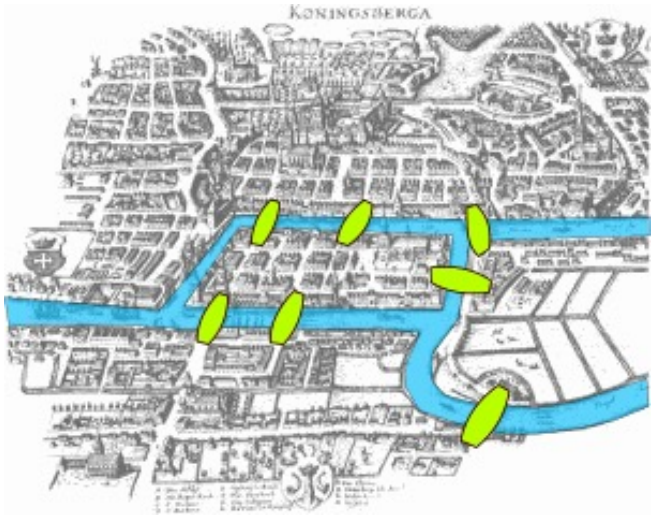


Graphs

(download slides and .py files to follow along)

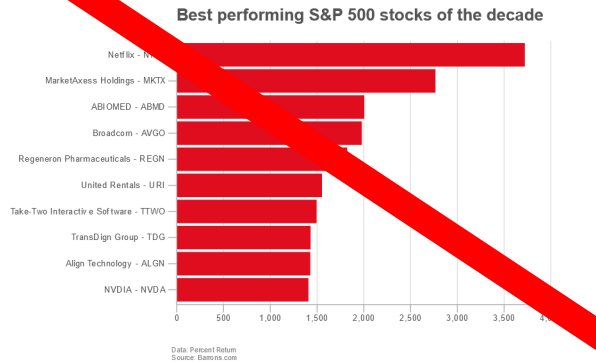
Tim Kraska

MIT Department Of Electrical Engineering and Computer Science

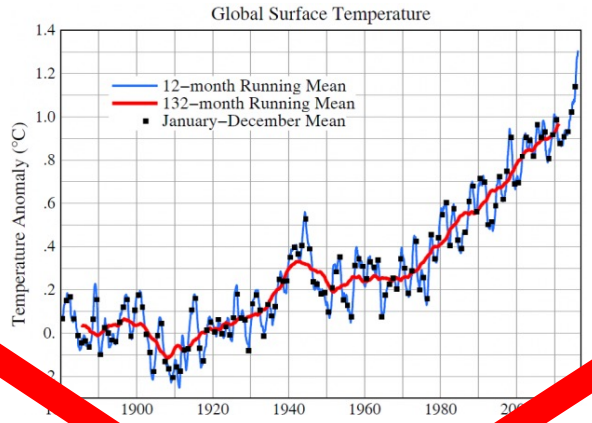
Topics

- Last week
  - Dictionaries
  - Tuples
  - Mutation
  - Alias
- Today
  - graph models and how to implement
  - depth-first search

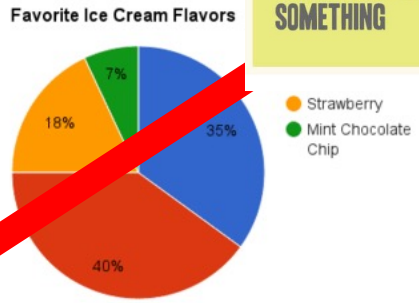
What is a Graph?



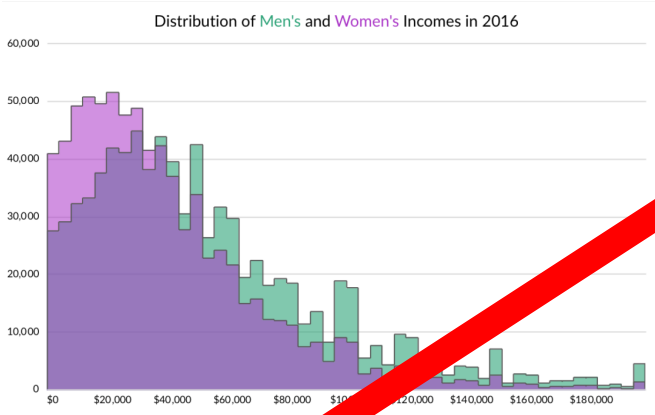
bar



line

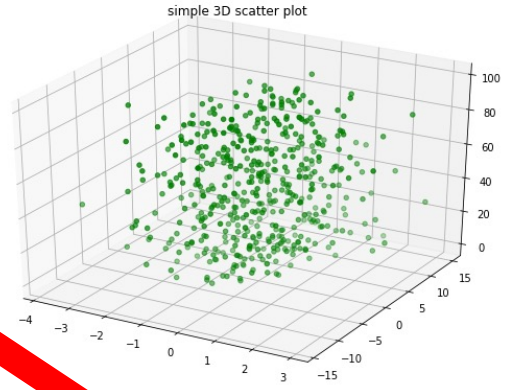


pie

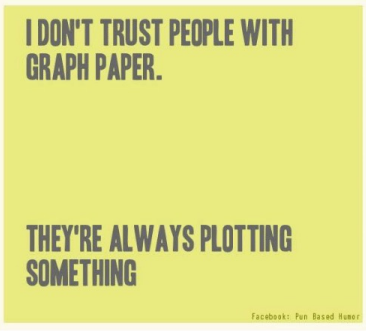


histogram

radar

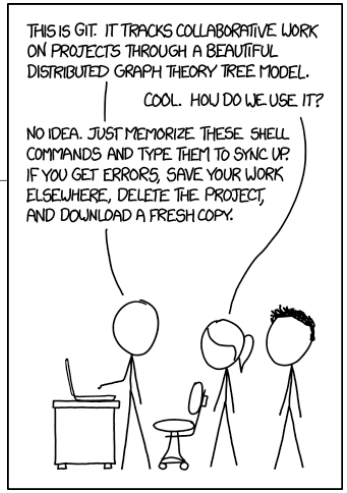


scatter plot

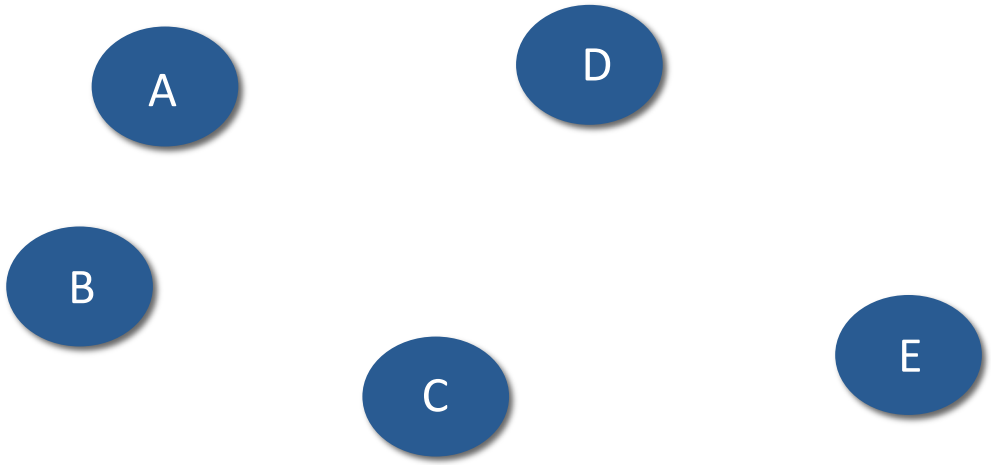


These are all visual presentations of information; we want a structure that supports computation or inference

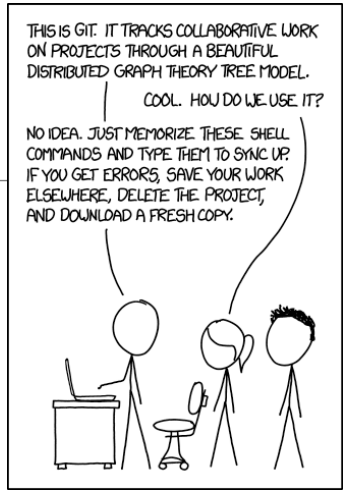
What is a Graph?



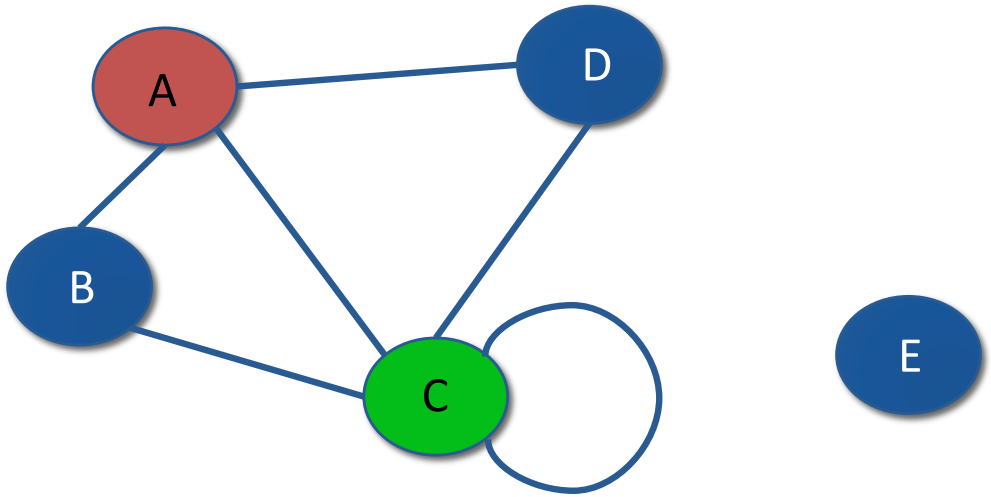
- Set of nodes (vertices)



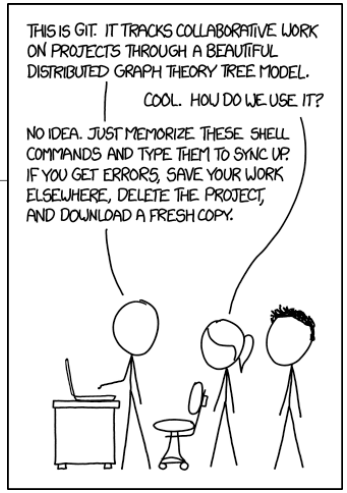
What is a Graph?



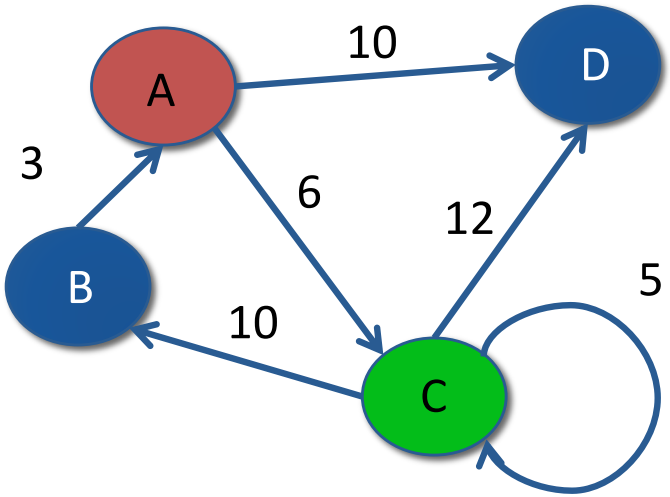
- Set of nodes (vertices)
  - Might have associated names or properties
- Set of edges (arcs) each connecting a pair of nodes
  - Undirected (graph)



What is a Graph?



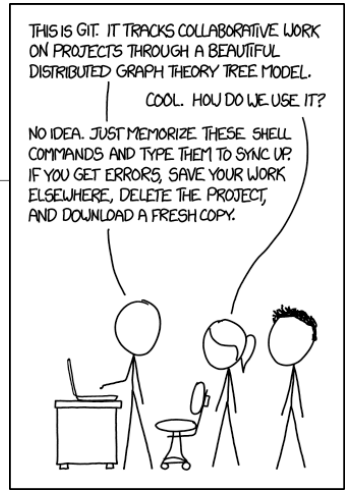
- Set of nodes (vertices)
  - Might have properties associated with them
- Set of edges (arcs) each connecting a pair of nodes
  - Undirected (graph)
  - Directed (digraph)
    - Source (parent) and destination (child) nodes
  - Unweighted or weighted
    - Assume non-negative



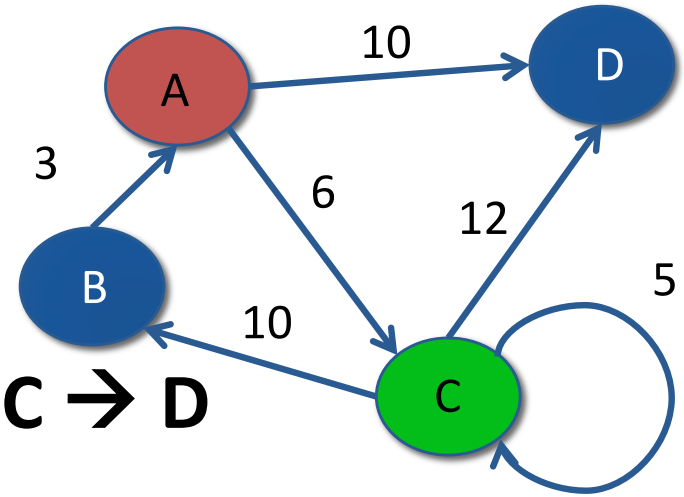
Graph:

- 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



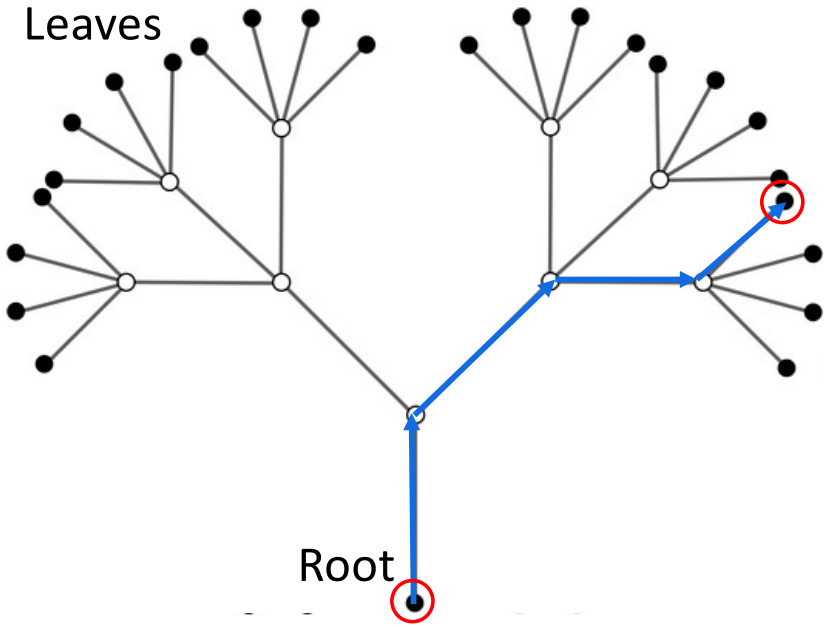
Path: B → A → C → D

Graph:

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

Trees: An Important Special Case

- A special kind of directed graph in which any pair of nodes is connected by a single path from the node closer to the root to the node further from the root



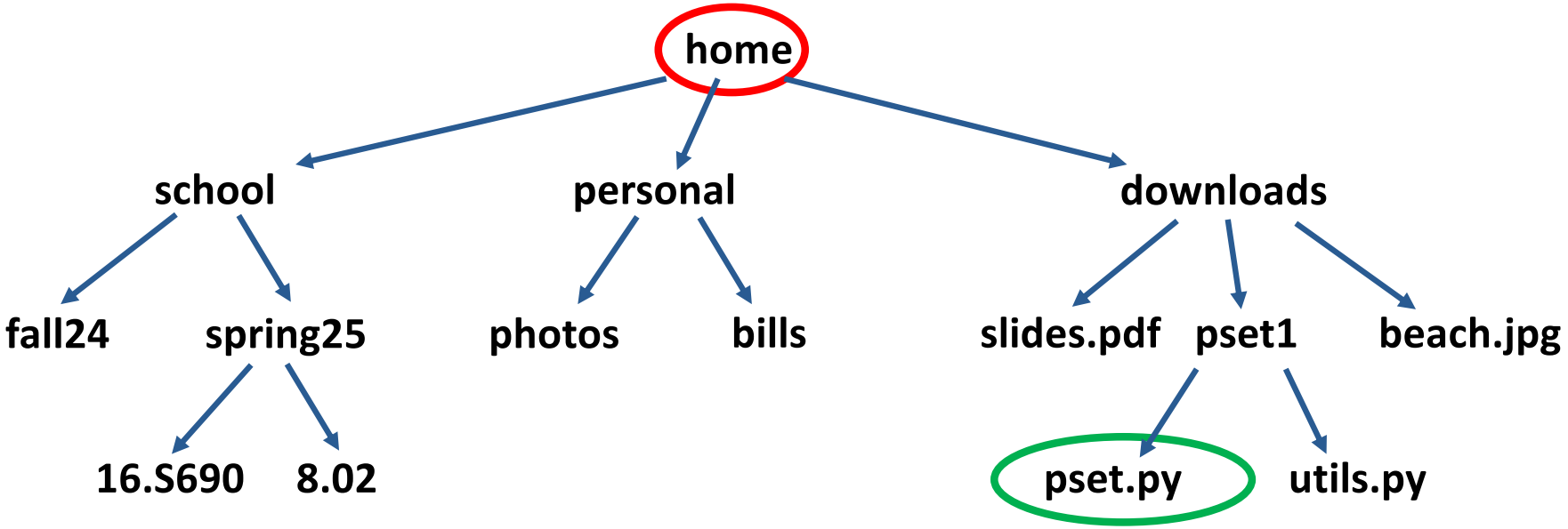
Unique path from root to each node

Computer scientists draw trees with root at top

Example tree

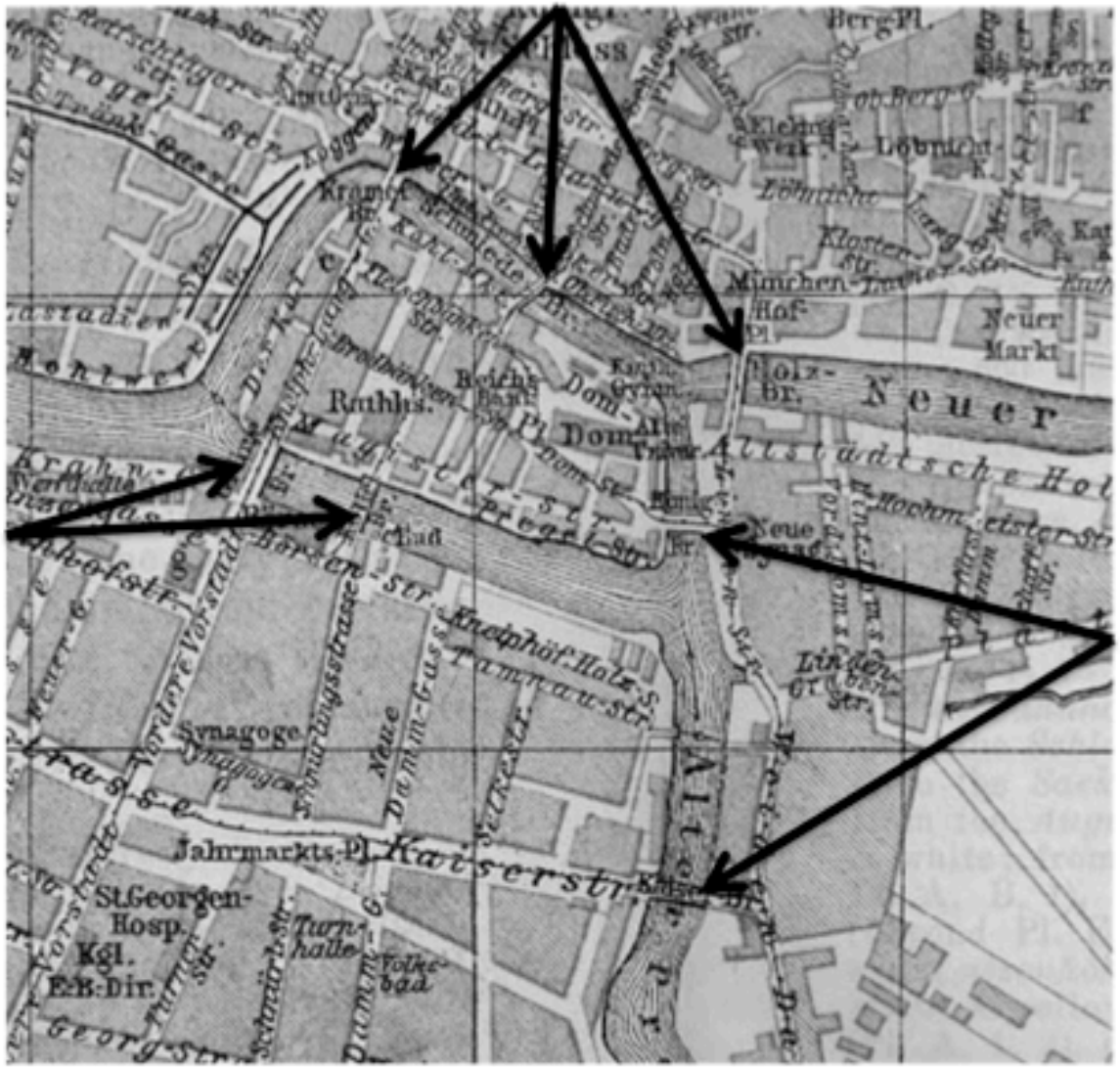
Question:
Where did I put my pset file?

Filesystem is a tree



First Reported Use of Graph Theory

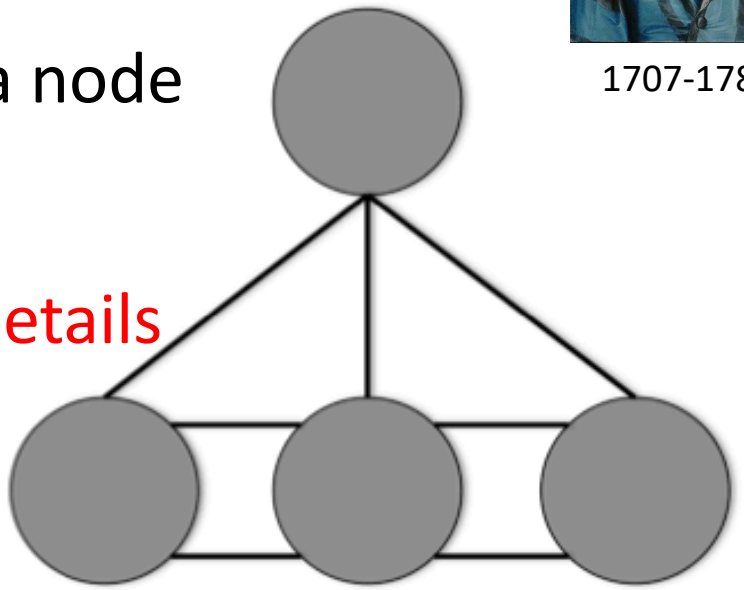
- Bridges of Königsberg problem (1735)
  - Known today as Kaliningrad
  - Two islands plus two mainland portions of city connected by 7 bridges
- Is it possible to take a walk that traverses each of the 7 bridges exactly once?



Leonhard Euler's Model

1707-1783

- Each island (or side of mainland) a node
- Each bridge an undirected edge
- Model abstracts away irrelevant details
  - Size of islands
  - Length of bridges

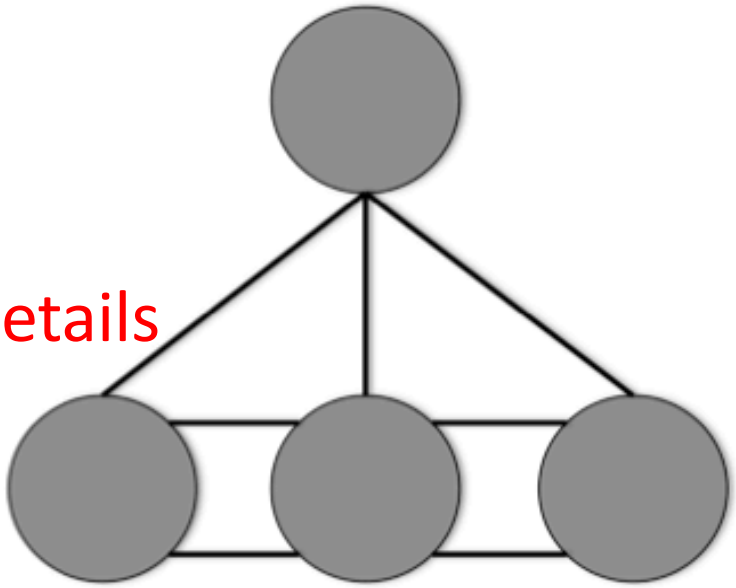


  - All that matters are the connections between nodes
- Is there a path that contains each edge exactly once?

Poll:
Do you think that such a path exists?

Leonhard Euler's Model

- Each island a node
- Each bridge an undirected edge
- Model abstracts away irrelevant details
  - Size of islands
  - Length of bridges



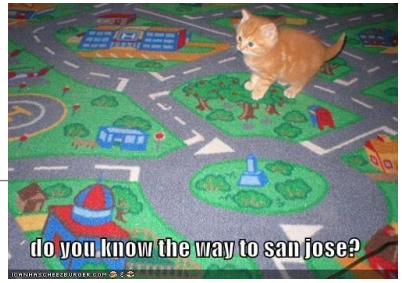
- Is there a path that contains each edge exactly once?
  - **No!**
    - For such a path to exist, each node except the first and last must have an even number of edges
    - No node has an even number of edges!

What's Interesting About This

1707-1783

- Not the Königsberg bridges problem itself
- Rather, the way Euler solved it
- A new way to think about a very large class of problems
  - Abstract out unnecessary details
  - Focus on nodes (key elements) and edges (connections between key elements)
  - Model cost of traversing edges in graph
  - Optimize paths across edges

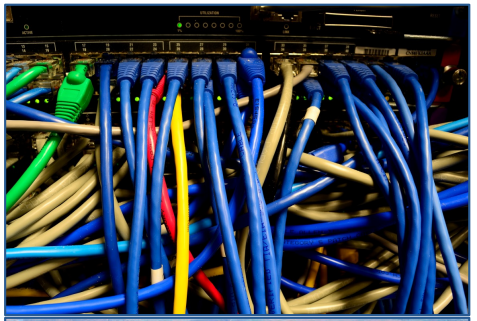
Graphs

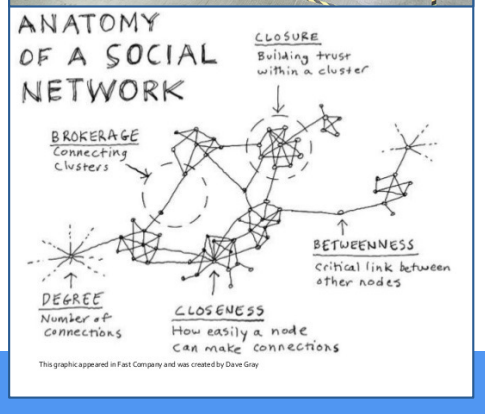


- Capture and reason about relationships among entities
  - Routes between Boston and San Jose
  - How the atoms in a molecule are related to one another
  - Ancestral relationships (family trees)
  - Business/social/political connections
  - Functions and expressions in python
  - …

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



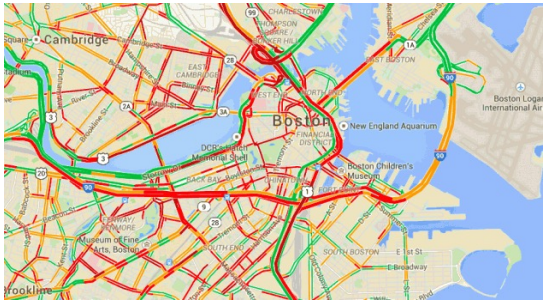


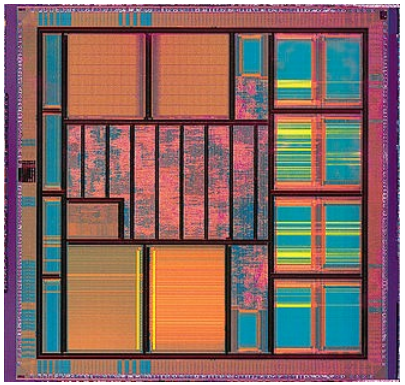
The first three examples all ask about finding an efficient path between nodes; the last example suggests that there can be other kinds of questions

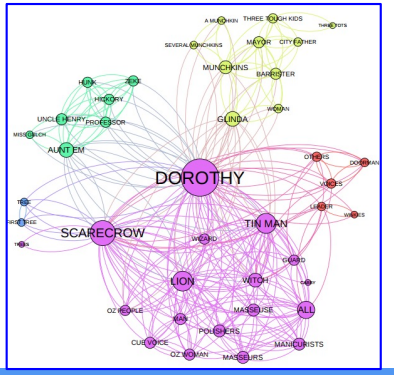
Why Graphs Are So Useful

- 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**)



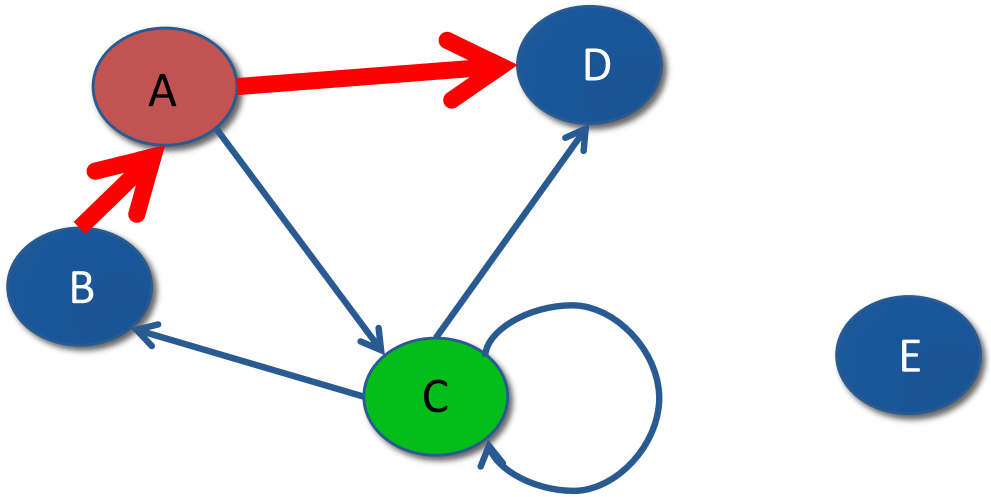




You'll see these problems in 6.120[6.042], 6.121[6.006], and other classes

A Classic Graph Optimization Problem

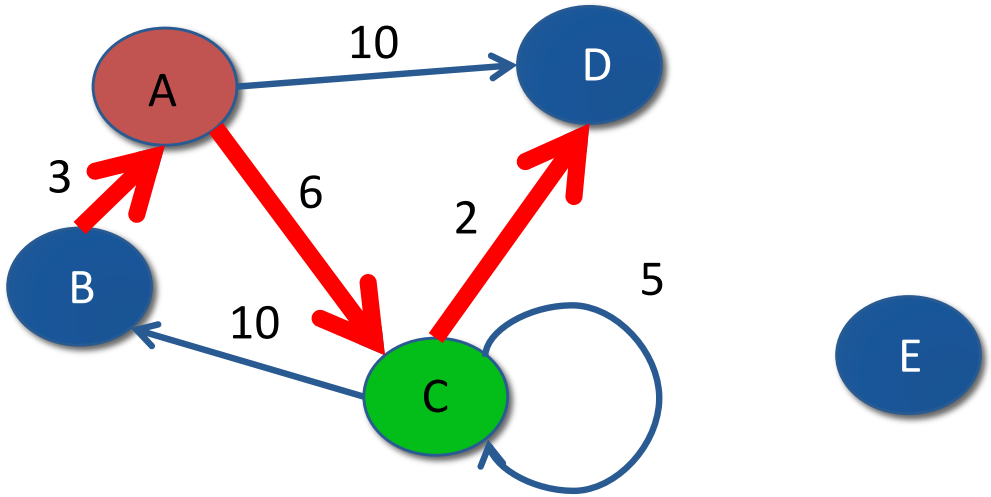
- Shortest (unweighted) path
  - Fewest number of edges from a source node to a destination node



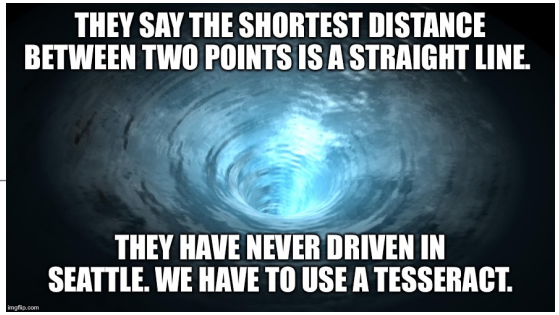
A Classic Graph Optimization Problem

Next lecture

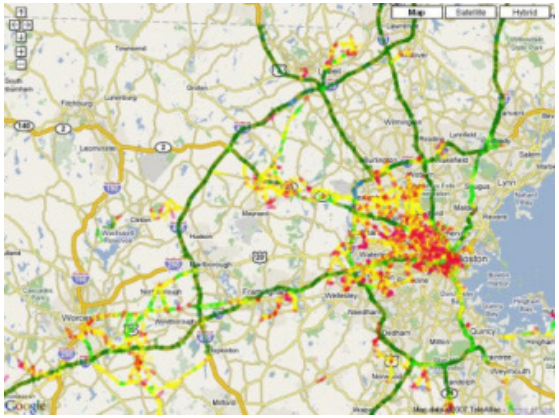
- Shortest (unweighted) path
  - Fewest number of edges from a source node to a destination node
- Shortest weighted path
  - Minimizes the sum of the weights of its edges

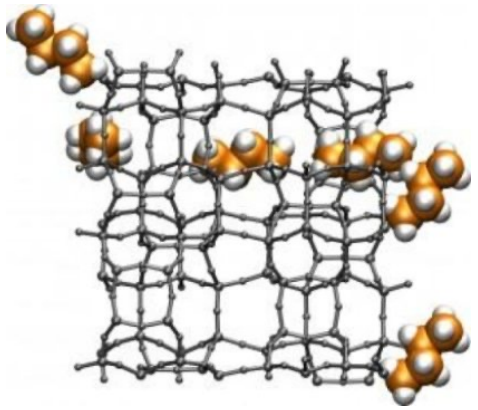


Some Shortest Path Problems



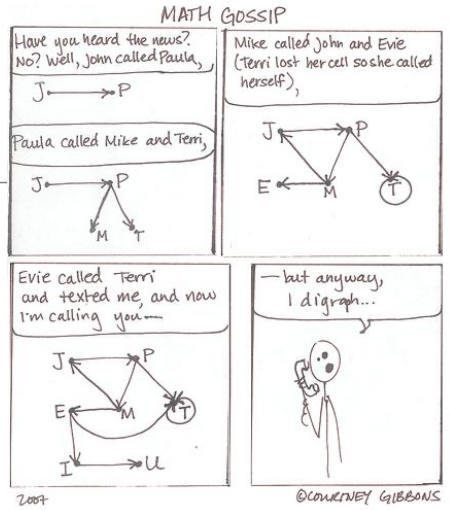
- Finding a route from one city to another
- Routing data on communication networks
- Warehouse logistics of storing and retrieving products
- Finding a path for a molecule through a chemical labyrinth



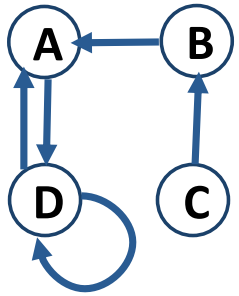


IMPLEMENTING GRAPHS

Representations of Digraphs



- **Digraph is a directed graph**
  - Edges pass in one direction only
  - Need to represent collection of edges



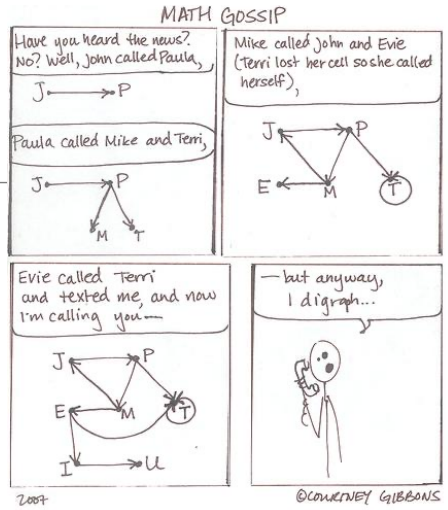
- **Adjacency matrix**
  - Rows: source nodes
  - Columns: destination nodes
  - Cell[s, d] = 1 if there is an edge from s to d
    = 0 otherwise
  - Note that in digraph, matrix is **not** symmetric
  - Uses **O(|nodes|**2) memory

d

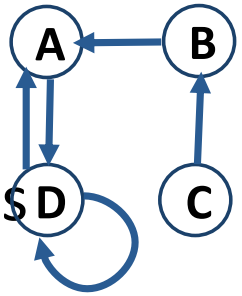
s	A	B	C	D
A				1
B	1			
C		1		
D	1			1

- **Assumes at most one arc between node pairs**
  - Easily generalized to multiple arcs with weights

Representations of Digraphs



- Digraph is a directed graph
  - Edges pass in one direction only
  - Need to represent collection of edges



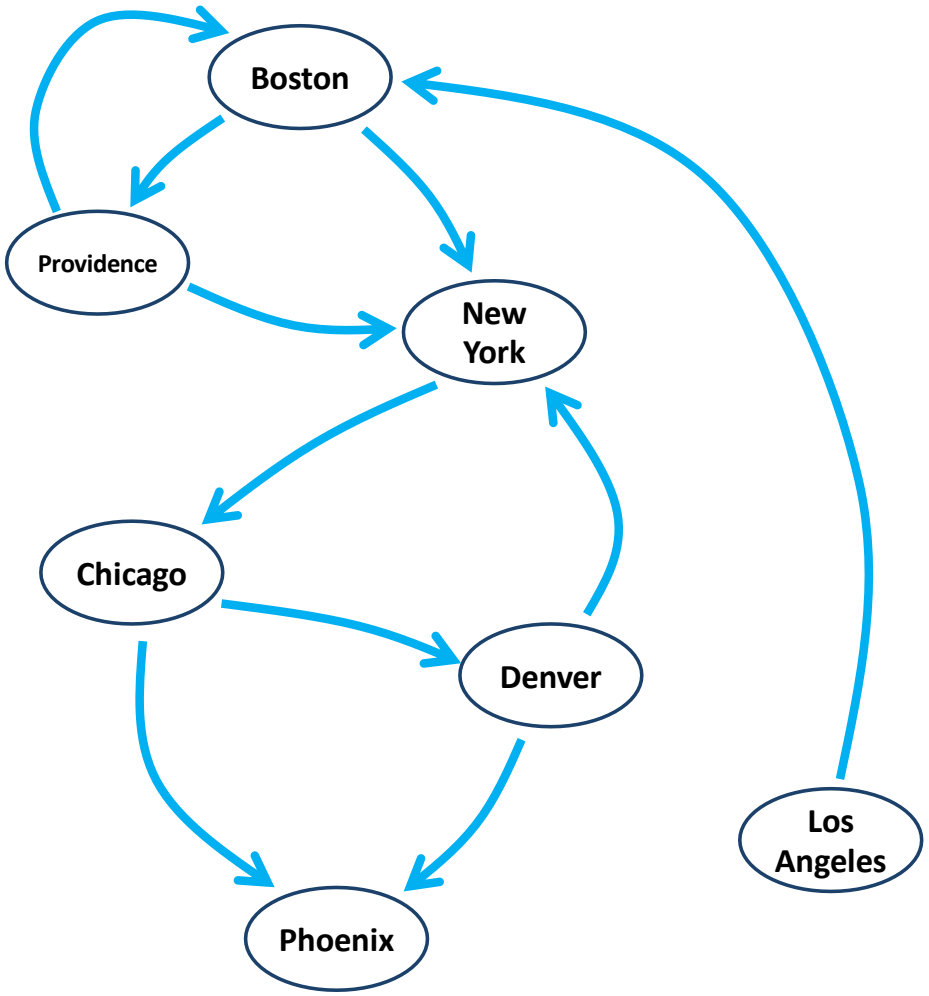
- **Adjacency matrix**
  - Rows: source nodes
  - Columns: destination nodes
  - Cell[s, d] = 1 if there is an edge from s to d
    = 0 otherwise
  - Note that in digraph, matrix is **not** symmetric
  - Uses **O(|nodes|**2) memory**
- **Adjacency list**
  - Associate with each node a list of destination nodes that can be reached by one edge
  - Uses **O(|edges|) memory**, therefore good for **sparse** graphs

A: [D]
B: [A]
C: [B]
D: [A, D]

An Example Digraph

Insight 1: Store nodes as dictionary keys, and edges using adjacency lists for each node

Insight 2: Store edge weights by turning each adjacency list into a dictionary



Adjacency Lists

Boston:
Providence:
New York:
Chicago:
Denver:
Phoenix:
Los Angeles:

Node

Edges from that node

An Example Digraph

Insight 1: Store nodes as dictionary keys, and edges using adjacency lists for each node

Insight 2: Store edge weights by turning each adjacency list into a dictionary

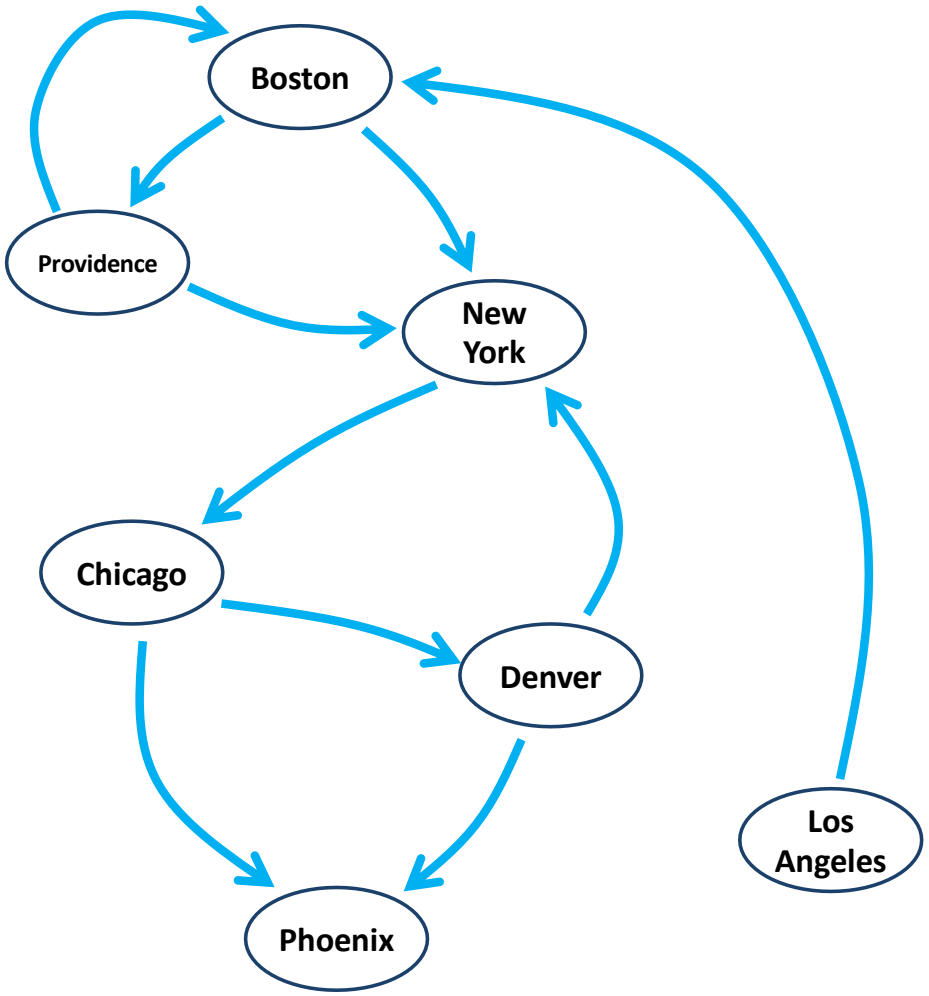
All nodes in graph

Adjacency Lists

Boston:	Providence(1), New York(1)
Providence:	Boston(1), New York(1)
New York:	Chicago(1)
Chicago:	Denver(1), Phoenix(1)
Denver:	Phoenix(1), New York(1)
Phoenix:	
Los Angeles:	Boston(1)

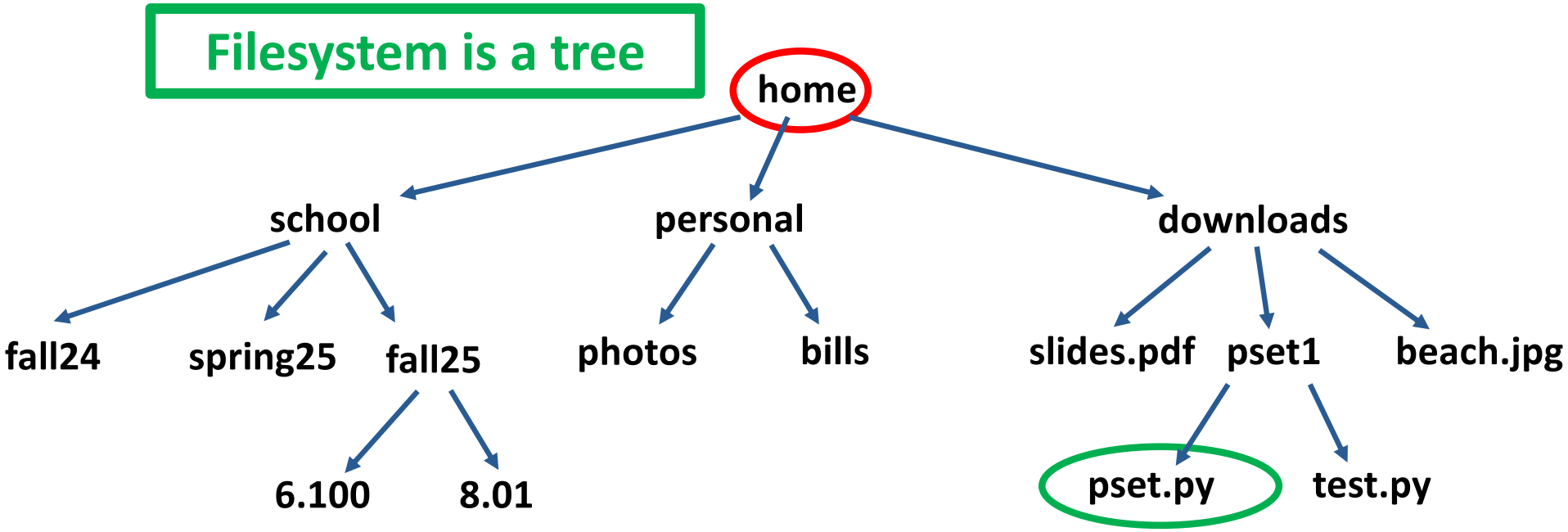
Node

Edges from that node



Unweighted edges all have weight 1

Let's look at some code



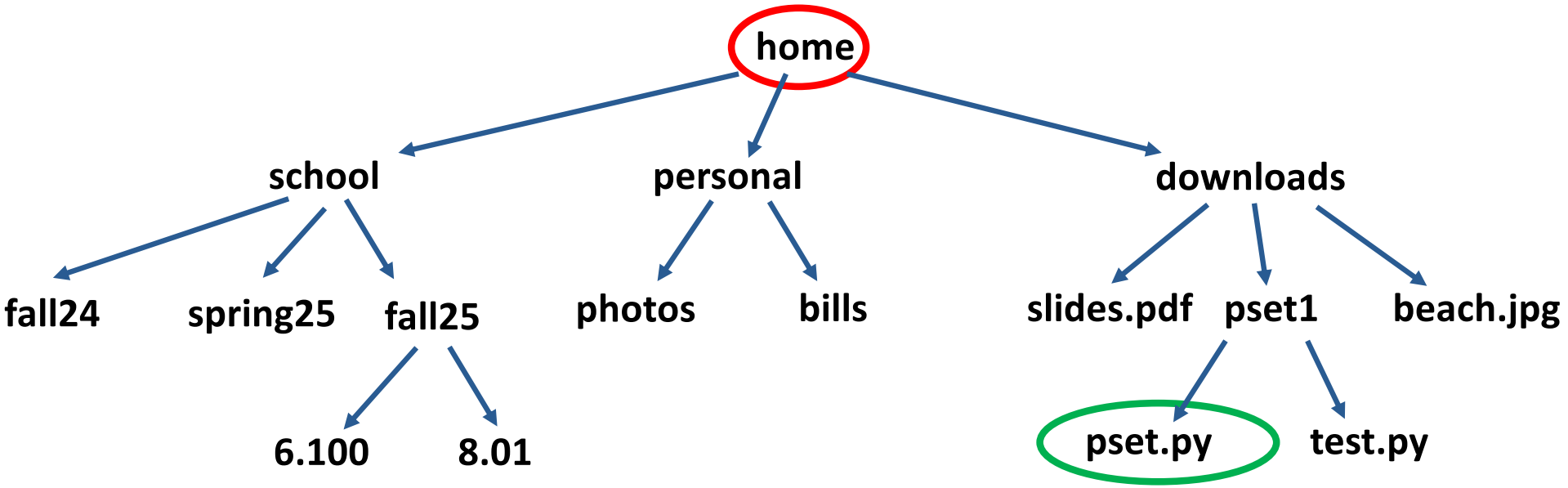

```
filesystem = {
    "home": ["school", "personal", "downloads"],
    "school": ["fall24", "spring25", "fall25"],
    "fall25": ["6.100", "8.01"],
    "personal": ["photos", "bills"],
    "downloads": ["slides.pdf", "ps1", "beach.jpg"],
    "ps1": ["pset.py", "test.py"],
}
```

FINDING PATHS IN TREES

Example tree

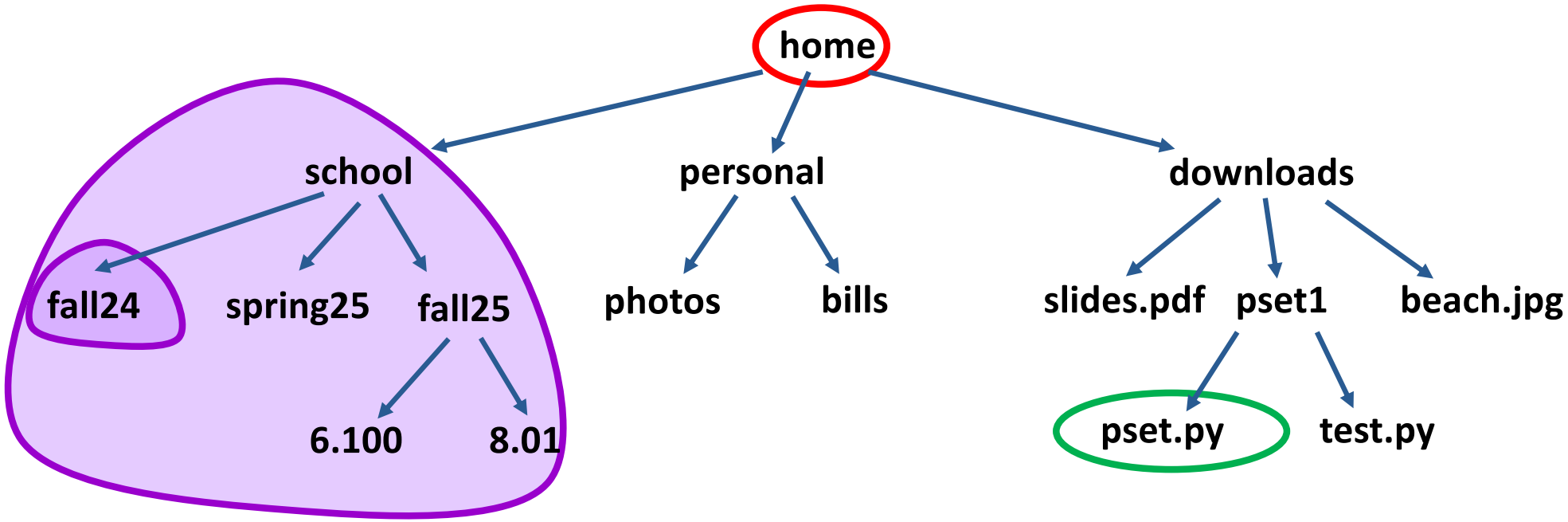
Question:
Where did I put my pset file?

Filesystem is a tree



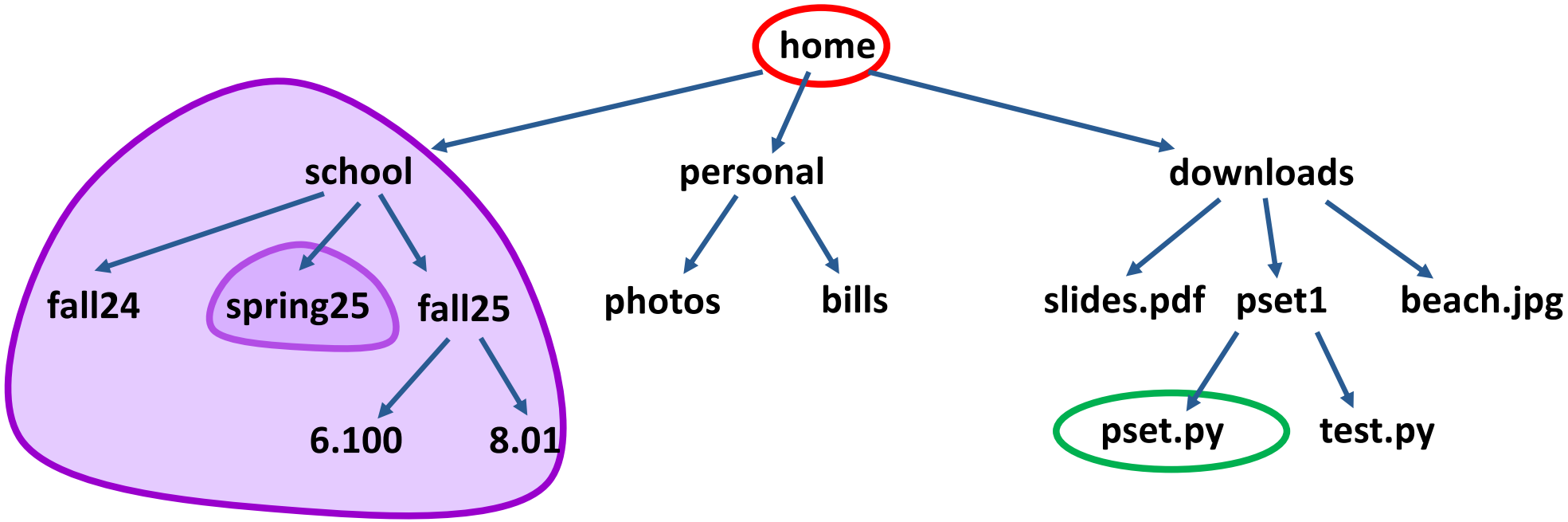
Approach 1: Leverage recursive structure

- Keep exploring children before considering siblings
- After branching on child, recursively find a path to target, using **child as new root**



Approach 1: Leverage recursive structure

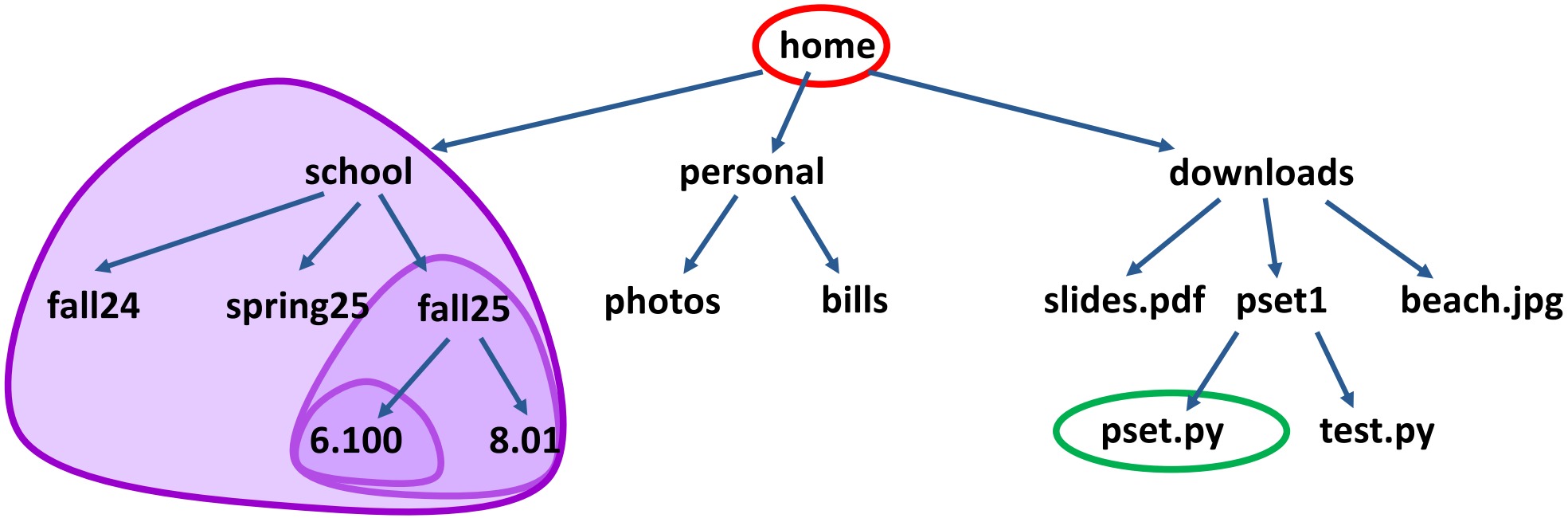
- Keep exploring children before considering siblings
- After branching on child, recursively find a path to target, using **child as new root**



Approach 1: Leverage recursive structure

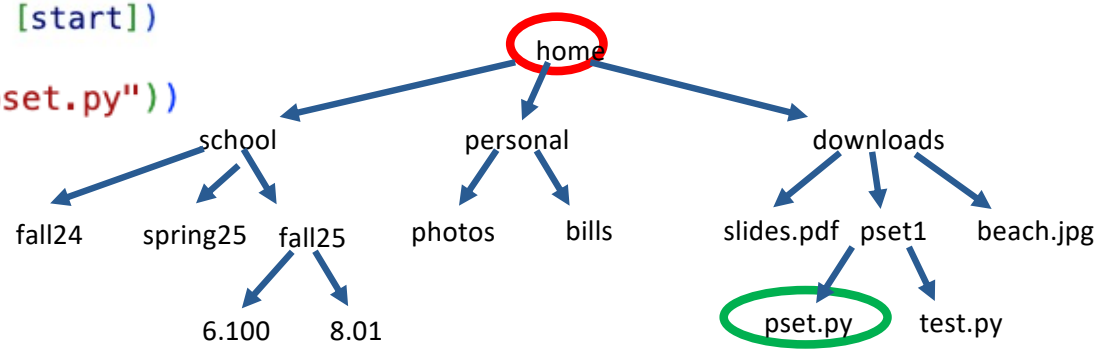
- Keep exploring children before considering siblings
- After branching on child, recursively find a path to target, using **child as new root**

Depth-First Search (DFS)



Example Implementation

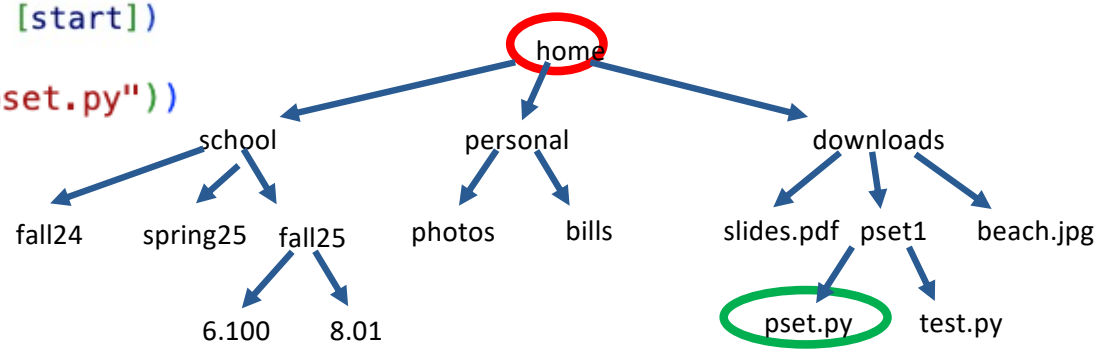
```
def dfs_tree_helper(graph, goal, path):
    print("Current DFS path:", path_to_string(path))

    # stop if reached goal
    current_node = path[-1]
    if current_node == goal:
        return path

    # check if goal in subtrees
    for next_node in get_neighbors(graph, current_node):
        result = dfs_tree_helper(graph, goal, path + [next_node])
        if result is not None:
            return result

    # no nodes in subtree are goal, backtrack to parent/caller
    return None

def dfs_tree(graph, start, goal):
    return dfs_tree_helper(graph, goal, [start])

print(dfs_tree(filesystem, "home", "pset.py"))
```



Example Implementation

```
def dfs_tree_helper(graph, goal, path):
    print("Current DFS path:", path_to_string(path))

    # stop if reached goal
    current_node = path[-1]
    if current_node == goal:
        return path

    # check if goal in subtrees
    for next_node in get_neighbors(graph, current_node):
        result = dfs_tree_helper(graph, goal, path + [next_node])
        if result is not None:
            return result

    # no nodes in subtree are goal, backtrack to parent/caller
    return None

def dfs_tree(graph, start, goal):
    return dfs_tree_helper(graph, goal, [start])

print(dfs_tree(filesystem, "home", "pset.py"))
```

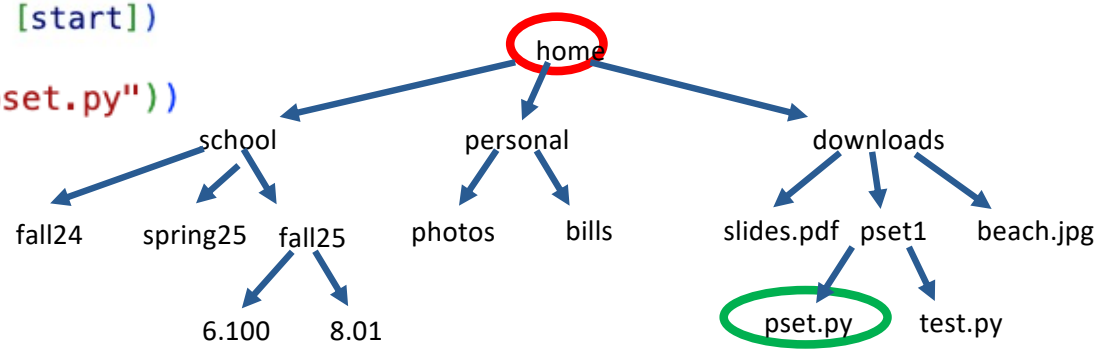
```
dfs_tree_helper(graph, “pset.py”, [“home”])
                        ↓
dfs_tree_helper(graph, “pset.py”, [“home”,”school”])
                        ↓
dfs_tree_helper(graph, “pset.py”,
[“home”,”school”,”fall24”])
```



Example Implementation

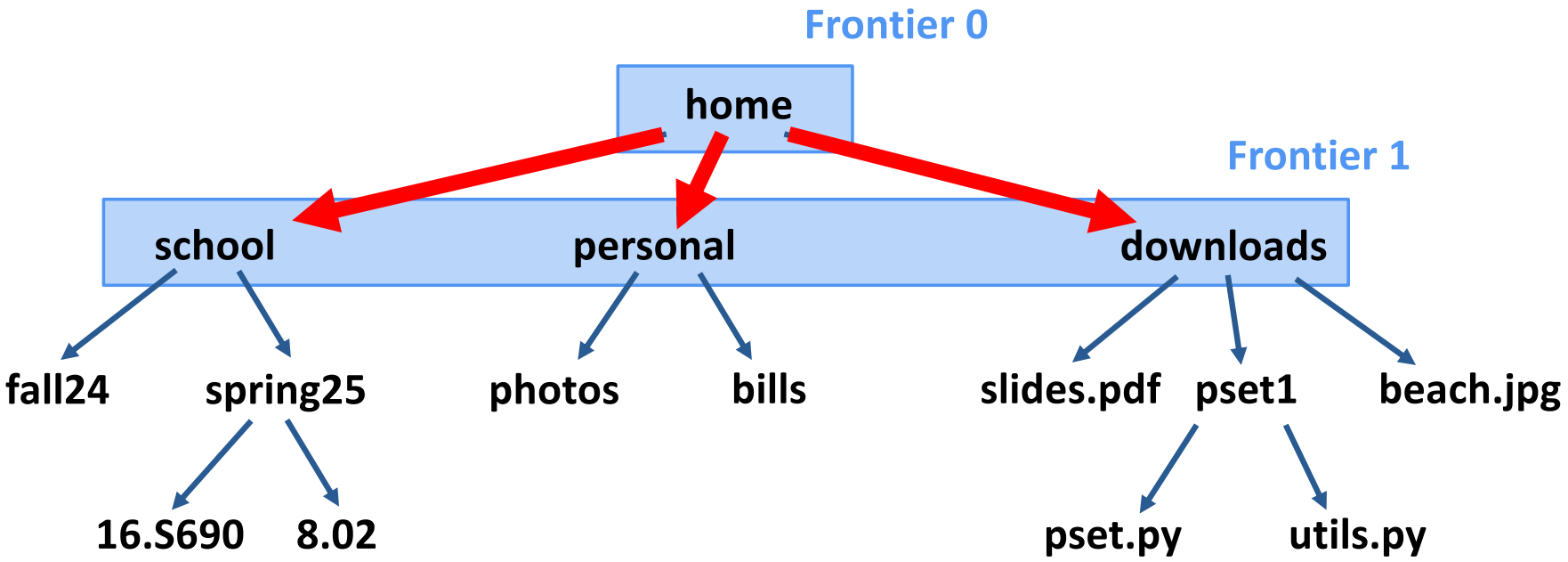
```
def dfs_tree_helper(graph, goal, path):
    print("Current DFS path:", path_to_string(path))

    # stop if reached goal
    current_node = path[-1]
    if current_node == goal:
        return path

    # check if goal in subtrees
    for next_node in get_neighbors(graph, current_node):
        result = dfs_tree_helper(graph, goal, path + [next_node])
        if result is not None:
            return result

    # no nodes in subtree are goal, backtrack to parent/caller
    return None

def dfs_tree(graph, start, goal):
    return dfs_tree_helper(graph, goal, [start])

print(dfs_tree(filesystem, "home", "pset.py"))
```

```
dfs_tree_helper(graph, “pset.py”, [“home”])
                        ↓
dfs_tree_helper(graph, “pset.py”, [“home”,”school”])
                        ↓
dfs_tree_helper(graph, “pset.py”,
[“home”,”school”,”spring25”])
```



Approach 2: Scan across branches

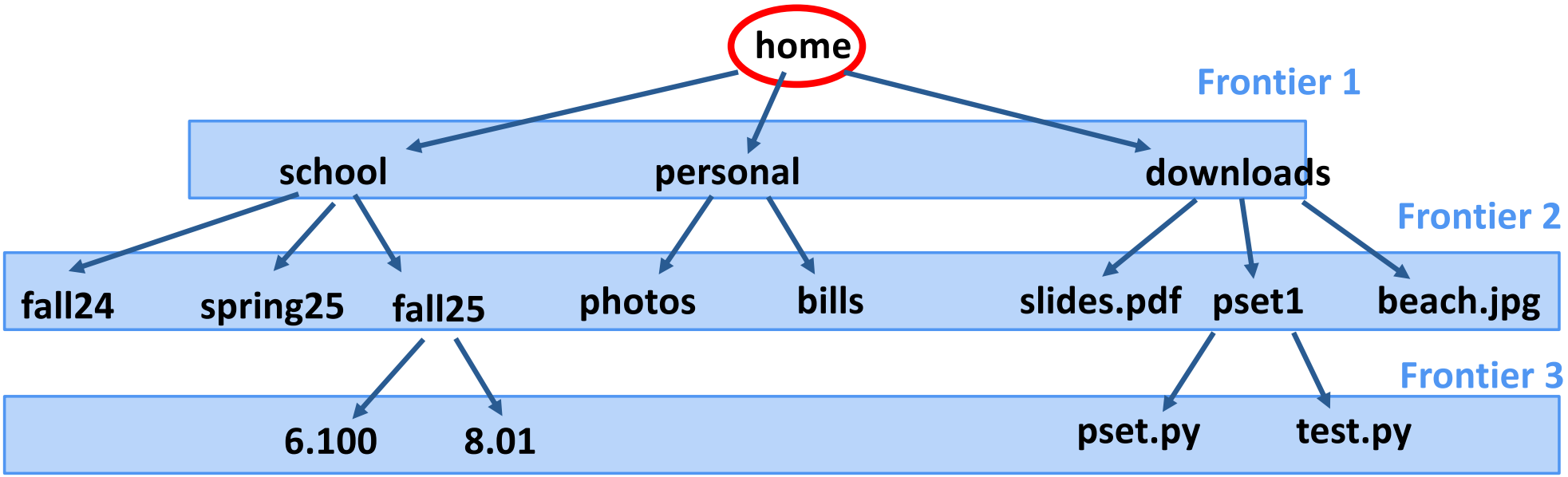
- Follow **successive layers** of depth away from the root
  - Layers are **"frontiers"**
- Each frontier is **one step away** from the previous frontier



Approach 2: Scan across branches

- Follow **successive layers** of depth away from the root
  - Layers are **"frontiers"**
- Each frontier is **one step away** from the previous frontier

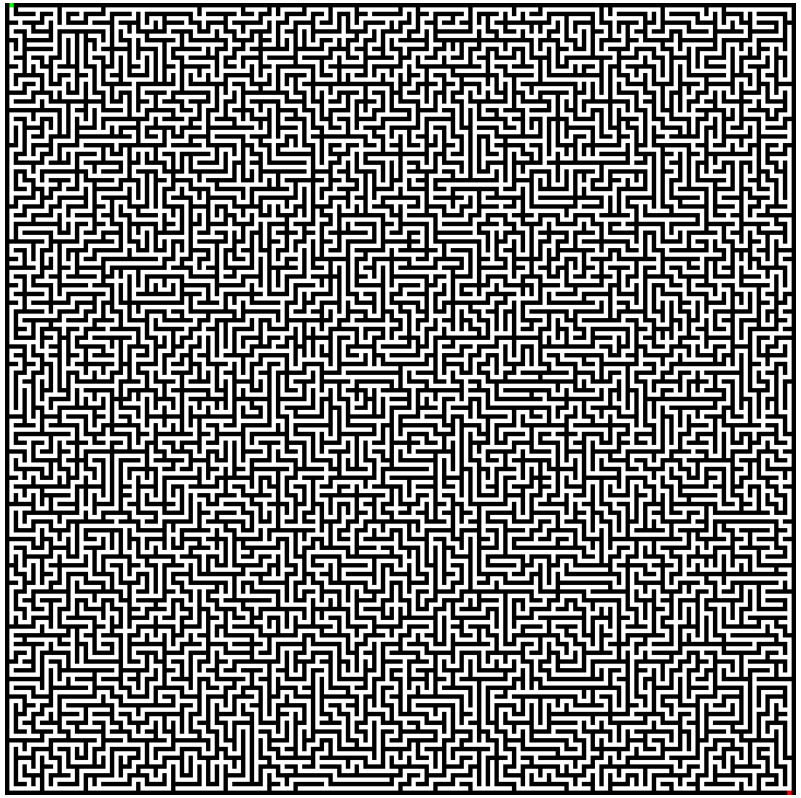
Breadth-First Search (BFS)

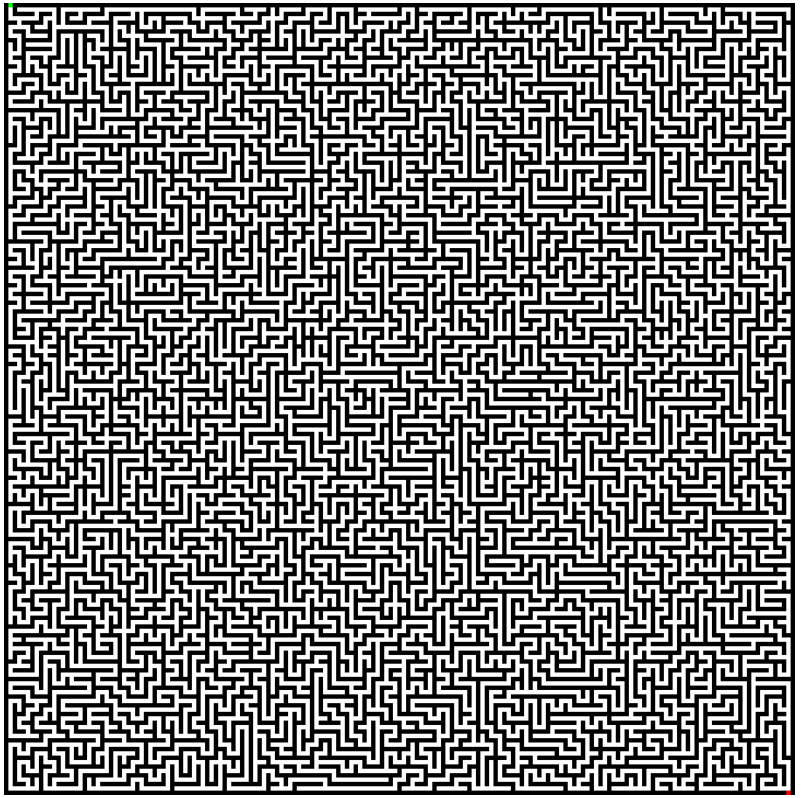


Factors affecting performance

- Shortest distance between start and goal
  - BFS guaranteed to find it
  - But time and memory can grow quickly with that distance, because…
- Branching factor
  - Exponential growth in the number of nodes/states at each frontier
  - Inherent in tree structure, affects DFS, too
- Order of exploring neighbors
  - Affects DFS more than BFS
  - Could get lucky and go down all the right branches
  - But going down wrong branch, especially early on, can be wasteful

Let’s take a short break

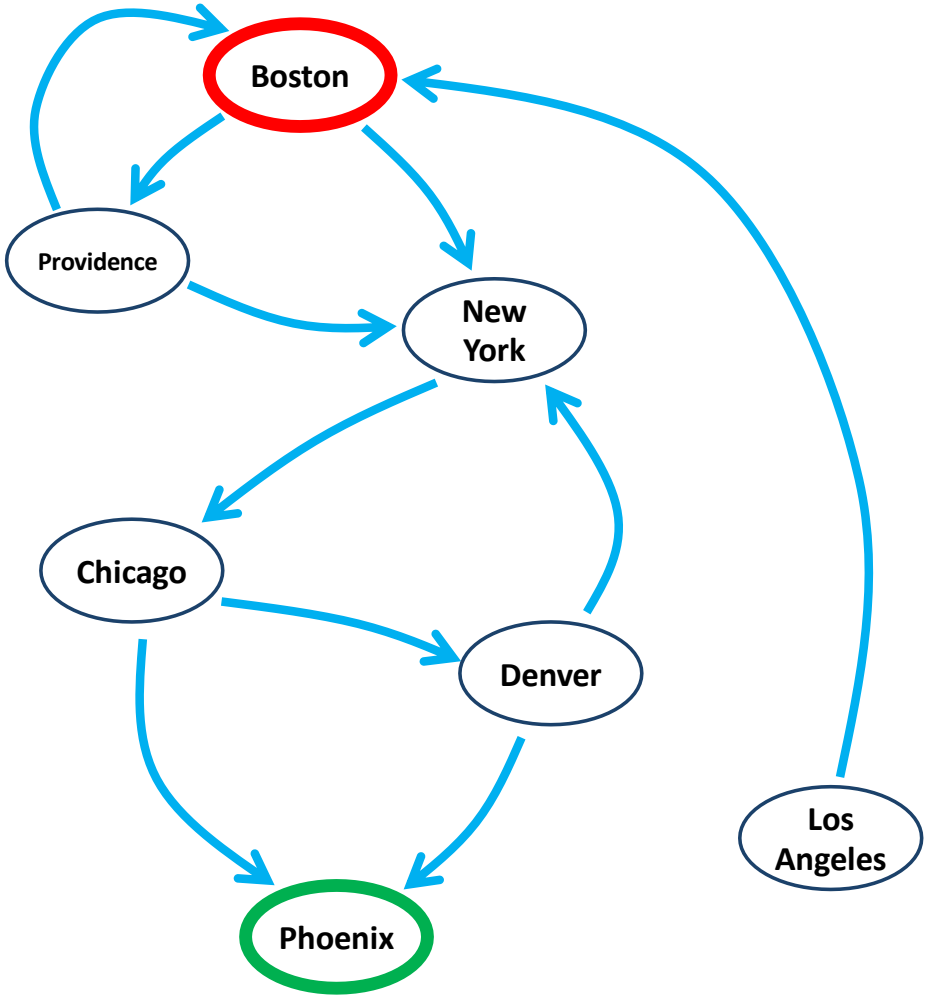




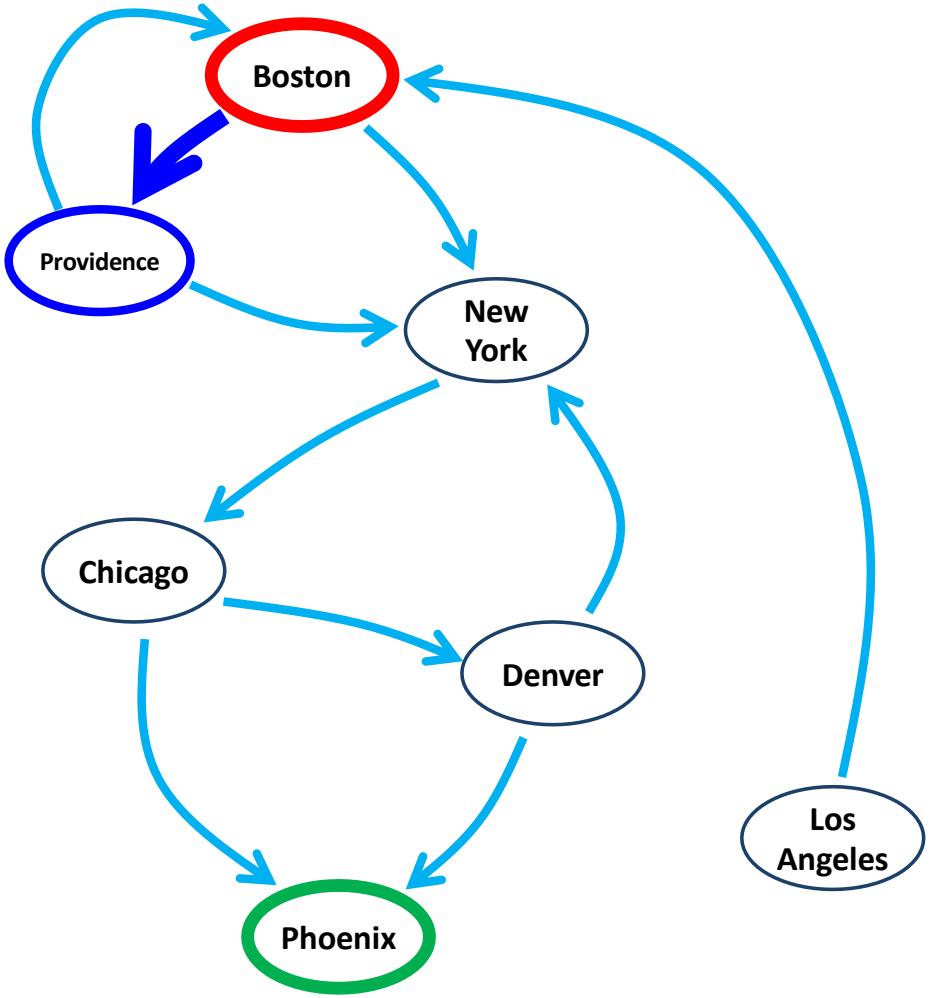
https://imgur.com/gallery/n5Ouj

GENERALIZING SEARCH TO GRAPHS

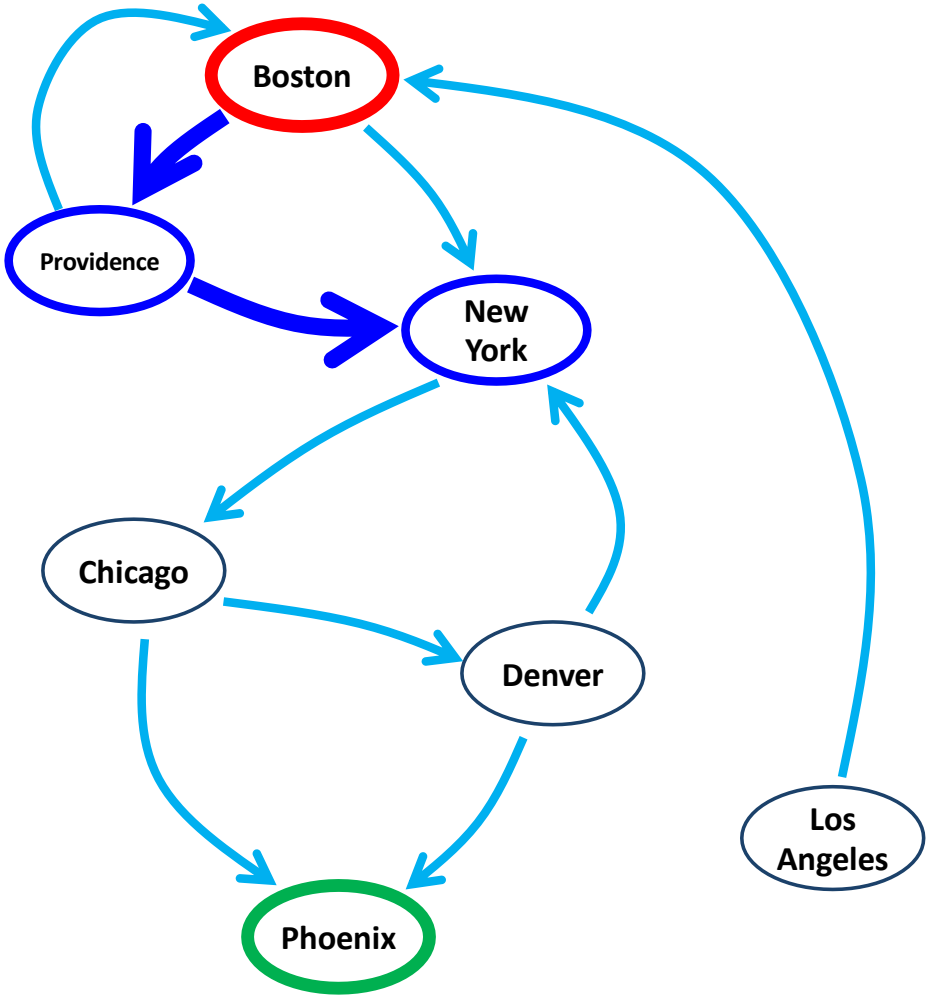
DFS: Avoiding infinite cycles



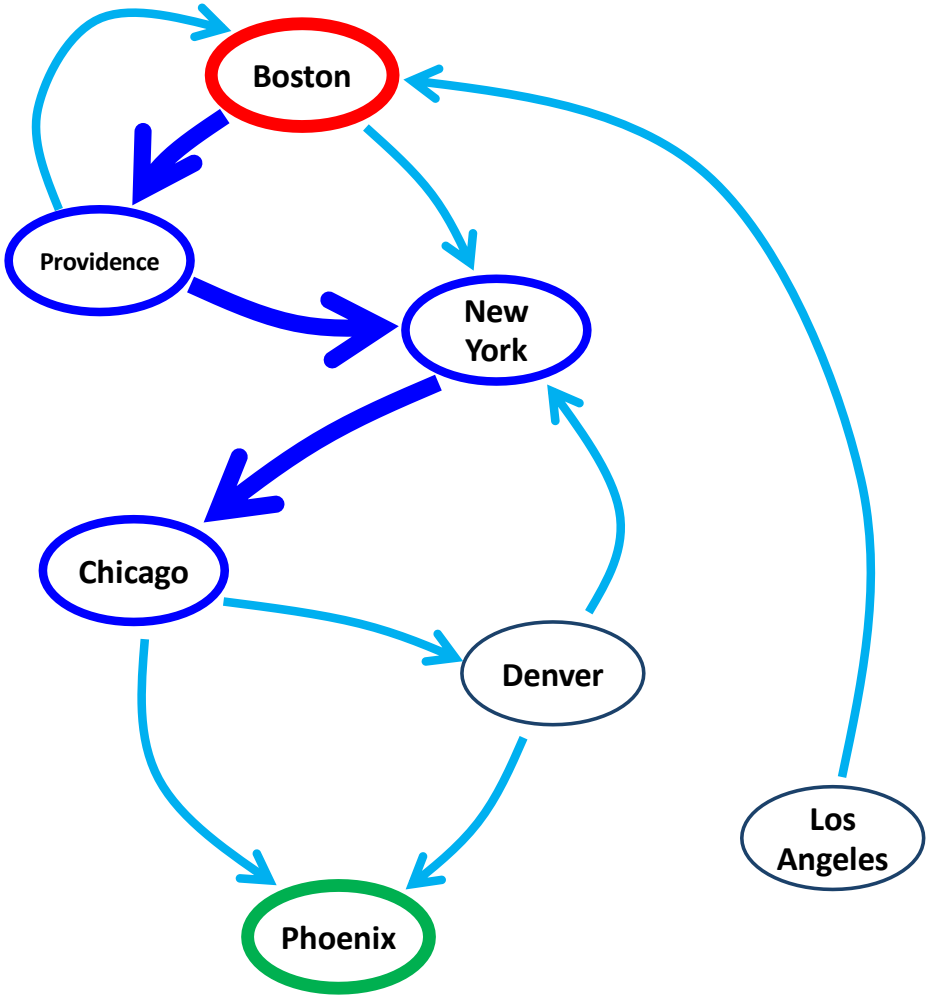
DFS: Avoiding infinite cycles



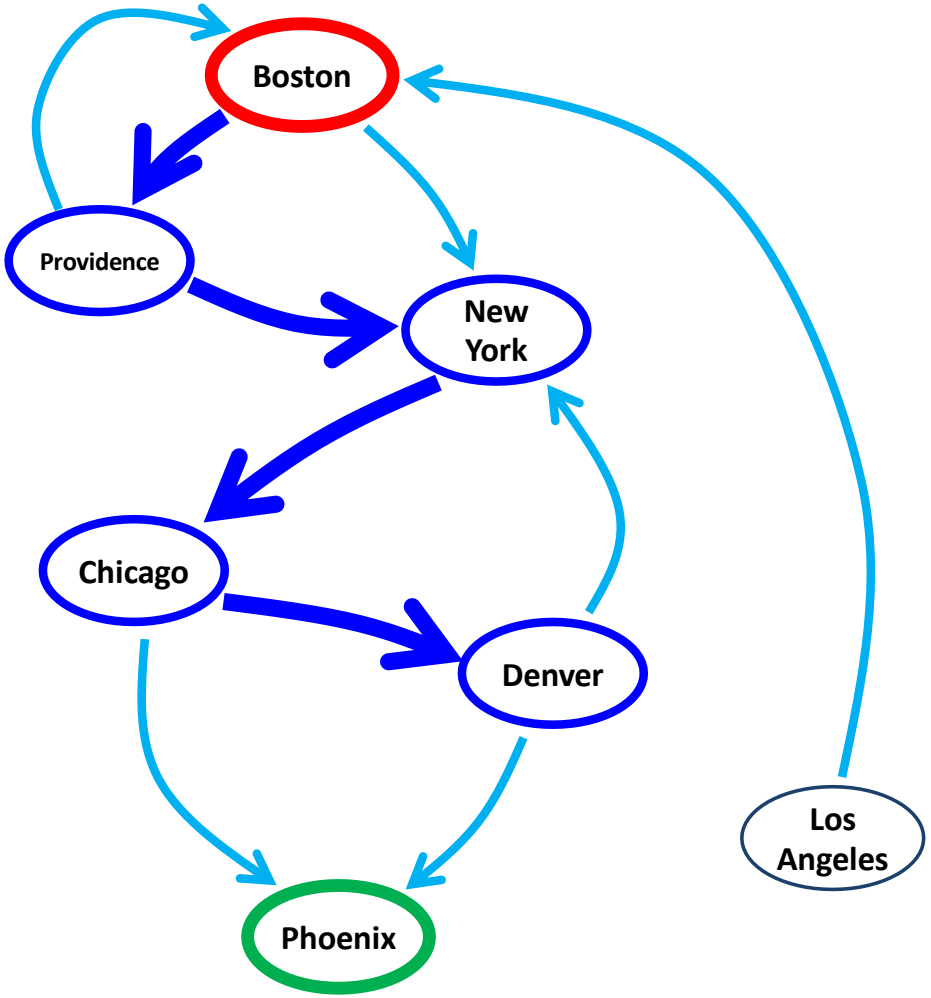
DFS: Avoiding infinite cycles



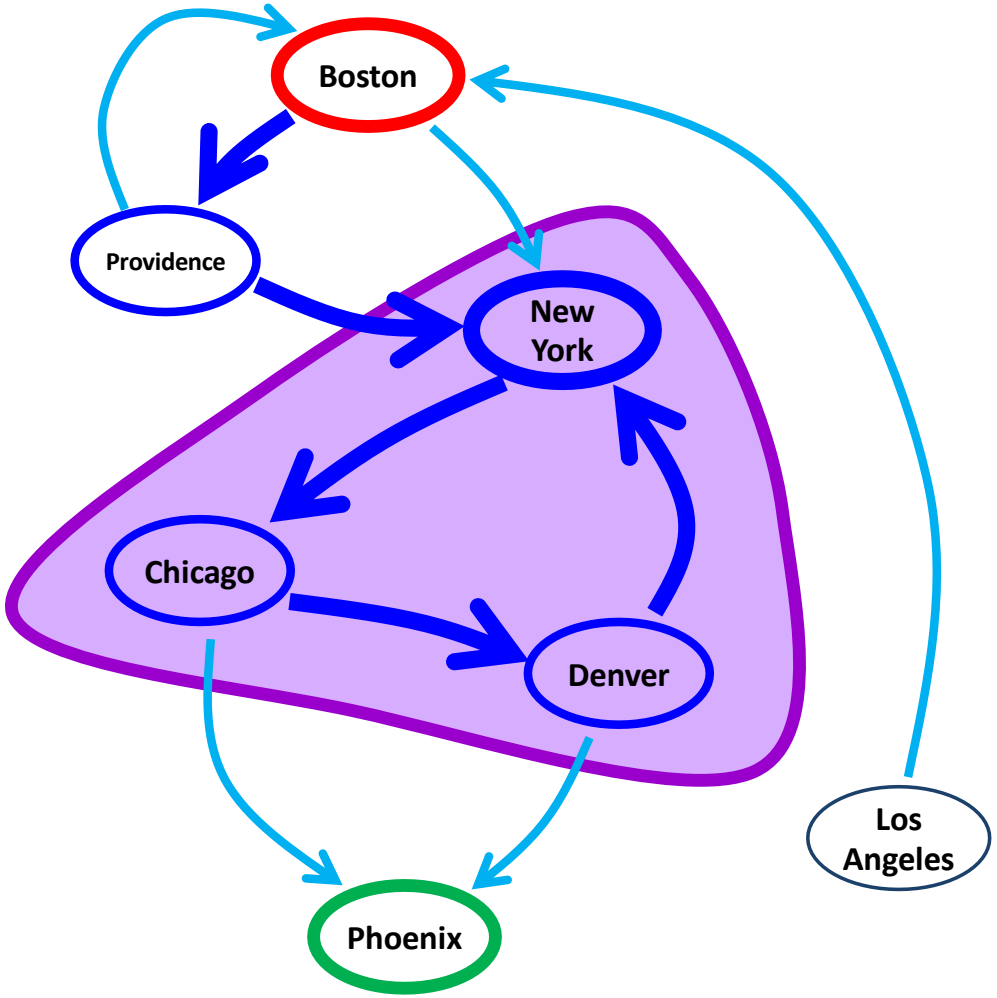
DFS: Avoiding infinite cycles



DFS: Avoiding infinite cycles

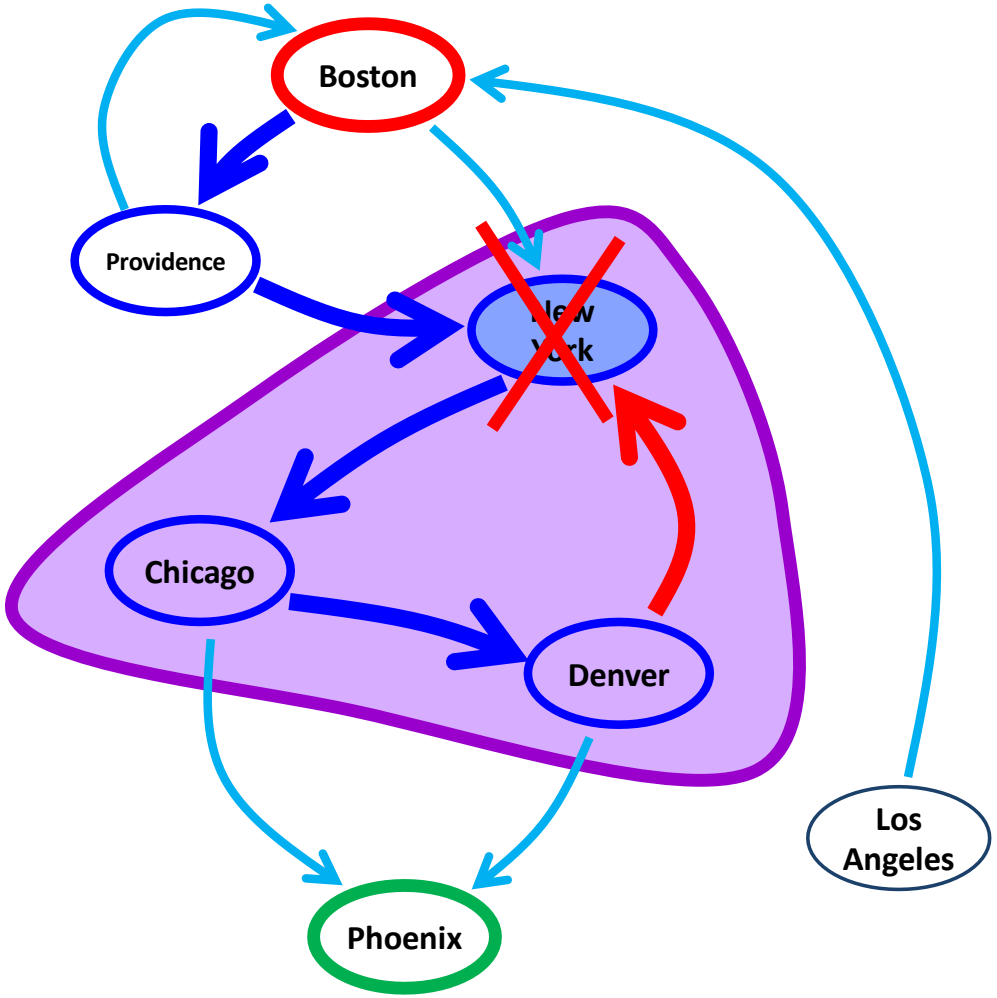


DFS: Avoiding infinite cycles



DFS: Avoiding infinite cycles

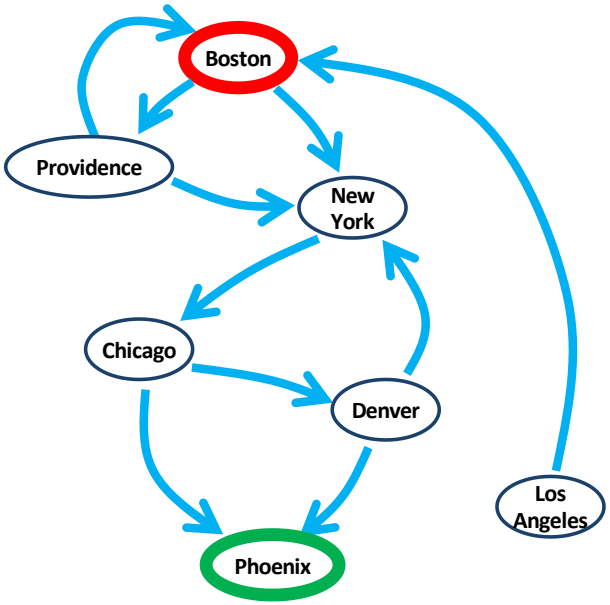
- Encounter cycles when expanding to **nodes already on the path**
- Prevent **DFS "subtrees"** from looping back on themselves



```
def dfs_graph_helper(graph, goal, path):
    print("Current DFS path:", path_to_string(path))

    current_node = path[-1]
    if current_node == goal:
        return path

    possible_paths = []
    for next_node in get_neighbors(graph, current_node):

        # avoid self-loops
        if next_node in path:
            print(f"  AVOID self-loop from {current_node} to {next_node}")
            continue

        result = dfs_graph_helper(graph, goal, path + [next_node])
        if result is not None:
            return result

    return None


def dfs_graph(graph, start, goal):
    return dfs_graph_helper(graph, goal, [start])
```

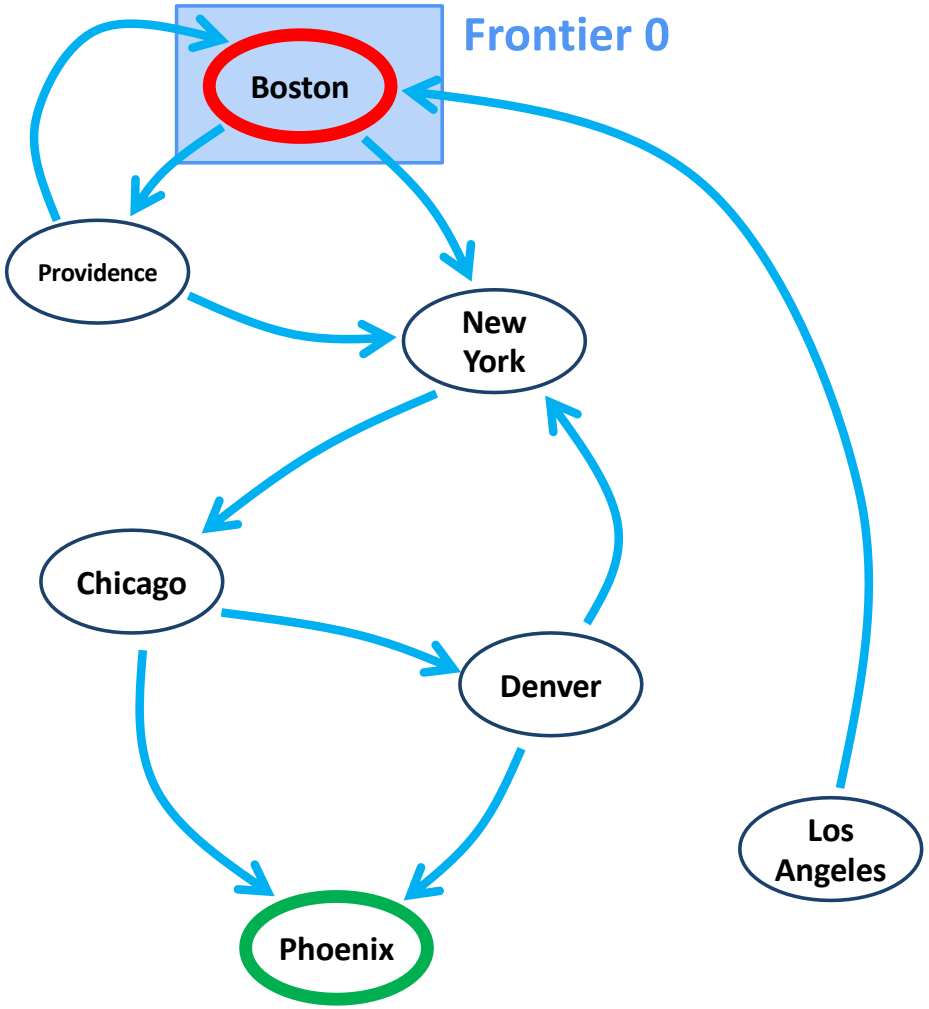
TRACE:

```
path: Boston
path: Boston -> Providence
  AVOID self-loop from Providence to Boston
path: Boston -> Providence -> New York
path: Boston -> Providence -> New York -> Chicago
path: Boston -> Providence -> New York -> Chicago -> Denver
  AVOID self-loop from Denver to New York
path: Boston -> Providence -> New York -> Chicago -> Denver -> Phoenix
['Boston', 'Providence', 'New York', 'Chicago', 'Denver', 'Phoenix']
```



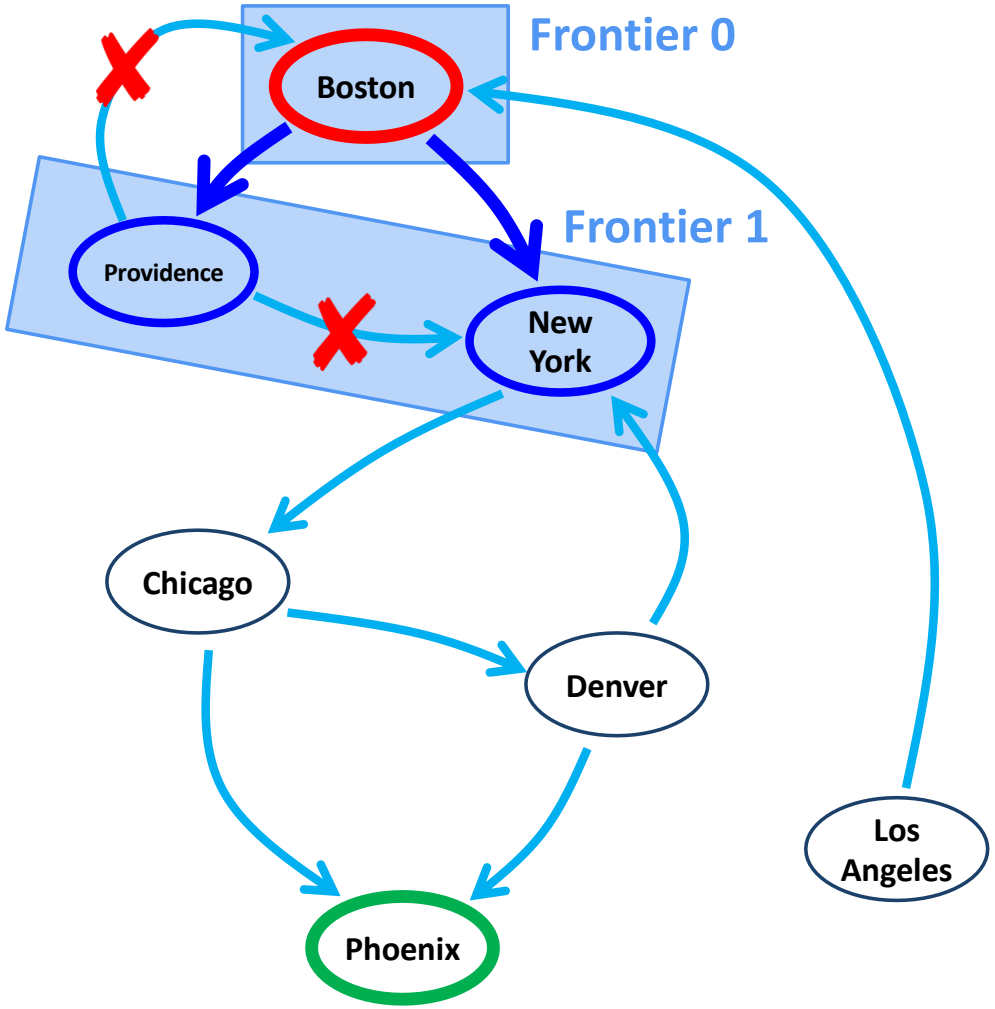
BFS: Avoiding previously seen nodes

- Already **storing paths** so far when expanding their end nodes
- Can do even better: Reject any new child already in a **previous or current frontier**



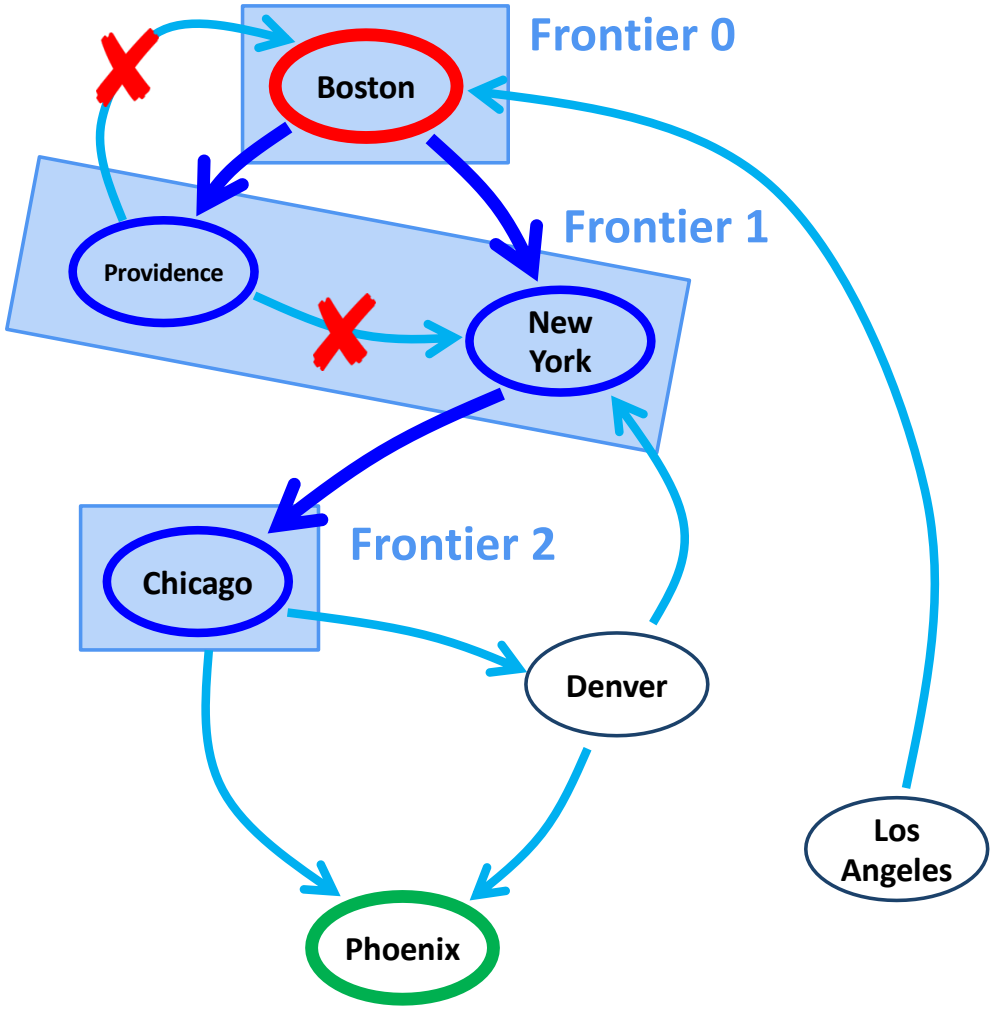
BFS: Avoiding previously seen nodes

- Already **storing paths** so far when expanding their end nodes
- Can do even better: Reject any new child already in a **previous or current frontier**



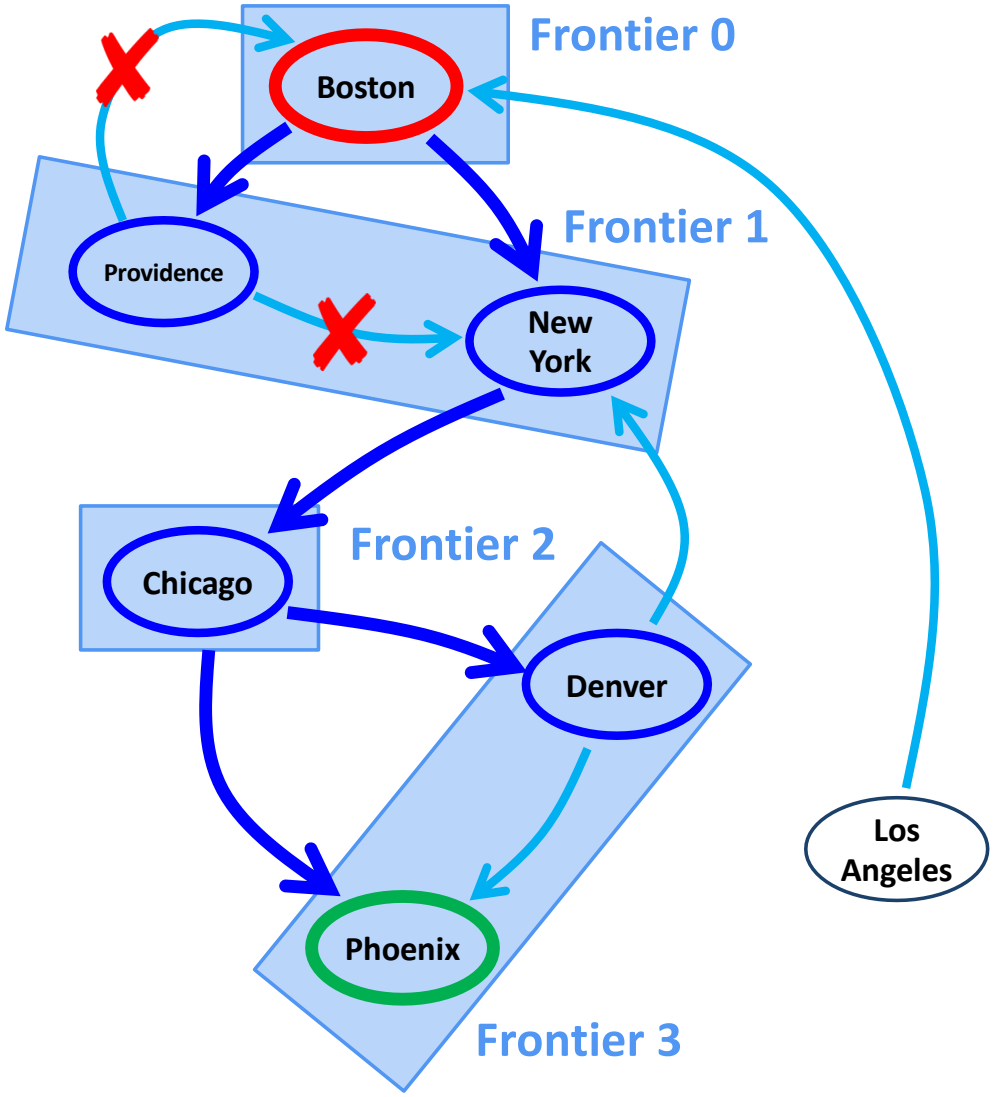
BFS: Avoiding previously seen nodes

- Already **storing paths** so far when expanding their end nodes
- Can do even better: Reject any new child already in a **previous or current frontier**



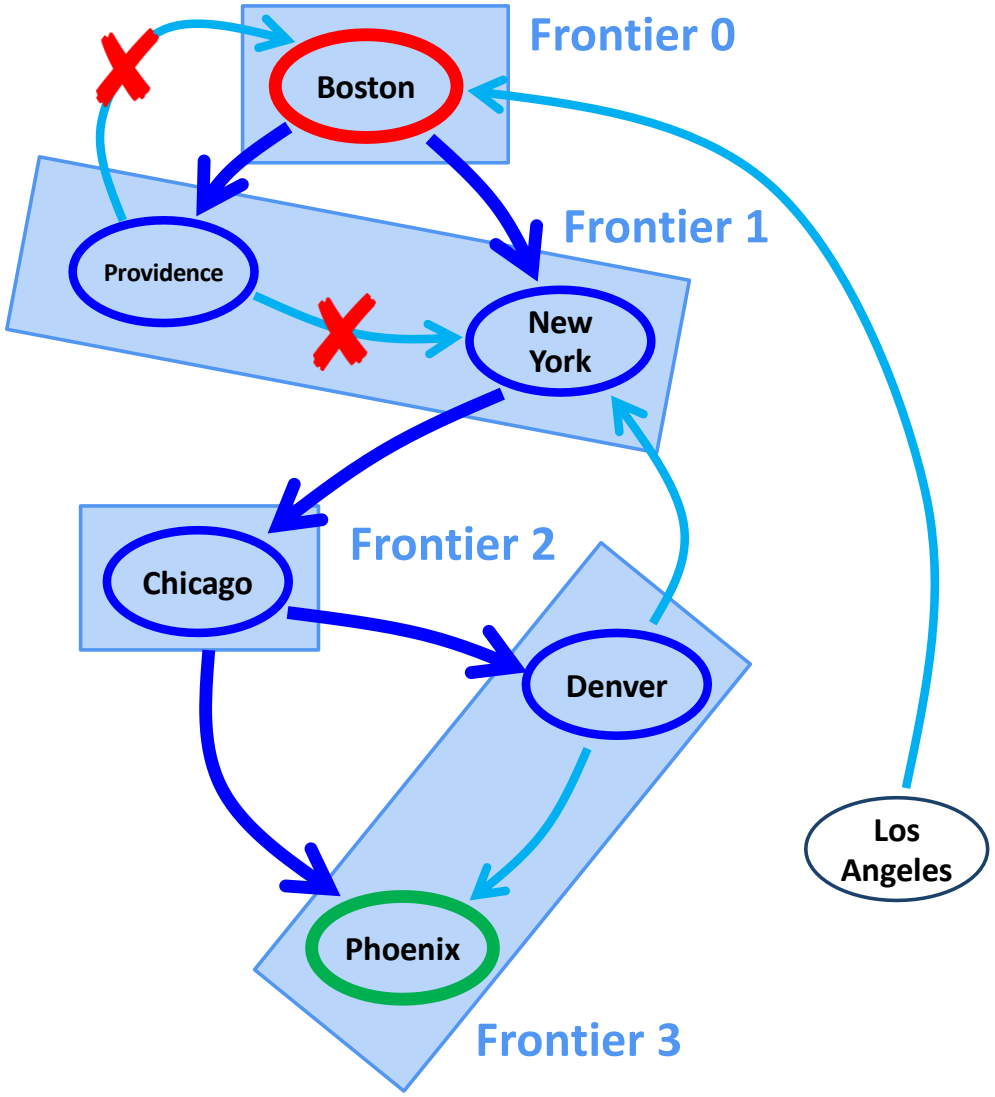
BFS: Avoiding previously seen nodes

- Already **storing paths** so far when expanding their end nodes
- Can do even better: Reject any new child already in a **previous or current frontier**



BFS: Shortest-path property

- Already **storing paths** so far when expanding their end nodes
- Can do even better: Reject any new child already in a **previous or current frontier**
- Each next frontier n contains exactly those nodes reachable in n **steps from the root**
  - **Nodes are discovered at their shortest distances**



TAKEAWAYS AND CONSIDERATIONS

Generalizing goal check to goal test function

- Goal may be defined in terms of properties rather than a single state
- Instead of checking **`state == goal`**, abstract away into **`goal_test(state)`**
- Beware of function's complexity
  - Gets checked on every state, has a multiplicative effect

Limitations of DFS and BFS

- No edge weights
  - Some actions may be more expensive than others
  - Same number of actions in plan does not guarantee same cost to execute

- Branching
  - Even small branching factors lead to explosion in exploring state space
    - Visited set helps, but only if action outcomes overlap
  - Branching order can lead to vastly different solutions and performance