Algoritmekonstruksjon TDT4125 - Assignment 3

Ola Kristoffer Hoff

12th April, 2023

Task 1

Inspired by tasks taken from McGill.

a For an arbitrary flow problem, there always exists a s-t cut with minimal capacity. Is it unique or can there be multiple s-t cuts with minimal capacity for a specific flow problem?

The s-t cut separates the flow graph into two sets, one with s and one with t, noted as A and B, respectively. Due to the conservation of flow constraint on the flow problem we can see that no matter where we cut, the cut will yield the same result, in terms of flow over the cut. Now to answer if there exists multiple such cuts with the same, and minimal, capacity, the answer is yes. We could easily have a graph with two identical bottle necks in sequence. As I have tried to illustrate in figure 1, where you can see that both the edge $d-e$ and $h-i$ has the same minimal cut.

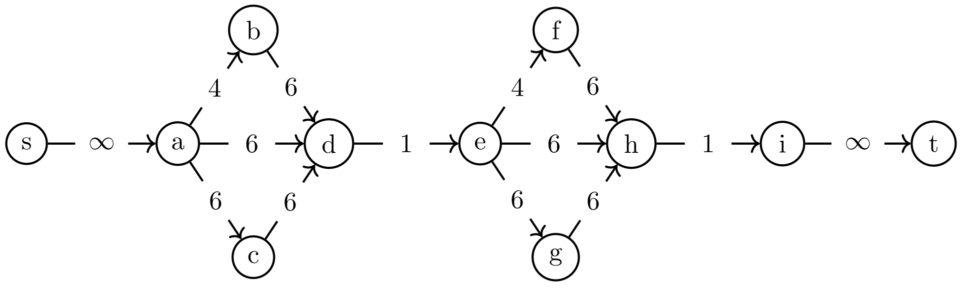


Figure 1: A network graph with two equal bottlenecks in sequence.

b Is there always only one unique way to maximize the flow in a flow problem?

The short answer is no. I will prove this with a simple example that show that there is multiple ways of maximising flow. As we see in figure 2, we can either take the flow up through b or down through c an we end up with two equal solutions of max flow for the same problem. Hence, there is not always only one unique way to maximise the flow in a flow problem.

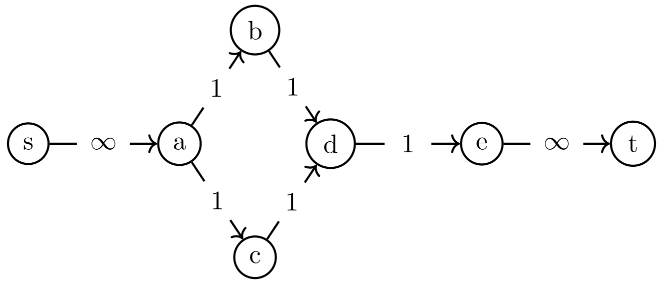


Figure 2: A network graph with two equal ways of maximising flow, the paths are in parallel.

Task 2

A software company will handle three projects over the next four years. The projects all have a start year and a deadline, which is respectively the earliest year in which they can start or must be completed by the end of. They also have a given size, measured in man-years. The company has 8 employees. Use flow to check if it is possible for the company to complete all the projects on time and if so use flow to create a plan for when they will work on each project.

Project	Start year	Deadline	Size
p_1	1	2	12
p_2	1	4	10
p_3	2	3	8

To solve this as a flow problem we could do the following: First we create a source s and a sink t. Then for each year we have a vertex, there is a capacity of 8 (employees) from source to each year-vertex. Then we have a vertex for each project, there is an edge of infinite capacity from the year-vertices the project is active in to the given project-vertex. Lastly, there is an edge from each project-vertex to the sink, t, with capacity of the size of the project. The problem is illustrated in figure 3. If the cut over t is equal to the sum of sizes of the projects, in other words all edges into the sink is fully saturated.

$$\sum_{i\in P} f(i) = \sum_{i\in P} c(i)$$

Hence, if the flow into t is maximum (not just maximal) it is possible to do the work within the given deadlines given the workforce.

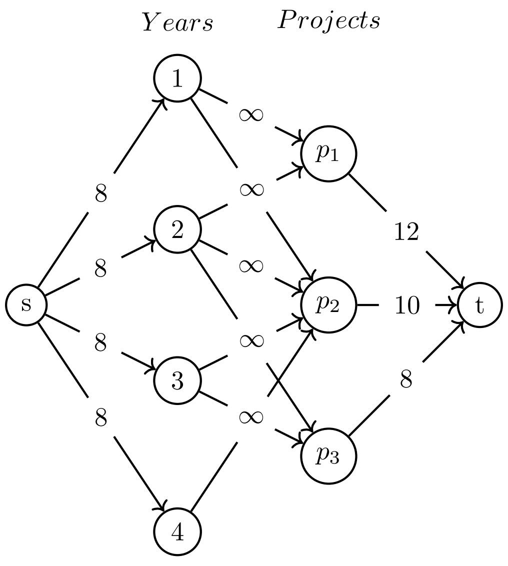


Figure 3: Flow problem for solving when to work on different projects to get everything done in time, or see if it's possible.

I will not show the steps of solving this since it's just a repeat of Ford-Fulkerson, which is prior knowledge to this course (and it takes quite some time to create the figures). However, in figure 4 we can see the flow of the solved problem. The solution below is to work 8 people on p_1 in year 1. In year 2, work 4 people on p_1 to finish it, and work 4 people on p_3. In year 3 work 4 people on p_3 to finish it, and work 4 people on p_2. In year 4 work 6 people on p_2 to finish it.

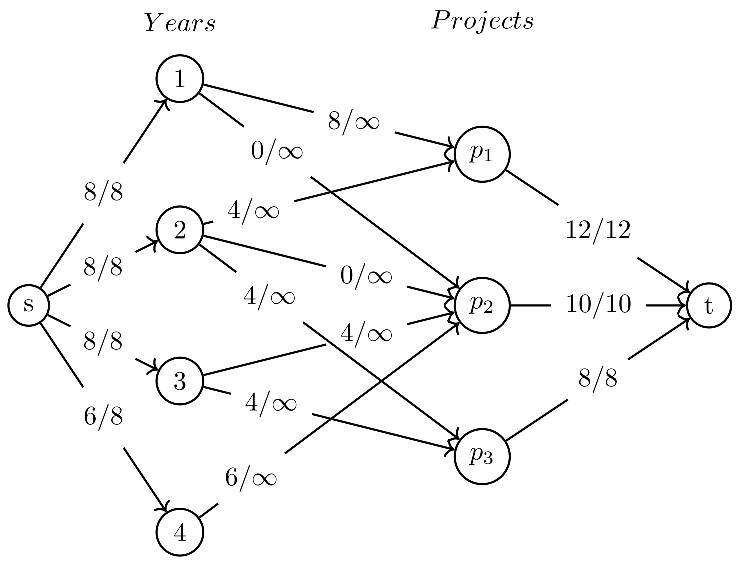


Figure 4: Flow problem for solving when to work on different projects to get everything done in time, or see if it's possible.

Task 3

Taken from the exam 2018S in TDT4120.

An orientation of an undirected graph $G = (V, E)$ is an assignment of direction to each edge $e \in E$, which results in a new directed graph $D = (V, A)$, where for every undirected edge $\{u, v\} \in E$ there exists either (u, v) or (v, u) in A. A k-orientation is an orientation where each vertex has at most $k(v)$ in-edges, where $k : V \to \mathbb{Z}^+$.

How can you efficiently find a k orientation using max-flow?

I understood the task as such; using max-flow, show how to find the k-orientation of D. Then answer how efficiently it is found. In other words take G and create D ensuring that the k-orientation constraint is withheld, this using max-flow.

If we do a similar trick as in the previous task. We have a source and sink, s and t, respectively. Then we have two sets with vertices, V_1 and V_2, both are equal to the set of vertices V. We connect, from the source, all vertices in V_1 with an edge of infinite capacity. Then we connect all vertices in V_2 to the sink with a capacity of k. Now we create an edge between all vertices in V_1 and V_2 that exist in E (in my illustration I have shown a complete simple graph). All these edges have a capacity of 1. An illustration graph is shown in figure 5.

If we now run maximum flow on this graph we will limit the indegree of all vertices to k because of the edges to the sink, t. However, we must assign a direction, so the sum of flow going from V_1 to V_2 must be equal to the number of edges, since each edge gives a flow of one. We must also here slightly modify the maximum flow algorithm, such that we don't orient an edge in both directions. This can be practically done by removing the opposite edge once the first is added to the solution, and if an edge is removed due to the residual graph, it's opposite edge must be added in again. More formally stated:

$$\sum_{e=\{u,v\}, u \in V_1 \land v \in V_2} f(e) = |E|, \qquad \forall e = \{u, v\} \text{ where } f(e) = 1 \Rightarrow \exists e' = \{v, u\} \text{ s.t. } f(e') = 0$$

If this condition is held, we have found a k-orientation for graph G and we can construct D by seeing which edges between V_1 and V_2 are equal to 1.

Note: I do not know if this is the most effective way of doing it. However, it's the way I figured it could be solved, using maximum flow at least. The running time should not increase too much with the extra bookkeeping of the removing and adding of opposite edges (described above). Those operations could be done in $O(1)$ time. Hence, the original complexity of the maximum flow is undisturbed. One might have to increase the storage of edges by a factor of 2, but that too is negligible in the asymptotic analysis.

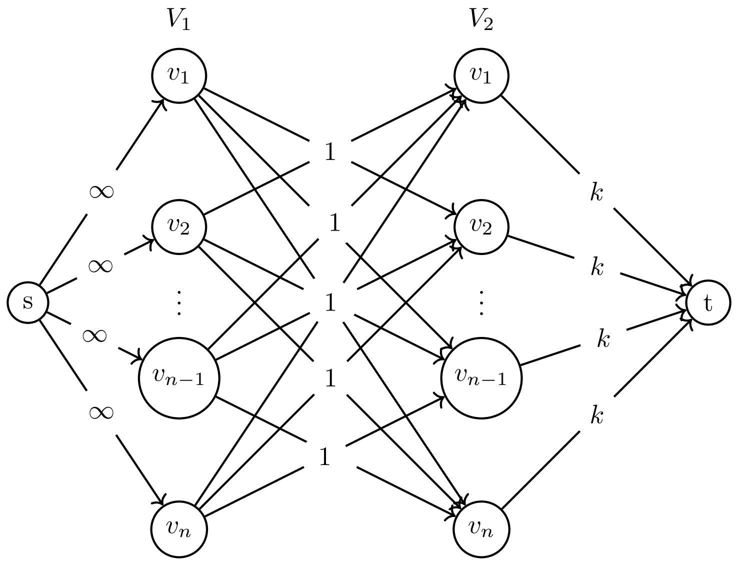
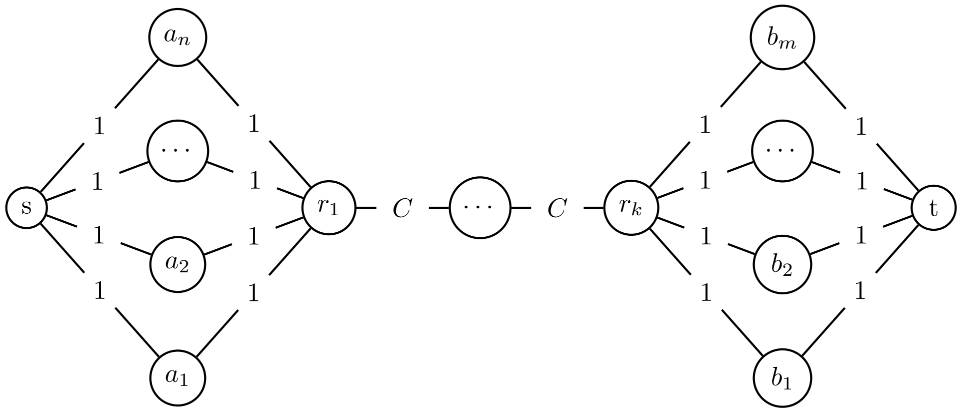


Figure 5: Illustration of a max flow representation of the k-orientation problem with an upper bound, k, on the indegree. $n = |V|$.

Task 4

In the compendium, we are shown that any augmenting path algorithm will be very slow on instances like the graph below. Use the *Push relabel* algorithm and show how it behaves. Let n be the height of the a tower of vertices on the left, and let m be the height of the b tower on the right and let the number r of vertices in the middle be a constant k. How many iterations does the *Push-relabel* algorithm need in the following cases:



Firstly I will give a short explanation of the *Push relabel* algorithm so that we are familiar with it when explaining it's running time in the task below.

I have written some pseudo code for the algorithm, see algorithm 1. First we initialise the graph; we set all flow to zero, then we set the flow from the source to all it's neighbours to their capacity on the edge connecting them; lastly the distance in s is sat to the number of vertices.

Then as long as there is an active vertex:

- We have $v \in G \backslash \{s\}$:
- $e(v) > 0$

- $d(v) \leq d(s)$
- There is no gap at a distance level less than $d(v)$. Meaning we can reach between all levels below.

We then check if this active vertex has any admissible edges to it's neighbours:

- Given active vertex v:
- $e = \{v, j\}, j \in \mathcal{N}(v)$
- $r_{vj} > 0$
- $d(v) = d(j) + 1$

If we find an admissible edge, we push as much of the excess, $e(v)$, as possible along the edge. Updating the residual and excess.

If no admissible edge is found then we relabel the vertex to a distance equal to 1 plus the minimum distance of the vertices in it's neighbourhood, that has a residual greater than zero. So we increase it to a point we possibly can push more flow later.

Then it continues until there are no more active points, then it's done. The runtime of this algorithm is $O(n^2 m)$.

Algorithm 1 *Push relabel.*

```
Ensure: $G = (V, E)$
  $f(e) \leftarrow 0, \forall e \in E$
  $f_{sj} \leftarrow c(\{s, j\})$
  $d(s) \leftarrow |V|$
  while $\exists v \in V$ s.t. $active(v)$ do
      if $\exists e = \{v, j\}, j \in \mathcal{N}(v)$ s.t. $admissible(e)$ then
          $\delta \leftarrow min(e(v), r_{vj})$
          $f_{vj} \leftarrow f_{vj} + \delta$
          $e(v) \leftarrow e(v) - \delta$
      else
          $d(v) \leftarrow min(d(i) + 1 : i \in \mathcal{N}(v) \wedge r_{vi} > 0)$
  return $G$
```

a $n = m = C$

Here we have enough capacity through r to push through all a-s and b-s. This means that we would have n cases of increasing the a-s to flow the excess into r_1 then k iterations propagating the flow down r to r_k. Then we would use m iterations to flow the flow into all b-s. Lastly, yet another m iterations to flow all b-s into t. Here I assume that we always choose a good active node and admissible edge. Since we could technically flow r_1 back into a and then a would have to increase even further. Then this would alternate until $d(a) = d(s)$ where it would no longer be regarded as active.

So optimistically we get: $Iterations = n+k+2m \Rightarrow O(n+m) \Rightarrow O(n)$. Since $n = m \Rightarrow n+m = n+n = 2n$.

b $n = C$ and $n \gg m$

Here we get the same start as above, and ending, only that we don't have enough b-s to use all the flow. So we have to double back and let the flow propagate back into the source again, hence doing the first step up until b twice.

$Iterations = 2(n + k) + 2m \Rightarrow O(n)$. We can remove the term m asymptotically since $n \gg m$.

c $n \ll m$ and $m = C$

Here we again get the same steps as earlier, however we now have to little flow to fill all b-s, but this is not a problem, and we don't have to double back as above. We only use n of the m b-s, hence we get:

$Iterations = n + k + 2n \Rightarrow O(n)$.

Here the a-s become a bottle neck for the flow and the rest of the flow moves undisturbed as in the first case, but not using all paths.

Task 5

This task is based on a previous exam task

You have n advisors advising you on a series of decisions. Before each decision, you must decide which of them you want to listen to. After the decision, you find out which of the advisers gave you good and bad advice. Describe a randomized algorithm that, in expectation, does not make many more mistakes than the expert who most often gives good advice.

For this problem we can use the *multiplicative weights algorithm* (see algorithm 2).

Here we would represent all adviser's in a column on length n. This for each decision we need to make T. Giving us a $T \times n$ matrix of choices. Imagine another matrix of the same dimensions where each recorded result of a potential choice is stored. The value we would have obtained if we had listened to the corresponding advisor, i, in the given time step, t. This denoted by $v_t(i) \in [0, 1]$.

We now imagine a third matrix like the other two. This one holds weights of probability. In the first time step all weights are equal to 1. In other words a uniform distribution of expected value from all advisers.

The algorithm is then very straightforward; we roll a "die" with as many faces as advisers, and with a probability of showing the advisers face equal to the weight divided my the sum of all weights, $p_t(i) = \frac{w_t(i)}{W_t}$, $W_t = \sum_{i=0}^{N} w_t(i)$. The choice in the first step is arbitrary. However, when we get the results after our choice we can update the weights into the next iteration by multiplying the previous weight with one plus the value obtained times a constant value: $(1 + \epsilon v_t(i)), \epsilon \in [0, \frac{1}{2}]$. The advisers which advice would have given a good result are increased more than the ones given bad results. (Note: weights does not decrease, but can remain the same). In the next iteration we then choose the next adviser and we do the whole thing over again. This time some advisers have a better chance of getting chosen, so if an adviser is actually better than the others it's probability will increase an the chances of being picked increase. This process repeats until all time steps are done. This is proven to yield a result close to as good as just choosing the same (best) adviser for the entire time span. [1] (page. 228, 7.3 Intermezzo: the Multiplicative Weights Algorithm).

Algorithm 2 *Multiplicative weights algorithm.* [1] (page. 229, Algorithm 7.1)

```
$w_1(i) \leftarrow 1 \quad \forall i = 1, 2, \dots, N$
for $t \leftarrow 1$ to $T$ do
    Make decision $i$ with probability proportional to $w_t(i)$, get value $v_t(i)$
    $w_{t+1}(i) \leftarrow w_t(i)(1 + \epsilon v_t(i)) \quad \forall i = 1, 2, \dots, N$
```

Task 6

Theorem 7.2 (page K229) compares the expected value of multiplicative weight updating and the value you would get if you made the same choice at all time steps. The result applies regardless of how the values, $v_t(i)$, are determined.

a An alternative is to compare with an optimal solution. Why is it not that useful to compare with an optimal solution in this situation?

I will here try to give an abstract but statistical explanation of the general case of why the optimal solution is not a good metric.

Here we are dealing with choices over time. When all choices are made we get clearly see which choices was the best and which adviser (to use the example of task 5) would was best. I will use a concept know as *regression to the mean* to explain the choices here. When all is well and done we can look at the results of a single adviser as a line of performance (as a percentage of how good they did), let's also assume that the accuracy of the individual does not increase over time, but stays constant. This we regard as the advisers talent, or expertise. Now the performance they show is a combination of their talent and luck: $result = talent + luck$. (luck here is also negative, as in unlucky). Imagine now a horizontal line, result on the y-axis, time on the x-axis. The line is the talent, a constant. Now imagine dots just above and below the line along it, these are the actual observed results. The "talent-line" could be found from linear regression. What the algorithm is essentially designed to do is identify that there is an underlying line of talent, and increase the probability of choosing the ones with greater talent. Then we compare our result with the adviser that turned out to be the best and we see we did nearly as good. If we were to use the optimal solution as a metric we would not look for the line, but for the individual points, the optimal chooses the best result, which is impossible to predict due to the fact it often is due to luck. We then compare our result to hitting the jackpot at each step, this is not a probabilistic metric. We cannot predict luck. Also notice that the algorithm looks for the best talent, so it makes sense to use the best talent as the metric.

b Would the proof work if, instead of choosing randomly based on the weight, we chose the one that currently has the highest weight (making the choice with the lowest i value when multiple choices have the same, highest weight)?

No, it would not, if I have understood the algorithm correctly it will give a comparable result to choosing one alternative and sticking with it. This is proved in the theorem (7.2). However, since there is no guarantee for an underlying distribution of correctness or value between the choices, and the results can be completely random. There exists a sequence of results which would lead the algorithm to pick the one result in step t that get zero value while any number of the others could max out. This leads to a possibility that the optimal sequence could be infinitely better than the algorithm, hence there is no performance/approximation guarantee. We could construct/rig such a sequence since we know have the algorithm will chose, now that we have removed the randomness properties of it, and it has become completely deterministic.

Note: to build further upon the argument in the previous subsection, about *regression to the mean*. Notice here that choosing the previous best would often yield poorer results. If say, the advisers were fairly even in talent, then if the best on had a bad day (unlucky) many others, having an OK or good day (lucky), might have better results. When a "bad" adviser has much luck, and thus had the best result in the previous iteration the *regression to the mean* clearly statistically state that the chance of replicating such a result is much less probable than performing worse than the previous iteration, and we can with good confidence expect the adviser's results in the current iteration to be worse. Thus, this might be a good algorithm for minimising results actually. Depending on the range of luck, and how even the talent is, but these are unknown factors until the results are in.

Task 7

Taken from "Network Flow Algorithms" (Williamson)

We will now look at a variant of multiplicative weight update, where instead of a choice having a value $v_t(i)$, it instead has a cost $c_t(i) \in [-1, 1]$. We want to make choices that minimize cost. Let $\epsilon \leq 1/2$ and change the update rule for the weighting in the algorithm (line 4 of Algorithm 7.1 on page K229) to be

$$w_{t+1}(i) = (1 - \epsilon c_t(i))w_t(i) \tag{1}$$

Show that after T steps the expected cost of the choices made in the algorithm will be such that for each decision, j, we have

$$\sum_{t=1}^{T}\sum_{i=1}^{N} c_t(i)p_t(i) \leq \sum_{t=1}^{T} c_t(j) + \epsilon\sum_{t=1}^{T}|c_t(j)| + \frac{lnN}{\epsilon} \tag{2}$$

Following the structure of the proof of the earlier case:

We start by finding the upper bound:

$$\begin{aligned}
W_{t+1} = \sum_{i=1}^{N} w_{t+1}(i) &= \sum_{i=1}^{N} w_t(i)(1 - \epsilon c_t(i)) \\
&= W_t - \epsilon W_t \sum_{i=1}^{N} c_t(i) p_t(i) \\
&= W_t(1 - \epsilon \sum_{i=1}^{N} c_t(i) p_t(i)) \\
&\leq W_t \frac{1}{exp(\epsilon \sum_{i=1}^{N} c_t(i) p_t(i))}
\end{aligned}$$

We used the relation: $1 - x \leq e^{-x}, \;\; x \in [-1, 1]$

$$\begin{aligned}
W_{T+1} &\leq W_T \frac{1}{exp(\epsilon \sum_{i=1}^{N} c_T(i) p_T(i))} \\
&\leq W_{T-1} \frac{1}{exp(\epsilon \sum_{i=1}^{N} c_{T-1}(i) p_{T-1}(i))} \frac{1}{exp(\epsilon \sum_{i=1}^{N} c_T(i) p_T(i))} \\
&\vdots \\
&\leq W_1 \prod_{t=1}^{T} \frac{1}{exp(\epsilon \sum_{i=1}^{N} c_t(i) p_t(i))} \\
&\leq N \frac{1}{exp(\epsilon \sum_{t=1}^{T} \sum_{i=1}^{N} c_t(i) p_t(i))}
\end{aligned}$$

Now we find the lower bound:

$$\begin{aligned}
W_{T+1} \geq w_{T+1}(j) &= \prod_{t=1}^{T} (1 - \epsilon c_t(j)) \\
&\geq (1 - \epsilon)^{\sum_{t=1}^{T} |c_t(j)|}
\end{aligned}$$

We used the relation: $1 - \epsilon x \geq (1 - \epsilon)^{|x|}, \;\; x \in [-1, 1]$

We now state that the lower bound must be less than or equal to the upper bound:

$$(1-\epsilon)^{\sum_{t=1}^{T}|c_t(j)|} \le N \frac{1}{exp(\epsilon \sum_{t=1}^{T}\sum_{i=1}^{N} c_t(i)p_t(i))}$$

$$(1-\epsilon)^{\sum_{t=1}^{T}|c_t(j)|} exp(\epsilon \sum_{t=1}^{T}\sum_{i=1}^{N} c_t(i)p_t(i)) \le N$$

$$ln(1-\epsilon)\sum_{t=1}^{T}|c_t(j)| + \epsilon\sum_{t=1}^{T}\sum_{i=1}^{N} c_t(i)p_t(i) \le lnN$$

$$\epsilon\sum_{t=1}^{T}\sum_{i=1}^{N} c_t(i)p_t(i) \le -ln(1-\epsilon)\sum_{t=1}^{T}|c_t(j)| + lnN$$

$$\epsilon\sum_{t=1}^{T}\sum_{i=1}^{N} c_t(i)p_t(i) \le -(-\epsilon-\epsilon^2)\sum_{t=1}^{T}|c_t(j)| + lnN$$

$$\epsilon\sum_{t=1}^{T}\sum_{i=1}^{N} c_t(i)p_t(i) \le \epsilon\sum_{t=1}^{T}|c_t(j)| + \epsilon^2\sum_{t=1}^{T}|c_t(j)| + lnN$$

$$\sum_{t=1}^{T}\sum_{i=1}^{N} c_t(i)p_t(i) \le \sum_{t=1}^{T}|c_t(j)| + \epsilon\sum_{t=1}^{T}|c_t(j)| + \frac{lnN}{\epsilon}$$

We used the relation: $ln(1-\epsilon) \ge -\epsilon - \epsilon^2, \;\; \epsilon \in [0, \frac{1}{2}]$

As we can see I did not get the exact formulation. I have an additional absolute value in the first expression on the right hand side. So this bound is not as tight as what was asked for, but I am convinced that I have looked at the maths too long an I am now blind, to the probably obvious, modification needed to get the exact answer. However, I hope this approximate solution is satisfactory to some extent.

References

[1] NTNU. *Kompendium, Algoritmekonstruksjon*. NTNU, 2023.