6.1 Can I Haz More Frendz?

Alyssa P. Hacker is interning at RenBook, a burgeoning social network website. She needs to implement a new friend suggestion feature. For two friends u and v, the 'EdgeRank' $ER(u,v)$ can be computed in constant time based on the interest u shows in v. Assume that EdgeRank is directional and asymmetric, and that its value falls in the range (0,1] A user u is 'connected' to a user v if they are connected through some mutual friends, i.e. $u=u_0$ has a friend u_1, who has a friend $u_2, \ldots$, who has a friend $u_k=v$. The integer k is the 'vagueness' of the connection Define the 'strength' of the connection to be

$$S(p) = \prod_{i=1}^{k} ER(u_{i-1}, u_i)$$

For a given user s, Alyssa wants to rank potential friend suggestions according to the strength of the connections s has with them. In addition, the vagueness of those connections should not be more than k, a value Alyssa will decide later.
Help Alyssa by designing an algorithm that computes the 'strength' of the strongest connection between a given user s and every other user v to whom s is connected with vagueness at most k, in $O(kE+V)$ time. Assume the network has $|V|$ users and $|E|$ friend pairs. Analyze the running time of your algorithm.

We want to maximize $S(p)\ \forall p \in V \setminus \{s\} \ni$ vagueness of $p \le k$.
Maximizing $S(p) \cong$ maximizing $\log S(p) = \log \prod_{i=1}^{k} ER(u_{i-1}, u_i) = \sum_{i=1}^{k} \log ER(u_{i-1}, u_i)$
$\cong$ minimizing $-\log(S(p)) = \sum_{i=1}^{k} \log \frac{1}{ER(u_{i-1}, u_i)} = w(p) \leftarrow$ path weight.
So we can modify this problem to be, find the shortest path, weighted by $w(p)$, from $s \to v$ ($\forall v \in V$) using only paths of length $\le k$. This is accomplished on the next page using a modified Bellman Ford algorithm, where we use a dictionary to ensure updates happen after searching edges, so paths of length $> k$ are not included.

```
KLENGTHBELLMANFORD(G,w,s)
for each vertex v∈G.V                        /initialize
    v.d = ∞                                             O(V)
s.d = 0
T = {}
for i = 1 to k                                                        O(V+kE
    for each edge (u,v)∈G.E      /search edges
        if v.d > u.d + w(u,v)                      O(E)    O(kE)
            T[v] = u.d + w(u,v)              /relax
    for each edge v∈T            /update after searching   O(E)
        v.d = T[v]
```

6.2. Renbook Competitor

You must install a webserver and all of its dependencies. Each library that you wish to install can depend on a number of other libraries, which you will have to install first. Each of those libraries can in turn have its own dependencies.
You will need to determine which libraries need to be installed and then generate the order in which the libraries will be installed so that there will be no dependency problems.
Examining the software library repository, you see that there are V total libraries, which together have a total of E dependencies. The repositories enforce the rule that dependencies cannot be cyclic. Libraries rarely all depend on each other, so you can safely assume that $E << V^2$.

a. An <u>installation order</u> is an ordering of all the libraries such that each library's dependencies appear prior to it in the sequence. If we install each library in this sequence in order, we are guaranteed to avoid dependency problems. Describe in detail how to generate an installation order for the entire repository in $O(V+E)$ time.

Let G be a graph of the libraries where each library is represented as a vertex and each dependency is represented as a directed edge ((u,v) if u depends on v). G is a DAG, so we can perform a topological sort on G (a linear ordering $\ni \forall (u,v) \in E$, u is before v). The resulting topological order will be our installation order. The graph can be constructed in $O(V+E)$ time (adding V vertices and E edges in constant time), and the topological sort can be performed via a stack based DFS in $O(V+E)$, allowing us to construct an installation order in $O(V+E)$.

We wish to install a web server library along with its depencies. Suppose that some libraries are already installed on your system, and that only P libraries remain to be installed (you can determine whether a library has already been installed by performing a dictionary look up in $O(1)$ time). Assume that the maximum number of dependencies for any given library is D.

b. Give pseudocode for an algorithm that generates an installation order for the non-installed libraries that are needed for installing the webserver library in $O(P+PD)$ time. Describe your algorithm. You may use any routine in CLRS as a subroutine in your pseudocode, and you can use a textual description, a clarifying example, or a correctness proof for the description.

Assuming we already have the graph from part A (which the solution assumes, but the problem does not state), we already have our graph, and now we simply topologically sort the remaining edges, which are $O(PD)$

(Borrowing & modifying pseudocode from the text)

```
INSTALL-ORDER(G)
  return ModTopoSort(G)

DFS-VISIT(G, u, order)
  for v in G.adj[u]
      if v.installed == FALSE and v.visited == FALSE:
          v.visited == TRUE
          DFS-VISIT(G, v, order)
  order.append(v)

ModTopoSort(G)
  order = []
  DFS-VISIT(G, u, order)
  return order
```

Topological sort is $O(V+E) \rightarrow O(P+PD)$ after pruning

6.3. Rubik's Cube

In this problem, you will develop algorithms for solving the 2x2x2 Rubik's Cube. Call a configuration of the cube "k levels from the solved position" if it can reach the solved configuration in exactly k twists, but cannot in any fewer.
The 'rubik' directory in the problem set package contains the Rubik's Cube library and a graphical user interface to visualize your algorithm.
We will solve the Rubik's Cube puzzle by finding the shortest path between two configurations (the start & the goal) using BFS.
A BFS that goes as deep as 14 levels (the diameter of the cube) will take a few minutes. This is too slow and requires too much memory.
With that in mind, we can instead do a two-way BFS, starting from each end at the same time, and meeting in the middle. At each step, expand one level from the start position, and one level from the end position, and then check to see whether any of the new nodes have been discovered in both searches. If there is such a node, we can read off parent pointers (in the correct order) to return the shortest path.
Write a function 'shortest_path' in 'solver.py' that takes two positions, and returns a list of moves that is a shortest path between two positions.
Test your code using 'test_solver.py'. Check that your code runs in < 5 seconds

My tests run in 2.2 sec
See solver.py.

6.4. From Berklee to Berkeley

Your task is implementing the method 'PathFinder.dijkstra(weight, nodes, source, destination)' using Dijkstra's algorithm. It is given a function 'weight(node1, node2)' that returns the weight of the link between 'node1' and 'node2', a list of all the 'nodes', a 'source' node and a 'destination' node in the network. The method should return a tuple of the shortest path from the 'source' to the 'destination' as a list of nodes, and the number of nodes visited during the execution of the algorithm. A 'node' is visited if the shortest path of it from the 'source' is computed. You should stop the search as soon as the shortest path to the destination is found.
You can run the following command to run all the tests on your Dijkstra's implementation
'python dijkstra_test.py'
When your code passes all tests in <40s, submit it.

See dijkstra.py
Passes all tests in 29s.