Analysis of algorithms I - Problem Set **4** Problem **1**

Name: **Cheng Liu**
Uni:**cl3173**

Collaborators and Sources: Introduction to algorithms
(**3rd edition**)

October 31, 2013

Problem 1.

Let b_i denote the ith bikestand in the map.

This problem can be treated as a two-layer shortest path problem.

The first stage we need to calculate all the shortest **distance** between two bikestands in the map. The result D'(i,j) means the shortest distance used to calculate the cost if we start from b_i to b_j without returning bike to other bikestand along the way and let **PathD(i,j)** be the path.

The second stage we just calculate the shortest cost-distance between b_s to b_t and let **Path(i,j)** be the path.Notice the fact that in this stage, it means in every bikestand we pass, we will return the bike and get the new food,and the decision for each edge is independent, thus **we can individually choose local optimum based on three nutrition options as the cost distance of the edge**.

For there won't be any negative edge in the map, in the first stage we can use Floyd Algorithm[1]to calculate the D' Matrix and record the path for every D'(i,j), and for the second stage we can just use Dijkstra Algorithm[2].

To simplify the proof, we won't give detail information about these two graph algorithms,the optimal substructure and correctness of these two classical algorithms have been used for many years and you can refer to the corresponding chapter of textbook in the footnotes.

Then I'll prove the correctness of my algorithm.

Let S_{op} be the optimum for the problem, $S_{op} = \{s_s, s_i, n_j, s_k, \cdots, s_t\}$, $Cost_{op}(i, j, path)$ denotes the optimum cost from b_i to b_j along the path without stop.

Then the total cost will be:

$Cost_{op} = \sum_{s_i, s_j \in S_{op} s_i s_j \text{ are neighbor } index(s_i) < index(s_j)} Cost_{op}(i, j, PathD_{op}(i, j))$.

s_i means we will return and take a new bike at bikestand b_i, n_j means that we just pass the bikestand b_j.

In the first stage, we shrink the solution space, we need to prove that at least global optimum won't be excluded(May path with same value as the optimum be excluded).

That is, if between two neighbor s_i and s_k in S_{op}, the mediate path of non-stop bikestand $n_j \cdots n_l$ (that is $PathD_{op}(i, k)$) must be equal to the result of PathD(i,k). The proof is simple, By Floyd algorithm, PathD(i,k) is the shortest distance between i,k, thus $Cost_{op}(i, j, PathD_{op}(i, k)) \geq Cost_{op}(i, j, PathD(i, k))$, because the cost of traveling more distance without stop must be no smaller than traveling less distance without stop. At least we can say that if we replace the orginial $PathD_{op}(i, k)$ with the result we calculate in the first stage($PathD(i, k)$), the new total cost $Cost_{new}$ won't be larger than $Cost_{op}$,that is, we can replace S_{op} with our new S_{new} without loss of the property of optimum. Otherwise our original S_{op} won't be optimum, that contradicts with our assumption.

[1] `IntroductiontoAlgorithms3rdEditionChap25.2`

[2] `IntroductiontoAlgorithms3rdEditionChap24.3`

So far, we have proved that if there is any non-stop sequence in S_{op} we can always replace it with our PathD from Floyd algorithm. Thus the shrink is safe.

At the second stage, we just need to prove that we can treat optimum cost as distance, because the Dijkstra algorithm will find the shortest cost path for us.

In the second stage, what we pick as next station is the bikestand where we will stop and change a bike. Because every such an s_i we need to change our nutrition and the distance between two s_i and s_j is only determined by D'(i,j) we calculated in the first stage, thus we can only make the local optimum by just checking the cost of travel over D'(i,j) distance.

Note that D'(i,j) is fixed, the result of $Cost_{op}(i, j, PathD_{op}(i, j))$ is also fixed, we just check the distance, and take the best nutrition option. Thus we can transfer our optimum cost to be the distance of a graph.

At this point, we have finished the proof.

As for the time complexity, the Floyd algorithm is $O(n^3)$,for there one three layer-loop in the algorithm, and because the graph we have it is a complete graph, we use matrix to represent it.The total cost of Dijkstra algorithm as well as the sequence output is $O(n^2)$,because there is a two-layer loop(while-for),the while sentence runs at most n times,and each while run inside loop n times and cost of each inside loop is just O(1). So the total cost of the whole problem is $O(n^3)$.

The pseudo-code is below:

Algorithm 1 Cost-Time

Require: d,r

Iteration:

$time = d/r$
$cost = \infty$
if $time \leq \frac{3}{4}$ **then**
  $cost = 0$
else if $time \leq 1$ **then**
  $cost = 5$
else if $time \leq 2$ **then**
  $cost = 10$
end if

Output: $cost$

Algorithm 2 Cost

Require: d

Iteration:

if d == 0 **then**
    cost = 0
end if
cost = 1 + Cost-Time(d,3)
cost2 = 2 + Cost-Time(d,5)
if $cost2 < cost$ **then**
    $cost = cost2$
end if
cost2 = 3 + Cost-Time(d,10)
if $cost2 < cost$ **then**
    $cost = cost2$
end if

Output: $cost$

Algorithm 3 Floyd-Algorithm

Require: D

Iteration:

let $D' = D$, PathD be the array$[1 \cdots n][1 \cdots n]$ and all the elments are -1
for k = 1 to n **do**
    for i = 1 to n **do**
        for j = 1 to n **do**
            if $D'[i,k] + D'[k,j] < D'[i,j]$ **then**
                $D'[i,j] = D'[i,k] + D'[k,j]$
                $PathD[i,j] = k$
            end if
        end for
    end for
end for

Output: $D', PathD$

Algorithm 4 FloydPath-Output

Require: i,j,PathD

Iteration:

location = j
while $PathD[i, location] \neq -1$ **do**
    Print $PathD[i, location]$
    $location = PathD[i, location]$
end while

Output: $D', PathD$

Algorithm 5 Dijkstra-Algorithm

Require: s,t,D'

Iteration:

let Path be the array$[1 \cdots n]$ and all the elments are -1
let CostDistance be the array$[1 \cdots n]$ and all the elments are ∞
let $V = \emptyset$
for i = 1 to n **do**
    $temp = Cost(s, i, D'[s, i])$
    if $temp < CostDistance[i]$ **then**
        $CostDistance[i] = temp$
        if $i \neq s$ **then**
            $Path[i] = s$
            Insert the i into V
        end if
    end if
end for
while $V \neq \emptyset$ **do**
    let **new** be the element in V with least value of CostDistance, and POP(new)
    if $CostDistance[new] == \infty$ **then**
        let Path be the array$[1 \cdots n]$ and all the elments are -1
        $tcost = \infty$
        RETURN
    else if $new == t$ **then**
        $tcost = CostDistance[new]$
        RETURN
    end if
    for every element next in the V **do**
        $temp = Cost(new, next, D'[m, next])$
        if $CostDistance[new] + temp < CostDistance[next]$ **then**
            $CostDistance[next] = temp + CostDistance[new]$
            $Path[next] = new$
        end if
    end for
end while

Output: $tcost, Path$

Algorithm 6 Dijkstra-Output

Require: s,t,Path,PathD

Iteration:

$location = t$
while $Path[t] \neq -1$ **do**
    $FLAG = 1$
    Print t
    FloydPath-Output(Path[t],t,PathD)
end while
if $FLAG == 1$ **then**
    Print s
end if

Output:

Algorithm 7 Two-Layer

Require: s,t,D

Iteration:

$totalcost = 0$
if $s == t$ **then**
    RETURN
else
    [D',PathD] = Floyd-Algorithm(D)
    [tcost,Path] = Dijkstra-Algorithm(s,t,D')
    totalcost = tcost
    Dijkstra-Output(s,t,Path,PathD)
end if

Output: totalcost

Analysis of algorithms I - Problem Set **4** Problem **2**

Name: **Cheng Liu**
Uni:**cl3173**

Collaborators and Sources: Introduction to algorithms
(**3rd edition**)

October 31, 2013

Problem 2.

Because we can only visit per bikestand once, the problem is treated as a non-preemptable job schedule problem, we treat refilling a certain bikestand i as a ceratin job must start before $t_i - c_i$.

At first we say that for such a problem if there is a possible schedule with idle time between any two neighbor jobs, we can remove those space to get a tighter scheduler and the solution is also possible, then we can always finish a job and start another job and can still find a optimum solution if there are. This shrinks our solution space.

Let B be the set of all the bikestands and S be any schedule.

And we define the work set B compatible with t(time) is that the smallest $t_i - c_i$ of a certain job is no larger than t. That is, it is still possible for all the jobs in the set B can be finished in time.

Now consider our problem E(B,t), the optimum substructure is below:(Notice that if there is possible solution, **E(B,t) = 1**)

$$E(B,t) = \left\{ \begin{array}{lll} 0 & \text{if} & \text{B is not compatible with t.} \\ 1 & \text{if} B = \emptyset & \\ max_{b_i \in B}(E(B - \{b_i\}, t + c_i)) & & \text{else} \end{array} \right\}$$

Now we prove the correctness of it.

1.If B is not compatible with t, it means that there is always at least a job cannot be finished after time t, thus the answer is obviously 0.

2.If B is empty set, it means that no job should be schedule after time t, then the answer is always 1.

3.Otherwise, notice that the problem is a binary-value problem, if $E(B,t)_{op} = 1$, the first job excuted is just b_i,then we take out b_i, and we say $E(B - \{b_i\}, t + c_i) = E(B - \{b_i\}, t + c_i)_{op} = 1$ is also the optimal solution,other wise ,the E(B,t) cannnot be 1.

If E(B,t) = 0, it means that the all the job cannot be finished, then it soon end at once.

Notice that the schedule of $E(B - \{b_i\}, t + c_i)$ may not be the same as $E(B - \{b_i\}, t + c_i)_{op}$, but for this binary-value problem, we can say $E(B - \{b_i\}, t + c_i)$ is also a optimum.

The greedy algorithm is that we always take the job b_i with the earliest t_i in the remaining job queue to execute.

Then we prove the safety of the greedy algorithm.

If the optimum of E(B,t) is 0, greedy algorithm will return 0,it is no doubt.

Then if the optimum of E(B,t) is 1, then we apply the greedy algorithm, we can always find a possible solution.

Notice that if

$$\{E(B,t)_{op} = 1\} \Leftrightarrow$$

{there is at least one sequence of B that will let the recursion continue untill B = $\emptyset$}

$$\{E(B,t)_{SE} = 1\} \Leftrightarrow$$
$$\{\text{sequence SE will let the recursion continue untill B} = \quad \emptyset\}$$

Assume that b_i is the bikestand with the earliest t_i in B for the problem E(B,t) and the solution of greedy algorithm is SE_{ga}.

Let one optimal solution sequence for E(B,t) be SE_{op}, if $SE_{op} = SE_{ga}$ then we have finished the proof.

Otherwise, W.L.O.G,we assume the only difference betwwen SE_{ga} and SE_{op} is the position of b_i,we assume there is only b_k before b_i in SE_{op}, we can simply exchange b_k with b_i, the new sequence is also compatible.

Because b_i is the job with the earliest t_i, and if in the SE_{op}, b_i finishes in f_i, then there must be $f_k < f_i \leq t_i \leq t_k$, after the exchange, $f_i' < f_k' = f_i \leq t_i \leq t_k$, so the new schedule must also be compatible.

Thus for our SE_{ga}, the recursion can also continue untill the set be empty, that is,so that is $E(B,t)_{SE_{ga}} = 1$.

Generally, if there are a lot of differences between SE_{op} and SE_{ga} we can gradually exchange elements pair in SE_{op} to make it SE_{ga} without avoiding compatibility.

So far we have proved that if there are schedules all the jobs can be finished then our greedy algorithm safely find one of them.

The pseudo-code is below the running time is $O(nlgn)$ dominated by the **sort**(we can apply a heap sort to this problem),the greedy algorithm only runs a length-n loop and costs $O(n)$,thus the total running time is still $O(n)$:

Algorithm 1 refill

Require: B,C,T

Iteration:

```
sort B in the ascending order correspoding to T
letK[1..|B|] = B
t = 0
current = 0
while B ≠ ∅ do
    current = POP(B)
    if t > T[current] − C[current] then
        K = ∅
    else
        t = t + C[current]
    end if
end while
```

Output: K

Analysis of algorithms I - Problem Set **4** Problem **3**

Name: **Cheng Liu**
Uni:**cl3173**

Collaborators and Sources: Introduction to algorithms
(**3rd edition**)

October 31, 2013

Problem 3.

a.

For the deterministic algorithm, as efficient as possible, we use Hash Table[3] to record the every elment, in reality, the average time to search for an element in a hash table is O(1) even the worst case is $\Theta(n)$. For this problem, we just take the size of hash table as the smallest prime larger than n and for there is no delete operation in Hash,we can use open-address hasing.

And because we must at least scan $\lceil n/4 \rceil$ of the array and the Hash table takes up O(n) space, thus the time complexity and space complexity is both O(n). And counter array and red-black tree are also possible choice, while counter array may take up a lot of space for we don't know the range of integer in the array A,and red-black tree can bound the time complexity to O(nlgn),but seldom will the hash table encounter the worst case.

The pseudo-code is below,this is only one loop in the algorithm and each one consumes O(1) time for the advantage of Hash Table,thus the total time complexity is O(n):

Algorithm 1 FrequentValue

Require: A,n

Iteration:

```
create the hash table and make all the elements in hash table is 0
for i = 1 to n do
    Hash(A[i]) = Hash(A[i]) + 1
    if Hash(A[i]) ≥ ⌈n/4⌉ then
        RETURN A[i]
    end if
end for
RETURN 0
```

b.

It is easy to know that the worse case is that we never find the frequent value, thus we will keep on looping forever,the algorithm will never end .

In this situation,no matter how we implement the algorithm, the running time will be $\Theta(\infty)$

[3] IntroductiontoAlgorithms3rdEditionChap11

c.

As for the requirement of the problem, it doesn't have any requirement of efficiency. We assume that random function cost O(1), in every loop we can scan the whole array to calculate the x.Because we don't know the real possibility distribution of element in array A, we can only find the expectation upper bound. that is, there is only one frequent value appearing just $\lceil \frac{n}{4} \rceil$.

It is easy to know if there is more frequent value and frequent value appearing more times, we will find them quickly. Let X be the random variableindicating the running time of the algorithm. If in every loop we scan the whole array to find the frequent value, thus the cost of each loop is O(n), then we have:

$$\begin{aligned}
E[X] &= \sum_{i=1}^{\infty} \frac{\lceil \frac{n}{4} \rceil}{n}(1 - \frac{\lceil \frac{n}{4} \rceil}{n})^{i-1} in \\
E[X] &= \frac{\lceil \frac{n}{4} \rceil}{n} \sum_{i=1}^{\infty}(1 - \frac{\lceil \frac{n}{4} \rceil}{n})^{i-1} in \\
(1 - \frac{\lceil \frac{n}{4} \rceil}{n})E[X] &= \frac{\lceil \frac{n}{4} \rceil}{n} \sum_{i=1}^{\infty}(1 - \frac{\lceil \frac{n}{4} \rceil}{n})^{i} in \\
E[X] &= n \sum_{i=1}^{\infty}(1 - \frac{\lceil \frac{n}{4} \rceil}{n})^{i-1} \\
E[X] &= \frac{n^2}{\lceil \frac{n}{4} \rceil} \\
E[X] &= O(n)
\end{aligned} \tag{1}$$

And we can also calculate the count of every element when first time scanning,with hash table($O(n)$),then every time we take x just consume O(1) time, and similarily, its expectation time is $O(n + \frac{n}{\lceil \frac{n}{4} \rceil}) = O(n)$, same with our method.

d.

The way we delete the element is that we swap the element with the tail element, then we modify the range of our random number generator, thus the tail element will never be selected. Pseudo-code is Algorithm 2. (you may find it in the next page) The time complexity of this algorithm depends on its concrete implementation, we will discuss it in subsection e,f.

e.

The worst time is that there is only one frequent value appearing just $\lceil n/4 \rceil$ times and all the non-frequent values have all selected before the first frequent value is selected.

Algorithm 2 FrequentDeleteValue

Require: A,n

Iteration:

```
bound = n
while bound!=0 do
    Pick a uniform random number i between 1 and bound
    Scan A[1 ··· bound],let x be the number of times A[i] appears in the array A
    if x ≥ ⌈n/4⌉ then
        Return A[i]
    else
        exchange A[i] with A[bound]
        bound = bound − 1
    end if
end while
```

This time the most loop times of our algorithm will be $n - \lceil n/4 \rceil + 1$. If we scan the whole array to find x each time,each loop cost O(n), the time complexity will be $O((n - \lceil n/4 \rceil + 1) * n) = O(n^2)$.

Or we use a hash table to store all the counters when first time scan the array costing O(n) first time, then take the different x in O(1) time, the time complexity will be $O((n - \lceil n/4 \rceil) * 1 + n) = O(n)$.

As we can see from the conclusion above, we successfully reduce the time complexity of the worst case.

f.

As for the expectation time,because we don't know the real possibility distribution of element in array A, we can only find the expectation upper bound. that is, there is only one frequent value appearing just $\lceil \frac{n}{4} \rceil$.

If we scan the whole array per loop.

$$E[X] = \sum_{i=1}^{n-\lceil \frac{n}{4} \rceil+1} \frac{\lceil \frac{n}{4} \rceil}{n-i+1} \left(\prod_{k=1}^{i-1} \left(1 - \frac{\lceil \frac{n}{4} \rceil}{n-k+1}\right)\right) in \tag{2}$$

the expecation number of iteration won't exceed the result we find in subsection c and the difference is only in the constant part, and intuitively, the algorithm will end soon than previous algorithm.

So the time complexity will also be $O(n)$

If we use hash table, because the iteration time is constant, so the first scaning dominates, the time complexity will also be $O(n)$.