VE477 Introduction to Algorithms
Homework 5

Taoyue Xia, 518370910087

2021/10/31

Ex. 1 — The partition problem

1. The linear partition problem is that given a set S of n positive integers and a positive integer k, create a partition of S into k subsets, and minimize the maximum sum of the largest subset.
   This problem arises in work allocation, which aims to balance the work for multiple processors to minimize the total runtime.

2. No, this is not a good solution. If we just partition the set into subsets by inserting dividers close to the average, we don't systematically evaluate all the possibilities, which may not give the right solution, only a greedy one.

   For example, given the set $S = \{5, 5, 2, 4, 7, 7\}$ and $k = 3$, firstly we can calculate the average $30/3 = 10$. Using the average partition method described in the question, we just partition S into $5, 5 \mid 2, 4, 7 \mid 7$. However, the right partition should give $5, 5, 2 \mid 4, 7 \mid 7$.

3. The minimum cost is the larger one between the last subset's sum or the minimum sum of one of the previous subsets. Therefore, denoting s_j as the j_{th} integer of the original set, the recursive function looks like:

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

   Where the basic case is:

$$M(1, k) = s_1, \quad \text{for all } k > 0$$

$$M(n, 1) = \sum_{i=1}^{n} s_i$$

4. If no duplicate quantity is stored, this algorithm will take exponential time complexity, since for every $i > j$, it will calculate $M(j, k)$ for many other times.

5. Each $M(i, k),\ i \in 1, \ldots, n$ should be stored to avoid recalculation. Also, the n prefix sums can be stored as $p[i] = \sum_{j=1}^{i} s_j = p[i-1] + s[i]$.

6. The pseudocode of a dynamic programming algorithm is shown below:

Algorithm 1: Dynamic Programming for Linear Partition Problem

```
Input  : A set $S$ with $n$ positive integers $s_1, \ldots, s_n$, a positive integer $k$
Output: The minimum cost over all the partitioning of the n elements into k ranges
        $M(n, k)$
Function LinearPartition($S$, $k$):
    $p[0] \leftarrow 0$;
    for $i = 1$ to $n$ do
        $p[i] \leftarrow p[i-1] + s_i$;
    end
    for $i = 1$ to $n$ do
        $M(i, 1) \leftarrow p[i]$;
    end
    for $i = 1$ to $k$ do
        $M(1, i) \leftarrow s_1$;
    end
    for $i = 2$ to $n$ do
        for $j = 2$ to $k$ do
            $M(i, j) \leftarrow \infty$;
            for $a = 1$ to $i - 1$ do
                $x \leftarrow \max M(x, j-1), p[i] - p[a]$;
                if $M(i, j) > x$ then
                    $M(i, j) \leftarrow x$;
                    $D(i, j) \leftarrow a$;
                end
            end
        end
    end
    return $M(n, k)$;
end
```

7. The calculation of the prefixed sum and the boundary condition $M(1, i)$ are both true, and they construct the base of the calculation of $M(i, j)$. Then, since we calculate each $M(i, j)$ from small to large, we don't need to calculate a duplicate value and will obtain the correct result. Therefore, it is equivalent to prove the all the previous conditions are true, finally, it will land in the boundary conditions. Therefore, the correctness is proved.

8. For calculating the prefixed sum and boundary conditions, the corresponding time complexity is $\mathcal{O}(n)$ and $\mathcal{O}(k)$. Then, for calculating the final $M(n, k)$, the loops needed are

$k + 2k + \cdots + nk = k\frac{n^2+n}{2} = \mathcal{O}(kn^2)$. Therefore, the time complexity is $\mathcal{O}(kn^2)$, proof done.

9. It can be realized by the $D(n, k)$ in the previous algorithm, which denotes the place of each divider. The pseudocode is shown below:

Algorithm 2: Reconstruct Path

```
Input  : the original set $S$, the divider matrix $D$, positive integers $n$ and $k$
Output: Print the partition with dividers
Function ReconstructPartition($S$, $D$, $n$, $k$):
    if $k = 1$ then
        Print the partition $s_1, \ldots, s_n$;
    else
        ReconstructPartition($S$, $D$, $D(n, k)$, $k - 1$);
        Print the partition $s_{D(n,k)+1}, \ldots, s_n$;
    end
end
```

Ex. 2 — Critical thinking

Since the black box B can generate a random number ranging from $0-4$, we can construct a new box based on B which works according to the following function: $B' = 5 * B + B$, which generates a random number ranging from $0-24$, totally 25 numbers. To convert the output into a random number in $0-7$, we can just drop the case when the generated number is 24, and the remaining 24 numbers $0-23$, will have equal probability of 1/24. Finally, we just need to convert the generated random number by taking its remainder when divide 8. In all, the generator works as $(B' \backslash \{24\}) \mod 8$.

As long as n is a positive number, we can use B to generate random integers in the range $0-n$. For $n = 5^i - 1$, where i is a positive integer, we can generate random numbers in $0-n$ using $5^{i-1}B + 5^{i-2}B + \cdots + 5^0 B$.

For more general cases, we can first decide the integer i such that $5^{i-1} - 1 < n \leq 5^i - 1$. Then we can determine a positive integer a, which makes $a * n \leq 5^{i+1} - 1$ and $(a+1) * n > 5^{i+1} - 1$. After that, we just need to drop those random numbers which are greater than $a * n$, and get the first one less or equal to $a * n$. Finally, calculate the random number $x \mod n$, we will get the random number. The probability of each number in $0-n$ is also equal.

The pseudocode below can make the previous explanation easier to understand:

```
Algorithm 3: Random number generation
Input  : A positive integer $n$
Output: A random number generated in range $0 - n$
1  Function randN($n$):
2      $i \leftarrow$ the positive integer which satisfies $5^{i-1} - 1 < n \leq 5^i - 1$;
3      $a \leftarrow 1$;
4      while $a * n \leq 5^i - 1$ do
5          $a \leftarrow a + 1$;
6      end
7      $B' \leftarrow$ the random generator following $5^{i-1}B + \cdots + 5^0 B$;
8      $r \leftarrow$ a random number generated by $B'$;
9      while $r > a * n$ do
10         $r \leftarrow$ a random number generated by $B'$;
11     end
12     $r \leftarrow r \mod n$;
13     return $r$;
14 end
```

Ex. 3 — Bellman-Ford algorithm

The pseudocode of the algorithm is shown below:

Algorithm 4: Determine negative cycle

```
Input  : A graph $G = \langle V, E \rangle$
Output: A boolean value indicating whether the graph has negative cycle
Function negCycle($G$):
    /* Denote $w$ as weight, $d$ as distance */
    $s \leftarrow$ a random vertex of $G$;
    for each vertex $v$ in $G$ do
        $v.d \leftarrow \infty$;
    end
    $s.d \leftarrow 0$;
    for $i = 1$ to $|G.V| - 1$ do
        foreach $(u, v) \in G.E$ do
            if $v.d > u.d + w(u, v)$ then
                $v.d = u.d + w(u, v)$;
            end
        end
    end
    foreach $(u, v) \in G.E$ do
        if $v.d > u.d + w(u, v)$ and $u.d \neq \infty$ then
            return true;
        end
    end
    return false;
end
```

Ex. 4 — Augmenting path

In the slide the capacity constraint property has been proved, so we just need to prove the flow conservation property.

Let G_f be the residual graph of G with respect to f, and f' be the flow in G_f. With the simple path P, we can observe that for vertices v not in P, $f_p(u, v) = f_p(v, w) = 0$. When a vertex $v \in P$ and $v \neq s, t$, then there only exists two vertices u_1 and u_2 such that the edges $e_1 = (u_1, v)$, $e_2 = (v, u_2)$ are in P.

Then, for the total flow at v, which is $\sum_u f_p(u, v)$, it only has two terms, $f_p(u_1, v) = b$ and $f_p(u_2, v) = -f_p(v, u_2) = -b$, so the excess flow at any vertex in $P.V \backslash \{s, t\}$ is 0.

Therefore, the flow conservation property of the residual graph is proved. Since the original graph G is initially flow conservation, by adding them together, we can still prove that the excess flow at each vertex other than $\{s, t\}$ is 0. Proof done.

Ex. 5 — Wifi network

Define the hostspots as $h_i, i = 1, \ldots, k$, the clients as $c_j, j = 1, \ldots, n$. Also, construct each hostspot and client as vertices, hostspots as sources, and clients as sinks. Finally, create a general source s which gives flow to the hostspots, and a general sink v which receives flows from clients. By applying Edmonds-Karp Algorithm, we can find out the maximum flow, and after comparing with the number of clients, we can finally get the answer. Then the following pseudocode will explain the whole algorithm:

Algorithm 5: Wifi network

```
Input  : range $\boldsymbol{r}$, load $\boldsymbol{l}$, hostspots number $\boldsymbol{k}$, clients number $\boldsymbol{n}$, $\boldsymbol{loc}$ as a set of
         locations of $k$ hostspots, and $\boldsymbol{pos}$ as a set of positions of $n$ clients
Output: True or False whether all clients can connect to the network
1  $G \leftarrow$ a directed graph with vertices $s$, $h_i$, $c_j$, $v$ initialized;
2  for $i = 1$ to $k$ do
3      add edge $(s, h_i)$ with capacity $l$ to $G$;
4  end
5  for $j = 1$ to $n$ do
6      add edge $(c_j, v)$ with capacity 1 to $G$;
7  end
8  for $i = 1$ to $k$ do
9      for $j = 1$ to $n$ do
10         dis $\leftarrow \sqrt{(loc[i].x - pos[j].x)^2 + (loc[i].y - pos[j].y)^2}$;
11         if $dis \leq r$ then
12             add an edge $(h_i, c_j)$ with capacity 1 to $G$;
13         end
14     end
15 end
16 $f \leftarrow$ Edmonds-Karp($G$);
17 if $f = n$ then
18     return True;
19 else
20     return False
21 end
```

From the pseudocode, we can see that the time complexity for initializing hostspots, clients and their interconnections are correspondingly $\mathcal{O}(k)$, $\mathcal{O}(n)$ and $O(kn)$. Regarding the Edmonds-Karp Algorithm, its time complexity is $\mathcal{O}(|V||E|^2)$, which in this case can be expressed as $\mathcal{O}((k+n+2)(k+n+kn)^2)$.

Therefore, the total time complexity is:

$$\mathcal{O}(k) + \mathcal{O}(n) + \mathcal{O}(kn) + \mathcal{O}((k+n+2)(k+n+kn)^2) = \mathcal{O}((k+n+2)(k+n+kn)^2)$$