Problem session 10

Dibran Dokter 1047390

November 23, 2019

10

10.1

$$D(0) = \begin{bmatrix} 0 & \infty & \infty & \infty & -1 & \infty \\ 1 & 0 & \infty & 2 & \infty & \infty \\ \infty & 2 & 0 & \infty & \infty & -8 \\ -4 & \infty & \infty & 0 & 3 & \infty \\ \infty & 7 & \infty & \infty & 0 & \infty \\ \infty & 5 & 10 & \infty & \infty & 0 \end{bmatrix}$$

$$D(1) = \begin{bmatrix} 0 & \infty & \infty & \infty & -1 & \infty \\ 1 & 0 & \infty & 2 & 0 & \infty \\ \infty & 2 & 0 & \infty & \infty & -8 \\ -4 & \infty & \infty & 0 & -5 & \infty \\ \infty & 7 & \infty & \infty & 0 & \infty \\ \infty & 5 & 10 & \infty & \infty & 0 \end{bmatrix}$$

$$D(2) = \begin{bmatrix} 0 & \infty & \infty & \infty & -1 & \infty \\ 1 & 0 & \infty & 2 & 0 & \infty \\ 3 & 2 & 0 & 4 & 2 & -8 \\ -4 & \infty & \infty & 0 & -5 & \infty \\ 8 & 7 & \infty & 9 & 0 & \infty \\ 6 & 5 & 10 & 7 & 5 & 0 \end{bmatrix}$$

$$D(3) = \begin{bmatrix} 0 & \infty & \infty & \infty & -1 & \infty \\ 1 & 0 & \infty & 2 & 0 & \infty \\ 3 & 2 & 0 & 4 & 2 & -8 \\ -4 & \infty & \infty & 0 & -5 & \infty \\ 8 & 7 & \infty & 9 & 0 & \infty \\ 6 & 5 & 10 & 7 & 5 & 0 \end{bmatrix}$$

$$D(4) = \begin{bmatrix} 0 & \infty & \infty & \infty & -1 & \infty \\ -2 & 0 & \infty & 2 & 0 & \infty \\ 0 & 2 & 0 & 4 & -1 & -8 \\ -4 & \infty & \infty & 0 & -5 & \infty \\ 8 & 7 & \infty & 9 & 0 & \infty \\ 3 & 5 & 10 & 7 & 2 & 0 \end{bmatrix}$$

$$D(5) = \begin{bmatrix} 0 & 6 & \infty & 8 & -1 & \infty \\ -2 & 0 & \infty & 2 & -3 & \infty \\ 0 & 2 & 0 & 4 & -1 & -8 \\ -4 & 2 & \infty & 0 & -5 & \infty \\ 8 & 7 & \infty & 9 & 0 & \infty \\ 3 & 5 & 10 & 7 & 2 & 0 \end{bmatrix}$$

$$D(6) = \begin{bmatrix} 0 & 6 & \infty & 8 & -1 & \infty \\ -2 & 0 & \infty & 2 & -3 & \infty \\ -5 & -3 & 0 & -1 & -6 & -8 \\ -4 & 2 & \infty & 0 & -5 & \infty \\ 5 & 7 & \infty & 9 & 0 & \infty \\ 3 & 5 & 10 & 7 & 2 & 0 \end{bmatrix}$$

10.2

We begin by modifying the Floyd-Warshall algorithm by adding a line that adds the vertices that make the path shorter to the list of predecessors for that path. In the end we return this array with all the predecessors for every path. We first create an array of lists so that we have a list of all the predecessors for every path. Then we initialize this list with the source vertex for the path. So for a path from 1 to 4 we initialize the list with vertex 1. If we find a shorter path we add this vertex to the list.

```
1   Floyd-Warshal2()
2   {
3       for (k = 1 to n)
4       {
5           D^k = new n x n matrix;
6           Set all values in D^k to infinity.
7           PI^k = new n x n matrix of lists;
8           Add the node itself to the list for its path in PI^k
9           for(i = 1 to n)
10          {
11              for(j = 1 to n)
12              {
13                  minVal = min (D^(k-1)[i][j], D^(k-1)[i][k] + D^(k-1)[k][j
                        ]);
14                  if (minVal < D^k [i][j])
15                  {
16                      D^k [i][j] = minVal; // Set the new shortest path
17                      PI^k.push_back(k);   // Add the new node to the path
18                  }
19              }
20          }
21      }
22      return pair(D^n, PI^n)
23  }
```

After we have the list of all the paths in PI^k we define PrintPaths2 to print all the values. To print them we first check whether the path has more than one node, if this is the case there is a path. Otherwise there is no path and we print an error.

```
1   PrintPaths2(PI^k)
2   {
```

```
3         for(path in PI^k)
4         {
5             if(length path == 1) -- It only contains the node itself (there is
                  no path)
6             {
7                 print ("error no path was found");
8             }
9             else
10            {
11            s = ""
12            foreach(node in path)
13            {
14                s ++ nodeId
15            }
16            print(s)
17            }
18        }
19    }
```

10.3

1. D.
2. B.
3. A.
4. C.
5. C.

10.4

We can do this using the following recursion equation:

$$local\_max[0] = v[0]$$
$$local\_max[n+1] = max(v[n+1], v[n+1] + local\_max[n]$$

Where $v[i]$ is the value of that index.

To do this we start at the right hand side of the list and we work towards the left. We then calculate the sum of every possible sub array from that point to the left. This will give us the local maximum for that point. Then when we calculate this for all the values from the right to the left. When we have done this we can look at the calculated sums and find the highest one from all the possible sub arrays.

This gives a linear time complexity since we run through the list only once to calculate the local maxima for all the sub arrays.

10.5

To solve this problem we first identify the overlapping sub problems. These problems are that we find the minimum penalty when we travel from the beginning of the route to hotel h_i where h_i is one of the hotels on the path.

We can then define the following recurrence equation for this sub problem:

$minimum\_penalty(h_0) = 0$, since 0 is the starting point and we have not traveled at all yet.

$minimum\_penalty(h_i) = min((minimum\_penalty(0) + (200 - (a[i] - a[0]))^2), (minimum\_penalty(1) + (200 - (a[i] - a[1]))^2), ..., (minimum\_penalty(i-1) + (200 - (d[i] - d[i-1]))^2))$

Where d[x] is the distance that hotel h_x is at on the path.

Thus the minimum penalty for reaching hotel h_i is found by trying all stopping places for the previous day and adding the penalty for the current day and taking the minimum of those.

In order to find the path, we store in a separate array which hotel we had to travel from in order to achieve the minimum penalty for that particular hotel. We can then reverse the final array with the hotels and find the path.

The time complexity of this algorithm is $\mathcal{O}(n^2)$ since for every hotel we go through the list of all other hotels to find the minimum penalty for that hotel.