Algorithms Re-Exam TIN093/DIT600

Course: Algorithms

Course code: TIN 093 (CTH), DIT 600 (GU)

Date, time: 22nd December 2016, 8:30–12:30

Building: M

Responsible teacher: Peter Damaschke, Tel. 5405

Examiner: Peter Damaschke

Exam aids: one A4 paper (both sides), dictionary, the printed Lecture Notes; any edition of Kleinberg, Tardos: "Algorithm Design".

Time for questions: around 9:30 and around 11:30.

Solutions: will appear on the course homepage.

Results: will appear in ladok.

Point limits: CTH: 28 for 3, 38 for 4, 48 for 5; GU: 28 for G, 48 for VG; PhD students: 38. Maximum: 60.

Inspection of grading (exam review):
Times will be announced on the course homepage.

Instructions and Advice:

- First read through all problems, such that you know what is unclear to you and what to ask the responsible teacher.
- Write solutions in English.
- Start every new problem on a new sheet of paper.
- Write your exam number on every sheet.
- Write legible. Unreadable solutions will not get points.
- Answer precisely and to the point, without digressions.
  Unnecessary additional writing may obscure the actual solutions.
- Motivate all claims and answers.
- Strictly avoid code for describing an algorithm.
  Instead *explain* how the algorithm works.
- If you cannot manage a problem completely, still provide your approach or partial solution to earn some points.

Remark: Points are not always proportional to the length or difficulty of a solution, but they also reflect the importance of a topic or skill.

Good luck!

Problem 1 (12 points)

Let x and y be two integers, each with at most n digits. We wish to calculate $z := \lfloor x/y \rfloor$, that is, the ratio x/y rounded to the next smaller integer. Clearly, z equals the largest integer t such that $yt \leq x$. This suggest a possible algorithm for computing z: Try several integers t using binary search and test whether $yt \leq x$ or $yt > x$ each time. For simplicity let us use the "standard" $O(n^2)$ time multiplication algorithm in order to compute the product yt in every step.

1.1. How much time would this division algorithm need? Give the time bound as a function of n in O-notation, and explain it. (But think twice: First, how many steps of binary search are needed here?) (8 points)

1.2. The usual algorithm for integer division follows the same idea but is more clever (and faster): Instead of multiplying y each time with an entire number t it determines the digits of the ratio z one by one. In every step it multiplies only one digit of z with y and then performs one subtraction of numbers. How much time does the usual division algorithm need? Again, give the time bound as a function of n in O-notation. A brief informal explanation of the time bound is enough. (4 points)

Problem 2 (10 points)

Let us review the dynamic-programming algorithm for sequence comparison, i.e., computing the edit distance of two strings of maximum length n. It computes all values $OPT(i, j)$ indicating the edit distance of the prefixes of length i and j, for all $i \leq n$ and $j \leq n$. These values are stored in a two-dimensional array, for the calculation itself and for the backtracing procedure. Since the values are bounded by n, we need $O(\log n)$ bits to store each value. In total these are $O(n^2 \log n)$ bits. However, if n is huge, it may be worthwhile or even necessary to save memory space. Here is an idea:
We compute the numbers $OPT(i, j)$ row by row, but we store them only for the current row i and the previous row $i - 1$. As soon as row i is finished, we also keep in mind, for each (i, j), whether the minimum came from $(i - 1, j - 1)$, $(i - 1, j)$, or $(i, j - 1)$. Since these are only three cases, $O(1)$ bits are enough to store this information for each (i, j). After that, we forget all numbers $OPT(i - 1, j)$.

2.1. How many bits are now, at most, simultaneously stored in memory? Give an upper bound in O-notation and explain it. The bound should be better than $O(n^2 \log n)$. (4 points)

2.2. Explain why, for calculating the $OPT(i, j)$ values, it is indeed sufficient to memoize always the last two rows. (2 points)

2.3. Describe how the backtracing procedure works for this modified algorithm. (Recall that the $OPT(i, j)$ values have been erased, thus we cannot use them any more.) So, how do we get an optimal alignment of the two strings, and how much time does the backtracing take now? (4 points)

Problem 3 (12 points)

Consider a directed acyclic graph (DAG) G whose nodes are already given in a topological order $v_1, \ldots, v_n$. Let $L(i, j)$ denote the length of the directed edge from v_i to v_j, for $i < j$. Let $OPT(j)$ be the distance (i.e., length of a shortest directed path) from v_1 to v_j. The distances can be computed by dynamic programming, using the formula

$$OPT(j) = \min_{i<j} OPT(i) + L(i, j)$$

where the minimum is taken over all indices $i < j$.

Now assume that some edges in the graph can become faulty and are not available any more. Formally, a faulty edge is deleted, or its length becomes infinite: $L(i, j) := \infty$. For a given integer f we want to compute the guaranteed minimum path length (distance) from v_1 to v_n under the assumption that at most f edges in G are faulty.
For this purpose we define a function to be used in dynamic programming: Let $OPT(j, k)$ denote the smallest number d such that there still exists some directed path of length d from v_1 to v_j, in every graph obtained from G by deleting k edges. (Note that $d = \infty$ if all connections are broken.) We are interested in $OPT(n, f)$.

Your task: Describe how you can compute all $OPT(j, k)$ efficiently. Provide a time bound for the resulting dynamic-programming algorithm. (12 points)

Problem 4 (10 points)

In the plane, an object can be localized if we can measure its usual, Euclidean distances to three devices with known coordinates. However, Euclidean distance is not always appropriate, e.g., due to obstacles in the plane. Let us consider the localization problem for arbitrary distance functions, in a discretized version, and with some number k of devices:
Let $G = (V, E)$ be an undirected, connected graph. For simplicity we assume that all edges have length 1. Some devices are placed on k nodes of G, and these nodes are known. The graph is completely known, too. An object O is located at an unknown node $u \in V$. Every device M can measure its distance $d(u, v)$ to O, where v is the node where M is placed. For clarity: The measurement returns only the number $d(u, v)$ but not the node u.

Your task: Once the distance values are collected and sent to a central computer, which calculations must be done in order to determine u? (Describe your algorithm in sufficient detail.) How much time do they take, as a function of k and of the size of the graph (in O-notation)? Is the result u always uniquely determined? (10 points)

Problem 5 (8 points)

The following structure is used in a heuristic for the Travelling Salesman problem, however we cannot explain the background here.
Let $G = (V, E)$ be an undirected graph with positive edge lengths. For simplicity assume that all edge lengths are distinct. Let $v \in V$ be some distinguished node, and let H be the graph obtained from G be deleting v and all edges incident to v. (Thus, H has the node set $V \setminus \{v\}$.) A *1-tree* is defined as a subset $F \subset E$ of the edge set E, consisting of a spanning tree of H and two further edges that are incident to v. (We suppose that the degree of v is at least 2.) The length of a 1-tree F is defined as the sum of lengths of all edges in F.
The problem is to compute a 1-tree of minimum length. We propose the following greedy algorithm: Compute a minimum spanning tree (MST) of H, using Kruskal's algorithm. Let T denote the edge set of this MST. Among all edges of E that are incident to v, let e and f be the two shortest edges. Output $F := T \cup \{e, f\}$.

Would this algorithm indeed compute a 1-tree F with minimum length?
If so, give a correctness proof. If not, give a counterexample. (8 points)

Problem 6 (8 points)

The software company SuperStarBugs offers you a program that can efficiently compute largest independent sets in graphs. In an information folder they describe their innovative program as follows.
"We take any input graph G and rewrite it as a bipartite graph B: Every node v of G becomes a node on the left side of B. Every edge e of G becomes a node on the right side of B. For every edge uv in G we insert two edges ue and ve in B. Now every independent set I of k nodes in G yields a matching with k edges in B. This is easy to see: For every node v in I we select an incident edge e. The k edges ve of B generated in this way form a matching. (Indeed, since no two nodes in I are joined by edges, all k selected edges e are different.) As the final step we invoke an efficient algorithm from our partner NetworkFlows, that computes a maximum bipartite matching. Altogether this solves your maximum independent set problem."

6.1. You refuse to buy this program: Argue why the offered program cannot be correct, referring to your knowledge on polynomial-time conplexity or NP-completeness of the mentioned graph problems. (2 points)

6.2. The representative of the company you talk to is, unfortunately, not familiar with NP-completeness and cannot follow your argument (from 6.1). So you have to convince the person in a different way: What precisely is the mistake in the algorithm described in the folder? Explain. (6 points)

Solutions (attached after the exam)

1.1. Let us use the decimal system (but due to O-notation, the base is arbitrary). Numbers with n digits are no larger than 10^n. Searching for z in this range requires $\log_2 10^n = O(n)$ steps of binary search. In every such step we do a multiplication that costs $O(n^2)$ time and a comparison that takes only $O(n)$ time. Altogether this yields $O(n^3)$ time. (8 points)

1.2. We obtain every digit of z by trying $O(1)$ digits, and for each candidate, the multiplication with y and the subtraction take only $O(n)$ time. Altogether this yields $O(n^2)$ time. (4 points)

2.1. At each time we store $O(n)$ integers of size at most n, hence with a total of $O(n \log n)$ bits, and we store $O(n^2)$ bits permanently. This yields $O(n \log n + n^2) = O(n^2)$, which is better than $O(n^2 \log n)$ by an $O(\log n)$ factor. (4 points)

2.2. This holds simply because the formula for $OPT(i, j)$ recurs only to the values for $(i-1, j-1)$, $(i-1, j)$, and $(i, j-1)$, but never to arguments smaller than $i - 1$. (2 points)

2.3. Starting from (n, n), simply go from (i, j) to the stored pair $(i-1, j-1)$, $(i-1, j)$, or $(i, j-1)$. These three cases correspond to an alignment of two symbols (with a mismatch or not), an insertion, or a deletion of a symbol. Trivally we are done after $O(n)$ such steps. (4 points)

3. For $k = 0$ faults we have $OPT(j, 0) = \min_{i<j} OPT(i, 0) + L(i, j)$, because every path from v_1 to v_j is composed of a shortest path from v_1 to some v_i and a final edge to v_j, and we can choose some v_i that yields the minimum. (Actually the formula is already stated in the exercise.)

Now consider any fixed $k > 0$. For every fixed j, some of the edges $v_i v_j$ can be faulty. Let g denote their number. Then at most $k - g$ edges ending earlier than v_j were faulty. In the worst case, the g edges $v_i v_j$ that yield the g smallest sums $OPT(i, k-g) + L(i, j)$ are faulty. Thus, we must take the $(g+1)$-st smallest value among all sums $OPT(i, k-g) + L(i, j)$, where $i < j$. Let us denote this value $M(j, k, g)$. Finally, since g can be any number up to k, we have to take the worst case and set $OPT(j, k) := \max_{g \le k} M(j, k, g)$.

The variables j and k have n and f values, respectively, thus $O(nf)$ values $OPT(j, k)$ must be calculated. For every fixed g we considered all $i < j \le n$

and did $O(n)$ calculations. Since $g \leq k \leq f$, this was done $O(f)$ times. The total time bound is therefore $O(n^2 f^2)$. (12 points)

4. Let n and m denote the number of nodes and edges, respectvely. We must find a node u such that all k values $d(u, v)$ equal the measured values. We do k times BFS, starting from every node with a device. This costs $O(km)$ time. Then we compare the n vectors of n distances with the measured distances. This takes $O(kn)$ time, which is contained in $O(km)$. The node u is in general not unique (we only obtain candidates for the node u). A trivial counterexample is a clique: Since all distances are 1, the measurements give no information in this case. (10 points)

5. The algorithm is correct. By definition, an arbitrary spanning tree of H together with an arbitrary pair of edges incident to v form a 1-tree. Hence we can independently compute an MST and add the two shortest edges at v, in order to obtain the best 1-tree. (Note that it is not necessary to prove correctness of Kruskal's algorithm here; this fact is already known. The only point is to argue that we can combine the two parts of a 1-tree.) (8 points)

6.1. The company pretends to use a polynomial-time reduction from Independent Set to Bipartite Metching. However, the latter problem is polynomial-time solvable, and the former one is known to be NP-complete. Hence such a reduction cannot exist, unless P=NP. (2 points)

6.2. It is correct that every independent set of k nodes in G yields a matching with k edges in B in the way described. But the developers forgot that the reverse implication must hold as well. (The problem instance constructed in a reduction must be equivalent to the given instance.) This is not the case here. Two edges of a matching in H may well correspond to two nodes in G that are adjacent. (6 points)