Dictionaries, Tuples, Graphs

6.100 LECTURE 11

SPRING 2026

Announcements

- Pset 3 due tonight
- Pset 3 checkoffs start tomorrow, due next Monday 3/16
- Pset 4 released after class, due next Friday 3/20

- Spring break is 3/23–3/27
- Exam 2 on Monday 4/6, one full week after spring break

Dictionaries

Python `dict` overview

- A mapping type from keys to values
  - https://docs.python.org/3/library/stdtypes.html#mapping-types-dict
  - https://docs.python.org/3/tutorial/datastructures.html#dictionaries
- Square bracket syntax connotes analogy to list indexing
  - **`list`**s have implicit sequential indicies
  - **`dict`**s have explicit key labels
- Keys can be (almost) any Python object
  - so ordering no longer makes sense
- **`dict`** objects in memory are conceptually two-column tables
  - left column contains references to keys
  - right column contains references to values associated with those keys
  - **like `lists`, no objects stored in `dicts`**

`dict` operations

- object creation
  - `{}` is empty `dict`
  - `{key1: val1, key2: val2}` is `dict` literal
  - `dict()` constructor copies any `dict` passed in
    - or creates new `dict` from sequence of `[key, value]` pairs
  - `len(`*`dict`*`)`
  - *`dict`*`.copy()`
  - *`dict`*`.clear()`
- key and value retrieval
  - `key in` *`dict`*
  - *`dict`*`[key]` → `value`
  - *`dict`*`.get(key, default)` → `value`
    - if `key` not in *`dict`*, returns `default` instead of raising `KeyError`

`dict` mutating operations

- add or update key-value pair
  - ***dict*`[key] = value`**
- delete key-value pair
  - **`del` *`dict`*`[key]`**
  - ***dict*`.pop(key, default)`**
- merge with other dictionaries
  - ***dict*` | other`**
  - ***dict*` |= other`**
  - ***dict*`.update(other)`**

Immutability of `dict` keys

- **`dict` keys must be hashable**
  - for Python's built-in types, hashable basically means **immutable**
- When you associate a key with a value:
  - expect to be able to retrieve the value by looking up with an equivalent (==) key, no matter how it was constructed
  - if keys are mutable (e.g., lists), code that runs after a key mutation may be unaware that the key has changed

Common types for `dict` keys

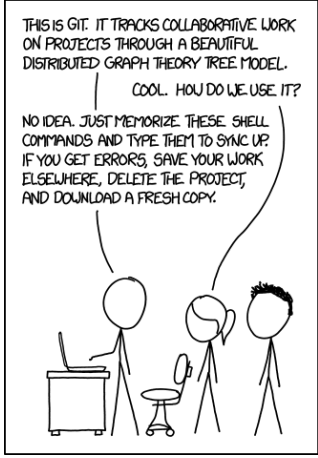
- **`int`**s
  - **`bool`**s are really **`int`**s underneath the hood, so avoid those
- **`float`**s are a bad idea
  - mathematically equivalent expressions may not yield equivalent **`float`**s
- **`str`**s are a great idea
  - natural labels
  - this is one reason why Python's **`str`**s are immutable
- **`tuple`**s are also good
  - **`tuple`**s are just like **`list`**s, but immutable
  - for a **`tuple`** to be hashable, all its nested contents must also be immutable

Iterating over `dicts`

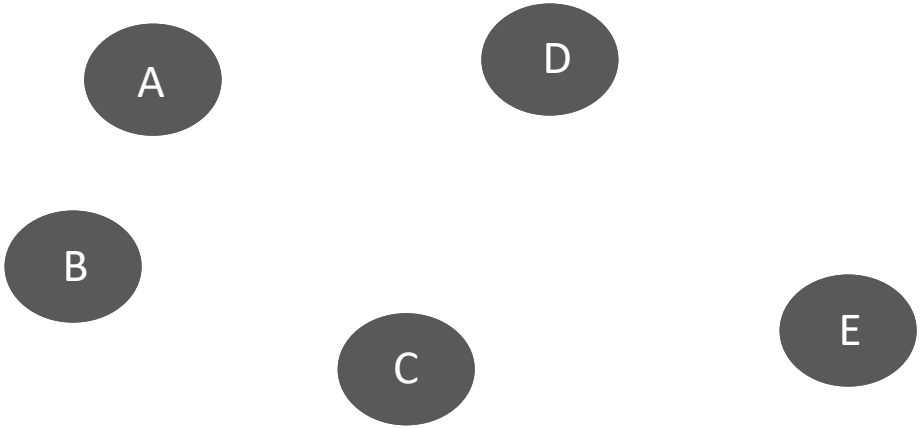
- No inherent ordering of **`dict`** keys, unlike **`list`** indicies
  - but still wish to retrieve all elements
  - nature of code/time means have to do so sequentially
- Python's **`for`** directly iterates over **`dict`** keys
  - ```
    for key in dict:
        value = dict[key]
    ```
  - **`list(`*`dict`*`)`** will product a **`list`** of ***`dict`***'s keys
- Can also iterate over dictionary views
  - ***`dict`*`.keys()`**
  - ***`dict`*`.values()`**
  - ***`dict`*`.items()`** → produces **`(key, value)`** tuples
    ```

Graphs

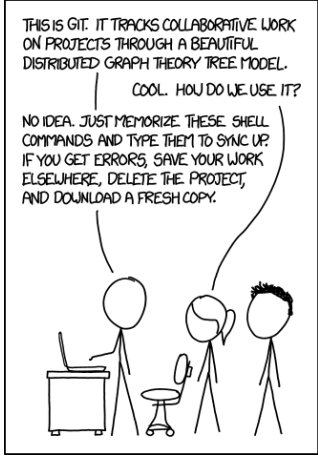
What is a Graph?



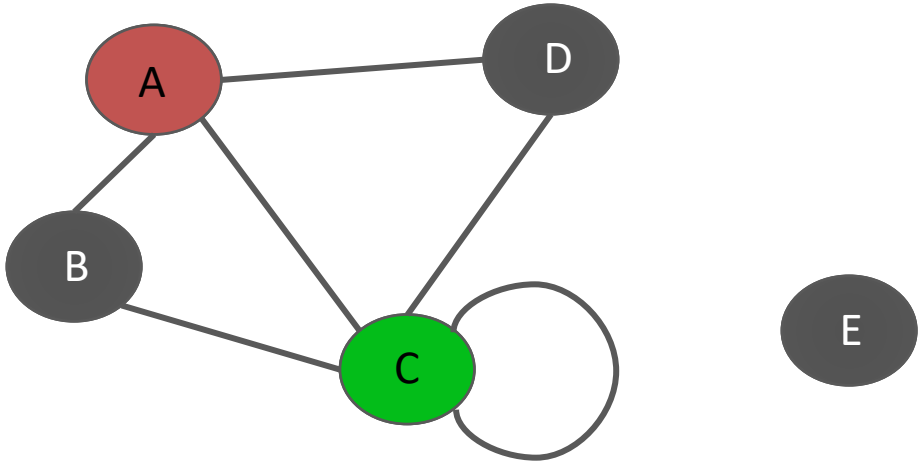
- Set of nodes (vertices)



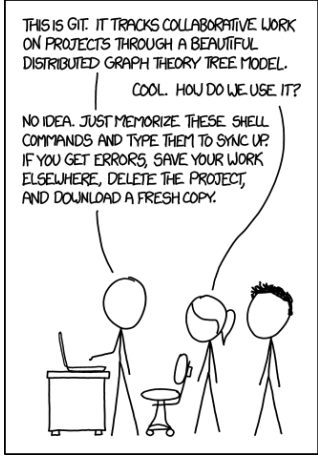
What is a Graph?



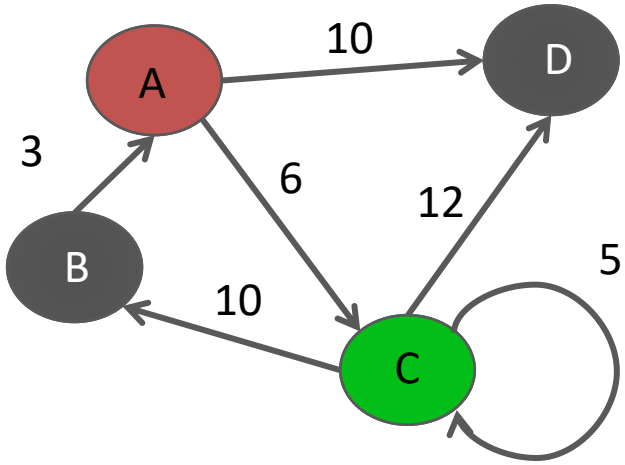
- Set of nodes (vertices)
  - Might have associated names or properties
- Set of edges (arcs) each connecting a pair of nodes
  - Undirected (graph)



What is a Graph?



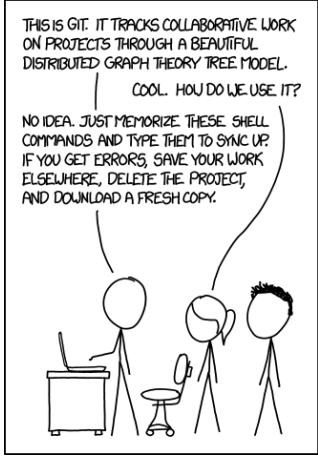
- Set of nodes (vertices)
  - Might have properties associated with them
- Set of edges (arcs) each connecting a pair of nodes
  - Undirected (graph)
  - Directed (digraph)
    - Source (parent) and destination (child) nodes
  - Unweighted or weighted
    - Assume non-negative



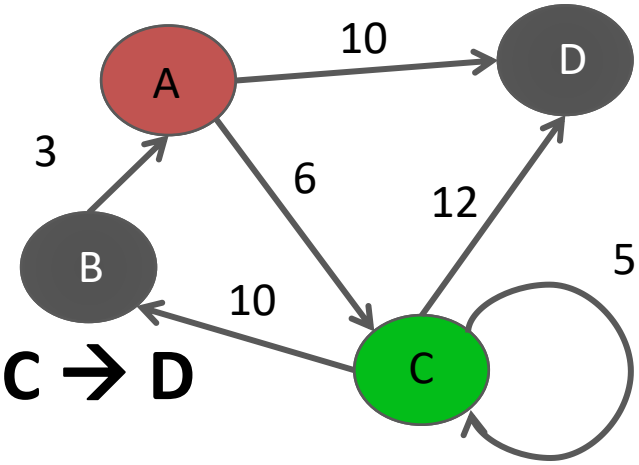
Graph:

- might not be completely connected
- could have loops, both single length and longer

What is a Graph?



- Set of nodes (vertices)
  - Might have properties associated with them
- Set of edges (arcs) each connecting a pair of nodes
  - Undirected (graph)
  - Directed (digraph)
    - Source (parent) and destination (child) nodes
  - Unweighted or weighted
    - Assume non-negative

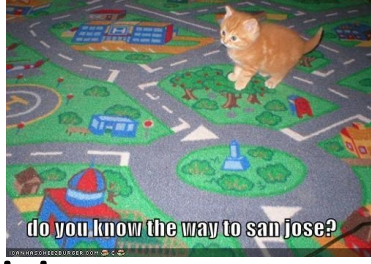


Path: B → A → C → D

Graph:

- might not be completely connected
- could have loops, both single length and longer

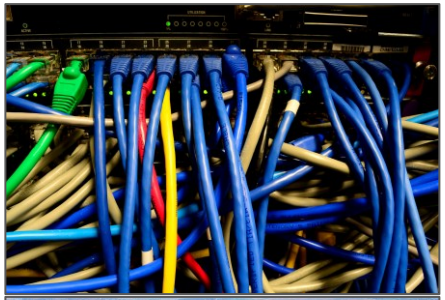
Graphs

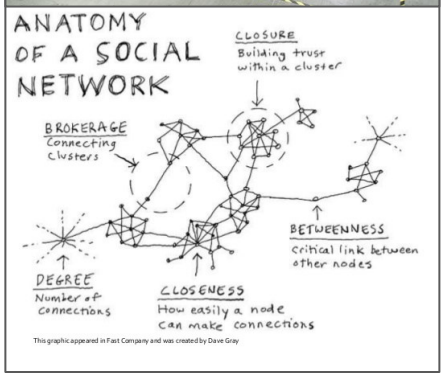


- Capture and reason about relationships among entities
  - Routes between Boston and San Jose
  - How the atoms in a molecule are related to one another
  - Ancestral relationships (family trees)
  - Business/social/political connections
  - Functions and expressions in python
  - …

Graphs model a wide range of systems

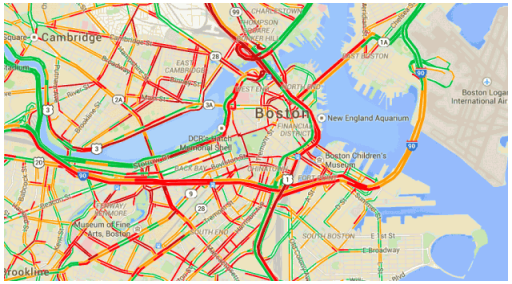
- Computer networks
  - Efficiently route information from one node to another, over large set of packets?
- Transportation networks
  - Efficiently get to a particular destination?
- Logistics networks
  - How can I efficiently move products to destinations in a warehouse or between centers?
- Social networks
  - How can I understand diffusion of misinformation, identify clusters of people with similar characteristics



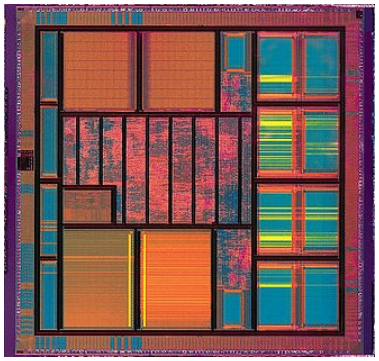


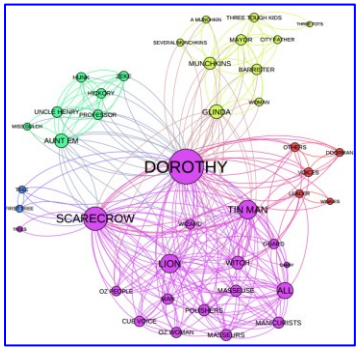
Why Graphs Are So Useful

- Not only do graphs capture relationships in connected networks of items, they support **inference** on those structures



- Find sequences of links between elements (aka the **path problem**)
- Find least expensive path between elements (aka the **shortest path problem**)
- Partition graph into k (equal) sized subgraphs with minimal connections between them (aka **graph partition problem** or **graph clique problem**)
- Find the most efficient way to separate sets of connected elements (aka the **min-cut/max-flow problem**)





You'll see these problems in 6.120, 6.121, 6.101, and other classes

Next time

- Wednesday: finding paths on graphs
  - Pset 4 largely builds on this