MTAT.03.238 Advanced Algorithmics

Home Assignment 9.

Deadline: April 20-21, 2011

1. Review of Algorithmic Techniques

Fill in the following table by marking, for each algorithm, those generic algorithmic techniques that were employed in its design. Think carefully: what is the exact problem each algorithm solves and what approach its authors have taken to tackle it. You might see more than one suitable option for some algorithms.

<table>
<thead>
<tr>
<th>Algorithm / Problem Statement</th>
<th>Exhaustive Search</th>
<th>Iterative improvement</th>
<th>Greedy</th>
<th>Divide &amp; Conquer</th>
<th>Single recurrence</th>
<th>Multiple recurrence / DP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bellman-Ford’s algorithm (SSSP)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary search</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dijkstra’s algorithm (SSSP)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Floyd-Warshall’s algorithm (APSP)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ford-Fulkerson’s algorithm (Max Flow)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prim’s algorithm (MST)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Topological sorting</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kruskal’s algorithm (MST)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Linear search</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Radix sort</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Quicksort</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Write out the time complexity of each algorithm near each line.
2. Adjacency matrix algebra

Let \( G = (V, E) \) be an undirected unweighted graph with \( n \) vertices. Let \( A \) be the \( n \times n \) adjacency matrix of \( G \). Define graph \( H = (V', E') \), where \( V' = V \) and

\[ E' = \{ (u, v) \mid u \neq v \text{ and there is a path of length exactly 2 between } u \text{ and } v \text{ in } G \} \]

i.e., the new graph has the same vertices as \( G \), but its edges correspond to paths of length 2 in \( G \) except for loops. Let the adjacency matrix of \( H \) be \( B \). Show how \( B \) may be computed from \( A \times A \).

Would it be possible if the graph was weighted? What if there were negative weights?

3. Witnesses

Let \( G = (V, E) \) be a (directed or undirected) graph with \( n \) nodes, and let \( W \) be an \( n \times n \) matrix of shortest path witnesses. More precisely, for every two nodes \( u, v \in V \):

\[ W_{u,v} = \begin{cases} 
  v & \text{if } (u, v) \in E, \\
  k & \text{if } (u, v) \notin E, \text{ but there exists a path between } u \text{ and } v, \\
  \text{nil} & \text{otherwise},
\end{cases} \]

where \( k \notin \{u, v\} \) is any vertex on a shortest path from \( u \) to \( v \).

Write a function \( \text{PRINTPATH}(u, v) \) which outputs the whole shortest path from \( u \) to \( v \).

4. Landmark embedding

Let \( G = (V, E) \) be a graph. Suppose you have selected four landmark vertices \( \ell_1, \ell_2, \ell_3, \ell_4 \in V \) and computed shortest path distances from each vertex in the graph to the four landmarks. You examine the results and find out that the computed distances from vertex \( v_1 \) to the four landmarks are:

\[ d[v_1] = (5, 3, 1, 7) \]

That is, \( d(v_1, \ell_1) = 5 \), \( d(v_1, \ell_2) = 3 \), etc.

You also observe, that:

\[ d[v_2] = (2, 1, 2, 3) \]
\[ d[v_3] = (4, 2, 3, 1) \]

You conclude that there must have been an error in your distance computation algorithm. Why?
5. Multiple-seed Dijkstra algorithm

Let $G = (V, E)$ be a weighted graph with with positive edge weights. Let $\ell_1, \ell_2, \ldots, \ell_k \in V$ be a pre-selected set of $k$ landmarks. The task is to compute, for each vertex $v \in L$ the distance to the closest landmark $\ell_i$ in the set.

In the lecture we discussed that this could be done, conceptually, by introducing a new “virtual” vertex into the graph and running the Dijkstra algorithm from this vertex. In a practical implementation of this idea, however, it is not necessary to actually add any vertices to the graph. Present a modification of the Dijkstra algorithm that does not modify the graph structure.

6*. Algorithmic complexity attacks

In the lecture you have seen an example of an application that had to take a lot of untrusted input and store it in a key-value store, implemented as a simple hash map. Consequently, the application was vulnerable to an algorithmic complexity attack on its hash map. Come up with other potential or actual real-life example, where a similar situation might take place.

7*. Euler tour trees

The figure below presents a tree together with its Euler tour representation.

![Directed rooted tree with edge numbering](image)

**Associated arrays:**

- **edge[ ] =** 
  
- **index[ ] =** 
  
- **depth[ ] =** 
  
- **first[ ] =** 
  
- **last[ ] =** 

Note the following:

- The `edge[ ]` array lists edge “names” in order of the Euler tour traversal starting from root. You can ignore this array for the purposes of this task.

3
• The **index[]** array stores, for each edge in the tour, the vertex ID that is visited at the particular step of the traversal. E.g. the first step of the tour leaves vertex 15, hence **index[0] = 15**.

• The **depth[]** array stores, for each step of the tour, the treedepth of the vertex visited at that step.

• The **first[v]** and **last[v]** store, for each vertex **v** the steps of the tour during which that vertex is visited for the first and last time. For example, vertex 10 is first visited on tour step 1 (i.e. **index[1]=10**), and has its last visit on tour step 9 (**index[9]=10**).

Suppose you are interested in being able to quickly locate the **lowest common ancestor (LCA)** for any two vertices **u** and **v** in the tree. For example, the LCA of 9 and 8 is 2, the LCA of 11 and 6 is 14, the LCA of 12 and 2 is 15, etc.

Now assume that you have at your disposal a method of indexing any array **a[]**, so that queries of the form **argmin(a[i..j])** can be answered in \(O(\log n)\) time. It turns out that by indexing the array **depth** in this way it is possible to answer LCA queries fast. How?

Show how the algorithm would process the query LCA(7, 4).

**Hint.** [http://users.informatik.uni-halle.de/~jopsi/dpar03/chap7.shtml](http://users.informatik.uni-halle.de/~jopsi/dpar03/chap7.shtml)