UM-SJTU JOINT INSTITUTE

Introduction to Algorithms
(VE477)

Homework #5

Prof. Manuel

Xinmiao Yu
518021910792

Nov. 1, 2020

Q1.

1. *Linear partition problem:* A given arrangement S consisting n nonnegative integers $s_1, ..., s_n$ and an integer k, partition S into k ranges so as to minimize the maximum sum over all the ranges.

   Often arises in parallel processing, because of the demand to balance the work done across processors so to minimize the total elapsed run time.

2. No, not a good solution. It will not systematically evaluate all the possibilities. Consider $S = \{3, 3, 3, 2, 5, 5\}$ and $k = 3$. With the this approach we have the average size of partition as $21/3 = 7$, then the set will be divided as $3, 3 \mid\mid 3, 2 \mid\mid 5, 5$ with maximum sum as 10. However, the solution should be $3, 3, 3 \mid\mid 2, 5 \mid\mid 5$ with maximum sum as 9.

3. The problem could be transformed into finding the minimum value of the lager one between 1) cost of the last partition $\sum_{j=i+1}^{n} s_j$ and 2) the cost of the largest partition cost formed to the left of i.

$$M(n,k) = min_{i=1}^{n}\ max(M[i,k-1], \sum_{j=i+1}^{n} s_j)$$

$$\text{With Basis M[1, k]} = s_1, \forall k > 0$$

$$M[n,1] = \sum_{i=1}^{n} s_i$$

4. If keep $M[i][j]\ \forall i \le n, j \le k$, total cell will be $k \cdot n$ in this table. For any $M[n'][k']$, need to find the minimum among n' quantities, each of which is the maximum through table lookup and a sum of at most n' elements. Then fill each cell need $\mathcal{O}(n^2)$. So total need $\mathcal{O}(kn^3) = \mathcal{O}(n^3)$

5. When update each cell, instead of selecting the best of up to n possible points to place the divider, each of which need to sum up to n possible terms, we could store the set of n prefixes sum

$$p[i] = \sum_{k=1}^{i} s_k, \text{ since p[i] = p[i − 1] + s}_\text{i}$$

Problem 1 continued on next page. . .

6. Dynamic programming approach

Algorithm 1: Linear Partition Problem

Input : arrangement S consisting n nonnegative integers $s_1, ..., s_n$ and an integer k

Output: the cost of the largest range when partition S into k ranges so as to minimize the maximum sum over all the ranges

```
/* compute prefix sum                                   */
p[0] ← 0
for i ← 1 to n do
    p[i] ← p[i − 1] + s_i
end for
/* boundary condition                                   */
for i ← 1 to n do
    M[i, 1] ← p[i]
end for
for j ← 1 to k do
    M[1, j] ← s_1
end for
/* evaluate main recurrence                             */
for i ← 2 to n do
    for j ← 2 to k do
        M[i, j] ← ∞
        for x ← 1 to i - 1 do
            s ← max(M[x, j − 1], p[i] − p[x])
            if M[i,j] > s then
                M[i, j] ← s
                D[i, j] ← x ;             /* used to reconstruct */
            end if
        end for
    end for
end for
return M[n,k]
```

7. First the prefix sum and boundary condition is obviously true. It settle the smallest possible values for each of the arguments of the recurrence. With the evaluation order such that computes the smaller values before the bigger values, it will obtain the right result as long the previous results are true, which must be true as the boundary conditions are true.

8. When update each cell, we do not need to select the best among n possible points to place the divider because of the prefix sum we stored. So for each call, only need linear time. Then the total time complexity would be $\mathcal{O}(kn^2)$.

9. This could be achieved by D, as it record which divider position required to achieve such cost. So to reconstruct the path used to get to the optimal solution, we work backward from

$D[n, k]$ and add a divider at each specified position.

Algorithm 2: Reconstruct

Input : S, D, n, k

Output: S with divider

```
Function Reconstruct(S, D, n, k):
    if k == 1 then
        print the first partition (s_1, s_2, ..., s_n)
    else
        Reconstruct(S, D, D[n,k], k-1)
        print the k-th partition (s_{D[n,k]+1}, ..., s_n)
    end if
end
```

Q2.

As B would produce number $0, 1, 2, 3, 4$ with equal probability $1/5$, we could get number in range $[0, 24]$ by $B * 5 + B$ with equal probability $1/25$. And if drop the number that larger than 23, the probability for generating number in $[0, 23]$ is $1/24$. Then the number in $[0, 7]$ with equal probability could be obtained by $[0, 23]/3 = 1/8$, which returns the integer part of the result.

To extend the generation procedure, the critical part is to generate numbers larger than expected n ($[0, 24]$ previously) with equal probability.

- Denote the original B that produce $[0, 4]$ as B_0
- Denote that produce $[0, 24]$ as B_1 such that $B_1 = 5 * B_0 + B_0 = (5 + 1)$
- $B_2 = 25 * B_0 + B_1$ produce $[0, 124]$ with equal probability as $1/125$
- $B_3 = 125 * B_0 + B_2$ produce $[0, 624]$ with equal probability $1/625$

Then summarize as

$$Range[B_n] = [0, 5^{n+1} - 1], \text{ with P} = \frac{1}{5^{\text{n+1}}}$$

Therefore, we could use the original B, to have any generator B_i we need to produce number in range $[0, 5^{i+1} - 1]$. Restriction on n will be $n \geq 0$.

As in the previous example, the random number in $[0, 7]$ is obtained through $(B_1.output < 24)/3$. Because the range is $[0, 5^2 - 1] = [0, 24]$, which is too large if we just simply keep the number that $B_1.output \leq 7$. So we could apply the same method that find an integer a such that

$$a * n < 5^{i+1} - 1, \quad where\ 5^i - 1 < n \leq 5^{i+1} - 1$$

The the random number is $B_i.output/a$

Algorithm 3: Random Number Generator

Input : nonnegative integer n

Output: a random number in range $[0, n]$

Find i that $5^i - 1 < n \leq 5^{i+1} - 1$
$a \leftarrow 1$
while $(a+1)*(n+1) \leq 5^{i+1} - 1$ **do**
  $a \leftarrow a + 1$
end while
Get the random number generator B_i
$num \leftarrow B_i.output$
while $num > (n+1)*a$ **do**
  $num \leftarrow B_i.output$
end while
return num/a

Q3.

Bellman-ford algorithm

Algorithm 4: Detect negative cycle

Input : weighted graph $G = (V, E)$

Output: whether the graph has negative cycle

Chosen a vertex s randomly
`/* Initialization */`
for *each vertex* $v \in G.V$ **do**
  $v.d \leftarrow \infty$
end for
$s.d \leftarrow 0$
`/* Relax */`
for $i \leftarrow 1$ **to** $|G.V| - 1$ **do**
  for *each edge* $(u, v) \in G.E$ **do**
    if $v.d > u.d + w(u,v)$ **then**
      $v.d \leftarrow u.d + w(u, v)$
    end if
  end for
end for
for *each edge* $(u, v) \in G.E$ **do**
  if $v.d > u.d + w(u,v)$ **then**
    return *True* ; `/* only possible when negative cycle exists */`
  end if
end for
return *False*

Q4.

Q5.

Denote the position of k internet hostspots as $loc_p = loc_1, loc_2, ..., loc_k$ and the position of n clients as $pos_c = pos_1, pos_2, ..., pos_n$.
Construct a graph with all the clients and hostspots as vertices, all the clients could be viewed as a sink nodes while hostspots as source nodes. If one of clients (t_i) could connect to one of the hostpots (s_j) (whether user in the range parameter r), add an edge with capacity $c(s_j, t_i) = 1$
Then we add a $supersource(s)$ such that add edge with capacity is the load of this hostspot such that $c(s, s_j) = l_j, \ \forall 1 \leq j \leq k$. Add a $supersink(t)$ such that add edge with capacity 1 such that $c(t_i, t) = 1, \ \forall 1 \leq i \leq n$.
Then apply the **Edmonds-Karp** algorithm to this graph, with source node s and sink node t. The maximum flow (n) happens only when all the users connects to the network. Because the supersource node connect each hostpot with *capacity* == *load* and ssuperink node connect each client with capacity 1.

Algorithm 5: Wifi network

Input : r, l, loc_p (location of k hostspots), pos_c (position of n clients)
Output: whether all user could connect to the network

`/* initialize the graph */`
Empty graph $G \leftarrow (V, E)$
for $i \leftarrow 1$ **to** n **do**
  for $j \leftarrow 1$ **to** k **do**
    $dis \leftarrow (loc_j.x - pos_i.x)^2 + (loc_j.y - pos_i.y)^2$
    if $\sqrt{dis} < r_j$ **then**
      Add edge (s_j, t_i) with capacity 1 to E
    end if
    Add vertex s_j, t_i into V
  end for
end for
for $j \leftarrow 1$ **to** k **do**
  add edge $((s, s_j))$ with capacity l_j
end for
for $i \leftarrow 1$ **to** n **do**
  add edge $((t_i, t))$ with capacity 1
end for
$f \leftarrow Edmonds - Karp(G)$
return $f{=}{=}n$

There are $k + n + 2$ vertices in the graph as we add all the hostspots, clients and one supersource, supersink. The maximum edge will be that all the clients could connect to all the hostpots, which is kn. Then the total number of edges are $kn + k + n$.
The three for loops took $\mathcal{O}(kn) + \mathcal{O}(k) + \mathcal{O}(n)$. Time complexity for Edmonds-Karp(G) will be same as discussed in lecture $\mathcal{O}(|V||E|^2) = \mathcal{O}((k + n + 2)(kn + k + n)^2)$

Total time complexity: $\mathcal{O}(kn)+\mathcal{O}(k)+\mathcal{O}(n)+\mathcal{O}((k+n+2)(kn+k+n)^2) = \mathcal{O}((k+n+2)(kn+k+n)^2)$