CSE 101 Homework 2 Solutions

Winter 2015

This homework is due Friday January 30th at the start of class. Remember to justify your work even if the problem does not explicitly say so. Writing your solutions in \LaTeX is recommended though not required.

**Question 1** (Differing Priority Queue Implementations for Dijkstra, 20 points). Consider running Dijkstra’s algorithm on the following graphs, implementing the priority queue as either an array or as a binary heap. Which one is more efficient, and what is the final runtime?

(a) A graph where \( V \) is a \( \sqrt{n} \times \sqrt{n} \) grid and so that there are edges between any vertex and its four neighbors. [5 points]

(b) A complete graph on \( n \) vertices (there is an edge between every pair of vertices). [5 points]

(c) A graph where \( V \) is a \( \sqrt{n} \times \sqrt{n} \) grid and there are edges between any pair of vertices in the same row or column. For this part, consider implementation of a priority queue using a \( d \)-ary heap for various values of \( d \). Which value of \( d \) gives the best runtime, and what is that runtime? [10 points]

**Solution 1.** In this problem we will investigate the efficiency of different priority queues depending on the graph. The runtime of Dijkstra’s algorithm for an array implementation is \( O(|V|^2) \) while that for a binary heap is \( O((|V| + |E|) \log(V)) \)

(a) Each vertex in this graph has a degree of at most 4. Thus, the total number of edges in the graph is at most \( 2n \).

\[ |V| = n, \quad |E| = O(n) \]

(i) **Array**: The runtime for this implementation would be \( O(|V|^2) = O(n^2) \)

(ii) **Binary Heap**: The runtime for this implementation would be \( O((|V| + |E|) \log(V)) = O(n \log(n)) \)

Thus, a binary heap implementation would be more efficient.

(b) In a complete graph, each vertex has a degree of \( n - 1 \). Thus, the total number of edges is \( |E| = \frac{n(n-1)}{2} \).

\[ |V| = n, \quad |E| = O(n^2) \]

(i) **Array**: The runtime for this implementation would be \( O(|V|^2) = O(n^2) \)

(ii) **Binary Heap**: The runtime for this implementation would be \( O((|V| + |E|) \log(V)) = O(n^2 \log(n)) \)

Thus, an array implementation would be more efficient.

(c) Each vertex is adjacent to every other vertex in the same row or column. Thus, the degree of each vertex is \( 2\sqrt{n} - 2 \). Thus, the total number of edges in the graph is \( n(\sqrt{n} - 1) \).

\[ |V| = n, \quad |E| = O(n^{3/2}) \]

(i) **Array**: The runtime for this implementation would be \( O(|V|^2) = O(n^2) \)

(ii) **d-ary Heap**: The runtime for each operation when using a \( d \)-ary heap is as follows:

Insert, decreasing priority, increasing priority: \( O(\log(n)/\log(d)) \)

Extract min: \( O(d \log(n)/\log(d)) \)

In Dijkstra’s algorithm, we perform the extract min operation at most \( n \) times. We do the insert/decrease key operation only when we find edges which cause relaxation. In the worst case,
every edge may cause a relaxation and hence we have to do at most $O(|E|)$ insert/decrease key operations.

Thus, we get a runtime of $O(nd\log(n)/\log(d) + n^{3/2}\log(n)/\log(d))$

Clearly, it is not possible to get a runtime better than $O(n^{3/2})$ because of the second term and because $d < n$. When $d = n^{1/2}$, we get a runtime of $O(n^{3/2}\log(n)/\log(n^{1/2}) + n^{3/2}\log(n)/\log(n^{1/2})) = O(n^{3/2})$.

Thus, a d-ary heap implementation with $d = n^{1/2}$ would be more efficient.

**Question 2** (Shortest Paths to Nearby Vertices, 30 points). Find an algorithm to do the following: Given a graph $G$ with positive integer edge weights, a vertex $s$ and a positive integer $L$ find the set of vertices within distance $L$ of $s$. Your algorithm should run in time $O(|V| + |E| + L)$. Hint: Modify Dijkstra’s algorithm to use an array whose $i^{th}$ entry holds a list of all vertices at distance $i$ for each $0 \leq i \leq L$. [Algorithm 10 points, Analysis 10 points, Runtime analysis 10 points]

**Solution 2.** The bottleneck in the array implementation of Dijkstra’s algorithm is finding the element with the minimum key. Therefore, to achieve linear time we avoid “searching” for the minimum element at every round.

To compute the set of vertices within distance $L$ of $s$, we maintain an array $\text{dist}_s$ where $\text{dist}_s[i]$ is a linked list containing vertices at distance $i$ from $s$. Our array will be of size $L + 1$ since we are only interested in distances less than or equal to $L$. We will have temporary distance labels initially on all the vertices of the graph corresponding to the “unseen” portion of the graph. As vertices are added to the “seen” portion, their labels will be made permanent. Therefore, we actually maintain two copies of $\text{dist}_s$, one for the unseen portion and one for the seen portion.

The key to this implementation is that the as-yet-unseen minimum distances are monotone increasing, since any path to a vertex $u$ must be at least as long as the paths to all vertices preceding $u$, and these are necessarily discovered first. Thus, rather than computing the minimum distance at every iteration of the algorithm, we simply maintain walk along the $\text{dist}_s$ array. Below is the full algorithm:

**Algorithm NearbyVertices(G, s, L):**

// input: G, a vertex s, a positive integer distance L
// output: a list of vertices within distance L of s

initialize two arrays, $\text{dist}_s$ and $\text{dist}_s_{\text{TEMP}}$, of size $L+1$
initialize array dist of size $n$
dist[s] = 0
$\text{dist}_s_{\text{TEMP}}[0] = s$

curr_min = 0 // pointer to minimum distance
while curr_min < L+1:
    while $\text{dist}_s_{\text{TEMP}}[\text{curr_min}]$ is not empty:
        u = first vertex in $\text{dist}_s_{\text{TEMP}}[\text{curr_min}]
        // update distances
        for w adjacent to u:
            if dist[w] > dist[u] + l(u,w):
                if w in $\text{dist}_s_{\text{TEMP}}$:
                    remove w from $\text{dist}_s_{\text{TEMP}}$
                    dist[w] = dist[u] + l(u,w)
                if dist[w] < L+1:
                    add w to $\text{dist}_s_{\text{TEMP}}[\text{dist}[w]]$
        // add u to the seen portion of the graph
        add u to $\text{dist}_s[\text{dist}[u]]$
    curr_min += 1
return all vertices in $\text{dist}_s$
We show the algorithm correctly computes minimum distances by analogy to Dijkstra. Namely, the array \( \text{dist}_{\text{from s TEMP}} \) acts as a priority queue for as-yet-unseen vertices in the graph. This is true because any unseen vertices cannot have their distances updated to be less than the value of \( \text{curr min} \) since edge lengths are all positive. Therefore, after one round of updates, the minimum distance will be the index of the first non-empty array element in \( \text{dist}_{\text{from s TEMP}} \).

The initialization of \( \text{dist}_{\text{from s}} \) and \( \text{dist}_{\text{from s TEMP}} \) each take time \( O(L) \), and the initialization of \( \text{dist} \) is \( O(|V|) \). As discussed above, finding minimum elements involves one linear scan of the array, taking time \( O(L) \), and updating distances takes a total of \( O(|E|) \). Therefore the total running time is \( O(|V|+|E|+L) \), as desired.

Question 3 (Commodity Trading, 50 points). Charlene is a commodities trader. She trades in \( n \) different types of goods. She knows \( m \) other merchants, each of which are willing to trade one specific good for one other good at a specified exchange rate. Say that the \( i \)th merchant is willing to trade one unit of good \( g_i \) for \( r_i \) units of good \( g'_i \).

(a) Given two specific goods, find an efficient algorithm by which Charlene can find a sequence of trades to exchange the first type for the second at the most favorable possible rate. Hint: Adapt Bellman-Ford. [Algorithm 5 points, Analysis 5 points]

(b) In some circumstances, it might be possible for Charlene to make some sequence of trades and eventually end up with strictly more of a good than she started with. Give an algorithm to determine whether or not this is possible. [Algorithm 5 points, Analysis 5 points]

(c) One way in which you might be able to show that the above is impossible is if there is a way to assign prices to every good in such a way that no merchant allows you to trade some collection of goods for a more valuable collection. Find a mathematical formulation of this condition [5 points] and show that it implies that it rules out sequences of trades that would allow Charlene to end up with more than she started with. [10 points]

(d) In fact, if the situation discussed in part (b) is impossible, it is always possible to assign prices as described in part (c). Show that this is the case. You may assume that given any two goods, there is some sequence of trades that allows you to exchange one for the other. Hint: pick some particular good \( g \) and find the best ways of exchanging it for each other type of good. Set prices so that all of these trades are cost-neutral. [15 points]

Solution 3 (Commodity Trading).

(a) We first abstract the market as a graph. Each commodity is a node, and each merchant is a directed edge \((u, v)\), where \( u \) is the commodity that the merchant wants to buy, and \( v \) is the commodity they're selling. The weight of edge \((u, v)\) represents the amount of \( u \) they charge for one unit of \( v \).

With this abstraction, we can adapt the Bellman-Ford algorithm so that each commodity keeps track of the most favorable rate at which it can be exchanged with all of the other commodities. Each commodity initially knows its exchange rates with its neighbors, and a rate of 1 with itself. On each round of the algorithm, each commodity updates its knowledge by making use of the knowledge of its reachable neighbors. Consider edge \((u, v)\) with rate \( r_{u,v} \) and commodity \( w \). Suppose \( v \) knows that it can be exchanged with \( w \) at rate \( r_{v,w} \). Then \( u \) knows it can be exchanged with \( w \) at rate \( r_{u,w} = r_{u,v} \times r_{v,w} \). In this way, commodity \( u \) updates its most favorable (smallest) known rate at which it can be exchanged with all other commodities through its neighbors.

The only modification necessary to the Bellman-Ford algorithm is the method by which each node updates its most favorable rate. Instead of adding the edge weight to the neighbor’s value, they multiply with each other, since this operation reflects the multiplicative nature of composite trading rates. Another approach is to set the edge weight to the logarithm of the merchant’s exchange rate, in which case no modification of BF is necessary at all, since multiplying the rates is the same as adding their logarithms:

\[
\log(r_x \cdot r_y) = \log(r_x) + \log(r_y)
\]
The number of multiplications (or additions) and comparisons does not change — $O(n \cdot m)$. Note that since the logarithms of the rates can be negative (or equivalently because some rates can be less than 1), we need to use Bellman-Ford rather than Dijkstra’s algorithm.

(b) We are looking for a commodity which finds a rate of exchange with itself which is less than 1; $r_{u,u} < 1$. In the logarithm-edge-weight formulation, this would be exactly the same as finding a negative cycle in a graph; $r_{u,u} < 1 \Rightarrow \log(r_{u,u}) < 0$. The Bellman-Ford algorithm knows that such a cycle exists if an update occurs at round $n + 1$, at which point we can look at each commodity and find out which one has found a favorable rate of exchange with itself. Again, the runtime for this algorithm is the same as BF, $O(n \cdot m)$.

(c) Each merchant wants to trade with Charlene so that the monetary value of their goods after the trade is greater than or equal to their value before the trade:

$$m_2 \geq m_1$$

Suppose merchant $i$ is selling $v$ for $u$. He sells $x$ units of $v$ for $r_i \cdot x$ units of $u$. The monetary value of $x$ units of a commodity $y$ is $x \cdot p_y$, so the above condition becomes:

$$m_2 = r_i \cdot x \cdot p_u \geq x \cdot p_v = m_1$$

Dividing out $x$ and rearranging to solve for $r_i$, we have:

$$r_i \geq \frac{p_v}{p_u}$$

Let us examine the implications of this. Suppose Charlene trades with one merchant between goods $u$ and $v$ at rate $r_1$, and then with another merchant between goods $v$ and $w$ at rate $r_2$. Then the composite rate of the exchange is $r_c = r_1 \cdot r_2$. We observe the following result:

$$r_1 \geq \frac{p_v}{p_u} \text{ and } r_2 \geq \frac{p_w}{p_v} \Rightarrow r_c = r_1 \cdot r_2 \geq \frac{p_v}{p_u} \cdot \frac{p_w}{p_v} = \frac{p_w}{p_u}$$

Thus, assuming all merchants trade to their advantage, the composite rate of exchange between two goods along any path must be greater than the ratio of their prices.

Thus, if Charlene makes any series of exchanges between good $g$ and itself, we get the following result:

$$r_c \geq \frac{p_g}{p_g} = 1$$

Thus, if there exist prices $p_1, p_2, \ldots, p_n$ for each commodity such that, for all merchants $i$ who trade commodities $(u_i, v_i)$ at rate $r_i$, $r_i \geq \frac{p_{v_i}}{p_{u_i}}$, then the best possible exchange rate between any good and itself, $\min_{g,g}$, must be greater than 1.

(d) Choose one good arbitrarily, $c$, as the “currency” for the market, such that one unit of that good is worth 1 milli-bitcoin (mBTC). One can then set the price of all goods in the rest of the market relative to the currency. For instance, to assign a price to good $g$, find an exchange path from $c$ to $g$; find the composite rate, $r_{c,g}$ of exchange between the currency and the other good along that path (that is, the product of the rates on each edge traversed, or the sum of their logarithms); and set the price $p_g = r_{c,g} \cdot p_c$.

Obviously, it is possible to find multiple paths from $c$ to $g$, or from any two goods. Suppose one path has composite rate $r_1$, and another has rate $r_2 > r_1$. If we follow one path, then we set $p_g = r_1 \cdot p_c$, and by the other path we set $p_g = r_2 \cdot p_c$. It turns out that in order for all merchants to trade to their benefit, we need to set $p_g$ according to the smallest possible rate $r_{c,g}$. To see this, suppose that we instead set the price according to the larger rate. Then we have:

$$p_g = r_2 \cdot p_c$$
Rearranging the equation, and taking into consideration the earlier supposition that \( r_1 < r_2 \), we have

\[
\frac{r_1}{r_2} = \frac{p_g}{p_c}
\]

So if we set prices according to an exchange rate which is NOT the smallest, then there is at least one merchant that trades to their own disadvantage. If on the other hand we set rates according to the smallest possible exchange rate, then we have the opposite situation:

\[
\frac{r_2}{r_1} = \frac{p_g}{p_c}
\]

Which is acceptable for all merchants.

Thus, our price setting algorithm is as follows: pick a currency \( c \) with arbitrary price \( p_c \); calculate the smallest possible exchange rate between the currency \( c \) and all other goods, \( r_{\text{min}}_{c,g} \), by adapting BF algorithm as we did in part A (this will be possible since, by assumption, the situation in part B is impossible); and set the price of good \( g \) to \( p_g = r_{\text{min}}_{c,g} \cdot p_c \).

We need to verify that this is an appropriate pricing scheme. Given the condition from part (c), this means that we need to show that for each merchant willing to trade \( g \) for \( g' \) at rate \( r \) that

\[
r \geq \frac{p_g}{p_{g'}} = \frac{r_{\text{min}}_{c,g}}{r_{\text{min}}_{c,g'}}
\]

This must be the case though since Chalene has a sequence of trades getting her from \( c \) to \( g \) by first making the optimal sequence of trades to turn \( c \) into \( g' \) and then using this merchant to exchange \( g' \) for \( g \). This gives an exchange rate of \( r \cdot r_{\text{min}}_{c,g'} \), which therefore, must be at least \( r_{\text{min}}_{c,g} \).