Algorithms Re-exam TIN093[*]/DIT602

Course: Algorithms

Course code: TIN 093, TIN 092 (CTH), DIT 602 (GU)

Date, time: 20th December 2017, 8:30–12:30

Building: M

Responsible teacher: Peter Damaschke, Tel. 5405

Examiner: Peter Damaschke

Exam aids: one A4 paper (both sides), dictionary,
printed Lecture Notes and slides (possibly with own annotations),
any edition of Kleinberg, Tardos: "Algorithm Design".

Time for questions: around 9:30 and around 11:00.

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):
Time will be announced on the course homepage.

[*]also forTIN092

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.
- Facts that are known from the course material can be used.
  You don't have to repeat their proofs.

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 (14 points)

We are given a connected and undirected graph $G = (V, E)$ whose m edges have positive weights, and a subset $F \subset E$ of edges. The problem is to find a spanning tree in G that includes (at least) all the edges of F and has minimum total weight under this condtion. (In short one could say: We want a minimum spanning tree with some enforced edges. In the original problem we have $F = \emptyset$.)

1.1. Because of the set F, an instance may not have a solution at all. Give an algorithm that tests in $O(m)$ time whether a solution exists. (But in the positive case it is not yet required to compute the minimum weight.) Explain briefly why your algorithm is correct. (6 points)

1.2. Now, for all instances which do have solutions, we also want to compute a *minimum* spanning tree including F. We can no longer manage this in $O(m)$ time, but still efficiently: *Change the weights of all edges in F to* 0*, then run Kruskal's algorithm.*
Finish the description of this algorithm. In particular, specify the output that you would return. Then, explain why the algorithm, in fact, yields a valid and optimal solution. (6 points)

1.3. Is the necessary time (in O-notation!) higher than in Kruskal's algorithm for the "plain" Minimum Spanning Tree problem, or is it still the same? Motivate your answer. (2 points)

Problem 2 (10 points)

We wish to pack $2n$ objects in n boxes. Each box can accommodate exactly two objects, thus we must form n pairs of objects. However, all objects have different weights $x_1 < \ldots < x_{2n}$. (Assume they are already sorted.) In order to avoid any overly heavy boxes, we want to find a pairing of our $2n$ objects that minimizes the maximum weight of the pairs. We claim that the optimal solution is simply to pair up the largest with the smallest object, and to do this iteratively:

$$x_1 + x_{2n}, x_2 + x_{2n-1}, \ldots, x_{i+1} + x_{2n-i}, \ldots x_n + x_{n+1}$$

In order to prove this claim, consider in the following any four indices with $i < j < k < l$.

2.1. Show that replacing $x_i + x_j$ and $x_k + x_l$ with $x_i + x_l$ and $x_j + x_k$ makes the maximum of these two sums smaller. (2 points)

2.2. Show that replacing $x_i + x_k$ and $x_j + x_l$ with $x_i + x_l$ and $x_j + x_k$ makes the maximum of these two sums smaller as well. (2 points)

2.3. Finally conclude from 2.1 and 2.2 the optimality of the proposed solution. Hint: *First* consider the partner of x_{2n}, and assume it differs from x_1. (6 points)

Problem 3 (12 points)

Let X be a chain of m jobs that must be done sequentially, in a given order. Let Y be another chain of n jobs that must also be done sequentially, in a given order. Time is divided in time slots of equal lengths. Every job needs one time slot to be done.
For any two jobs $x \in Y$ and $y \in Y$ we also know whether they have a *conflict* or not. (A conflict could mean, for example, that x and y need the same resource.) Jobs x and y can be done in the same time slot, if and only if they have no conflict. Of course, jobs from the same chain can never be done in the same time slot. We want to finish all jobs in X and Y as early as possible, that is, minimize the total length of the schedule.
We define $OPT(i, j)$ as the minimum number of time slots needed to finish the first i jobs of X and the first j jobs of Y.

3.1. Provide a formula that allows to compute all $OPT(i, j)$ by dynamic programming. Give both the "main" formula for general i, j and the necessary initial values. (4 points)

3.2. Explain all terms in your formula(s), and their correctness. (6 points)

3.3. How much time would your resulting dynamic programming algorithm need? Explain why. (2 points)

Problem 4 (10 points)

In some applications the "cost" of a path in a network is determined by costs at the nodes rather than edge lengths. Formally: In a directed graph whose nodes v have costs $c(v)$, we define the cost of any directed path P as the sum of all $c(v)$, where the v are all nodes on P (including the first and last node of P).
Now let $v_1, \dots, v_n$ be the nodes of a DAG, in topological order. The problem is to compute the cost of a cheapest path from v_1 to v_n.

4.1. Describe an algorithm for this task. (5 points)

4.2. Argue why your algorithm correctly solves the problem. (3 points)

4.3. Give a time bound. It must, of course, be a correct upper bound for your algorithm, but not be unnecessarily generous. (2 points)

Problem 5 (14 points)

This is a very basic case of data compression: We wish to store a family of n unordered pairs, that is, sets with exactly two elements each. For convenience let us write sets $\{a, b\}$ simply as ab, omitting the comma and brackets. Assume that every element needs exactly one memory cell.
Clearly we might store our set family in $2n$ cells. But the sets have many common elements, so perhaps it is not necessary to store them all repeatedly. Instead of writing a family of m sets, which all share an element a, explicitly as $ab_1, ab_2, \ldots, ab_m$, we could "bundle" these sets and abbreviate them as $a : b_1, b_2, \ldots, b_m$ with the common element a as "header". This way we use only $m + 1$ cells rather than $2m$. (The space for punctuation symbols is not counted here, for simplicity.)
Here is a complete example. The set of unordered pairs
AN, AS, AT, BE, BY, DO, GO, HE, IT, ME, MY, NO, OF, OR, SO, TO, US, WE
may be abbreviated as six bundles
A:NS, E:BHMW, Y:BM, O:DGNFRS, S:U, T:AIO.
However, the headers can be selected in many different ways, leading to very different sets of bundles. Our problem, which we call Pair Compression, is now defined as follows: Given a family of pairs and a number c, can we store them as bundles that need in total at most c memory cells for the elements? (In the optimization version we would aim at minimizing c.)
Unfortunately, optimal compression is hard, already in this simple scenario. We will see below that our problem is NP-complete.

5.1. Pair Compression belongs to the complexity class NP. Explain in detail why. (3 points)

5.2. Describe a polynomial-time reduction from Vertex Cover to Pair Compression. (But remember that reductions are done between decision problems: "Does there exist a vertex cover of size at most k?" etc.). (4 points)

5.3. Show that your reduction is correct, in other words, that the Pair Compression instance constructed by your reduction is equivalent to the given Vertex Cover instance. (5 points)

5.4. Explain how the previous statements imply NP-completeness of Pair Compression. You may answer this even if you did not manage the reduction itself. (2 points)

Solutions (attached after the exam)

1.1. Claim: A spanning tree including F exists if and only if the edges of F do not form any cycles.
Clearly, this condition is necessary. Conversely, if the condition is satisfied, then the connected components formed by the edges of F are trees. Since the whole graph is connected, we can merge these components successively by adding further edges from E, until some spanning tree is built.
Thus, we only need to test the said condition. As we know, in linear time we can determine the connected components and check whether each one is a tree. (6 points)

1.2. In the end we output exactly the edges selected by Kruskal's algorithm, and we only change the costs of edges in F back to their true values. Correctness is seen as follows:
We know already that Kruskal's algorithm returns a minimum spanning tree. Moreover, it always chooses a cheapest edge that does not create cycles together with any previous ones. Since the edges in F have (temporarily) zero weight and do not form any cycles, the algorithm takes them first. Since every valid solution must include F, we get in fact a minimum spanning tree with this property. (6 points)

1.3. The only additional work is to change the weights of the edges in F, which takes $O(|F|) = O(m))$ time. The time for the actual computation is larger. As we do not care about constant factors, the total time bound does not increase. (2 points)

2.1. Clearly, $x_k + x_l$ was the maximum of the given sums, and both $x_i + x_l$ and $x_j + x_k$ are smaller than this. (2 points)

2.2. Similarly, $x_j + x_l$ was the maximum of the given sums, and both $x_i + x_l$ and $x_j + x_k$ are smaller than this. (2 points)

2.3. If x_{2n} is combined with some x_a, $a > 1$, then x_1 is combined with some x_b, $b < 2n$, Thus we are in the situation of 2.1 or 2.2, and the exchange yields the pair $x_1 + x_{2n}$, without increasing the largest sum. Now we can proceed similarly with the remaining sequence $x_2 < \ldots < x_{2n-1}$, and so on. (This last argument might be formalized as induction.) (6 points)

3.1. If job i from X and job j from Y are in conflict then $OPT(i,j) = \min\{OPT(i-1,j), OPT(i,j-1)\} + 1$. If they are not in conflict then $OPT(i,j) = \min\{OPT(i-1,j), OPT(i,j-1), OPT(i-1,j-1)\} + 1$. Initialization: $OPT(i,0) = i$, $OPT(0,j) = j$. (4 points)

3.2. The initial values are obvious: As long as only one chain is processed, the optimal length is the number of jobs. Now for the general situation: The last slot of a partial solution may contain only job i from X, or only job j from Y, or both. The schedule before this slot can be chosen optimally. This yields the three different terms on the right-hand side, and $+1$ accounts for the last slot. However, if the jobs are in conflict, then the option to schedule them both is not available. (6 points)

3.3. The time is $O(mn)$, as we compute mn values, each one in $O(1)$ time. (2 points)

4.1. We apply dynamic programming, closely following the algorithm for shortest paths when edge lengths are given. Let $OPT(j)$ be the cost of a cheapest path from v_1 tp v_j. Then $OPT(1) = c(v_1)$, and $OPT(j)$ is the minimum of all $OPT(i) + c(v_j)$, where the i are all indices $i < j$ such that a directed edge (v_i, v_j) exists. (5 points)

4.2. Since we have a topological order, any cheapest directed path to v_j consists of a cheapest directed path to some v_i, $i < j$, and another edge to v_j, whose cost is simply $c(v_j)$. Since we take the minimum of all the cost options, we obtain the minimum overall cost. (3 points)

4.3. Precisely as in the shortest path algorithm in DAGs, every edge is processed only once. therefore the time is $O(m)$, where m is the number of edges. (2 points)

5.1. Given a set of bundles, we need to check: every pair appears in some bundle, every pair appearing in a bundle is also in the given family, and at most c cells are used. All this can be done in polynomial time. (3 points)

5.2. Let $G = (V, E)$ be a graph with n nodes and m edges, and k a threshold for the size of a vertex cover. We create a symbol for every node and a pair for every edge, in the obvious sense. We set $c := k + m$. Clearly this costs only polynomial time. (4 points)

5.3. If G has a vertex cover with k nodes, then we can form k bundles that contain all pairs, with total length $k + m$. Conversely, consider a set of bundles with altogether $k + m$ cells. Let C be the set of all headers of the bundles. Since all pairs are represented, C is a vertex cover in the graph. Since we have exacty m pairs, the number of headers is $(k + m) - m = k$. (5 points)

5.4. If a problem belongs to NP, and some NP-complete problem can be reduced to it in polynomial time, then that problem is NP-complete itself. Since Vertex Cover is known to be NP-complete, and we have reduced it to Pair Compression, the assertion follows. (2 points)