1. You are given a list $D[n]$ of n words each of length k over an alphabet Σ in a language you don't know, although you are told that words are sorted in lexicographic order. Using $D[n]$, describe an algorithm to efficiently identify the order of the symbols in Σ. For example, given the alphabet $\Sigma = \{Q, X, Z\}$ and the list $D = \{QQZ, QZZ, XQZ, XQX, XXX\}$, your algorithm should return QZX. You may assume D always contains enough information to completely determine the order of the symbols. (Hint: use a graph structure, where each node represents one letter.)

Solution: Consider two words, $D[i]$, $D[i+1]$. Consider j such that $D[i][j] \neq D[i+1][j]$ and $\forall k < j$, $D[i][k] = D[i+1][k]$. That is, j is the index of the first different letter between $D[i]$ and $D[i+1]$. The comparison of $D[i][j]$ and $D[i+1][j]$ reveals the order between the two letters. Any further comparison of $D[i]$ and $D[j]$ would not help, since the following letters do not affect lexicographic order of $D[i]$ and $D[i+1]$. Also, for arbitrary x, y, z such that $x < y < z$, if you are given the comparisons of $D[x], D[y]$ and $D[y], D[z]$, then the comparison of $D[x], D[z]$ does not reveal any additional information about the order(Why? Let j, k be the first different index between $D[x], D[y]$ and $D[y], D[z]$ respectively. Reason about three cases: $j < k$, $j = k$, $j > k$). Therefore, the problem can be solved by constructing the following directed graph.

- $V = \{v \mid v \in \Sigma\}$
- $E = \{(u, v) \mid u, v \in \Sigma, u \neq v, \exists i, j \text{ s.t. } D[i][j] = u, D[i+1][j] = v, \forall k < j, D[i][k] = D[i+1][k]\}$

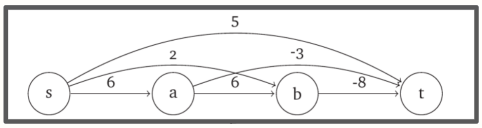
Note that for any edge $(u, v) \in E$, there is a corresponding pair of consecutive words $(D[i], D[i+1])$ such that if j is the index of the first different letter, then $D[i][j] = u$ and $D[i+1][j] = v$. This means for any edge (u, v), we know for sure that u comes before v in their language. Since there can be no cycle in the graph, it can be topologically sorted to obtain the order of symbols. The running time of the algorithm is $O(nk)$, since in worst case we should iterate over every symbol in D to construct the graph.

■

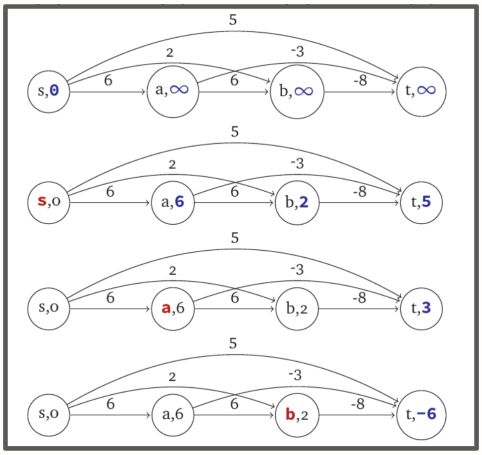
2. Given a directed-acyclic-graph ($G = (V, E)$) with integer (positive or negative) edge weights:

   (a) Give an algorithm to find the **shortest** path from a node s to a node t.

Solution: Because there are negative edge weights we cannot use a greedy method like Dijkstra's algorithm; however because the graph is directed and acyclic we can use a topological sort. Topologically sorting the graph means that every vertex can only reach vertices below them in the sort and cannot reach vertices above them in the sort.



This means after topologically sorting the graph we start at node s and compute the shortest path from node s to each of the nodes below it in sequential order. To do this we initialize the distance from s to all the other nodes as ∞ then we update the distance from node s to be the weight of the edges from s to the neighbors of s. Then we move on to the next sequential node say u, and look at its neighbor, say v. If the distance from s to u plus the edge weight from u to v is less than the current distance from s to v then we update the value. This repeats until we reach node t.



Let s and t be the nth and mth node in the topological sort and let $N(u)$ be the neighbor set of u. Then the algorithm is

```
DAGSP(V, E, s, t):
  (Vs, Es) ← TopSort(V, E)
  n ← index(s)
  m ← index(t)
  D[n] ← 0
  for k ← n + 1 to m
      D[k] ← ∞
  for i ← n to m − 1
      for v in N(v_i)
          j ← index(v)
          D[j] ← min{D[j], D[i] + Es[i][j]}
  return D[m]
```

A topological sort takes $\boldsymbol{O(V+E)}$ time and the for loops in the algorithm takes $\boldsymbol{O(V+E)}$. So the total running time is $\boldsymbol{O(V+E)}$. ■

(b) Give an algorithm to find the **longest** path from a node s to a node t.

Solution (Direct): To find the longest path from node s to node t we can do the same process as part a) but instead we initialize the values to $-\infty$ then take the maximum value.

```
DAGLP(V, E, s, t):
  (Vs, Es) ← TopSort(V, E)
  n ← index(s)
  m ← index(t)
  D[n] ← 0
  for k ← n + 1 to m
      D[k] ← −∞
  for i ← n to m − 1
      for v in N(v_i)
          j ← index(v)
          D[j] ← max{D[j], D[i] + Es[i][j]}
return D[m]
```

This has a running time of $\boldsymbol{O(V+E)}$.

■

Solution (Reduction): This problem can be reduced to the part a). Multiplying all of the edge weights by -1 then finding the shortest path then multiplying that value by -1 yields the longest path.

```
DAGLP(V, E, s, t):
  Et ← −1 ∗ E
  x ← DAGSP(V, Et, s, t)
  return −1 ∗ x
```

This has a running time of $\boldsymbol{O(V+E)}$.

■

3. Suppose you are given two dags $G(V_G, E_G)$ and $H(V_H, E_H)$ in which every node has a *label* from some finite alphabet; different nodes may have the same label. The label of a *path* in either dag is the string obtained by concatenating the labels of its vertices.

   (a) Describe and analyze an algorithm to compute the length of a longest string that is both the label of a path in G and the label of a path in H.

Solution: Let $L(v)$ denote the label of $v \in V_G \cup V_H$. We can reduce to a longest path problem in an *unweighted dag* $G' = (V, E)$ as follows:

- $V = (V_G \times V_H) \cup \{s\}$
- $E = E_1 \cup E_2$ where
  - $E_1 = \{(u_1, u_2) \to (v_1, v_2) \mid (u_1, u_1) \in E_G, (v_1, v_2) \in E_H, L(v_1) = L(v_2)\}$
  - $E_2 = \{(s, (v_1, v_2)) \mid v_1 \in V_G, v_2 \in V_H, L(v_1) = L(v_2)\}$
- We need to compute the length of the longest path from s to any vertex in G'. We can do this by *dynamic programming* using the algorithm in Jeff's lecture notes as G' is a dag.
- Constructing the G' by brute force and computing the length of the longest path in G' takes $O(V_G V_H + E_G E_H)$ since we need to iterate through every pairs of $V_G \times V_H$ and $E_G \times E_H$. Overall, the algorithm requires ***time*** $\boldsymbol{O(V_G V_H + E_G E_H)}$.

Claim 1. *The length of the longest path in G' is the length of the longest string that is both the label of a path in G and the label of a path in H.*

Proof: Let i be the length of the longest path in G'. Let j be the length of the longest common string in the paths of G and H. For the sake of contradiction, suppose $i \neq j$.

- Suppose $i < j$. Then, there exists a path $g_1 g_2 ... g_j$ in G and a path $h_1 h_2 ... h_j$ in H such that $L(g_k) = L(h_k)$ for every $1 \leq k \leq j$. However, by the definition of G', $s, (g_1, h_1), (g_2, h_2) ... (g_j, h_j)$ is a path in G' of length j, which contradicts our assumption on the length of the longest path in G'. Therefore, we have $i \geq j$.
- Suppose $i > j$. Then, there exists a path $s, (d_1, f_1), (d_2, f_2) ... (d_i, f_i)$ in G'. However, by definition, $d_1 d_2 ... d_i$ is a path of length i in G and $f_1 f_2 ... f_i$ is a path of length i in H such that $L(d_k) = L(f_k)$ for every $1 \leq k \leq i$, which contradicts our assumption on the longest common string in the paths of G and H. Therefore, we have $i \leq j$.

We conclude that $i = j$, which means the length of the longest path in G' is the length of the longest string that is both the label of a path in G and the label of a path in H. □

(b) Describe and analyze an algorithm to compute the length of the longest string that is both a subsequence of the label of a path in G and a subsequence of the label of a path in H.

Solution: We reduce to a longest path problem on a *weighted* dag $G' = (V, E)$ as follows:

- $V = (V_G \times V_H) \cup \{s, t\}$
- We add an edge $(u_1, u_2) \to (v_1, v_2)$ between two vertices $(u_1, u_2), (v_1, v_2) \in V \setminus \{s, t\}$ in the following three cases:
  - $(u_1, v_1) \in E_G$ and $u_2 = v_2$. These edges have weight 0.
  - $u_1 = v_1$ and $(u_2, v_2) \in E_H$. These edges have weight 0.
  - $(u_1, v_1) \in E_G, (u_2, v_2) \in E_H$ *and* $L(v_1) = L(v_2)$. These edges have weight 1.

  Then, we add edges from s to every vertex in $V \setminus \{s, t\}$ with weight 0, and then from every vertex in $V \setminus \{s, t\}$ to t with weight 0.
- We need to compute the length of the longest path from s to t in G'. We can do this by *dynamic programming* using the algorithm in Jeff's lecture notes as G' is a dag.
- We have that $V = O(V_G V_H)$ and $E = O(V_G E_H + V_H E_G + E_G E_H)$. Thus, the size of the graph is $O(V_G V_H + V_G E_H + V_H E_G + E_G E_H) = O((V_G + E_G)(V_H + E_H))$. Thus, constructing G' by brute force and computing the length of the longest path in G' requires time $O((V_G + E_G)(V_H + E_H))$ as G' is a dag. Overall, the algorithm requires time $\boldsymbol{O((V_G + E_G)(V_H + E_H))}$.

Claim 2. *The length of the longest path from s to t in G' is the length of the longest string that is both a subsequence of a path in G and a subsequence of the label of a path in H.*

Proof: Let i be the length of the longest path in G'. Let j be the length of the longest common string in the paths of G and H. Let $E_1 \subseteq E$ denote the set of all edges in E weighted as 1. For the sake of contradiction, suppose $i \neq j$.

- Suppose $i > j$. Then, there exist a path $p = s n_1 n_2 ... n_k t$ in G' for some $k \geq 0$ that contains $i > j$ edges from E_1. Let us denote $n_l = (u_l, v_l)$ for $1 \leq l \leq k$. By the definition of E_1, there exists a set S of i indices such that $L(u_s) = L(v_s)$ for any $s \in S$. However, by the definition of E, we can obtain a path in G that contains every elements in the set $\{u_s | s \in S\}$ by concatenating $u_1, u_2, ..., u_k$ and removing the repeated vertices. Similarly, we can obtain a path in H that contains every elements in the set $\{v_s | s \in S\}$. Since we have $L(u_s) = L(v_s)$ for every $s \in S$ and $|S| = i > j$, the length of the longest common subsequence of the labels of the paths in G and H is at least i, which is a contradiction.
- Suppose $i < j$. Then, there exists a path $p_G = g_1 g_2 ... g_n$ for some n in G and a path $p_H = h_1 h_2 ... h_m$ for some m in H such that p_G and p_H has $w = w_1 w_2 ... w_j$ as the longest common subsequence of the labels. Then, we can construct a path with length j in G' by the following method:
  - Initially take the edge $(s, (g_1, h_1))$

- If for the current vertex g_q, h_r, both g_q and h_r are parts of w(that is, if $L(g_q) = L(h_r)$ and those were indeed used as a pair to get w), then take the edge $((g_q, h_r), (g_{q+1}, h_{r+1}))$.
- If g_q is a part of w but h_r is not, then take the edge $((g_q, h_r), (g_q, h_{r+1}))$
- If only h_r is a part of w, then take the edge $((g_q, h_r), (g_{q+1}, h_r))$
- If both g_q and h_r are not a part of w, take either $((g_q, h_r), (g_q, h_{r+1}))$ or $((g_q, h_r), (g_{q+1}, h_r))$.

The length of the path increments when taking the edge $((g_q, h_r), (g_{q+1}, h_{r+1}))$. Since we can obtain a path of G' with length greater than i, we have a contradiction.

Therefore, we conclude that $i = j$. □

(c) Describe and analyze an algorithm to compute the length of the shortest string that is both a supersequence of the label of a path in G and a supersequence of the label of a path in H.

Solution: The following is an algorithm to do this:

```
SCS(G(V_G, E_G), H(V_H, E_H)):
  for i ← 1 to V_G + V_H:
      seenLabels[i] ← FALSE
  for v in V_G:
      seenLabels[L(v)] ← TRUE
  for v' in V_H:
      if seenLabels[L(v')] ← TRUE:
          return 1
  return 2
```

The correctness of this algorithm follows from the following observations. If there exists a vertex v in G and a vertex v' from H such that $L(v) = L(v')$, then $L(v)$ is the shortest string that is a supersequence of the label of the path v in G and the label of the path v' in H. If there is no such pair of vertices, the string must contain at least two symbols to be a supersequence of both the label of a path in G and the label of a path in H. This algorithm requires ***time*** $\boldsymbol{O(V_G + V_H)}$. ■

4. (a) Define the *width* of a walk in a graph to be the maximum weight of any vertex in the walk. Let $G(V, E)$ be an *undirected* graph with weighted edges, $w(v)$ be the weight of each vertex v, s be a starting vertex, t be a target vertex and L be a maximum length. Describe and analyze an algorithm that returns the *maximum width* of any walk from s to t in G whose total length is at most L. You may assume G has no negative cycles but *not* that it has no edges of negative weight.

Solution: Let $G^R = (V, E^R)$ where $u \to v \in E \iff v \to u \in E^R$. Fix $v \in V$. Notice that a shortest t-v path *in* G^R is a v-t shortest path *in* G. Thus, Bellman-Ford from t *in* G^R suffices to compute the shortest-path distances from *every vertex* to t *in* G as G^R has no negative weight cycles. This implies the following algorithm solves the problem:

```
MaxWidth(G(V,E), s, t, L):
  construct G^R(V, E^R) from G
  d(s, ·) ← Bellman-Ford(G , s)
  d(·, t) ← Bellman-Ford(G^R, t)
  maxWidth ← −∞
  for v in V:
      if d(s, v) + d(v, t) ≤ L:
          maxWidth ← max{maxWidth, w(v)}
  return maxWidth
```

Constructing $G^R(V, E^R)$ by brute force takes time $O(V + E)$. Thus, the runtime of this algorithm is dominated by the application of the Bellman-Ford algorithm on G and G^R which requires ***time $O(VE)$***. ■

(b) Let $G(V, E)$ be a *directed* graph with *nonnegative* edge weights, let s and t be vertices of G and let $H(V, E')$ be a subgraph of G where $E' \neq E$. Suppose we want to reinsert *exactly one* edge from G back into H, so that the shortest path from s to t in the resulting graph is as short as possible. Describe and analyze an algorithm that chooses the best edge to reinsert.

Solution: Let $H^R = (V, E^R)$ where $u \to v \in E' \iff v \to u \in E^R$. As G (and thus H) has nonnegative weight edges, this algorithm solves the problem:

```
OneMoreEdge(G(V,E), H(V,E'), s, t):
  construct H^R(V, E^R) from H
  d(s, ·) ← Dijkstra(H , s)
  d(·, t) ← Dijkstra(H^R, t)
  minDist ← d(s, t)
  bestEdge ← None
  for u → v to E \ E':
      if minDist > d(s, u) + ℓ(u → v) + d(v, t):
          minDist ← d(s, u) + ℓ(u → v) + d(v, t)
          bestEdge ← u → v
  return bestEdge
```

Constructing $H^R(V, E^R)$ by brute force takes time $O(V + E)$. Thus, the runtime of this algorithm is dominated by the application of Dijkstra's algorithm on H and H^R which requires ***time $O(E + V \log V)$***. ■