HW2 - Theoretical Part

Xinyuan Cao
Uni: xc2461

October 15, 2018

1. Actually I'll briefly introduce my idea at first. And then give the pseudocode and precise proof following it. The main idea is to use BFS to solve this problem. For each node $p \in G$. Denote $S(p)$ as the number of shortest $v - p$ paths in the graph G. Let the node v be the root of the tree, and search until we explore w. Denote the nodes in w's parent's layer which have an edge connecting w as $x_1, \cdots, x_k$, then

$$S(w) = \sum_{i=1}^{k} S(x_i)$$

Then I'll show my pseudocode in Algorithm 1.

Next I'll prove its correctness. For $p \in V$, denote $S(p)$ as the number of shortest $v-p$ paths in the graph G. Denote l_p as the layer of p in BFS tree, that is, p belongs to the layer l_p in the BFS tree. By default, we assume the layer of the root node is equal to 0.

Claim 1 *For a BFS tree constructed by $G = (V, E)$ with root v. $\forall w \in G$, the length of the shortest $v - w$ path is equal to l_w.*

Proof I'll prove by contradiction. Denote that w is in the k_{th} layer of the BFS. Then obviously we can get a $v - w$ path by backtracking parent nodes from w, and then reverse the list. For example, if w's parent is w_1, for $1 \leq i \leq k-2$, w_i's parent is w_{i+1}, w_{k-1}'s parent is v. Then we get a $v - w$ path: $v - w_{k-1} - w_{k-2} - \cdots - w_1 - w$. Then I'll show it is one of the shortest $v - w$ paths in G.

Assume that there exists a shorter path with length $t < k$. Denote this path as $v - v_1 - v_2 - \cdots - v_{t-1} - w$. So we have $(v, v_1), (v_1, v_2), \cdots, (v_{t-1}, w) \in E$. From Claim 1 in BFS slides page 33, we know, $|l_v - l_{v_1}| \leq 1$, for $1 \leq i \leq t-2$, $|l_{v_i} - l_{v_{i+1}}| \leq 1$, $|l_{v_{t-1}} - l_w| \leq 1$. Hence,

$$|l_v - l_w| \leq |l_v - l_{v_1}| + \sum_{i=1}^{t-2} |l_{v_i} - l_{v_{i+1}}| + |l_{v_{t-1}} - l_w| \leq t$$

Algorithm 1 Compute number of shortest paths

input: undirected graph $G = (V, E)$, two nodes $v, w \in V$
output: the number of shortest $v - w$ paths in G

```
1: function NUMBER-SHORTEST-PATH(G = (V, E), v, w ∈ V)
2:     Array discovered[u]                ▷ initialized to 0, which means not discovered
3:     Array layer[u]                     ▷ initialized to ∞, the layer of BFS tree
4:     Array number[u]                    ▷ initialized to 0, the number of the shortest v − u paths
5:     Queue q
6:     discovered[v] = 1                  ▷ set v as the root
7:     layer[v] = 0
8:     number[v] = 1
9:     enqueue(q, v)
10:    while size(q) > 0 do
11:        x = dequeue(q)
12:        if x == w then                 ▷ we have explore all the parent's of w
13:            break
14:        end if
15:        for (x, y) ∈ E do
16:            if discovered[y] == 0 then
17:                discovered[y] = 1
18:                layer[y] = layer[x] + 1
19:                enqueue(q, y)
20:            end if
21:            if layer[y] == layer[x] + 1 then   ▷ y is in the next layer of x in BFS tree
22:                number[y] = number[x] + number[y]   ▷ Add number of short paths via x
23:            end if
24:        end for
25:    end while
26:    return number[w]
27: end function
```

Since v is the root of the tree, we have $l_v = 0$. So $|l_w| \leq t < k$, thus leading to a contradiction to the fact that w is in the k_{th} layer of the BFS tree. Therefore I've proved that claim 1 is correct. #

Claim 2 *Denote $P = \{P_i\}$ as the set of all shortest $v - w$ path in G. Then for each path $P_i \in P$, the second to last node of P_i is in w's parent's layer.*

Proof We continue to use the notation is above. Denote the second to last node of P_i is u. Then we have $(u, w) \in E$. By Claim 1 in BFS slides page 33, we know $|l_u - l_w| \leq 1$. So there is only three scenarios: u is in w's parent's layer, or u is in the same layer of w, or u is in the w's child's layer.

(a) If u is in w's parent's layer, then the claim is proved.

(b) If u is in the same layer of w. By Claim 1, we know the shortest $v - u$ path's length is equal to $l_u = l_v$. Because $P_i = v - \cdots - u - w$. Remove w and we get a new path from v to u with length $l_v - 1$, which causes a contradiction to the fact that the shortest length of $v - u$ path is equal to l_v.

(c) If u is in the w's child's layer. By Claim 1, we know the shortest $v - u$ path's length is equal to $l_u = l_v + 1$. Because $P_i = v - \cdots - u - w$. Remove w and we get a new path from v to u with length $l_v - 1$, which causes a contradiction to the fact that the shortest length of $v - u$ path is equal to $l_v + 1$.

So we get our claim proved. #

Then for our original problem. We prove it by induction.

Base: for layer 0, that means $w = v$. Then there is obvious one shortest $v - v$ path. From our algorithm, in the first while loop, we have $x == v$, so it break the while loop, and return $number[v] = 1$. This means our algorithm is correct.

Hypothesis: for nodes w in $1, 2, \cdots, k-1$ layer of the BFS tree rooted by v, Algorithm1 can find the correct answer.

Induction: consider the node w in the k_{th} layer of the BFS tree rooted by v. The construction of BFS tree is just the same as the class slides, so here we will not show its correctness proof. Since the output is $number[w]$, let's jump to the part relevant to the update of $number[w]$. From line 15, 21, 22 of the pseudocode, we know that we update $number[w]$ each time we find an x satisfying the following conditions: 1) there is an edge $(x, y) \in E$; 2) we explore x before y; 3) node x is in y's parent's layer. (*) From hypothesis, we node that for such kind of x, $number[x]$ is the number of the shortest $v - x$ path in G. This holds the time we call such $number[x]$ when updating $number[y]$ because we'll no longer update anything on $number[x]$ after we start to update $number[y]$. And this holds because in this algorithm we only update $number[x]$ when we are processing the parents of x, which is prior to the process of

updating $number[y]$. So from here we know that in the end, $number[w]$ will be the sum of the number of shortest $v - x$ path for all x satisfying the three conditions (*).

Denote $U = \{u_j\}$ as the set of all nodes in the w 's parent's layer. Denote $\{v-u_j-w\} \subset P$ as the set of shortest $v - w$ paths via u_j. From claim 2 we know that all shortest $v - w$ path could be a member of $\{v - u_j - w\}$ for some j. Plus, for $j_1 \neq j_2$, then the second to last node of $\{v - u_{j_1} - w\}$ is not the same as the second to last node of $\{v - u_{j_2} - w\}$. So we have $\{v - u_{j_1} - w\} \bigcap \{v - u_{j_2} - w\} = \varnothing$.So the number of all shortest $v - w$ path is equal to the sum of the element number of $\{v - u_j - w\}$. (1)

For a path $P_j = v - u_j^* - w \in \{v - u_j - w\}$, because the length of $u_j^* - w$ is equal to 1, the length of $v - u_j^* = l_w - 1 = l_u$. By Claim 1, we know this path $v - u_j^*$ is one of the shortest $v - u_j$ path. On the other hand, for each shortest $v - u_j^*$ path, we can add node w in the end, and get a path $v - u_j^* - w$ with length l_u, that means this path belongs to $\{v - u_j - w\}$. So we can get a bijection mapping of $\{v - u_j\}$, which is a set of element removing w from each element in $\{v - u_j - w\}$ and the set of all shortest $v - u_j$ paths. And here we know that there is a bijection $\phi : \{v - u_j\} \to \{v - u_j - w\}$. That is for each $v - u_j^* \in \{v - u_j\}$

$$\phi(v - u_j^*) = v - u_j^* - w$$

For distinct $v - u_j^*$ and $v - u_j^{**}$, we will get different results of ϕ because part of the path is different. Also for distinct$v - u_j^* - w$ and $v - u_j^{**} - w$, we will have different inverse image because the last edge of this path is the same, so there must be some different in the part except the last edge.

Therefore we know the element number of $\{v - u_j - w\}$ = the element number of $\{v - u_j - w\}$ = the number of shortest $v - u_j$ paths = $number[v_j]$.

And from (1) we know that the number of all shortest $v - w$ path = the sum of the element number of $\{v-u_j-w\} = \sum_{j:u_j \text{ssatisfies condition (*)}}$ the element number of $\{v-u_j-w\}$ = the sum of the number of shortest $v - x$ path for all x satisfying the three conditions (*) , which is just what our algorithm done. #

After proving the correctness, I'll compute its running time. In our pseudocode, the initializing process is $O(n)$ time. The code between initializing and while loop is constant time. The while loop will have at most n loops. For each loop of x, line 11-14 cost constant time. For each for loop, it takes $deg(x)$ times. So for the whole part of while loop, the running time = $\sum_{x \in V} deg(x)O(1) = O(m)$. So all in all,

$$\text{Algorithm 1's running time} = O(n) + O(m) = O(m + n)$$

2. My algorithm is basically based on Dijkstra-v3, but when updating the distance from the origin, I'll also consider the situation when the situation when "equation" $dist[v] = dist[u] + w_{uv}$ holds, and take down the minimal number of edges so far in an extra array. The pseudocode is shown in Algorithm 2.

   Then I'll prove the correctness of Algorithm 2.

   Claim 3 *Consider the set S at any point in the algorithm's execution. For each u in S, every shortest path from s to u contains no nodes in $V - S$.*

   Proof I'll use induction to prove it.

   Base: $|S| = 1$, $dist(s) = 0$, the shortest path from s to s only contains s, that means it includes no nodes in $V - S$.

   Hypothesis: suppose claim 3 is true for $|S| = k$. That is, for every $u \in S$, each shortest path from s to u contains no nodes in $V - S$.

   Induction: Denote $l(x_1 - \cdots - x_k)$ as the length of path $x_1 - \cdots - x_k$. Let v be the $k+1$ node added to S. And then we Update $dist$. By our algorithm we know that $dist[v]$ is the shortest by now, and there exists some node $u \in S$ so that $l(s - u - v) = dist(v)$. Assume that there is one shortest path $s - y - v$ from s to v that contains a node $y \in V - S$. That means $l(s - y - v) \leq dist(v)$. Here we denote $dist$ as the $dist$ array at this time, and $dist^*$ as the $dist$ array in the end of the whole program, that is after each edge of the graph has been used to update this array.

   First I'll show $dist^*(y) \geq dist(v)$. Assume that $dist^*(y) < dist(v)$, then there exists a node $y_1 \in V - S$ so that $dist^*(y_1) + w_{z,y_1} = dist^*(y)$. If $dist^*(y_1) == dist(y_1)$, then we know $dist(y_1) + w_{y,y_1} = dist^*(y)$. Or else, similarly there exists a node $y_2 \in V - S$ so that $dist^*(y_2) + w_{y_1,y_2} = dist^*(y_1). \cdots$ Do it again and again, finally you'll definitely find a y_i so that $dist^*(y_i) == dist(y_i)$. So here we have

$$dist^*(y) = dist(y_i) + w_{z,y_1} + w_{y_1,y} + \sum_{j=1}^{i-1} w_{y_j,y_{j+1}}$$

   Because $dist(v) \leq dist(y_i)$,

$$dist(v) < dist(y_i) + w_{z,y_1} + w_{y_1,y} + \sum_{j=1}^{i-1} w_{y_j,y_{j+1}} = dist^*(y)$$

   So there is a contradiction. So $dist^*(y) \geq dist(v)$.

   So now, we have $l(s - y - v) > dist^*(y) \geq dist(v)$, which leads to a contradiction to $l(s - y - v) \leq dist(v)$. Therefore we have each shortest node from s to v contains no nodes in $V - S$. So the claim is also correct for this step.

   Finally by induction we prove our claim. #

Algorithm 2 Compute minimum number of edges in shortest paths from s to v

input: weighted graph $G = (V, E, w)$, origin node $s \in V$

output: an array *best*, where $best[v]$ = the minimum number of edges in shortest paths from s to v

```
1: function MIN-EDGE-SHORTEST-PATH(G = (V, E, w), s ∈ V)
2:     Initialize(G, S)
3:     Q = BuildQueue(V, dist)
4:     S = ∅
5:     while Q ≠ ∅ do
6:         u = Extractmin(Q)
7:         S = S ⋃ {u}
8:         for (u, v) ∈ E do
9:             Update(u, v)
10:        end for
11:    end while
12:    return best
13: end function

14: function INITIALIZE(G, S)
15:    for v ∈ V do
16:        dist[v] = ∞
17:        best[v] = ∞
18:    end for
19:    dist[s] = 0
20:    best[s] = 0
21: end function

22: function UPDATE(u, v)
23:    if dist[v] > dist[u] + w_uv then
24:        dist[v] = dist[u] + w_uv
25:        best[v] = best[u] + 1
26:    else if dist[v] = dist[u] + w_uv then   ▷ if the paths have the same length, keep the one with fewer edges
27:        if best[v] > best[u] + 1 then
28:            best[v] = best[u] + 1
29:        end if
30:    end if
31: end function
```

Claim 4 *Consider the set S at any point in the algorithm's execution. For each u in S, every shortest path from s to u has no fewer edges than $best[u]$.*

Proof I also use induction to prove this claim.

Base: $|S| = 1$, $dist(s) = 0$, the shortest path from s to s, it has 0 edges, which is equal to $bests$.

Hypothesis: suppose claim 4 is true for $|S| = k$. That is, for every $u \in S$, every shortest path from s to u has no fewer edges than $best[u]$.

Induction: Let v be the $k+1$ node added to S. Assume that there is a shortest path from s to v whose edge number is smaller than $best[v]$. From Claim 3 we know that all nodes in this path is in S. Denote the second to last point of $s - v$ path is w. Then $w \in S$. By hypothesis we know that every shortest $s - w$ path's edge number $\geq best[w]$. Because path $s - w - v$ is the shortest path, so the path $s - w$ that is to remove node v from the original path is the shortest path of $s - w$. (Otherwise, we can use one shortest path of $s - w$ to substitute this path $s - w$, and will have a shorter $s - w - v$ path, which causes a contradiction.) So the number of edges of $s - w - v = 1+$ the number of edges of $s - w \geq 1 + best[w]$.

On the other hand, by the algorithm, after we update(w, v), we have get $best[v] \leq best[w]+1 \leq$ the number of edges of $s-w-v$. This will lead to a contradiction saying that this page's edge number is smaller than $best[v]$. So the claim is also correct for this step.

Finally by induction we prove our claim. #

So now let's come to our original problem. And we consider the situation when we ended the algorithm. By claim 4 we know that $best[u]$ = minimum number of edges in a shortest path from s to u. And we have our algorithm is correct! #

And the final step is to compute the running time. The running time of initialize is $O(n)$. BuildQueue needs $O(n)$ time. Inside each while loop: Extractmin(Q) needs $O(\log n)$, for each (u, v), update takes $O(\log n)$ time. So the running time is $n(O(\log n)) + \sum_u outdeg(u)O(\log n) = O(n \log n + m \log n) = O(m \log n)$.

3. Let M be an array of n entries, such that $M[i]$ =total penalty of the optimal sequence of the first i hotels. The pseudocode is shown in Algorithm 3.

Algorithm 3 Find optimal hotel sequence

input: the location of the hotels, in a dataset $A = \{a_i\}_{i=1}^{n}$
output: a number $M[n]$, denotes the total penalty of the optimal sequence of n hotels at which to stop

```
1: function FIND-OPTIMAL-STOPS(A)
2:     M[1] = (200 − a_1)^2
3:     for j = 2 to n do
4:         M[j] = min_{1≤i≤j−1} [M[i] + (200 − (a_j − a_i))^2]
5:     end for
6:     return M[n]
7: end function
```

Firstly I'll prove the correctness of this method.

Base: when $n = 1$. There's only one hotel a_1, and you have to stop there. So the penalty $M[1] = (200 - a_1)^2$. That's correct.

Hypothesis: assume that for each $i : 1 \leq i \leq k$ the method can find the best way to stop to minimize the total penalty of hotels $a_1, \cdots, a_i$. And we get the optimal total penalty $M[i]$.

Induction: for the situation with $k+1$ hotels: $a_1, \cdots, a_{k+1}$. From the algorithm we know

$$M[k+1] = \min_{1 \leq j \leq k} [M[j] + (200 - (a_{k+1} - a_j))^2]$$

Assume that we find a kind of stop sequence that generates an even smaller $M[k+1]^* < M[k+1]$. Denote the stop hotels in this situation as $b_1, \cdots, b_m$. From the problem we know that $b_m = a_n$. Let's consider $b_{m-1} = a_i$, $i < k+1$. This means that in the last day we traveled from a_i to a_{k+1}. Then we consider the situation of i hotels. From hypothesis we know the minimal total penalty for it is $M[i]$. So $M[k+1]^*$ = the total penalty of $\{b_1, \cdots, b_m\}$ = the total penalty of $\{b_1, \cdots, b_m\}$ + the total penalty of $\{b_{m-1}, b_m\} \geq M[i] + (200 - (a_{k+1} - a_i))^2 \geq \min_{1 \leq j \leq k} [M[j] + (200 - (a_{k+1} - a_j))^2] = M[k+1]$.

This leads to a contradiction to $M[k+1]^* < M[k+1]$. So $M[k+1]$ is the minimal total penalty for $k+1$ hotels situation. Hence we prove the algorithm is correct in this step.

So by induction we can know that $M[k]$ is the minimal total penalty for k hotels situation. Hence let $k = n$, we know this algorithm provides the right answer for n hotels. #

Next, I'll compute the time complexity. The computation of $M[1]$ takes constant time. The for loop will loop for $O(n)$ time. Inside each of the for loop, finding the minimum

takes $O(n)$ time. Return takes constant time. So the running time of this algorithm $= O(n^2)$.

Finally as we need to keep an array M, the space complexity is $O(n)$.

4. The pseudocode is shown in Algorithm 4.

   Then I'll prove the correctness of this algorithm. My algorithm uses quick-sort to sort the customers by their service time. That is, the more service time you have, the later your order is. Next I'll prove this situation can minimize the total waiting time. Since the input of Algorithm 4 is $\{t_1, \cdots, t_n\}$, after quick-sort, it becomes to be $\{t_{(1)}, \cdots, t_{(n)}\}$. Here $\{t_{(1)}, \cdots, t_{(n)}\}$ is the reorder of $\{t_1, \cdots, t_n\}$, so that $t_{(1)} \leq \cdots \leq t_{(n)}$. And for $1 \leq i \leq n$, we have (i) is the original index of $t_{(i)}$ in $\{t_1, \cdots, t_n\}$. From line 7 in the pseudocode, we know the output of the algorithm is $[(1), \cdots, (n)]$. So here the total waiting time is

$$T = \sum_{i=1}^{n} \sum_{j=1}^{i} t_{(j)} = \sum_{i=1}^{n} (n+1-i) t_{(i)}$$

   Assume that there exists another order $\{t^*_{(1)}, \cdots, t^*_{(n)}\}$ which can have the smallest waiting time T^*. Because $\{t^*_{(1)}, \cdots, t^*_{(n)}\}$ is different from $\{t_{(1)}, \cdots, t_{(n)}\}$, so there must exist $i < j$ s.t. $t^*_{(i)} > t^*_{(j)}$. (Otherwise $\{t^*_{(1)}, \cdots, t^*_{(n)}\}$ is monotonically increasing, than it will be same as $\{t_{(1)}, \cdots, t_{(n)}\}$, contradiction.) Then if we swap $t^*_{(i)}$ and $t^*_{(j)}$, and remains the other order the same, we will get a new total waiting time T^{**}.

$$\begin{aligned}
T^{**} &= \sum_{k=1 \text{ to } n, k \neq i,j} (n+1-k) t^*_{(k)} + (n+1-i) t^*_{(j)} + (n+1-j) t^*_{(i)} \\
&= \sum_{k=1 \text{ to } n, k \neq i,j} (n+1-k) t^*_{(k)} + (n+1-i) t^*_{(i)} + (i-j) t^*_{(i)} + (n+1-j) t^*_{(j)} + (j-i) t^*_{(j)} \\
&= \sum_{k=1 \text{ to } n} (n+1-k) t^*_{(k)} + (j-i)(t^*_{(j)} - t^*_{(i)}) \\
&< \sum_{k=1 \text{ to } n} (n+1-k) t^*_{(k)} \\
&= T^*
\end{aligned}$$

   This holds because $i < j$, $t^*_{(i)} > t^*_{(j)}$. So we have $(j-i)(t^*_{(j)} - t^*_{(i)}) < 0$. Here we have a new order $\{t^*_{(1)}, \cdots t^*_{(i-1)}, t^*_{(j)}, t^*_{(i+1)}, \cdots, t^*_{(j-1)}, t^*_{(i)}, t^*_{(j+1)}, \cdots, t^*_{(n)}\}$ And this order has a smaller total waiting time than T^*, leading to a contradiction that T^* is the smallest waiting time.

   So we know that $\{t_{(1)}, \cdots, t_{(n)}\}$ is the optimal order to minimize the total waiting time. Hence the output of our algorithm is optimal. We've proved its correctness. #

   Finally I'll compute the time complexity. The two for loops in the function find-optimal-order needs $O(n)$. The quick-sort is almost the same as in the slides, except

Algorithm 4 Compute the optimal order to minimized total waiting time

input: n; the service time for each customer, in a dataset $T = \{t_i\}_{i=1}^{n}$
output: an array P, where the $P[i]_{th}$ customer should be the i_{th} to be served

```
1: function FIND-OPTIMAL-ORDER(T)
2:     for i = 1 to n do
3:         S[i] = (i, t_i)                ▷ Use a list S to store index and waiting time
4:     end for
5:     quick-sort(S, 1, n)
6:     for i = 1 to n do
7:         P[i] = S[i][1]
8:     end for
9:     return P
10: end function

11: function QUICK-SORT(A, left, right)
12:     if |A| == 0 then
13:         return
14:     end if
15:     split = partition(A, left, right)
16:     quick-sort(A, left, split − 1)
17:     quick-sort(A, split − 1, right)
18: end function

19: function PARTITION(A, left, right)
20:     pivot = A[right]
21:     split = left − 1
22:     for j = left to right − 1 do
23:         if A[j][2] ≤ pivot[2] then
24:             swap(A[j], A[split + 1])
25:             split = split + 1
26:         end if
27:     end for
28:     swap(pivot, A[split + 1])
29:     return split + 1
30: end function
```

that we sort the array of two elements, and sort it by the second element. So here the running time is the same as the quick-sort in slides, that is $O(n \log n)$. This is because sorting a one-element list needs the same time as sorting a two-element list by the order of one element. So the total running time is $O(n) + O(n \log n) = O(n \log n)$.

5. (a) Denote $cost(n, p) = \max_i |w(i) - \frac{\sum_{l=1}^{n} A[l]}{k+1}|$ for n nodes and partition it into p arrays. Here we only consider the situation when $n \geq p$, because we do not allow empty subarrays. My algorithm is shown in Algorithm 5.

Algorithm 5 Determine the partition with minimal imbalance

input: array A, array's length n, number of subarrays $k+1$
output: The partition indices $j_1, \cdots, j_k$ with minimal imbalance

function FIND-SUBARRAYS(A,n,k)
  Initialize $M[1,1] = A[1]\frac{k}{k+1}$
  for $i = 2$ to n **do**
    $M[i,1] = M[i-1,1] + A[i]\frac{k}{k+1}$ ▷ efficient way to initialize
  end for
  $M[i,1] = |M[i,1]|$ ▷ also for initialize
  for $p = 2$ to $k+1$ **do**
    for $i = p$ to n **do** ▷ avoid $n < p$ situation
      $M[i,p] = \min_{j<i-1} \max(M[j,p-1], |\sum_{l=j+1}^{i} A[l] - \frac{\sum_{l=1}^{i} A[l]}{k+1}|)$
    end for
  end for
end function

function TRACE-SUBARRAYS(i, p)
  if $p == 1$ **return then**
  else
    Find $1 \leq j < i-1$ such that $M[i,p] = \max(M[j,p-1], |\sum_{l=j+1}^{i} A[l] - \frac{\sum_{l=1}^{i} A[l]}{k+1}|)$
    Trace-subarrays($j, p-1$)
    output j
  end if
end function
Initial Call: Trace-subarrays($n, k+1$)

Firstly I'll prove the correctness of function Find-subarrays using induction.

Base: for $k = 1$, $n = n_0$. there is only one subarray, that is the array itself. So here the $cost(n_0, 1) = |\sum_{l=1}^{n_0} A[l] - \frac{\sum_{l=1}^{n_0} A[l]}{k+1}| = |\sum_{l=1}^{n_0} A[l]|\frac{k}{k+1}$. This is corresponding to our initialization. So it's correct for base case.

Hypothesis: assume that for each pair (i_0, p_0) where $1 \leq i_0 \leq p$, $1 \leq p_0 \leq p$ it is correct.

Induction: I need to prove that it is correct for $(i, p+1)$, $(i+1, p)$ and $(i+1, p+1)$. From the algorithm we know that

$$M[i, p+1] = \min_{j<i-1} \max(M[j+1, p], |\sum_{l=j+1}^{i} A[l] - \frac{\sum_{l=1}^{i} A[l]}{k+1}|)$$

For each j, $j+1 < i, p \leq p$. So from induction hypothesis, we know $M[j+1, p]$ provides the minimum of the cost function. Assume that there exists another partition that has a small $M[i, p+1]$, denote as $M[i, p+1]^*$. Denote its last subarray as $A[j_p + 1, n]$. Then $M[i, p+1]^* = \max(M[j_p, p], |\sum_{l=j_k+1}^{i} A[l] - \frac{\sum_{l=1}^{i} A[l]}{k+1}|) \leq M[i, p+1]$. Thus it leads to a contradiction to $M[i, p+1]^* < M[i, p+1]$. So it is also correct for $M[i, p+1]$.
Similarly we can prove that it is correct for $(i+1, p)$ and $(i+1, p+1)$.
So by induction we know that this algorithm is true.

Secondly for the function trace-subarrays(i, p). We can also use induction.
Base: it is true for 1 subarray situation. That is output nothing.
Hypothesis: assume for each pair (i_0, p_0) where $1 \leq i_0 \leq p$, $1 \leq p_0 \leq p$ we can trace the partition nodes.
Induction: I need to prove that it is correct for $(i, p+1)$, $(i+1, p)$ and $(i+1, p+1)$. From line 16 we know we find and output the last indice j_k which must hold to get the optimal solution. Then it turns to situations in hypothesis, and we can get each indices finally. So this is correct.
By induction, we know that my algorithm proves right. #

Then I'll compute it's running time. For initializing the boundary, it takes $O(n)$ time. The first for loop with implement $O(k)$ times, the second for loop will implement $O(n)$ times. Inside the for loop, it needs $O(n)$ time. So the running time of function Find-Subarrays is $O(n) + O(k)O(n)O(n) = O(kn^2)$ For the trace-subarrays function, finding j needs $O(n)$ time. Consider the worst situation, the total running time is $O(kn^2)$. So the running time of the algorithm is $O(kn^2)$.
For space complexity, we need to maintain M, so it takes $O(nk)$.

(b) I'll change the recursive equation from Algorithm 5.
Change Line 9 to:

$$M[i, p] = \min_{j<i-1} \{M[j, p-1] + |\sum_{l=j+1}^{i} A[l] - \frac{\sum_{l=1}^{i} A[l]}{k+1}|\}$$

Also for tracing part, change line 16 to:
Find $1 \leq j < i-1$ such that $M[i, p] = M[j, p-1] + |\sum_{l=j+1}^{i} A[l] - \frac{\sum_{l=1}^{i} A[l]}{k+1}|$

After changing the recursive equation, we can show the correctness very similarly to the proof of question(a), because here we want to minimize $\sum_i |w(i) - \frac{\sum_{l=1}^{n} A[l]}{k+1}|$. And by doing the recursion, we can find all possible last subarray $A[j_k + 1, n]$, and using the inductive hypothesis that we can find optimal cost function with $A[1, j_k]$ with k parts. So the minimum of each possible solution is the optimal solution of the whole problem.

And then, about time complexity. Actually this change does not change the time complexity inside the for loop, which is still $O(n)$. So in the same way, we have its running time is $O(kn^2)$.

Finally for space complexity, also the changes do not affect the space complexity, so it takes $O(nk)$ space, just the same as question (a).