Algorithms Exam TIN093/DIT602

Course: Algorithms

Course code: TIN 093 (CTH), DIT 602 (GU)

Date, time: 24th October 2020, 14:00–18:00

Building: –

Responsible teacher: Peter Damaschke, Tel. 5405, email `ptr@chalmers.se`

Examiner: Peter Damaschke

Exam aids: As this is an individual home exam, all aids are allowed, except any help from other persons.

Time for questions: email at any time during the exam

Solutions: will be published after the exam

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): to be announced.

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.
- Write your name and personal number in the submitted file.
  Start every new problem on a new page.
- Submit your solutions as **one PDF**, preferably produced with some text editor. If you scan handwritten pages, they must be legible (with characters being easy to recognize and not too pale). Unreadable solutions will not get points. Use the submission system.
- Answer precisely and to the point, without digressions.
  Unnecessary additional writing does not only cost time.
  It may also obscure the actual solutions.
- But motivate all claims and answers.
- Strictly avoid code for describing a complex algorithm.
  Instead *explain* in your words how the algorithm works.
- If you cannot manage a problem completely, still provide your approach or partial solution to earn some points.
- Facts from the course material can be assumed to be known.
  You don't have to repeat their proofs.
- **Most important: Always write coherent text in your own words describing your own understanding.**

Remark: The number of points is not always "proportional" to the length or difficulty of a solution, but it may also be influenced by the importance of the topics and skills.

Good luck!

Problem 1 (5 points)

Some algorithm for converting an integer n given in decimal notation into binary notation can be sketched as follows. Generate 2^i in decimal notation, for all integers $i = 0, 1, 2, \ldots, k$, where k is the largest exponent with $2^k \leq n$. Then do the following for $i = k, k-1, k-2, \ldots, 0$. If $2^i \leq n$ then set $n := n - 2^i$ and write 1. If $2^i > n$ then do not change n and write 0. (Correctness is pretty obvious, you need not prove it here.)

Derive a time bound for this algorithm, as a function of k, in O-notation. Make sure to explain it in sufficient detail. Operations with single digits are assumed to be the elementary operations. Your bound should not be more generous than necessary.

Problem 2 (15 points)

Let d be a fixed positive integer. We call a graph *d-sparse* if its nodes can be ordered such that every node v is adjacent to at most d nodes to the right of v (and to arbitrarily many nodes to the left of v).
These graphs are a natural generalization of trees; one can show that the trees are exactly the connected 1-sparse graphs. However, the problem we consider here is: Given a graph G with n nodes, decide whether G is a d-sparse graph, and if so, produce an ordering of the nodes with the above property. The following greedy algorithm easily comes to mind:

for $i = 1$ to n do the following:
pick any node v of smallest degree,
put it at the i-th position of the ordering,
and delete v and its incident edges from G

Clearly, if this algorithm goes through, and every selected node v has a degree at most d in the remaining graph, then G is d-sparse. But the concern is whether the converse is also true.

2.1. Prove that the proposed greedy algorithm is, in fact, correct. You need to show: If G is d-sparse, then the algorithm will always succeed. Hint: Induction on the number of nodes might be a good idea. If you do not manage induction, you may give an informal (but still conclusive) argument. (6 points)

2.2. We slightly "relax" the greedy algorithm as follows: Let us always pick any node with degree at most d (not necessarily with the smallest degree). Is this modified algorithm still correct? If you think yes, briefly argue why. If you think no, give a counterexample. (2 points)

2.3. Explain how to implement the algorithm in 2.1 in such a way that it runs in $O(n)$ time. (Remember that d was fixed.) Your description may be informal, but it should be specific enough for deriving the time bound. Ideally, state the data structure(s) that you use. (7 points)

Problem 3 (10 points)

A "hidden subword" of a word A is any word obtained from A by deleting some symbols and closing the gaps between the remaining symbols. For instance, USE is a hidden subword of SUNSET (via -U-SE- as an intermediate step).

Consider the following problem: For two given words A and B of length n and m, respectively, find some longest common hidden subword C. That is, C must be a hidden subword of both A and B and have the maximum length among all words with this property.

Example, just for fun: The longest common subword hidden in
SCHOOL and COROLLARY is COOL.

Your task: Describe a dynamic programming algorithm that solves this problem in $O(nm)$ time.
Do not give pseudocode but provide the "recursive formula" for dynamic programming, including initial values. Motivate correctness and time bound. The details of backtracing can be omitted.

Problem 4 (5 points)

The $O(n \log n)$-time algorithm for finding two closest points among n given points in the plane was designed for the Euclidean distance. That is, the distance between two points (u, v) and (x, y) was $\sqrt{(u-x)^2 + (v-y)^2}$.

Now let us consider the Manhattan distance $|u - x| + |v - y|$ instead. This is the distance in a plane where one can "walk" on horizontal and vertical lines only, similarly to the avenues and streets in Manhattan.

The question: Does the mentioned algorithm still work correctly for the same problem with Manhattan distance, and with the same time bound? Motivate your answer: "yes, without any changes, because ...", or "yes, with the following modifications, because ...", or "no, it breaks down completely, because ...". (Of course, only one of these answers can be correct.) A short summarizing statement of a few lines is sufficient; no long "essay" is expected here.

Problem 5 (15 points)

We define a problem called Majority Independent Set as follows. Given an undirected graph $G = (V, E)$ with n nodes, does G contain an independent set with at least $n/2$ nodes?

5.1. Show that Majority Independent Set is NP-complete.

But be careful: This is not a trivial consequence of the NP-completeness of Independent Set, since our current problem is only a special case of it!

Hint: Give a polynomial-time reduction from Independent Set, by adding to the given instance of the Independent Set problem either a clique or another independent set, of suitable size. Do not forget to show equivalence of the instances. (10 points)

5.2. After elections in a country, it is known how many seats every political party has obtained in parliament. Now the parties want to find a coalition that holds the majority of the seats. However, certain pairs of parties definitely cannot work together in a coalition. Show that the problem of finding such a coalition is NP-complete, where the number of parties is considered to be the instance size.

Remarks: Here you can argue more informally (yet logically precise!). You can do this exercise also if you did not manage 5.1. From a practical point of view, NP-completeness is not really important here, as the number of political parties is very small. But there might be similar applications in other domains. (5 points)

Problem 6 (10 points)

The Longest Path problem in DAGs was motivated by finding critical (i.e., most time-consuming) paths is project plans with precedence constraints. Now we consider a slightly modified variant of this problem:

Given a DAG $G = (V, E)$ with positive edge lengths. and a specific node $v \in V$, find a directed path in G that contains v and is the longest path with this property.

Present some algorithm that solves this problem in linear time (in the number of nodes and edges), with correctness argument and time analysis.

Solutions (attached after the exam)

1. The $k+1$ powers of 2 have at most k digits each, and doubling every power takes $O(k)$ time. Hence all powers are generated in $O(k^2)$ time. Every comparison and subtraction takes $O(k)$ time as well. Hence the overall time bound remains $O(k^2)$. (5 points)

2.1. Suppose that the claim is true for graphs with n nodes. Let G be a graph with $n+1$ nodes. Since G is d-sparse, it has some ordering O with the desired property. We do not know O yet, but we know that there *exists* some node of degree at most d (for instance, the first node in O). Hence the algorithm will pick some node v with degree at most d. Let G' be the graph without v. Since deletion of nodes cannot produce more adjacent nodes, G' is also d-sparse. By the inductive hypothesis, the algorithm finds an ordering for G' as desired. Together with v as the first node, this yields an ordering for G. – This is exactly the same type of argument as for topological orderings of DAGs, albeit for undirected graphs, and for a very different graph property. (6 points)

2.2. True. The proof in 2.1 does not hinge on the fact that v has the smallest degree, but only on the degree being at most d. (2 points)

2.3. First we count the edges incident to every node. Since a d-sparse graph has at most dn edges and d is fixed, counting needs $O(n)$ time. (If more edges exist, we can abort the algortihm with a negative result.) We put all nodes of degrees $1 \ldots, d$ in d doubly-linked lists. In each of the n steps of the algorithm we take a node v from the first non-empty list, and we update the degrees of all neighbors of v (since v gets deleted), and move them to the correct lists. Since all actions need only $O(d)$ time for every node v, the time bound remains the same. – Extra remark: In the algorithm in 2.2, one list is enough, but this does not affect the time bound. (7 points)

3. One way is to show equivalence of the problem to a variant of String Editing where replacing of symbols is forbidden, however, here we present a solution from scratch. Let $OPT(i,j)$ be the length of a longest common substring of the prefixes of the given strings, of lengths i and j, respectively. Then we have $OPT(i,0) = OPT(0,j) = 0$ for all i and j, and

$$OPT(i,j) = \max\{OPT(i-1,j), OPT(i,j-1), OPT(i-1,j-1) + e_{i,j}\},$$

where $e_{i,j} = 1$ if the i-th symbol of A equals the j-th symbol of B, and $e_{i,j} = 0$ otherwise. Correctness is seen as follows. Whenever two symbols a from A and b from B contribute to a common substring, they must fulfill $a = b$, and they extend a common substring of the prefixes ending before a and b. This situation is represented by the third case in the formula. The first two cases just extend either of the prefixes, without adding a synbol to the common substring. We compute $OPT(n, m)$ and recover an optimal solution by backtracing. The time is $O(nm)$ since that many values are computed, each in $O(1)$ time. (10 points)

Clarification provided during exam: A "word" means any string of symbols.

4. The same algorithm can be used; the only difference is in the formula for the distance (and possibly in the result for a given set of points). The divide and conquer steps work exactly as before, and even the arguments in the time analysis would be literally the same. (5 points)

5.1. Consider an instance of the original Independent Set problem, consisting of a graph $G = (V, E)$ and an integer k. We construct a graph H as follows. If $k < n/2$ then we add s isolated nodes to G. Then G has an independent set of size k if and only if H has some of size $k+s$. Since we want $k+s = (n+s)/2$, we choose $s := n-2k$. If $k \geq n/2$ then we add to G a clique with c nodes and connect them by edges to all nodes of G. Then G has an independent set of size k if and only if H has. (Here the new nodes do not change the solution size.) Since we want $k = (n + c)/2$, we choose $c := 2k - n$. Polynomial time is obvious in both cases. Membership in NP is obvious, too: One can count the nodes in a given solution and check the absence of edges in polynomial time. (10 points)

5.2. Here the problem is to find, in a graph with weighted nodes (the parties with their election results), an independent set whose weight is at least half of the total weight. Membership in NP is obvious again: One can add weights and check the absence of edges in polynomial time. Furthermore, we can reduce Majority Independent Set to our problem, because the former problem is the special case where all node weights are equal. Finally, the result of 5.1 implies NP-completeness. (5 points)

6. Any solution must consist of a directed path P that ends in v and a directed path Q that begins in v. Since G is a DAG, P and Q cannot share

any nodes. Moreover, P and Q are longest paths ending and starting in v, respectively, otherwise we could replace any of them with a longer path and improve the solution. Hence, instead of developing a new algorithm, we can reduce our problem to two instances of the original Longest Path problem in DAGs, one in the subgraph of all nodes from which v is reachable, and one in the subgraph of all nodes reachable from v. These two subgraphs can first be determined by BFS, using once the reversed edges and once the given edges. (This preparatory step is not really necessary, but it may make the separation of the two sub-problems clearer.) Hence the overall time remains linear. (10 points)