Solutions to Homework 2 Practice Problem

[DPV] 4.11 (length of shortest cycle on a graph)

Solution: As presented, we are given a ***directed graph*** with ***positive edge weights***. These conditions assure that there is no negative-weight cycle within the graph. How do we find the length of the shortest cycle? Note that the shortest cycle visiting vertices u and v has length $d(u, v) + d(v, u)$, hence the answer we are looking for is

$$\min_{u,v}(d(u, v) + d(v, u))$$

Notice that if the quantity above is equal to infinity it means the graph is acyclic, and we report that.

Our approach: run `Floyd-Warshall` on the given graph to get an array $\{d(u, v)\}_{u,v \in V}$ of the distances between any pair of points. It is assumed that at the beginning any (u, v) pair which is not represented by an edge $[(u, v) \notin E]$ is set $d(u, v) = \infty$. Now, inspect the diagonal in the array and our shortest path is the $\min(d(v, v))$ across all vertex $v \in V$

To see why this algorithm is correct: note first that every cycle visiting vertices u and v can be broke into two directed paths: from u to v and from v back to u. Each such path achieves its minimum length at $d(u, v)$ and $d(v, u)$ respectively, hence the cycle of minimal length is given by this sum.

`Floyd-Warshall` takes $O(|V|^3)$ and the minimum can be found in $O(|V|)$, leading to an $O(|V|^3)$ runtime.

[DPV] 4.21 (Currency trading)

Solution: We are presented with a directed graph problem, so hopefully we can use a known algorithm as the basis for our solution. The key is to recognize two things: (a) the currency calculation is a product (b) we want to maximize that product. We solve for the former by converting the exchange rates to logs (recalling that $\log(a) + \log(b) = \log(a \times b)$), and solve for the latter by using the negative of the log for our edge weight (flipping maximization to minimization).

So, for part (a), we create a graph where the vertices represent countries, and the edges between them have a weight $w_{i,j} = -\log r_{i,j}$. We then run Bellman-Ford from vertex s to vertex t, and the minimal weight path represents the most advantageous sequence of currency trades. This takes $O(|V||E|)$ time.

Why does this work? By converting the exchange rates to logs and then negating that value the weight of the shortest path (the sum of the negated log values) will represent the series of exchanges which maximizes the product of the exchange rates. Note that since edge weights are negative we cannot use Dijkstra's.

For part (b), we run one more iteration of Bellman-Ford, and if any of the distances (weights) change, we have detected a negative cycle – such a cycle represents the trading anomaly which allows for infinite profit.

[DPV] Problem 2.1 – Practice Multiplication

The product is 28,830. See ***2.1 Practice Multiplication.pdf*** for the value calculated at each level of the recursive stack.

[DPV] Problem 2.5 – Recurrence

Solution:
(a) $T(n) = 2T(n/3) + 1 = O(n^{\log_3 2})$ by the Master theorem

(b) $T(n) = 5T(n/4) + n = O(n^{\log_4 5})$ by the Master theorem

(c) $T(n) = 7T(n/7) + n = O(n \log_7 n) = O(n \log n)$ by the Master theorem

(d) $T(n) = 9T(n/3) + n^2 = O(n^2 \log_3 n) = O(n^2 \log n)$ by the Master theorem

(e) $T(n) = 8T(n/2) + n^3 = O(n^3 \log_2 n) = O(n^3 \log n)$ by the Master theorem

(f) $T(n) = 49T(n/25) + n^{3/2} \log n = O(n^{3/2} \log n)$
Hint: the contribution of level i of the recursion is $(\frac{49}{125})^i O(n^{3/2} \log n)$.

(g) $T(n) = T(n-1) + 2 = O(n)$

This can be found through substitution:

$$\begin{aligned} T(n) &= T(n-1) + 2 \\ T(n) &= (T((n-1)-1) + 2) + 2 \\ T(n) &= T(n-2) + 2 + 2 \end{aligned}$$

This pattern repeats n times until $T(n-n) = T(0)$; on the right we have $2n = O(n)$

(h) $T(n) = T(n-1) + n^c = \sum_{i=1}^{n} i^c + T(0) = O(n^{c+1})$

$$\begin{aligned} T(n) &= T(n-1) + n^c \\ T(n) &= (T((n-1)-1) + (n-1)^c) + n^c \\ & \quad T(n-2) + (n-1)^c + n^c \qquad \textit{we simplify} \\ T(n) &= (T((n-2)-1) + (n-2)^c) + (n-1)^c + n^c \\ & \quad (T((n-3) + (n-2)^c) + (n-1)^c + n^c \end{aligned}$$

This pattern repeats n times until $T(n-n) = T(0)$; the $(n-1)^c$ resolves to a polynomial where n^c is the dominating term. This repeats n times, giving $O(n * n^c) = O(n^{(c+1)})$

(i)

$$T(n) = T(n-1) + c^n = \sum_{i=1}^{n} c^i + T(0) = O(c^n)$$

This can be solved similar to (h); This pattern repeats n times until $T(n-n) = T(0)$; we have a series $c^0 + c^1 + \ldots + c^n$, the last term dominates giving $O(c^n)$

(j)

$$T(n) = 2T(n-1) + 1 = \sum_{i=0}^{n-1} 2^i + 2^n T(0) = O(2^n)$$

(k)

$$T(n) = T(\sqrt{n}) + 1 = \sum_{i=0}^{k} 1 + T(b)$$

where k is an integer such that $n^{\frac{1}{2^k}}$ is a small constant b (the size of the base case). This implies that $k = O(\log \log n)$ and $T(n) = O(\log \log n)$.

[DPV] Binary Search Modified

Design an O(log(n)) algorithm to find the smallest missing natural number in a given sorted array. The given array only has natural numbers. For example, the smallest missing natural number from A = 3, 4, 5 is 1 and from A = 1, 3, 4, 6 is 2.

Solution:

If we have an array of length n we expect the numbers $\{1, 2, 3, \ldots, n\}$ to be in the array. Since, one number is missing this means there is at least one number such that its position does not match its value.

Our approach: Consider mid as the position of the middle element, low as the first element, and $high$ as the last. We first check to see if $A[mid] = mid$. Since the array is sorted, if $A[mid] = mid$ it means there are not numbers missing from $1, 2, \ldots, mid$, so we have to check the right (latter) half of the array; to do this we update $low = mid + 1$. If there was a missing number, it would have been replaced by a bigger number. This means, if $A[mid] > mid$, we have to search the left half; we do this by setting $high = mid$. As you reduce the input by half, track what the value at $A[low]$ value should be (initially it's 1), call that S. If $A[low] \neq S$ then the missing value is S. If they match, then you divide the current input in half based on the value of $A[mid]$, update S and low or $high$ accordingly, and recurse. If you get to a single element and $A[i] = S$ then the missing value is $A[i] + 1$; if not, the missing value is S.

Why this works: our input is sorted in non-decreasing order. If the number sequence begins at 1, then the missing value is the lowest i such that $A[i] \neq i$. When we check if $A[mid] = mid$, if they match we know all the values $1 \ldots mid$ are present. if they don't match, we know some value in the range $1 \ldots mid - 1$ is missing.

Runtime: To check the running time is logarithmic, note that the recurrence relation is $T(n) = T(\frac{n}{2}) + O(1)$ which solves to $O(\log(n))$ by the Master Theorem.