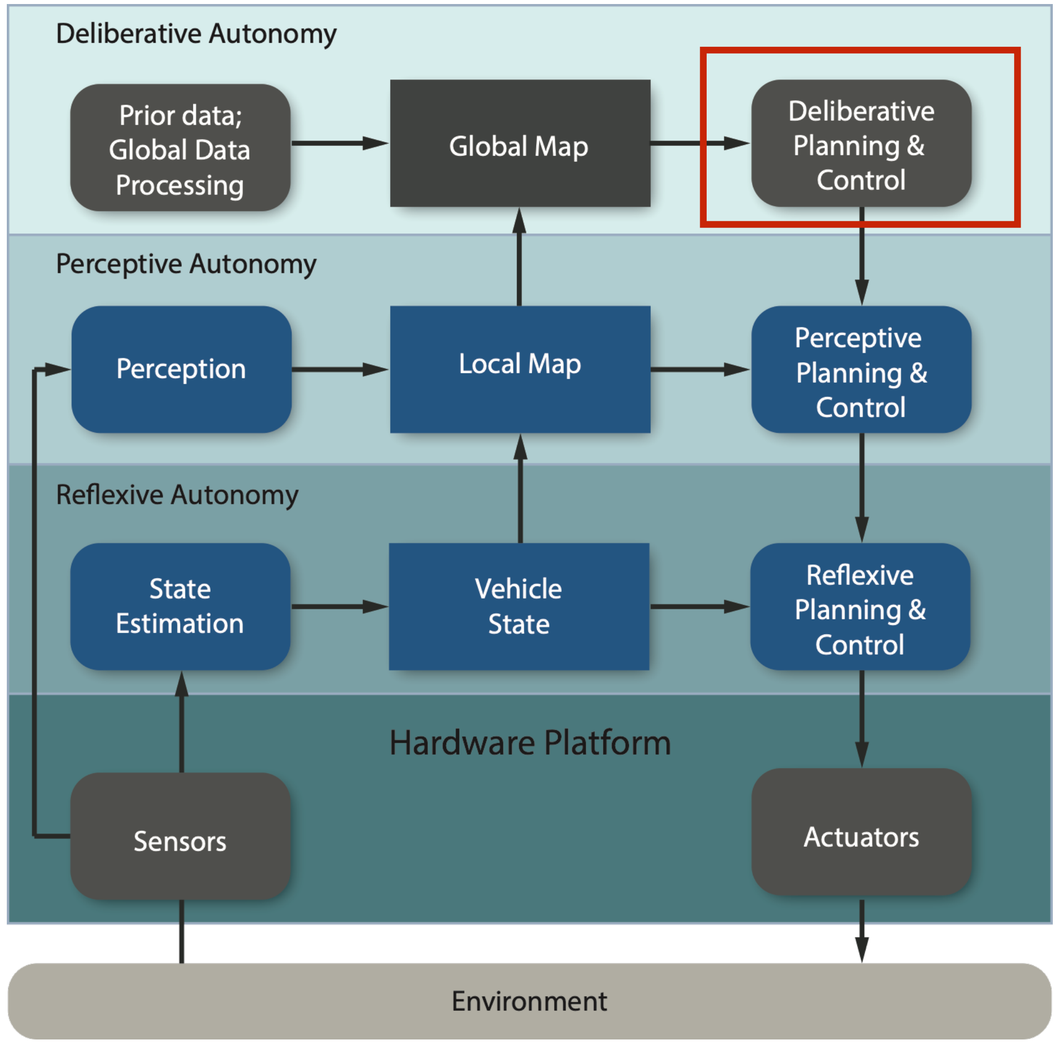
More on Breadth-first and Depth-first Search

Last Time

- Discussed different levels of autonomy
- In the coming lecture, we will focus on the “Deliberative Planning” system
- In future lectures, we will explore different parts of this architecture

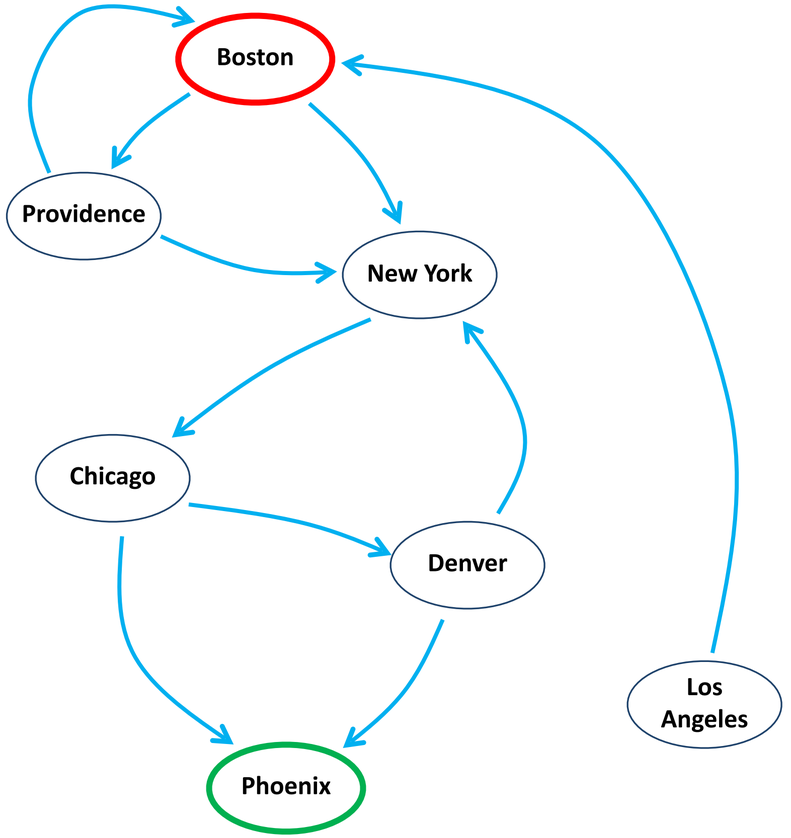


Graphs and Trees

- **Graph** is a key data structure that encodes relationships:

$$G = (V, E)$$

- Typically, we are interested in finding a path (sequence of edges) that connects a **start** vertex with a **goal** vertex.



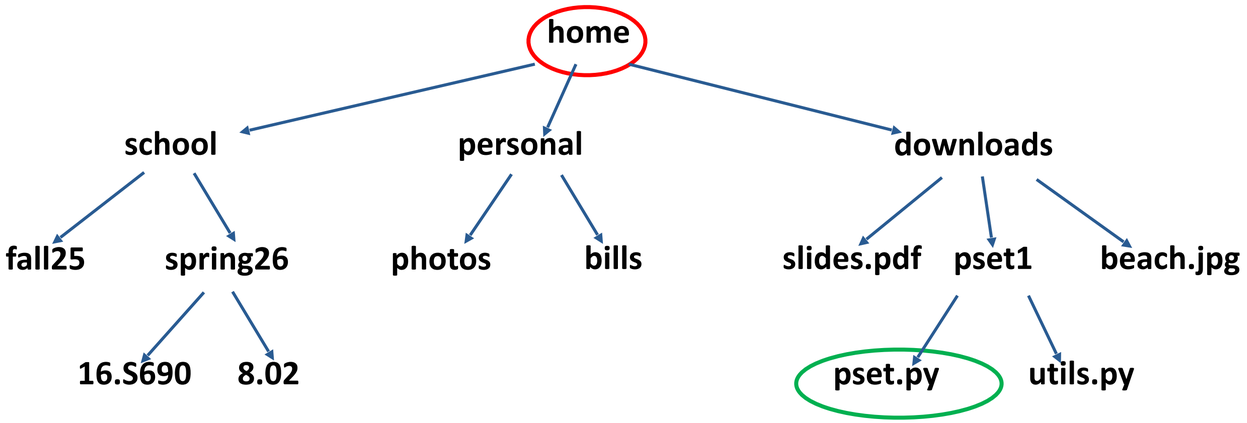
Graphs and Trees

- **Graph** is a key data structure that encodes relationships:

$$G = (V, E)$$

- **Tree** is a graph that has no cycles
  - This implies each vertex in a tree has exactly one parent, except for a "root" vertex that has no parent.

Filesystem is a tree!



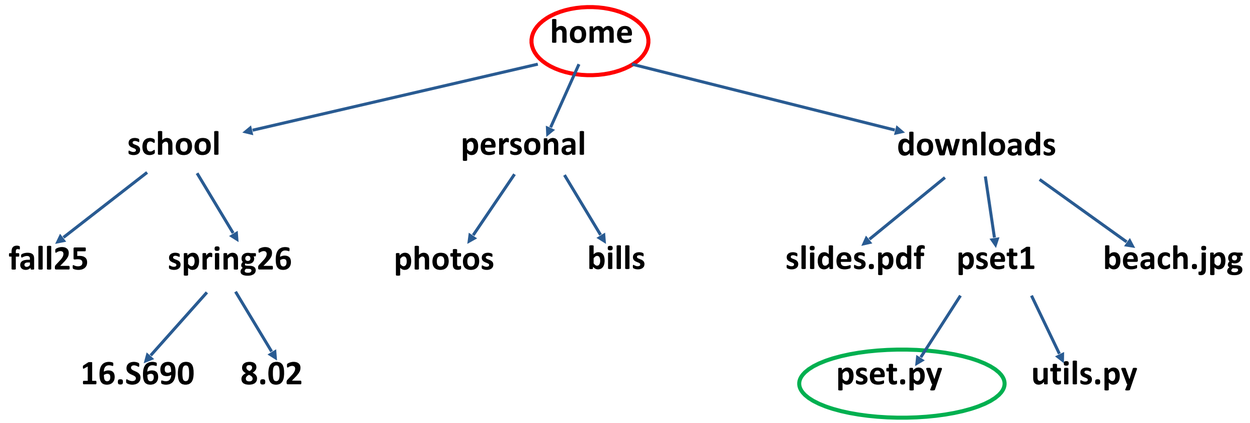
Graphs and Trees

- **Root** is the top-level vertex with no parent

- A vertex *v* is a **parent** to a vertex *w,* if there is an edge from v to *w*. In other words, *(v,w)* is an edge in the graph *G*. In this case, *w* is said to be a **child** of

- A vertex *v* is a **leaf** if it has no children.

Filesystem is a tree!

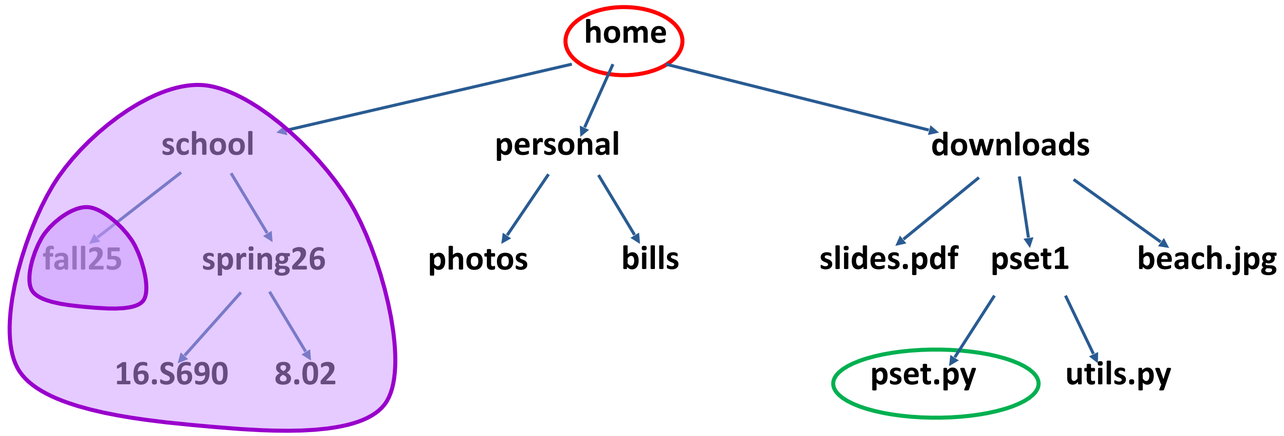


Exploring Trees for Search:
Approach 1 - Leverage Recursive Structure

- **Tree** has a repeating structure: Each child is a root to its own subtree.

- **Search** by repeating the search process for each child vertex

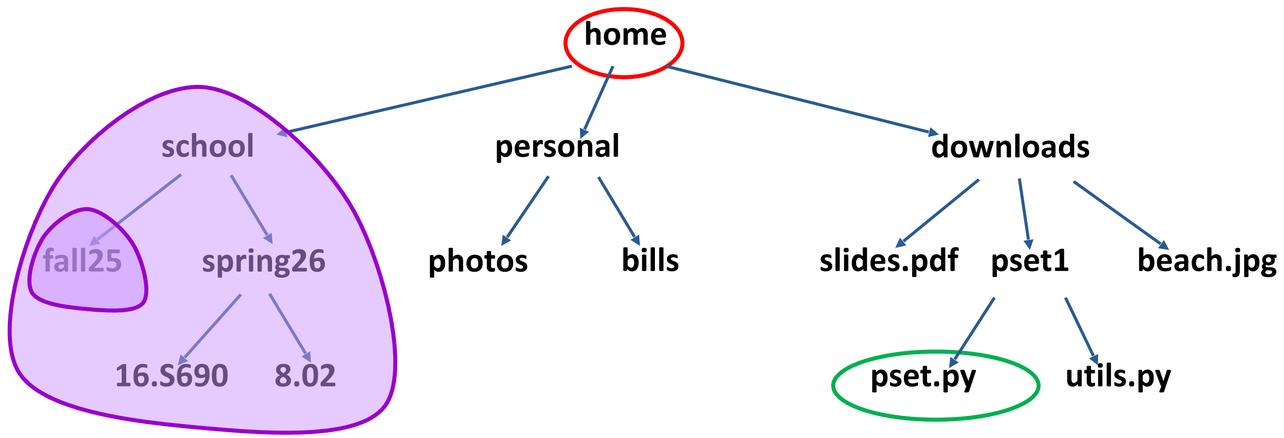
Filesystem is a tree!



Exploring Trees for Search:
Approach 1 - Leverage Recursive Structure

- The recursive search described above results in **Depth-first Search:**
  - Explores the deeper down into the tree, reaching the closest leaf vertex first.

Filesystem is a tree!



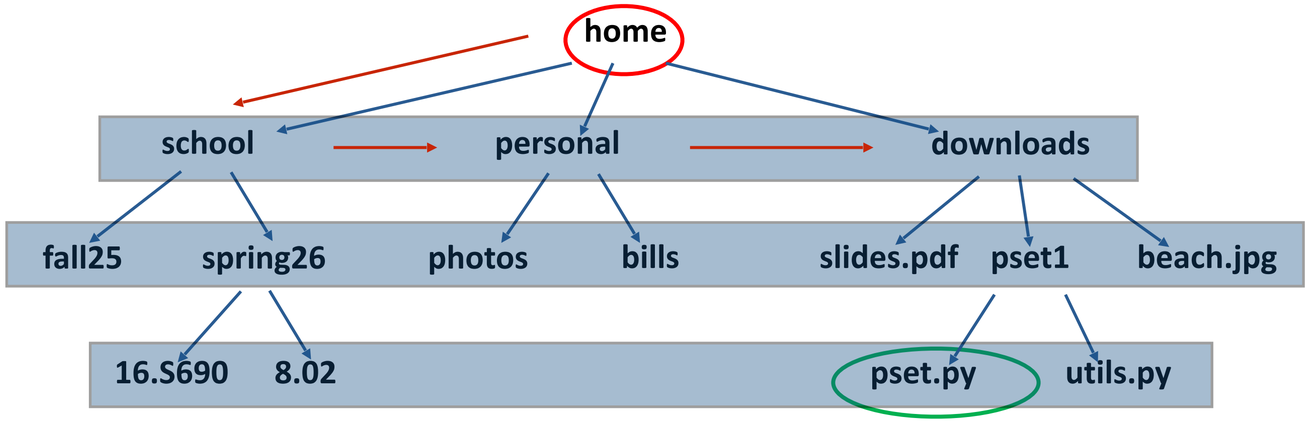
- The depth-first search may get lucky and find the goal vertex quickly. But, in general it has to search the whole tree.

Exploring Trees for Search:
Approach 2 - Use a Queue

- Starting from the root vertex, process each vertex in the order that they were encountered.

- Initially:
  - Put the root vertex in the queue
- At each iteration:
  - Pull a vertex *v* from the queue
  - If *v* is the goal vertex, then return *v*.
  - Put each child of v at the end of the queue

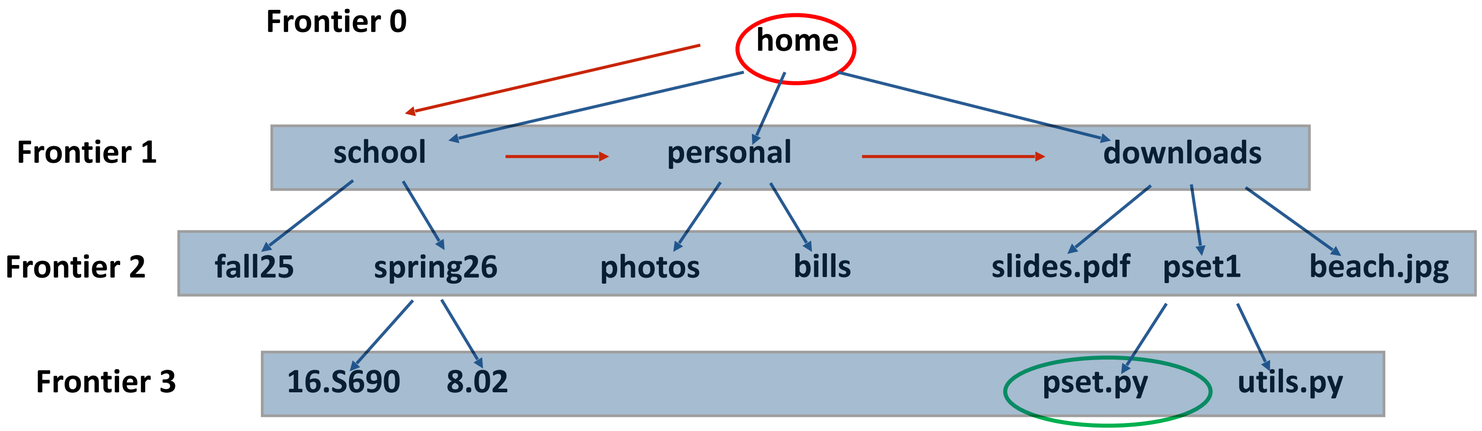
Filesystem is a tree!



Exploring Trees for Search:
Approach 2 - Use a Queue

- The search process described above results in the **Breadth-First Search**:
  - Explore “frontiers” - each vertex in a given frontier is at the same distance from the root. For instance, all vertices in “Frontier 2” is at distance 2 from the root.
  - The algorithm first searches the entire first frontier, and then the entire second frontier, and then the entire third frontier, and so on.

Filesystem is a tree!

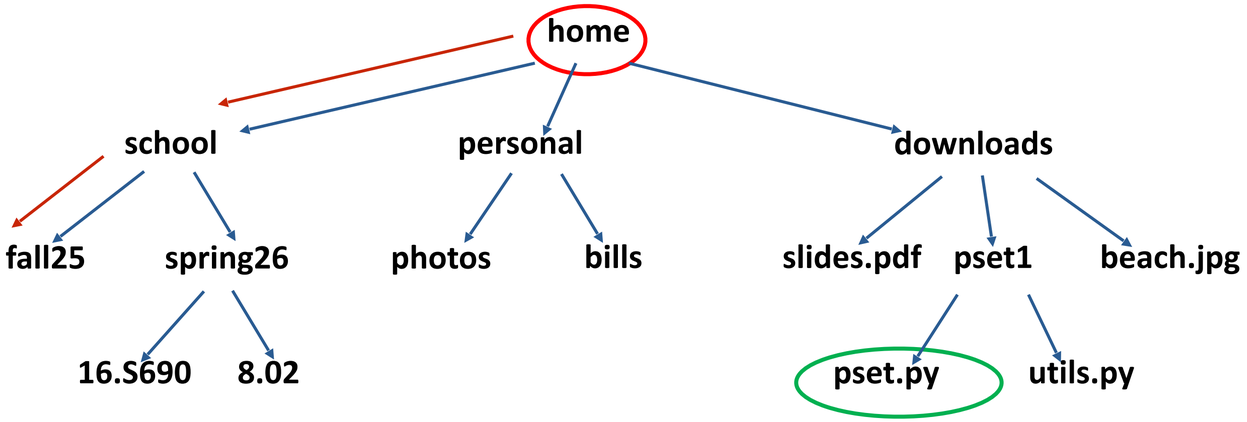


Exploring Trees for Search:
Approach 2 - Use a Queue for Depth-First Search

- It is possible to implement **Depth-First Search using a queue** (instead of recursion) At a high level:

- Initially:
  - Put the root vertex in the queue
- At each iteration:
  - Pull a vertex *v* from the queue
  - If *v* is the goal vertex, then return *v*.
  - Put each child of v in the <u>beginning</u> of the queue

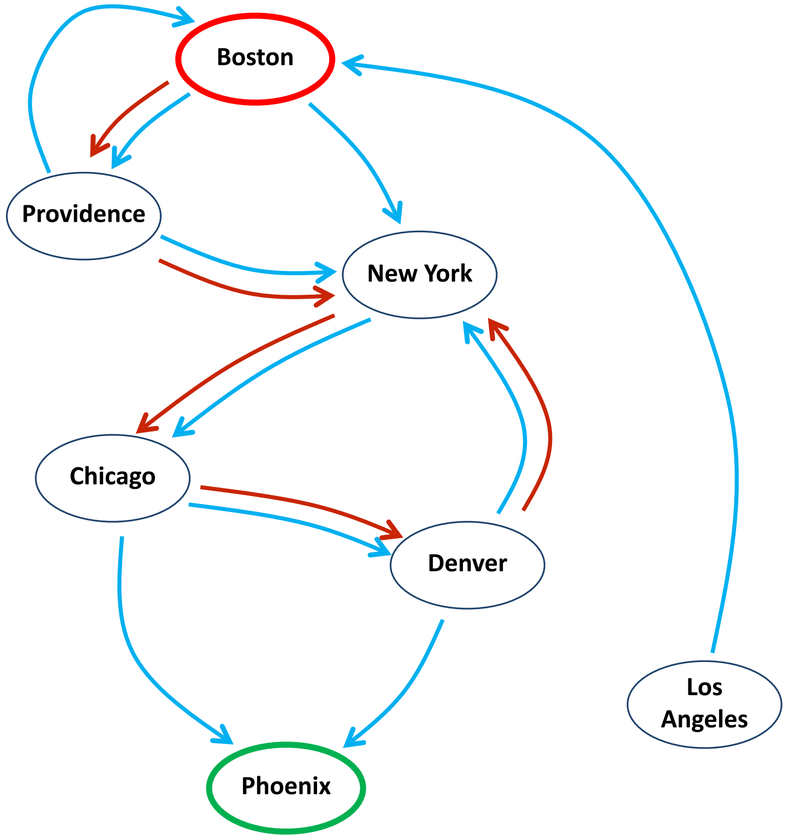
Filesystem is a tree!



How to search graphs?

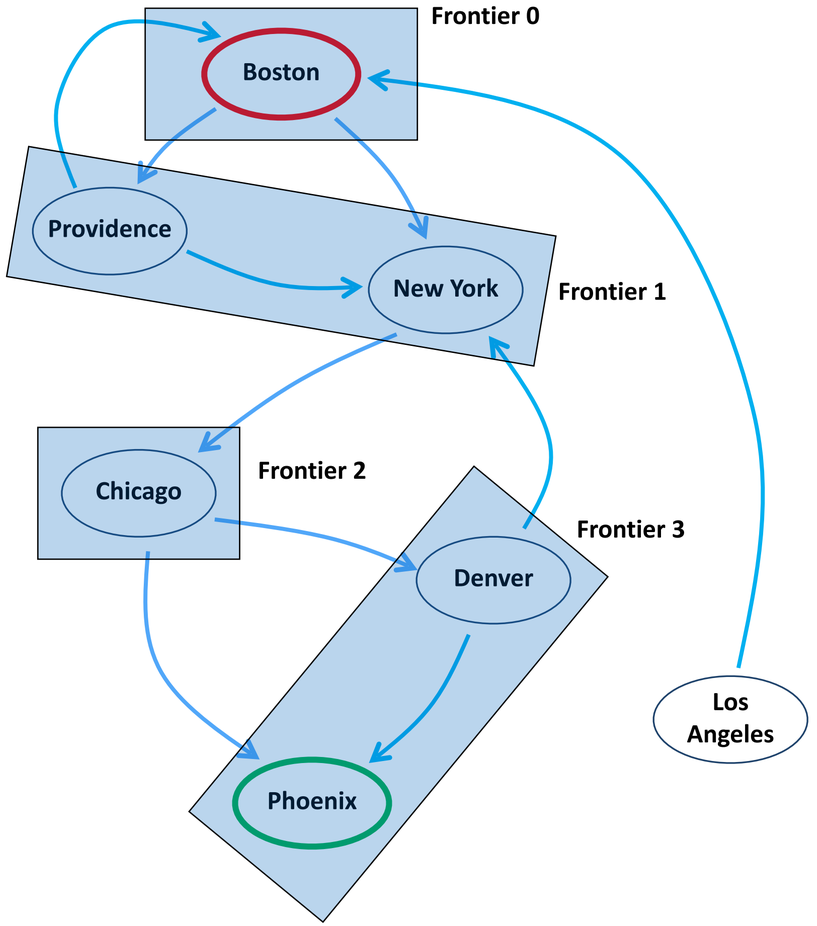
- When searching a tree BFS and DFS will encounter each vertex at most once.

- However, a general graph has cycles. Hence, it is possible that the BFS or DFS search will encounter a vertex a second time. At this point, the algorithm may go into an infinite loop…



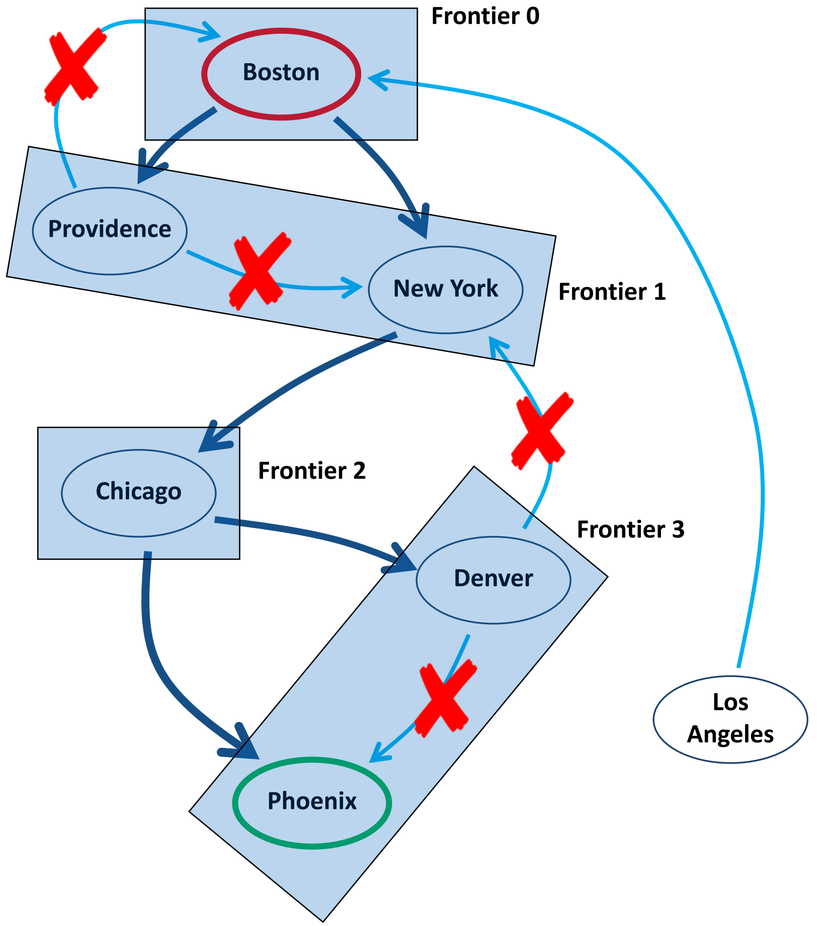
How to search graphs?

- We can apply the key principle behind Breadth-First Search:
  - **Search the frontiers** in increasing distance from the start vertex



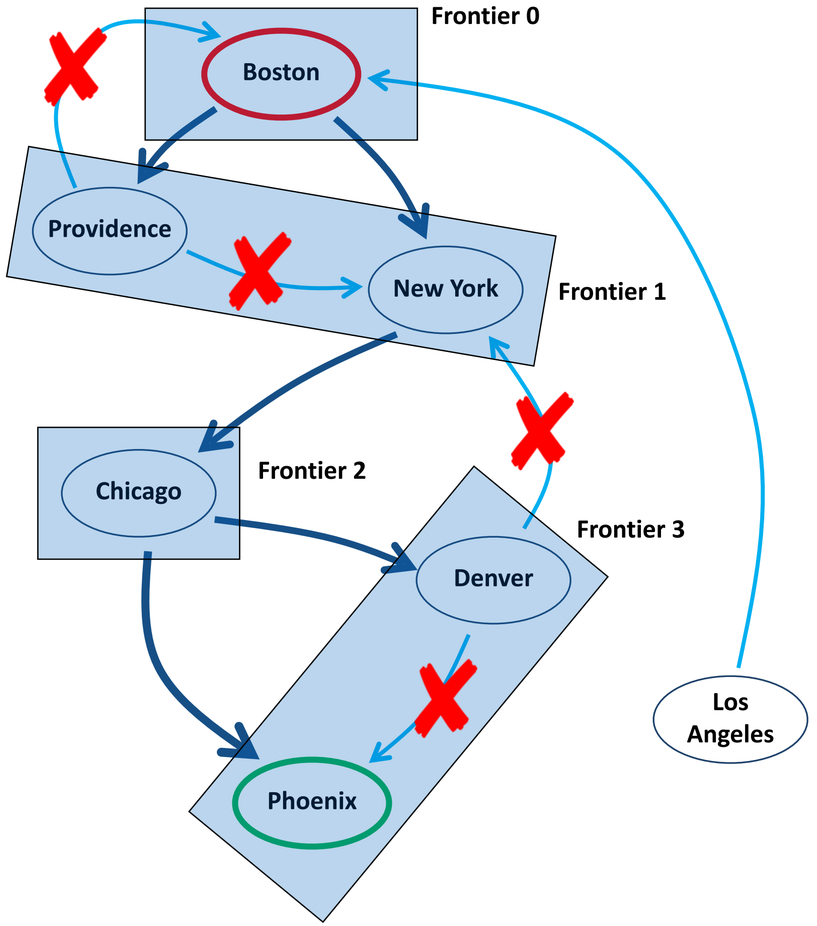
How to search graphs?

- We can apply the key principle behind Breadth-First Search:
  - **Search the frontiers** in increasing distance from the start vertex
  - Simply keep track of all vertices that were visited, and do not consider a vertex that was already visited. (For instance, do not put into the queue a vertex that has already been visited.)



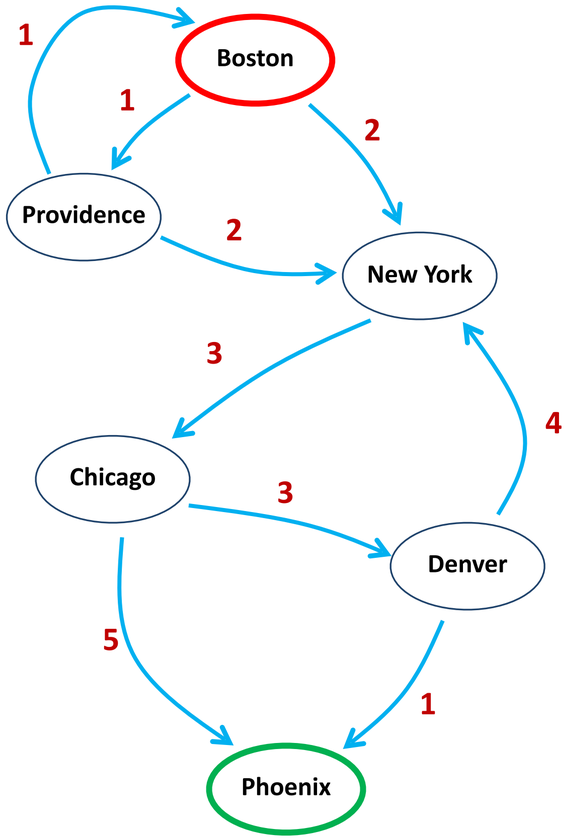
How to search graphs?

- We can apply the key principle behind Breadth-First Search:
  - **Search the frontiers** in increasing distance from the start vertex
  - Simply keep track of all vertices that were visited, and do not consider a vertex that was already visited. (For instance, do not put into the queue a vertex that has already been visited.)



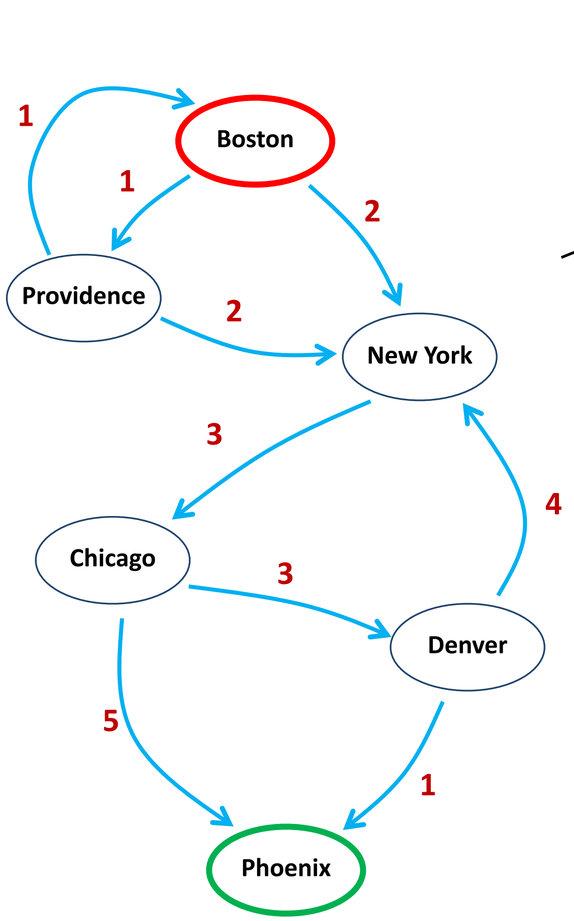
How to Find Shortest Paths with Costs on Edges?

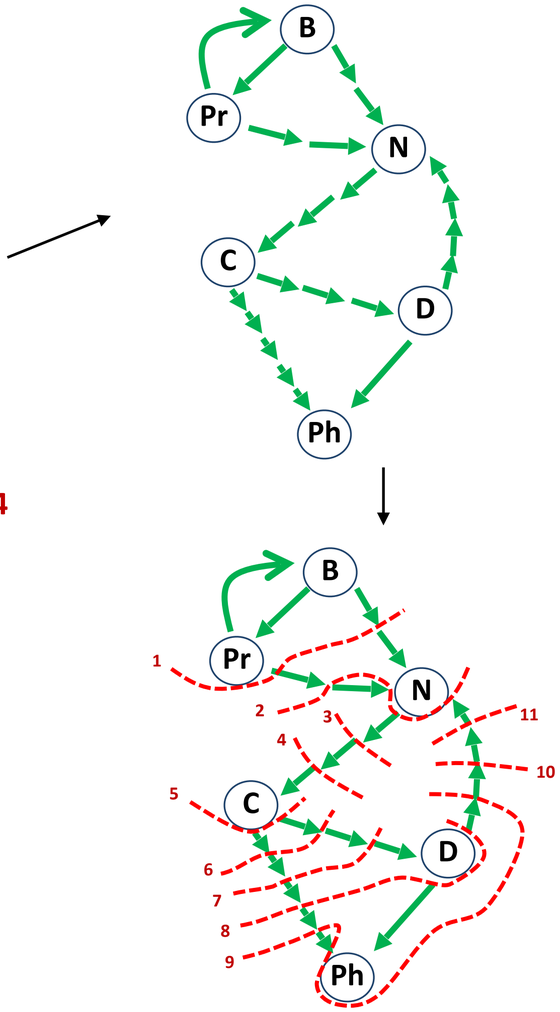
- **Weighted Graph:** Suppose each edge has a "weight" associated with it, which encodes the cost of traversing that edge.
- **Shortest Path Search:** We would like to find a path from start to goal that has minimum cost.



How to Find Shortest Paths with Costs on Edges?

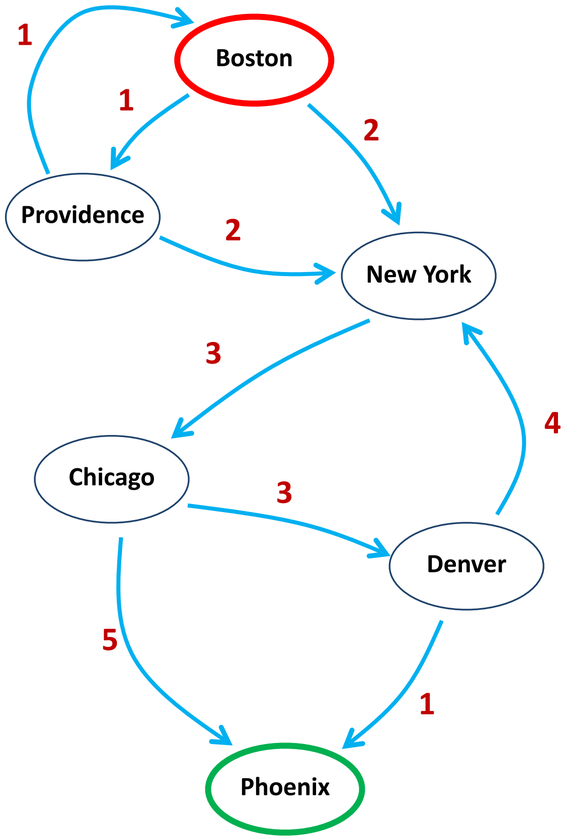
- **Discretizing the graph:**
  - In principle, we could transform this graph into a new graph where each edge has cost exactly 1, simply by placing k-1 vertices between two vertices v and w if the cost of the edge (v,w) is k.
  - We can run BFS in this modified graph. It will return the shortest path.
  - This is very inefficient, even though it does return the shortest path:
    - It particularly struggles if the common denominator of edge costs is very high.





Quick introduction to Dijkstra's algorithm (more later)

- **Dijsktra's algorithm:**
  - Keep track of the cost to get to each vertex, and keep track of the 'parent' that gets to the vertex at the lowest cost.
  - Allow revisiting a vertex, if it lowers the cost to get to that vertex. Otherwise, operates much like the BFS.



Quick introduction to Dijkstra's algorithm (more later)

- Start with (Boston, 0)
- Insert children of Boston: (Pr, 1), (Ny, 1)
- Insert children of Providence:
  - Ny was visited before. Current cost to get to New York is 1. Consider going to Ny from Pr: (Ny, 1+2), which leads to cost of 3. It does not improve the current cost to get to Ny. So, do not change the cost to get to Ny.
- Insert children of New York: (Ch, 2+3)
- Insert children of Chicago: (Ph, 5+5), (D,5+3)
- Phoenix has no children
- Insert children of Denver:
  - Ph was visited before. Current cost to get to Phoenix is 10. Consider going to Ph from D: (Ph, 8+1), which leads to cost of 9. It improves the cost of getting to Phoenix!! So, update the Phoenix cost as 9 and update its parent as Denver.
- Now, you can follow the parent of each vertex to find the shortest path: Ph <– D <– C <– Ny <– B.