Prof. Helmut Alt
Team 31. 20140174 Hyungrok Kim

Exercise 1

Assume there are n number of stations, pair to each number of 1 to n.

t is the time data, initial input t is the departure time.

s is the int data one of 1 to n which stands for the departure station

d is the int data one of 1 to n which stands for the arrival station

for the graph G,

not only the edges, vertices has own cost.

c(u,v) is the needed time for moving to v from u

c(v), which is the cost vertex v is the allocation interval in v station

S is vertices to which shortest path has been found

Q is queue of others

T[v] is expected arrival time of vertex v

```
procedure ArrivalTime(s, t, G, d)
        for all vertices v∈V do T[v] := ∞;
        T[S] := t;
        Q := V;   S := ∅;
        while Q≠ ∅ do
            let u be a vertex with T[u] is minimum
            Q := Q – {u} ;   S := S U {u};
            for all v∈Q adjacent to u do
                if T[v] > (T[u]%c(u)+1)*c(u) + c(u, v) then
                T[v] := (T[u]%c(u)+1)*c(u) + c(u, v);
        return T[d];
```

this returned T[d] is the expected arrival time to d station.

The equation (T[u]%c(u)+1)*c(u) means the boarding time at u station.

For example, assume that

s : Seoul is departure station,

d : Daejeon is intermediate station

b : Busan is arrival station

c(s,d) = 150 minutes

c(b,d) = 60 minutes

c(b) = 60 minutes
and the departure time in 00:00 am which is treated as 0
through ArrivalTime(s, 0, G, b)
initial T[d] and T[b] = ∞
T[d] becomes 150 which stands for 01:00 am
From Daejeon, T[d] is 150 and c(d) is 60 so
(T[d]%c(d)+1)*c(d)is like
(150%60+1)*60 = (2+1)*60 = 180
which stands for the possible fastest boarding time at Daejeon, so,
T[b] := (T[d]%c(d)+1)*c(d) + c(d, b) = 180 + 60 = 240
Which stands for 04:00 am which is the output,
expected arrival time at Busan.

Exercise 2
(a)
we can modify Dijkstra's algorithm by adding π[v] array data which saves the vertex v's predecessor vertex. Then we can use additional modification to printout the shortest paths.
Modified Dijkstra's algorithm is like below,
s : source
G : graph
S : vertices to which shortest path has been found
Q : others
ST : stack
tmp : temporary data to store vertex
array π[v] for v∈V, predecessor of vertex v we adjusted so far.
array D[v] for v∈V, length of shortest path to v found so far.

<u>procedure</u> Dijkstra(s, G)
    <u>for</u> all vertices v∈V <u>do</u> π[v] := null;
    <u>for</u> all vertices v∈V <u>do</u> D[v] := ∞;
    D[s] := 0;

```
Q := V;   S :=s  ∅;
while Q≠ ∅  do
    let u be a vertex with D[u] is minimum
    Q := Q – {u} ;   S := S U {u};
    for all v∈Q adjacent to u do
        if D[v] > D[u] + c(u, v) then
          D[v] := D[u] + c(u, v);
          π[v] := u;
    stack ST;
    for every v∈V do
        tmp = v;
        while tmp is not null do
            push temp to ST;
            tmp = π[tmp];
        while ST is not empty do
            pop and print ST;
```

then when we execute Dijkstra(s, G) shortest paths to every vertex from vertex s will be printed out.

(b)

Floyd-warshall algorithm we learned from class is like below,

```
procedure Floyd-warshall(G,c)
    for i:= 1 to n do
        for j:=1 to n do
            d^0_ij:= 0        if i=j
                   c(i,j)     if i!=j and (i,j) ∈E
                   ∞          otherwise
    for k:= 1 to n do
        for i:=1 to n do
            for j:=1 to n do
                d^k_ij:= min(d^{k-1}_ij , d^{k-1}_ik + d^{k-1}_kj)
```

it has 3-dimensional array $d^k{}_{ij}$ and it doesn't reuse any data space during execution, so the original algorithm need data space of n^3 distances, so, the space requirement = $\theta(n^3)$

We can make improvement in space complexity here because the equation

$d^k{}_{ij}$:= min($d^{k-1}{}_{ij}$, $d^{k-1}{}_{ik} + d^{k-1}{}_{kj}$)

only uses $d^k{}_{lr}$ and $d^{k-1}{}_{lr}$ (l and r are any variable in i j k)

then we can the first dimension which has k arrays to have just 2 and keep replacing the data.

Like below

d1 and d2 are 2-dimensional n*n arrays

```
procedure Floyd-warshall(G,c)
    for i:= 1 to n do
        for j:=1 to n do
            d1_ij:= 0        if i=j
                   c(i,j)    if i!=j and (i,j) ∈E
                   ∞         otherwise
    for k:= 1 to n do
        for i:=1 to n do
            for j:=1 to n do
                if k%2==1
                    d2_ij:= min(d1_ij , d1_ik + d1_kj)
                else
                    d1_ij:= min(d2_ij , d2_ik + d2_kj)
    if n%2==1
        d2 array has the lengths of the shortest paths
    else
        d1 array will be the lengths of the shortest paths
```

by this method data will be stored in two of 2-dimensional arrays, so the space requirement of the algorithm will be reduced to $\theta(n^2)$

(c)

from the original Floyd-warshall algorithm, I will change

$d^k{}_{ij}$:= min($d^{k-1}{}_{ij}$, $d^{k-1}{}_{ik} + d^{k-1}{}_{kj}$)

this line to add $\pi^k{}_{ij}$ which show the predecessor of j adjusted so far

set the initial $\pi^k{}_{ij}$'s values to null

<u>procedure</u> Floyd-warshall(G,c)
    <u>for</u> i:= 1 <u>to</u> n <u>do</u>
        <u>for</u> j:=1 <u>to</u> n <u>do</u>
            $d^0{}_{ij}$:= 0 if i=j
                c(i,j) if i!=j and (i,j) ∈E
                ∞ otherwise
    <u>for</u> k:= 1 <u>to</u> n <u>do</u>
        <u>for</u> i:=1 <u>to</u> n <u>do</u>
            <u>for</u> j:=1 <u>to</u> n <u>do</u>
                if $d^{k-1}{}_{ij} < d^{k-1}{}_{ik} + d^{k-1}{}_{kj}$
                    $d^k{}_{ij} = d^{k-1}{}_{ij}$
                    $\pi^k{}_{ij} = \pi^{k-1}{}_{ij}$
                else
                    $d^k{}_{ij} = d^{k-1}{}_{ik} + d^{k-1}{}_{kj}$
                    $\pi^k{}_{ij} = k$
    stack ST;
    <u>for</u> i:=1 to n <u>do</u>
        for j:=1 to n do
            tmp = j;
            <u>while</u> tmp is not i <u>do</u>
                push tmp to ST;
                tmp = $\pi^k{}_{i\ tmp}$
                <u>while</u> ST is not empty <u>do</u>
                    pop and print ST;

then when we execute, the algorithm will print all shortest paths.

Exercise 3

```
int M[n][n] // instructed input nxn matrix with the (i,j)th entry of c(i,j) if (i,j) is
            // an edge of the graph and ∞ otherwise.
Int d[n][n][n] // 3-dimensional array which stands for d^k_ij when k, i, j are
               // each 1 to n.
for(int i=0; i<n ; i++){
        for(int j=0; j<n; j++){
                d[0][i][j] = M[i][j];
        }
}
for(int k=1; k<n; k++){
        for(int i=0; i<n; i++){
                for(int j=0; j<n; j++){
                        if ( d[k-1][i][j]>(d[k-1][i][k]+d[k-1][k][j]) ){
                                d[k][i][j]=( d[k-1][i][j]+d[k-1][k][j] );
                        }
                        else{
                                d[k][i][j]=d[k-1][i][j];
                        }
                }
        }
}
```

d[n][i][j] array will be the length of shortest path from i to j.