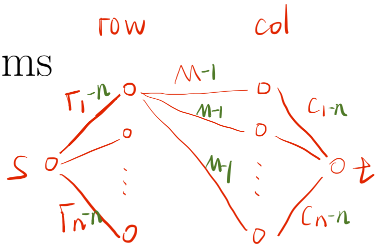
Algorithms: CSE 202 — Additonal Flow Problems



Problem 1: 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.

saturate r_i, c_i
& $\sum r_i = \sum c_i$

数独.

网格题
+ lower bound

Solution: Number puzzle

Idea: The idea is to view the problem as that of distributing the sum of each row among the n columns while respecting the sum of every column.

Since we know that each entry should be at least 1, we allocate 1 to each entry of the matrix and distribute each row sum $r_i - n$ to the columns so the column sums turn out to be $c_j - n$. Now the entries in the matrix will be in the range $\{0, 1, \ldots, M-1\}$. If any of the the $2n$ values r_i and c_i are less than n or $\sum_i (r_i - n) \neq \sum_j (c_j - n)$, then output "impossible". Observe that if r_i and c_i were indeed row and column sums respectively, then $\sum_i r_i = \sum_i c_i$ must be equal to the sum of the entries of the matrix.

Flow network: We will construct a flow network which contains a node for every row and column in such a way that the flow corresponds to the distribution of row sums.

Let the flow network $G = (V, E, c)$ where c is capacity function. V contains a node for each row and a node for each column in addition to the source node s and sink node t. More specifically, $V = \{s, t\} \cup R \cup C$ where $R = \{u_1, \ldots, u_n\}$ and $C = \{v_1, \ldots, v_n\}$. The nodes in R represent rows and the nodes in C represent columns. The source node s is connected by a directed edge to each node u_i with capacity $r_i - n$. Each node in C is connected to the sink node t by a directed edge with capacity $c_i - n$. Each node u_i is connected to each node v_j with a directed edge of capacity $M - 1$. Formally,

$$\begin{aligned} V &= \{s, t\} \cup R \cup C \\ E &= \{s\} \times R \cup \{(u_i, v_j) \mid 1 \leq i, j \leq n\} \cup C \times \{t\} \end{aligned}$$

and the edge capacities are

$$\begin{aligned} c(s, u_i) &= r_i - n \\ c(u_i, v_j) &= M - 1 \\ c(v_j, t) &= c_j - n \end{aligned}$$

where c is the capacity function on edges.

We now run the Preflow-Push algorithm on the flow network. If the maximum flow is not equal to $\sum_i (c_j - n)$, then output "impossible". Otherwise, output the matrix M with $M_{i,j} = f(u_i, v_j) + 1$ where f is the flow function on the edges of G.

Complexity: The flow network has $O(n)$ vertices and $O(n^2)$ edges and it can be constructed in $O(n^2)$ time. One can apply the Preflow-Push algorithm for determining the maximum flow on this network to get an overall time bound of $O(n^3)$.

Correctness: Let $\sum_i (r_i - n) = \sum_j (c_j - n) = x$. The following two claims will establish the correctness.

Claim 1. *If there is a matrix A satisfying the constraints, then the max-flow is equal to x.*

Proof. We prove the claim by constructing a flow for G with value x. Define the following flow function for G:

$$\begin{aligned} f(s,u_i) &= r_i - n && \text{for } 1 \le i \le n \\ f(v_j,t) &= c_j - n && \text{for } 1 \le j \le n \\ f(u_i,v_j) &= A[i,j] - 1 && \text{for } 1 \le i,j \le n \end{aligned}$$

It is easy to check that f is a valid flow and its value is x. □

Claim 2. *If the max-flow of G is equal to x, then the algorithm outputs a matrix A satisfying the constraints.*

Proof. Let f be a maximum flow in G with value x. Let $A[i,j] = f(u_i,v_j) + 1$ for $1 \le i,j \le n$. We will argue that A has the desired properties.

Since the value of f is x, it must be that $f(s,u_i) = r_i - n$, $f(t,v_j) = c_i - n$. Since the flow is conserved at u_i,

$$\begin{aligned} \sum_{j=1}^{n} A[i,j] &= \sum_{j=1}^{n} f(i,j) + 1 \\ &= r_i. \end{aligned}$$

Similarly, since the flow is conserved at v_j,

$$\begin{aligned} \sum_{i=1}^{n} A[i,j] &= \sum_{i=1}^{n} f(i,j) + 1 \\ &= c_j. \end{aligned}$$

Clearly, $1 \le A[i,j] \le M$ for $1 \le i,j \le n$. A satisfies all the constraints and the claim is proved. □

Problem 2: Rounding (KT 7.39)

You are consulting for an environmental statistics firm. They collect statistics and publish the collected data in a book. The statistics are about populations of different regions in the world and are recorded in multiples of one million. Examples of such statistics would look like the Table 1. We will assume here for simplicity that our data is such that all row and column sums are integers. The Census Rounding Problem is to round all data to integers without changing any row or column sum. Each fractional number can be rounded either up or down. For example, a good rounding for our table data would be as Table 2.

Table 1: Examples of census statistics.

Country	A	B	C	Total
grown-up men	11.998	9.083	2.919	24.000
grown-up women	12.983	10.872	3.145	27.000
children	1.019	2.045	0.936	4.000
Total	26.000	22.000	7.000	55.000

reduce lower bound to zero.

1. Consider first the special case when all data are between 0 and 1. So you have a matrix of fractional numbers between 0 and 1, and your problem is to round each fraction that is between 0 and 1 to either 0 or 1 without changing the row or column sums. Use a flow computation to check if the desired rounding is possible.

countries

Table 2: Rounding results for census statistics.

population categories

Country	A	B	C	Total
grown-up men	11.000	10.000	3.000	24.000
grown-up women	13.000	10.000	4.000	27.000
children	2.000	2.000	0.000	4.000
Total	26.000	22.000	7.000	55.000

2. Consider the Census Rounding Problem as defined above, where row and column sums are integers, and you want to round each fractional number α to either $\lfloor \alpha \rfloor$ or $\lceil \alpha \rceil$. Use a flow computation to check if the desired rounding is possible.

3. Prove that the rounding we are looking for in (a) and (b) always exists.

Problem 3: Rounding (KT 7.39)

1. Let P be the population categories and C be the countries, $m = |P|$, $n = |C|$. We model the problem as follows.

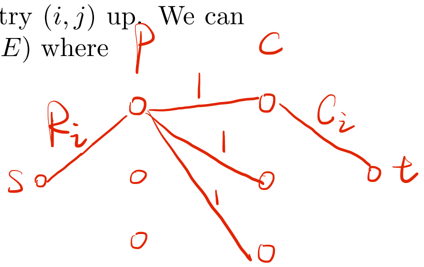
    - input: $R_i, C_j \in N$ s.t. $i \in P$, $j \in C$
    - output: $x_i, j \in \{0, 1\}$ s.t. $i \in P$, $j \in C$ or "not solvable" if the constraint cannot be met
    - constraint: $\forall i \in P \sum_{j \in C} x_{i,j} = R_i \wedge \forall j \in C \sum_{i \in P} x_{i,j} = C_j$

    行/列之和

    ↑

    网格题

    Here R_i is the ith row sum, C_j is the jth column sum, $x_{i,j}$ is 1 iff we round entry (i, j) up. We can solve this problem by reduction to network flow. Construct the digraph $G = (V, E)$ where

    $$\boxed{\begin{aligned} V &= \{s, t\} \cup P \cup C \\ E &= \{s\} \times P \cup P \times C \cup C \times \{t\} \end{aligned}}$$

    and with edge capacities

    $$c(s, i) = R_i \quad c(i, j) = 1 \quad c(j, t) = C_j.$$

    The problem is solvable iff the max flow is $\sum_{i \in P} R_i = \sum_{j \in C} C_j$. This is true despite the integer constraints on the variables since the Ford-Fulkerson method yields an integer flow given integer inputs. Given an integer max flow, optimal $x_{i,j}$ can be read off the flow from i to j. There is always a solution since the original fractional values in the matrix forms an optimal flow. (This is highly nontrivial. That the existence of a fractional solution implies the existence of an integer solution would be very hard to prove directly, i.e. without appealing to the flow integrality theorem.)

    Note that $|V| = m + n$, $|E| = O(mn)$. The Edmonds-Karp algorithm can solve this problem in time $O(|E|^2) = O(m^2n^2)$. To see this, note that each augmenting path found increases the flow by exactly 1 and costs $O(|E|)$ time to find. The max flow is $\leq |E|$ for this problem, and so $\leq |E|$ such augmenting paths can be found.

    $R_i \geq \sum_j \lfloor x_{ij} \rfloor$

2. We can reduce this problem to part (a) as follows: For each matrix entry $x_{i,j}$, subtract $\lfloor x_{i,j} \rfloor$ from R_i and C_j.

3. We proved the existence of a solution in part (a). The input gives a fractional solution which corresponds to a fractional flow of value $\sum_{i \in P} R_i$. Since there cannot be a larger flow, this flow must be a maxflow. Since all capacities are integer, the flow integrality theorem implies that there is also an integer flow with the same value $\sum_{i \in P} R_i = \sum_{j \in C} C_j$.

    Besides, we can also construct a graph $G = (V, E)$ in Figure 1, where

$V = \{s, t\} \cup P \cup cells \cup cells' \cup C$

cells are all the cell of the table, and cells' is a copy of cells.

$E = \{s\} \times P \cup P \times cells \cup cells' \times C \cup C \times \{t\}$

and with edge capacities

$$c(s, i) = R_i \quad c(i, cell(i, j)) = \infty \quad c(cell(i, j), cell(i, j)') = 1$$
$$c(cell(i, j)', j) = \infty \quad c(j, t) = C_j.$$

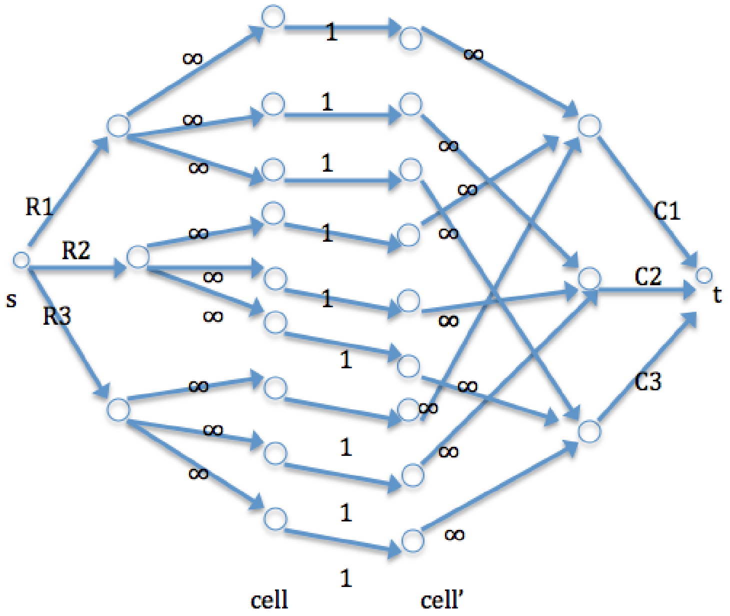


Figure 1: Flow network for proof in rounding problem

In that network flow, we can run the same algorithm in part(a) to get the rounding. But in Figure 1, for every cut, the capacity must greater or equal to $\sum Ri = \sum C_i$. Using the property that min-cut equals maximum-flow, combining $\sum Ri = \sum C_i$ is the min-cut, thus, there always exists a solution.

Problem 4: Driving Schedule

Some of your friends with jobs out West decide they really need some extra time each day to sit in front of their laptops, and the morning commute from Woodside to Palo Alto seems likely the only option. So they decided to carpool to work.

Unfortunately, they all hate to drive, so they want to make sure that any carpool management they agree upon is fair and does not overload any individual with too much driving. Some sort of simple round-robin scheme is out, because none of them goes to work every day, and so the subset of them in the car varies from day to day.

Here is one way to define *fairness*. Let the people be labeled $S = \{p_1, \ldots, p_k\}$. We say that the *total driving obligation* of p_j over a set of days is the expected number of times that p_j would have driven, had a driver been chosen uniformly at random from among the people going to work each day. More concretely, suppose the carpool plan lasts for d days, and on the i-th day a subset $S_i \subseteq S$ of the people go to work.

Then the above definition of the total driving obligation Δ_j for p_j can be written as $\Delta_j = \sum_{i:p_j \in S_i} \frac{1}{|S_i|}$. Ideally, we would like to require that p_j drives at most Δ_j times. However, Δ_j may not be an integer.

So let us say that a *driving schedule* is a choice of a driver for each day, that is, a sequence $p_{i_1}, \ldots, p_{i_d}$ with $p_{i_t} \in S_t$ and that a *fair driving schedule* is one in which each p_j is chosen as the driver on at most $\lceil \Delta_j \rceil$ days.

1. Prove that for any sequence of sets $S_1, \ldots, S_d$, there exists a fair driving schedule.

2. Give an algorithm to compute a fair driving schedule with running time in polynomial in k and d.

Design an efficient polynomial-time algorithm.
Provide a high-level description of your algorithm, prove its correctness, and analyze its time complexity.

Problem 5: Hospital scheduling

Consider the following scenario. Due to large-scale flooding in a region, paramedics have identified a set of n injured people distributed across the region who need to be rushed to hospitals. There are k hospitals in the region, and each of the n people needs to be brought to a hospital that is within 30 minutes driving time of their current location so different people will have different options for hospitals, depending on where they are right now.

At the same time, one doesn't want to overload any one of the hospitals by sending too many patients its way. The paramedics are in touch by cell phone, and they want to collectively work out whether they can choose a hospital for each of the injured people in such a way that the load on the hospitals is balanced: Each hospital receives at most $\lceil n/k \rceil$ people.

Give a polynomial-time algorithm that takes the given information about the people's locations and determine whether it is possible to balance the load among the hospitals. You are given the driving time for each of the n patients to each of the k hospitals.

Argue that your solution is correct.

In writing the solution, you may find it useful to following the following protocol.

1. Formally describe your problem.

2. Breifly describe how you plan to solve the problem using a couple of sentences.

3. Clealy any flow network you plan to employ.

4. Explan what you would do with the flow network and specify any maximum flow minimum cut algorithm you plan to deploy.

5. Explain how you would solve your problem using the flow network and the algorithm

6. Prove the correctness of your algorithm.

7. Determine the complexity of your algorithm.

Problem 6: Job Assignment

You are given a set of n jobs and a set of m machines. For each job you are given a list of machines capable of performing the job. An assignment specifies for each job one of the machines capable of performing it. The overhead for an assignment is the maximum number of jobs performed by the same machine. The job assignment problem is: given such lists and an integer $1 \leq k \leq n$, is there an assignment with overhead at most k?

Clearly describe the flow network and state the maximum flow-minimum cut algorithm you plan to use. Describe how you would solve your problem using the flow network. Prove the correctness of your algorithm and determine its complexity.

Problem 7: 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.

Solution: Job scheduling (KT 7.41)

We will set up a flow network where the processing units flow from the source to jobs, from the jobs to time intervals, from the time intervals to the sink.

Let J denote the set of jobs and let $n = |J|$. For $j \in J$, let a_j be its arrival time, d_j its deadline, and l_j its processing time. For job j, we use $M_j = [a_j, d_j]$ to denote the time interval that begins when the job arrives and ends at its deadline.

Let P denote the set of processors and let $k = |P|$. For $p \in P$, let $N_p = [t_p, t'_p]$ denote the time interval during which the processor is available.

Let $\mathbb{T} = \{a_j | j \in J\} \cup \{d_j |\ j \in J\} \cup \{t_p | p \in P\} \cup \{t'_p | p \in P\}$. Let $s_1, s_2, \ldots, s_{|\mathbb{T}|}$ be the sorted sequence of the elements in $\mathbb{T}$. Let I be the set of intervals of the form $I_h = [s_h, s_{h+1}]$ for $1 \leq h \leq |\mathbb{T}| - 1$.

For an interval $[t, t']$ where $t \leq t'$, we define its length $|[t, t']| = t' - t$. We say two intervals are disjoint if they do not share any points except the end points.

For any processor $p \in P$ and interval $I_h \in I$, either N_p includes I_h or it is disjoint from I_h. Let $P_h \subseteq P$ be the set of processors p such that N_p contains the interval I_h. In other words, P_h is the set of processors available during the time interval I_h.

Similarly, for any job $j \in J$ and interval I_h, either M_j includes I_h or it is disjoint from I_h.

Flow network: The flow network $\mathbb{G} = (\mathbb{V}, \mathbb{E}, c)$ is defined as follows.

1. $\mathbb{V} = \{s, t\} \cup J \cup I$.
2. For each $j \in J$, there is an edge (s, j) from s to job $j \in J$ with capacity l_j.
3. For each $j \in J$ and each interval $I_h \subseteq M_j$, there is an edge (j, h) with capacity $|I_h|$.

4. For each interval I_h, there is an edge (h, t) with capacity $|P_h||I_h|$ where P_h is the set of available processors during I_h.

We will now run Preflow-Push maximum-flow algorithm on $\mathbb{G}$ to compute the flow in time $O((|\mathbb{V}|)^3) = O((n+k)^3)$. Let f be the maximum flow obtained by running the algorithm. For an edge $(a, b) \in \mathbb{E}$, we use $f(a, b)$ to denote the flow along the edge (u, v) according to the flow function f. Let $L = \sum_{j \in J} l_j$. We show that the jobs can be completed if and only if the maximum flow is L.

For each interval I_h, let

$$J_h = \{(j, f(j, h))|\ j \in J \text{ and } f(j, h) > 0\}.$$

be the set of jobs that would need to be run during the interval I_h on some processor in P_h. The elements of the set J_h are called *job requests* for I_h. A job request (j, t) for I_h is a requirement that the job j needs to be run for time $t \leq |I_h|$ on one or more processors in P_h during the time interval I_h.

By a processor time slice, we mean objects of the type $(p, [\tau, \tau'])$ where $\tau \leq \tau'$ and the processor p is available during $[\tau, \tau']$. We say that two processor time slices are disjoint if the corresponding time intervals are disjoint. We say that an allocation A_h that assigns processor time slices $(p, [\tau_{jh}, \tau'_{jh}])$ to job requests $(j, f(j, h))$ in J_h is valid if the following conditions are satisfied.

- $p \in P_h$ and $[\tau_i, \tau'_i] \subseteq I_h$
- the processor time slices allocated to a job request must be pair wise disjoint.
- the total time of the processor time slices allocated to a job request $(j, f(j, h))$ must be $f(j, h)$.
- the total time required of a processor across all its time slices must be no more than $|I_h|$.

Theorem 0.1. *If the maximum flow is L, then there is a valid schedule to complete all the jobs.*

Proof. Since $l_j = \sum_h f(j, h)$ due to conservation of flow and since the intervals $\{I_h\}$ are disjoint, for each interval I_h, it is enough to show that there is a valid allocation of processor time slices of processors in P_h to job requests in J_h. The following claim provides an algorithm that produces a valid allocation. □

Claim 3. *Let $I = [0, T]$ be a time interval and P_I be a set of v processors which are available during I. Let $J_I = \{(j, T_j)|0 \leq j \leq u-1,\ T_j \leq T\}$ be a set of u job requests which need to be run during I on P_I. If $\sum_{i=0}^{u-1} T_i \leq vT$, there is a valid allocation of processor time slices to job requests.*

Proof. Assume that the processors are indexed by $\{0, 1, 2, \ldots, v-1\}$. Let $H_0 = 0$ and $H_j = \sum_{b=1}^{j-1} T_b$ for $1 \leq j < u$. Write each H_j uniquely as $H_j = n_j T + q_j$ where $n_j \geq 0$ is an integer and $0 \leq q_j < T$. We allocate the following processor time slices to the job request (j, T_j) for $0 \leq j \leq u-1$:

- If $q_j = 0$, allocate the time slice $[0, T_j]$ on the processor n_j.
- If $q_j > 0$, allocate the time slice $[q_j, \min(T_j, T - q_j)]$ on processor n_j and the time slice $[0, T_j - T + q_j]$ on processor $n_j + 1$ if $T_j - T + q_j > 0$.

Note that each job request gets at most two processor time slices such that the total time allocated to a job request (j, T_j) is exactly $T_j \leq T$. The allocation also ensures that the processor time slices allocated to a job request are disjoint. It is also clear that no processor is required to run more than T units of time.

We leave it to the reader to verify other details.

□

Theorem 0.2. *If there is a valid schedule to complete all the jobs, then the maximum flow of $\mathbb{G}$ is L.*

Proof. We will construct a flow f based on the schedule as follows.

- For each processor $j \in J$, the flow along the edge (s, j) is l_j.
- For each processor $j \in J$ and time interval I_h, the flow along the edge (j, h) is equal to the time the job j was run on some processor during the interval I_h.

- For each time interval I_h, the flow along the edge (h, t) is equal to $\sum_{p \in P_h} t_{ph}$ where t_{ph} is the time spent by the processor p during the interval I_h.

It is easy to verify that f is indeed a flow and its value is L. Students are expected to supply the missing details. □

Problem 8: 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 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 $S = \{\texttt{name, start date}\}$, then the projection $T[S]$ would be the table consisting of just the first and third columns.

Table 3: 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

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 $\{\texttt{name, phone number}\}$ and $\{\texttt{name, start date, phone number,}\}$ 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 $\{\texttt{name, start date}\}$, 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

$$\begin{aligned} S_1 &= \{\texttt{name, phone number}\}, \\ S_2 &= \{\texttt{name, start date}\}, \\ S_3 &= \{\texttt{name, manager's name, start date}\}, \end{aligned}$$

and $\ell = 2$. Then there is a valid solution to the instance, and it could be achieved by the two permutations

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

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}.

Solution: Database projections (KT 7.38)

Flow network: Let $S_1, \ldots, S_k$ be the given subsets of columns. We construct a bipartite graph $G = (U \cup V, E)$ where $U = \{u_1, \ldots u_k\}$ and $V = \{v_1, \ldots, v_k\}$. u_i and v_i represent the set S_i. E consists of exactly those edges (u_i, v_j) where $S_i \subset S_j$.

We construct a flow network $\mathbb{G} = (\mathbb{V}, \mathbb{E}, c)$ based on the bipartite graph.

- $\mathbb{V} = \{s, t\} \cup U \cup V$.
- $\mathbb{E}$ consists of the following edges with capacities as indicated:
  - (s, u_i) with capacity 1
  - (u_i, v_j) with capacity ∞
  - (v_j, t) with capacity 1

For the example given above, the graph is shown in Figure 2.

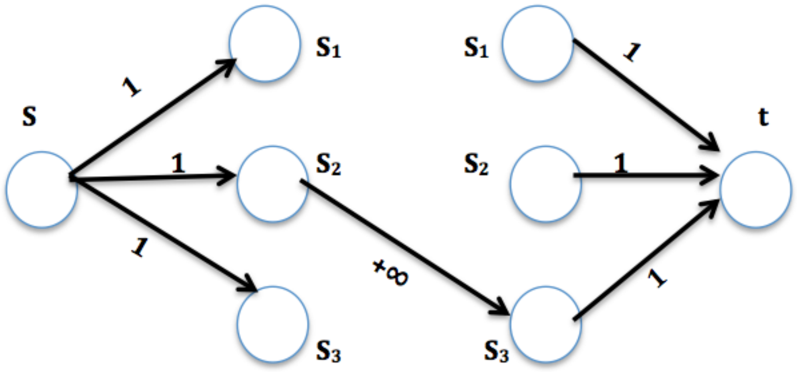


Figure 2: Flow network $\mathbb{G}$ for database projection

$S_1 \subset S_2 \subset S_3 \cdots$

不可能为圈，所以不可能为k

Output: Suppose the maximum flow of $\mathbb{G}$ is m. You will later see that $0 \leq m \leq k - 1$. We assert that we need at least $k - m$ permutations. Therefore, if $k - m \leq \ell$, there is a valid solution.

Claim 4. *If the maximum flow of $\mathbb{G}$ is m, we can cover all subsets with $k - m$ permutations.*

Proof. Given a flow f of value m, we want to construct $k - m$ permutations that together cover all subsets. We assume without loss of generality that our flow is integral. Since the capacities of the outgoing edges of s and the incoming edges of t are 1, we conclude that all the flow values are either 0 or 1.

We use the fact that if a family of subsets form a chain, then they can be covered by the same permutation, e.g., if $S_1 \subset S_2 \subset S_3$, then we can construct a permutation that starts with all elements in S_1 (in any order), followed by all elements in $S_2 - S_1$, followed by all elements in $S_3 - S_2$, followed by the rest. This permutation has prefixes for S_1, S_2, and S_3.

We now show that if the value of the flow is m, we can construct $k - m$ chains such that each set is in exactly one chain.

Consider the set P of all paths of the form $s \to u_i \to u_j \to t$ in $\mathbb{G}$ where the flow along each of the edges in the path is 1. Since the value of the flow is m, $|P| = m$. Let P' be the set of pairs of subsets of the form (S_i, S_j) such that there is a path in P with the edge (u_i, v_j). Since the flow leaving u_i is at most 1, we have at most one pair of the form $(S_i, X) \in P'$. Similarly, since the flow leaving v_j is at most 1, we have at most one pair of the form $(X, S_j) \in P'$. P' can be viewed as a graph H on $S_1, \ldots, S_k$ with exactly m edges where the indegree and outdegree of each node is at most 1. Furthermore H is acyclic. We can partition H into maximal chains where each chain is formed by starting with a node of indegree 0 and following its outgoing edge as long as one exists. In this partition, a chain covers i sets if and only if it employs $i-1$ edges. Since we have k vertices and m edges in the graph, the partition consists of exactly $k-m$ disjoint chains.

Therefore, with a flow of m, we can cover all subsets with $k-m$ permutations. □

Claim 5. *If the maximum flow of $\mathbb{G}$ is m, we cannot cover all subsets with fewer than $k-m$ permutations.*

Proof. We prove the claim using a proof by contradiction. Suppose there is a set of $k-d$ permutations with $d > m$ that covers all subsets. We then construct a flow of value d in $\mathbb{G}$, contradicting the fact that the maximum flow is m.

Suppose we can cover all subsets with $k-d$ permutations where $d > m$. By assigning each set to a permutation that covers it, we partition the sets into $k-d$ groups. If a set is covered by more than one permutation, choose one arbitrarily. Let σ be one of the permutations. The subsets covered by σ must form a chain. Let $S_1^\sigma \subseteq S_2^\sigma \subseteq \cdots \subseteq S_h^\sigma$ be the chain covered by σ for some $h \geq 1$. We call $S_i^\sigma \subseteq S_{i+1}^\sigma$ a segment. Since the $k-d$ permutations partition k sets into chains, there are exactly d segments in all chains together. We use these segments to define a flow of value d in $\mathbb{G}$ which leads to the desired contradiction.

Each segment corresponds to a unique augmenting path $s \to u_i \to v_j \to t$ in the flow network $\mathbb{G}$. So we can construct d augmenting paths in $\mathbb{G}$ which leads to a flow of value d. □

From the two claims, we conclude that we need at least $k-m$ permutations to cover all subsets where m is the maximum flow of $\mathbb{G}$ which implies that there is a valid solution if $\ell \geq k-m$.

Time complexity: We have k subsets. The construction of $\mathbb{G}$ takes $O(k^2)$. It takes $O(k^3)$ time to run the Preflow-Push algorithm on $\mathbb{G}$ which leads to a total time complexity of $O(k^3)$.

Problem 9: Spanning subgraph

出度

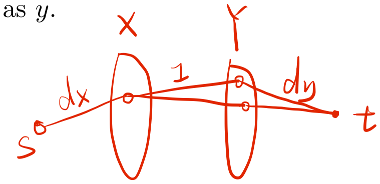
Given a bipartite graph $G = (V, E)$ and an integer d_v for each node v, does there exist a spanning subgraph H of G such that each node has degree d_v in H. Give an efficient algorithm to answer this question, and also necessary and sufficient conditions for the existence of such a subgraph. A spanning subgraph of $G = (V, E)$ is a subgraph whose vertex set is V and whose edge set is a subset of E.

Solution: Spanning subgraph

G is a bipartite graph, thus, we can divide V into two sets, X and Y such that every edge connects a vertice in X to one in Y. We denote the nodes in X as x and the node in Y as y.

Construct the graph $G' = (V', E')$ in Figure 3 where

$$V' = \{s, t\} \cup X \cup Y$$
$$E' = \{s\} \times X \cup E \cup Y \times \{t\}.$$



and with edge capacities

$$c(s,x) = d_x \quad c(x,y) = 1 \quad c(y,t) = d_y.$$

Two conditions must be satisfied in order to get such a spanning subgraph that each node has degree d_v.

1. In the spanning subgraph, sum of the degree of nodes in set X must equal to the sum of the degree of the nodes in set Y, s.t. $\sum_{x \in X} d_x = \sum_{y \in Y} d_y$. Otherwise, there is no such spanning subgraph H, that each node in H has degree d_v.

   This is because in bipartite graph, all the edges come from set X will end at set Y. Consider every edge, it will add one degree to each set X and Y, thus the sum of the degree of nodes in set X equal to the sum of the degree of the nodes in set Y.

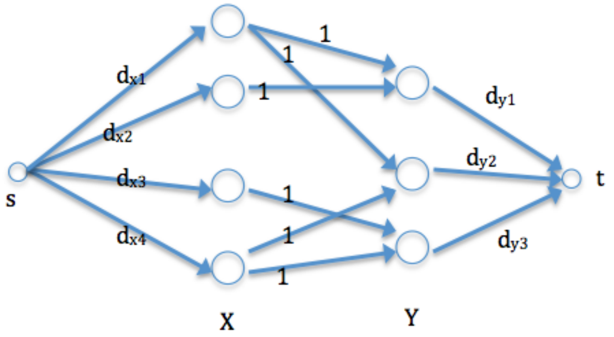


Figure 3: Flow network

2. The maximum-flow of the graph in Figure 3 need to be equal to $\sum_{x \in X} d_x$.

   To prove it is an necessary and sufficient condition for the existence of such a subgraph, we need to prove two aspects.

   - If the maximum-flow equals $\sum_{x \in X} d_x$, we can construct a spanning subgraph H, such that each node in H has degree d_v.

     Proof. For the first statement, because there are $\sum_{x \in X} d_x$ flow coming from source node s, and the maximum-flow equals $\sum_{x \in X} d_x$, thus each node has saturate flow d_x, and according to the conservation property of maximum-flow, d_x flow will output from node x. Plus the capacity of edges out of x is 1, thus, it has d_x edges out of x, which will be the edges in the spanning subgraph. For nodes in set Y, easy to prove it applying the same arguments. □

   →
   ←

   - If there is a spanning subgraph H, with each node having degree d_v, then we can have a maximum-flow of $\sum_{x \in X} d_x$.

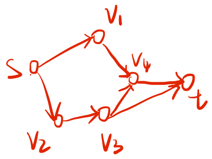
     Proof. First, we construct the same flow graph in Figure 3. We can prove the graph have maximum-flow $\sum_{x \in X} d_x$ using min-cut.
     For each cut $c(A, B)$, where $s \in A$ and $t \in B$, assume there are k nodes of X in the set A, thus, we have $|X| - k$ nodes from set X in the set B, where $0 \leq k \leq |X|$. For the x not in A, the cut will cross the edges between them and s, thus, the capacity of the cut of this part is $\sum_{x \notin A} d_x$. For those x in A, the flow will cross the cut or go to set Y. For those flow go to set Y will finally flow out and cross the cut. The capacity of the nodes not is A, we can have at least $\sum_{x \in A} d_x$ capacity. Thus, we can have the total capacity of the cut $\leq \sum x \in X d_x$. For the base case $k = 0$ or $k = |X|$, we can use the same idea to prove it. Besides, we have only $\sum_{x \in A} d_x$ flow out from s, which is the min-cut.
     Because the capacity of min-cut equals maximum-flow, thus, if there exists a spanning subgraph H, with degree of each node to be d_v, we can construct a maximum-flow $\sum_{x \in X} d_x$. □

Problem 10: 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 for $i \in V$, finds the failure point F with the maximum failure probability.

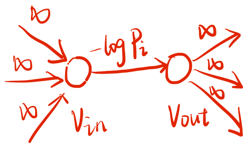
max $\sum_{i \in F} \log p_i$ → min $\sum_{i \in F} \log(-p_i)$

Solution: Maximum likelihood points of failure



Flow network: Let $G = (V, E)$ be the input graph. We construct a flow network $\mathbb{G} = (\mathbb{V}, \mathbb{E}, c)$. The vertex set $\mathbb{V} = \{s', t'\} \cup V_{\text{in}} \cup V_{\text{out}}$ consists of the following nodes:

- s' and t' where s' is the source node and t' is the sink node.
- $V_{\text{in}} = \{v_{\text{in}} |\ v \in V - \{s,t\}\}$ and $V_{\text{out}} = \{v_{\text{out}} |\ v \in V - \{s,t\}\}$



The edges in $\mathbb{E}$ are as follows:

- For every $(s, v) \in E$, we have a directed edge (s', v_{in}) with capacity ∞.
- For every $(v, t) \in E$, we have a directed edge (v_{out}, t') with capacity ∞.
- For every other $(u, v) \in E$, we have a directed edge $(u_{\text{out}}, v_{\text{in}})$ with capacity ∞.

只考虑 forward edges

- For every $v \in V$ other than s and t there is a directed edge $(v_{\text{in}}, v_{\text{out}})$ with capacity $-\log p_v$.

Output: Compute a minimum cut of $\mathbb{G}$ using Preflow-Push maximum flow algorithm. Let (A, B) be a minimum cut of $\mathbb{G}$ determined by the algorithm. Define $F = \{v |\ v_{\text{in}} \in A \text{ and } v_{\text{out}} \in B\}$. F is exactly the set of vertices v in G corresponding to the cut edges in $\mathbb{G}$ of the form $(v_{\text{in}}, v_{\text{out}})$. We output F as the failure point with maximum failure probability.

Correctness: We claim that there is a one-to-one correspondence between finite cuts in the flow network and points of failure in the input graph. From the claim, we conclude that a minimum cut would correspond to a failure point with maximum failure probability.

We first establish the following correspondence between G and $\mathbb{G}$.

Claim 6. *There is a one-one correspondence between s-t paths in G involving the vertex u and s'-t' paths in $\mathbb{G}$ involving the edge $(u_{\text{in}}, u_{\text{out}})$. More precisely, there is an s-t path in G which involves the vertex u if and only if there is an s'-t' path in $\mathbb{G}$ that involves the edge $(u_{\text{in}}, u_{\text{out}})$.*

Proof. Students are asked to provide the proof of the claim. □

Claim 7. *There is a cut in the flow network with capacity c if and only if there is a point of failure in the graph with probability 2^{-c}.*

Proof. Consider any cut with capacity c. Since the cut capacity is finite, all cut edges are of the form $(v_{\text{in}}, v_{\text{out}})$ for some $v \in V$. Let $F \subseteq V$ be the set of vertices v such that the corresponding edge $(v_{\text{in}}, v_{\text{out}})$ is a cut edge. We claim that F is a point of failure with probability 2^{-c}.

To argue that F is a point of failure, we notice that removing all cut edges of the form $(v_{\text{in}}, v_{\text{out}})$ in the flow network disconnects s' from t' (by the definition of a cut). However, by the construction of $\mathbb{G}$, it is easy to see that removing F from G would disconnect s from t in G.

The capacity c of the cut (A, B) is given by

$$\begin{aligned} c &= \sum_{v \in F} -\log p_v \\ &= -\log \left(\prod_{v \in F} p_v \right) \end{aligned}$$

Hence $\prod_{v \in F} p_v = 2^{-c}$.

For the other direction let $F \subseteq V$ be a point of failure with probability p. If we delete the vertices in F, G will not have any s-t paths. Let $F_{\text{in}} = \{v_{\text{in}} \in \mathbb{V} |\ v \in F\}$ and $F_{\text{out}} = \{v_{\text{out}} \in \mathbb{V} |\ v \in F\}$. We define

$$\begin{aligned} A &= \{s'\} \cup V_{\text{in}} \cup (V_{\text{out}} - F_{\text{out}}) \\ B &= \mathbb{V} - A \end{aligned}$$

We argue that (A, B) is a finite capacity cut for $\mathbb{G}$ with cut edges $\{(v_{\text{in}}, v_{\text{out}}) |\ v \in F\}$ (explain why).

The capacity of the cut is $\sum_{v \in F} -\log p_v = -\log \left(\prod_{v \in F} p_v \right) = -\log p$. □

Since the function 2^{-c} is monotonically decreasing with c we conclude that the maximum likelihood point of failure corresponds to the minimum cut.

Complexity analysis: Since we have $|\mathbb{V}| = O(|V|)$ we can find the minimum cut in time $O(|V|^3)$ using the Preflow-Push algorithm. Note that we have irrational capacities, hence Ford-Fulkerson is not be guaranteed to terminate. It is therefore important that we chose an algorithm with a runtime independent of the capacities.