CSE 101 Homework 2

CSE 101 Course Staff

Winter 2022

This homework is due on gradescope Friday January 28th at 11:59pm on gradescope. Remember to justify your work even if the problem does not explicitly say so. Writing your solutions in \LaTeX is recommended though not required.

Note: questions 3 and 4 here may make use of material that will not be covered in lecture until January 24th.

**Question 1** (Supply Chains, 25 points). Kenzie runs a widget factory. This factory uses a number of different inputs, and also needs to make a number of intermediate products on the way to making a single widget. Each product produced by Kenzie’s factory (either a widget or an intermediate product) requires a certain number of other, simpler products to build. Design an algorithm that given:

- A list of all the inputs and intermediate products produced by Kenzie’s factory
- For each non-input $P$ a list of all of the necessary products needed to produce one copy of $P$ and how many of each are required

computes the number of each input needed to manufacture a single widget.

For full credit, your algorithm should run in linear time.

**Solution 1.** First, we note that we can represent the production process as a directed acyclic graph, where vertices are products and directed edges from a product $p$ to a product $q$ represent needing $p$ to manufacture $q$ (there are no cycles since each product is made from a simpler product). We can formulate the problem as calculating a function $f$ on the vertices of this DAG, where $f(v)$ represents how many of $v$ we need to create a single widget. If vertex $v$ is required in products $p_1, p_2, \ldots, p_k$, and each requires $a_1, a_2, \ldots, a_k$ of $v$ to build, then

$$f(v) = \sum_i a_i f(p_i)$$

Since this is a function on a DAG that only requires the values of a node’s children to compute, we can compute a reverse topological sort of the DAG and then calculate each node in the reverse topological ordering in order using the above formula (where our widget, the base case, has $f(W) = 1$). After our algorithm runs, we can read off the amount of each input required to make a single widget from the values of $f$ computed at those inputs.

The runtime of this algorithm is linear, since topological sorting runs in linear time, and computing each function value only requires its child edges (and so over the whole run of the algorithm, we will only “see” each edge once when calculating $f$). The correctness of the algorithm follows from the fact that each node only relies on its child nodes to compute $f$, so running through the nodes in reverse topological order will calculate the correct value of $f$ for each input node.

We show more formally that this algorithm works by strong induction on $n$, the index in our reverse topological sort:

**Base case $n = 0$**

In the base case, we’re at the first index of our reverse topological sort. We assume WLOG that every input and intermediate product can be used to build the final widget $W$, and thus the widget is the only sink, and the very first element in the reverse topological sort. Since we only need a single widget to make one widget, $f(W) = 1$ is correct.
Inductive step
Assume that for \( k < n \), \( f(v_k) \) is the correct number of widgets for the \( k^{th} \) vertex in the reverse topological sort, \( v_k \). Since we are running through in reverse topological order, all \( c_i \) must have come before \( v_n \) in the topological ordering, and thus all \( f(v_i) \) are correct (by the inductive assumption). If \( v_n \) has children \( c_1, \ldots, c_m \), and is required \( a_i \) times to make a single \( c_i \), then we require \( a_i f(c_i) \) of \( v_n \) to make the number of \( c_i \) necessary to create the widget. Then, since the total number of \( v_n \) required to make a single widget is the total number of \( v_n \) required to make the total number of \( c_1, \ldots, c_m \) to make a single widget, the sum \( \sum_i a_i f(c_i) \) gives the correct value of \( f(v_n) \). This is precisely what our algorithm computes, and thus our algorithm will compute the correct number of each vertex required to make one \( W \).

Question 2 (Book Exchange, 30 points). Chris has a long reading list this summer, but unfortunately only has money to purchase a single book. Fortunately, he knows of a local book exchange. Each person at the book exchange is willing to swap a copy of some particular book for a copy of some other particular book. They are willing to make this same exchange as many times as necessary, but not all of these books are on Chris’ reading list. Chris would like to be able to purchase just a single book and then by repeatedly exchanging it eventually get to every book on his reading list. Give an algorithm that given Chris’ reading list and a list of potential exchanges determines whether or not this is possible to do.

For full credit, your algorithm should run in linear time.

Solution 2. We begin by noticing that this graph forms a directed (not necessarily acyclic) graph, where each vertex is a book and each directed edge is a trade that the book exchange offers (i.e. \( (u, v) \in E \) means that Chris can trade book \( u \) for book \( v \)). Then the problem reduces to finding a path in this graph that goes through every book in the reading list.

One way to simplify this problem is to notice that in a strongly connected component (SCC) of the graph, Chris can freely trade around to get any book he wants in that SCC, by the definition of an SCC. Then if we compute the metagraph, we just need to find a path in a DAG that hits every reading list book.

We can use the topological sort of the metagraph to calculate a function that describes such a path. There are many ways to define this function, but we detail one simple one here. We can label each node \( v \) of the metagraph with the number of reading list books it contains, \( n_v \). Then, we can define a function \( f \) that describes “the maximum possible number of reading list books that we can hit on any path ending in SCC \( v \)”. If there is a path through the metagraph that hits every reading list book, this recursively-defined function, which can be written

\[
 f(v) = \max_{(u, v) \in E} f(u) + n_v
\]

will hit the total number of reading list books somewhere. This is true because \( f \) gives the maximum number of reading list books that can be hit on any path in the DAG ending at vertex \( v \), and so if some path hits every reading list book, \( f \) will equal the number of reading list books at the last vertex of that path. Conversely, if \( f \) equals the number of reading list book at some vertex, we can follow the arguments that achieve the maximum back through the metagraph to find a path which, by the definition of \( f \), contains the number of books on the reading list, and thus contains every reading list book.

This is quite a number of steps, so we provide a very high-level pseudocode to clarify the process. We take as input the graph \( G \), as well as a list of reading list books \( R \):

1. Compute \( M \), the metagraph of \( G \).
2. Set \( n_v = 0 \) for every node in the metagraph.
3. For every book in the reading list, add 1 to the metagraph node which represents the SCC containing that book in \( G \). After this step, \( n_v \) will hold the number of books held in each metagraph node.
4. Set \( N \) to the length of \( R \).
5. Compute \( T_M \), a topological sorted list of the vertices in \( M \).
6. Run through $T_M$ in order, computing $f(v)$ for each $v$ using the formula above (using connectivity information about $v$, previous values of $f(\cdot)$, and $n_v$). If $f(v) = N$ for any node, return TRUE. Otherwise, return FALSE.

This algorithm runs in linear time because it uses a few linear time algorithms in sequence (computing the metagraph and topological sort takes linear time, and computing $f$ on the topological ordering takes $O(|E|)$ time).

We also show more formally that our algorithm is correct. We begin by proving by induction on $n$, the index in our topological sort of the metagraph, that $f(v)$ is correctly computed as “the maximum possible number of reading list books that we can hit on any path ending in SCC $v$":

**Base case** $n = 0$

In the case $n = 0$, we must be in a source node of the metagraph. Since source nodes have no parents, our algorithm will compute $f(v)$ to be $n_v$, the number of books contained in that SCC. This is correct, since if there are reading list books $b_1, \ldots, b_{n_v}$ in SCC $v$, and since $b_{i+1}$ is reachable by some path from $b_i$ by the definition of an SCC, we can create a path $b_1, \ldots, b_{n_v}$ in that SCC by placing these smaller paths end-to-end, and this path has $n_v$ books. It is not possible to construct a path containing more than $n_v$ books ending in this SCC, since such a path would require $v$ to either contain more than $n_v$ reading books, contradicting the number of reading books in the SCC, or to have edges leading into this SCC from other SCCs, which contradicts $v$ being a source.

**Inductive step**

We assume that for $k < n$, our algorithm has correctly computed $f(v_k)$. Then the by the topological ordering, each of $v_n$’s parents $p_1, \ldots, p_m$ have been correctly computed. Any path ending in $p_i$ can be extended to a path ending in $v_n$ by crossing any path from $p_i$ to $v_n$. This path can be adjoined to a path that crosses through $n_v$ additional reading books by a similar argument to the base case. Thus we can achieve a path ending in SCC $v_n$ of reading-book-count max $f(p_i) + n_v$. If a path ending in $v_n$ had more than this number of books, then it must either begin in $v_n$ (which is impossible by a similar argument as in the base case) or pass through one of $v_n$’s parents. But if such a path has more than max $f(p_i) + n_v$ books, this contradicts the maximality of $f(p_i)$ (implied by the inductive hypothesis), since we can cut the path before $n_v$ and construct a path ending in $p_i$ that has more than $f(p_i)$ books.

Thus $f(v)$ correctly gives the maximum possible number of reading list books that can be hit on any path ending in SCC $v$, and the inductive proof also shows that this path can be constructed. If such a path has length $N$, then, we can construct a path of this length, which corresponds to a series of trades that Chris can make to read every book on his reading list. Conversely, if there is a path of length $N$ in the graph, it must end at some vertex $v$, and $f(v)$ must’ve been correctly computed by our algorithm. Therefore for some $v$, $f(v) = N$. So our algorithm returns TRUE if and only if there is some path in the original graph that runs through all $N$ reading list books.

**Question 3** (Shortest Paths with Bounded Edge Weights, 25 points). Let $G$ be a graph where each edge is assigned an integer weight between 1 and $M$. Give an algorithm that given two vertices $s$ and $t$ in $G$ computes the length of the shortest $s - t$ path in time $O(|V| \log(M) + |E|)$.

**Hint:** You will want to use some variation of Dijkstra’s algorithm but might need to change the way you implement the priority queue.

**Solution 3.** The key to understanding why bounding the edge weights gives a faster runtime is to realise that at any given point in time, the priority queue can have at most $M$ different values in it. This is because if we visit a vertex at distance $d$ ($d > M$) from the starting point, we must have already visited all vertices of distance $d - M - 1$. We can prove this by contradiction: if there are two vertices $u$ and $v$ in the priority queue at the same time such that $v$ has distance $d$ and $u$ has distance $d - M - 1$, then $v$ must’ve been last enqueued by a vertex with distance at least $d - M$ (since edges are of length at most $M$). But since Dijkstra visits vertices in order of distance, it would’ve visited all vertices of distance $d - M - 1$ first, and thus we reach a contradiction.

Next, we must choose a structure to hold our data that takes advantage of the fact that it only needs
to hold $M$ values. We choose to use a Fibonacci heap as a black box which holds distances, and keep a hash map that maps distances to sets of vertices that have that distance. Below, we show a diagram of this data structure and give a description of its operations.

![Diagram of Fibonacci Heap and Hashmap](image)

- **To insert** a vertex, we insert it into its distance entry in the hashmap, and if that distance has no other vertices in it, we also insert the same distance into the priority queue. Since insertion into both data structures takes constant time, this operation takes $O(1)$ time.

- **To decreasekey** on a vertex, we move it to a different set in the hashmap (finding the vertex in the graph, deleting it from a set, and inserting it into a different set can all be done in constant time) and also change the Fibonacci heap accordingly (call the Fibonacci heap’s decreasekey if the new entry exists, or the Fibonacci heap’s insert if it doesn’t) if this operation empties the old distance entry in the hashmap. Since all of the mentioned operations take constant time in their respective data structures, this operation takes $O(1)$ time.

  **Note:** There was also an unforeseen technicality with the decreasekey operation which may cause it to cost $O(\log M)$ time using only the implementation details above, but students were not required to correctly identify and fix this issue. We detail this issue and a possible solution here, for those interested.

If decreasekey moves the last item in a bin into another bin, you cannot just delete this bin (unless you’re willing to incur $O(\log M)$ for the deletemin operation).

The solution to this is for the old bin (now empty) to remain in the priority queue, but for it to keep track of the new bin of the last element it deleted. Then, it continues to keep this information until one of the following happens:

1. A new vertex is enqueued with the old distance value. Then the old bin is no longer empty, and can stop tracking and continue as usual. This takes $O(1)$ time.

2. A vertex with a tracked value has its key decreased. Then one of the orphaned bins tracking that value changes its tracked value to the new key value. There are a few things that might happen in this case:
   - If there isn’t a bin already at that new key value, the orphaned bin’s key is decreased to the new key value, and it can stop tracking.
   - If there is already an existing bin at that new key value, the orphaned node starts tracking the new key value instead.
   - If the new key value contains an orphaned bin, the vertex fills that orphaned bin, and the tracking bin begins tracking the value that orphaned bin was formerly tracking. In each of these cases, we either do $O(1)$ bookkeeping, or use the Fibonacci heap’s $O(1)$ decreasekey.

3. A vertex with a tracked value is removed by deletemin. In this case, one of the bins tracking that vertex’s value is removed from the heap (by decreasing key to $-\infty$ and then applying deletemin). This takes $O(\log M)$ time, as it uses one decreasekey and one deletemin.

Why does this work? It isn’t difficult to show that if a key value has $m$ vertices in it, there are never more than $m - 1$ bins tracking it. Thus every orphaned bin is only ever tracking values smaller than it (so deletemin cannot ever return an orphaned bin). Additionally, since nearly all of the operations
defined here take $O(1)$ time, and we only Fibonacci-heap-deletemin an orphaned bin 0 or 1 extra times when a deletemin operation happens in the larger data structure, these changes do not incur any extra asymptotic cost.

- To deletemin, we deletemin from the heap and delete any vertex that has that same distance value in the hashmap (and erase the entry of the hashmap if it’s the last vertex that has that distance). We also reinsert the min into the heap if the hashmap bin for that distance value is not empty. The hashmap operations take $O(1)$ time and deletemin in a Fibonacci heap takes $O(\log n)$ time (where $n$ is the number of entries in the heap). Since we use the hashmap to hold the vertices at each distance and only hold the distinct distances in the Fibonacci heap (of which there are at most $M$, as shown above), this operation only incurs $O(\log M)$ time.

The operations in this combined data structure all take the same amount of time as a vanilla Fibonacci heap, but deletemin only takes $O(\log M)$ time rather than $O(\log |V|)$ time, and thus we use $O(|V| \log M + |E|)$ total. Correctness follows from the correctness of Dijkstra’s algorithm as well as the fact that the new priority queue we designed will behave the same as a standard priority queue.

**Question 4** (Finding a Restaurant, 20 points). *Sveta is hungry and needs to find a restaurant. Unfortunately, all of the nearby ones have wait times and do not take reservations. She has a graph $G$ of her city with edge weights denoting the time it takes to traverse that edge. Several vertices are marked as restaurants each with the amount of time that Sveta would need to wait after arriving these before getting her food. Given this information, produce an algorithm that for each vertex $v$ of $G$ computes the minimum time Sveta would need to spend starting at $v$ before she could get food. For full credit, your algorithm should run in time $O(|V| \log(|V|) + |E|)$ or better.*

**Solution 4.** We can create a new vertex $e$ in the graph that represents Sveta eating. This vertex is connected to every restaurant in the graph by an edge whose length is the same as the wait time for that restaurant. Then, since our desired solution is a labelling of the vertices representing the amount of time needed till Sveta gets food, we can run Dijkstra’s algorithm backwards from $e$, labelling each vertex with the distance from $e$ given by the algorithm.

The runtime of this algorithm is $O(|V| \log |V| + |E|)$ since we are running Dijkstra, and our changes to the graph only cost linear time. Correctness follows from the property that Dijkstra will find the shortest path in the graph from $e$ to any point $v$ in the city, and this path will represent the wait time at the restaurant (since it must go through one of our new edges) plus the time it took to get to that restaurant from $v$.

**Question 5** (Extra credit, 1 point). *Approximately how much time did you spend working on this homework?*