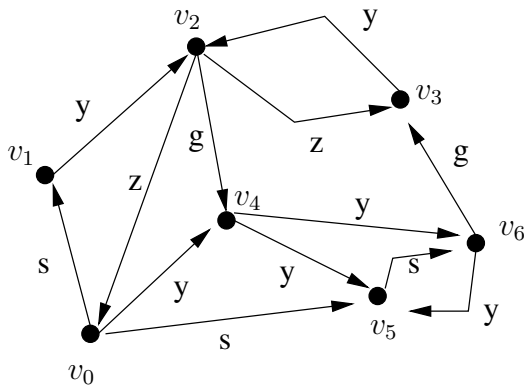
Notes on CLRS Problem 15-7 (Viterbi algorithm)

There are several things that I think are confusing in the presentation of this problem. First, they start the problem by mentioning the application to speech recognition. While it is great to see the context/usefulness of an algorithm, I think that is a distractor at this point. In fact, besides speech recognition, the Viterbi algorithm has a wider range of applications in computational linguistics and machine learning. (The general use of the Viterbi algorithm is, given a hidden Markov model and a sequence of observations, what is the most likely sequence of states the model would have passed through to produce those observations? But you don't need to know anything about that.)

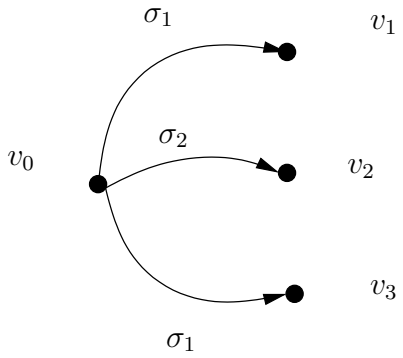
Instead, I suggest you first think of the problem just in terms of graphs. In part (a), your input is a graph $G = (V, E)$, but assume that every edge is labeled; $\Sigma = \{\sigma_a, \sigma_b \ldots\}$ is the set of labels. You're also given a sequence of labels drawn from Σ, call is $s = \langle \sigma_1, \sigma_2, \ldots \sigma_k \rangle$. You're looking for a path from a given starting point v_0 with that sequence of labels on its edges. Suppose we are given the following graph with labels $\Sigma = \{g, s, y, z\}$:



If the label sequence is *syzygy*, then a path starting at v_0 that has that label sequence is $v_0, v_1, v_2, v_3, v_2, v_4, v_5$ (although it is not the *only* such path; $v_0, v_1, v_2, v_3, v_2, v_4, v_6$ would also work).

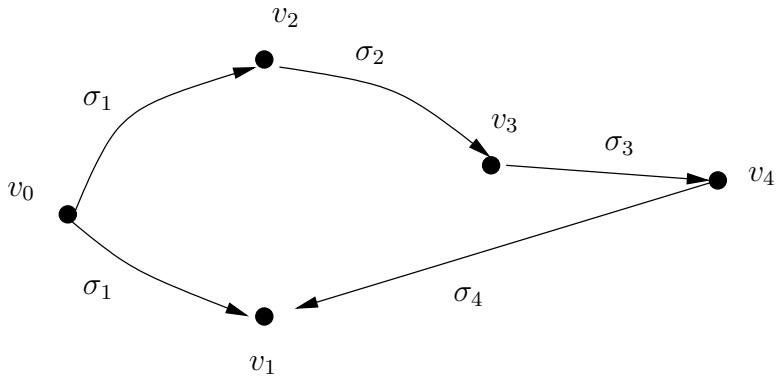
In part (b), you need to assume that each edge has not only a label but also a *probability*; you want to find the path that is most likely, that is, has the greatest probability. (You actually don't need to think of this value as a probability to do this problem. . . just think of them as some positive numeric value, and you want to maximize the product of all such values on the path.)

Even after we understand the big picture of the problem, it still seems like various assumptions are omitted from CLRS's presentation of problem—assumptions that lead to subtle gotchas in what would be intuitive solutions and great difficulties in the analysis. For example, is the graph simple? It seems plausible that there may well be parallel edges between some v and u that have different labels. We certainly should not assume that a vertex has at most one outgoing edge for a given σ, that is, we certainly could have



Now, the problem is phrased as "find an efficient algorithm," which is code for "use dynamic programming." However, it is worthwhile to consider, as scratch work, how one would modify BFS and

DFS to do this. Here is the difficulty with BFS. Suppose our sequence is $s = \langle \sigma_1, \sigma_2, \sigma_3, \sigma_4 \rangle$ and our graph includes



So we can't just associate each vertex with a "parent" to retrace our steps. Instead, let's try DFS. Assume that our input is the *current* list r (of edge labels) and the *current* vertex v (this is a recursive algorithm, and the first call to it will have v_0 and s as its input). It returns a list of vertices (the path found) or NSP (no such path) (the return type is a union type of sorts).

(I'm using a pseudo-ML here since we're using lists a lot.)

```
Find_Seq(v, r):
    if r = NIL
        return [v]
    else
        σ = hd(r)]]
    for all u ∈ v.adj
        if (v, u) has label σ
            t = Find_Seq(u, tl(r))
            if t ≠ NSP
                return cons(v, t)
    return NSP
```

Analysis: the worst case (as far as I can imagine it) is a super-complete graph where every vertex pair having an edge for every sound in Σ except that only one pair (say (x, y) has σ_k. Any $k-1$-length paths will work as a prefix, but how many $k-1$-length false paths can it have? Well, it's basically (or, slightly less than) the number of permutations of of size k of the vertices:

$$\frac{|V|!}{(|V|-k)!} = O(V!)$$

. . . and that assumes there are no self-loops or backedges! Ok, let's see if BFS makes more sense. We will keep a worklist of paths (as lists). Note this worklist doesn't need to be a queue;

```
Find_Seq(v_0, s):
    w = [[v_0]]        //initial worklist
    for σ ∈ s
        if w is empty, return NSP
        w' = []        //the next worklist
        for p ∈ w
            v = hd(p)
            for u ∈ v.adj
                if (v, u) has label σ
```

$$w' = \text{cons}(\text{cons}(u, \text{clone}(p)), w')$$
$$w = w'$$
return $\text{hd}(w)$

Even an incomplete analysis (of an insane worst case) should be enough to scare people off. How many iterations will there be for the second loop? Well, the first iteration of the outermost loop will have only one iteration of the second loop; in the worst case, v_0 may have σ_1-labelled edges to every other vertex, so on the second iteration of the outermost loop, the second loop may have $|V| - 1$ iterations. Actually, let's not assume there are no self-loops: make it $|V|$ iterations. If each of those have σ_2-labeled edges to every other one, then the next iteration of the outermost loop will have $|V|^2$ iterations of the second loop. This gives us

$$\sum_{i=0}^{k-1} |V|^i$$

... and let's not forget that this same worst-case would mean that the innermost loop will have $|V|$ iterations every time. We have $O(V^{k+1})$.

I would note that for each of these, if we had a reasonably sparse graph, they shouldn't be such bad algorithms, and that certain assumptions might allow us to optimize these algorithms further.

Finally, here is how to set up a dynamic programming approach, which I believe is the right answer, or at least a right answer. Remember, we want to find a path, so subproblems will involve subpaths. We can break paths down by at-which-vertex-are-we-after-so-many-steps questions. Also, assume we can refer to vertices by number.

Let $M_{i,j}$ be a vertex that precedes v_j on a path of length i from v_0 to v_j, if any, where the edges in that path are labeled with the first i labels in the input sequence. Note that there might be more than one vertex that would be a correct value for $M_{i,j}$; in part (a), any such vertex will do. If there is no such vertex, you can store something like `None` (to use pseudo-Python).

Once you have populated the table for M, you can reconstruct the path by picking any vertex j for which $M_{k,j}$ is not `None` and using the values in M to work backwards to v_0.

(Hint: In my solution I assumed there was a Set data structure available that had an `any()` operation that returned an arbitrary element of the set.)

Part (b) has the following difference: We also need the concept (and corresponding table) of $P_{i,j}$, the probability of the most probable path from v_0 to v_j, and $M_{i,j}$ is *the* previous vertex on *that most probable path.*