Lecture 15 - DFS/BFS

Eric A. Autry

Last time: Use-It-or-Lose-It and Dynamic Programming

This time: DFS/BFS

Next time: Shortest Path

Project 1 due this Friday, the 2nd.

HW 6 due next Thursday, the 8th.

Project 2 will be assigned this Thursday, will cover today's material.

Project 1 Clarification

My testing code assumes that your functions actually edit the input array A.

The return value of your code does not matter.

The testing code assumes that after the call `MergeSort(A)` is made, the array A is actually sorted.

If you use a helper function to do the merging, you will likely run into this problem. You will see that your sorting function returns a sorted list, but fails the test cases because the array A does not get altered.

See the announcement I sent yesterday for more details.

Project 1 Report

The report for project 1 should **not** simply list off answers to the questions I asked.

Instead, I want you to write a report in paragraph form where you discuss the results of the experiments you have performed using the timing code from this project.

In these paragraphs, you should try to address the questions that I ask.

I want you to give me a discussion of what you think the answers to my questions are, and why. And I want you to try to provide some evidence using the figures and output of the `measureTime()` function.

In order to actually answer some of the questions, you may have to experiment a little with this function (for example, change the number of trials or do something like open an internet browser while the code is running to see what happens).

In-Place MergeSort

Last time, I mentioned that MergeSort cannot be implemented in-place *using the same divide & conquer and merging approach that we have discussed*.

It is possible to implement in-place merging and still maintain the $O(n \log n)$ performance.

This is a rather complicated procedure that requires changing the size and order of some of the recursive calls.

Overall, the algorithm still works in fundamentally the same way.

I will not introduce this in-place MergeSort and do not expect you to use it. I just wanted you to know that it does exist.

Use-it-or-Lose-it

There are two major ideas behind Use-it-or-Lose-it:

- We want to iteratively shrink our input with each recursive call to ultimately get to an easily understood base case.
- So we consider each item in the input one at a time, and recursively compare **all possible situations** regarding that item.
  - Change Problem: we looked at each coin one at a time.
  - We could use-it, i.e., we could use that given coin as part of the returned change.
  - We could lose-it, i.e., we could decide not to use that given coin as part of the returned change.
  - These are the only two possible options. So when we return the minimum of these two options, we have found the minimum possible solution.

I have uploaded new, detailed lecture notes for Use-it-or-Lose-it.

Dynamic Programming

Unfortunately, with a recursive Use-it-or-Lose-it program, we only shrink the size of the input by 1 each time.

This would give us exponential runtimes!

But we noticed that we ended up repeating a lot of recursive calls.

Dynamic Programming stores the results of the different recursive calls in a single look-up table.

As long as this table is filled in the correct order (an order that respects the dependencies of the the recursive calls), we can avoid the exponential runtime.

Instead, the runtime is proportional to the size of the table.

Hirshberg method

Several students noted last time that we did not actually need to keep the entire table.

If each entry in the DP table relied only on the entries up, to the left, or up and to the left, then we should only need the previous row and previous column.

This would allow us to use much less space (although it would not change the runtime).

The problem with this idea is that it does not allow us to reconstruct the solution afterwards.

Not all is lost. There is an advanced technique called the Hirshberg method that has this reduction in storage space while still allowing for the reconstruction of the solution.

Mazes

If you are stuck in a maze, how can you find your way out?

What approach would you take if you had a ball of string and some chalk?

- Tie one end of the string to your starting place and **unwind** it as you walk.
- Use the chalk to mark each room you enter so you don't end up walking in circles.
- When you reach a dead end, **rewind** the string to go back to the last room and try another door.

Mazes

How to solve a maze on the computer?

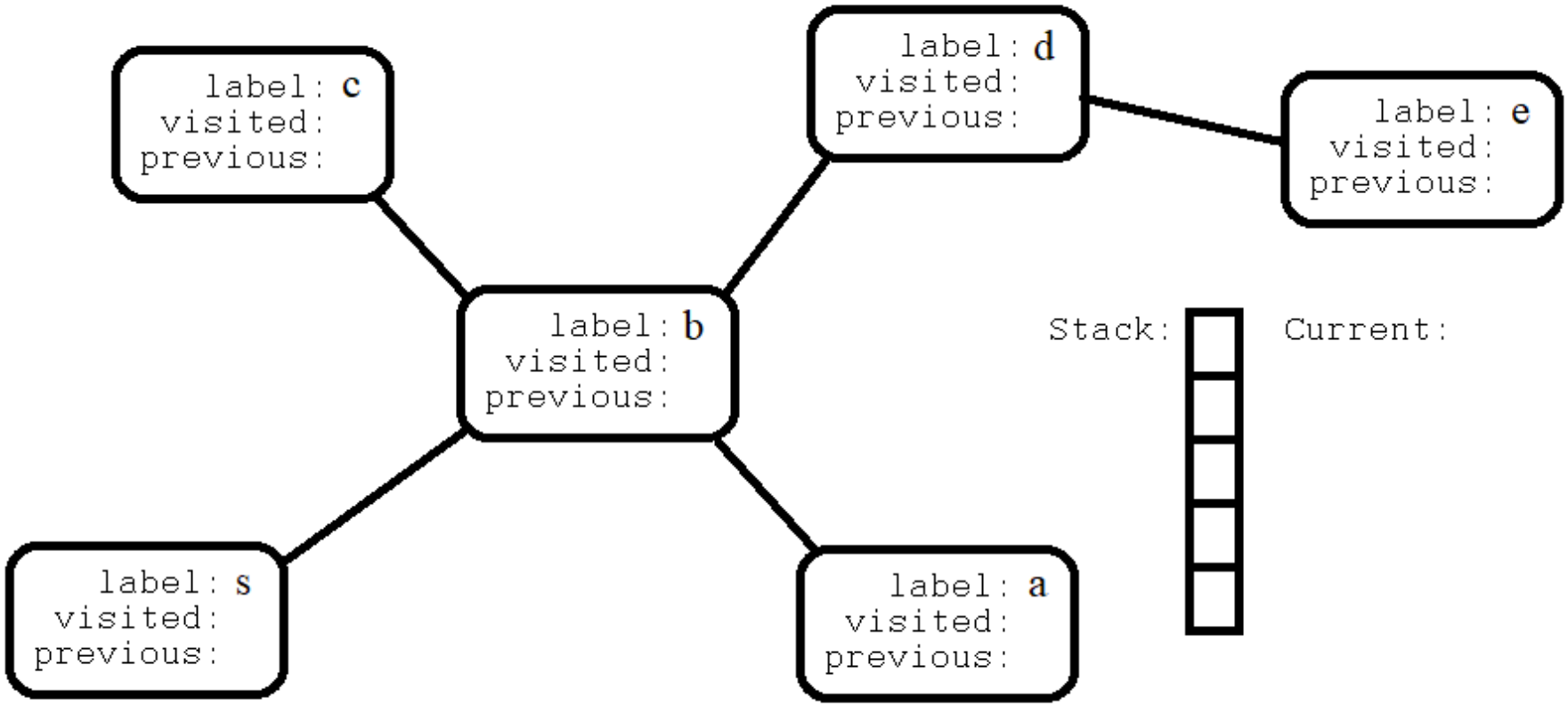
- Represent the maze as a graph! Each room is a vertex connected by edges to the neighboring rooms.
- The chalk marks correspond to a boolean value for each vertex: visited or unvisited.
- The string has two basic operations: unwind and rewind.
  - In what order are the rooms visited with the string?
  - When you go to rewind, the most recent room you visited will be the room you return to.
  - Last in, first out! The string is a stack.
  - Unwind = Pushing a new vertex onto the stack.
  - Rewind = Popping a vertex from the top of the stack.
- We can even retrace the path by recording, for each room we visit, which room we had just come from.

This method is called Depth First Search (DFS).

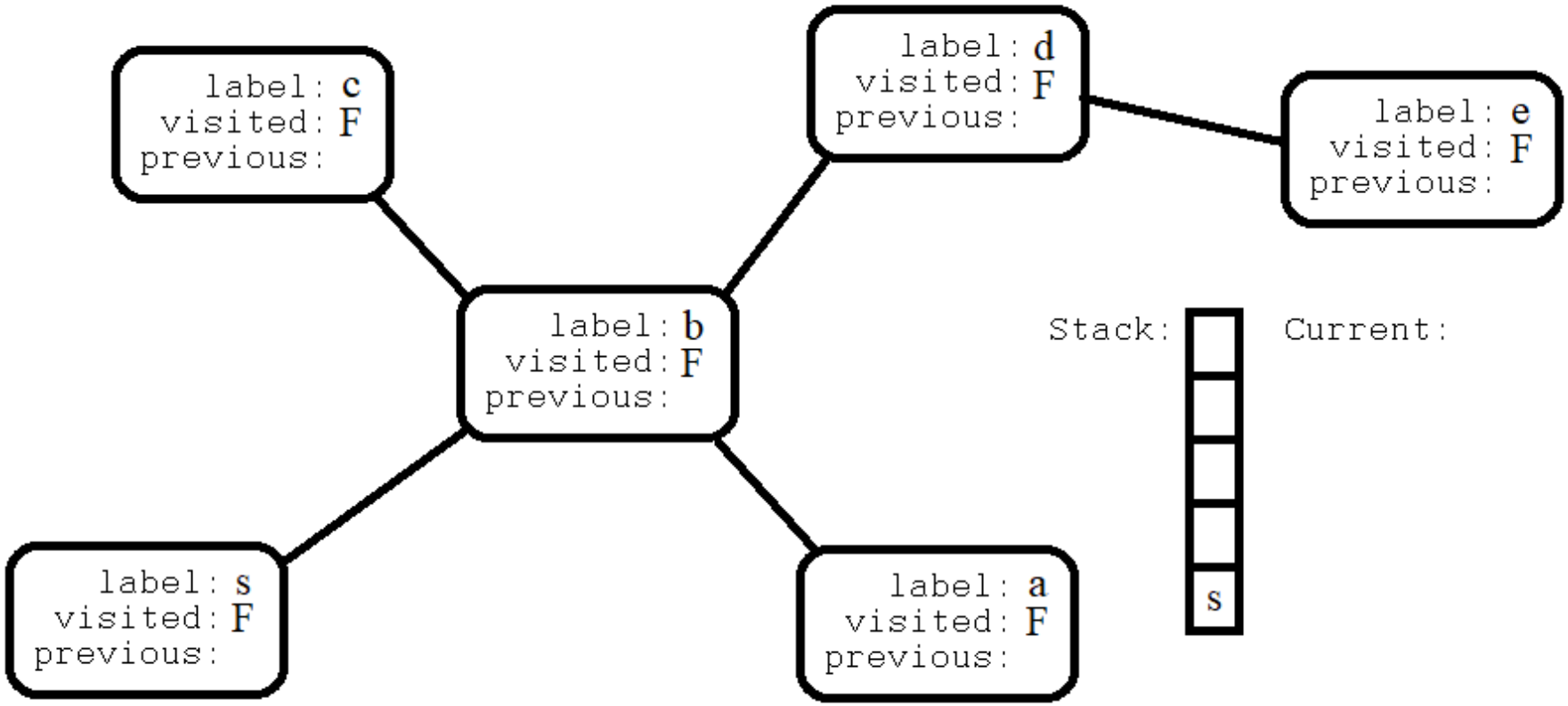
DFS Implementation

```
def DFS(graph, start):
    # No vertex visited yet.
    for vertex in graph:
        vertex.visited = False

    # Push the start vertex onto our stack.
    stack.push(start)

    while not stack.isEmpty():
        # Get the current room.
        current = stack.pop()

        # Mark as visited.
        current.visited = True

        # Push all neighbors onto the stack.
        # Then record where we came from.
        # If they have not already been visited.
        for neighbor of current:
            if neighbor.visited == False:
                stack.push(neighbor)
                neighbor.previous = current
```

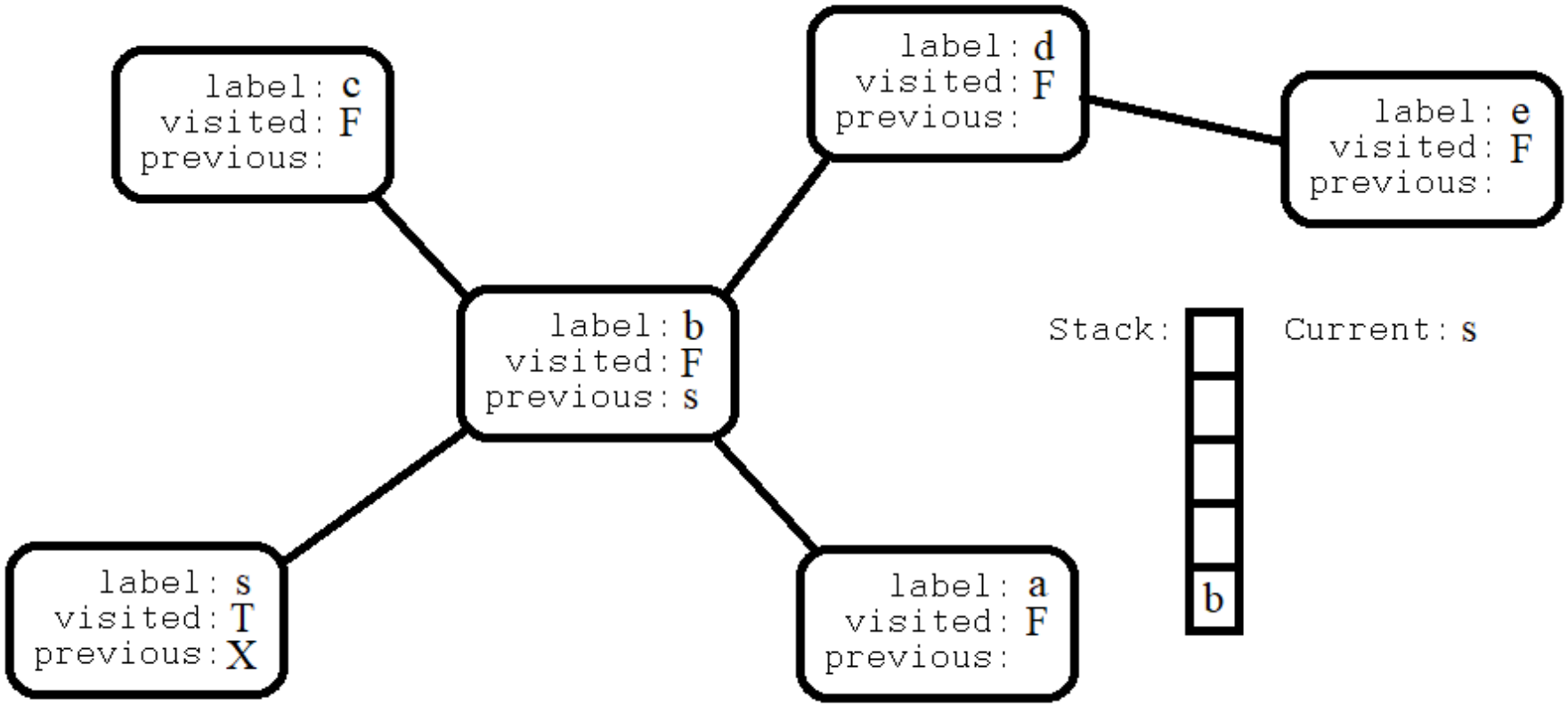
DFS - Worked Example



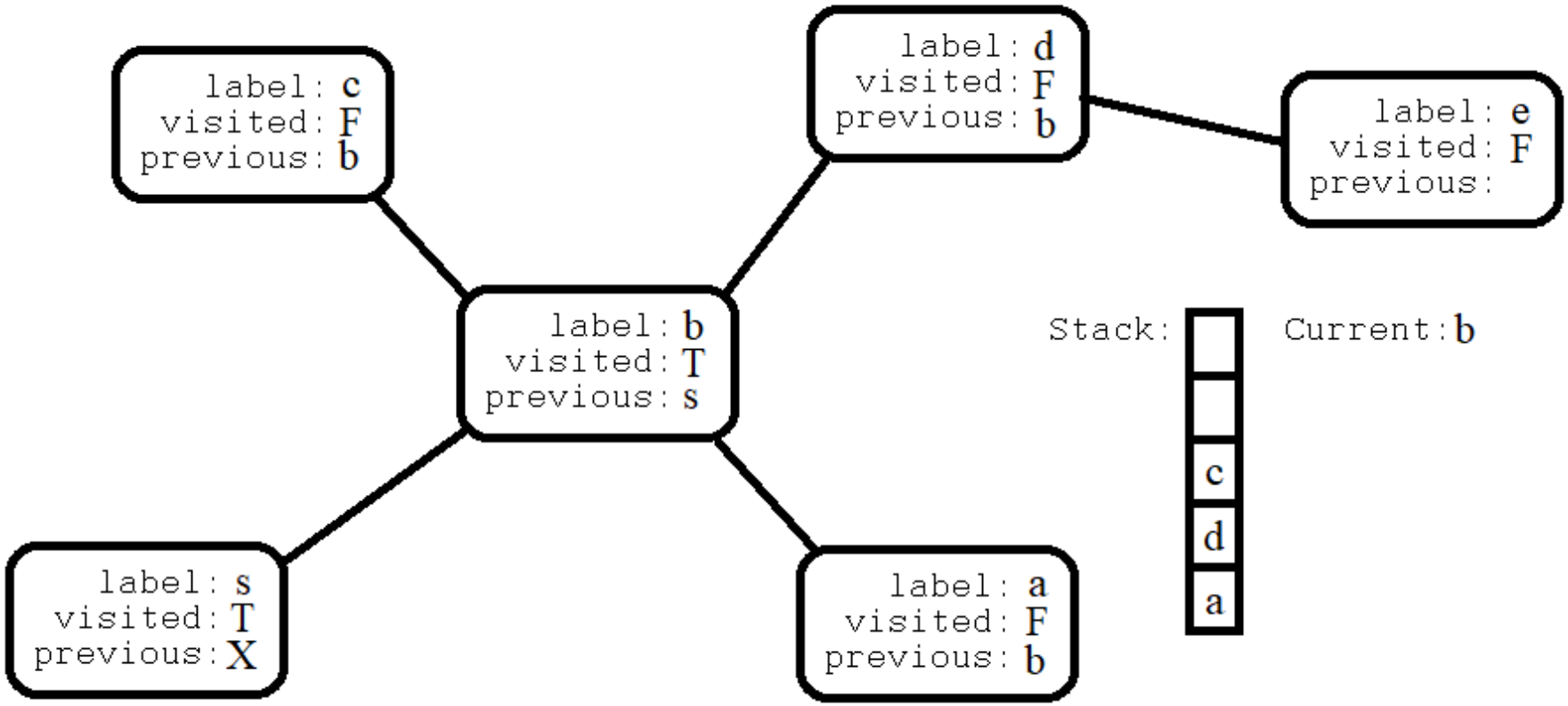
DFS - Worked Example



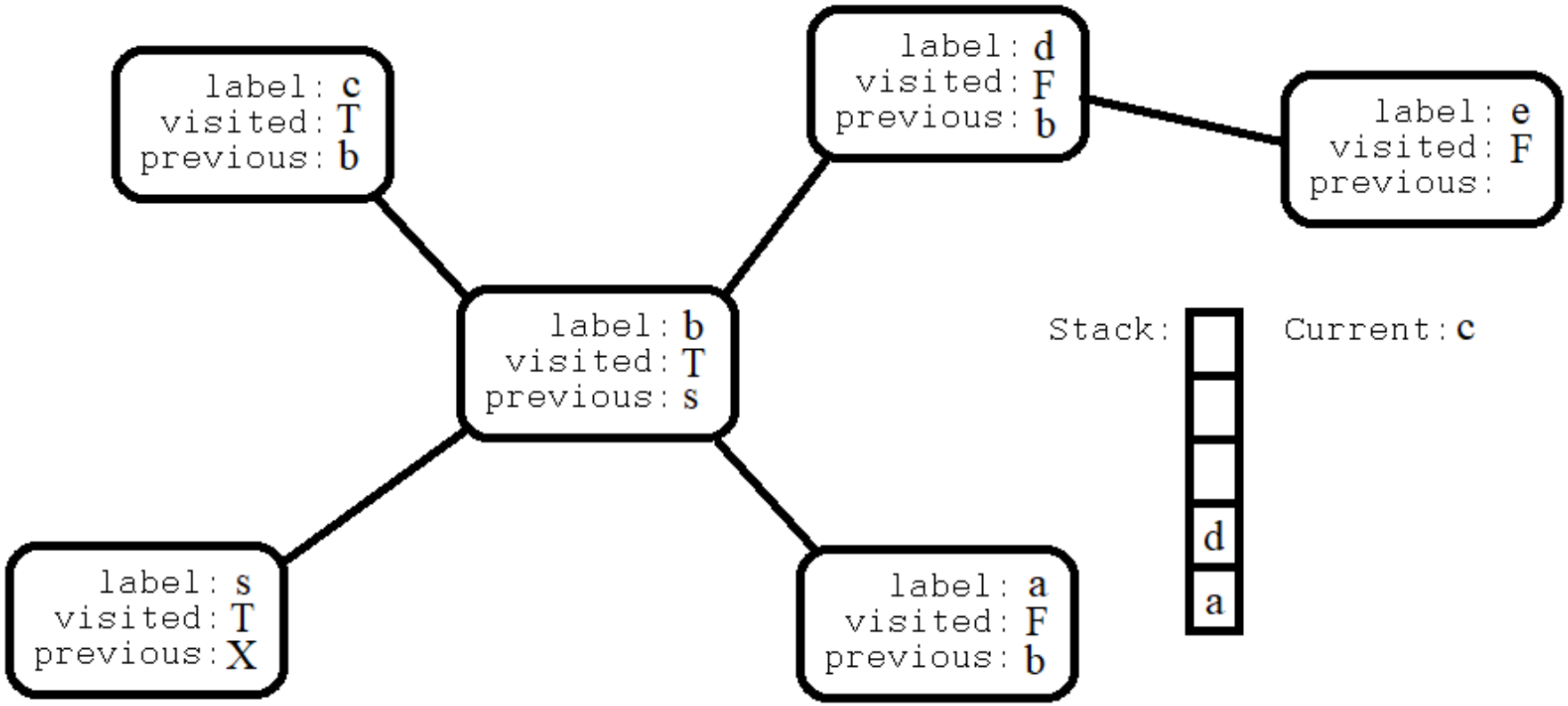
DFS - Worked Example



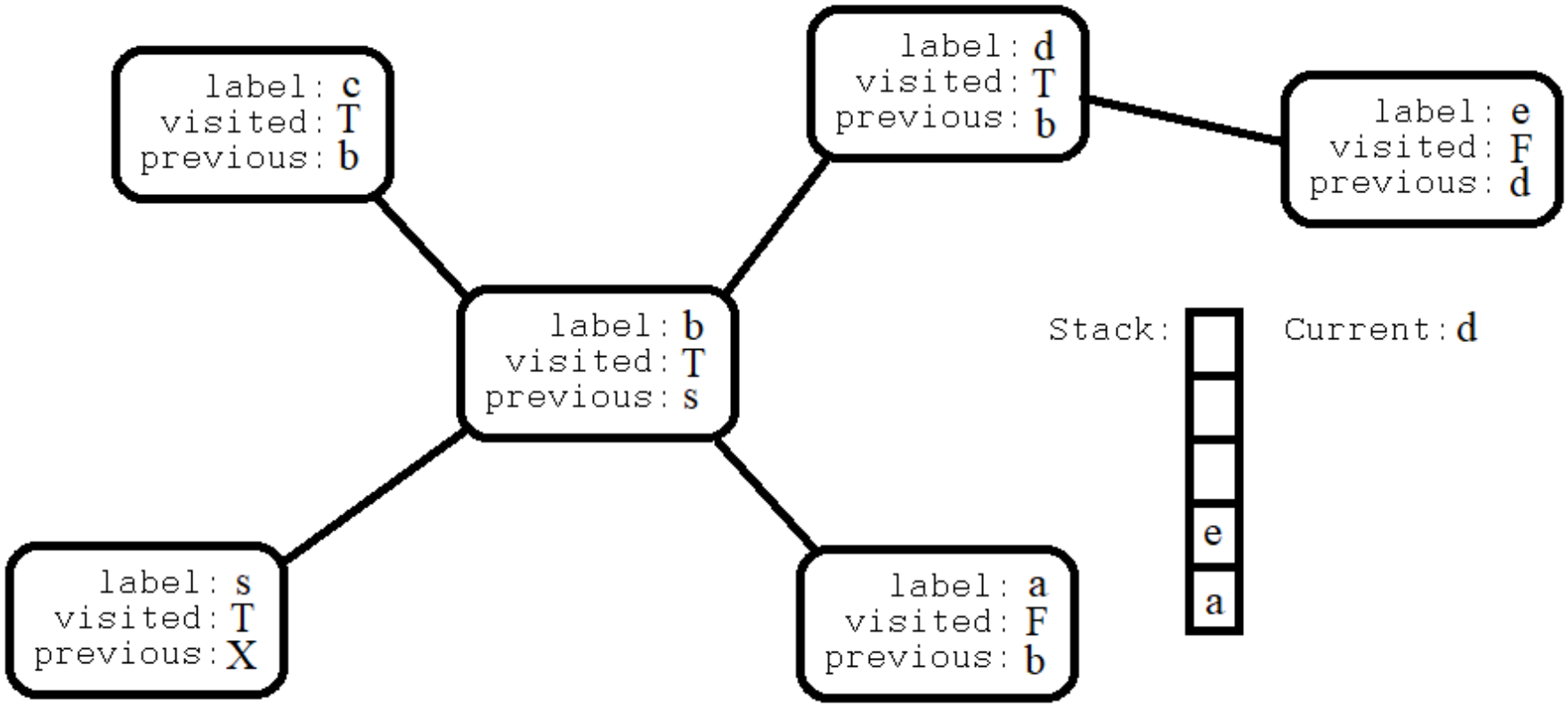
DFS - Worked Example



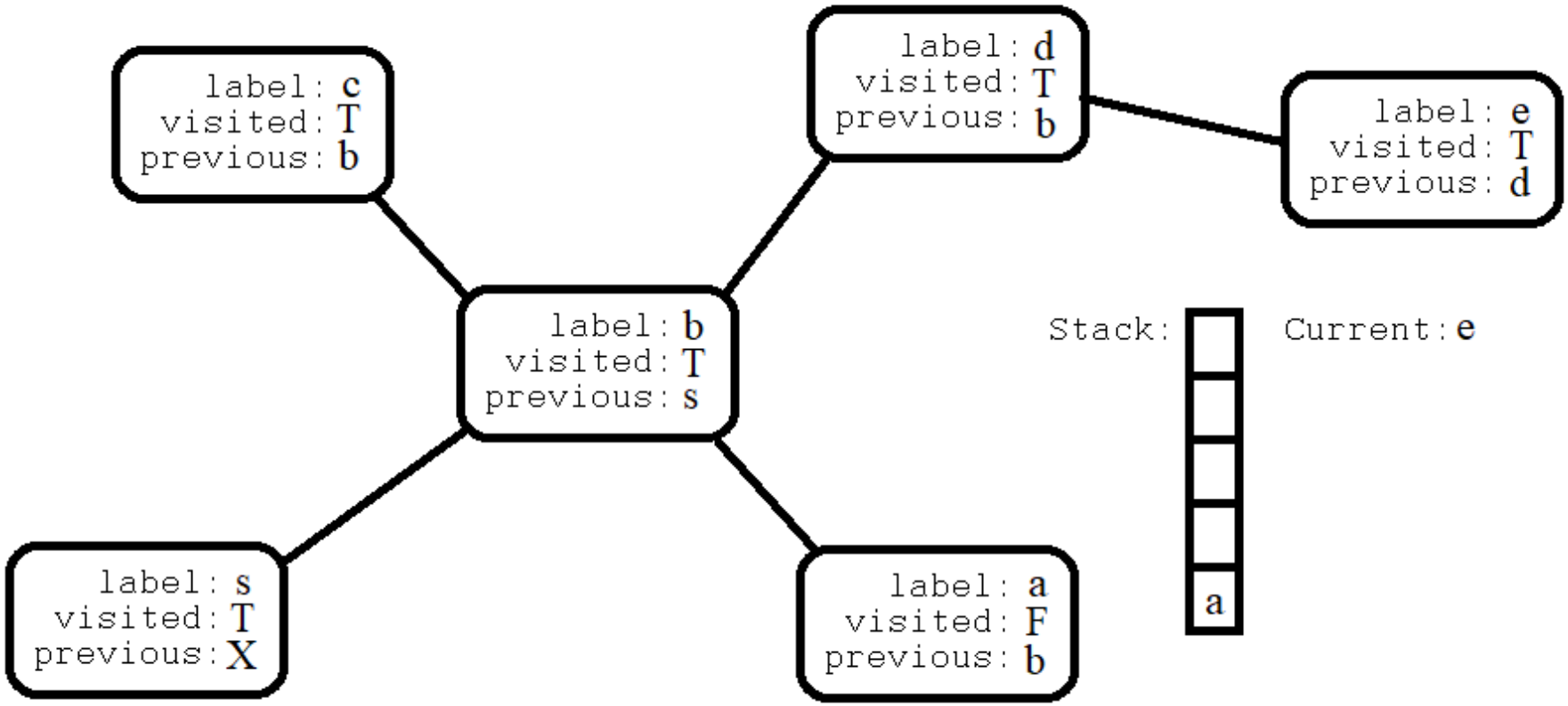
DFS - Worked Example



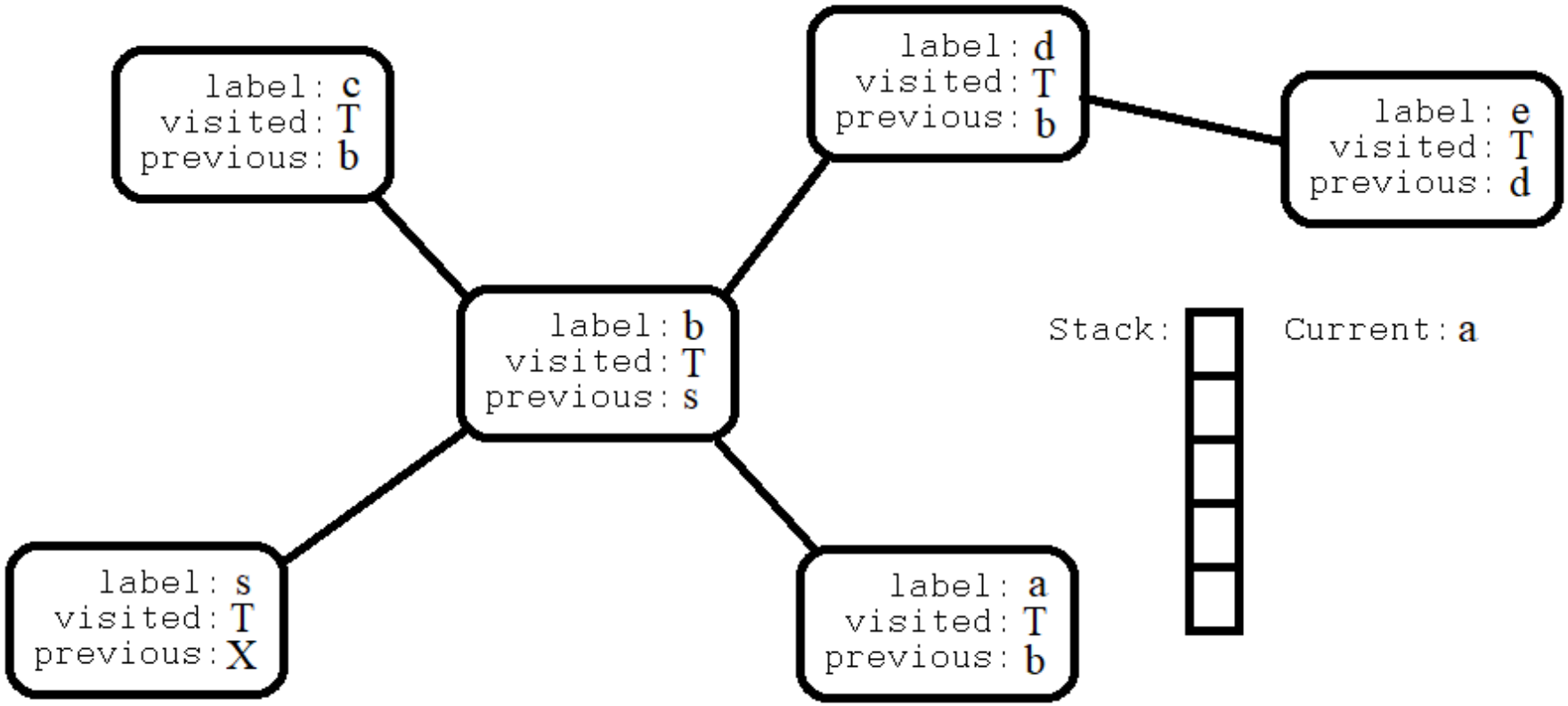
DFS - Worked Example



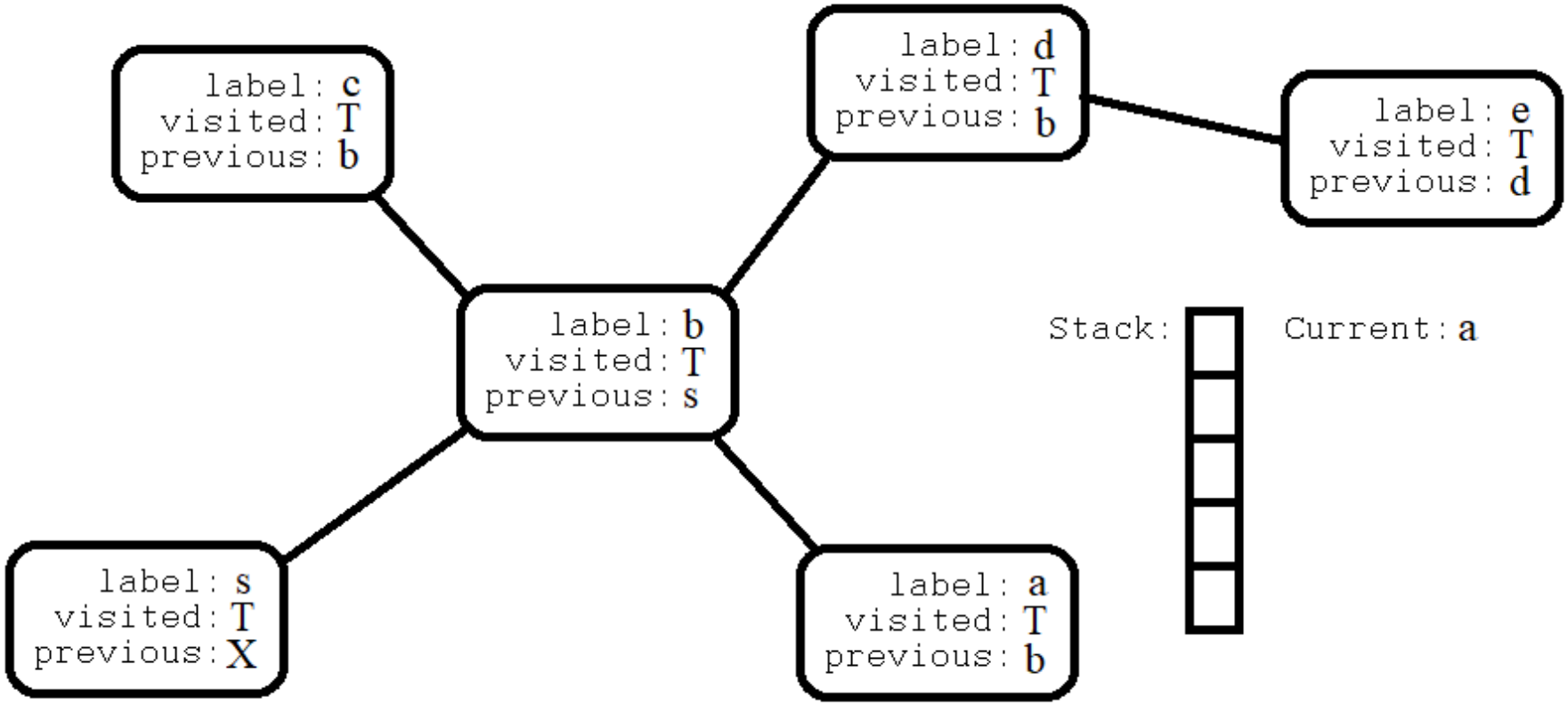
DFS - Worked Example



DFS - Worked Example



DFS - Worked Example



Trace path from s to e:

- e came from d
- d came from b
- b came from s

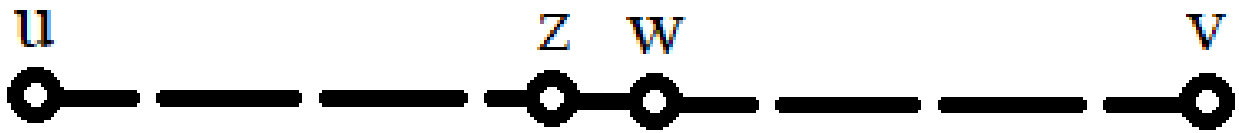
So path: $s \rightarrow b \rightarrow d \rightarrow e$

Proving Correctness of DFS

Can we prove that this algorithm always visits all reachable vertices? Say we start at vertex u.

Let's assume (by way of contradiction) that there is some reachable vertex v that our DFS algorithm does not visit.

That means there was a path from u to v, but our DFS algorithm did not follow the entire path, i.e., there is a vertex z on the path that was the furthest one visited by DFS.



Now consider the next vertex on the path after vertex z: call it vertex w.

Then z was visited by DFS while w was not. But w is a neighbor of z and would have been pushed onto the stack and eventually visited!

DFS Runtime

What work is done by DFS?

- Some constant amount of work required to pop the new vertex and mark it as visited.
  - This is constant work per vertex, so $O(|V|)$ where $|V|$ is the number of vertices.
- A loop that looks at each neighboring vertex to see if it leads somewhere new.
  - Looking at each neighbor of a vertex is equivalent to checking each edge!
  - This is constant work per edge, so $O(|E|)$ where $|E|$ is the number of edges.

So the overall runtime is $O(|V| + |E|)$.

Note that if there are n vertices, then there can be at most $O(n^2)$ edges. So the worst case search is $O(n^2)$.

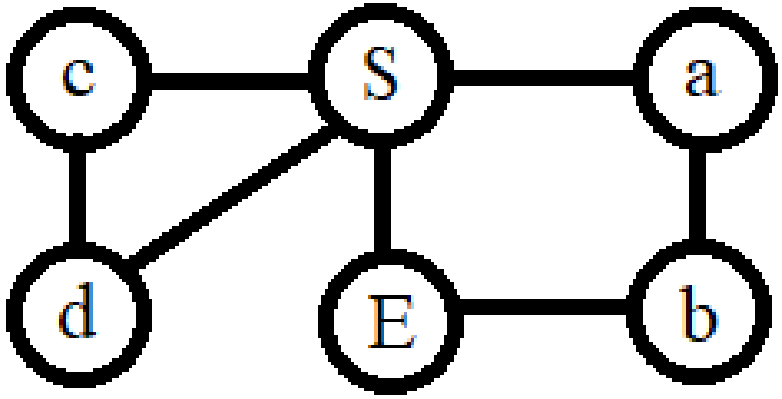
More DFS

It turns out that there are a lot of things we can do with DFS beyond just solving a maze.

- DFS can be used to detect connected components of a graph.
- DFS can be used to label different connected components of a graph (by rerunning it on a new, unvisited component of the graph when it ends computation).
- We can perform DFS in a directed graph, and can use it to detect cycles.
- DFS is used in internet crawling. We may not know what the overall graph is, but we can crawl through it using DFS to ensure that we visit all possible sites.

Shortest Path

DFS always finds us a path out of the maze, but it does not always find the shortest path out of the maze:



If we start at vertex S and want to get to vertex E, the shortest path is to go straight there.

But what if vertex a was pushed onto the stack before vertex E when we were looping through the neighbors of S?

Then we will visit vertex a, followed by vertex b, followed by vertex E.

We find a path to vertex E, but it is not the shortest path.

Breadth First Search (BFS)

Notice that a computer is not limited like a person would be walking around a maze. The computer can jump from vertex to vertex.

Idea: look at all vertices distance 1 away from the start, then all vertices distance 2 away, then all vertices distance 3 away, etc.

DFS was last in, first out (unwind/rewind).

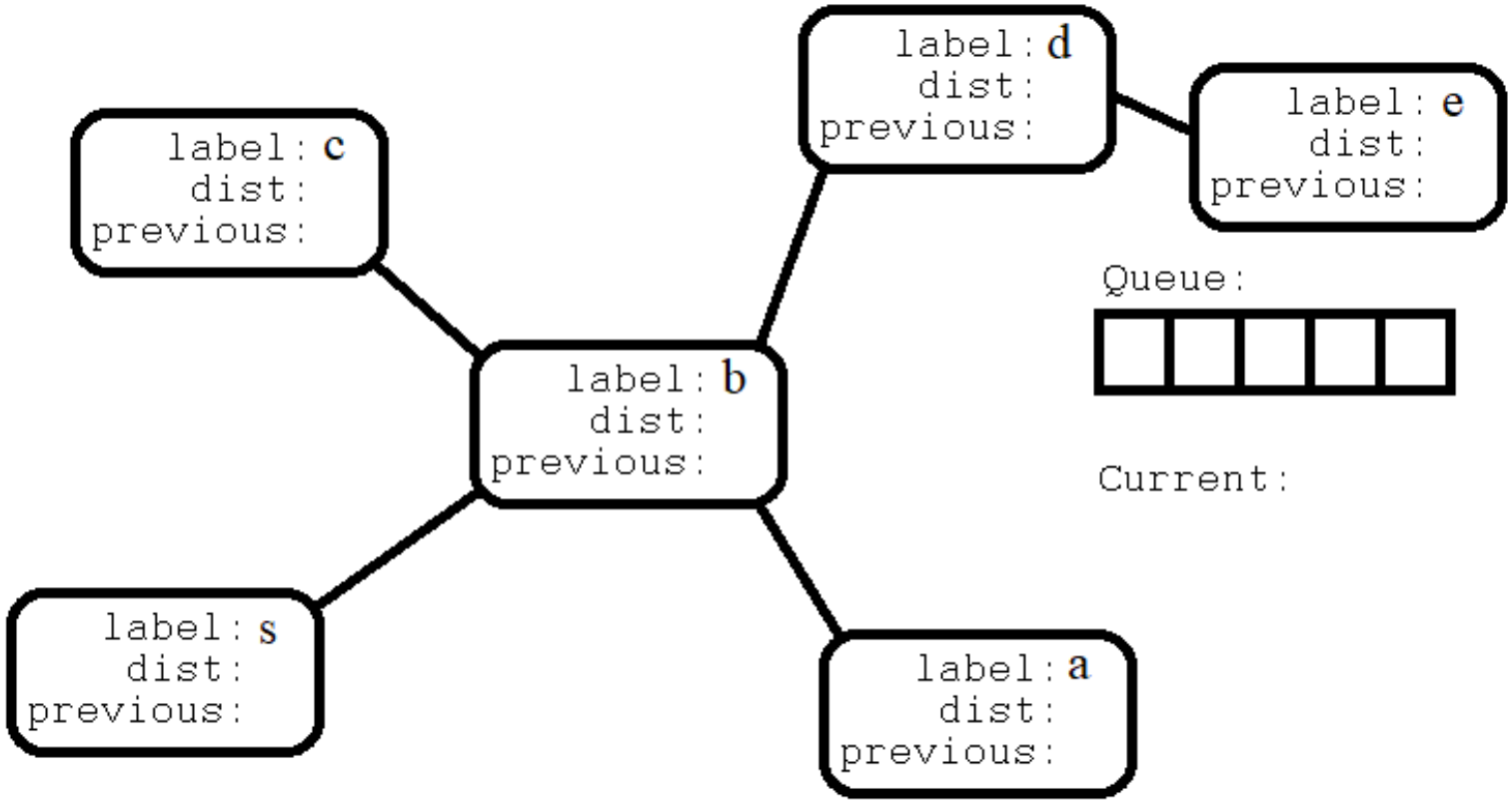
BFS is first in, first out (since consider all vertices 1 away before moving on).

We use a queue instead of a stack!

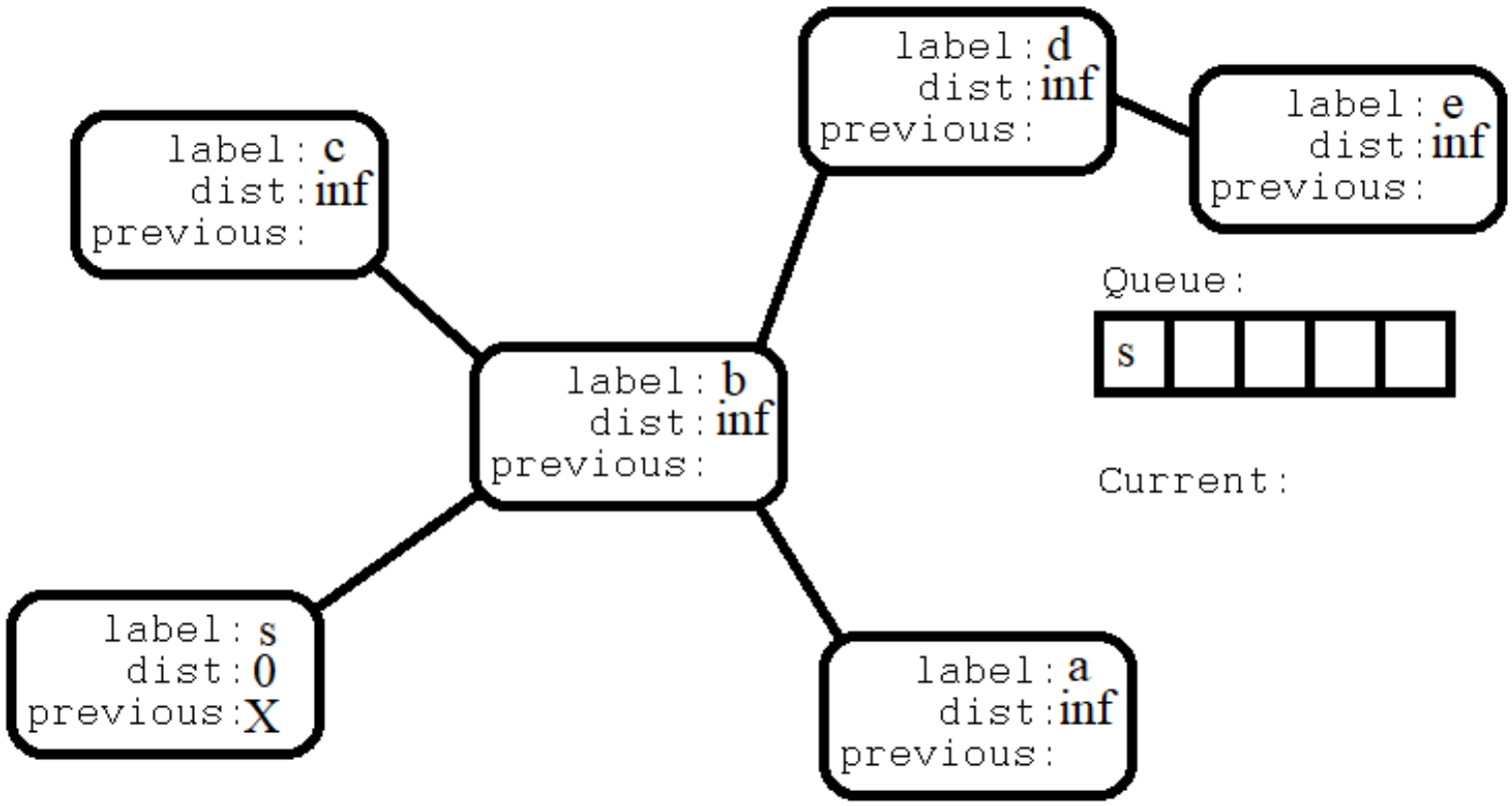
BFS Implementation

```
def BFS(graph, start):
    # Set all vertices to an infinite distance.
    for vertex in graph:
        vertex.dist = ∞ # math.inf in python 3

    # Push the start vertex into the queue and set dist = 0.
    queue.push(start)
    start.dist = 0

    while not queue.isEmpty():
        current = queue.pop() # Get the current vertex.

        # Look at all of its neighbors.
        for neighbor of current:
            # If the neighbor's dist not updated.
            if neighbor.dist == ∞:
                # Push the neighbor into the queue.
                queue.push(neighbor)

                # Update its distance and track path.
                neighbor.dist = current.dist + 1
                neighbor.previous = current
```

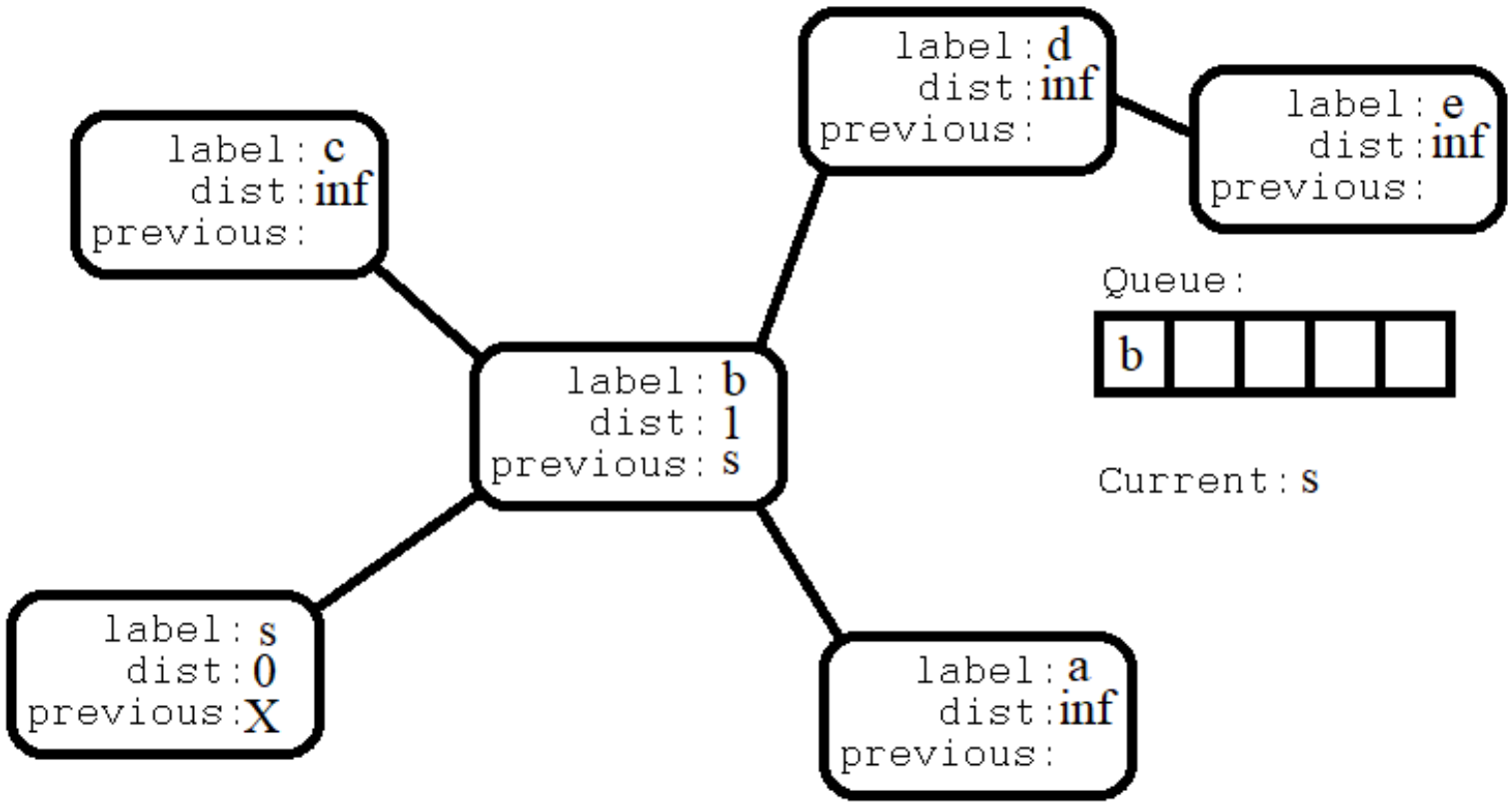
BFS - Worked Example



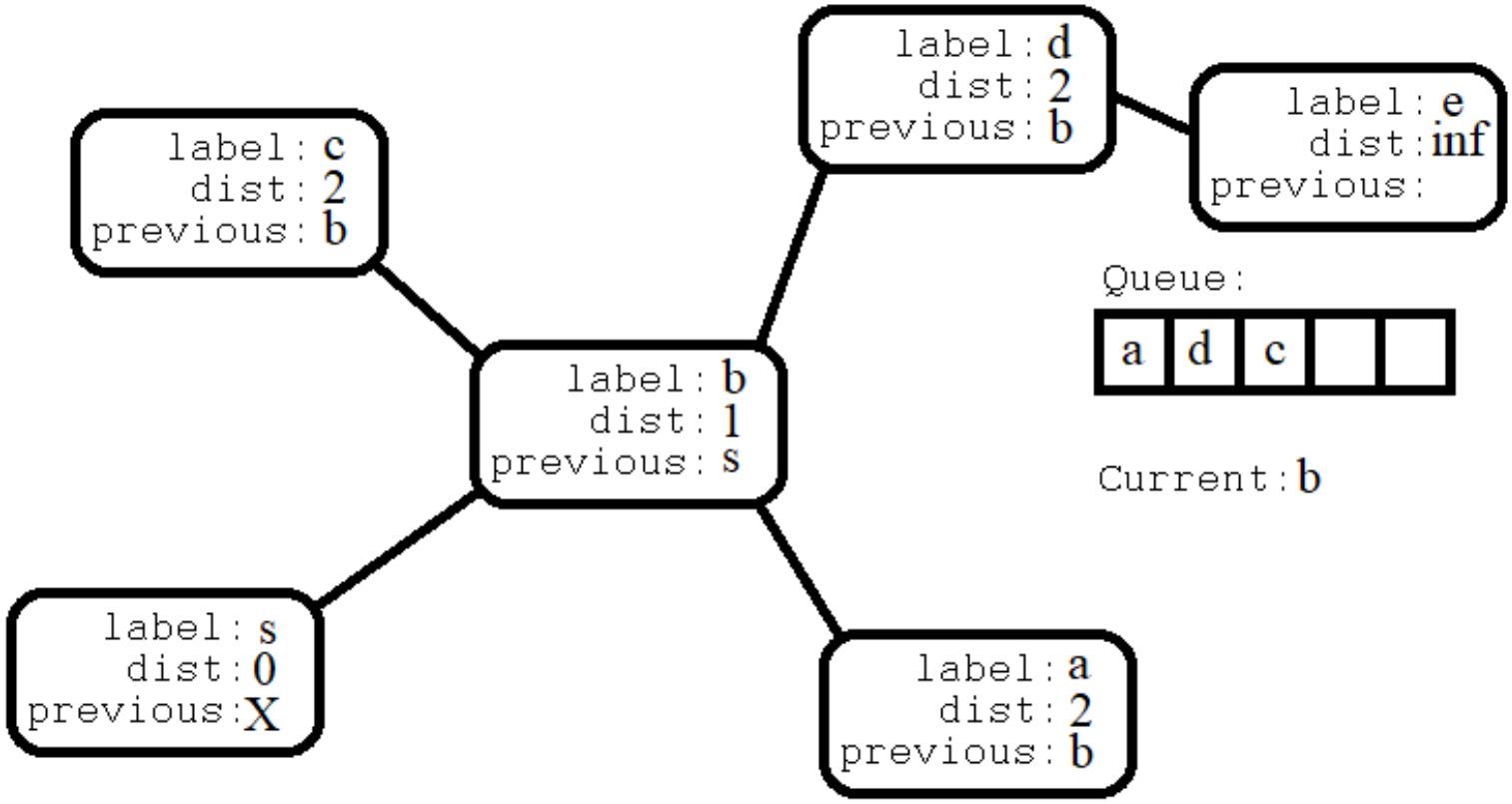
BFS - Worked Example



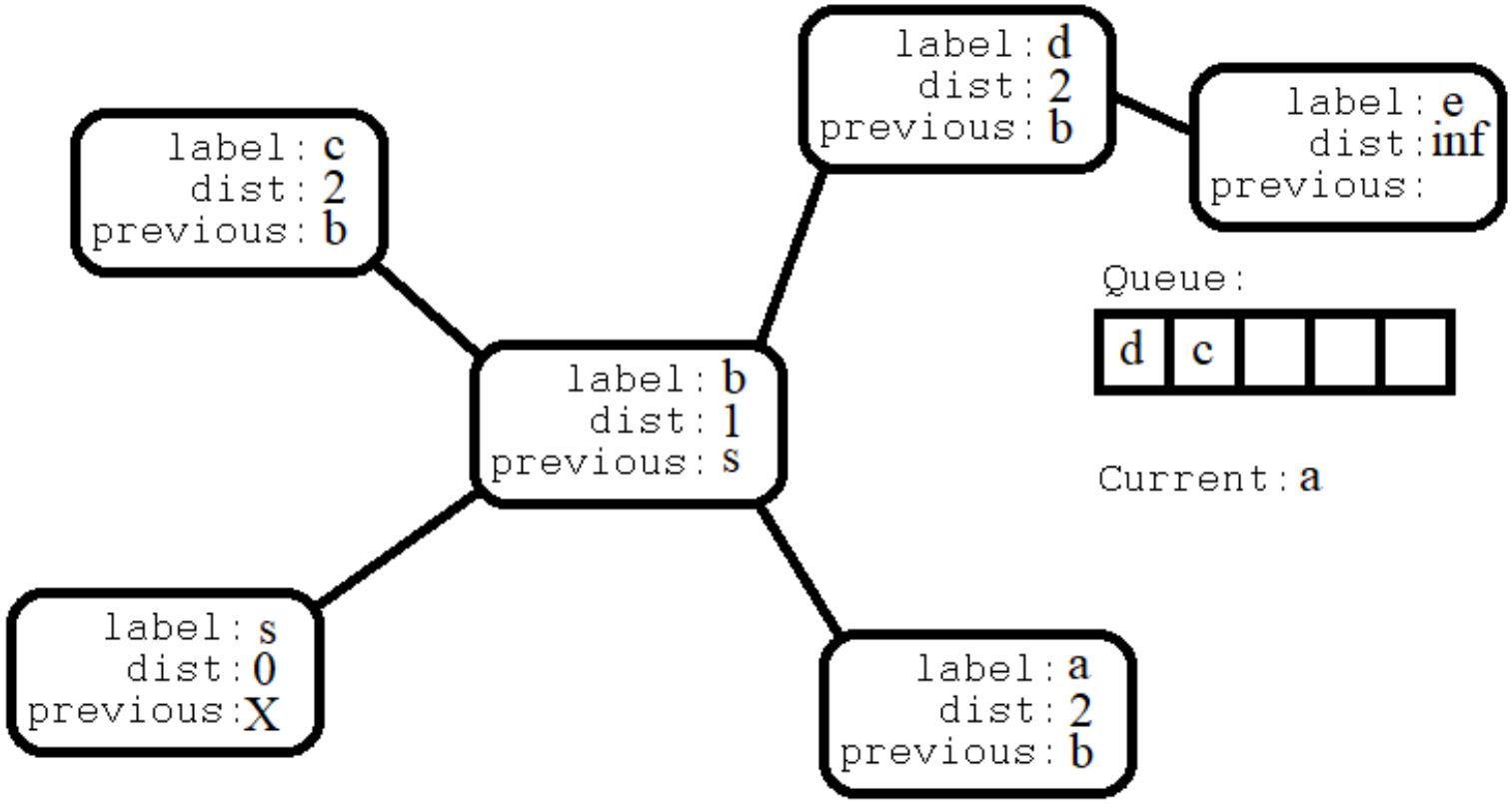
BFS - Worked Example



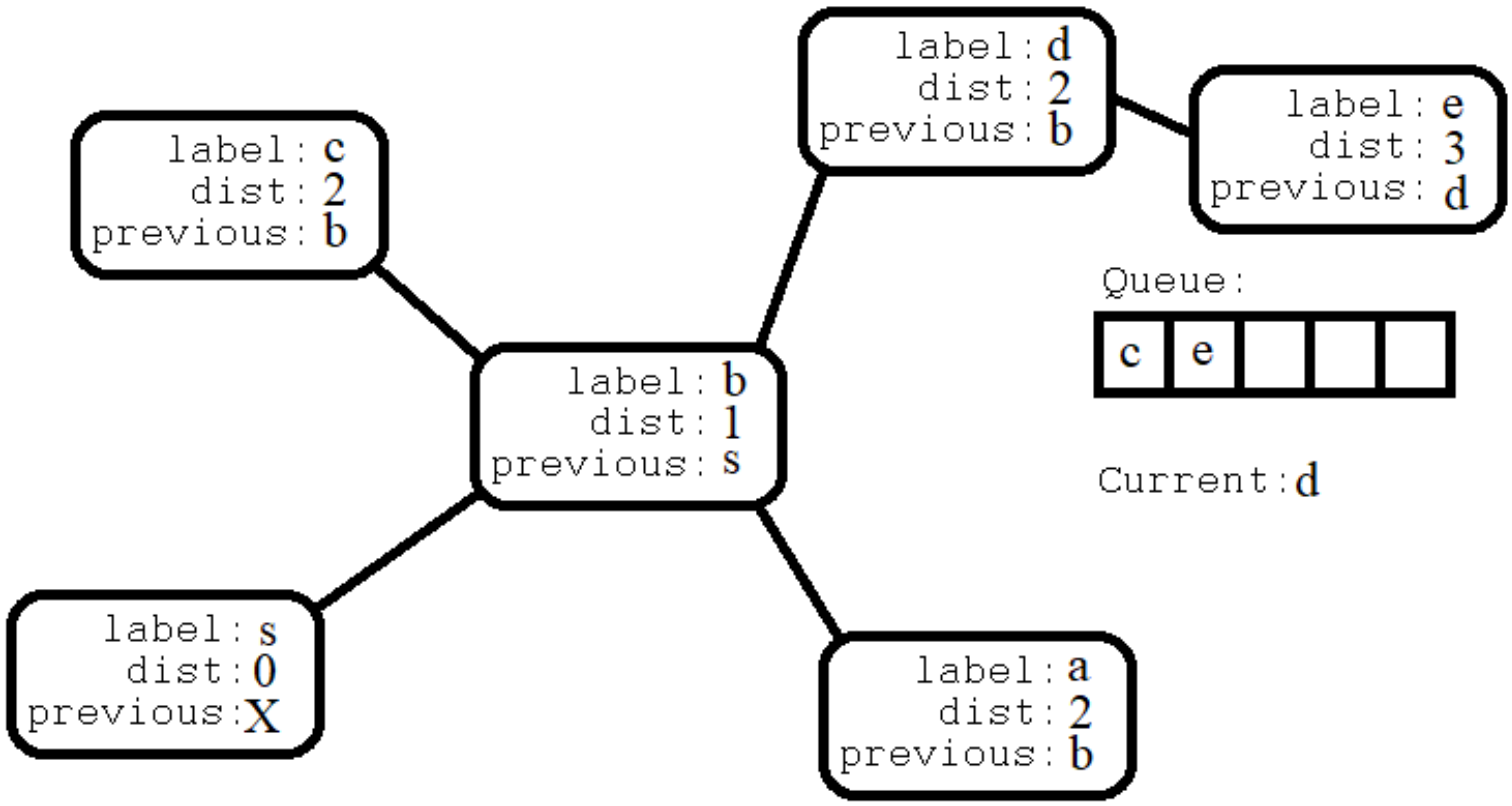
BFS - Worked Example



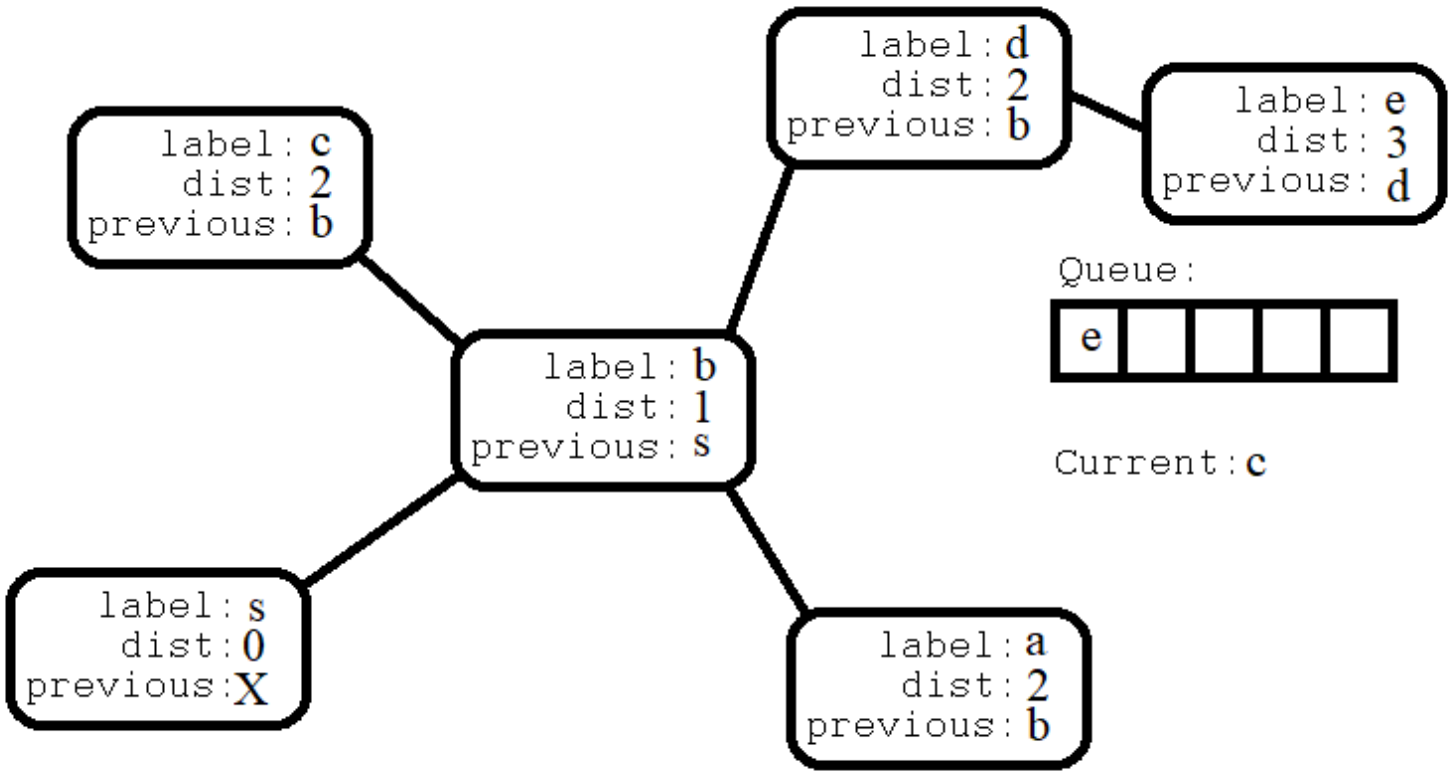
BFS - Worked Example



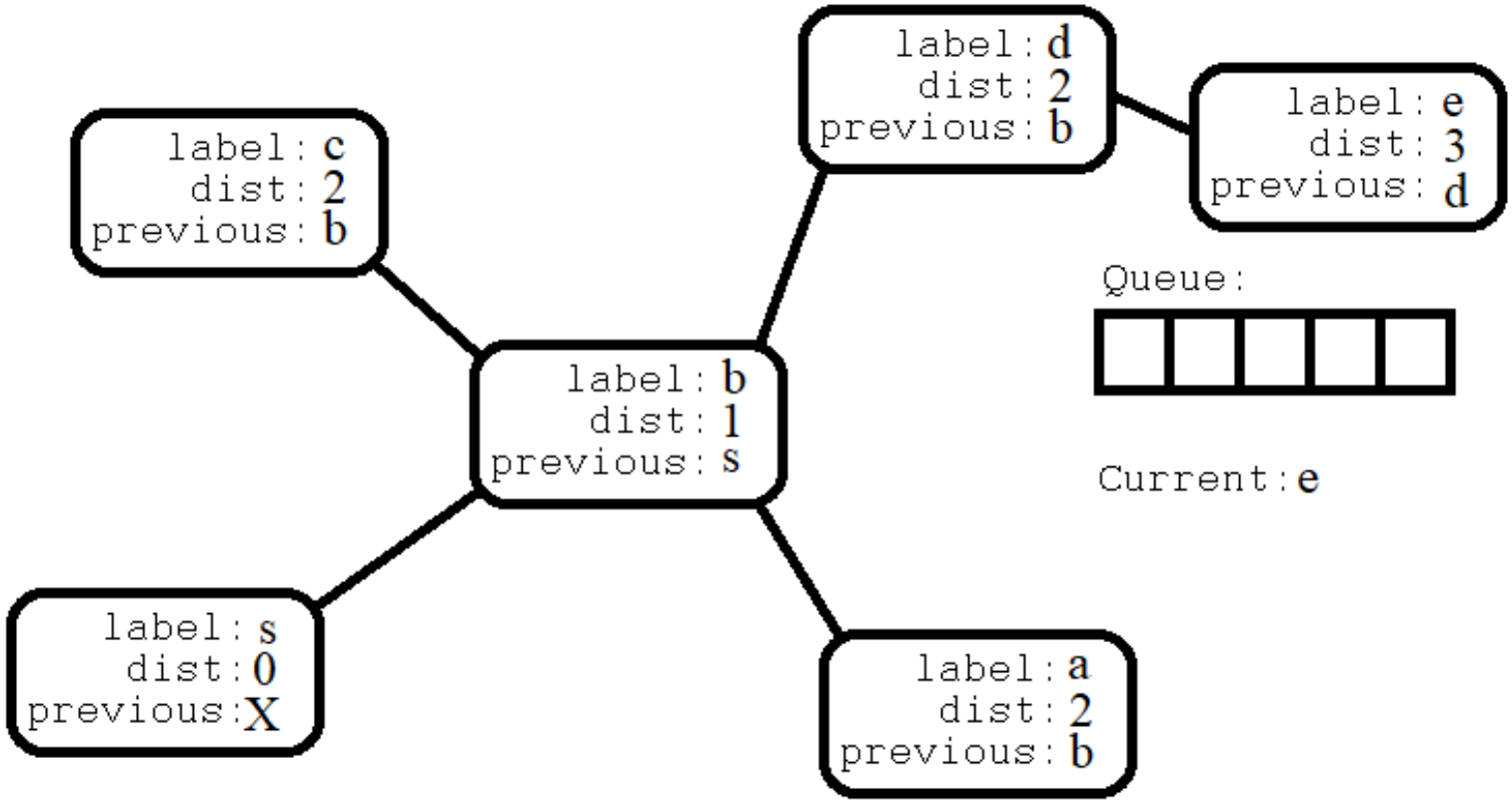
BFS - Worked Example



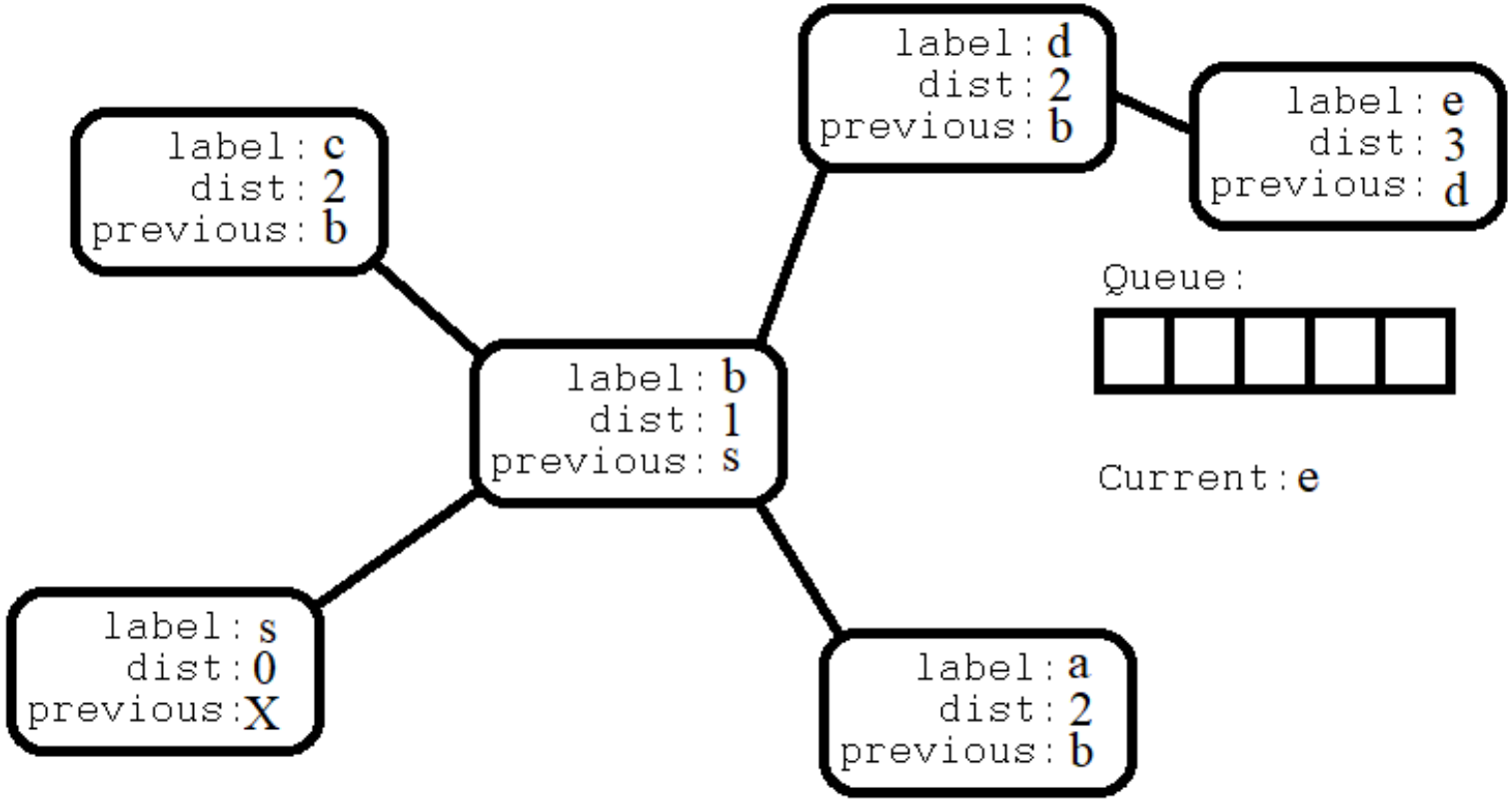
BFS - Worked Example



BFS - Worked Example



BFS - Worked Example



Trace the shortest path of length 3 from s to e:

- e came from d
- d came from b
- b came from s

So path: $s \to b \to d \to e$

BFS Proof of Correctness

(Quick) Proof by Induction:

Base Case: the start vertex has distance 0.

Inductive Hypothesis: Assume that at step k, all vertices of distance k or less have the correct distance, while all other vertices have a distance of ∞.

Inductive Step: Consider step $k+1$. All of the vertices of distance k are popped from the queue and their unvisited neighbors' distances are updated by adding 1. So clearly all vertices of distance k or less were untouched (and still correct). And, all vertices of distance $k+1$ are correctly updated and pushed into the queue.

BFS Runtime

What is the runtime of BFS?

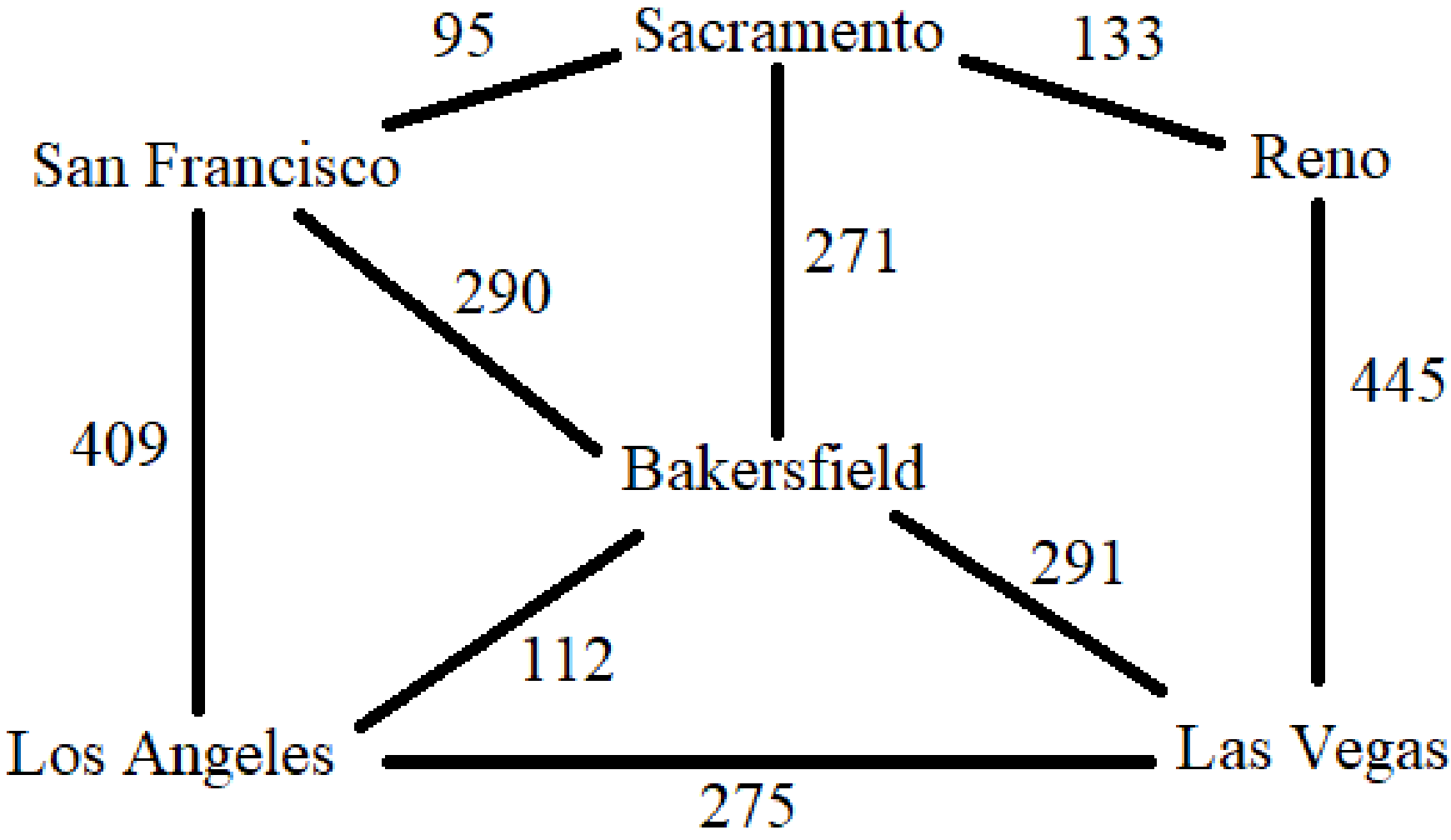
It is the same as DFS!

We do constant work visiting each vertex, and constant work looking at each neighbor (looking at each edge).

$O(|V| + |E|)$

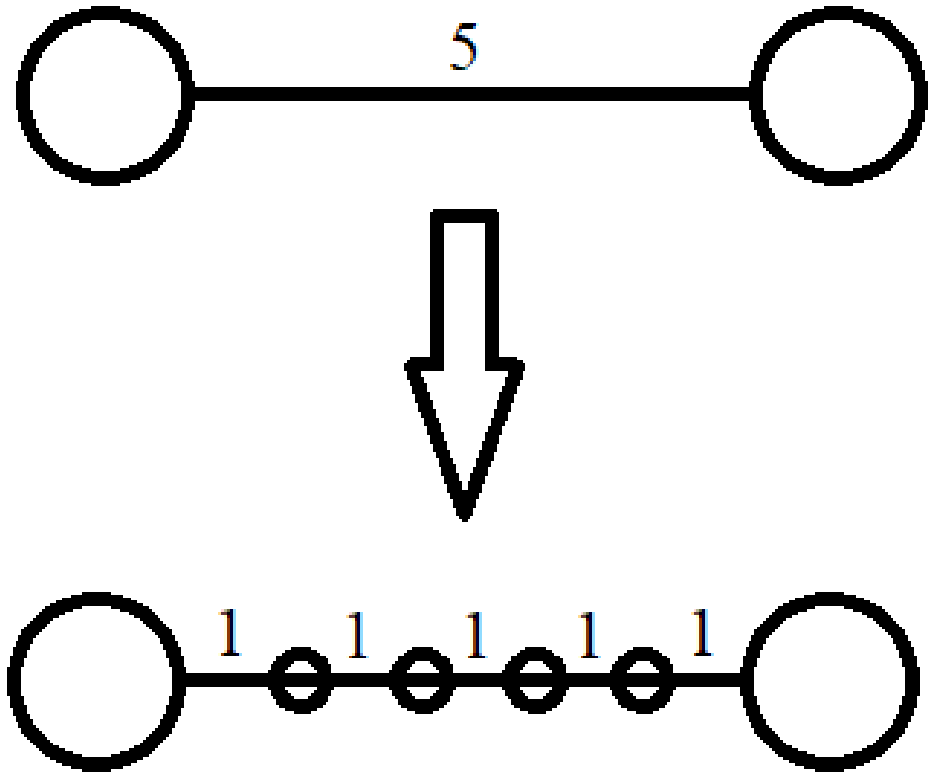
Weighted Graphs

What should we do with a weighted graph, if distances between vertices are not 1?



BFS in a Weighted Graph

Idea #1: an edge weighted as 5 is the same as a twig that is 5 edges long, so introduce dummy edges/vertices in an extended graph and run regular BFS!



Problem: introducing a lot of dummy edges/vertices slows down runtime (what if an edge had a very large weight?)

Weighted Graphs

Idea #2: use a stopwatch.

We will track when the BFS algorithm would have reached each vertex in the extended graph.

This gives us Dijkstra's algorithm (next time).