2 Updating Labels

Solution. *At every step of the algorithm, we will maintain the labels of the ancestors of the current node in a separate array. To ensure that our array only contains vertices on our current path down the DFS tree, we'll only add a vertex to our array (at index equal to the current depth) when we've actually visited it once. Since a path can have at most n vertices, the length of this array is at most n. Once we've processed all the children of a node, we can index into the array and set its label equal to the index of its k-th ancestor. Notice that if we relabel the vertex before processing its children, we overwrite a label that the children of the vertex could depend on.*

3 Where's the Graph?

(a) We can view this as a BFS problem, where the nodes in the graph are numbers that have been calculated. There are at most five edges from each node: $+1$, 1, $+y$, y, and $/y$. Our algorithm starts from x and run BFS on this graph until the node 2022 is reached.

(b) This is a directed graph, with each node representing some species and an edge from x to y indicating that y descended from x. We run DFS on this graph, storing the post-numbers and pre-numbers for each node. When the program is queried, it checks whether the edge (a, b) is a back edge (a is descended from b), a tree or forward edge (b is descended from a), or a cross edge (a and b share a common ancestor but are not descended from each other).

(c) We can view each box as a node in a directed graph, with an edge from a to b indicating that a fits in b. Since Falexa only gives the edges in the opposite direction (in that it tells Bob the boxes that can fit into b), we have to construct a graph and then reverse the edges, which is doable in linear time. This graph is a DAG. If box a can fit in box b and box c can fit in box a, box b cannot fit into box c. We set incoming edge weights to a node a to be the weight w of the corresponding box. We can linearize this DAG by running DFS starting at the smallest box x, and then find the shortest path from the x to any other box in linear time by iterating through the DAG's vertices in topological order to compute each new shortest path by building on paths to previous nodes in the DAG.

4 The Greatest Roads in America

Solution. *We want to build a new graph G' such that we can apply Dijkstra's algorithm on G' to solve the problem. We'll start by creating $k+1$ copies of G. Call these $G_0, G_1, ..., G_k$. These copies include all the edges and vertices in G, as well as the same weights on edges. Let the copy of v in G_i be denoted by v_i. For each road $(u, v) \in R$, we also add the edges $(u_0, v_1), (u_1, v_2), ...(u_{k-1}, v_k)$, with the same weight as (u, v). The intuition behind creating these copies is that each time we use an edge in G' corresponding to an edge in R, we can advance from one copy of G to the next, and this is the only way to advance to the next copy. So if we've reached v_i from a_0, we know we must have used (at least) i edges in R so far. Now, consider any path p' from a_0 to a_k in G'. If we take each edge (u_i, v_i) or (u_i, v_{i+1}) and replace it with the corresponding edge (u, v) in G, we get a path p in G from a to itself. Furthermore, since the path goes from a_0 to ak, it contains k edges of the form (u_i, v_{i+1}), where (u, v) is an edge in R. So, p will contain at least k edges in R. Our algorithm is now just to create G' as described above, and find the shortest path p from a_0 to a_k using Dijkstra's, and then output the corresponding path p in G.*

Proof. Assume there is a valid path p in G that is shorter than the one produced by this algorithm. Consider the equivalent path p' in G' formed by modifying the path to go to the next copy of G whenever an edge of R is crossed. Since p is valid, p' must go from a in G_0 to a in G_k. □

Runtime Analysis. *Since G' includes $k+1$ copies of G, Dijkstra's algorithm will run in time $\mathcal{O}((km + kn)\log(kn))$.*

5 Detecting Cyclic Vertices

Solution. *Run Kosaraju's algorithm on G to compute all of its strongly connected components and slightly modify the algorithm so that it also computes the number of vertices in each SCC. Then, the desired set of vertices can be obtained by including all vertices $v \in V$ that lie in an SCC with size at least 2.*

Proof. Any SCC $S \subset G$ is a subgraph in which all its vertices $v \in S$ can reach one another. Thus, the vertices of any SCC with size at least two by definition have a cycle because any vertex v can reach any other vertex $u \in S$, which in turn can return to v. □

Runtime Analysis. *Running Kosaraju's takes $\mathcal{O}(m+n)$ time, and traversing through all of the vertices to determine the desired set of vertices takes $\mathcal{O}(n)$ time. This is assuming that during Kosaraju's, we construct a hashmap with vertices as keys and SCC size as values. Thus, the overall runtime is*

$$\mathcal{O}(m+n) + \mathcal{O}(n) \in \mathcal{O}(m+n)$$