Shortest paths, Breadth-first search

(download slides and .py files to follow along)

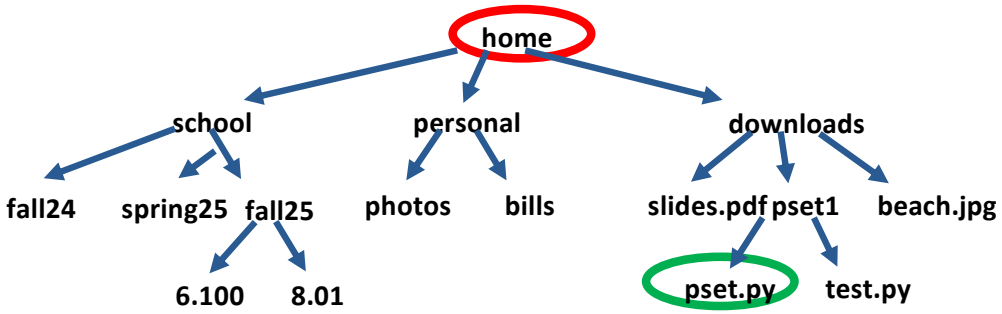
Tim Kraska

MIT Department Of Electrical Engineering and Computer Science

Topics

- Last week
  - Graphs and how to implement them
  - Depth-first search
- Today
  - Breadth-first search
  - Shortest path

Recap

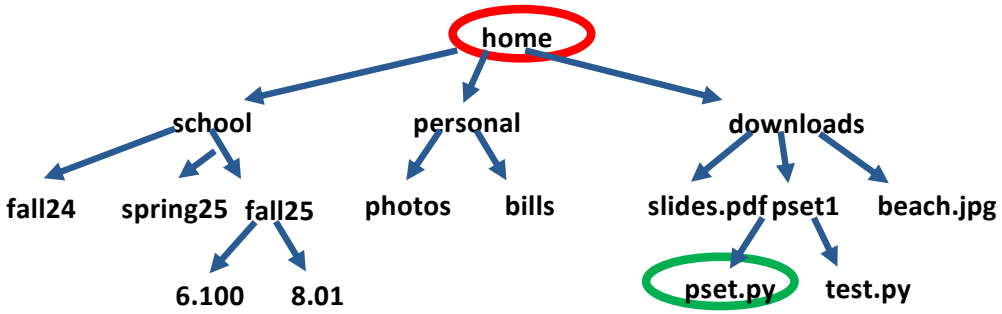



```
tiny_tree = {
        "home": ["school", "personal", "downloads"],
        "school": ["fall24", "spring25", "fall25"],
        "fall25": ["6.100", "8.01"],
        "personal": ["photos", "bills"],
        "downloads": ["slides.pdf", "ps1", "beach.jpg"],
        "ps1": ["pset.py", "test.py"],
   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

def exists_in(current_node, goal):
    if current_node == goal:
        return True
    for next_node in get_neighbors(tiny_tree, current_node):
        result = exists_in(next_node,goal)
        if result:
            return result
    return False

print(exists_in("home", "pset.py"))
```

Function calls

```
exists_in("home", "pset.py")
        ↓
exists_in(”school", "pset.py")
        ↓
exists_in(”fall24", "pset.py")
```

Recap

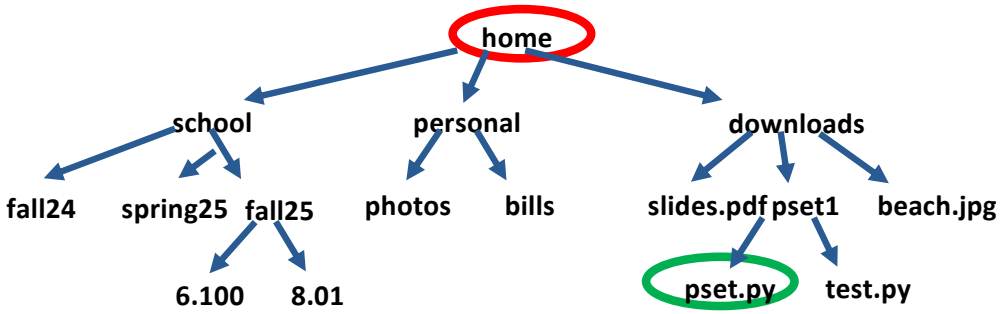



```
tiny_tree = {
        "home": ["school", "personal", "downloads"],
        "school": ["fall24", "spring25", "fall25"],
        "fall25": ["6.100", "8.01"],
        "personal": ["photos", "bills"],
        "downloads": ["slides.pdf", "ps1", "beach.jpg"],
        "ps1": ["pset.py", "test.py"],
   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

def exists_in(current_node, goal):
    if current_node == goal:
        return True
    for next_node in get_neighbors(tiny_tree, current_node):
        result = exists_in(next_node,goal)
        if result:
            return result
    return False

print(exists_in("home", "pset.py"))
```

Function calls

```
exists_in("home", "pset.py")
        ↓
exists_in(”school", "pset.py")
        ↓
exists_in(”spring25", "pset.py")
```

Recap

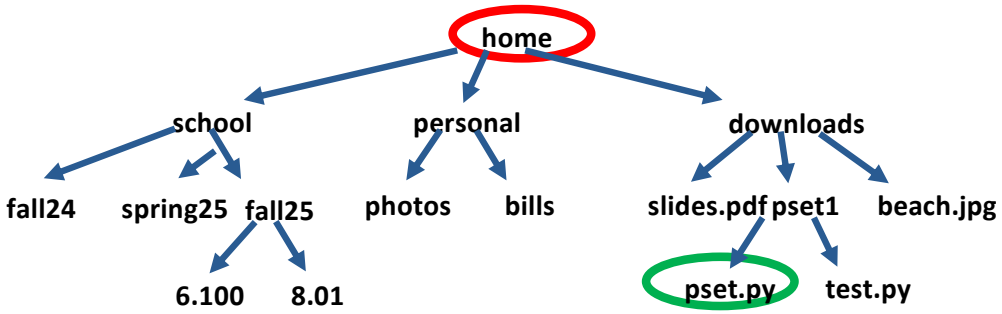



```
tiny_tree = {
        "home": ["school", "personal", "downloads"],
        "school": ["fall24", "spring25", "fall25"],
        "fall25": ["6.100", "8.01"],
        "personal": ["photos", "bills"],
        "downloads": ["slides.pdf", "ps1", "beach.jpg"],
        "ps1": ["pset.py", "test.py"],
   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

def exists_in(current_node, goal):
    if current_node == goal:
        return True
    for next_node in get_neighbors(tiny_tree, current_node):
        result = exists_in(next_node,goal)
        if result:
            return result
    return False

print(exists_in("home", "pset.py"))
```

Function calls

```
exists_in("home", "pset.py")
        ↓
exists_in(”school", "pset.py")
        ↓
exists_in(”fall25", "pset.py")
        ↓
exists_in(”6.100", "pset.py")
```

Recap

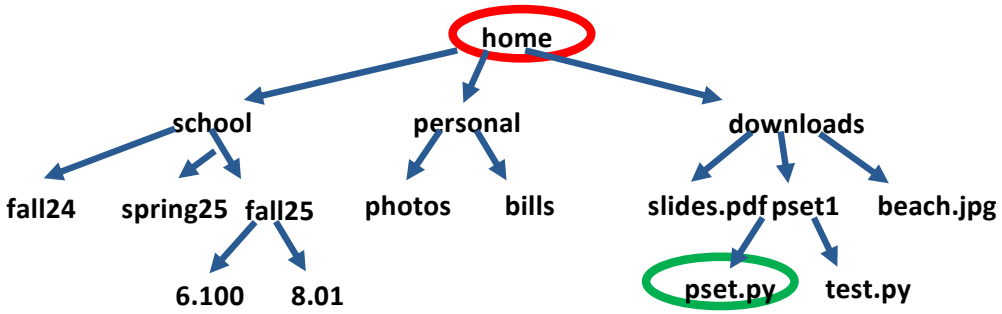



```
tiny_tree = {
        "home": ["school", "personal", "downloads"],
        "school": ["fall24", "spring25", "fall25"],
        "fall25": ["6.100", "8.01"],
        "personal": ["photos", "bills"],
        "downloads": ["slides.pdf", "ps1", "beach.jpg"],
        "ps1": ["pset.py", "test.py"],
   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

def exists_in(current_node, goal):
    if current_node == goal:
        return True
    for next_node in get_neighbors(tiny_tree, current_node):
        result = exists_in(next_node,goal)
        if result:
            return result
    return False

print(exists_in("home", "pset.py"))
```

Function calls

```
exists_in("home", "pset.py")
        ↓
exists_in(”school", "pset.py")
        ↓
exists_in(”fall25", "pset.py")
        ↓
exists_in(”8.01", "pset.py")
```

Recap

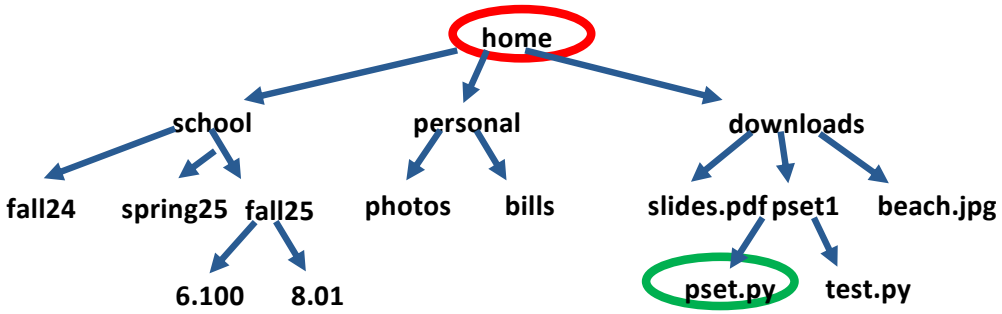



```
tiny_tree = {
        "home": ["school", "personal", "downloads"],
        "school": ["fall24", "spring25", "fall25"],
        "fall25": ["6.100", "8.01"],
        "personal": ["photos", "bills"],
        "downloads": ["slides.pdf", "ps1", "beach.jpg"],
        "ps1": ["pset.py", "test.py"],
   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

def exists_in(current_node, goal):
    if current_node == goal:
        return True
    for next_node in get_neighbors(tiny_tree, current_node):
        result = exists_in(next_node,goal)
        if result:
            return result
    return False

print(exists_in("home", "pset.py"))
```

Function calls

```
exists_in("home", "pset.py")
        ↓
exists_in(”school", "pset.py")
        ↓
exists_in(”fall25", "pset.py")
        ↓
exists_in(”8.01", "pset.py")
```

Recap

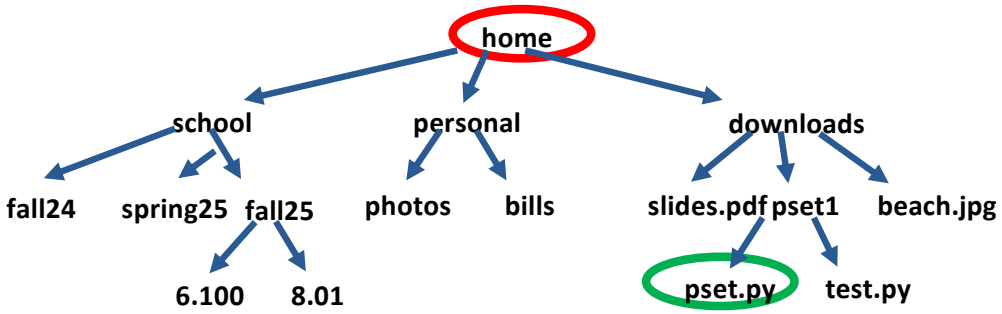



```
tiny_tree = {
        "home": ["school", "personal", "downloads"],
        "school": ["fall24", "spring25", "fall25"],
        "fall25": ["6.100", "8.01"],
        "personal": ["photos", "bills"],
        "downloads": ["slides.pdf", "ps1", "beach.jpg"],
        "ps1": ["pset.py", "test.py"],
   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

def exists_in(current_node, goal):
    if current_node == goal:
        return True
    for next_node in get_neighbors(tiny_tree, current_node):
        result = exists_in(next_node,goal)
        if result:
            return result
    return False

print(exists_in("home", "pset.py"))
```

Function calls

```
exists_in("home", "pset.py")
        ↓
exists_in(”personal", "pset.py")
        ↓
exists_in(”photos", "pset.py")
```

Recap

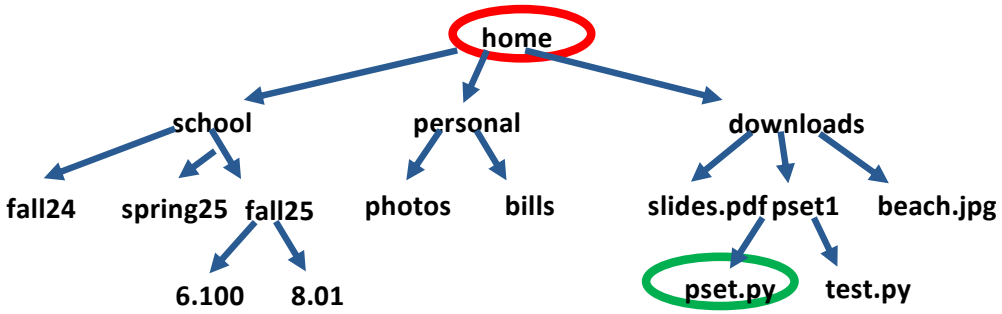



```
tiny_tree = {
        "home": ["school", "personal", "downloads"],
        "school": ["fall24", "spring25", "fall25"],
        "fall25": ["6.100", "8.01"],
        "personal": ["photos", "bills"],
        "downloads": ["slides.pdf", "ps1", "beach.jpg"],
        "ps1": ["pset.py", "test.py"],
   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

def exists_in(current_node, goal):
    if current_node == goal:
        return True
    for next_node in get_neighbors(tiny_tree, current_node):
        result = exists_in(next_node,goal)
        if result:
            return result
    return False

print(exists_in("home", "pset.py"))
```

Function calls

```
exists_in("home", "pset.py")
        ↓
exists_in(”personal", "pset.py")
        ↓
exists_in(”bills", "pset.py")
```

Recap

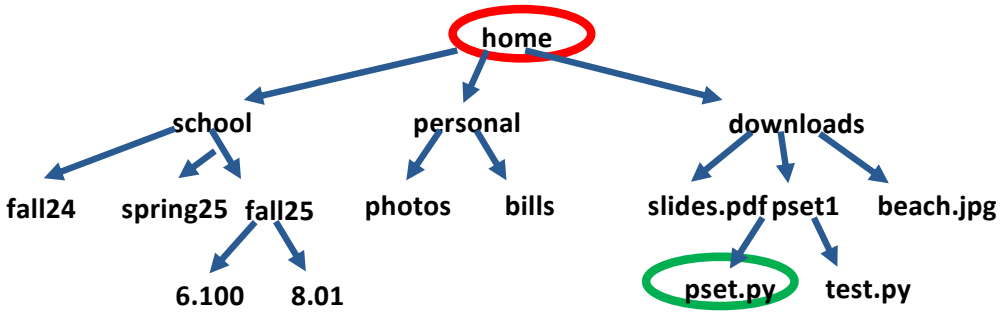



```
tiny_tree = {
        "home": ["school", "personal", "downloads"],
        "school": ["fall24", "spring25", "fall25"],
        "fall25": ["6.100", "8.01"],
        "personal": ["photos", "bills"],
        "downloads": ["slides.pdf", "ps1", "beach.jpg"],
        "ps1": ["pset.py", "test.py"],
   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

def exists_in(current_node, goal):
    if current_node == goal:
        return True
    for next_node in get_neighbors(tiny_tree, current_node):
        result = exists_in(next_node,goal)
        if result:
            return result
    return False

print(exists_in("home", "pset.py"))
```

Function calls

```
exists_in("home", "pset.py")
        ↓
exists_in(”personal", "pset.py")
        ↓
exists_in(”bills", "pset.py")
```

Recap

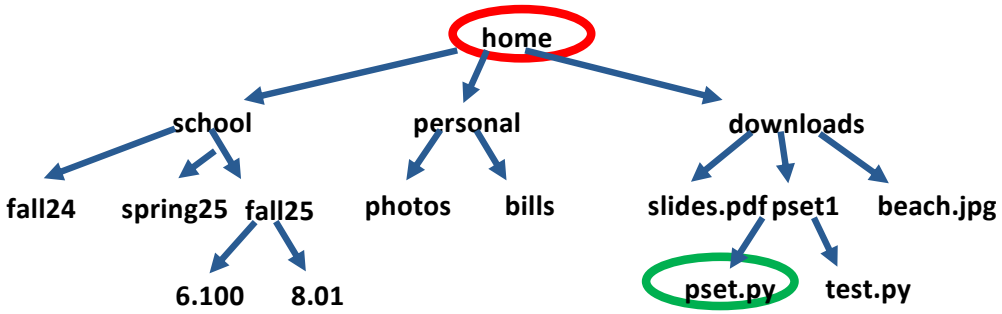



```
tiny_tree = {
        "home": ["school", "personal", "downloads"],
        "school": ["fall24", "spring25", "fall25"],
        "fall25": ["6.100", "8.01"],
        "personal": ["photos", "bills"],
        "downloads": ["slides.pdf", "ps1", "beach.jpg"],
        "ps1": ["pset.py", "test.py"],
   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

def exists_in(current_node, goal):
    if current_node == goal:
        return True
    for next_node in get_neighbors(tiny_tree, current_node):
        result = exists_in(next_node,goal)
        if result:
            return result
    return False

print(exists_in("home", "pset.py"))
```

Function calls

```
exists_in("home", "pset.py")
        ↓
exists_in(”downloads", "pset.py")
        ↓
exists_in(”slides.pdf", "pset.py")
```

Recap




```
tiny_tree = {
        "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"],
    }

def exists_in(current_node, goal):
    if current_node == goal:
        return True
    for next_node in get_neighbors(tiny_tree, current_node):
        result = exists_in(next_node,goal)
        if result:
            return result
    return False

print(exists_in("home", "pset.py"))
```

Function calls

```
exists_in("home", "pset.py")
```
Returns true and terminates

↓

```
exists_in(”downloads", "pset.py")
```
Returns true and terminates

↓

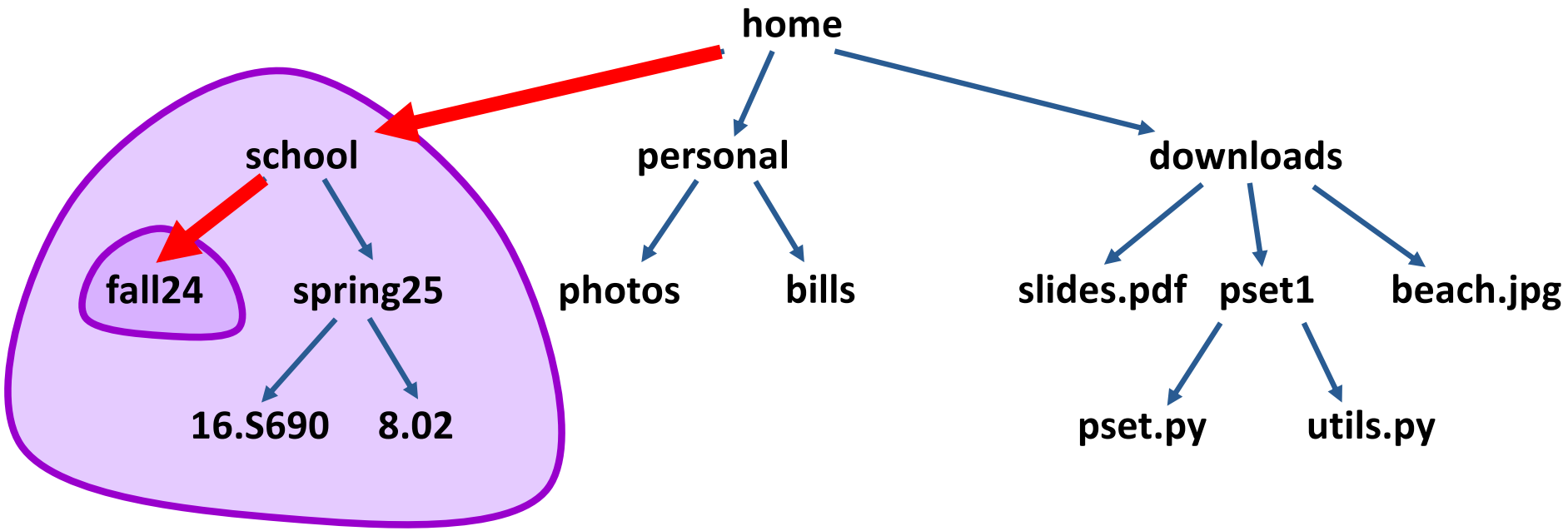
```
exists_in(”pset1", "pset.py")
```
Returns true and terminates

↓

```
exists_in(”pset.py", "pset.py")
```

Depth-first search (last lecture)

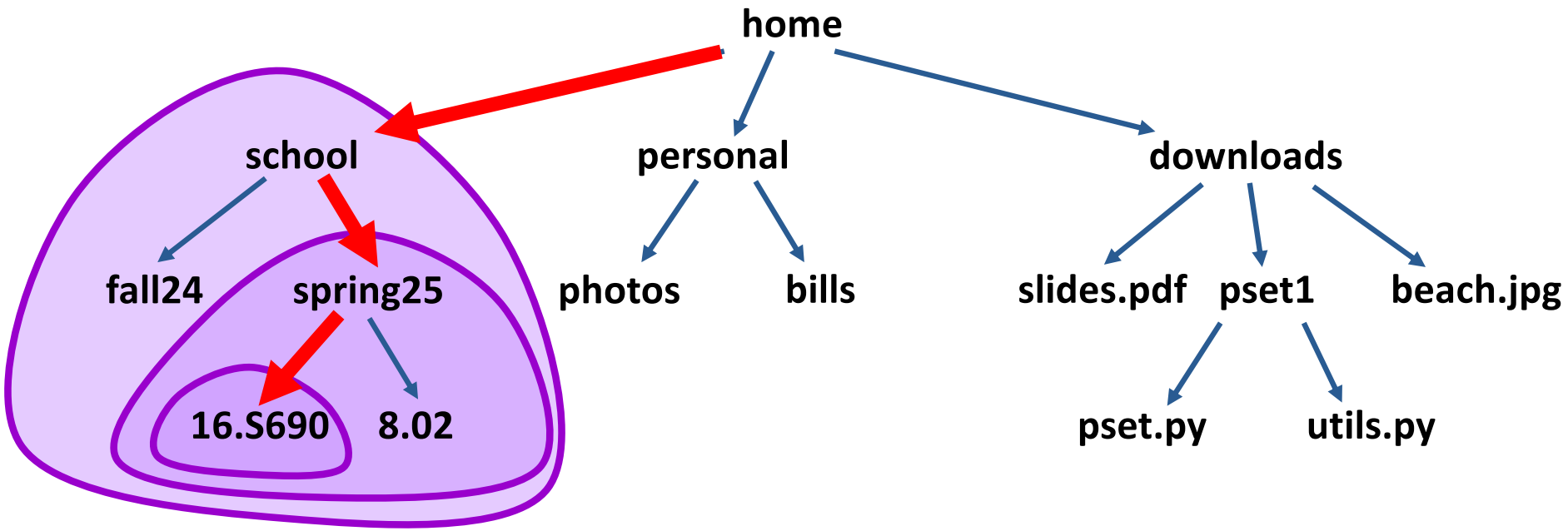
- Keep exploring children before considering siblings
- After branching on child, recursively find a path to target, using **child as new root**



Approach 1: Leverage recursive structure

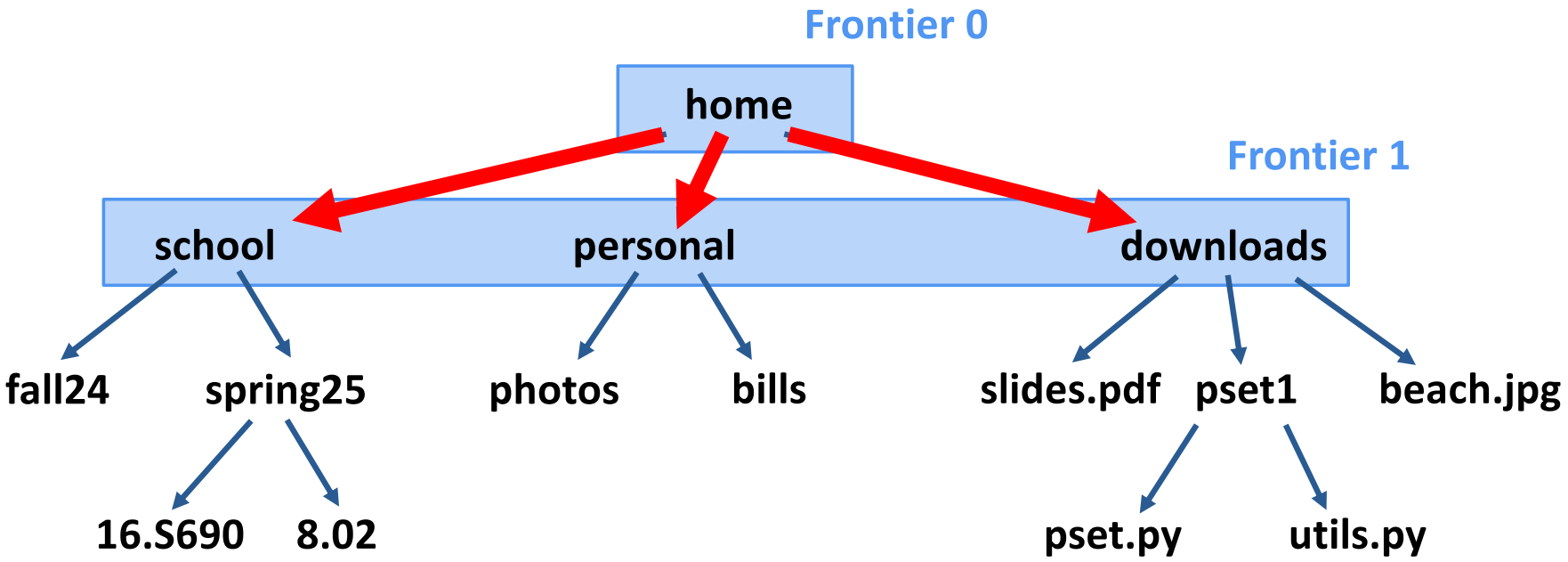
- Keep exploring children before considering siblings
- After branching on child, recursively find a path to target, using **child as new root**

Depth-First Search (DFS)



Approach 2: Scan across branches

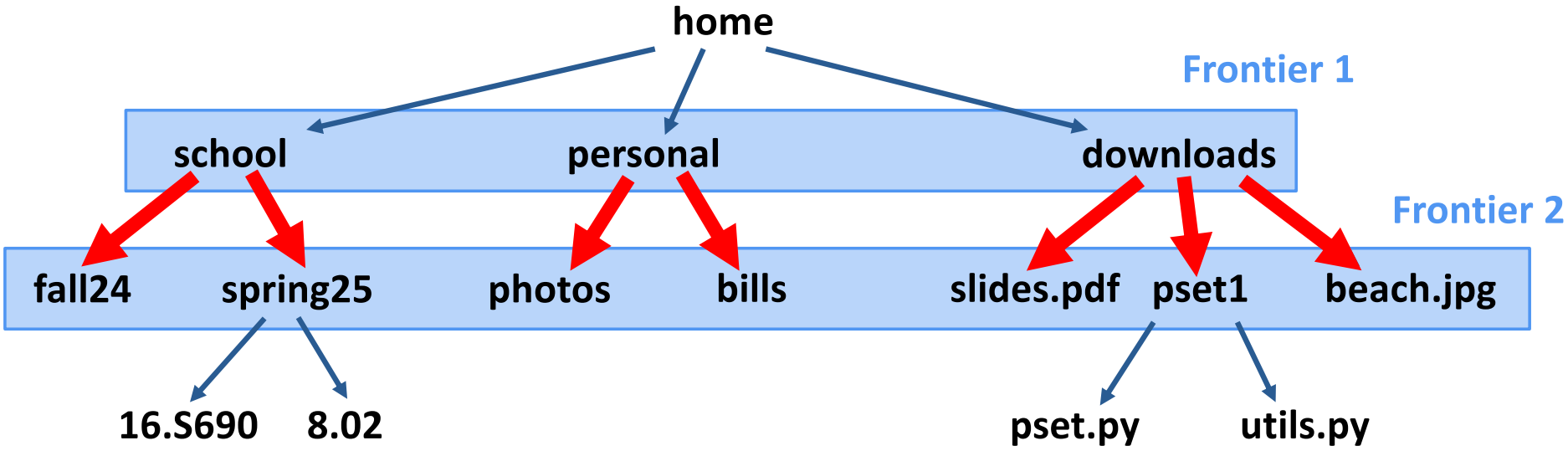
- Follow **successive layers** of depth away from the root
  - Layers are **"frontiers"**
- Each frontier is **one step away** from the previous frontier



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)



BFS on Trees

```
def bfs_tree(graph, start, goal):
    current_frontier = [[start]]
    next_frontier = []

    while len(current_frontier) > 0:
        print("Current frontier:", pathlist_to_string(current_frontier))

        for path in current_frontier:
            print("  Current BFS path:", path_to_string(path))

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

            for next_node in get_neighbors(graph, current_node):
                next_frontier.append(path + [next_node])

        current_frontier, next_frontier = next_frontier, []

    return None
```

Each frontier is a list of paths from the root into the nodes in that frontier

Stop search once we reach target

Append all children of current frontier nodes into next frontier

Slide frontier window down a level

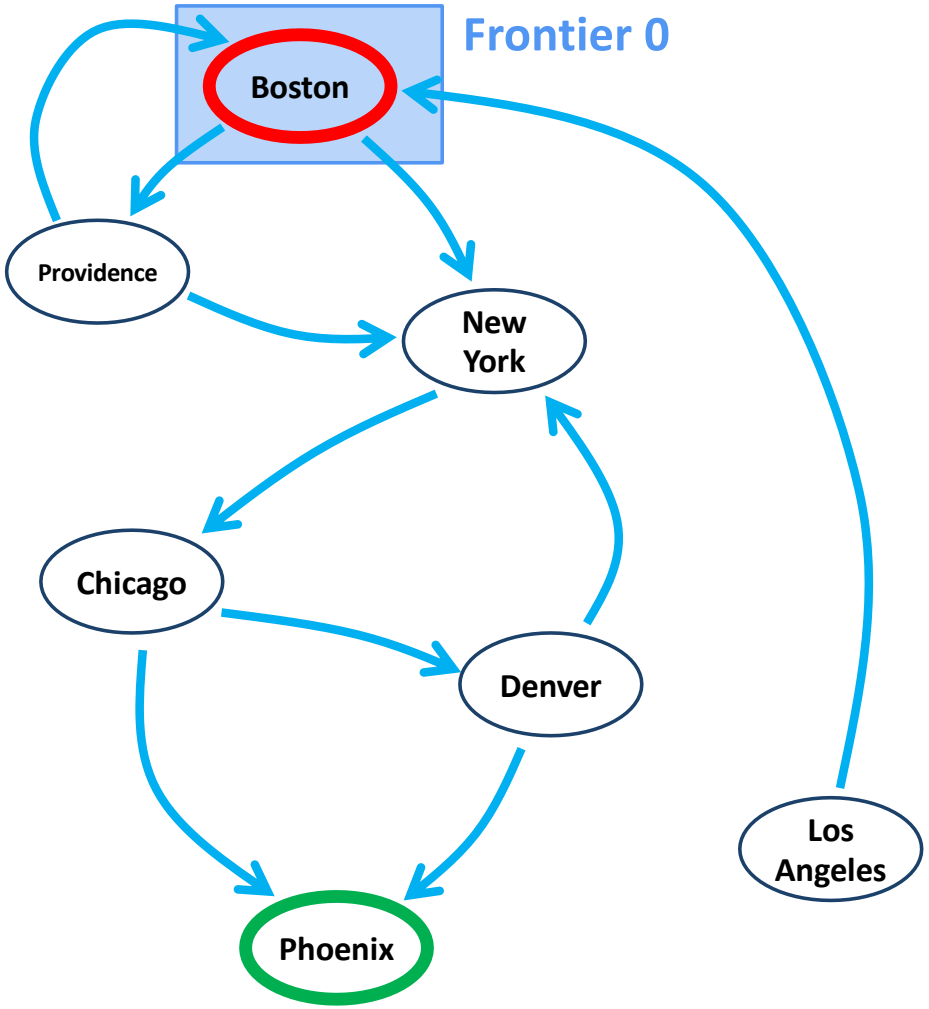
Factors affecting performance

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

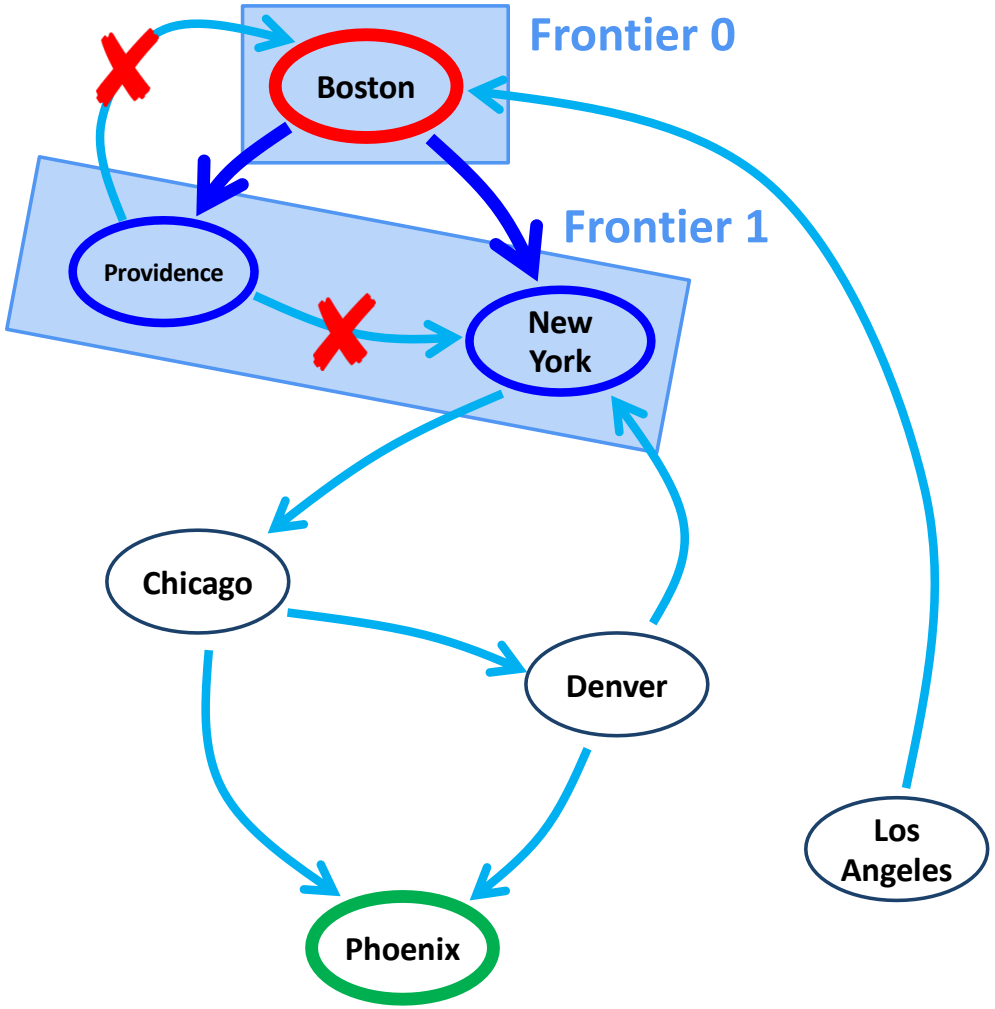
BFS for graphs

- 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**



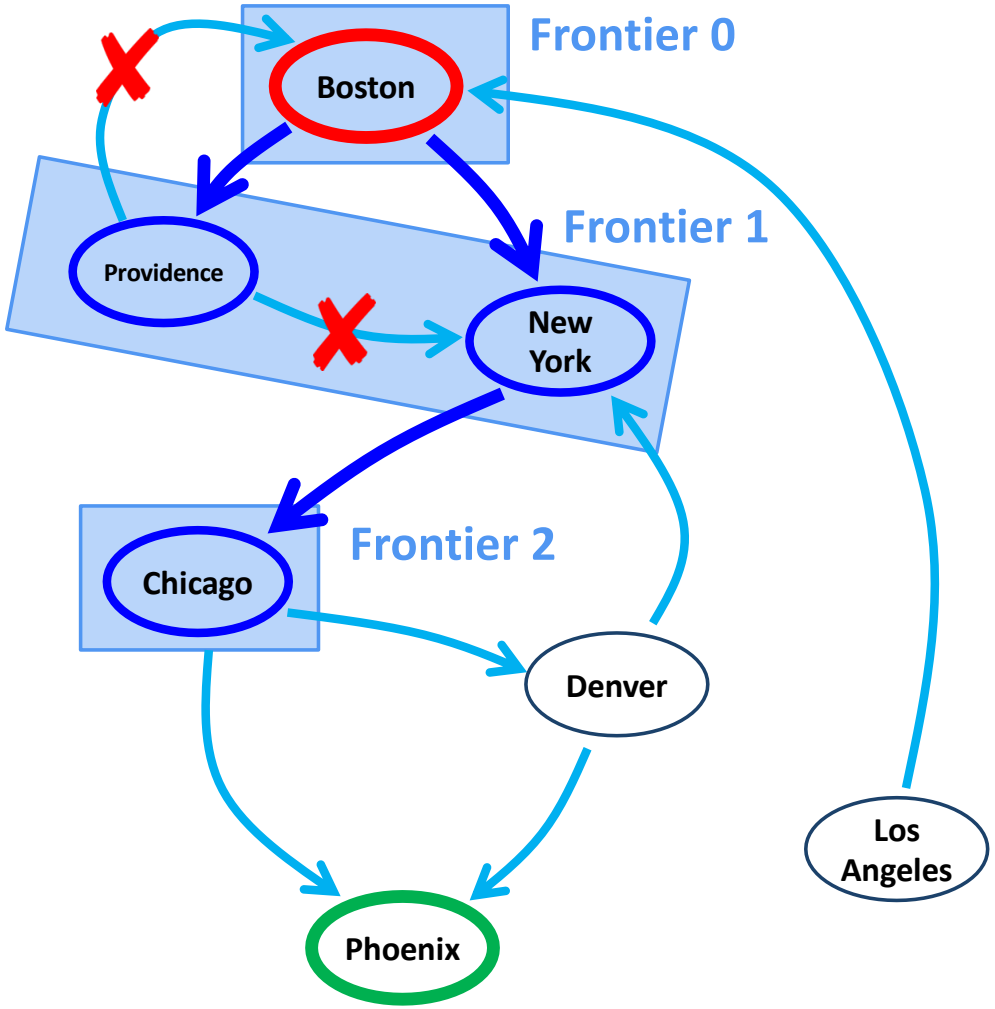
BFS: Avoiding previously seen nodes

- 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**



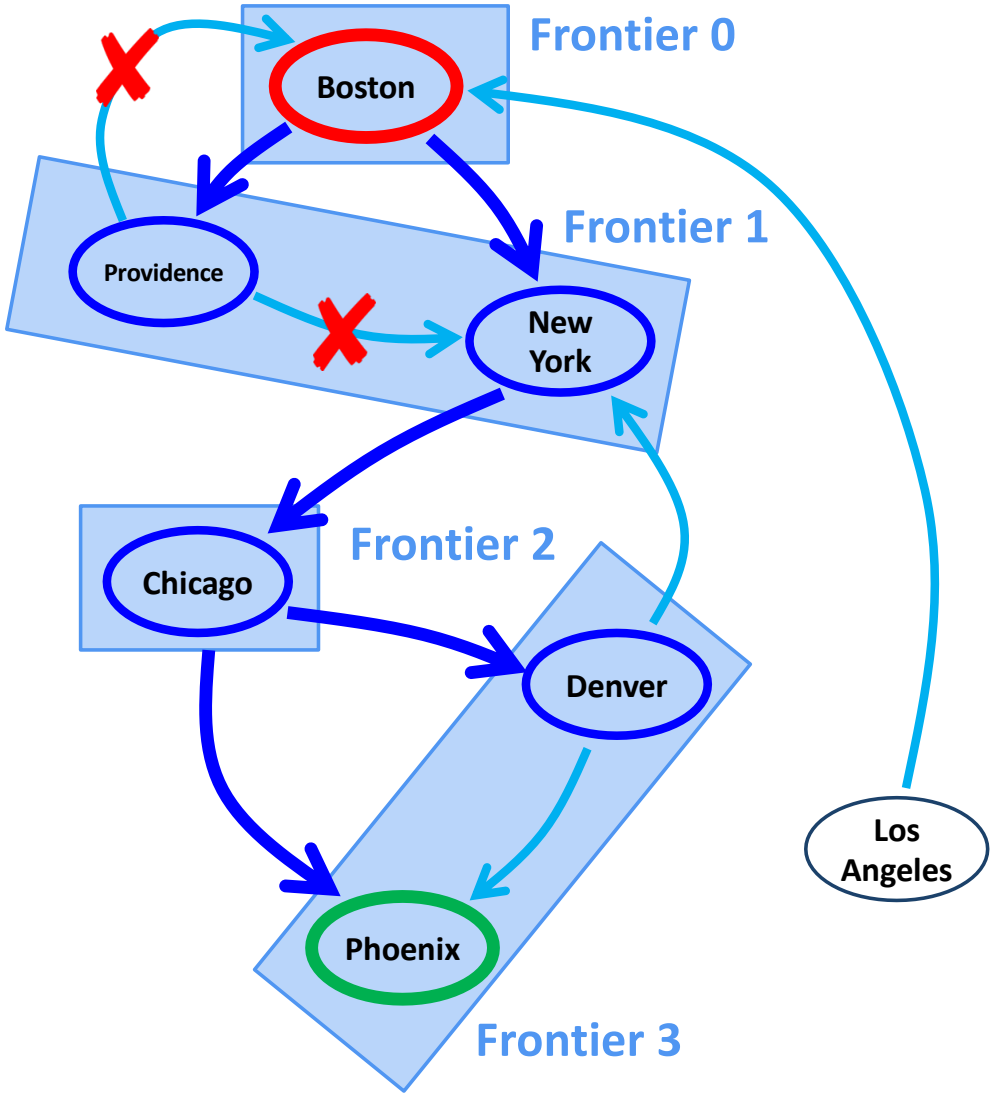
BFS: Avoiding previously seen nodes

- 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**



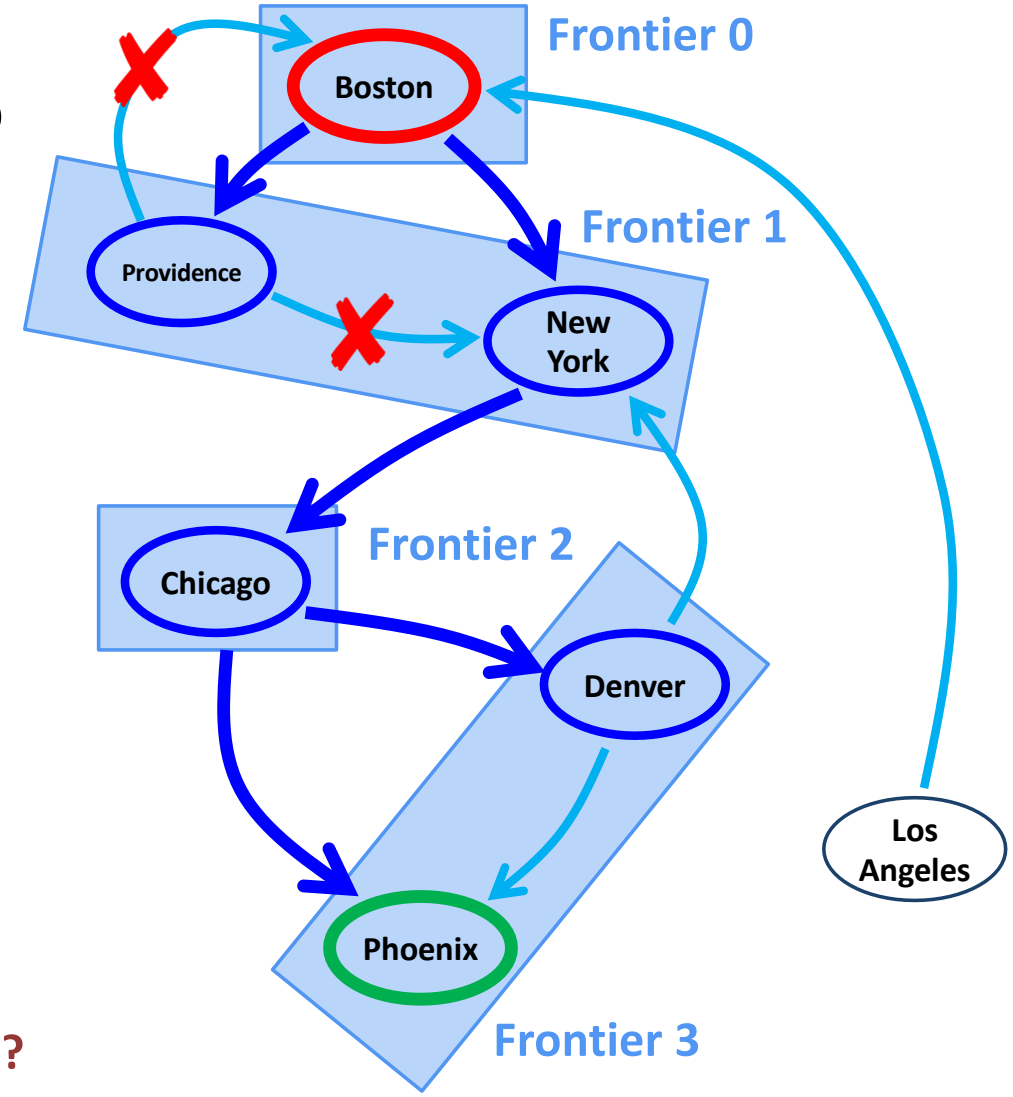
BFS: Avoiding previously seen nodes

- 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: Avoiding previously seen nodes

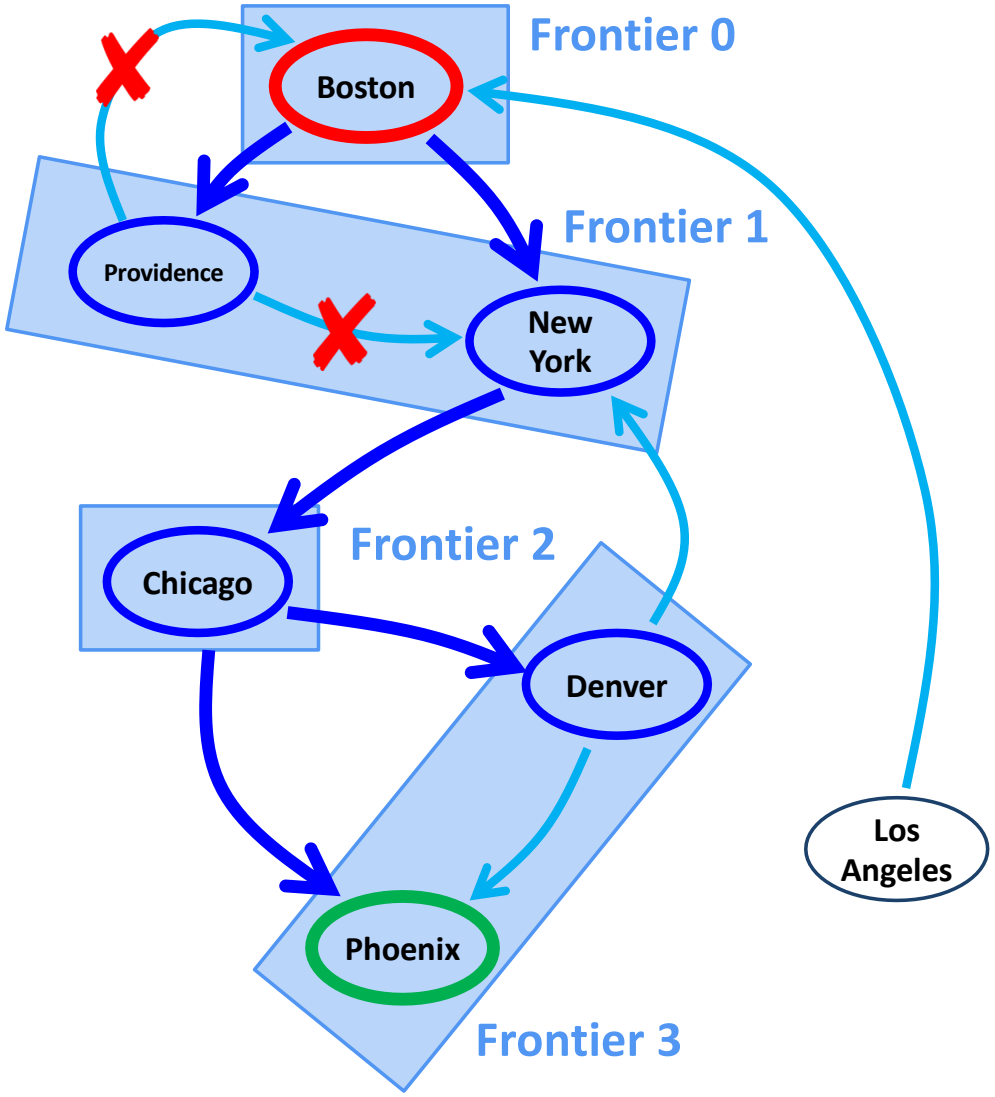
- Already **storing paths** so far when expanding their end nodes
- Can do even better: Reject any new child already in a **previous or current frontier**



Question:
Will this always return the shortest path?

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**



BFS on Graphs

```
def bfs_graph(graph, start, goal):
    current_frontier = [[start]]
    next_frontier = []
    visited = {start}

    while len(current_frontier) > 0:
        print("Current frontier:", pathlist_to_string(current_frontier))

        for path in current_frontier:
            print("  Current BFS path:", path_to_string(path))

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

            for next_node in get_neighbors(graph, current_node):

                # avoid nodes already seen in current or previous frontier
                if next_node in visited:
                    print(f"    AVOID revisiting {next_node}")
                    continue
                visited.add(next_node)
                next_frontier.append(path + [next_node])

        current_frontier, next_frontier = next_frontier, []

    return None
```

TAKEAWAYS AND CONSIDERATIONS

Generalizing goal check to goal test function

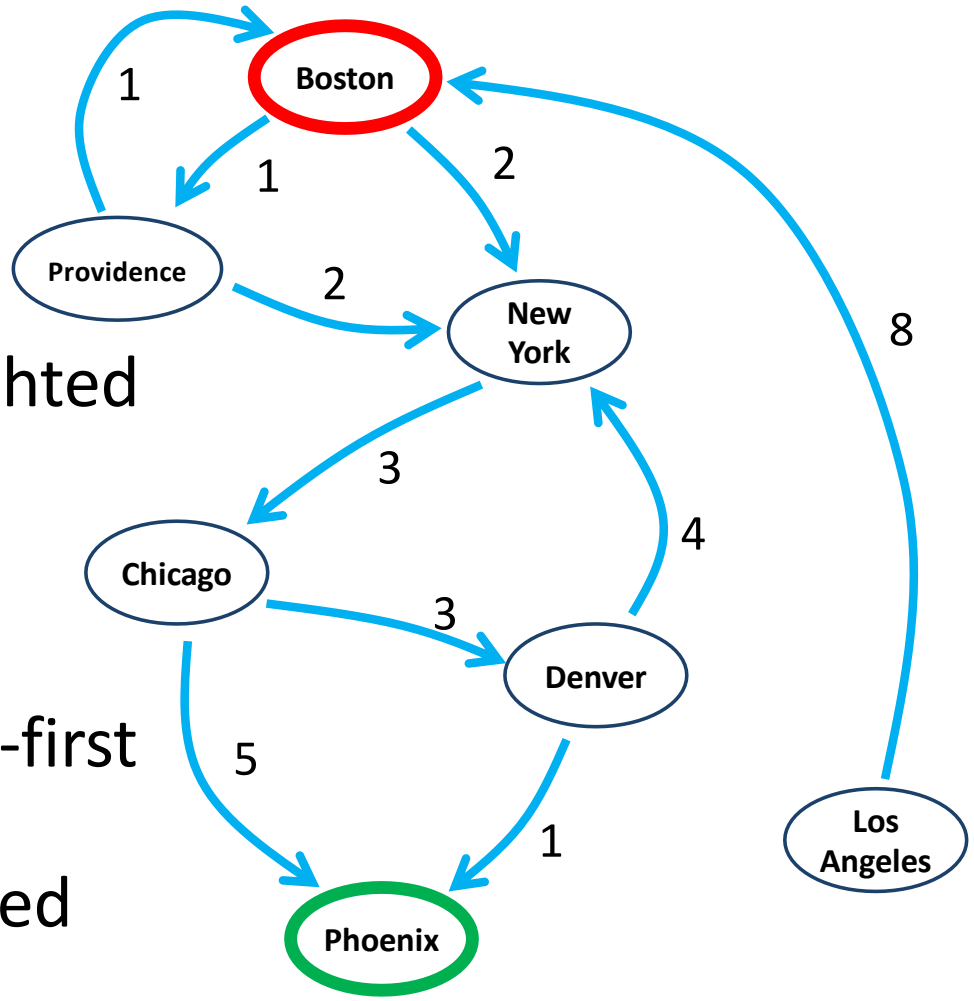
- Goal may be defined in terms of properties rather than a single state
- Instead of checking **`state == goal`**, abstract away into **`goal_test(state)`**
- Beware of function's complexity
  - Gets checked on every state, has a multiplicative effect

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

What if the graph is weighted?

- BFS always returns the shortest path for unweighted graph
- Is that also true for weighted graphs?
- If it isn't can you think of a "simple" way to use our existing breadth-first search to also find the shortest path for weighted graphs?



How do we store a weights

```
flights = {
    "Boston": [("Providence", 1), ("New York", 2)],
    "Providence": [("Boston", 1), ("New York", 2)],
    "New York": [("Chicago", 3)],
    "Chicago": [("Denver", 3), ("Phoenix", 5)],
    "Denver": [("New York", 4), ("Phoenix", 1)],
    "Los Angeles": [("Boston", 8)],
}
```

```
def add_edge(graph, u, v):
        if u not in graph:
            graph[u] = []
        graph[u].append(v)

def unroll_weighted_graph(graph):
    new_graph = {}
    for u, edges in graph.items():
        for v, w in edges:
            if w == 1:
                add_edge(new_graph, u, v)
            else:
                # create intermediate nodes
                prev = u
                for i in range(1, w):
                    mid = f"{u}->{v}#{i}"
                    add_edge(new_graph, prev, mid)
                    prev = mid
                add_edge(new_graph, prev, v)
    return new_graph
```