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: Job scheduling (KT 7.41)

The basic idea is to reformulate the problem as a (multi-)assignment problem. We are assigning jobs to timesteps, such that the assignment satisfies the following conditions:

- A job can only be assigned to a timestep, if the time lies between the start time and the deadline of the job.
- The number of timesteps a job has to be assigned to must be equal to the length of the job.
- For any one timestep, the number of jobs assigned is not more than the number of available machines at that time.

More formally, let the jobs be $J = \{J_1, \dots, J_m\}$; the processors be $P = \{P_1, \dots, P_n\}$; J_i have arrival time a_i, deadline d_i, length l_i; P_j have arrival time t_j, ending time t'_j.

Let $T = \cup_{i \in [m]} [a_i, d_i)$, where the interval $[a_i, d_i)$ is to be interpreted as a subset of the integers. T is the set of times during which jobs may be scheduled. (We will see that the network flow problem we construct actually allows jobs to be scheduled at non-integer times, but that, due to the integrality theorem, there is always an optimal solution that only schedules during integer times.)

Construct the graph $G = (V, E)$ where

$$V = \{s, t\} \cup J \cup T$$
$$E = \{s\} \times J \cup \{(J_i, j) \mid j \in [a_i, d_i)\} \cup T \times \{t\}$$

and with edge capacities

$$c(s, J_i) = l_i \quad c(J_i, j) = 1 \quad c(j, t) = |P_k \mid j \in [t_k, t'_k)|$$

The capacity of edge (s, Ji) is the number of quanta of processing J_i needs. The capacity of the edge (j, t) is the number of processors available at time j. There is an edge with capacity 1 from P_i to time j iff P_i is available for scheduling at time j.

We have $|E| = O(\sum_i (d_i - a_i))$ and a max flow has value $\leq \sum_i l_i$, so a max flow can be found with the Edmonds-Karp algorithm in time $O(\sum_i l_i \sum_i (d_i - a_i))$.

We claim that there is a valid job schedule if and only if there is a flow with value $\sum_i l_i$. We first note that if there is such a flow, then it is a maximum flow, as the cut $\{s\}$ has the same value. Hence if there is such a flow there is also an integer flow achieving it (since all capacities are integer). The assignment of job quanta to times is then given by the flow on the (J_i, j) edges, and this assignment is valid since we assign every job to exactly least l_i timesteps, and the number of jobs assigned to any timestep is bounded by the number of available machines. Likewise, if we have a valid assignment, the flow that sends a flow of 1 through all edges where we assigned the job to a timestep is a maxflow that matches the cut $\{s\}$. Note that if a time j receives k jobs, the assignment of jobs to its $\geq k$ available processors can be arbitrary.

Problem 3: 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.

Solution: Graph cohesiveness (KT 7.46)

- Undirected graph $G = (V, E)$
- For any subset $S \subseteq V$, let $e(S)$ be the number of edges in E with both ends in S
- Define *cohesiveness* of $S = \frac{e(S)}{|S|}$

0.1 Determine whether graph cohesiveness is strictly larger than a rational number α

Let α be a rational number. We design an efficient algorithm to determine whether there exists a vertex set S with cohesiveness strictly *larger* [1] than α.

[1]We will show in Section 0.2 that cohesiveness of all subsets of G is a finite set of discrete rational numbers D. For an arbitrary rational number α, let β be the largest rational number in D such that $\beta < \alpha$. Then determining whether graph cohesiveness is *at least* α is equivalent to determining whether graph cohesiveness is strictly *larger* than β.

1. Construct a flow network $\mathcal{G}$ as shown in Figure 1.
2. We have the source node s and sink node t.
3. For each vertex $x \in V$, we include a node v_x in the flow network.
4. For each edge $(x, y) \in E$, we include a node $u_{x,y}$ in the flow network.
5. We have the following edges in $\mathcal{G}$:
    - Edges $(s \longrightarrow u_{x,y})$ with capacity 1
    - Edges $(u_{x,y} \longrightarrow v_x)$ and $(u_{x,y} \longrightarrow v_y)$ with capacity ∞
    - Edges $(v_x \longrightarrow t)$ with capacity α
6. Note that there are $|E|$ edges leaving the source s, so the capacity of min cut of $\mathcal{G}$ must be $\leq |E|$.
7. Since α is a rational number, we can scale edge capacities to integers. Time complexity to run the Preflow-Push Maximum-Flow algorithm on $\mathcal{G}$ is $O((|V| + |E|)^3)$.

Theorem 1. There exists a vertex set S with cohesiveness strictly larger than α if and only if max flow *cannot* saturate all edges leaving the source s.

Proof.

- Use max flow formulation, and consider min cut (A, B).
- Define S^* as the vertices on the source side of the min cut.
- Observe that $u_{x,y} \in A$ iff both $x \in S^*$ and $y \in S^*$:

  i. Infinite capacity edges ensure that if $u_{x,y} \in A$ then $x \in A$ and $y \in A$.

  ii. If $x \in A$ and $y \in A$ but $u_{x,y} \notin A$, then adding $u_{x,y}$ to A only decreases cut capacity.
- Capacity of cut (A, B): $cap(A, B) = \sum_{u_{x,y} \notin A} 1 + \sum_{v_x \in A} \alpha = (\sum_{(x,y)\in E} 1 - \sum_{u_{x,y}\in A} 1) + \sum_{x \in S^*} \alpha = |E| - e(S^*) + \alpha|S^*|$
- $\Leftarrow$: if max flow cannot saturate all edges leaving source s, then by the max-flow min-cut theorem, $cap(A, B) < |E|$, so $e(S^*) > \alpha|S^*|$, i.e. the set S^* has cohesiveness strictly larger than α.
- $\Rightarrow$: suppose there exists a set S with cohesiveness strictly larger than α, but max flow reaches $|E|$ units, i.e. the capacity of min cut is E. However, by forming a cut (A', B') according the S, we get its capacity $cap(A', B') = |E| - e(S) + \alpha|S| < |E| =$ min cut capacity, which is a contradiction of min cut definition.

$\square$

0.2 Find a vertex set S of G with maximum cohesiveness

Since $S \subseteq V$, there are $|V|$ choices of the size of subset S. Then there are $\binom{|S|}{2} = \frac{|S|(|S|-1)}{2}$ choices for $e(S)$. Therefore, we enumerate all possible rational numbers α in $O(|V|^3)$ to form a set D of cohesiveness values, and apply the algorithm in Section 0.1 for each rational number in D, from largest to smallest.

Problem 4: 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 $\sum_i a_{i,j} = r_j$ for $1 \leq j \leq n$ and $\sum_j a_{i,j} = c_i$ for $1 \leq i \leq n$; if no such matrix exists, you should output, "Impossible". Give an efficient algorithm for this problem.

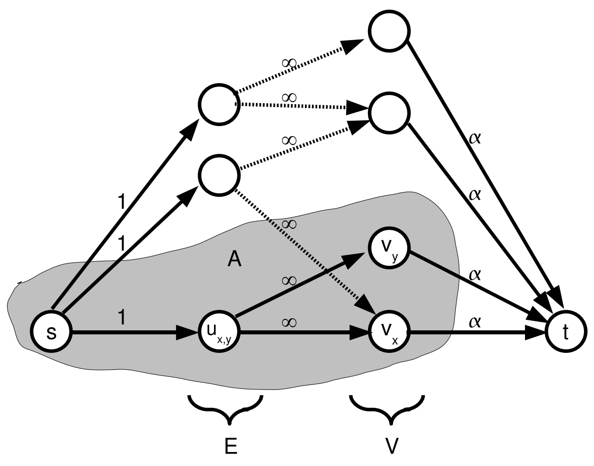

Figure 1: Flow network $\mathcal{G}$ to determine if graph cohesiveness is strictly larger than α

Solution: Number puzzle

Idea :
The main task is to find a way to distribute the sum of each row among the n columns while respecting the sum of every column. Here we know that each entry should atleast be 1, so for our future distribution we reduce all the $2n$ input values by n assuming that for each row we already have distributed value one to each column. Now the values we distribute will be in the range $\{0, 1, \ldots, M-1\}$ and not in $\{1, 2, \ldots, M\}$. As a pre-check if any of the the $2n$ values are less than n or $\sum_i (r_i - n) \neq \sum_j (c_j - n)$, then output "Impossible" (as both these values represent the total sum of all the entries of the matrix).

We will construct a network flow with nodes representing rows and columns. The flow through the edges going from a node corresponding to some row i to nodes corresponding to columns will give us the distribution of the value $r_i - n$ among the columns.

Algorithm :
Consider two sets R and C of size n each. Corresponding to each row i, R contains element R_i, and corresponding to each column j, C contains element C_j. Using these sets we will construct the graph $G = (V, E)$ where

$$V = \{s, t\} \cup R \cup C$$
$$E = \{s\} \times R \cup \{(R_i, C_j) \mid \forall i, j\} \cup C \times \{t\}$$

and the edge capacities are

$$c(s, R_i) = r_i - n \quad c(R_a, C_b) = M - 1 \quad c(C_j, t) = c_j - n$$

Now run any max-flow algorithm on this constructed network. If the max-flow is not equal to the cut $\{s\}$, i.e. $\sum_i (c_j - n)$, then output "Impossible". Else, output the matrix M with $M_{i,j} = f(R_i, C_j) + 1$.

Complexity :
One can construct the network from the given input in polynomial time. As the max-flow algorithm also runs in polynomial time, the total time taken by the algorithm is polynomial.

Correctness :
Let $\sum_i (r_i - n) = \sum_j (c_j - n) = x$. First we will argue that if there is a valid matrix A satisfying all the input constraints, then max-flow is equal to x. For this we just need to construct a flow of value x (since max-flow can't exceed x). To any edge from the sets $\{s\} \times R$ and $C \times \{t\}$ assign a flow equal to the edge capacity, and to any edge (R_i, C_j) give weight equal to $A_{i,j} - 1$ (which is $\leq M - 1$). It's easy to see that

flow through all the nodes is conserved. Thus, it's a valid flow and max-flow is equal to x.

Now we will argue that if max-flow $= x$, then our algorithm outputs a valid matrix. The flow through all the edges $\{s\} \times R$ and $C \times \{t\}$ is forced to be equal to the respective edge capacities. Using this fact we can prove that our output matrix will satisfy all the required constraints $\rightarrow$

- Each entry lies between 1 and M -
  $0 \leq f(R_i, C_j) \leq M - 1 \quad \rightarrow \quad 1 \leq A_{i,j} \leq M$

- Each row i sums up to r_i -
  $\sum_j A_{i,j} = \sum_j (f(R_i, C_j) + 1) = n + \sum_j f(R_i, C_j) = n + f(s, R_i) = n + (r_i - n) = r_i$

- Each column j sums up to c_j -
  $\sum_i A_{i,j} = \sum_i (f(R_i, C_j) + 1) = n + \sum_i f(R_i, C_j) = n + f(C_j, t) = n + (c_j - n) = c_j$

Problem 5: 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

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

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}.

Solution: Database projections (KT 7.38)

Algorithm description: Firstly we construct a bipartite graph of two node sets A and B as follows:

- For each subset S_i, we create a node in both sides of the graph.
- For each $S_i \subset S_j$ where $S_i \in A, S_j \in B$, we include an edge (S_i, S_j) in the graph.

Then we construct a flow network $\mathcal{G}$ based on the bipartite graph:

- We have the source node s and sink node t.
- Edges $(s \longrightarrow S_i)$ with capacity 1
- Edges $(S_i \longrightarrow S_j)$ with capacity $+\infty$
- Edges $(S_j \longrightarrow t)$ with capacity 1

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

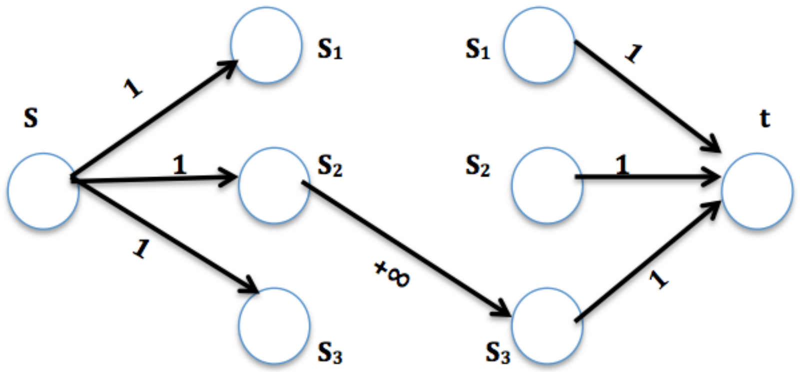

Figure 2: Flow network $\mathcal{G}$ for database projection

Suppose we have k subsets, the maximum flow of $\mathcal{G}$ is m, then we need at least $k - m$ permutations. Therefore, if $k - m \leq \ell$, there is a valid solution.

Correctness Proof: Let's prove our algorithm as follows:

1. *We can cover all subsets with $k - m$ permutations*

   Given a flow of size m, we want to construct a set of $k - m$ permutations that covers all subsets. We can assume without loss of generality that our maxflow is integral. For our graph this means in particular that for every edge, the flow in that edge is 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 given an integer flow for our graph of size m, we can construct $k - m$ chains such that each set is in exactly one chain.

   In our flow, if a path $s \to S_i \to S_j \to t$ in $\mathcal{G}$ contributes to the maximum flow, then $S_i \subset S_j$. Consider the set P of pairs (S_i, S_j) identified in that way. We have $|P| = m$ and furthermore the edges of capacity 1 in our graph ensure that for every S_i, there is at most one set S_j such that $(S_i, S_j) \in P$ and at most one set S_k such that $(S_k, S_i) \in P$.

   Using the set P we can easily construct exactly $k - m$ chains by picking the smallest set set S_i that is not covered yet, and following the unique paths $(S_i, S_{j_1}), (S_{j_1}, S_{j_2}), \ldots (S_{j_{k-1}}, S_{j_k})$, which gives us a chain of $S_i \subset S_{j_1} \subset \cdots \subset S_{j_k}$. Note that by starting with the smallest set (or one of the smallest), there is no pair (S_k, S_i) in P. Given m pairs this gives us exactly $k - m$ disjoint chains.

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

2. *We cannot cover all subsets with less than $k - m$ permutations*

   We want to do a proof by contradiction. Suppose there is a set of $k - c$ many permutations with $c > m$ that covers all subsets. We then construct a flow of value c, contradicting the fact that the maximum flow has value m.

   - Define a subset chain as $S_{a_1} \subset S_{a_2} ... \subset S_{a_n}$ where $1 \leq n \leq k$.
   - Define a segment as (S_i, S_j) if S_i and S_j are adjacent on the chain.

   We have the following assertions:

   - Each permutation corresponds to one subset chain(it could be a single subset).
   - Each subset appears only on one chain(if one subset appears on multiple chains, just keep one and drop others)
   - The number of chains is the number of subsets minus the number of segments on all chains

   Suppose we can cover all subsets with $k - c$ permutaitons where $c > m$, this means there are c segments on all chains. Let's show this is impossible.

   Each segment corresponds to a subset relation $S_i \subset S_j$ which corresponds to an augmenting path $s \to S_i \to S_j \to t$ in the graph $\mathcal{G}$. So we can find c augmenting paths using the c segments. Note that these paths won't intersect at node other than s and t since all subset appears on only one chain. In other words, these augmenting paths are compatible that all of them can contribute to the maximum flow. Therefore the maximum flow is at least c which is greater than m by assumption. This contradicts the fact that m is the maximum flow. Therefore, it's impossible to cover all subsets with $k - c$ permutaitons where $c > m$. We need at least $k - m$ permutations.

According to the proofs above, we can conclude that we need at least $k - m$ permutations to cover all subsets. Therefore, if $\ell \geq k - m$, there is a valid solution. Otherwise, we can not use ℓ permutations to cover all given subsets.

Time Complexity: We have k subsets. Constructing flow network takes $O(k^2)$ since we need to compare any two subsets to determine whether there is an edge. The complexity of running the Ford Fulkerson algorithm is $O(k^2 * k)$, the total time complexity is $O(k^3)$.

Problem 6: 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* $0 \le p_i \le 1$ 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 $p_i, i \in V$, finds the failure point F with the maximum failure probability.

Solution: Maximum likelihood points of failure

Let $G = (V, E)$ be the input graph. We construct a flow network $G' = (V', E', c)$. The vertex set V' consists of the following nodes:

- s, t as the source and sink of the flow network.
- For every $v \in V$, not including s and t, we have two vertices v_{in} and v_{out}

The edges 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 two directed edges $(u_{\text{out}}, v_{\text{in}})$ and $(v_{\text{out}}, u_{\text{in}})$ in E', both with capacity ∞.
- For every $v \in V$ other than s and t there is a directed edge $(v_{\text{in}}, v_{\text{out}})$ with capacity $-\log p_v$

We claim that there is a one to one correspondence of finite cuts in the flow network and points of failure in the input graph.

Claim 0.1. *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. For the first direction we construct a point of failure from a cut. 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 $A \subseteq V$ be the set of edges such that the corresponding edge is a cut edge. We claim that A is a point of failure with probability 2^{-c}.

To argue that A is a point of failure, we notice that removing all edges $(v_{\text{in}}, v_{\text{out}})$ in the flow network disconnects s from t (by the definition of a cut). Therefore removing this edge along with the vertices v_{in} and v_{out} also disconnects s from t. However, by the construction of G' this is equivalent to removing v from G, hence removing C in G disconnects s from t. For the cut capacity we have $c = \sum_{v \in A} -\log p_v = -\log\left(\prod_{v \in A} p_v\right)$. Hence $\prod_{v \in A} p_v = 2^{-c}$.

For the other direction let $A \subseteq V$ be a point of failure with probability p. By the construction of G' and the fact that A disconnects s from t there is a cut in the flow network such that only the edges $(v_{\text{in}}, v_{\text{out}})$ for every $v \in A$ is a cut edge. The capacity of the cut is $\sum_{v \in A} -\log p_v = -\log\left(\prod_{v \in A} p_v\right) = -\log p$. □

Since the function 2^{-c} is monotonely decreasing we can conclude that the maximum likelihood point of failure corresponds to the min cut.

Since we have $|V'| = O(|V|)$ and $|E'| = O(|V| + |E|)$ we can find the minimum cut in time $O(|V||E|^2 + |V|^3)$ using the preflow-push algorithm. Note that we have irrational capacities, hence Ford-Fulkerson would not be guaranteed to terminate. It is therefore important that we chose an algorithm with a runtime independent of the capacities.