# Improved Capacity Upper Bounds for the Deletion Channel using a Parallelized Blahut-Arimoto Algorithm

Martim Pinto* João Ribeiro†

**Abstract**

We present an optimized implementation of the Blahut-Arimoto algorithm via GPU parallelization, which we use to obtain improved upper bounds on the capacity of the binary deletion channel. In particular, our results imply that the capacity of the binary deletion channel with deletion probability $d$ is at most $0.3578(1-d)$ for all $d \geq 0.64$.

*Departamento de Matemática, Instituto Superior Técnico, Universidade de Lisboa. martim.velasco.santos.pinto@tecnico.ulisboa.pt

†Instituto de Telecomunicações and Departamento de Matemática, Instituto Superior Técnico, Universidade de Lisboa. jribeiro@tecnico.ulisboa.pt

# Contents

# 1 Introduction

Synchronization errors, such as deletions and insertions, are a common occurrence in communication and data storage systems, most notably in emerging DNA-based data-storage technologies [YGM17, OAC+18, HMG19, WGGH23]. This motivates the study of channels with synchronization errors, also called *synchronization channels*. One of the simplest synchronization channels is the *binary deletion channel* (BDC), which on input a bitstring $x \in \{0,1\}^n$ independently deletes each bit of $x$ with some fixed deletion probability $d$. The corresponding output of this channel would then be the subsequence of $x$ consisting of its "undeleted" bits.

The BDC is closely related to the binary erasure channel (BEC), the only difference being that we do not replace the deleted bits by a "?". However, despite this similarity, our state of knowledge about these two channels is wildly different. We have known the capacity of the BEC since Shannon's early seminal work [Sha48] and the study of efficient coding for this channel has led to a rich mathematical theory. On the other hand, we still only know relatively loose bounds on the capacity of the BDC (let alone insights on its other properties), despite an extensive research effort on deriving both capacity lower bounds [Gal61, VD68, Zig69, DG06, MD06, DM07, DK07, Mit08, KD10, MTL12, RA13, VTR13, CK15, ISW16, RC23] and capacity upper bounds [DMP07, Mit08, FD10, Dal11, MTL12, RD15, Che19, CR19, RC23] for the BDC and related synchronization channels. The main reason behind this is that, although the behavior of the BDC is memoryless like the BEC, it causes a loss of synchronization between sender and receiver: when the receiver looks at the $i$-th output bit, they are not sure to which input bit it corresponds. For a more detailed discussion of the challenges imposed by this loss of synchronization, see the surveys [Mit09, MBT10, CR21, HS21].

## 1.1 State-of-the-art capacity upper bounds for the BDC and the underlying barriers

From here onward we denote the BDC with deletion probability $d$ by $\mathrm{BDC}_d$, and its capacity by $C(d)$. The best known upper bounds on $C(d)$ are obtained by numerically approximating the capacity of a "finite-length" version of the $\mathrm{BDC}_d$, an approach first studied by Fertonani and Duman [FD10]. More precisely, for any given integer $n \geq 1$ we may consider the discrete memoryless channel (DMC) with input alphabet $\{0,1\}^n$ and output alphabet $\{0,1\}^{\leq n} = \bigcup_{i=0}^{n} \{0,1\}^i$ which on input $x \in \{0,1\}^n$ behaves exactly like the $\mathrm{BDC}_d$ on input $x$. By the noisy channel coding theorem, the capacity of this channel, which we denote by $C_n(d)$, is given by

$$C_n(d) = \sup_{X^n} I(X^n; Y),$$

where the supremum is over all random variables $X^n$ supported on $\{0,1\}^n$, $Y$ is the corresponding output distribution of the $\mathrm{BDC}_d$ on input $X^n$, and $I(\cdot;\cdot)$ denotes mutual information. A simple argument (found, for example, in [FD10, Dal11]) combining the subadditivity of the sequence $\left(\frac{1}{n} C_n(d)\right)_{n \geq 1}$, Fekete's lemma, and the fact, established by Dobrushin [Dob67], that

$$\lim_{n \to \infty} \frac{1}{n} C_n(d) = C(d),$$

implies that

$$C(d) \leq \frac{1}{n} C_n(d) \tag{1}$$

for all $n \geq 1$. Therefore, we can upper bound $C(d)$ by upper bounding $C_n(d)$ for any $n \geq 1$. For example, by taking $n = 1$ we recover the easy upper bound $C(d) \leq 1 - d$, valid for all $d \in [0,1]$, and we can obtain better upper bounds by considering larger $n$.

A key observation is that $C_n(d)$ is the capacity of a DMC with finite input alphabet (of size $2^n$) and finite output alphabet (of size $2^{n+1}-1$). The well-known Blahut-Arimoto algorithm [Ari72, Bla72] can, in principle, numerically approximate the capacity of any DMC with finite input and output alphabets to any desired accuracy. By the connection above, arbitrarily good numerical approximations of $C_n(d)$ would lead to arbitrarily good approximations of $C(d)$, for any deletion probability $d$.[1]

The main issue with the approach in the previous paragraph is that the time and space complexity of the Blahut-Arimoto scales badly with the size of the input and output alphabets of the DMC under analysis. Therefore, a naive implementation of the Blahut-Arimoto algorithm will only produce results in a reasonable timeframe for small values of the input length $n$. For example, Fertonani and Duman [FD10] were able to run the BAA only up to $n = 17$. This motivates the following challenge:

> *Can we optimize the implementation of the Blahut-Arimoto with the deletion channel in mind so that we can obtain good bounds on $C_n(d)$ for significantly larger $n$?*

Recently, Rubinstein and Con [RC23] took on this challenge and presented an implementation of the Blahut-Arimoto algorithm with lower space complexity for this problem. They were able to apply this algorithm to input lengths up to $n = 28$, obtaining the current state-of-the-art upper bounds on $C(d)$.

### 1.2 Our contributions

We present an optimized implementation of the Blahut-Arimoto algorithm using GPU parallelization, and use it to obtain improved upper bounds on the capacity of the BDC for the entire range of the deletion probability $d$.

More precisely, we use our optimized implementation of the Blahut-Arimoto algorithm to compute upper bounds on $C_n(d)$ for input lengths up to $n = 31$.[2] The resulting improved bounds on $C(d)$ are reported in Table 3 for several values of $d$. Here, we expand on their consequences in the asymptotic high-noise setting where $d \to 1$, also studied in prior works [MD06, DMP07, FD10, Dal11, RD15, RC23]. Combining our improved bound for $d = 0.64$ with a result of Rahmati and Duman [RD15], we conclude that

$$C(d) \leq 0.3578(1-d)$$

for all $d \geq 0.64$. This improves on the previous best bound in the high-noise regime due to Rubinstein and Con [RC23], which was $C(d) \leq 0.3745(1-d)$ for all $d \geq 0.68$.

The implementation used to obtain the upper bounds is publicly available in a GitHub repository [Pin26].

[1] Fertonani and Duman [FD10] and later works, including ours, actually numerically approximate the capacity of the related *exact deletion channels* parameterized by $n \geq 1$ and $0 \leq k \leq n$ which receive an $n$-bit string $x$ and delete a uniformly random subset of $n-k$ bits of $x$, and then use these values to upper bound $C(d)$. This discussion applies equally well to those channels. For the sake of simplicity, we focus on a direct analysis of $C_n(d)$ here and leave a discussion of these proxy channels to Section 2.2.

[2] To be more precise, we computed good approximations of the exact deletion channel capacities $C_{n,k}$ (see Section 2.2) for all $k \leq n \leq 29$, and for $n = 31$ and all $k \leq 18$. In contrast, Rubinstein and Con [RC23] were able to compute good approximations of the $C_{n,k}$ values for all $n \leq 28$ and all $k$ satisfying $k + n \leq 39$. They were not able to approximate $C_{n,k}$ for some values of $k$ when $22 \leq n \leq 28$, due to the high space and time complexity of their implementation.

## 2 Preliminaries

### 2.1 Notation

We denote random variables and sets by uppercase Roman letters such as $X$, $Y$, and $Z$. Sets are sometimes also denoted by uppercase calligraphic letters such as $\mathcal{S}$ and $\mathcal{T}$. For an integer $n$, we define $\{0,1\}^{\leq n} = \bigcup_{i=0}^{n}\{0,1\}^i$. We index strings starting at 0, and for $y \in \{0,1\}^n$ we define $y_{[a:b]} = (y_a, y_{a+1}, \ldots, y_b)$. We write log for the base-2 logarithm.

### 2.2 Exact finite-length deletion channels

As already discussed in Section 1, our starting point is a finite-length version of the $\mathrm{BDC}_d$. More precisely, given an integer $n \geq 1$ this is the DMC that accepts inputs from $\{0,1\}^n$ and, given $x \in \{0,1\}^n$, sends $x$ through the $\mathrm{BDC}_d$ and returns its output $y \in \{0,1\}^{\leq n}$. We denote the capacity of this DMC by $C_n(d)$. As mentioned before, the following inequality holds.

**Lemma 1** ([FD10, Dal11])**.** *For every $d \in [0,1]$ and integer $n \geq 1$ we have*

$$C(d) \leq \frac{1}{n} C_n(d).$$

Since this finite-length channel is a DMC, its capacity $C_n(d)$ can be numerically approximated in principle using the Blahut-Arimoto algorithm, provided sufficient computational resources. A disadvantage of this approach is that we would need, at least at first sight, to restart the computation from scratch if we change the deletion probability $d$. With this in mind, it is also useful to consider another family of finite-length versions of the BDC, called *exact deletion channels*.

An *exact deletion channel* is parameterized by an input length $n \geq 1$ and another integer $0 \leq k \leq n$. We denote this channel by $\mathrm{BDC}_{n,k}$. On input $x \in \{0,1\}^n$, $\mathrm{BDC}_{n,k}$ outputs a length-$k$ subsequence $y$ of $x$ uniformly at random over all such $\binom{n}{k}$ subsequences. In other words, $\mathrm{BDC}_{n,k}$ chooses a uniformly random subset of $n-k$ coordinates of $x$ and deletes those bits. This channel is a DMC with input alphabet $\mathcal{X} = \{0,1\}^n$, output alphabet $\mathcal{Y} = \{0,1\}^k$, and channel rule $P_{n,k}$ satisfying

$$P_{n,k}(y|x) = \frac{\#\text{times } y \text{ appears as a subsequence of } x}{\binom{n}{k}}.$$

Its capacity, denoted $C_{n,k}$, is thus given by

$$C_{n,k} = \sup_{X^n} I(X^n; Y),$$

where the supremum is over all distributions $X^n$ supported on $\{0,1\}^n$ and $Y$ is the corresponding channel output on input $X^n$. The values of $C_{n,k}$ for $1 \leq k \leq n$ can be used to bound $C_n(d)$ by taking an appropriate convex combination.

**Lemma 2** ([FD10])**.** *For every $d \in [0,1]$ and all integers $n \geq 1$ and $1 \leq k \leq n$ we have*

$$C_n(d) \leq \sum_{k=1}^{n} \binom{n}{k} d^{n-k}(1-d)^k C_{n,k}.$$

An advantage of this approach is that once we upper bound $C_{n,k}$ for all $1 \leq k \leq n$, we can then easily get upper bounds on $C_n(d)$ for any value of $d$. We will follow this approach in this work.

Rubinstein and Con [RC23] used their optimized implementation of the Blahut-Arimoto algorithm to approximate $C_{n,k}$ for $k + n \leq 39$ with $n \leq 28$. However, they were not able to compute

$C_{n,k}$ for some values of $k$ when $22 \leq n \leq 28$, due to the high space and time complexity. In the following sections, we will see how to further decrease the time and space complexity of the algorithm specifically for deletion channels.

### 2.3 The Blahut-Arimoto algorithm

The Blahut-Arimoto algorithm (BAA) is a well-known tool for numerically approximating the capacity of finite-input/finite-output DMCs [Ari72, Bla72]. We present the standard formulation of the BAA here, and later show how it can be optimized for the BDC.

A finite-input/finite-output DMC is characterized by a finite input alphabet $\mathcal{X}$, a finite output alphabet $\mathcal{Y}$, and the channel law $P$. For each $x \in \mathcal{X}$ and $y \in \mathcal{Y}$ the probability that the channel outputs $y$ on input $x$ is denoted by $P(y|x)$. The BAA is an iterative procedure for approximating the capacity of any such DMC. After a prescribed number of iterations, it returns an input distribution $X$. The corresponding achievable rate $I(X;Y)$ (here $Y$ is the channel output distribution given input $X$) of the input distribution $X$ returned by the BAA is guaranteed to converge to the capacity of the channel as the number of iterations increases. In fact, we have more precise knowledge about the rate of convergence, as stated in the following result.

**Theorem 1** ([Ari72])**.** *Fix an finite-input/finite-output DMC, and let $C$ be its capacity. For any threshold $a > 0$, the BAA with $O(1/a)$ iterations returns an input distribution $X$ whose information rate $I(X;Y)$ satisfies*

$$C - a \leq I(X;Y) \leq C.$$

*Here, the $O(\cdot)$ notation hides a multiplicative constant that depends only on the choice of the DMC. Furthermore, the convergence to $C$ is monotonic.*

We now describe the BAA with a conservative stopping criterion. For a more detailed discussion, see [Yeu08, Chapter 9]. The starting point for the BAA is an initial input distribution $X^{(0)}$. This distribution may be chosen arbitrarily subject to having full support over $\mathcal{X}$. A common choice is to take $X^{(0)}$ to be the uniform distribution on $\mathcal{X}$. Then, for $t \geq 0$, the algorithm proceeds as follows on the $t$-th iteration given $X^{(t)}$:

1. **Compute channel output distribution of $X^{(t)}$:** This is the distribution $Y^{(t)}$ satisfying

$$Y^{(t)}(y) = \sum_{x \in \mathcal{X}} X^{(t)}(x) P(y|x)$$

   for all $y \in \mathcal{Y}$.

2. **Compute refined input distribution:** This is divided into two steps.

   (a) We compute the "unnormalized distribution" $W^{(t)}$ given by

$$W^{(t)}(x) = \prod_{y \in \mathcal{Y}} \left( \frac{X^{(t)}(x) P(y|x)}{Y^{(t)}(y)} \right)^{P(y|x)}$$

   for all $x \in \mathcal{X}$.

   (b) We normalize $W^{(t)}$ to get the new input distribution $X^{(t+1)}$. That is, we compute

$$X^{(t+1)}(x) = \frac{W^{(t)}(x)}{\sum_{x' \in \mathcal{X}} W^{(t)}(x')}$$

   for all $x \in \mathcal{X}$.

3. **Stopping criterion:** Suppose that we wish to output an input distribution $X$ such that its information rate $I(X;Y)$ satisfies $I(X;Y) \geq C - a$, with $C$ the capacity of the DMC and $a > 0$ some approximation threshold. Then (e.g., see [Ari72, Equation (32)]), it suffices to check whether

$$\max_{x \in \mathcal{X}} \log \left( \frac{X^{(t+1)}(x)}{X^{(t)}(x)} \right) < a.$$

If this condition is satisfied, we stop and return $X = X^{(t+1)}$. Otherwise, we move to the next iteration of the BAA.

### 2.3.1 The optimizations of Rubinstein and Con

A naive implementation of the BAA requires (i) storing the whole $|\mathcal{X}| \times |\mathcal{Y}|$ channel transition matrix (that for each $x \in \mathcal{X}$ and $y \in \mathcal{Y}$ stores $P(y|x)$), and (ii) storing all the values $W^{(t)}(x)$ for $x \in \mathcal{X}$ in Item 2 of the BAA. We wish to apply the BAA to numerically approximate the capacities $C_{n,k}$, corresponding to a DMC with input alphabet size $|\mathcal{X}| = 2^n$ and output alphabet size $|\mathcal{Y}| = 2^k$. Therefore, the memory costs of a naive implementation of the BAA quickly become prohibitive as the input length $n$ increases.

Rubinstein and Con [RC23] developed a more efficient implementation of the BAA for computing the $C_{n,k}$ values through time-memory tradeoffs and by leveraging sparsity of the relevant matrices when $k$ is close to $n$. Since their methods are also relevant to our optimized implementation of the BAA, we describe them in more detail. To handle the transition matrix $P$, there are two extremes we can consider:

- Pre-compute and store the whole transition matrix (requiring time and space $\Omega(2^{n+k})$). The advantage of this method is that, once this is done, we can retrieve the $P(y|x)$ values in time $O(1)$;
- Every time we require $P(y|x)$, compute it from scratch. Each such computation can be done in time $\Theta(nk)$ through dynamic programming.

Both extremes (high storage/low retrieval costs vs. low storage/high retrieval costs) turn out to be too costly even for small values of $n$, due to memory or time constraints. Rubinstein and Con adopted an approach between the two extremes. They devise a "loop-nest" optimized implementation of the BAA, consisting in cleverly changing the order of operations in the BAA, and combine it with the pre-computation of a smaller table (cache) that can then be used to compute the transition probabilities $P(y|x)$ much faster than the baseline $\Theta(nk)$ time procedure. To further speed up the application of the BAA, they leverage sparsity when $k$ is close to $n$.

For completeness, Algorithm 1 describes the loop-nest optimized implementation of the BAA from [RC23] verbatim with adapted notation. The method used in [RC23] to balance storage and time requirements for computing the transition probabilities is described verbatim in Algorithm 2. We give a more detailed description of how this approach works. In order to compute the transition probabilities $P_{n,k}(y|x)$ for the $\mathrm{BDC}_{n,k}$, the algorithm leverages only pre-computed tables containing the transition probabilities for the "smaller" exact deletion channels $\mathrm{BDC}_{n',k'}$ with $n' \approx n/2$ and $k' \leq k$. Then, to compute $P_{n,k}(y|x)$ for some input string $x \in \{0,1\}^n$ and output string $y \in \{0,1\}^k$, the algorithm partitions $x$ into two halves, $x_1$ and $x_2$, of lengths $n_1 = \lfloor n/2 \rfloor$ and $n_2 = \lceil n/2 \rceil$, respectively. Then, it computes $P_{n,k}(y|x)$ based on the pre-computed table by decomposing it as

$$P_{n,k}(y|x) = \sum_{k'=0}^{k} P_{n_1,k'}(y_{[0:k'-1]}|x_1) \cdot P_{n_2,k-k'}(y_{[k':k-1]}|x_2),$$

where $y_{[a:b]} = (y_a, y_{a+1}, \ldots, y_b)$ and we recall that in this work we index strings starting at 0, and so write $y = (y_0, y_1, \ldots, y_{k-1})$.

This recursive formulation reduces the per-query time complexity to $O(k)$, compared with the naive $O(nk)$ computation, while requiring $O(2^{\frac{n}{2}+k})$ space to store the pre-computed tables. In practice, this trade-off provides a favorable balance between time and space.

**Algorithm 1:** Loop nest optimized BAA [RC23, Algorithm 4 with adapted notation]

**Input:** Input/output alphabets $\mathcal{X}, \mathcal{Y}$; channel law $P$; convergence threshold $a > 0$

**Output:** Input distribution $X$ with $I(X;Y) = R$

$t \leftarrow 0$;

Choose $X^{(0)}$ to be the uniform distribution on $\mathcal{X}$;

**repeat**

`// Compute output distribution`

**foreach** $y \in \mathcal{Y}$ **do**

$$Y^{(t)}(y) \leftarrow \sum_{x \in \mathcal{X}} X^{(t)}(x) P(y|x);$$

`// Compute auxiliary function`

**foreach** $x \in \mathcal{X}$ **do**

$$W^{(t)}(x) \leftarrow \prod_{y \in \mathcal{Y}} \left( \frac{X^{(t)}(x) P(y|x)}{Y^{(t)}(y)} \right)^{P(y|x)};$$

`// Update input distribution`

**foreach** $x \in \mathcal{X}$ **do**

$$X^{(t+1)}(x) \leftarrow \frac{W^{(t)}(x)}{\sum_{x' \in \mathcal{X}} W^{(t)}(x')};$$

$t \leftarrow t + 1$;

**until** $\max_{x \in \mathcal{X}} \left| \log \left( \frac{X^{(t)}(x)}{X^{(t-1)}(x)} \right) \right| < a$;

`// Compute information rate`

$R \leftarrow I(X^{(t)}; Y^{(t)})$;

**return** $X^{(t)}, R$;

## 3 An overview of our optimizations

We provide an optimized implementation of the BAA by leveraging the observation that various steps in an iteration of the BAA lend themselves easily to parallelization. We set up parallelized versions of these steps that fit nicely into *CUDA kernels*.[3] Briefly, a CUDA kernel is divided into a grid of blocks, *with each block executing the same function in parallel.* Within each block, up to 1024 threads execute concurrently, but this number can also be any power of 2 below that.

For example, with Algorithm 1 in mind, consider the task of computing the auxiliary function

[3]For an introduction to CUDA, see the CUDA C++ programming guide here.

**Algorithm 2:** Cache-based computation of transition probabilities [RC23, Algorithm 3 with adapted notation]

**Input:** Input string $x \in \mathcal{X} = \{0,1\}^n$; output string $y \in \mathcal{Y} = \{0,1\}^k$; cache table of transition probabilities $P_{n',k'}(y'|x')$ for all $n' \leq \lceil n/2 \rceil$ and $k' \leq k$

**Output:** Transition probability $P_{n,k}(y|x)$

$n_1 \leftarrow \lceil \frac{n}{2} \rceil$;

$n_2 \leftarrow \lfloor \frac{n}{2} \rfloor$;

$x_1 \leftarrow x_{[0:n_1-1]}$ $x_2 \leftarrow x_{[n_1:n-1]}$ ;

$P_{n,k}(y|x) \leftarrow \sum_{k'=0}^{k} P_{n_1,k'}(y_{[0:k'-1]}|x_1) \cdot P_{n_2,k-k'}(y_{[k':k-1]}|x_2)$;

**return** $P_{n,k}(y|x)$;

$W^{(t)}(x)$ for all $x \in \mathcal{X}$, which we may write equivalently as

$$\log W^{(t)}(x) = \sum_{y \in \mathcal{Y}} P(y|x) \log \left( \frac{X^{(t)}(x) P(y|x)}{Y^{(t)}(y)} \right)$$

for increased numerical stability. We now discuss a way of parallelizing this computation using CUDA kernels, specialized for the exact deletion channel $\mathrm{BDC}_{n,k}$ (but it should be clear that this strategy generalizes beyond the BDC):

1. Assign a distinct input $x \in \mathcal{X}$ to each block in the kernel;
2. Let $\mathcal{S}_{x,k}$ denote the set of all length-$k$ subsequences of $x$. We partition $\mathcal{S}_{x,k}$ into up to 1024 disjoint subsets $\mathcal{T}_i$ for $i \in \{1, \ldots, 1024\}$. Then, the $i$-th thread of the block corresponding to $x$ computes the partial sum

$$A_i = \sum_{y \in \mathcal{T}_i} P_{n,k}(y|x) \log \left( \frac{X^{(t)}(x) P_{n,k}(y|x)}{Y^{(t)}(y)} \right),$$

   where $X^{(t)}(x)$ and $Y^{(t)}(y)$ are already known for all $x \in \mathcal{X}$ and $y \in \mathcal{Y}$. Concretely, if $y_i$ denotes the $i$-th length-$k$ subsequence of $x$ in some pre-specified ordering, then we take $\mathcal{T}_i = \{i, i + 1024, i + 2 \cdot 1024, \ldots\}$.
3. Compute $W^{(t)}(x) = \sum_{i=1}^{1024} A_i$.

Note that our description of our parallelization of the computation of $W^{(t)}(x)$ above is not complete. We still need to describe how we compute $\mathcal{S}_i$ and the relevant transition probabilities $P_{n,k}(y|x)$ in Item 2. In other words, we must be able to efficiently enumerate the subsequences of $x$ in $\mathcal{S}_i$ (to carry out the partial sum), and, for each $y \in \mathcal{S}_i$, compute the transition probabilities $P_{n,k}(y|x)$.

For computing the transition probabilities $P_{n,k}(y|x)$ we rely on the approach of Rubinstein and Con [RC23] discussed in Section 2.3.1 (in particular, see Algorithm 2). Namely, we pre-compute the full table of transition probabilities for input lengths $n' \approx n/2$. Then, each thread in our parallel computation accesses this table to compute $P_{n,k}(y|x)$ for the subsequences $y$ it enumerates over.

For enumerating over the required subsequences, a naive approach would be to iterate over all $\binom{n}{k}$ subsets of coordinates of $x$. However, given the typical memory constraints of GPUs, particularly

limited RAM, coupled with the concurrent execution of multiple blocks, it is infeasible to maintain large per-block lookup tables to track which subsequences have already been generated. Instead, we rely on dynamic programming-based techniques that only require a look-up table of size $O(nk)$ that can also be constructed in time $O(nk)$. Denote by $N_{x,k}$ the number of length-$k$ subsequences of $x$. Each thread in the block of the CUDA kernel assigned to input $x$ needs to enumerate over $N \approx \frac{N_{x,k}}{1024}$ length-$k$ subsequences of $x$. Using our dynamic programming-based method, this can be done in time $O(Nk)$, with small hidden constants. Since $k \ll 1024$, this yields a significant efficiency improvement over having a single thread enumerate all $N_{x,k}$ subsequences of $x$ in time $\Theta(N_{x,k})$, and so we improve significantly over previous implementations of the BAA.

Other computations in an iteration of the BAA, such as the computation of $Y^{(t)}(y)$ for all $y \in \mathcal{Y}$, can be similarly parallelized. In this case, we assign each $y \in \mathcal{Y}$ to a different block, and partition the length-$n$ supersequences $x$ of $y$ into up to 1024 disjoint subsets. Since the implementation of these ideas is similar to the above, we avoid discussing them further to avoid cluttering the exposition.

We discuss the methods we use to enumerate subsequences and supersequences in more detail in Sections 4 and 5, respectively. In Appendix A we discuss an alternative method for computing the channel output probabilities $Y^{(t)}(y)$ via subsequence enumeration, leveraging some simple symmetries of $\mathrm{BDC}_{n,k}$, that is faster for certain values of $k$.

# 4 Enumerating subsequences of a given length

In this section, we discuss the method we use to enumerate subsequences. More precisely, our goal is to, given a string $x \in \{0,1\}^n$, a subsequence length $k$, and an integer $j$, return the $j$-th length-$k$ subsequence of $x$ in some arbitrary but pre-specified order. As discussed in Section 3, our aim is an enumeration algorithm that is well-suited for execution on GPUs, which have limited memory constraints. We consider a dynamic programming-based approach that uses a small look-up table. For completeness, we describe the algorithm and prove its correctness.

## 4.1 Pre-computed tables

The enumeration is based on two precomputed tables, described below. In our CUDA implementation of the BAA, each block of the CUDA kernel constructs separate tables (because each block is assigned to a different $x$). In each block only one thread pre-computes the tables and stores them in dedicated memory.

- **NextPos**: for $0 \le i \le n$ and $b \in \{0,1\}$, **NextPos**$[i][b]$ stores the smallest index $p \ge i$ with $x_p = b$, or $n$ if no such index exists.

  The **NextPos** table is standard and computed in linear time by scanning from right to left. Algorithm 3 describes the procedure we use to construct the **NextPos** table.

- **Count**: for $0 \le i \le n$, $0 \le t \le k$, **Count**$[i][t]$ satisfies

  $$\mathbf{Count}[i][t] = \left|\{s \in \{0,1\}^t : s \text{ is a subsequence of } x_{[i:n-1]}\}\right|.$$

  We use the boundary conditions **Count**$[i][0] = 1$ for all $i$ (the empty subsequence) and **Count**$[n][t] = 0$ for all $t > 0$.

  The **Count** table is computed recursively using the next-occurrence indices

  $$j_0 = \mathbf{NextPos}[i][0], \qquad j_1 = \mathbf{NextPos}[i][1],$$

where we recall that $\mathbf{NextPos}[i][b] = n$ indicates that the symbol $b$ does not appear in $x_{[i:n-1]}$. For $t > 0$, we have

$$\mathbf{Count}[i][t] = \begin{cases} 0, & \text{if } j_0 = n \text{ and } j_1 = n, \\ \mathbf{Count}[j_0 + 1][t-1], & \text{if } j_0 < n \text{ and } j_1 = n, \\ \mathbf{Count}[j_1 + 1][t-1], & \text{if } j_1 < n \text{ and } j_0 = n, \\ \mathbf{Count}[j_0 + 1][t-1] + \mathbf{Count}[j_1 + 1][t-1], & \text{if } j_0 < n \text{ and } j_1 < n. \end{cases}$$

**Algorithm 3:** BUILDNEXTPOS($x$)

**Input:** Binary string $x_{[0:n-1]}$
**Output:** $\mathbf{NextPos}[0 \ldots n][0 \ldots 1]$

```
Initialize NextPos[i][b] ← n for all i, b;
for i ← n − 1 downto 0 do
    NextPos[i][0] ← NextPos[i + 1][0];
    NextPos[i][1] ← NextPos[i + 1][1];
    NextPos[i][x[i]] ← i;
return NextPos
```

**Computational complexity.** Computing the **NextPos** and **Count** tables requires time and space $O(nk)$.

### 4.2 The enumeration algorithm

We are now in place to describe our enumeration, or *unranking*, algorithm. The algorithm is described in Algorithm 4. We discuss it and prove its correctness in this section.

### 4.3 The enumeration algorithm

We are now in place to describe our enumeration, or *unranking*, algorithm. The algorithm is described in Algorithm 4. We discuss it and prove its correctness in this section.

Let $N_{x,k}$ denote the number of length-$k$ subsequences of a string $x$. The next lemma states, in particular, that the unranking algorithm always returns a length-$k$ subsequence of $x$ when $0 \le j < N_{x,k}$.

**Lemma 3.** *If* $0 \le j < N_{x,k}$*, then* stringUnrank$(x, k, j)$ *returns a length-*$k$ *subsequence of* $x$*. More precisely, at the start of iteration* $t$ *(i.e., after* $t$ *bits have been appended), the invariant*

$$0 \le j < \mathbf{Count}[i][\text{rem}], \qquad \textit{where } \text{rem} = k - t$$

*holds, and* subseq *is a subsequence of* $x_{[0:i-1]}$*.*

*Proof.* We prove this statement by induction.

**Initialization.** At the start of the execution of the algorithm we have $t = 0$, rem $= k$, $i = 0$, and the precondition requires $0 \le j < N_{x,k} = \mathbf{Count}[0][k]$. The empty string subseq is trivially a subsequence of the empty prefix.

**Algorithm 4:** stringUnrank$(x, k, j)$

**Input:** Binary string $x$ of length $n$; **NextPos** table; **Count** table; target length $k$; rank $j$ with $0 \le j < \textbf{Count}[0][k]$

**Output:** The $j$-th lexicographically smallest distinct subsequence of length $k$

```
1  i ← 0;
2  rem ← k;
3  subseq ← empty list;
4  while rem > 0 do
5      j_0 ← NextPos[i][0];
       // Find the index of the first 0 at or after position i
6      if j_0 < n then
7          c_0 ← Count[j_0 + 1][rem − 1];
           // Number of subsequences of x of length rem − 1 starting after j_0
8      else
9          c_0 ← 0
10     if j < c_0 then
11         append 0 to subseq;
12         i ← j_0 + 1;
           // Increment index to the position after the chosen 0
13         rem ← rem − 1;
14         continue;
15     j ← j − c_0;
16     j_1 ← NextPos[i][1];
       // Find the index of the first 1 at or after position i
17     if j_1 < n then
18         c_1 ← Count[j_1 + 1][rem − 1];
           // Number of subsequences of x of length rem − 1 starting after j_1
19     else
20         c_1 ← 0
21     if j < c_1 then
22         append 1 to subseq;
23         i ← j_1 + 1;
           // Increment index to the position after the chosen 1
24         rem ← rem − 1;
25         continue;
26 return subseq;
```

**Inductive step.** Assume the invariant holds at the $t$-th iteration, i.e.,

$$0 \le j < \mathbf{Count}[i][\text{rem}], \quad \text{subseq} \subseteq x_{[0:i-1]}, \quad \text{rem} = k - t > 0.$$

Let

$$j_0 = \mathbf{NextPos}[i][0], \qquad c_0 = \mathbf{Count}[j_0 + 1][\text{rem} - 1],$$

with $c_0 = 0$ if $j_0 = n$. Similarly, let

$$j_1 = \mathbf{NextPos}[i][1], \qquad c_1 = \mathbf{Count}[j_1 + 1][\text{rem} - 1],$$

with $c_1 = 0$ if $j_1 = n$. By the recurrence relation for **Count** we have

$$\mathbf{Count}[i][\text{rem}] = c_0 + c_1.$$

We proceed by cases.

1. *($j < c_0$)* In this case the algorithm appends 0 at position $j_0$, sets $i \leftarrow j_0 + 1$, rem $\leftarrow$ rem $- 1$, and leaves $j$ unchanged. Since $j < c_0 = \mathbf{Count}[j_0 + 1][\text{rem} - 1]$, we obtain

   $$0 \le j < \mathbf{Count}[i][\text{rem}],$$

   with the updated $i, \text{rem}$. The new subsequence is valid because it extends by a 0 occurring at $j_0$.

2. *($j \ge c_0$)* In this case we update $j \leftarrow j - c_0$. From the recurrence

   $$\mathbf{Count}[i][\text{rem}] = c_0 + c_1,$$

   the condition $j < \mathbf{Count}[i][\text{rem}]$ implies $j - c_0 < c_1$. The algorithm appends 1 at position $j_1$, sets $i \leftarrow j_1 + 1$, rem $\leftarrow$ rem $- 1$. Thus the invariant becomes

   $$0 \le j < \mathbf{Count}[i][\text{rem}],$$

   with the new $i, \text{rem}$, and the subsequence remains valid.

In both cases the invariant is preserved.

**Termination.** Each loop decreases rem by one. After $k$ iterations we have rem $= 0$ and exactly $k$ bits appended. By the invariant, subseq is a subsequence of $x$ of length $k$. Thus the algorithm always returns a valid subsequence of length $k$. □

Lemma 3 shows that the unranking algorithm in Algorithm 4 always returns a length-$k$ subsequence of $x$. Therefore, we are done if we prove that running the unranking algorithm with $j \neq j'$ yields distinct subsequences. To prove this we analyze how the state of the algorithm evolves from one iteration to the next. More precisely, for fixed $i$ and $r > 0$ define the "transition map"

$$f_{i,r} : \{0, \ldots, \mathbf{Count}[i][r] - 1\} \longrightarrow \{0, 1\} \times \{0, \ldots, n\} \times \{0, \ldots, r - 1\} \times \mathbb{N}$$

by

$$f_{i,r}(j) = \begin{cases} (0, \mathbf{NextPos}[i][0] + 1, r - 1, j), & j < c_0, \\ (1, \mathbf{NextPos}[i][1] + 1, r - 1, j - c_0), & \text{otherwise}, \end{cases}$$

where $c_0 = \mathbf{Count}[\mathbf{NextPos}[i][0] + 1][r - 1]$ (with $c_0 = 0$ if $\mathbf{NextPos}[i][0] = n$).

**Lemma 4.** *For fixed $i$ and $r > 0$ the map $f_{i,r}$ is injective.*

*Proof.* Fix any $j \neq j'$. If $f_{i,r}(j) = f_{i,r}(j')$ then, by the last coordinate of the output and the fact that $j \neq j'$, we may assume without loss of generality that $j' = j + c_0$. But this implies that $j' \geq c_0$, and so $f_{i,r}(j)$ and $f_{i,r}(j')$ differ in the first coordinate. □

**Lemma 5.** *Fix $x \in \{0,1\}^n$ and $k \in \{1,\ldots,n\}$. Then, the map $j \mapsto \text{stringUnrank}(x,k,j)$ for $j \in \{0,\ldots,\textbf{Count}[0][k]-1\}$ is injective.*

*Proof.* Suppose that $\text{stringUnrank}(x,k,j) = \text{stringUnrank}(x,k,j')$ with $0 \leq j < j' < \textbf{Count}[0][k]$. Let $(i_t, \text{rem}_t, j_t)$ and $(i'_t, \text{rem}'_t, j'_t)$ denote the states at the start of the $t$-th iteration when running the algorithm with $j$ and $j'$, respectively.

Since $\text{stringUnrank}(x,k,j) = \text{stringUnrank}(x,k,j')$, the chosen bits, i.e., the bit we add to subseq at each iteration of the algorithm, are equal. This means that $i_t = i'_t$ and $\text{rem}_t = \text{rem}'_t$ for all $t$ (the next search index depends only on the previous $i_t$ and the chosen bit).

We now argue by induction on $t$ that $j_t \neq j'_t$ for all $t$. For the base case $t = 0$, note that $j_0 = j < j'_0 = j'$ by assumption. For the induction step, we assume that $j_t \neq j'_t$ and show that $j_{t+1} \neq j'_{t+1}$. Because the chosen bit at iteration $t$ is the same in both $\text{stringUnrank}(x,k,j)$ and $\text{stringUnrank}(x,k,j')$, and the local transition $f_{i_t,\text{rem}_t}$ is injective by Lemma 4, the next values satisfy $j_{t+1} \neq j'_{t+1}$. By induction, we conclude that $j_k \neq j'_k$. However, when the algorithm terminates we have $\text{rem}_k = 0$ and $\textbf{Count}[i_k][0] = 1$, so $j_k = j'_k = 0$, a contradiction. □

Combining Lemmas 3 and 5 immediately yields the following theorem.

**Theorem 2.** *For any fixed $x \in \{0,1\}^n$ and $k \in \{1,\ldots,n\}$, the map*

$$j \mapsto \text{stringUnrank}(x,k,j)$$

*for $0 \leq j < \textbf{Count}[0][k]$ is a bijection between $\{0,\ldots,N_{x,k}-1\}$ and the set of length-$k$ subsequences of $x$.*

### 4.4 Computational complexity

We now discuss the complexity of the subsequence enumeration procedure described in Algorithm 4. As mentioned above, pre-computing the **NextPos** and **Count** tables only needs to be done once, and requires time $O(nk)$. Afterwards, each execution of the stringUnrank algorithm takes time $O(k)$, for any rank $j$. To see this, note that each iteration of the while cycle in Algorithm 4 takes time $O(1)$ and decreases rem by at least 1. Since rem is initially set to $k$, the claim follows.

To see the advantage of combining this method with our parallelization of a BAA iteration, recall from Section 3 that each thread in the block of the CUDA kernel associated with input $x$ enumerates over $N \approx \frac{N_{x,k}}{1024}$ length-$k$ subsequences of $x$. By the previous paragraph, the worst-case running time of a thread is $O(nk) + O(Nk)$, with mild hidden constants. In particular, since we only consider $k \ll 1024$, this worst-case running time improves significantly over a single thread enumerating over $N_{x,k}$ subsequences in time $O(nk + N_{x,k})$.

## 5 Enumerating supersequences of a given length

As discussed in Section 3, we also parallelize the computation of the channel output probabilities $Y^{(t)}(y)$ for every output $y \in \{0,1\}^k$. As a sub-routine of this computation, every thread in the block associated to output $y$ in the parallel implementation must enumerate over a certain subset

of length-$n$ supersequences of $y$, ordered in some pre-specified way. We discuss the methods we use to enumerate these supersequences. As in Section 4, our methods are tailored to our parallel architecture.

Before we begin, we note that the number of length-$n$ supersequences of $y$ only depends on the length of $y$. More precisely, the following holds.

**Lemma 6** ([CS75], for binary strings)**.** *For any $y \in \{0,1\}^k$, the number of length-$n$ supersequences of $y$ is*

$$\sum_{i=k}^{n} \binom{n}{i} = \sum_{i=0}^{n-k} \binom{n}{i}.$$

We divide our enumeration into two parts. In the first part we enumerate subsets of $\{1,\ldots,n\}$ of a given size $0 \le w \le n-k$. Intuitively, these subsets represent the coordinates of bits in the supersequence that are not matched to occurrences of $y$ as a subsequence. In the second part, we show how to map each such subset to a distinct supersequence of $y$. By Lemma 6 we cover all supersequences of $y$.

### 5.1 Enumerating subsets of unmatched bits

We begin by analyzing our procedure for enumerating size-$w$ subsets of $\{1,\ldots,n\}$. Recall that $\binom{n}{\le t} = \sum_{j=0}^{t} \binom{n}{j}$. We use the convention that $\binom{n}{\le -1} = 0$. For each $i \in \left\{0,\ldots,\binom{n}{\le n-k}-1\right\}$, let $w \in \{0,1,\ldots,n-k\}$ be the unique integer such that

$$\binom{n}{\le w-1} \le i < \binom{n}{\le w}.$$

We then define $j = i - \binom{n}{\le w-1}$, which satisfies $0 \le j < \binom{n}{w}$.

Algorithm 5 describes the procedure that, given $n$, $w$, and $j$, returns the $j$-th size-$w$ subset of $\{1,\ldots,n\}$ according to a pre-specified order.

It is clear that Algorithm 5 returns a size-$w$ subset of $\{1,\ldots,n\}$. It remains to prove that for fixed $n$ and $w$ different $j$'s yield different subsets.

**Lemma 7.** *Fix a sequence $y$ of length $k$. For any integers $n, w$ with $0 \le w \le n-k$ and any $0 \le j < j' < \binom{n}{w}$ we have*

$$\text{subsetUnrank}(n,w,j) \neq \text{subsetUnrank}(n,w,j').$$

*Proof.* We prove this result by induction on the pair $(w,n)$ under the lexicographic order.

**Base case.** If $w = 0$ then $\binom{n}{0} = 1$, so there is no pair $j < j'$.

**Inductive step.** Fix $(w,n)$ with $1 \le w < n-k$. Assume the lemma holds for all pairs $(w',n')$ with $(w',n') < (w,n)$. Fix also integers $0 \le j < j' < \binom{n}{w}$. The subsetUnrank$(n,w,j)$ procedure first finds the smallest $t \in \{1,\ldots,n\}$ such that

$$\sum_{i=0}^{t-1} \binom{n-i-1}{w-1} \le j < \sum_{i=0}^{t} \binom{n-i-1}{w-1}.$$

Likewise, subsetUnrank$(n,w,j')$ finds $t'$. We consider two cases:

**Algorithm 5:** subsetUnrank$(n, w, j)$

**Input:** Integers $n$, $w$, and $j$, where $0 \le j < \binom{n}{w}$
**Output:** List `subset` containing the $j$-th size-$w$ subset of $\{1, \ldots, n\}$

```
Initialize empty list subset ← [];
// Recall that indices start at 0
for t ← 0 to n − 1 do
    if w = 0 then
        break;
    c ← (n−t−1 choose w−1);
    // Number of size-w subsets whose smallest element is t
    if j < c then
        Append t to subset;
        w ← w − 1;
    else
        j ← j − c;
return subset;
```

- If $t \neq t'$, then the subsets returned by subsetUnrank$(n, w, j)$ and subsetUnrank$(n, w, j')$ differ in their smallest element.
- If $t = t'$, then both subsetUnrank$(n, w, j)$ and subsetUnrank$(n, w, j')$ add $t$ to their subsets. Then, they generate a size-$(w-1)$ subset of $\{t+1, \ldots, n\}$, exactly like a call to subsetUnrank$(n-(t-1), w-1, j_{\text{rem}})$ and subsetUnrank$(n-(t-1), w-1, j'_{\text{rem}})$, respectively, where

$$j_{\text{rem}} = j - \sum_{i=0}^{t-1} \binom{n-i-1}{w-1}, \quad j'_{\text{rem}} = j' - \sum_{i=0}^{t-1} \binom{n-i-1}{w-1}.$$

  Note that $(w-1, n-t-1) < (w, n)$ in lexicographic order. Hence, by the induction hypothesis, the two calls produce two distinct $(w-1)$-size subsets of $\{t+1, \ldots, n\}$. Prepending the same element $t$ to each yields two distinct $w$-size subsets of $\{1, \ldots, n\}$. □

### 5.2 From subsets to supersequences

We now analyze a procedure that constructs a length-$n$ supersequence of $y$ based on the subset output by subsetUnrank. This procedure described in Algorithm 6.

We begin by proving that RecSuperSeq always outputs a length-$n$ supersequence of $y$.

**Lemma 8.** *For every $i$ with $0 \le i < \binom{n}{\le n-k}$, the output $x =$ RecSuperSeq$(i, n, k, y)$ is a supersequence of $y$.*

*Proof.* Let $w$ and $M$ be the integer and subset computed in a call to RecSuperSeq$(i, n, k, y)$. By construction, $M$ has size $w \le n-k$. In the for loop we increment $r$ only when $p \notin M$ and $r < k$, in which case we match the next bit of $y$ to the next bit of $x$. Since the complement of $M$ has size $n - w \ge k$, when we exit the for loop we have $r = k$, and so we match all bits of $y$ to bits of $x$. □

Now we prove that calling RecSuperSeq on distinct $i$'s yields distinct supersequences of $y$.

**Algorithm 6:** RecSuperSeq$(i, n, k, y)$

**Input:** Index $i$ with $0 \le i < P(n-k)$; integers $n \ge k$; string $y \in \{0,1\}^k$.

**Output:** The $i$-th length-$n$ supersequence $x$ of $y$ according to some pre-specified order.

Find unique $w \in \{0, \ldots, n-k\}$ such that $\binom{n}{\le w-1} \le i < \binom{n}{\le w}$;

`// Determine the number of positions not used to match symbols of` $y$

Set $j \leftarrow i - \binom{n}{\le w-1}$;

`// Index among the` $\binom{n}{w}$ `choices of unused positions`

$S \leftarrow$ subsetUnrank$(n, w, j)$;

`// Compute the` $j$`-th size-`$w$ `subset of` $\{0, \ldots, n-1\}$

Initialize $x$ as an $n$-bit vector; set $r \leftarrow 0$;

`//` $r$ `indexes the next symbol of` $y$ `to be matched`

**for** $p = 0$ *to* $n-1$ **do**

  **if** $p \in S$ **then**

    **if** $r < k$ **then**

      set $x_p \leftarrow 1 - y_r$

    **else**

      set $x_p \leftarrow 1 - y_{k-1}$

    `// Choose a value that does not match the next symbol of` $y$

  **if** $r < k$ **then**

    set $x_p \leftarrow y_r$;

    $r \leftarrow r + 1$;

    `// Match the next symbol of` $y$

  **else**

    set $x_p$ arbitrarily (e.g., 0);

**return** $x$;

**Lemma 9.** *If $i \neq i'$, then* $\text{RecSuperSeq}(i, n, k, y) \neq \text{RecSuperSeq}(i', n, k, y)$.

*Proof.* Let $(w, S, j)$ be the integers and subset first computed by $\text{RecSuperSeq}(i, n, k, y)$, and $(w', S', j')$ the integer and subset first computed by $\text{RecSuperSeq}(i', n, k, y)$. Denote the supersequences by these calls by $x$ and $x'$, respectively.

There are two cases to consider. If $w \neq w'$ then $|S| = w \neq w' = |S'|$, and so $S \neq S'$ in particular. If $w = w'$ then $j \neq j'$, and so $S = \text{subsetUnrank}(n, w, j) \neq \text{subsetUnrank}(n, w, j') = S'$ by Lemma 7. Therefore, it is always the case that $S \neq S'$.

Let $\ell$ be the smallest index at which $S$ and $S'$ differ. By construction, this means that $S \cap \{0, \ldots, \ell - 1\} = S' \cap \{0, \ldots, \ell - 1\}$. Therefore, the construction of the supersequences $x$ and $x'$ up to position $\ell - 1$ is identical. In particular, both have consumed the same number $c$ of symbols from $y$ by position $\ell$, where

$$c = |\{u < j : u \notin S\}|.$$

At index $\ell$ the two calls behave differently. Without loss of generality, assume that $\ell \in S$ and $\ell \notin S'$. Then,

$$x_\ell = 1 - y_c, \quad x'_\ell = y_c,$$

and so $x_\ell \neq x'_\ell$. □

Combining Lemmas 6, 8 and 9 immediately yields the following result.

**Theorem 3.** *For any $y \in \{0,1\}^k$ and $n \geq k$ the map*

$$i \mapsto \text{RecSuperSeq}(i, n, k, y)$$

*is a bijection between $\{0, \ldots, S_{k,n} - 1\}$ and the set of length-$n$ supersequences of $y$.*

### 5.3 Computational complexity

We now discuss the complexity of the supersequence enumeration procedure described in Algorithm 6. Each call to subsetUnrank takes $O(n)$ time. The remainder of Algorithm 6 also runs in $O(n)$ time, and so, overall, this algorithm runs in $O(n)$ time.

As in Section 4, we gain an advantage by combining this method with our parallelization of a BAA iteration. Each thread in a block associated with output $y \in \{0,1\}^k$ enumerates over $N \approx \frac{S_{k,n}}{1024}$ length-$n$ supersequences of $x$. By the previous paragraph, the running time of a thread is $O(N \cdot n)$, with a mild hidden constant. Since $n \ll 1024$ in our setting, this improves significantly over a single thread enumerating over $S_{k,n}$ supersequences.

## 6 Results

Using our optimized parallelized implementation of the BAA, we managed to compute good upper bounds on $C_{n,k}$ for all pairs $(n, k)$ with $k \leq n \leq 29$. Going even further, we also managed to compute good upper bounds on $C_{31,k}$ for all $k \leq 18$. For most of the upper bound computations a tolerance of $a = 0.005$ was enforced, ensuring that the true value of $C_{n,k}$ exceeds the information rate returned by the BAA by at most 0.005 (see Theorem 1 and the surrounding discussion in Section 2.3). The only exceptions are in the approximation of $C_{31,k}$ for $14 \leq k \leq 18$, where we enforced a tolerance of $a = 0.05$ due to the rapid increase in time per iteration, reaching 2500 seconds per iteration for $k = 18$. Reported capacity upper bounds are obtained by adding the respective tolerance value to the rate output by the BAA.

All computations were done with an *RTX 5070 Ti* GPU. For $n = 29$, the computations took at most 1100 iterations to complete, and for $k \approx \frac{n}{2}$ each iteration took approximately 400 seconds, taking 5 days to complete. For $n = 31$, the computations for $k \leq 13$ took less then 500 seconds to complete, taking up to one week to complete, especially for $k = 13$. For $14 \leq k \leq 18$, each iteration took more than 800 seconds to complete, hence the need for a higher tolerance.

The upper bounds on $C_{29,k}$ for all $k \in \{1, \ldots, 28\}$ are reported in Table 1. The upper bounds we obtained on $C_{31,k}$ are reported in Table 2. Links to files containing the nearly-optimal input distributions output by the BAA for the $\mathrm{BDC}_{29,k}$ channels for all $k$ and the $\mathrm{BDC}_{31,k}$ channels for all $k \leq 18$ can be found in our repository [Pin26].

To upper bound the $C_{31,k}$ values for $k > 18$ we combine our upper bounds on $C_{n,k}$ for $n \leq 29$ and the following known lemma that upper bounds $C_{n,k}$ based on $C_{n',k'}$ for $n' < n$.

**Lemma 10** ([RC23])**.** *For every $s \in \{1, \ldots, n\}$ we have that*

$$C_{n,k} \leq \sum_{i=0}^{s} \frac{\binom{s}{i}\binom{n-s}{k-i}}{\binom{n}{k}} (C_{s,i} + C_{n-s,k-i}).$$

More precisely, we set $s = 2$ and combine our upper bounds on $C_{29,i}$ reported in Table 1 with easy to compute upper bounds on $C_{2,j}$, for all relevant $i$ and $j$.

We use our upper bounds on the $C_{29,k}$ and $C_{31,k}$ values to obtain upper bounds on the capacity $C(d)$ of the binary deletion channel with deletion probability $d$ via Lemma 2. These bounds are reported in Table 3, where they are also compared to the previous best known upper bounds [RC23]. Note that since our upper bounds on $C_{31,k}$ are loose for $k > 18$, the upper bounds on $C(d)$ obtained by plugging our upper bounds on $C_{29,k}$ into Lemma 2 are better than those obtained via our upper bounds on $C_{31,k}$ when $d$ is not large.

**Capacity upper bounds in the high-noise regime.** Our improved upper bounds on $C(d)$ also lead to an improved upper bound on $C(d)$ in the asymptotic regime $d \to 1$. This is obtained via the following result due to Rahmati and Duman [RD15].

**Lemma 11** ([RD15])**.** *Let $\lambda, d' \in [0, 1]$ and define $d = \lambda d' + 1 - \lambda$. Then,*

$$\frac{C(d)}{1-d} \leq \frac{C(d')}{1-d'}.$$

When $d' = 0.64$, our new upper bound yields

$$\frac{C(d')}{1-d'} \leq 0.3578.$$

Therefore, instantiating Lemma 11 with this upper bound yields

$$C(d) \leq 0.3578(1-d)$$

for all $d \geq 0.64$.

## Acknowledgments

We thank Roni Con for insightful discussions and detailed feedback on earlier versions of this work.

## References


[Ari72] Suguru Arimoto. An algorithm for computing the capacity of arbitrary discrete memoryless channels. *IEEE Transactions on Information Theory*, 18(1):14–20, 1972.

[Bla72] Richard E. Blahut. Computation of channel capacity and rate-distortion functions. *IEEE Transactions on Information Theory*, 18(4):460–473, 1972.

[Che19] Mahdi Cheraghchi. Capacity upper bounds for deletion-type channels. *J. ACM*, 66(2):9:1–9:79, March 2019.

[CK15] Jason Castiglione and Aleksandar Kavčić. Trellis based lower bounds on capacities of channels with synchronization errors. In *2015 IEEE Information Theory Workshop - Fall (ITW)*, pages 24–28, Oct 2015.

[CR19] Mahdi Cheraghchi and João Ribeiro. Sharp analytical capacity upper bounds for sticky and related channels. *IEEE Transactions on Information Theory*, 65(11):6950–6974, Nov 2019.

[CR21] Mahdi Cheraghchi and João Ribeiro. An overview of capacity results for synchronization channels. *IEEE Transactions on Information Theory*, 67(6):3207–3232, 2021.

[CS75] V. Chvátal and D. Sankoff. Longest common subsequence of two random sequences. *Journal of Applied Probability*, (12):306–315, 1975.

[Dal11] Marco Dalai. A new bound on the capacity of the binary deletion channel with high deletion probabilities. In *2011 IEEE International Symposium on Information Theory (ISIT)*, pages 499–502. IEEE, 2011.

[DG06] Suhas Diggavi and Matthias Grossglauser. On information transmission over a finite buffer channel. *IEEE Transactions on Information Theory*, 52(3):1226–1237, March 2006.

[DK07] Eleni Drinea and Adam Kirsch. Directly lower bounding the information capacity for channels with i.i.d. deletions and duplications. In *2007 IEEE International Symposium on Information Theory (ISIT)*, pages 1731–1735, 2007.

[DM07] Eleni Drinea and Michael Mitzenmacher. Improved lower bounds for the capacity of iid deletion and duplication channels. *IEEE Transactions on Information Theory*, 53(8):2693–2714, 2007.

[DMP07] Suhas Diggavi, Michael Mitzenmacher, and Henry D. Pfister. Capacity upper bounds for the deletion channel. In *2007 IEEE International Symposium on Information Theory (ISIT)*, pages 1716–1720, 2007.

[Dob67] Roland L. Dobrushin. Shannon's theorems for channels with synchronization errors. *Problemy Peredachi Informatsii*, 3(4):18–36, 1967.

[FD10] Dario Fertonani and Tolga M. Duman. Novel bounds on the capacity of the binary deletion channel. *IEEE Transactions on Information Theory*, 56(6):2753–2765, 2010.

[Gal61] Robert G. Gallager. Sequential decoding for binary channels with noise and synchronization errors. Technical report, MIT Lexington Lincoln Laboratory, 1961.

[HMG19] Reinhard Heckel, Gediminas Mikutis, and Robert N. Grass. A characterization of the dna data storage channel. *Scientific reports*, 9(1):9663, 2019.

[HS21] Bernhard Haeupler and Amirbehshad Shahrasbi. Synchronization strings and codes for insertions and deletions—a survey. *IEEE Transactions on Information Theory*, 67(6):3190–3206, 2021.

[ISW16] Aravind R. Iyengar, Paul H. Siegel, and Jack Keil Wolf. On the capacity of channels with timing synchronization errors. *IEEE Transactions on Information Theory*, 62(2):793–810, 2016.

[KD10] Adam Kirsch and Eleni Drinea. Directly lower bounding the information capacity for channels with i.i.d. deletions and duplications. *IEEE Transactions on Information Theory*, 56(1):86–102, Jan 2010.

[MBT10] Hugues Mercier, Vijay K. Bhargava, and Vahid Tarokh. A survey of error-correcting codes for channels with symbol synchronization errors. *IEEE Communications Surveys Tutorials*, 12(1):87–96, First Quarter 2010.

[MD06] Michael Mitzenmacher and Eleni Drinea. A simple lower bound for the capacity of the deletion channel. *IEEE Transactions on Information Theory*, 52(10):4657–4660, 2006.

[Mit08] Michael Mitzenmacher. Capacity bounds for sticky channels. *IEEE Transactions on Information Theory*, 54(1):72–77, 2008.

[Mit09] Michael Mitzenmacher. A survey of results for deletion channels and related synchronization channels. *Probability Surveys*, 6:1–33, 2009.

[MTL12] Hugues Mercier, Vahid Tarokh, and Fabrice Labeau. Bounds on the capacity of discrete memoryless channels corrupted by synchronization and substitution errors. *IEEE Transactions on Information Theory*, 58(7):4306–4330, 2012.

[OAC+18] Lee Organick, Siena Dumas Ang, Yuan-Jyue Chen, Randolph Lopez, Sergey Yekhanin, Konstantin Makarychev, Miklos Z Racz, Govinda Kamath, Parikshit Gopalan, Bichlien Nguyen, et al. Random access in large-scale DNA data storage. *Nature Biotechnology*, 36(3):242, 2018.

[Pin26] Martim Pinto. GPU code for computing capacity upper bounds for the binary deletion channel. https://github.com/Pintz45/BDC-Upper-Bounds-GPU, 2026.

[RA13] Mahdi Ramezani and Masoud Ardakani. On the capacity of duplication channels. *IEEE Transactions on Communications*, 61(3):1020–1027, 2013.

[RC23] Ittai Rubinstein and Roni Con. Improved upper and lower bounds on the capacity of the binary deletion channel. In *2023 IEEE International Symposium on Information Theory (ISIT)*, pages 927–932, 2023.

[RD15] Mojtaba Rahmati and Tolga M. Duman. Upper bounds on the capacity of deletion channels using channel fragmentation. *IEEE Transactions on Information Theory*, 61(1):146–156, 2015.

[Sha48] Claude E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423, 1948.

[VD68] Nikita D. Vvedenskaya and Roland L. Dobrushin. The computation on a computer of the channel capacity of a line with symbol drop-out. *Problemy Peredachi Informatsii*, 4(3):92–95, 1968.

[VTR13] Ramji Venkataramanan, Sekhar Tatikonda, and Kannan Ramchandran. Achievable rates for channels with deletions and insertions. *IEEE Transactions on Information Theory*, 59(11):6990–7013, 2013.

[WGGH23] Franziska Weindel, Andreas L. Gimpel, Robert N. Grass, and Reinhard Heckel. Embracing errors is more effective than avoiding them through constrained coding for DNA data storage. In *2023 59th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pages 1–8, 2023.

[Yeu08] Raymond W. Yeung. *Information Theory and Network Coding*. Springer Science & Business Media, 2008.

[YGM17] S. M. Hossein Tabatabaei Yazdi, Ryan Gabrys, and Olgica Milenkovic. Portable and error-free DNA-based data storage. *Scientific reports*, 7(1):5011, 2017.

[Zig69] Kamil Sh. Zigangirov. Sequential decoding for a binary channel with drop-outs and insertions. *Problemy Peredachi Informatsii*, 5(2):23–30, 1969.

Table 1: Upper bounds on $C_{29,k}$.

| $k$ | Upper bound on $C_{29,k}$ |
|---|---|
| 1 | 1.0000 |
| 2 | 1.1898 |
| 3 | 1.5165 |
| 4 | 1.8137 |
| 5 | 2.0916 |
| 6 | 2.3761 |
| 7 | 2.6647 |
| 8 | 2.9598 |
| 9 | 3.2663 |
| 10 | 3.5867 |
| 11 | 3.9247 |
| 12 | 4.2841 |
| 13 | 4.6727 |
| 14 | 5.0930 |
| 15 | 5.5573 |
| 16 | 6.0713 |
| 17 | 6.6464 |
| 18 | 7.2964 |
| 19 | 8.0364 |
| 20 | 8.8850 |
| 21 | 9.8659 |
| 22 | 11.0096 |
| 23 | 12.3545 |
| 24 | 13.9487 |
| 25 | 15.8526 |
| 26 | 18.1454 |
| 27 | 20.9387 |
| 28 | 24.4132 |
| 29 | 29.0000 |

Table 2: Upper bounds on $C_{31,k}$.

| $k$ | Upper bound on $C_{31,k}$ |
|---|---|
| 1 | 1.0000 |
| 2 | 1.1888 |
| 3 | 1.5136 |
| 4 | 1.8086 |
| 5 | 2.0831 |
| 6 | 2.3632 |
| 7 | 2.6458 |
| 8 | 2.9332 |
| 9 | 3.2298 |
| 10 | 3.5384 |
| 11 | 3.8605 |
| 12 | 4.2005 |
| 13 | 4.5623 |
| 14 | 4.9511 |
| 15 | 5.3895 |
| 16 | 5.8826 |
| 17 | 6.3947 |
| 18 | 6.9592 |
| 19 | 8.3882 |
| 20 | 9.1247 |
| 21 | 9.9515 |
| 22 | 10.8865 |
| 23 | 11.9522 |
| 24 | 13.1766 |
| 25 | 14.5945 |
| 26 | 16.2486 |
| 27 | 18.1927 |
| 28 | 20.4969 |
| 29 | 23.2603 |
| 30 | 30.0000 |
| 31 | 31.0000 |

# A Computing channel output probabilities using subsequences

In this section we present an alternative method to compute the channel output probabilities $Y^{(t)}(y)$ of the $\mathrm{BDC}_{n,k}$ in an iteration of the BAA using subsequence enumeration and exploiting symmetries of the $\mathrm{BDC}_{n,k}$ and the input distributions obtained through the BAA. We found this method to be faster in practice than the method discussed in Section 5 for approximating $C_{n,k}$ using the BAA when $k$ is close to $n/2$. We note that the idea of using channel symmetries to speed up computations is not new: it was used before in the optimized implementation of the BAA in [RC23], although it was not analyzed in the paper itself. We exploit these symmetries in a somewhat different way, and we provide an analysis for completeness.

We begin by providing some intuition behind the method. Recall that in Section 3 we computed $Y^{(t)}(y)$ for all $y \in \{0,1\}^k$ in parallel by assigning each block in the CUDA kernel to a different output $y \in \{0,1\}^k$, and then having threads within that block enumerate over appropriate subsets of length-$n$ supersequences of $y$ using the method from Section 5. Alternatively, we can also compute $Y^{(t)}(y)$ for all $y \in \{0,1\}^k$ by maintaining an array (with all entries initialized to 0) storing $Y^{(t)}(y)$ for all $y$. Then, instead we assign each block in the CUDA kernel to a different *input* $x \in \{0,1\}^n$, and have threads within that block enumerate over appropriate subsets of length-$k$ subsequences of $x$ using the method from Section 5, updating the array storing $Y^{(t)}$ as they go along.

This approach can be sped up by taking into account some simple symmetries of the $\mathrm{BDC}_{n,k}$. In

Table 3: Upper bounds on $C(d)$ obtained by combining Lemma 2 with the upper bounds on $C_{29,k}$ from Table 1 or the upper bounds on $C_{31,k}$ from Table 2, and taking the minimum between these two. For each value of $d$, we list which $n$ (29 or 31) gave the best upper bound. We also list the previous best known upper bounds [RC23] for comparison.

| $d$ | New upper bound | Previous upper bound [RC23] |
|---|---|---|
| 0.01 | 0.9557 ($n = 29$) | 0.9583 |
| 0.02 | 0.9141 ($n = 29$) | 0.9189 |
| 0.03 | 0.8751 ($n = 29$) | 0.8817 |
| 0.04 | 0.8385 ($n = 29$) | 0.8467 |
| 0.05 | 0.8039 ($n = 29$) | 0.8139 |
| 0.10 | 0.6577 ($n = 29$) | 0.6762 |
| 0.15 | 0.5454 ($n = 29$) | 0.5660 |
| 0.20 | 0.4574 ($n = 29$) | 0.4786 |
| 0.25 | 0.3876 ($n = 29$) | 0.4083 |
| 0.30 | 0.3314 ($n = 29$) | 0.3513 |
| 0.35 | 0.2857 ($n = 29$) | 0.3045 |
| 0.40 | 0.2480 ($n = 29$) | 0.2648 |
| 0.45 | 0.2164 ($n = 29$) | 0.2309 |
| 0.50 | 0.1896 ($n = 29$) | 0.2015 |
| 0.55 | 0.1652 ($n = 31$) | 0.1755 |
| 0.60 | 0.1438 ($n = 31$) | 0.1524 |
| 0.64 | 0.1288 ($n = 31$) | – |
| 0.65 | 0.1253 ($n = 31$) | 0.1313 |
| 0.68 | 0.1151 ($n = 31$) | 0.1199 |

more detail, for an arbitrary integer $n \geq 1$ and a string $x \in \{0,1\}^n$ let $\mathrm{cpl}(x)$ denote its coordinate-wise complement (so that $\mathrm{cpl}(x)_i = 1 - x_i$) and $\mathrm{rev}(x)$ denote its reversal (so that $\mathrm{rev}(x)_i = x_{n-1-i}$), where we write $x = (x_0, \ldots, x_{n-1})$. Note that these functions are their own inverses. Then, for $g = \mathrm{cpl}$ or $g = \mathrm{rev}$ and any $x \in \{0,1\}^n$ and $y \in \{0,1\}^k$, we have

$$P_{n,k}(g(y)|g(x)) = P_{n,k}(y|x). \tag{2}$$

The same extends directly to the composition $\mathrm{cpl} \circ \mathrm{rev}$. Using Equation (2), we can derive analogous symmetries of the input and output distributions produced by the various iterations of the BAA.

**Theorem 4.** *Suppose the BAA applied to the* $\mathrm{BDC}_{n,k}$ *is initialized with a uniform input distribution* $X^{(0)}$*. Then, for every* $t \geq 1$*,* $x \in \{0,1\}^n$*,* $y \in \{0,1\}^k$*, and* $g = \mathrm{cpl}$ *or* $g = \mathrm{rev}$ *we have*

$$X^{(t)}(g(x)) = X^{(t)}(x)$$

*and*

$$Y^{(t)}(g(y)) = Y^{(t)}(y).$$

*Proof.* We established the desired statement by induction in $t$. For brevity, we write $P = P_{n,k}$.

The base case $t = 0$ is clear since $X^{(0)}$ is uniform over $\{0,1\}^n$. Now fix $t \geq 1$ and suppose that $X^{(t)}(g(x)) = X^{(t)}(x)$ for all $x \in \{0,1\}^n$. We show that the same thing holds for $X^{(t+1)}$. First, note that

$$\begin{aligned} Y^{(t)}(g(y)) &= \sum_{x \in \{0,1\}^n} X^{(t)}(x) P(g(y)|x) \\ &= \sum_{x \in \{0,1\}^n} X^{(t)}(g(x)) P(y|g(x)) \\ &= \sum_{x' \in \{0,1\}^n} X^{(t)}(x') P(y|x') \\ &= Y^{(t)}(y) \end{aligned} \tag{3}$$

for all $y \in \{0,1\}^k$. The second equality uses the induction hypothesis. The third equality uses the fact that $g$ is a bijection. Then,

$$\begin{aligned} W^{(t)}(g(x)) &= \prod_{y \in \{0,1\}^k} \left( \frac{X^{(t)}(g(x)) P(y|g(x))}{Y^{(t)}(y)} \right)^{P(y|g(x))} \\ &= \prod_{y \in \{0,1\}^k} \left( \frac{X^{(t)}(x) P(g(y)|x)}{Y^{(t)}(g(y))} \right)^{P(g(y)|x)} \\ &= \prod_{y' \in \{0,1\}^k} \left( \frac{X^{(t)}(x) P(y'|x)}{Y^{(t)}(y')} \right)^{P(y'|x)} \\ &= W^{(t)}(x). \end{aligned}$$

The second equality uses the induction hypothesis and Equation (3). Since $X^{(t+1)}$ is obtained by normalizing $W^{(t)}$ the desired result follows. □

We can use Theorem 4 to slightly simplify the computation of $(Y^{(t)}(y))_{y\in\{0,1\}^k}$. Given a string $y \in \{0,1\}^k$, we define its orbit $\mathcal{O}_y = \{y, \mathrm{cpl}(y), \mathrm{rev}(y), \mathrm{rev} \circ \mathrm{cpl}(y)\}$. To each orbit $\mathcal{O}_y$ we associate as its *representative* the smallest string $y' \in \mathcal{O}_y$ with respect to the lexicographic order, and denote it by $\mathrm{rep}(y)$. We denote the set of all representatives in $\{0,1\}^k$ by $\mathcal{R}_k$.

By Theorem 4, it suffices to compute $Y^{(t)}(r)$ for all representatives $r \in \mathcal{R}_k$. Furthermore, we have

$$\begin{aligned} Y^{(t)}(r) &= \sum_{x\in\{0,1\}^n} X^{(t)}(x) P_{n,k}(r|x) \\ &= \sum_{x\in\mathcal{R}_n} X^{(t)}(x) \sum_{x'\in\mathcal{O}_x} P_{n,k}(r|x') \\ &= \sum_{x\in\mathcal{R}_n} X^{(t)}(x) \cdot \frac{|\mathcal{O}_x|}{|\mathcal{O}_r|} \sum_{y\in\mathcal{O}_r} P_{n,k}(y|x). \end{aligned}$$

The second equality uses Theorem 4. To prove the last equality, we can, for example, use a group-theoretic argument. Let $G$ be the group generated by cpl and rev via composition of functions. For a binary string $z$, define the *stabilizer* $\mathrm{Stab}_z = \{g \in G : g(z) = z\}$. Note that $g^{-1} = g$ for every $g \in G$, and that $\mathcal{O}_z$ is the orbit of $z$ under the action of $G$. Then,

$$\begin{aligned} \sum_{x'\in\mathcal{O}_x} P_{n,k}(r|x') &= \frac{1}{|\mathrm{Stab}_x|} \sum_{g\in G} P_{n,k}(r|g(x)) \\ &= \frac{1}{|\mathrm{Stab}_x|} \sum_{g\in G} P_{n,k}(g(r)|x) \\ &= \frac{|\mathrm{Stab}_r|}{|\mathrm{Stab}_x|} \sum_{y\in\mathcal{O}_r} P_{n,k}(y|x) \\ &= \frac{|\mathcal{O}_x|}{|\mathcal{O}_r|} \sum_{y\in\mathcal{O}_r} P_{n,k}(y|x). \end{aligned}$$

The second equality uses the fact that $P_{n,k}(r|g(x)) = P_{n,k}(g^{-1}(r)|x) = P_{n,k}(g(r)|x)$ for any $r$ and $x$, and the last equality uses the fact that $|\mathrm{Stab}_z| = |G|/|\mathcal{O}_z|$ for any $z$.

The discussion above motivates the procedure described in Algorithm 7.

**Algorithm 7:** Computing $Y^{(t)}(y)$ for all $y \in \{0,1\}^k$

**Input :** Input size $n$, output size $k$, input distribution $X^{(t)}$
**Output:** Array $Y^{(t)}$ indexed by orbit representatives of outputs

Initialize $Y^{(t)}(r) \leftarrow 0$ for all orbit representatives $r \in \mathcal{R}_k$
**foreach** $x \in \mathcal{R}_n$ **do**
  $o_x \leftarrow |\mathcal{O}_x|$
  **foreach** subsequence $y$ of $x$ **do**
    $o_y \leftarrow |\mathcal{O}_y|$
    $r \leftarrow \mathrm{rep}(y)$
    $Y^{(t)}(r) \leftarrow Y^{(t)}(r) + \frac{o_x}{o_y} \cdot X^{(t)}(x) \cdot P_{n,k}(y|x)$
**return** $Y^{(t)}$