Le-6. Graph Algorithms and Algorithmic Paradigms

1. Solution not included, see the textbook G&T 4th ed. 13.7.1 and 13.6.2.

2. (a) Use Dijkstra’s algorithm. The algorithm computes the costs in non-decreasing order. We can modify it so that the computation stops when the affordable cost is exceeded at some vertex.

   (b) The following table shows the tentative and the final (bold) costs associated with the vertices of the graph in the consecutive steps of the computation.

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
<tr>
<td>0</td>
<td>50</td>
<td>60</td>
<td>200</td>
<td>160</td>
<td>$\infty$</td>
</tr>
<tr>
<td>50</td>
<td>60</td>
<td>170</td>
<td>160</td>
<td>$\infty$</td>
<td>210</td>
</tr>
<tr>
<td>60</td>
<td>170</td>
<td>120</td>
<td>$\infty$</td>
<td>210</td>
<td></td>
</tr>
<tr>
<td>150</td>
<td>120</td>
<td>190</td>
<td>210</td>
<td></td>
<td></td>
</tr>
<tr>
<td>150</td>
<td>170</td>
<td>210</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>170</td>
<td>210</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

3. The problem reduces to finding minimum spanning tree. It can be solved by Kruskal algorithm (p.443).

   Selecting bridges in order of increasing length and checking the connectivity condition we obtain:

   (I3,I6)
   (I3,I5)
   (I5,I6) rejected: connects vertices of a connected component
   (I4,I5)
   (I3,I4) rejected: connects vertices of a connected component
   (I1,I5)
   (I1,I4) rejected: connects vertices of a connected component
   (I2,I4)
Now all islands are connected: total cost 81

4. Perform the topological sort of the graph. For each vertex taken in the topological order perform the relaxation of the outgoing edges. See:

http://ww0.java4.datastructures.net/handouts/ShortestPath.pdf

5. (a) The following directed graph reflects the given connections between the airports: the vertices correspond to the airports. Two vertices are connected if there exists a direct flight between the airports.

![Diagram of a directed graph with vertices A, B, C, and D]

To each edge we assign a cost equal to the flight duration. To solve the problem we modify Dijkstra’s algorithm. The modified algorithm will label each vertex with two values: the earliest arrival time from a given start vertex and the previous visited vertex. As in Dijkstra’s algorithm the labeling may be preliminary or final. The initial preliminary values are \( (t, A) \) for the start vertex and \( (\infty, A) \) for all remaining vertices. At every step the algorithm:

- Selects a vertex \( M \) with preliminary labeling that has the earliest arrival time \( t_M \) and changes its labeling to final;

- for each vertex \( N \) connected with \( M \):
  - it searches for the flight \( f \) from \( M \) to \( N \) with departure time \( d_f \) as early as possible after \( t_M \) but at earliest 45 min after \( t_M \) unless \( M \) is the start vertex.
  - using the flight duration it computes the arrival time \( a_f \) of \( f \) at \( N \).
  - if the preliminary arrival time in the actual label of \( N \) is later than \( a_f \) the label of \( N \) is replaced by \( (a_f, M) \)

The computation ends as soon as the node \( Y \) obtains the final labeling. The final labels of the graph vertices determine the itinerary.

The correctness of the algorithm follows from the correctness of Dijkstra’s algorithm.

(b) We illustrate the computation by showing the labels of the vertices at subsequent steps of the computations. At each state the label of a node \( Z \) is shown as \( Z(a, F) \), where \( a \) denotes the arrival time and \( F \) the previous stop.

\[
A(10.00, A), B(\infty, A), C(\infty, A), D(\infty, A)
\]

\[
A(10.00, A), B(13.45, A), C(12.00, A), D(\infty, A)
\]

\[
A(10.00, A), B(13.35, C), C(12.00, A), D(18.50, C)
\]
$A(10.00, A), B(13.35, C), C(12.00, A), D(15.35, B)$

The itinerary is thus $A \ d10.15\ C\ a12.00\ d12.50\ B\ a13.35\ d14.25\ D\ a15.35$

e) The main difference between Dijkstra’s algorithm and ours is that at each step our algorithm has to search in the tables of flights between the selected node $M$ and each of its neighbours. Assume that these tables are associated with the edges of the graph and that maximal length of such a table does not exceed $m$. Assume that we use a binary search for connecting flights. Assume also that the graph is dense and has $n$ nodes, thus close to $n(n-1)$ edges. As discussed in p.450 in that case an efficient implementation of Dijkstra’s algorithm runs in time $\Theta(n^2)$. In the worst case this algorithm examines each edge of the graph for updating the labelling of vertices. However a single update in Dijkstra’s algorithm is done in constant time while in our algorithm it requires a search in a flight table, in time $\log m$. Hence the complexity of the algorithm is $O(n^2 \log m)$.

If the graph is not dense, one can take as a basis another implementation of Dijkstra’s algorithm, discussed in p. 450. As explained in p.450, the latter runs in time $O((n + e) \log n)$, where $e$ is the number of edges. As in our modification each edge requires $O(\log m)$ time the modified algorithm will run in time $O(n \log n + e \log n \log m)$.

6. a) The main part of the algorithm is a loop computing the sum of the values $a_i x^i$ for $i = 0, 1, \ldots, n$. Assume that addition and multiplication take constant time. Thus, we perform $n+1$ additions and $1+2+\ldots+(n+1) = \frac{(n+1)(n+2)}{2}$ multiplications. Hence the algorithm has time complexity $\Theta(n^2)$.

b) We notice that in the above version of the algorithm $x^2$ is computed $n-2$ times, $x^3$ is computed $n-3$ times, ..., $x^i$ is computed $n-i$ times. To avoid this, we can take the following equivalent version of the polynomial as the basis of the algorithm:

$$a_0 + x(a_1 + x(a_2 + \cdots + x(a_{n-1} + a_n x)\ldots))$$

Here we only need $n$ additions and $n$ multiplications so that the time complexity is $\Theta(n)$. This can be seen as an example of dynamic programming paradigm, since the value of $x^i$ is computed only once for each $i = 1, \ldots, n$.

7. See G&T 4th ed. 12.4.2. Also Dijkstra’s Algorithm and Kruskal’s Algorithm are greedy algorithms.
8. The frequencies are:

space: 7, a: 1, c: 1, d: 2, g: 1, h: 1, n: 1, o: 7, p: 2, r: 1, s: 4, t: 5

\[
\begin{array}{cccc}
\text{space} & 0 & / & \backslash \\
7 & 7 & 8 & 11 \\
/ & \backslash & / & \backslash \\
4 & 4 & t & 6 \\
/ & \backslash & / & \backslash \\
d & 2 & 2 & 2 \\
2 & \backslash & / & \backslash & \backslash & \backslash & 2 & 4 \\
a & c & g & h & n & r \\
\end{array}
\]

post step: 111001111111011111100011110

9. The principle of dynamic programming.
The array:

\[
\begin{array}{cccccccc}
-1 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
-1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\
2 & 0 & 0 & 0 & 1 & 2 & 2 & 2 & 2 & 2 \\
3 & 0 & 0 & 0 & 1 & 2 & 2 & 3 & 3 & 3 \\
4 & 0 & 0 & 0 & 1 & 2 & 2 & 3 & 3 & 3 \\
5 & 0 & 0 & 0 & 1 & 2 & 2 & 4 & 4 & 4 \\
6 & 0 & 0 & 0 & 1 & 2 & 2 & 3 & 4 & 4 & 5 \\
\end{array}
\]

The longest common subsequence is: ACDFG

10. The recursive procedure based on the definition for executes in time \( T(n) = 1 \)
for \( n = 0,1 \) and in time \( T(n) = c + T(n - 1) + T(n - 2) > 2T(n - 2) \), hence
\( T(n) \in \Omega(2^n) \). An iterative procedure that starts computation from 0 and keeps
at each step only two previous Fibonacci numbers executes in linear time.

11. (a) Divide and Conquer.
   (b) Merge Sort. The auxiliary procedure bar corresponds to Merge.

12. (a) \( \text{cost}(x_{a,b}) = x_{a,b} + \max(\text{cost}(x_{a+1,b}, \text{cost}(x_{a+1,b+1})) \)
For every row the number of calls is doubled:

\[
T(n) = \sum_{i=1}^{n} 2^{i-1} = \frac{2^n - 2^0}{2 - 1} = 2^{n-1} \in \Theta(2^n)
\]
\( \Theta \) gives both the upper- and the lower bound.

(b) For every \( x_{i,j} \) save the value \( c_{i,j} \) which is the maximal cost from \( x_{i,j} \) down to the lowest level. Compute all \( c_{i,j} \) bottom-up.

\[
\text{for } i \text{ from } 1 \text{ to } n \text{ do} \\
\quad c_{n,j} := x_{n,j} \\
\text{for } i \text{ from } n \text{ downto } 1 \text{ do} \\
\quad \text{for } j \text{ from } 1 \text{ to } i \text{ do} \\
\quad \quad c_{i,j} := x_{i,j} + \max(c_{i+1,j}, c_{i+1,j+1})
\]

(c) Divide-and-conquer and dynmaic programming, respectively.