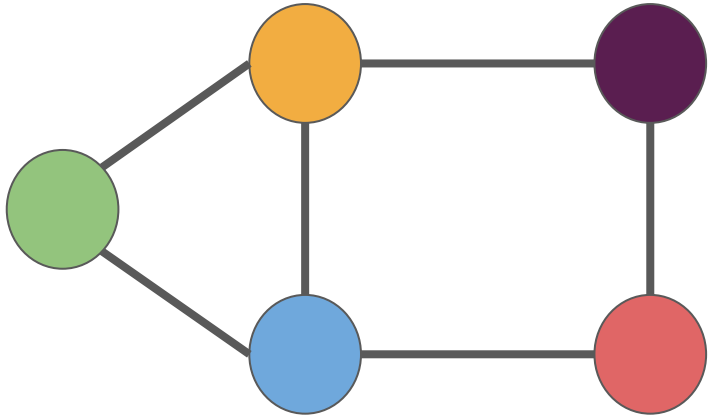
Take 2: Adjacency Matrix

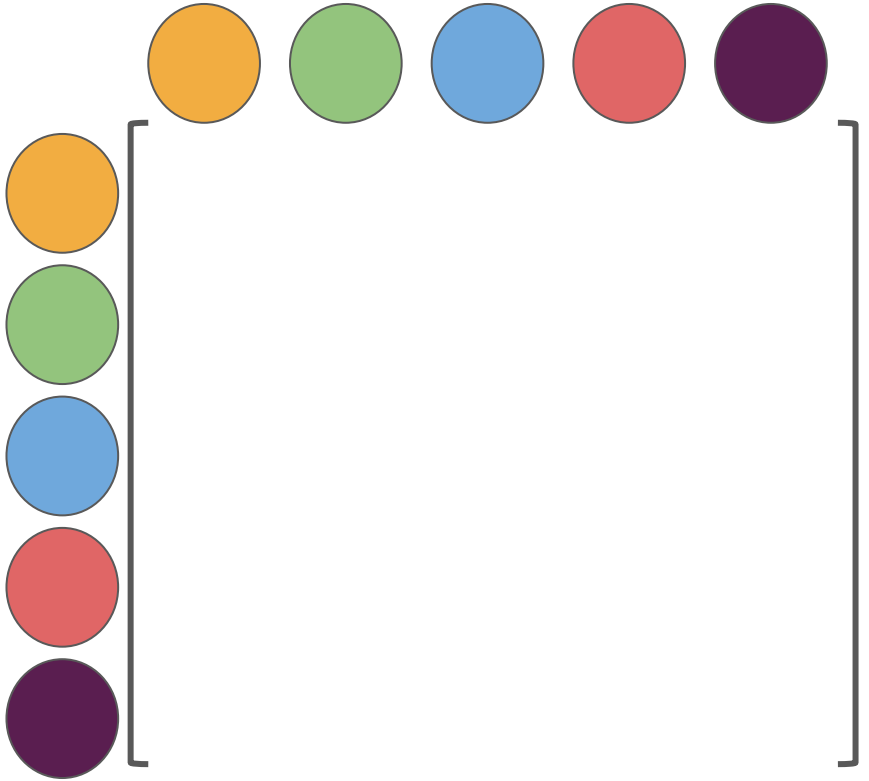
- We can also use a two-dimensional matrix to represent the relationships in a graph.



Take 2: Adjacency Matrix

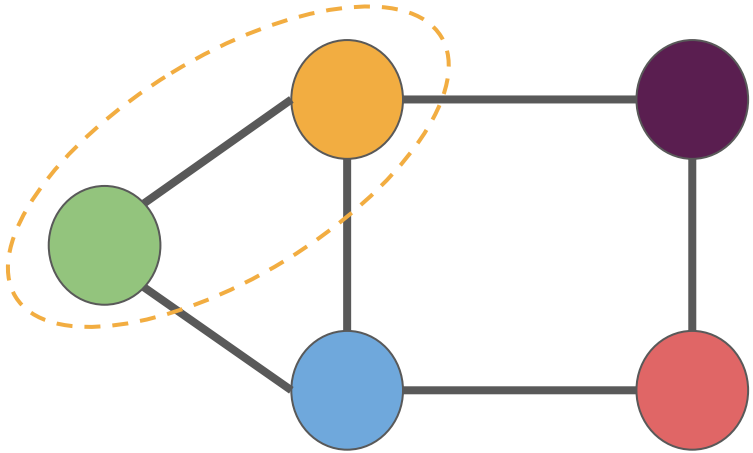
- We can also use a two-dimensional matrix to represent the relationships in a graph.

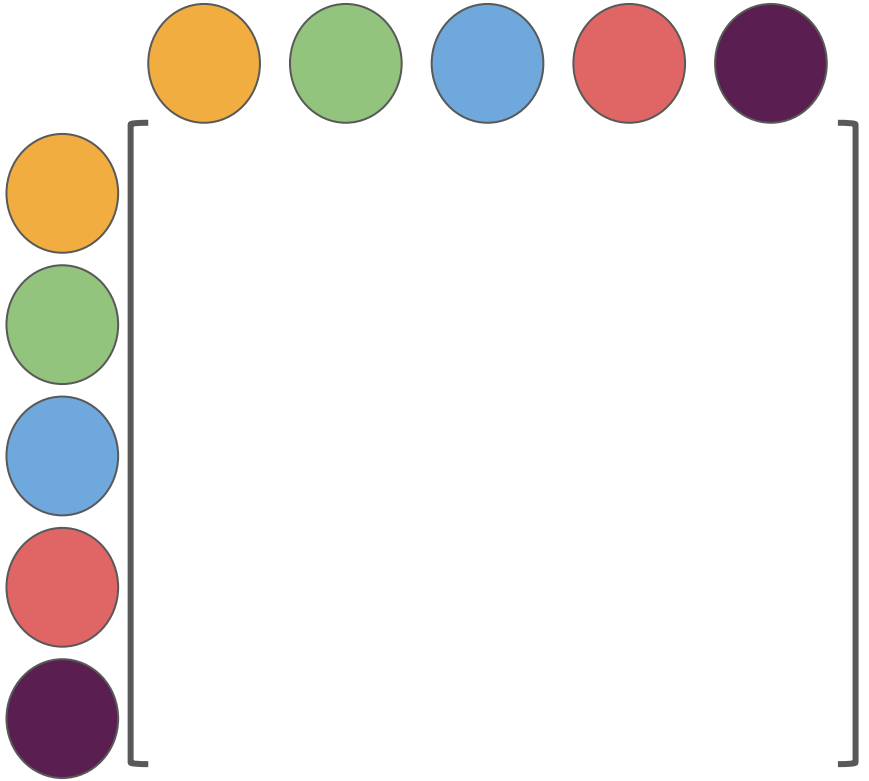




Take 2: Adjacency Matrix

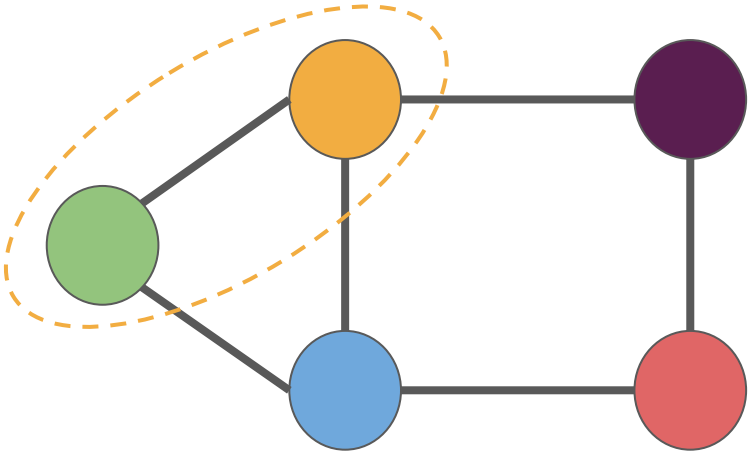
- We can also use a two-dimensional matrix to represent the relationships in a graph.

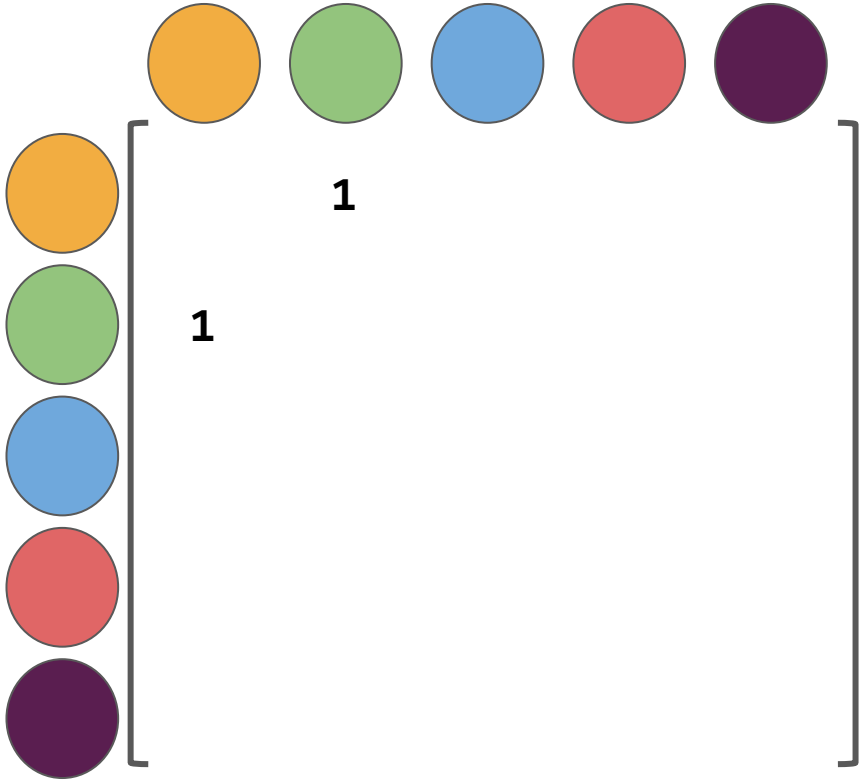




Take 2: Adjacency Matrix

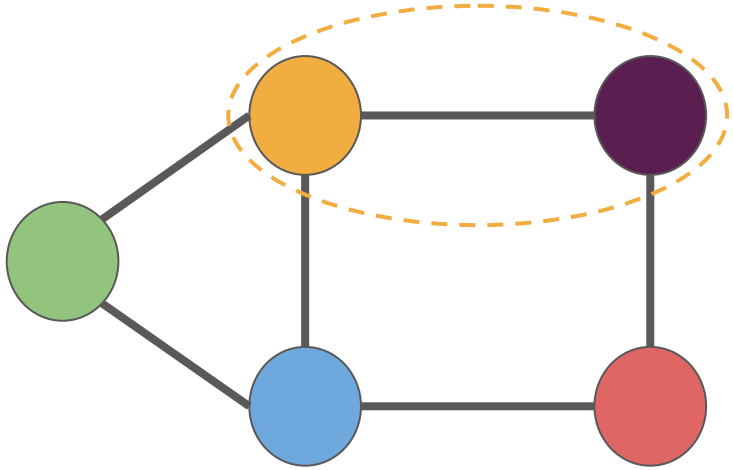
- We can also use a two-dimensional matrix to represent the relationships in a graph.

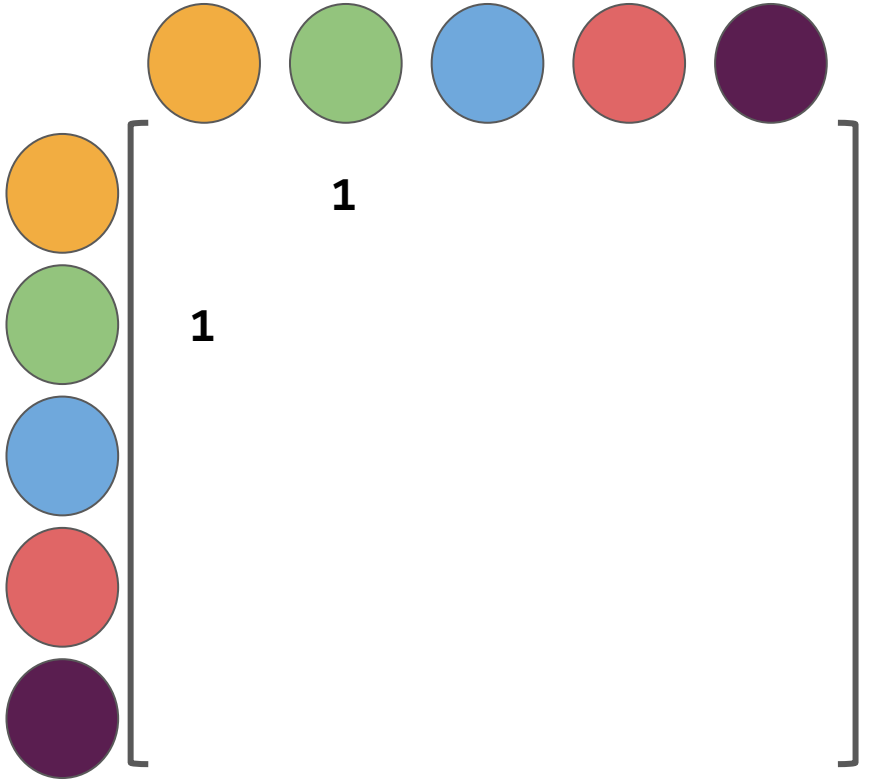




Take 2: Adjacency Matrix

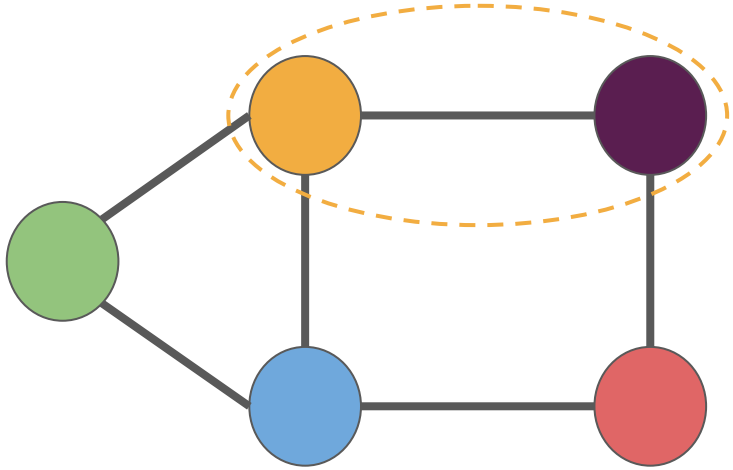
- We can also use a two-dimensional matrix to represent the relationships in a graph.

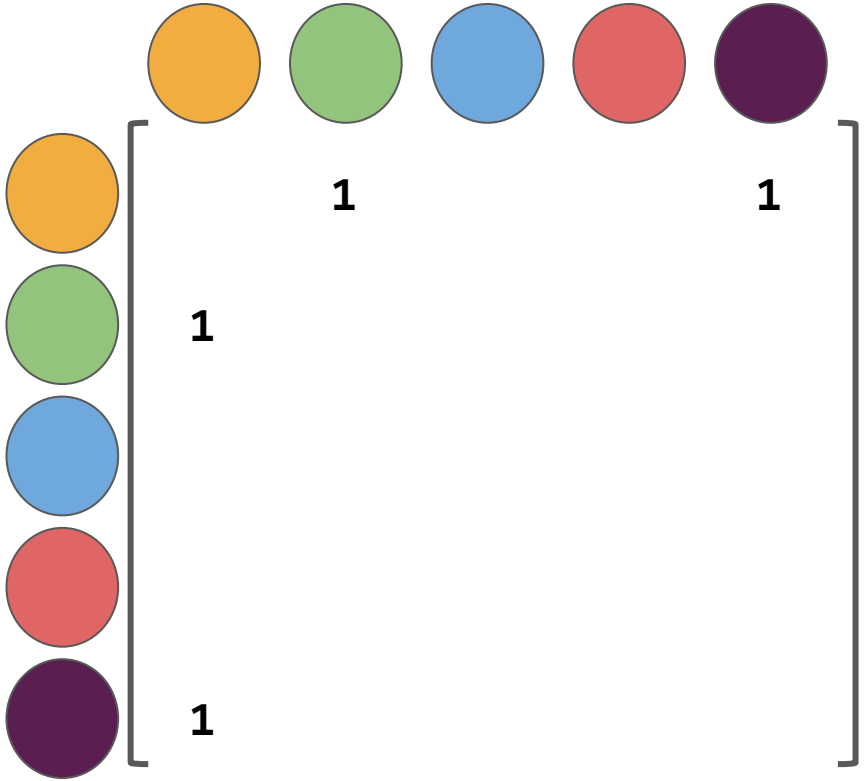




Take 2: Adjacency Matrix

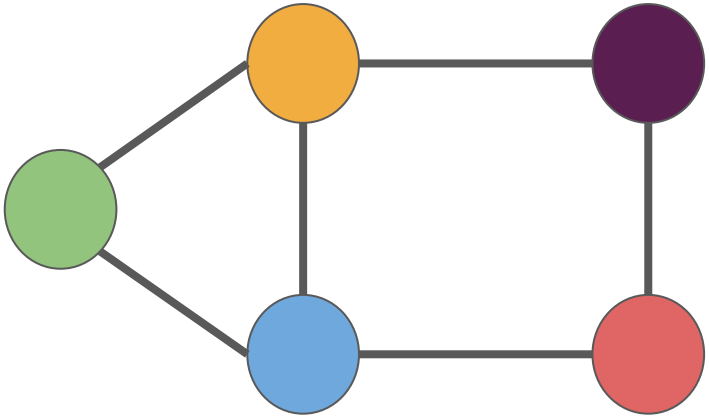
- We can also use a two-dimensional matrix to represent the relationships in a graph.

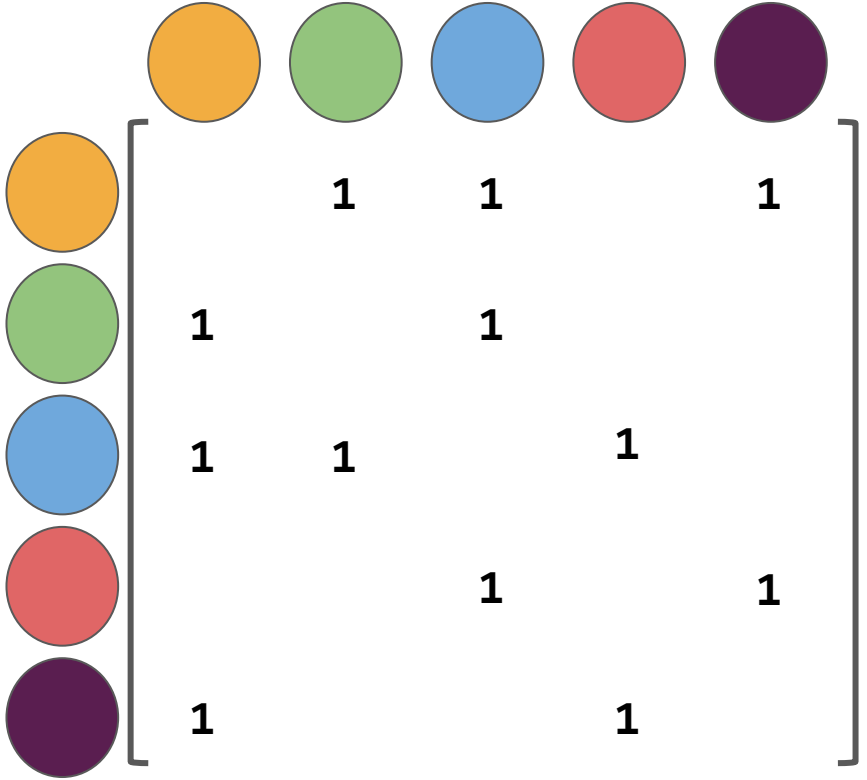




Take 2: Adjacency Matrix

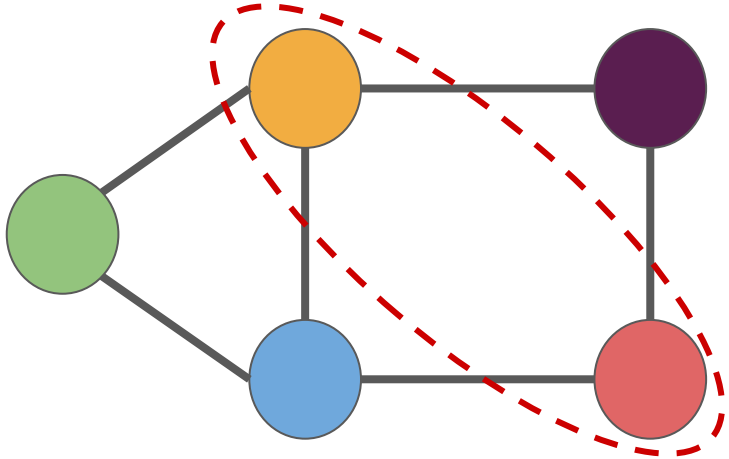
- We can also use a two-dimensional matrix to represent the relationships in a graph.

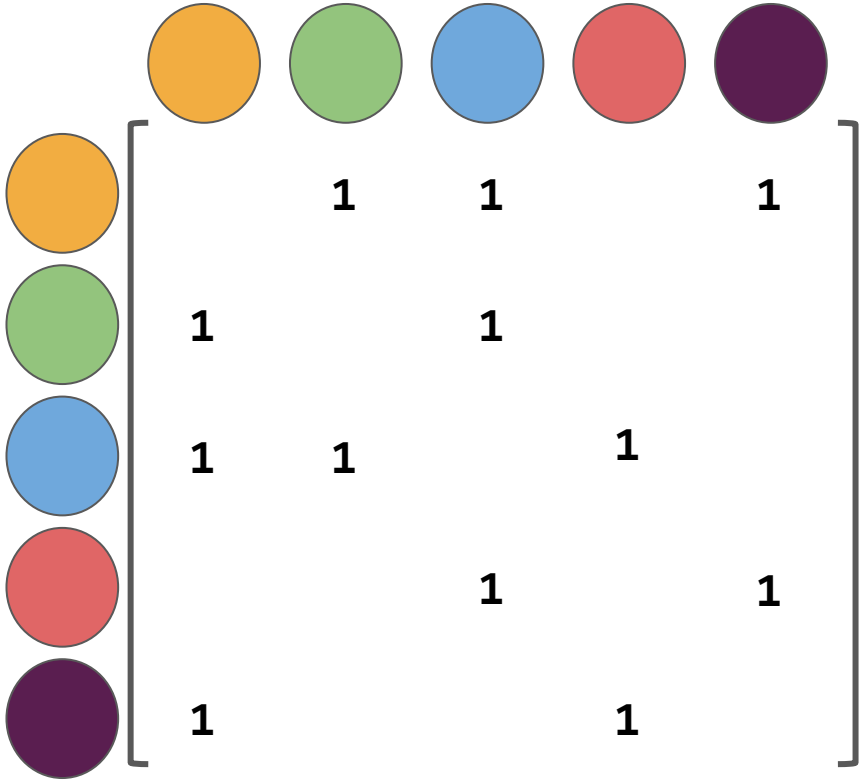




Take 2: Adjacency Matrix

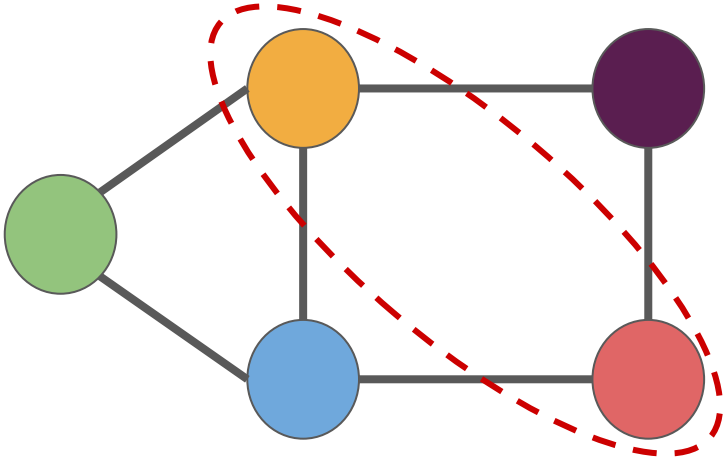
- We can also use a two-dimensional matrix to represent the relationships in a graph.

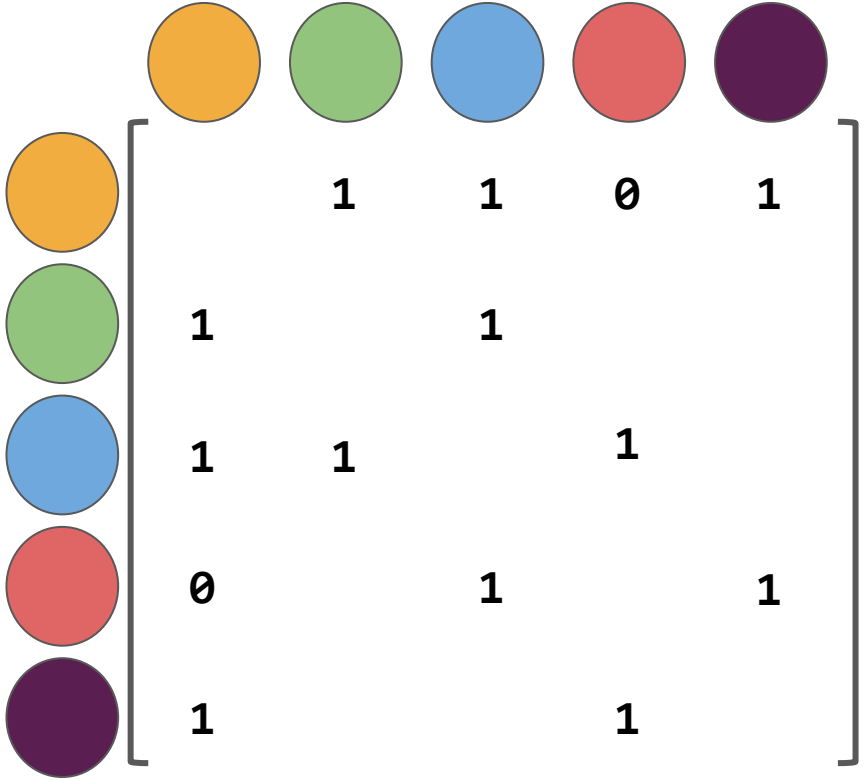




Take 2: Adjacency Matrix

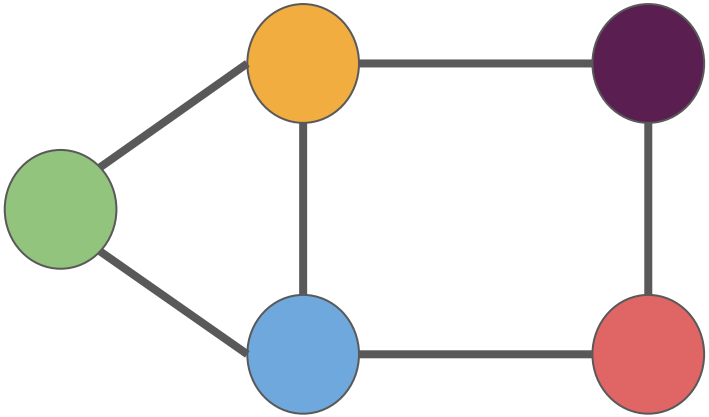
- We can also use a two-dimensional matrix to represent the relationships in a graph.

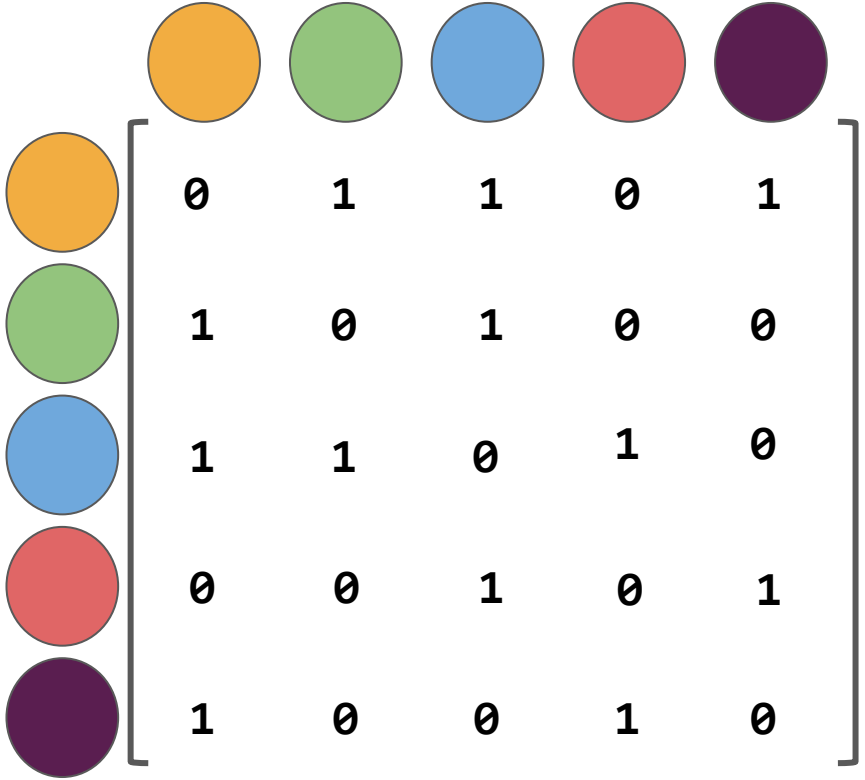




Take 2: Adjacency Matrix

- We can also use a two-dimensional matrix to represent the relationships in a graph.





Going forward, unless stated otherwise, assume we're using an **adjacency list** representation.

Announcements

Announcements

- Assignment 6 is due on **Wednesday, August 12 at 11:59pm PDT**. This is a hard deadline – there is **no grace period, and no submissions will be accepted after this time**.

- Make sure to sign up for a final project presentation time slot on Paperless! You should be expecting to present for 30 minutes, sometime between Thursday and Sunday of this week.

- Remember that minors will be asked to access tomorrow's lecture in a slightly modified format. More details will be posted on Ed by tomorrow morning.

Graph Algorithms

Graph Traversal

Iterating over a Graph

- In a singly-linked list, there’s pretty much one way to iterate over the list: start at the front and go forward!

Iterating over a Graph

- In a singly-linked list, there's pretty much one way to iterate over the list: start at the front and go forward!

- In a tree, there are many traversal strategies:
  - Pre-order traversal
  - In-order traversal
  - Post-order traversal

Iterating over a Graph

- In a singly-linked list, there’s pretty much one way to iterate over the list: start at the front and go forward!

- In a tree, there are many traversal strategies:
  - Pre-order traversal
  - In-order traversal
  - Post-order traversal

- There are many ways to iterate over a graph, each of which have different properties.
  - First idea: Let's revisit breadth-first search!

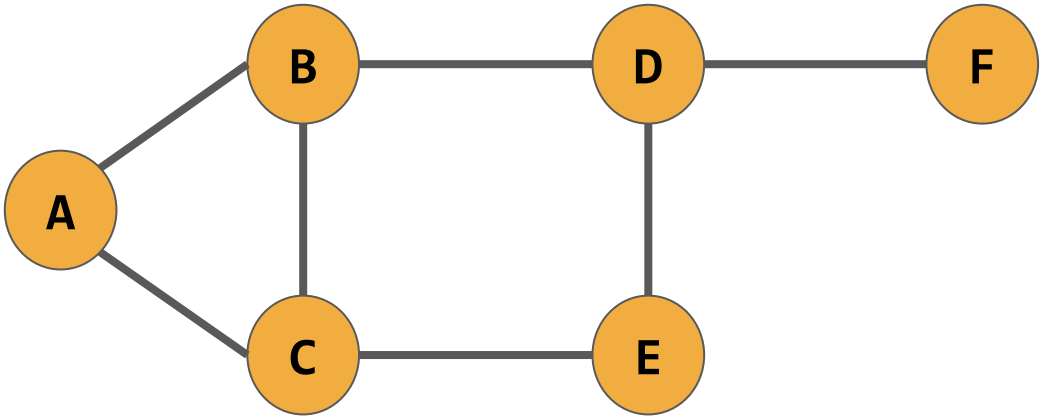
Breadth-First Search

Revisiting Breadth-First Search

- Core Idea: Find everything one hop away from the start, then two hops away, then three hops away, etc.

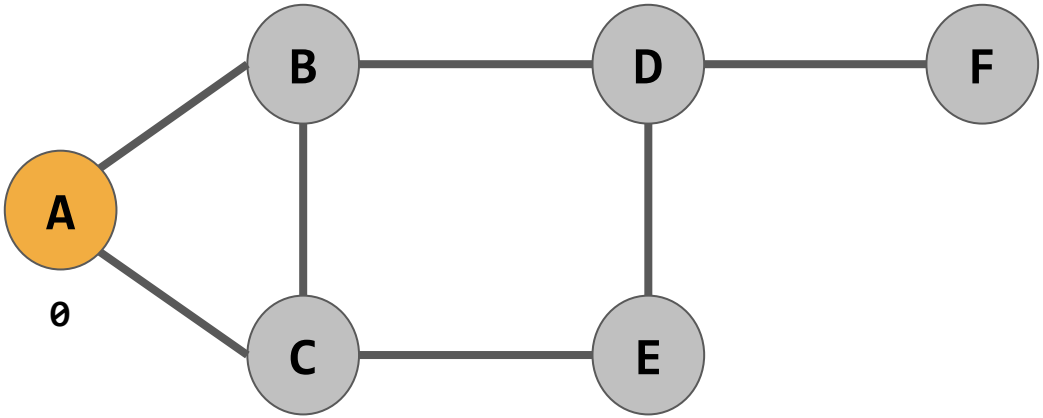
Revisiting Breadth-First Search

- Core Idea: Find everything one hop away from the start, then two hops away, then three hops away, etc.
- Goal: Find the shortest path from A to F.



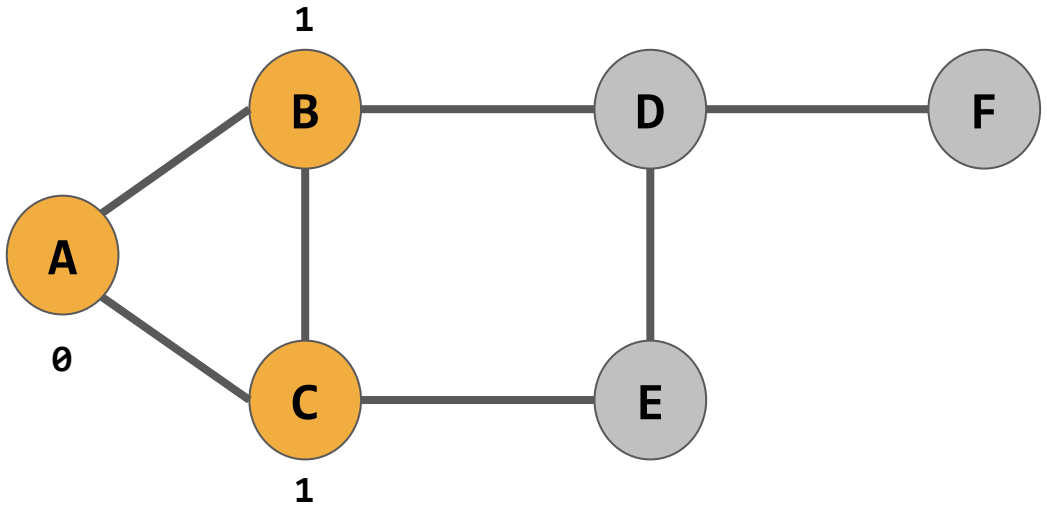
Revisiting Breadth-First Search

- Core Idea: Find everything one hop away from the start, then two hops away, then three hops away, etc.
- Goal: Find the shortest path from A to F.



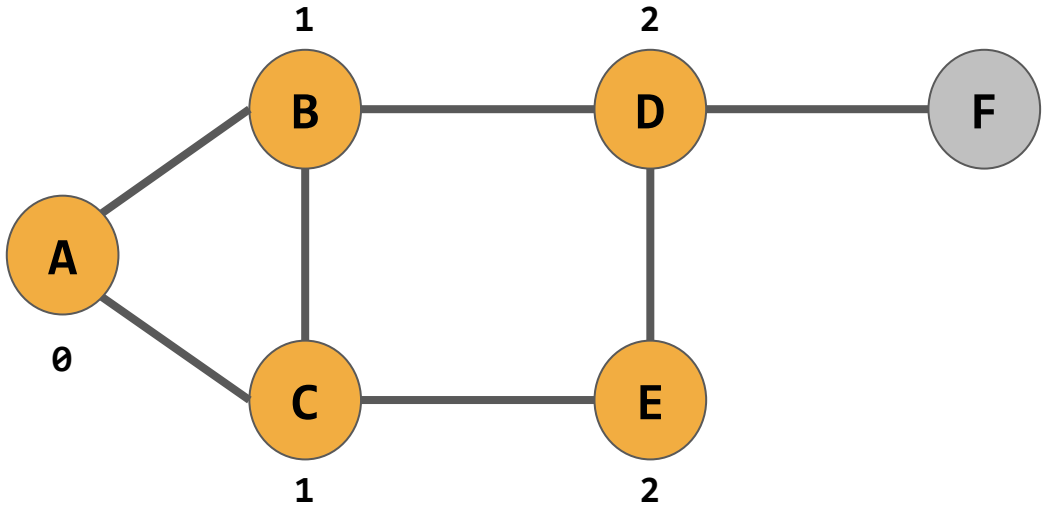
Revisiting Breadth-First Search

- Core Idea: Find everything one hop away from the start, then two hops away, then three hops away, etc.
- Goal: Find the shortest path from A to F.



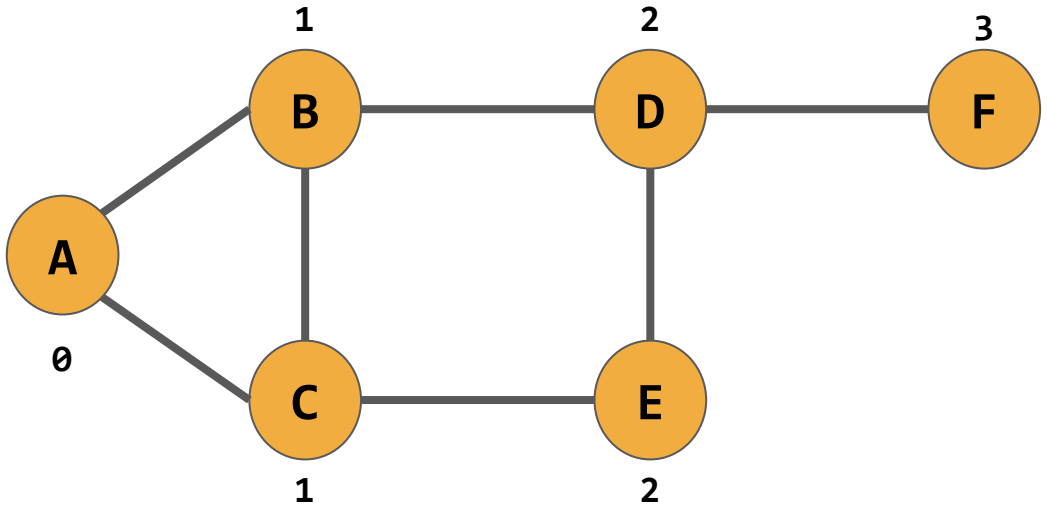
Revisiting Breadth-First Search

- Core Idea: Find everything one hop away from the start, then two hops away, then three hops away, etc.
- Goal: Find the shortest path from A to F.



Revisiting Breadth-First Search

- Core Idea: Find everything one hop away from the start, then two hops away, then three hops away, etc.
- Goal: Find the shortest path from A to F.



Graph Breadth-First Search

- The BFS algorithm on graphs looks very similar to what we saw way back in Week 2. The main difference is we just keep track of nodes rather than partial paths.
- BFS Pseudocode
- Visualization 1
- Visualization 2

```
bfs-from(node v) {
    make a queue of nodes, initially seeded with v

    while (queue not empty) {
        curr = dequeue from queue
        "process" curr
        for each node adjacent to curr {
            if that node hasn't yet been visited, enqueue it
        }
    }
}
```

Breadth-First Search Properties

- Breadth-First Search allows us to find the shortest path/distance between any two nodes in an **unweighted graph**.

- However, BFS doesn't do anything to incorporate edge weights when applied to a weighted graph.

- Most real-world applications of finding the shortest path between two nodes in a graph occur on weighted graphs.

- **How can we improve BFS to take into account edge weights?**

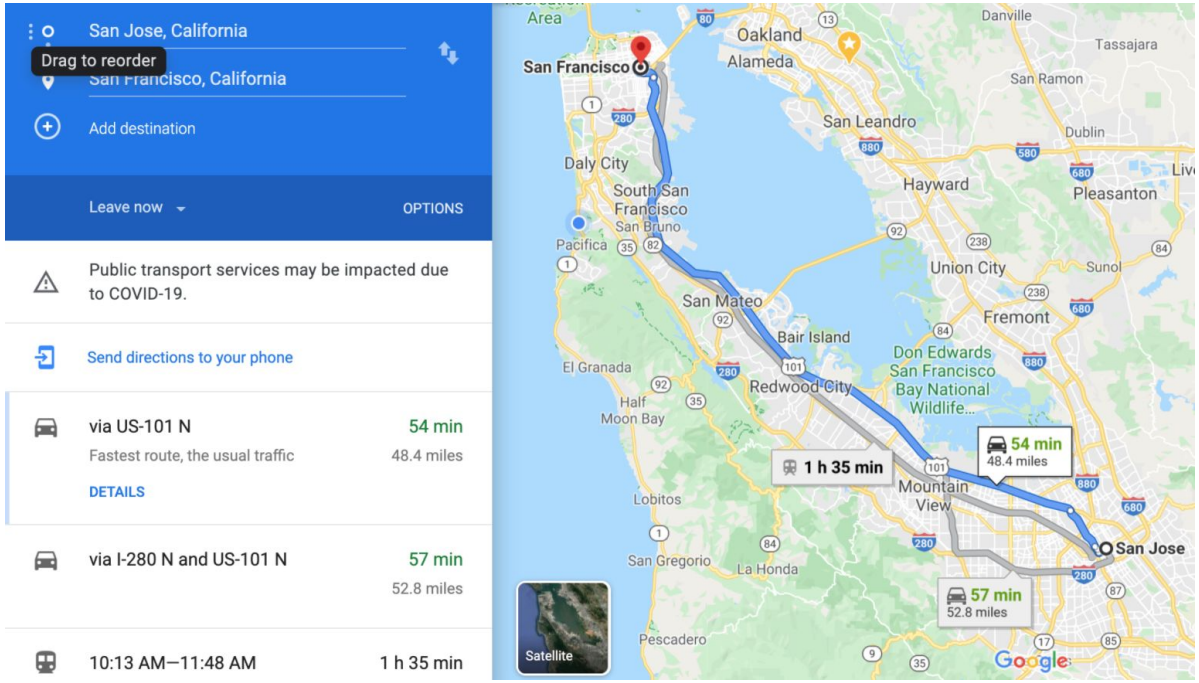
Dijkstra's Algorithm

The Problem

- Let's implement Google Maps!

The Problem

- Let's implement Google Maps!



The Problem

- Let's implement Google Maps!

- As we've previously discussed, a road network can be thought of as a weighted graph between many different destination points.
    - The graph weights are based on many factors including physical distance, traffic, historical data about stop light patterns, etc.

The Problem

- Let's implement Google Maps!

- As we've previously discussed, a road network can be thought of as a weighted graph between many different destination points.
    - The graph weights are based on many factors including physical distance, traffic, historical data about stop light patterns, etc.

- We want to prioritize finding the quickest route between our starting point and our destination point, on this weighted graph.

The Problem

- Let's implement Google Maps!

- As we've previously discussed, a road network can be thought of as a weighted graph between many different destination points.
    - The graph weights are based on many factors including physical distance, traffic, historical data about stop light patterns, etc.

- We want to prioritize finding the quickest route between our starting point and our destination point, on this weighted graph.

- How can we do it?

The Idea

- Rather than simply organizing the nodes in the order in which we visit them, order them by the sum of the weights on the shortest path to that node.

The Idea

- Rather than simply organizing the nodes in the order in which we visit them, order them by the sum of the weights on the shortest path to that node.

- What data structure will be useful for this? A priority queue!

The Idea

- Rather than simply organizing the nodes in the order in which we visit them, order them by the sum of the weights on the shortest path to that node.

- What data structure will be useful for this? A priority queue!

- The seed node (starting point) is enqueued with priority 0. Every subsequent node is enqueued with priority equal to the current node's priority + the weight of the edge being traversed.

The Idea

- Rather than simply organizing the nodes in the order in which we visit them, order them by the sum of the weights on the shortest path to that node.

- What data structure will be useful for this? A priority queue!

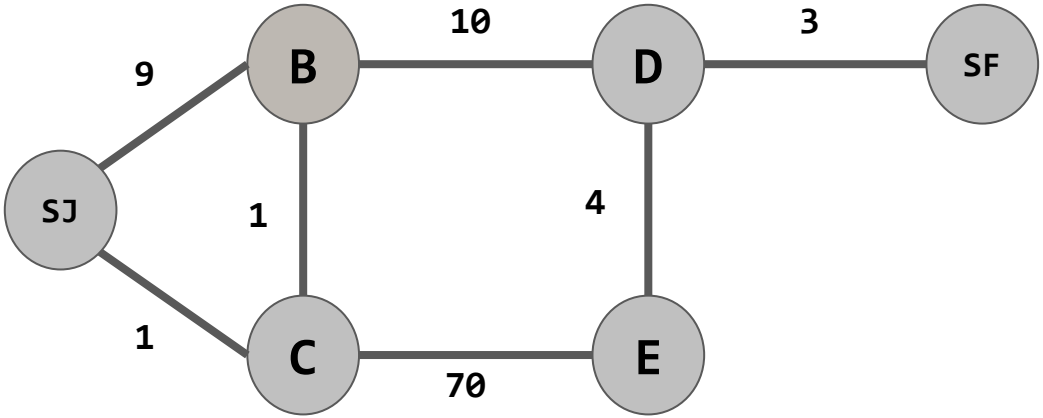
- The seed node (starting point) is enqueued with priority 0. Every subsequent node is enqueued with priority equal to the current node's priority + the weight of the edge being traversed.

- The priority queue guarantees we will visit nodes in order of increasing distance from the seed node.

Dijkstra's Algorithm Pseudocode

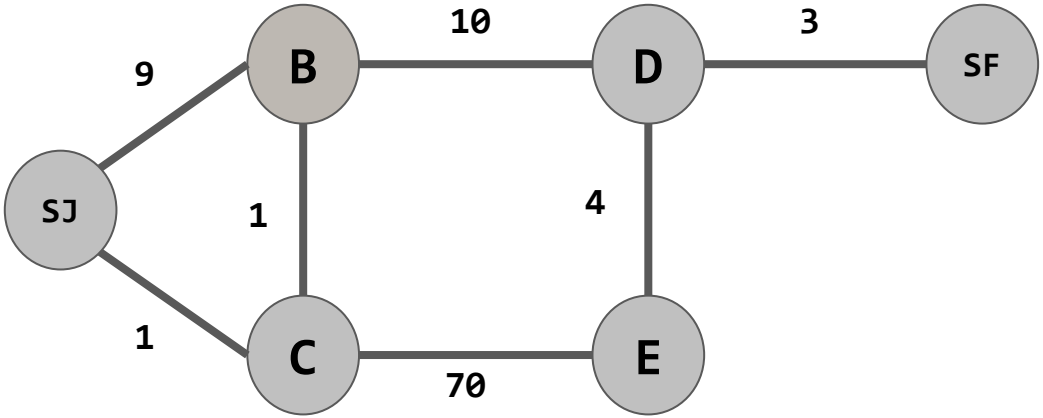
```
dijkstras-from(node v) {
    Initialize an empty priority queue of nodes
    Add v to the priority queue with priority 0

    while (queue not empty) {
        currPriority = peek priority of first element in queue
        curr = dequeue from queue
        "process" curr
        for each node adjacent to curr {
            if that node hasn't yet been visited, enqueue it with priority
            equal to currPriority + edge weight between curr and node

            if that node has been visited and is still in the priority queue,
            update its priority to be currPriority + edge weight
        }
    }
}
```

Dijkstra's In Practice



Goal: Find the shortest path/distance from SJ to SF

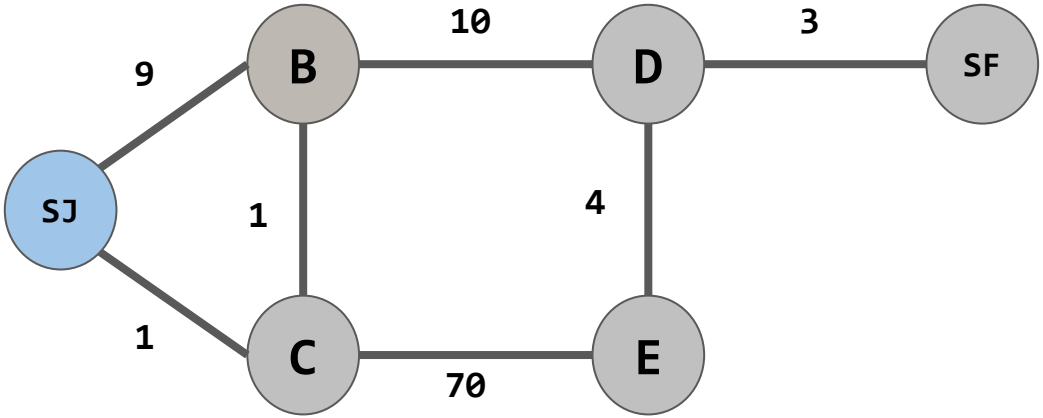
Dijkstra's In Practice



higher priority

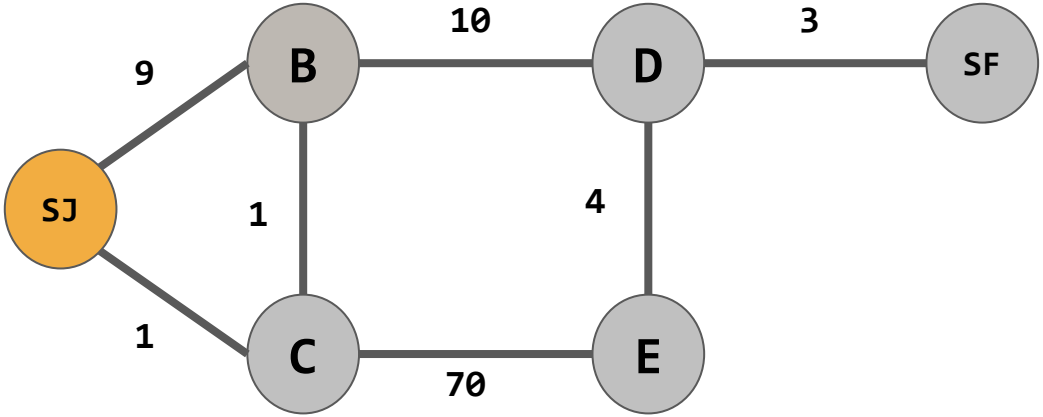
lower priority

Dijkstra's In Practice



{ 0 SJ }

Dijkstra's In Practice



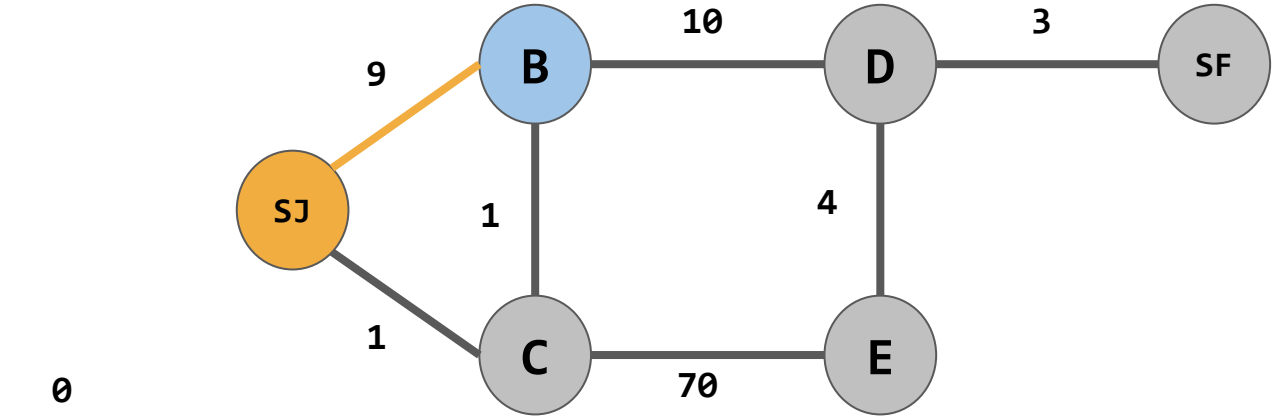
0

SJ

{

}

Dijkstra's In Practice



0

SJ

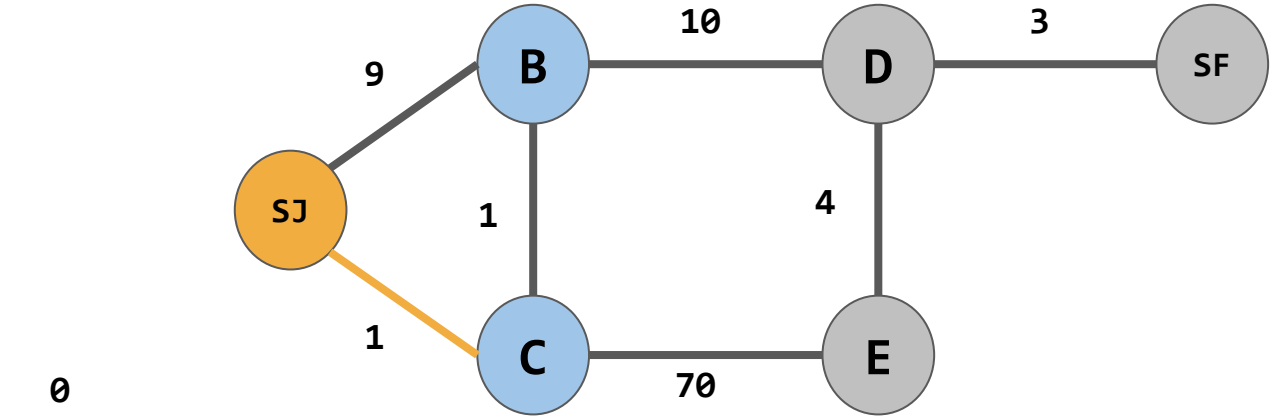
{

9

B

}

Dijkstra's In Practice

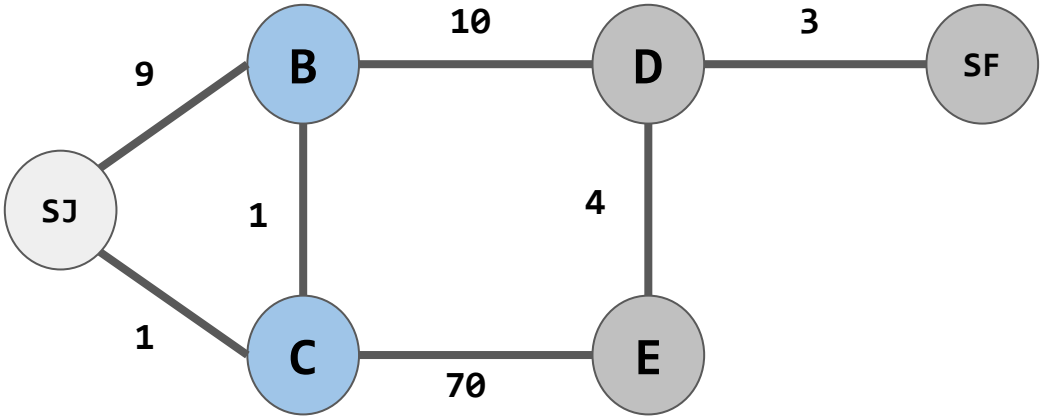


0
SJ

1
C

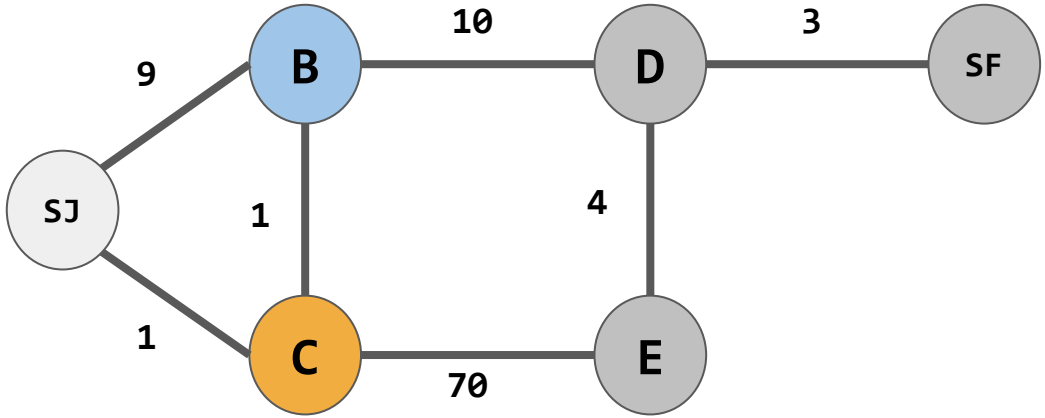
9
B

Dijkstra's In Practice



{ 1 C 9 B }

Dijkstra's In Practice



1

C

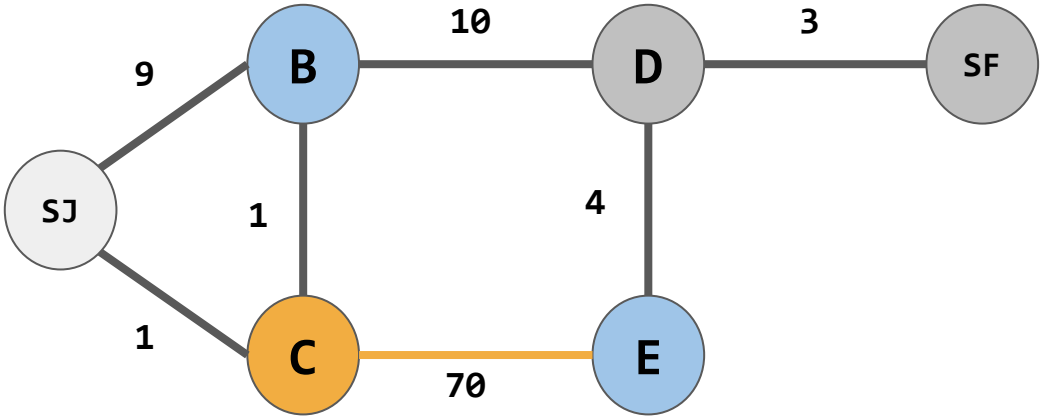
{

9

B

}

Dijkstra's In Practice



1

C

{

9

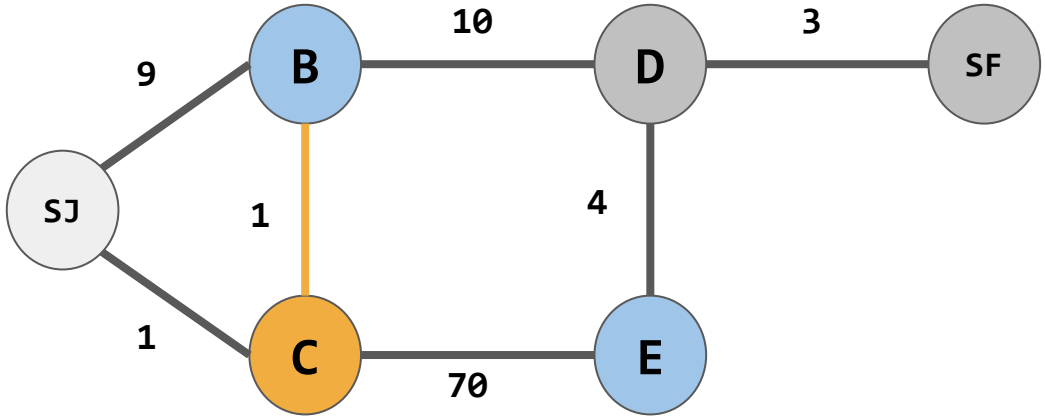
B

71

E

}

Dijkstra's In Practice



1

C

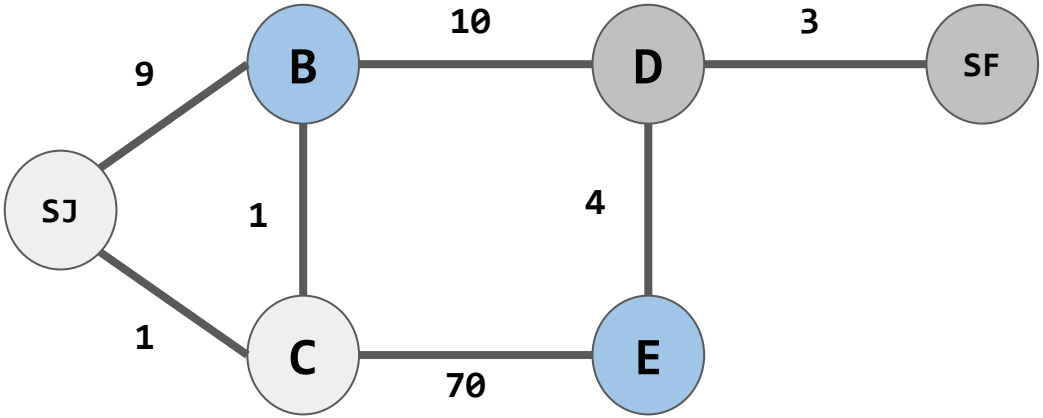
2

B

71

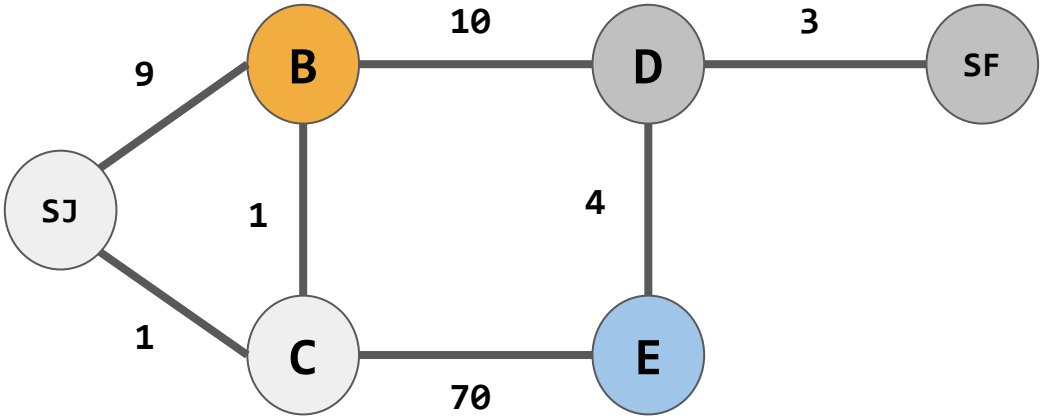
E

Dijkstra's In Practice



{ 2 B 71 E }

Dijkstra's In Practice



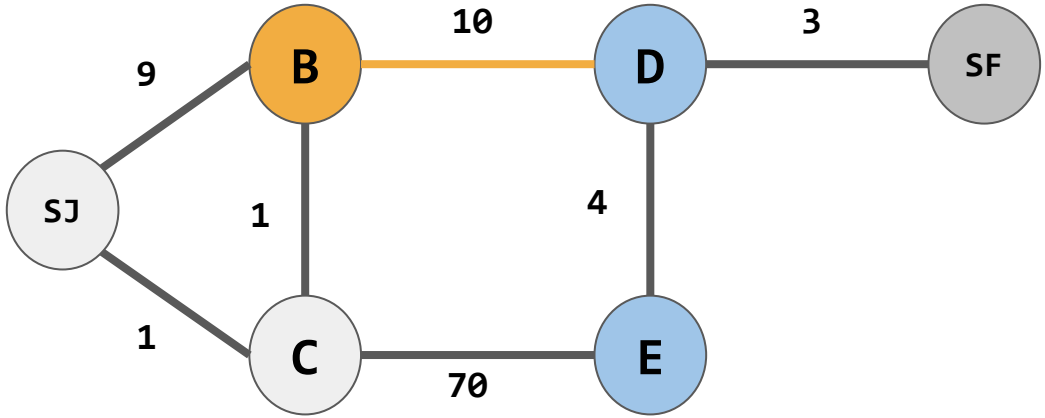
2

B

71

E

Dijkstra's In Practice



2

B

{

12

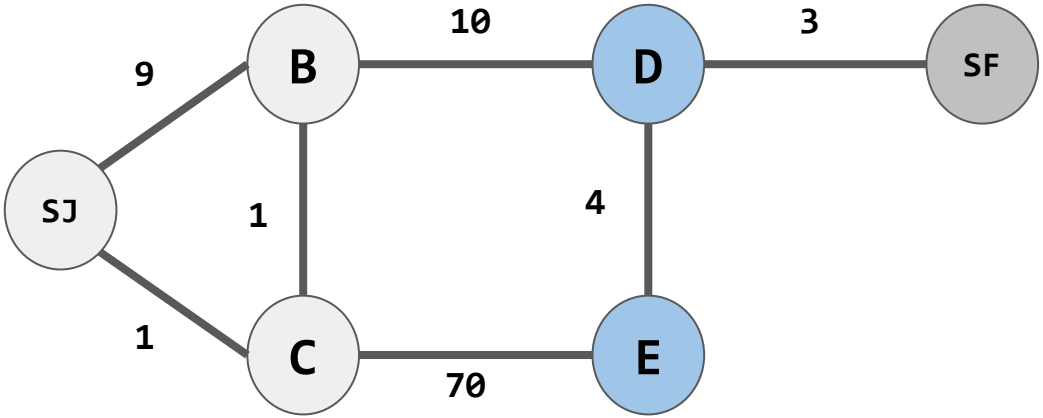
D

71

E

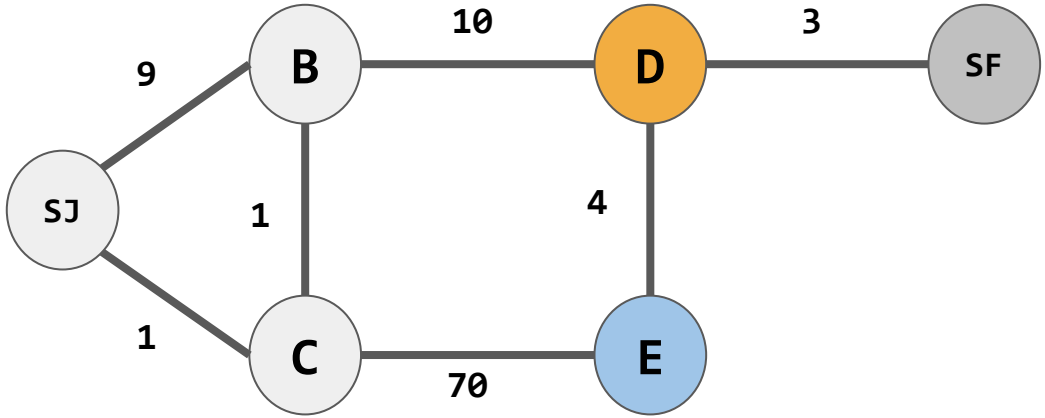
}

Dijkstra's In Practice



{ 12 D, 71 E }

Dijkstra's In Practice



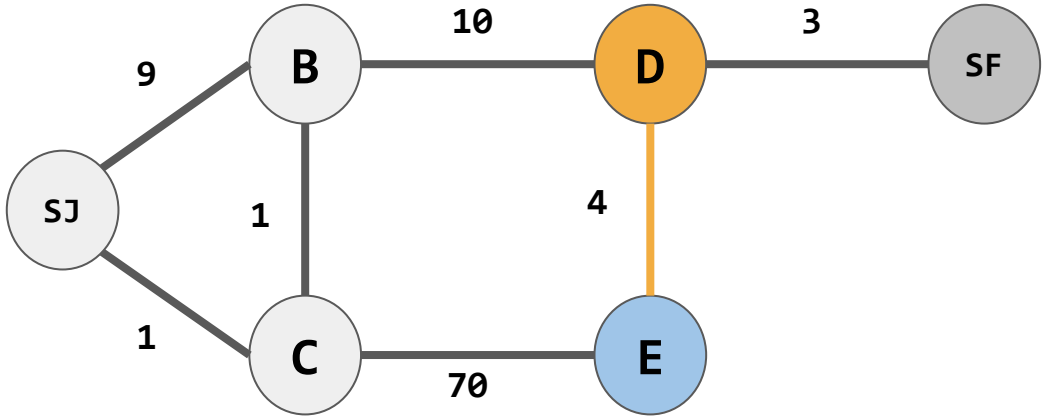
12

D

71

E

Dijkstra's In Practice



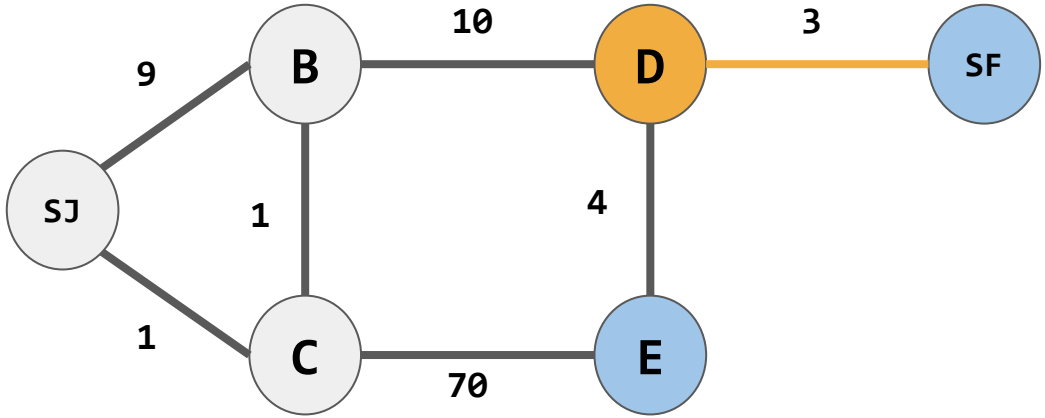
12

D

16

E

Dijkstra's In Practice



12

D

{

15

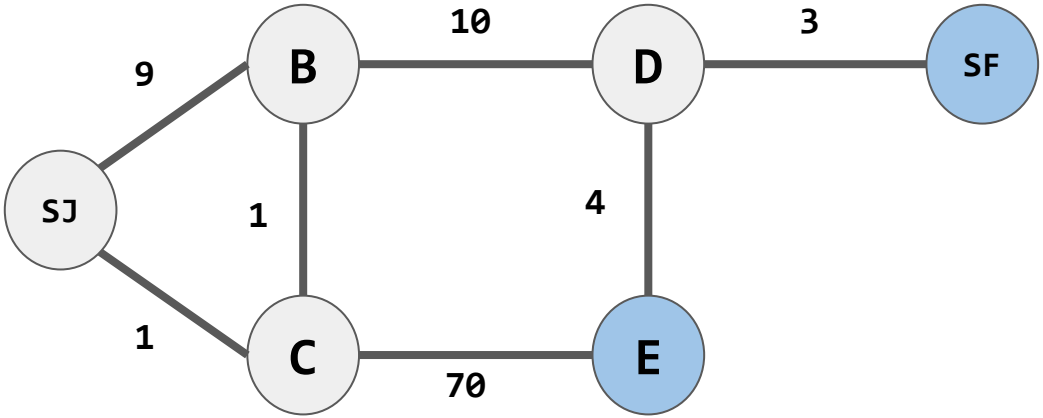
SF

16

E

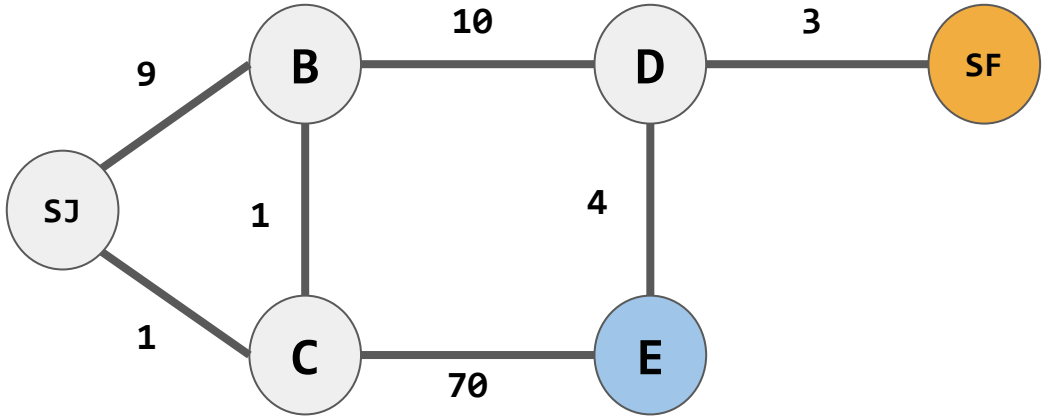
}

Dijkstra's In Practice



{ 15 SF 16 E }

Dijkstra's In Practice



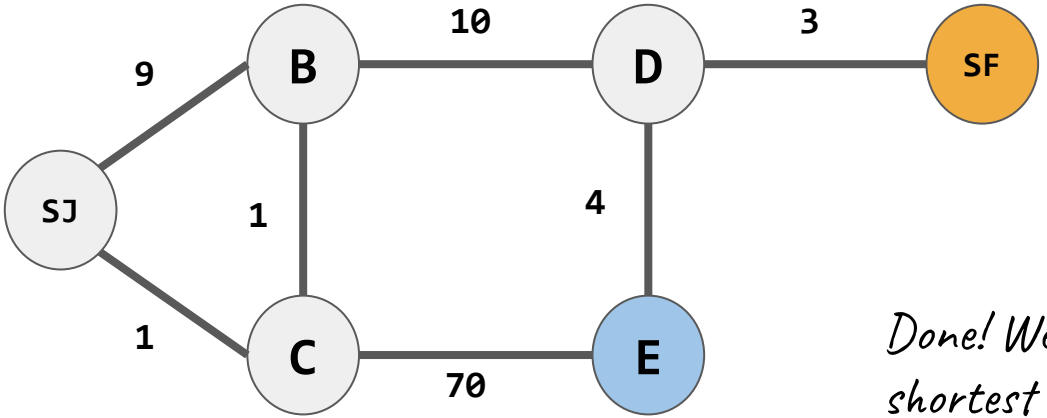
15

SF

16

E

Dijkstra's In Practice



15

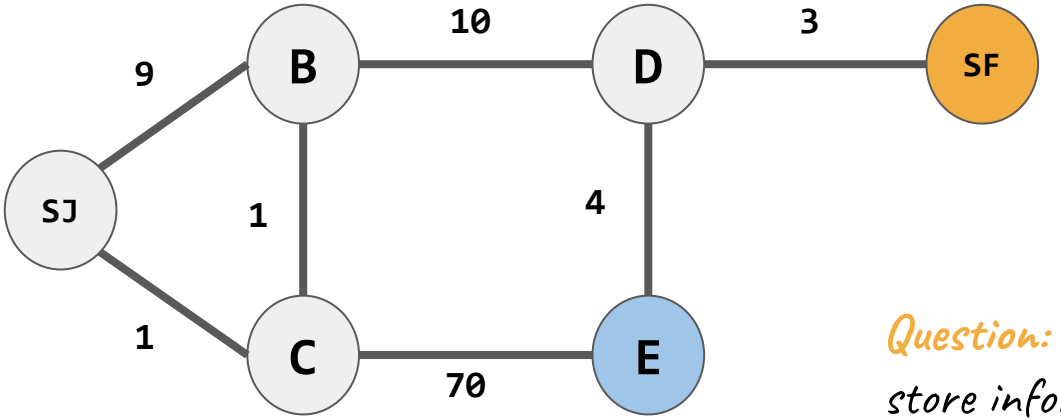
SF

16

E

Done! We know the shortest path from SJ to SF has a total path weight of 15.

Dijkstra's In Practice



15

SF

16

E

Question: How would you store information along the way to be able to reconstruct the path?

Dijkstra's Algorithm Properties

- Dijkstra's Algorithm allows us to find the shortest path/distance between any two nodes in a **weighted graph**.

- Dijkstra's Algorithm forms the basis of many powerful real-world systems that are built on top of graphs!

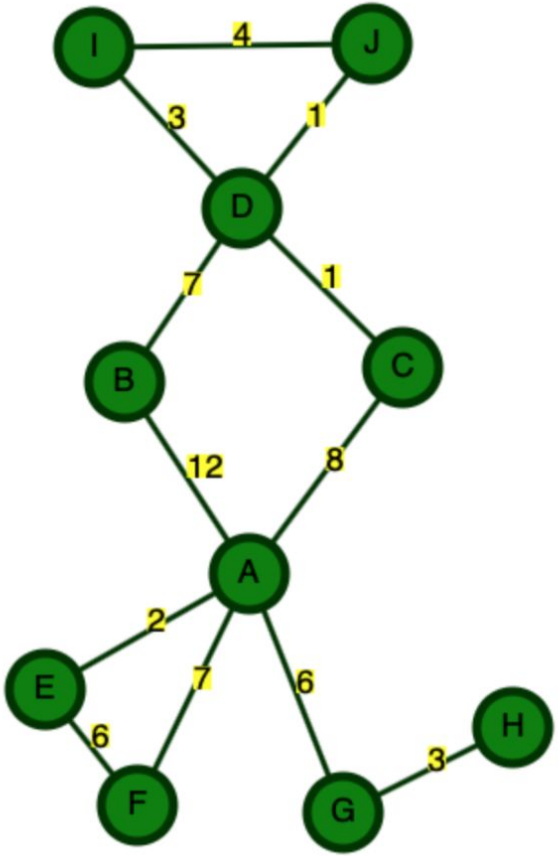
- However, one of the downsides to Dijkstra's algorithm is that it can, in many circumstances, ignore **other sources of information** that might prove useful to finding the shortest path in the fewest number of steps.

- Can we find the solution while using less steps than with Dijkstra's Algorithm?

A* Search

A* Search

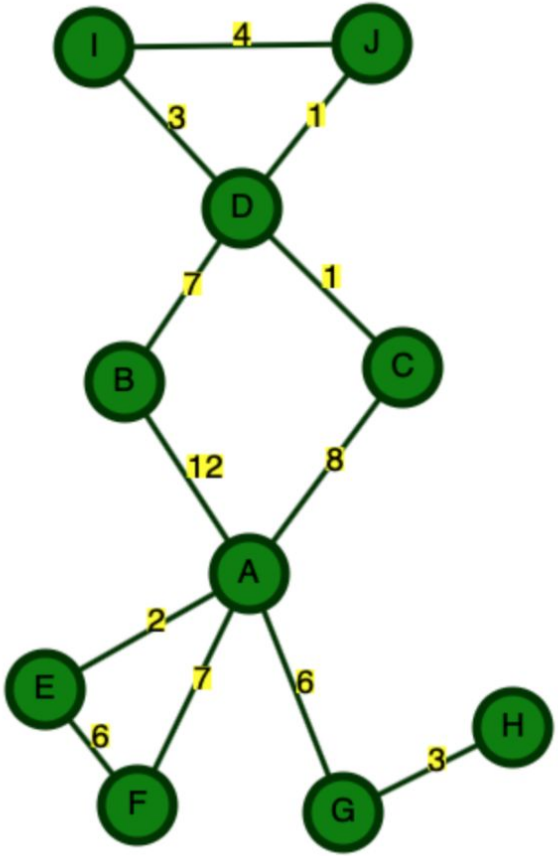
- Suppose we wanted to find the shortest path from **A** to **J** in the graph to the right.
  - Given no other information, we can do no better than using Dijkstra's.



A* Search

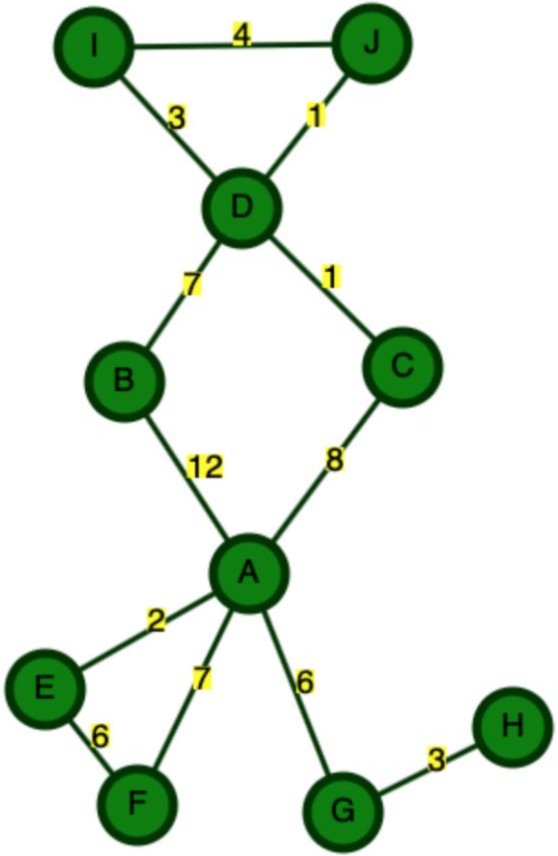
- Suppose we wanted to find the shortest path from **A** to **J** in the graph to the right.
  - Given no other information, we can do no better than using Dijkstra's.

- But, if we know that this graph represents a map, we can start **reasoning about cardinal directions**.



A* Search

- Suppose we wanted to find the shortest path from **A** to **J** in the graph to the right.
  - Given no other information, we can do no better than using Dijkstra's.
- But, if we know that this graph represents a map, we can start **reasoning about cardinal directions**.
- **Idea:** If we our goal is to go north from A to J, exploring paths to the south probably doesn't make sense!



Heuristics

- We call the idea of using external information about a graph a **heuristic**.
  - The heuristic estimates the cost of the cheapest path to the goal.
  - It is different for every problem and corresponds to some real-world information.

Heuristics

- We call the idea of using external information about a graph a **heuristic**.

- A heuristic should always **underestimate the distance to the goal**.
  - If it overestimates the distance, it could end up finding a solution that is not actually optimal (though it will do so relatively fast).

Heuristics

- We call the idea of using external information about a graph a **heuristic**.
- A heuristic should always **underestimate the distance to the goal**.
- We use the heuristic as an **addition to the value for the priority**.
  - For the case of maps, if the distance to the destination is closer, this will weight the nodes in that direction to be preferable (i.e., they will actually have a smaller numerical priority value).
  - In other words, **`priority(u) = weight(s, u) + heuristic(u, d)`**, where **`s`** is the start, **`u`** is the node we are considering, and **`d`** is the destination.

Heuristics

- We call the idea of using external information about a graph a **heuristic**.
- A heuristic should always **underestimate the distance to the goal**.
- We use the heuristic as an **addition to the value for the priority**.
- Common heuristics for distance-based graphs include Manhattan distance, as-the-crow-flies distance, and Chebyshev distance.

Graph Search Demo

https://qiao.github.io/PathFinding.js/visual/

Beyond Traversal

More Graph Algorithms

- There are many, many different graph algorithms out there.

More Graph Algorithms

- There are many, many different graph algorithms out there.
- Some famous examples include:

More Graph Algorithms

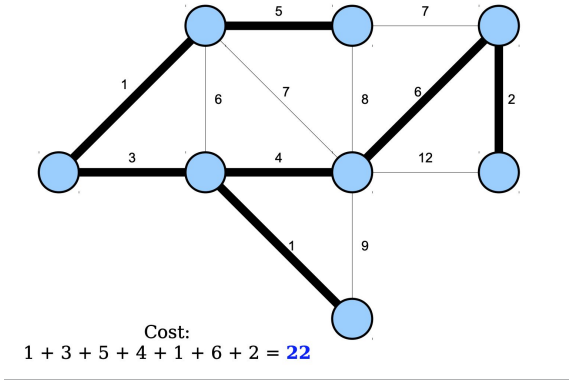
- There are many, many different graph algorithms out there.

- Some famous examples include:
  - **BFS, Dijkstra's algorithm, and A* Search**: Find the shortest path between two nodes in a graph.

More Graph Algorithms

- There are many, many different graph algorithms out there.

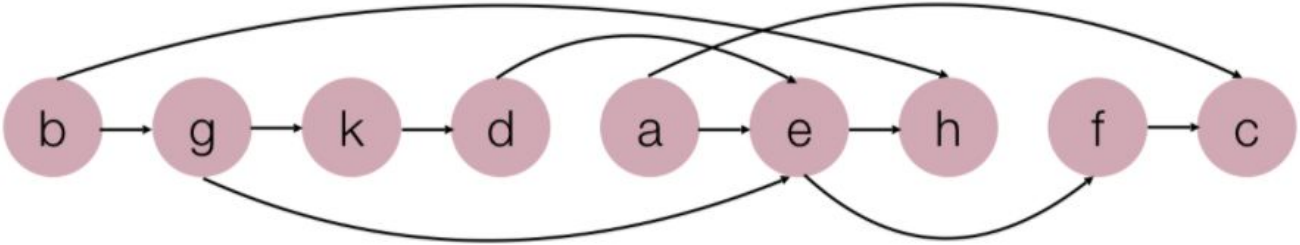
- Some famous examples include:
  - **BFS, Dijkstra's algorithm, and A* Search**: Find the shortest path between two nodes in a graph.
  - **Kruskal's Algorithm**: Find a minimum spanning tree from a given graph.



More Graph Algorithms

- There are many, many different graph algorithms out there.

- Some famous examples include:
  - **BFS, Dijkstra's algorithm, and A* Search**: Find the shortest path between two nodes in a graph.
  - **Kruskal's Algorithm**: Find a minimum spanning tree from a given graph.
  - **Topological Sort**: "Sort" the nodes in a dependency graph in such a way that traversing the nodes in order results in all dependencies being fulfilled at each point in time.



More Graph Algorithms

- There are many, many different graph algorithms out there.

- Some famous examples include:
  - **BFS, Dijkstra's algorithm, and A* Search**: Find the shortest path between two nodes in a graph.
  - **Kruskal's Algorithm**: Find a minimum spanning tree from a given graph.
  - **Topological Sort**: "Sort" the nodes in a dependency graph in such a way that traversing the nodes in order results in all dependencies being fulfilled at each point in time.
  - **Traveling salesman**: Given a map of cities and the distances between them, find the shortest path that traverses all cities in the map.

More Graph Algorithms

- There are many, many different graph algorithms out there.

- Some famous examples include:
  - **BFS, Dijkstra's algorithm, and A* Search**: Find the shortest path between two nodes in a graph.
  - **Kruskal's Algorithm**: Find a minimum spanning tree from a given graph.
  - **Topological Sort**: "Sort" the nodes in a dependency graph in such a way that traversing the nodes in order results in all dependencies being fulfilled at each point in time.
  - **Traveling salesman**: Given a map of cities and the distances between them, find the shortest path that traverses all cities in the map.

- Graphs can also be used in conjunction with machine learning algorithms to accomplish cool things. Take CS224W to learn more!

Summary

Graphs Summary

- Graphs are the most powerful and flexible manner for organizing data in a linked data structure, particular when expressing complex patterns and relationships between different data entities.

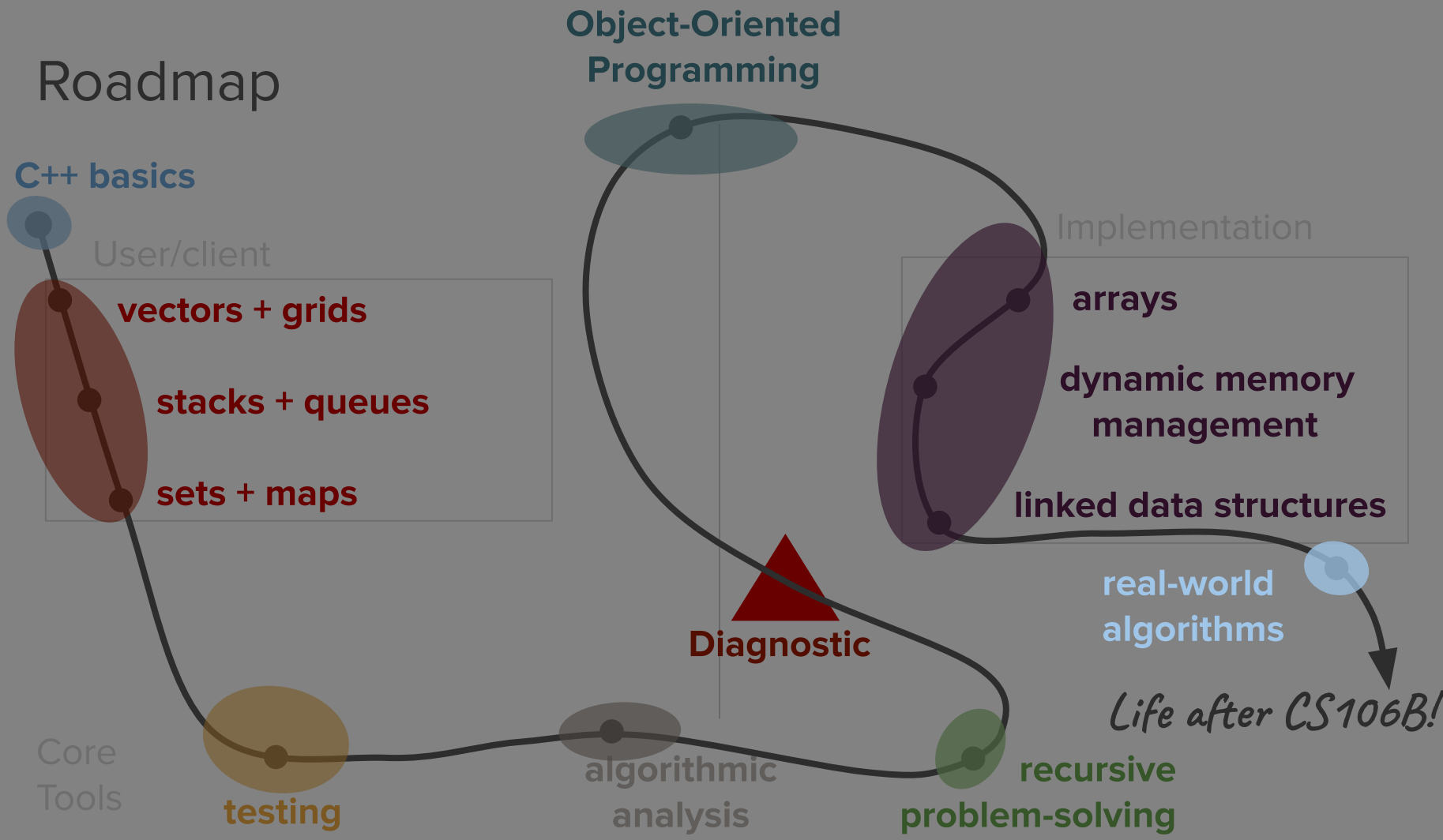
- Graphs are composed of nodes connected by edges.

- Graphs can be directed, undirected, weighted, or unweighted.

- Graph algorithms can be used to find interesting properties of graphs. BFS, Dijkstra's Algorithm, and A* Search are three ways to find the shortest path between two nodes in a graph.

What’s next?

Roadmap


Object-Oriented Programming
C++ basics
User/client
vectors + grids
stacks + queues
sets + maps
Implementation
arrays
dynamic memory management
linked data structures
real-world algorithms
Diagnostic
Life after CS106B!
Core Tools
testing
algorithmic analysis
recursive problem-solving

Multithreading and Parallel Computing