Algorithms: CSE 202 — Homework III

Problem 1: Job scheduling (KT 7.41)

Suppose you're managing a collection of processors and must schedule a sequence of jobs over time.

The jobs have the following characteristics. Each job j has an arrival time a_j when it is first available for processing, a length ℓ_j which indicates how much processing time it needs, and a deadline d_j by which it must be finished. (We'll assume $0 < \ell_j \leq d_j - a_j$.) Each job can be run on any of the processors, but only on one at a time; it can also be preempted and resumed from where it left off (possibly after a delay) on another processor.

Moreover, the collection of processors is not entirely static either: You have an overall pool of k possible processors; but for each processor i, there is an interval of time $[t_i, t'_i]$ during which it is available; it is unavailable at all other times.

Given all this data about job requirements and processor availability, you'd like to decide whether the jobs can all be completed or not. Give a polynomial-time algorithm that either produces a schedule completing all jobs by their deadlines or reports (correctly) that no such schedule exists. You may assume that all the parameters associated with the problem are integers.

Example. Suppose we have two jobs J_1 and J_2. J_1 arrives at time 0, is due at time 4, and has length 3. J_2 arrives at time 1, is due at time 3, and has length 2. We also have two processors P_1 and P_2. P_1 is available between times 0 and 4; P_2 is available between times 2 and 3. In this case, there is a schedule that gets both jobs done.

- At time 0, we start job J_1 on processor P_1.
- At time 1, we preempt J_1 to start J_2 on P_1.
- At time 2, we resume J_1 on P_2. (J_2 continues processing on P_1.)
- At time 3, J_2 completes by its deadline. P_2 ceases to be available, so we move J_1 back to P_1 to finish its remaining one unit of processing there.
- At time 4, J_1 completes its processing on P_1.

Notice that there is no solution that does not involve preemption and moving of jobs.

Problem 2: Graph cohesiveness (KT 7.46)

In sociology, one often studies a graph G in which nodes represent people and edges represent those who are friends with each other. Let's assume for purposes of this question that friendship is symmetric, so we can consider an undirected graph.

Now suppose we want to study this graph G, looking for a "close-knit" group of people. One way to formalize this notion would be as follows. For a subset S of nodes, let $e(S)$ denote the number of edges in S-that is, the number of edges that have both ends in S. We define the *cohesiveness* of S as $e(S)/|S|$. A natural thing to search for would be a set S of people achieving the maximum cohesiveness.

1. Give a polynomial-time algorithm that takes a rational number α and determines whether there exists a set S with cohesiveness at least α.
2. Give a polynomial-time algorithm to find a set S of nodes with maximum cohesiveness.

Problem 3: Number puzzle

You are trying to solve the following puzzle. You are given the sums for each row and column of an $n \times n$ matrix of integers in the range $1 \ldots, M$, and wish to reconstruct a matrix that is consistent. In other words, your input is M, $r_1, \ldots, r_n$, $c_1, \ldots, c_n$. Your output should be a matrix $a_{i,j}$ of integers between 1 and M so that $\sum_i a_{i,j} = c_j$ for $1 \leq j \leq n$ and $\sum_j a_{i,j} = r_i$ for $1 \leq i \leq n$; if no such matrix exists, you should output, "Impossible". Give an efficient algorithm for this problem.

Problem 4: Database projections (KT 7.38)

You're working with a large database of employee records. For the purposes of this question, we'll picture the database as a two-dimensional table T with a set R of m rows and a set C of n columns; the rows correspond to individual employees, and the columns correspond to different attributes.

To take a simple example, we may have four columns labeled

`name, phone number, start date, manager's name`

and a table with five employees as shown here. Given a subset S of the columns, we can obtain a new, smaller

Table 1: Table with five employees.

name	phone number	start date	manager's name
Alanis	3-4563	6/13/95	Chelsea
Chelsea	3-2341	1/20/93	Lou
Elrond	3-2345	12/19/01	Chelsea
Hal	3-9000	1/12/97	Chelsea
Raj	3-3453	7/1/96	Chelsea

table by keeping only the entries that involve columns from S. We will call this new table the *projection* of T onto S, and denote it by $T[S]$. For example, if $S = \{$`name, start date`$\}$, then the projection $T[S]$ would be the table consisting of just the first and third columns.

There's a different operation on tables that is also useful, which is to *permute* the columns. Given a permutation p of the columns, we can obtain a new table of the same size as T by simply reordering the columns according to p. We will call this new table the *permutation* of T by p, and denote it by T_p.

All of this comes into play for your particular application, as follows. You have k different subsets of the columns $S_1, S_2, \ldots, S_k$ that you're going to be working with a lot, so you'd like to have them available in a readily accessible format. One choice would be to store the k projections $T[S_1], T[S_2], \ldots, T[S_k]$, but this would take up a lot of space. In considering alternatives to this, you learn that you may not need to explicitly project onto each subset, because the underlying database system can deal with a subset of the columns particularly efficiently if (in some order) the members of the subset constitute a *prefix* of the columns in left-to-right order. So, in our example, the subsets $\{$`name, phone number`$\}$ and $\{$`name, start date, phone number,`$\}$ constitute prefixes (they're the first two and first three columns from the left, respectively); and as such, they can be processed much more efficiently in this table than a subset such as $\{$`name, start date`$\}$, which does not constitute a prefix. (Again, note that a given subset S_i does not come with a specified order, and so we are interested in whether there is *some* order under which it forms a prefix of the columns.)

So here's the question: Given a parameter $\ell < k$, can you find ℓ permutations of the columns $p_1, p_2, \ldots, p_\ell$ so that for every one of the given subsets S_i (for $i = 1, 2, \ldots, k$), it's the case that the columns in S_i constitute a prefix of at least one of the permuted tables $T_{p_1}, T_{p_2}, \ldots, T_{p_\ell}$? We'll say that such a set of permutations constitutes a valid solution to the problem; if a valid solution exists, it means you only need to store the ℓ permuted tables rather than all k projections. Give a polynomial-time algorithm to solve this problem; for instances on which there is a valid solution, your algorithm should return an appropriate set of ℓ permutations.

Example. Suppose the table is as above, the given subsets are

$S_1 = \{$`name, phone number`$\}$,

$S_2 = \{$`name, start date`$\}$,

$S_3 = \{$`name, manager's name, start date`$\}$, and $\ell = 2$. Then there is a valid solution to the instance, and it could be achieved by the two permutations

$$p_1 = \{\texttt{name, phone number, start date, manager's name}\},$$
$$p_2 = \{\texttt{name, start date, manager's name, phone number}\}.$$

This way, S_1 constitutes a prefix of the permuted table T_{p_1}, and both S_2 and S_3 constitute prefixes of the permuted table T_{p_2}.

Problem 5: Maximum likelihood points of failure

A network is described as a directed graph (not necessarily acyclic), with a specified *source* s and *destination* t. A set of nodes (not including s or t) is a *failure point* if deleting those nodes disconnects t from s. For each node of the graph, i, a *failure probability* $0 \leq p_i \leq 1$ is given. It is assumed that nodes fail independently, so the failure probability for a set $F \subseteq V$ is $\prod_{i \in F} p_i$. Give an algorithm which, given G and $p_i fori \in V$, finds the failure point F with the maximum failure probability.