CSCE629 Analysis of Algorithms
Homework 6

Texas A&M U, Fall 2019 — *10/07/19*
Lecturer: Fang Song — *Due: 10am, 10/18/19*

Instructions.

- Typeset your submission by LaTeX, and submit in PDF format. Your solutions will be graded on *correctness* and *clarity*. You should only submit work that you believe to be correct, and you will get significantly more partial credit if you clearly identify the gap(s) in your solution. You may opt for the "I'll take 15%" option.
- You may collaborate with others on this problem set. However, you must **write up your own solutions** and **list your collaborators and any external sources** for each problem. Be ready to explain your solutions orally to a course staff if asked.
- For problems that require you to provide an algorithm, you must give a precise description of the algorithm, together with a proof of correctness and an analysis of its running time. You may use algorithms from class as subroutines. You may also use any facts that we proved in class or from the book.
- **If you describe a Greedy algorithm, you will get no credit without a formal proof of correctness, even if your algorithm is correct.**

This assignment contains 3 questions, 5 pages for the total of 60 points and 0 bonus points. A random subset of the problems will be graded.

1. (15 points) (Print neatly) Consider the problem of neatly printing a paragraph with a mono-spaced font (all characters having the same width) on a printer. The input text is a sequence of n words of length $\ell_1, \ell_2, \ldots, \ell_n$ measured in characters. We want to print this paragraph neatly on a number of lines that hold a maximum of M characters each. Our criterion of "neatness" is as follows.

   If a given line contains words i through j, where $i \leq j$, and we leave exactly one space between words, the number of extra space characters at the end of line is $M - j + i - \sum_{k=i}^{j} \ell_k$, which must be non-negative to fit the words on the line. We wish to *minimize* the sum, over all lines except the last. Describe and analyze an algorithm (both time and space) to print a paragraph of n words neatly.

   Solution: First observe that the problem exhibits optimal substructure in the following way: Suppose we know that an optimal solution has k words on the first line. Then we must solve the subproblem of printing neatly words $l_{k+1}, \ldots, l_n$. We build a table of optimal solutions solutions to solve the problem using dynamic programming. If $n - 1 + \sum_{k=1}^{n} l_k < M$ then put all words on a single line for an optimal solution. If $OPT[i]$ represent the optimal solution for starting line with word i then we can write,

$$OPT[i] = min_{i+1 \le j \le n}\{M - j + i - \sum_{k=i+1}^{j} l_k + OPT[j]\}$$

In the following algorithm $PrintingNeatly(n)$, $C[k]$ contains the cost of printing neatly words l_k through l_n. We can determine the cost of an optimal solution upon termination by examining $C[1]$. The entry $P[k]$ contains the position of the last word which should appear on the first line of the optimal solution of words $l_1, l_2, \ldots, l_n$.

```
Algorithm 1 PrintingNeatly(n)
Initialize arrays $P[1, \ldots, n]$ to $NIL$ and $C[1, \ldots, n]$ to $\infty$
for $i = n$ to 1
  if $\sum_{k=i}^{n} l_k + n - k < M$
    $C[i] = 0$
  $q \leftarrow \infty$
  for $j = i + 1$ to $n$
    if $\sum_{m=i}^{j} l_m + m - i < M$ and $M - j + i - \sum_{m=i}^{j} l_m + C[i + j] < q$
      $q = M - j + i - \sum_{m=i}^{j} l_m + C[i + j]$
      $P[i] = i + j$
  $C[i] = q$
```

Runtime for this algorithm will be $O(n^2)$, and we need to store two arrays of size n hence space requirement will be of the order $O(n)$. Array P will gives us the values that will tell us where to break the sequence and print the paragraph.

2. (Shortest path with bounded negative edges) Suppose we are given a directed graph G with weighted edges and two vertices s and t.

   (a) (10 points) Describe and analyze an algorithm to find the shortest path from s to t when exactly one edge in G has negative weight.

   (b) (15 points) Describe and analyze an algorithm to find the shortest path from s to t when exactly k edges in G have negative weights. How does the running time of your algorithm depend on k?

Solution:

(a) Let G denote the input graph, let $w(x \rightarrow y)$ denote the weight of the edge $x \rightarrow y$, and let $u \rightarrow v$ denote the unique edge in G with negative weight. Remove edge $u \rightarrow v$ from G and let $G^{'}$ denote the resulting graph. Note that $G^{'}$ has no negative length edges. For any nodes x and y, let $dist(x, y)$ and $dist^{'}(x, y)$ denote the distances from x to y in G and $G^{'}$, respectively.

First we will check if the graph G has negative length cycle. If G has negative length cycle then the cycle must contain the only negative edge $u \rightarrow v$, and that cycle must pass through this edge. This implies that there is a path from v to u, which has only positive weights. This gives us a way to check for negative length cycles. If G has a negative length cycle then it must contain the edge $u \rightarrow v$: the shortest length cycle containing this arc can be seen to consist of a shortest path P from v to u in $G^{'}$ together with the arc $u \rightarrow v$. The length of this cycle is $dist^{'}(v, u) + w(u \rightarrow v)$. G has a negative length cycle iff this quantity is negative. Thus, we can check if G has a negative length cycle by computing $dist^{'}(v, u)$ in $G^{'}$ via Dijkstra's algorithm.

Suppose G does not have a negative length cycle. The shortest path in G from s to t either traverses the edge $u \rightarrow v$ or it doesn't; we consider each case separately. Then we have

$$dist(s, t) = min \left\{ dist^{'}(s, t), dist^{'}(s, u) + w(u \rightarrow v) + dist^{'}(v, t) \right\}.$$

Thus, we can compute $dist(s, t)$ by running Dijkstra twice in $G^{'}$: once starting at s to compute both $dist^{'}(s, t)$ and $dist^{'}(s, u)$, and once starting from v to compute $dist^{'}(v, t)$. The algorithm runs in $O(E \log V)$ time because, when the graph is connected then running time $O((E + V) \log V)$ can be reduced to $O(E \log V)$ source.

(b) We will develop a similar algorithm like above but we will also use Bellman-Ford algorithm in combination with Dijkstra's algorithm. Key idea is here we will reduce the problem into a problem that we know how to solve. First we will remove all the negative edges from the graph and create new graph. All the definitions that are explained will be used here. After we get new graph $G^{'}$ we will create another graph such that has only following vertices $s, u_1, v_1, u_2, v_2, \ldots, u_k, v_k, t$. Where s, t are source and destination and u_i, v_i are endpoints/vertices of the negative edges, lets call this graph $G^{''}$. Here an edge between two vertices x and y will be represented by the length of the shortest path from x and y with no negative edges. To get these values we will apply Dijkstar's algorithm in graph $G^{'}$ for all the vertices in graph of

$G^{''}$. Then we will add back all the negative edges to graph $G^{''}$ and while doing so if edge $x \to y$ has more weight than the actual edge present between them in original graph (assuming it is present, if not do nothing) G, i.e. $w^{''}(x \to y) > w(x \to y)$ then we will replace the $w^{''}$ with w. Also, note that by construction $w^{''}(x \to y) = dist^{'}(x \to y)$.

Now we have graph $G^{''}$ with reduced edges and with negative weights. If we apply Bellman-Ford Algorithm on this starting from s we will get shortest distance from s to t. The running time of this algorithm will be $O(k(E+V)\log V + k^3)$. We will apply Dijkstra's algorithm on graph $G^{'}$ for $O(k)$ times hence we get $O(k(E+V)\log V)$ and we apply Bellman-Ford algorithm on graph $G^{''}$ which has $O(k)$ vertices and at most k^2 edges, so it takes $O(k \times k^2)$ running time.

3. (Arbitrage) Arbitrage is the use of discrepancies in currency exchange rates to transform one unit of a currency into more than one unit of the same currency. For example (exchange rates not up to date), suppose 1 US dollar buys 71 Indian rupees, 1 Indian rupee buys 1.6 Japanese yen, and 1 Japanese yen buys 0.0093 US dollars. Then by converting currencies, a trader can start with 1 US dollar and buy $71 \times 1.6 \times 0.0093 = 1.0565$ US dollars, thus making a profit of 5.65 percent.

   Suppose that you are given n currencies $c_1, c_2, \ldots, c_n$ and an $n \times n$ table R of exchange rates, such that one unit of currency c_i buys $R[i,j]$ units of currency j.

   (a) (10 points) Describe and analyze an algorithm to determine whether or not there exists a sequence of currencies $\langle c_{i_1}, \ldots, c_{i_k} \rangle$ such that

   $$R[i_1,i_2] \cdot R[i_2,i_3] \cdots R[i_{k-1},i_k] \cdot R[i_k,i_1] > 1\,.$$

   (b) (10 points) Describe and analyze an algorithm to print out such a sequence if one exists.

Solution:

(a) To do this we take the negative of the log of all the values c_i that are on the edges between the currencies. Then, we detect the presence or absence of a negative weight cycle by applying Bellman-Ford. To see that the existence of an arbitrage situation is equivalent to there being a negative weight cycle in the original graph, consider the following:

$$R[i_1,i_2] \cdot R[i_2,i_3] \ldots R[i_k,i_1] > 1$$
$$\log(R[i_1,i_2]) + \log(R[i_2,i_3]) + \ldots + \log(R[i_k,i_1]) > 0$$
$$-\log(R[i_1,i_2]) - \log(R[i_2,i_3]) - \ldots - \log(R[i_k,i_1]) < 0$$

(b) To do this, we first perform the same modification of all the edge weights as mentioned above. Then, we wish to detect the negative weight cycle. To do this, we relax all the edges $|V| - 1$ many times, as in Bellman-Ford algorithm. Then, we record all of the d values of the vertices. Then, we relax all the edges $|V|$ more times. Then, we check to see which vertices had their d value decrease since we recorded them. All of these vertices must lie on some (possibly disjoint) set of negative weight cycles. Call S this set of vertices. To find one of these cycles in particular, we can pick any vertex in S and greedily keep picking any vertex that it has an edge to that is also in S. Then, we just keep an eye out for a repeat. This finds us our cycle. We know that we will never get to a dead end in this process because the set S consists of vertices that are in some union of cycles, and so every vertex has out degree at least 1.