CS 170 Algorithms
Fall 2014 David Wagner Sol 5

1. (10 pts.) Super-long path in a DAG

Design a linear-time algorithm for the following task:

Input: A directed acyclic graph G

Question: Does G contain a directed path that touches every vertex exactly once?

Solution #1:

Main idea. If a path exists that touches each vertex exactly once, there must be an edge between any two adjacent nodes in the topological sort, since the edges can only go in the increasing direction in the linearized order. Thus, we just need to check if the DAG has an edge (v_i, v_{i+1}) for every pair of consecutive vertices labeled v_i and v_{i+1} in the linearized order. Such a path is called a Hamiltonian Path.

Pseudocode.

```
FindHamiltonianPath(Graph $G$):
1.  Linearize the graph $G$, and let $v_1, \dots, v_n$ denote the vertices of $G$ in topologically sorted order.
2.  For $i := 1, 2, \dots, n-1$:
3.      If $(v_i, v_{i+1}) \notin E$:
4.          Return "No."
5.  Return "Yes."
```

Correctness. The correctness of the algorithm follows from the following: the following are equivalent for a dag G:

(a) G admits a unique linearization

(b) in any linearization of the vertices of G, every two consecutive vertices are connected with an edge

(c) G contains a directed path that touches all vertices exactly once

Here is how we can prove that:

- (a) $\Rightarrow$ (b): if a linearization is unique, then this means that there exist edges between every two consecutive vertices in the linearization, otherwise we can swap the position of two consecutive vertices and still have a (different) valid linearization.
- (b) $\Rightarrow$ (c): obviously the edges (between consecutive nodes) give us a directed path that touches every vertex of the graph exactly once.
- (c) $\Rightarrow$ (a): if there is a directed path that touches all vertices exactly once, then the order in which vertices are visited along this path consists a valid linearization of the graph; no other linearization is possible, as swapping the order of any two vertices, say u and v, would result in edge (u, v) going from right to left.

Running time. The topological sort in line 1 takes $\Theta(|V| + |E|)$ time. The loop iterates $|V|$ times, examining 1 edge in each iteration. So overall, we have a running time of $\Theta(|V| + |E|)$.

Solution #2: This is similar to Homework 4's program compilation problem (Q1), where we saw an algorithm to compute the longest path in a DAG. If the longest path in G is of length $|V|-1$, the answer to the question is yes (since any path that touches every vertex exactly once must be of length $|V|-1$), otherwise the answer is no.

We get the following algorithm:

```
FindHamiltonianPath(Graph $G$):
1. Call FindCompletionTime($G$) (from HW4 Q1 solutions), and let $r$ be the result.
2. If $r = |V|-1$, return "Yes", otherwise return "No."
```

No path in a DAG can be longer than $|V|-1$. If the longest path has length $|V|-1$, then the above algorithm is correct, as such a path must touch each vertex exactly once (since the graph is acyclic, no vertex will appear more than once in that path, and since the path touches $|V|$ vertices, each vertex must be touched exactly once). On the other hand, if the longest path has length $< |V|-1$, then there is no path that touches each vertex exactly once (since such a path would have length $|V|-1$).

Comment: This illustrates the power of reductions. If we can reduce the problem to a previously solved problem (namely, HW4 Q1), then we can just invoke the algorithm for that other problem. There is no need to re-prove the correctness of that algorithm; we can assume it is correct, and merely need to show that its output helps us produce the correct answer to our problem.

2. **(15 pts.) Number of shortest paths**

Often there are multiple shortest paths (all of the same length) between two nodes of the same graph. Design an efficient algorithm to count the number of shortest paths from vertex s to vertex t in a directed graph G (possibly containing cycles; all edges are of length 1).

Hint: throw away some of the edges, then you have a...

Solution #1:

Main idea. Run BFS starting from s. If $(u,v) \in E$ and $\text{dist}(u)+1 > \text{dist}(v)$, then the edge (u,v) cannot be part of a shortest path from s to t, so we can throw away this edge. Once you throw away these edges, you have a dag, so you can solve the problem by visiting the vertices in topologically sorted order.

Pseudocode.

```
FindNumberShortestPaths(Graph $G = (V,E)$):
1. Run BFS starting from $s$, to get the distances $\text{dist}(v)$ from $s$ to $v$ for each vertex $v \in V$.
2. For each $(u,v) \in E$:
3.     If $\text{dist}(u)+1 > \text{dist}(v)$, delete $(u,v)$.
4. Linearize $G$, to get the vertices $v_1, v_2, \ldots, v_n$ in topologically sorted order.
6. Set $\text{numpaths}(s) := 1$, and $\text{numpaths}(v) := 0$ for all other $v$.
7. For $i := 1,2,3,\ldots,n$ such that $v_i \neq s$:
8.     Set $\text{numpaths}(v_i) = \sum_u \text{numpaths}(u)$, where the sum is over all predecessors $u$ of $v_i$.
9. Return $\text{numpaths}(t)$.
```

Proof of correctness. We first show that the edge deletion in line 3 does not change the number of shortest paths:

Lemma 1 *If $\text{dist}(u)+1 > \text{dist}(v)$, then no shortest path from s to t can go through the edge (u,v).*

Proof: Consider any path $s \to \cdots \to u \to v \to \cdots \to t$. The length of the part $s \to \cdots \to u$ must be at least dist(u), so the length of the part $s \to \cdots \to u \to v$ is at least dist$(u)+1$. However, by definition of the distance, there exists a path from s to v with length dist(v), which is shorter. Therefore, if we replace the part $s \to \cdots \to u \to v$ of the original path with this path, then the entire path from s to t gets shorter. Therefore the original path through (u,v) must not have been a shortest path from s to t. □

Lemma 2 *At line 4, every remaining edge (u,v) satisfies dist$(v) =$ dist$(u)+1$.*

Proof: By the definition of shortest paths, we must have dist$(v) \leq$ dist$(u)+1$ (otherwise we could take a shortest path from s to u, append the edge (u,v) to the end, and we'd get a shorter path from s to t). And by the time we reach line 4, every remaining edge (u,v) satisfies dist$(u)+1 \leq$ dist(v) (otherwise it would have been deleted in lines 2–3). Therefore, dist$(v) =$ dist$(u)+1$ for each such edge. □

Lemma 3 *At line 4, the graph is acyclic.*

Proof: Suppose there was a cycle of length $k > 0$, say $u_1, u_2, \ldots, u_k, u_1$. Then by repeatedly applying Lemma 2, we find

$$\text{dist}(u_1) = \text{dist}(u_k)+1 = \text{dist}(u_{k-1})+2 = \cdots = \text{dist}(u_2)+k-1 = \text{dist}(u_1)+k.$$

But this implies $k = 0$, a contradiction, so there couldn't have been any cycle in the first place. □

These lemmas imply that we have a dag at line 4, so we can topologically sort it.

Also, any shortest path from s to v_i must consist of a shortest path from s to u, followed by the edge (u,v), for some predecessor u of v_i. Each such graph is distinct. Therefore, the number of shortest paths from s to v_i is the sum of the number of shortest paths from s to each of the predecessors of v_i. Therefore, by induction on i, lines 6–8 ensure that numpaths(v_i) is set to the number of shortest paths from s to v_i, for all i. The correctness of algorithm follows immediately.

Running time. The running time is $\Theta(|V|+|E|)$. BFS runs in $\Theta(|V|+|E|)$ time. Steps 2–3 can be done in $O(|V|+|E|)$ time, as can the topological sort. Steps 6–8 can also be implemented in $O(|V|+|E|)$ time, if we have a list of all of the edges incoming into each vertex (i.e., adjacency lists for the reverse graph), and those lists can be computed in $O(|V|+|E|)$ time. Therefore, the total running time is $\Theta(|V|+|E|)$.

Solution #2:

Main idea. If we use a modified BFS, we can keep track of the number of shortest paths to a given node. The length of the shortest path to v will be the lowest depth at which BFS first encounters v. If any other path to v has the same length, it is also a shortest path, so we should increment the number of the shortest paths to the node. If $x_1, x_2, \ldots x_k$ are vertices at depth l in the BFS tree and x is a vertex at depth $l+1$ such that $(x_1,x), \ldots (x_k,x) \in E$ then we want to set numpaths$(x) =$ numpaths$(x_1) + \cdots +$ numpaths(x_k). Thus we get the following algorithm.

Pseudocode.

```
FindNumberShortestPaths(Graph $G = (V,E)$):
1. For each $v \in V$:
2.     Set dist$[v] := \infty$.
3.     Set numpaths$[v] := 0$.
4. Set dist$[s] := 0$.
```

```
5.  Set numpaths$[s] = 1$.
6.  Push $s$ onto queue $Q$.
7.  While $Q$ is not empty:
8.      $u$ = pop($Q$)
9.      For each $(u,v) \in E$:
10.         If dist$[v] = \infty$:
11.             Set dist$[v] :=$ dist$[u] + 1$.
12.         If dist$[v] =$ dist$[u] + 1$:
13.             Set numpaths$[v] :=$ numpaths$[v] +$ numpaths$[u]$.
```

Correctness. A useful invariant is this: at the end of each iteration of the while loop in line 7, numpaths$[u]$ contains the number of shortest paths from s to u. Let's prove it, by induction on the number of iterations of the while loop.

Base case: In the first iteration of the while loop, the u popped off Q is s. numpaths$[s] = 1$, which is correct, since there is precisely one path from s to s, traversing no edges.

Inductive case: Now, for a vertex u popped of the queue, we know that any node at a higher depth in the BFS tree has already been explored. Consider any edge $(v,u) \in E$ into u. By the inductive hypothesis, numpaths$[v]$ correctly counts the number of shortest paths from s to v, for each edge $(v,u) \in E$. The number of shortest paths to u is equal to the sum of all the shortest paths of all u's parents, which are precisely the nodes v previously mentioned. Therefore, at the end of this iteration of the loop, numpaths$[u]$ contains the number of shortest paths from s to u.

Running time. Since we do a constant amount of work per iteration of the loop in line 7, this has the same running time as BFS, $\Theta(|V| + |E|)$.

3. **(20 pts.) The farmer and the river**

A farmer has a wolf, sheep, and a cabbage. Initially they are all on the west bank of a river; he wants to transport them all to the east bank of the river. He can carry at most one of them in his boat at a time. Unfortunately, if he leaves the wolf and the sheep alone on the same bank, the wolf will eat the sheep; if he leaves the sheep and the cabbage alone, the sheep will eat the cabbage. In each step, the farmer can cross the river in his boat from west to east or from east to west.

(a) Find a way to get them all to the east bank, in the least number of crossings, by forming a graph and then doing something with the graph. Draw your graph. How many steps does the shortest solution require?

(b) How many different such solutions (with that number of crossings) are there?

Solution: We can construct a graph where the nodes are the states, and there is a directed edge from state a to state b if by one crossing we get state b from a. We name the states using four-bit binary strings, where the four bits correspond to the location of the farmer, wolf, sheep, and cabbage respectively, and 0 means East, and 1 means West. For example, the state 0101 is the state where the farmer and sheep are at the east bank and the wolf and cabbage are on the west bank. There are $2^4 = 16$ states in total. Among them 10 states are safe (nothing will be eaten):

$$V = \{0000, 1010, 0010, 1110, 1011, 0100, 0001, 1101, 0101, 1111\}$$

There are 10 edges in total:

$$0000 \to 1010, 1010 \to 0010, 0010 \to 1110, 1110 \to 0100,$$
$$0100 \to 1101, 1101 \to 0101, 0101 \to 1111, 0010 \to 1011,$$
$$1011 \to 0001, 0001 \to 1101.$$

(a) Running breadth-first search by hand, we find that the shortest path from 0000 to 1111 has length 7. Therefore, the shortest solution takes 7 river crossings (7 steps). There are two such paths of length 7 from 0000 to 1111, namely

$$0000 \to 1010 \to 0010 \to 1110 \to 0100 \to 1101 \to 0101 \to 1111$$
$$0000 \to 1010 \to 0010 \to 1011 \to 0001 \to 1101 \to 0101 \to 1111$$

(You did not need to list either path in your solution.)

(b) Running the algorithm from Q2, we find that there are two paths of length 7 from 0000 to 1111, namely the two given above.

4. (15 pts.) Equality contraints

Here's a problem that shows up in program analysis. Suppose we have a list of variables $x_1, x_2, \ldots, x_n$ (over the real numbers). We are given some equality constraints, of the form $x_i = x_j$, and some disequality constraints, of the form $x_i \neq x_j$. Design an efficient algorithm to determine whether it is possible to satisfy them all, when we are given m constraints over n variables.

For instance, the constraints $x_1 = x_2$, $x_2 = x_3$, $x_1 \neq x_3$ cannot be simultaneously satisfied, so your algorithm should output "No" on that input.

Solution:

Main idea. Let the variables x_i be the nodes of our graph. We first examine the equality constraints, adding an undirected edge (x_i, x_j) for each constraint of the form $x_i = x_j$. We run DFS to discover connected components, labeling each node with its connected component number. The connected components represent transitive equality between all nodes in the connected component. Now we examine the inequality constraints. If we have a constraint $x_i \neq x_j$ where x_i and x_j are in the same connected component, then the equalities require $x_i = x_j$, so we output "No". Otherwise, if all inequalities occur between nodes belonging in separate connected components, we can output "Yes".

Pseudocode.

```
SatisfyConstraints(List $V$ of variables, List $E$ of constraints):
1. Construct an adjacency list of the $n$ variables
2. For constraint $x_i = x_j \in E$:
3.     Add $x_i$ to $x_j$'s neighbor list.
4.     Add $x_j$ to $x_i$'s neighbor list.
5. Run DFS, marking using previsit(v): ccnum[v] = cc,
       where cc is initialized to 0 and incremented each time DFS calls explore.
6. For constraint $x_i \neq x_j \in E$:
7.     If ccnum[$x_i$] = ccnum[$x_j$]:
8.         Return "No."
9. Return "Yes."
```

Correctness. Line 5 correctly computes the connected components, as it is just the algorithm found in Chapter 3.2.3 of the book. After line 5, any two variables x_i, x_j in the same connected component must satisfy $x_i = x_j$. In particular, if x_i and x_j are in the same connected component, then there exists a path from x_i to x_j, meaning that either there is an edge (x_i, x_j) (given by an equality constraint when we constructed the graph), or there is some path $x_i \to v_1 \to v_2 \to \cdots \to v_k \to x_j$ (meaning there are equality constraints $x_i = v_1$, $v_1 = v_2$, $\ldots$, $v_k = x_j$). Because equality is transitive, a path between x_i and x_j implies $x_i = x_j$. Thus if we look at inequality constraints, if two nodes x_i and x_j lie in the same connected component, an equality constraint exists between them, and the given inequality cannot be satisfied.

Running time. Constructing the graph takes linear time, then DFS takes $\Theta(|V|+|E|)$. Finally, doing a pass over the inequality constraints takes $\Theta(|E|)$ and we do a constant time check for each constraint. So overall we have a running time of $\Theta(|V|+|E|)$.

5. (20 pts.) Telephone keypad entry

Amalgamated Systems Inc. is building flip-phones (so 80's!) and wants to develop a system to make it easy for people to enter in text messages via the numeric keypad. The usual solution is to map letters to sequences of numbers like this: A=2, B=22, C=222, D=3, E=33, F=333, G=4, etc. However it's annoying to have to type such a long sequence of numbers. Their marketing folks have come up with a great idea[1]: on their phone, they'll use the mapping A=2, B=2, C=2, D=3, E=3, F=3, G=4, etc.

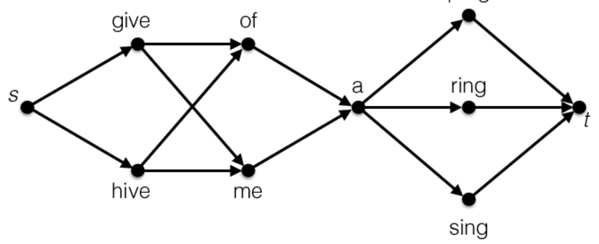
The problem, of course, is that some words are ambiguous. For example, 22737 could represent CARDS, CAPER, or a number of other English words. The company is counting on you to invent some software that somehow uses context to infer what the user was typing. Assume that the user will be typing English sentences, where all of the words are properly spelled. We'll use the * key to represent a space.

You are given a dictionary of all English words and the frequency of each of them: for each English word w, we are given $p(w)$, which represents the probability that any particular word will be w (i.e., the relative frequency of w in English text). You are also given frequency information about which words appear next to each other: for each pair w,x of English words, we are given $p(x|w)$, which represents the conditional probability that the next word is x, given that the current word is w. For instance, $p(\text{THE})$ is large (close to 1), because THE is a common word, while $p(\text{MALAPROPISM})$ is small (close to 0), because MALAPROPISM is rarely used in normal English. $p(\text{SKELTER}|\text{HELTER})$ might be very large (close to 1), while $p(\text{GIRAFFE}|\text{SMIRKING})$ might be very small (close to 0).

You are given a number sequence, e.g., 4483*63*2*7464. Your goal is to output the sequence of English words that has the highest total probability. The total probability of a sequence $w_1, w_2, \ldots, w_k$ of words is

$$p(w_k|w_{k-1})\cdots p(w_3|w_2)p(w_2|w_1)p(w_1).$$

For any particular number sequence, we can form a directed graph; below we show the graph corresponding to 4483*63*2*7464. The graph has one source node s and one sink node t. The ith level of the graph has one vertex for each possibility for the ith word, and there is an edge from each vertex at level i to each vertex at level $i+1$. Each path from s to t in the graph below provides a possible decoding of 4483*63*2*7464.



We want to find the path that corresponds to a word sequence whose total probability is highest. In this case, based upon word statistics, the decoding with the highest probability is GIVE ME A RING.

Design an efficient algorithm to solve the following task:

[1]It's such a great idea that it's been deployed in practice, under the name T9.

Input: The directed graph G obtained from the number sequence (e.g., the graph shown above), and the probabilities $p(\cdot)$ and $p(\cdot|\cdot)$

Output: The sequence of words $w_1, w_2, \dots, w_k$ that maximizes $p(w_k|w_{k-1}) \cdots p(w_3|w_2)p(w_2|w_1)p(w_1)$, out of all possible decodings consistent with the graph.

Try to find an algorithm whose running time is $O(|V| + |E|)$. If you cannot find a linear-time algorithm, we will accept one that is slower by a log-factor.

Solution #1:

Main idea. Put lengths on the edges that correspond to the probabilities, then visit the vertices of the graph in topologically sorted order. For each vertex v, we compute the highest-probability path from s to v (using information about the paths from s to each predecessor of v). Then when we are done, we have computed the highest-probability path from s to t.

Pseudocode.

```
FindSequence($G$):
1. Label each edge $s \to w_1$ with $p(w_1)$. Label each edge $w_i \to w_{i+1}$ with $p(w_{i+1}|w_i)$.
   Label each edge $w_k \to t$ with 1.
2. Linearize the graph, to get the vertices $v_1, \dots, v_n$ in topologically sorted order.
3. Set $P(s) := 1$ and best($s$) := [$s$] (the list containing just $s$).
4. For each $i := 2, 3, \dots, n$:
5.     Set $P(v_i) := 0$.
6.     For each edge $(u, v) \in V$:
7.         If $P(u) \times \ell(u,v) > P(v_i)$, set $P(v_i) := P(u) \times \ell(u,v)$ and best($v_i$) := best($u$) concat $v_i$.
8. Return best($t$).
```

Proof of correctness. This algorithm maintains the invariant that, after the end of the loop iteration for vertex v_i, best(v_i) represents the highest-probability path through the graph from s to v_i, and $P(v_i)$ represents the probability of that path (the product of the probabilities on the edges in the path). This invariant can be proven by induction on the number of iterations of the loop.

Running time. The running time is $\Theta(|V| + |E|)$, since topological sort can be done in that time, and we examine each edge exactly once in lines 6–7.

Solution #2:

Main idea. This is like a *almost* a shortest-path problem, but we are maximizing instead of minimizing. We can take a couple of different approaches to modify the problem slightly to use a shortest-path algorithm.

With a cool trick, we can do very little modification: For every edge (u, v) define the length

$$\ell(u,v) := -\log p(v|u).$$

Notice that, because $p(v|u) \in [0,1]$, it's clear that $\ell(u,v) \geq 0$ (it may even be infinite if $p(v|u) = 0$). So, using these edge lengths, we can apply Dijkstra's algorithm to get the shortest path from s to t, where "shortest" is defined by the new lengths $\ell(\cdot,\cdot)$. Assume that this shortest path follows vertices $s, v_1, v_2, \dots, v_{k-1}, t$ in that order. Now, we claim that this path is *also* the most probable path in the graph, i.e., that it maximizes $p(v_1|s) \times p(v_2|v_1) \times \cdots \times p(t|v_{k-1})$ (where we define $p(v_1|s) = p(v_1)$, and $p(t|v_{k-1}) = 1$). Why? Notice that $-\log(\cdot)$ is a monotonically decreasing function. Also, notice that

$$\begin{aligned} \ell(s,v_1) + \ell(v_1,v_2) + \cdots + \ell(v_{k-1},t) &= -\log p(v_1|s) - \log p(v_2|v_1) - \cdots - \log p(t|v_{k-1}) \\ &= -\log\left(p(v_1|s) \times p(v_2|v_1) \times \cdots \times p(t|v_{k-1})\right). \end{aligned}$$

So, minimizing the sum of the lengths is *exactly* the same as maximizing the product of the probabilities $p(\cdot|\cdot)$.

We can see that the provided graph will be a DAG, since edges only go from a vertex at level i to a vertex at level $i+1$. Given this fact, we can apply the algorithm for shortest paths in a DAG from Chapter 4.7 of the book, to our graph. The running time is $\Theta(|V|+|E|)$.

Alternately, running Dijkstra's algorithm on this graph will also give the shortest path, with $\Theta((|V|+|E|)\log|V|)$ running time.

Solution #3: We can change our update procedure in the following manner:

```
update($(u,v) \in E$):
    If dist($v$) $<$ dist($u$) $\times$ $p(v|u)$:
        Set dist($v$) := dist($u$) $\times$ $p(v|u)$.
        Set prev($v$) := $u$.
```

This maximizes the total probability, rather than minimizing the length of the probability. We can then follow the prev($\cdot$) pointers, to find the appropriate path to return.

Again, we can use either shortest-paths-in-DAGs or Dijkstra's algorithm.

However, it's necessary to prove that this modified algorithm is correct. That will essentially involve re-iterating the entire proof of correctness for shortest-path-in-DAGs and Dijkstra's algorithms, but with changes throughout to reflect the changes to the algorithm. That's a rather tedious and lengthy exercise.

Comment: Because proving that the tweaked shortest-path-in-DAGs or Dijkstra's algorithm is correct is rather burdensome, Solution #2 seems preferable. This is a valuable lesson to learn.

Modifying the *inputs* to an algorithm is often preferable to modifying the algorithm itself, because then we can re-use the existing proof of correctness for the unchanged algorithm. Any time that you can re-use an existing algorithm as a subroutine, you will likely save yourself a considerable amount of effort. As we will see later in this class, this is the notion of a *reduction*: we've reduced the problem of finding the most reliable path in a computer network, to the shortest paths problem.

Reductions play a central role in the study of algorithms.

Additionally, this log trick has an added bonus: it also lets you have very good precision. For instance, if we did the second solution, the numbers are going to get so small that a standard 64-bit double may not be able to represent it accurately. So, in a lot of applications like natural language processing, taking the log probability lets us work with these very tiny probabilities without having to resort to arbitrary precision arithmetic.

6. (5 pts.) Optional bonus problem: Golden graphs

(This is an *optional* bonus challenge problem. Only solve it if you want an extra challenge.)

Let $G = (V, E)$ be a graph where each edge is labelled with a non-negative length. Suppose we are given a set $E_g \subseteq E$ of "golden" edges. We call a path "golden" if at least k of its edges are golden. Let $d_g(s,t)$ denote the length of the shortest golden path from s to t. Describe an efficient algorithm to solve the following problem:

Input: A graph $G = (V, E)$, with non-negative edge lengths $\ell(u, v)$; a set $E_g \subseteq E$; and an integer $k \geq 0$

Output: $d_g(s, v)$ for each $v \in V$

Solution:

Main idea. Make a new graph G' with $k+1$ copies of the vertices in G. The vertices will be a tuple (v,i), where v is one of the original vertices, and i is a number between 0 and k. If the edge (u,v) in the original graph traverses a golden edge, we put edges in G' between nodes (u,i) and $(v,i+1)$ for all i. Also, we put edges in G' between nodes (u,i) and (v,i), for all $(u,v) \in E$ and all i. Now we are looking for the shortest path from $(s,0)$ to (v,k), which we can find by performing Dijkstra's algorithm on G'.

Pseudocode.

```
FindShortestGoldenPath(Graph $G = (V,E)$):
1.  $G'$ := empty graph
2.  For $v \in V$:
3.      For $(i = 0, i \leq k, i$++$)$:
4.          Add vertex $(v,i)$ to $G'$
5.  For each $(u,v) \in E$:
6.      For $i := 0, 1, \dots, k$:
7.          Add an edge from vertex $(u,i)$ to vertex $(v,i)$ in $G'$
8.          If $(u,v) \in E_g$ and $i < k$, add an edge from vertex $(u,i)$ to vertex $(v,i+1)$ in $G'$.
9.  Run Dijkstra's algorithm on $G'$ from source vertex $(s,0)$.
10. Return dist($(v,k)$) for each vertex $v \in V$.
```

Correctness. Since we are using Dijkstra's algorithm to find the shortest path we only need to prove that our formulation of the graph is correct for the desired output. G' allows precisely the same paths that G allows because an edge in G' between vertices $(u,\cdot)$ and $(v,\cdot)$ only exists if an edge (u,v) exists in G. We are seeking a path that uses k golden edges. We ensure that k edges are traversed because we only increment the second value of our vertex tuple in G' if a golden edge is traversed. Thus a path $(\cdot,k)$ means that k golden edges have been traversed.

Running time. Creating the vertices in G' takes $\Theta(|V| \cdot k)$ from the loops in lines 2 ($|V|$) and 3 (k). Creating the edges takes $\Theta(|E| \cdot k)$ from the loops in lines 5–7. Finally, running Dijkstra's algorithm (using a binary min-heap for our priority queue) on G' gives a running time of $\Theta((|V'| + |E'|)\log|V'|)$. In terms of the given graph, G, this is a running time of $\Theta((k \cdot |V| + k \cdot |E|)\log(k \cdot |V|)) = \Theta(k(|V| + |E|)\log(k|V|))$. This is about k times slower than Dijkstra's algorithm on the original graph.