1 Week 1

1.1 Multiway paper

This week, I mainly worked on the Ahlswede's [2] paper. Here is a summary of what I did:

- The "$\sqrt{n}$ trick" is described in [1]. The idea behind the section F's 3 step algorithm is also described in previous works of Ahlswede. I will take a look at the referenced article next week.
- I think the section B. can be described more intuitively as follows:
  - Ω is the set of *terminals* which are basically the communication devices.
  - Γ is the set of *messengers* which can be viewd as a pair of transmitter and receiver. For each messenger γ, $\mathcal{N}_\gamma$ is the set of messages that the transmitter and receiver communicate with.
  - For each terminal $\omega \in \Omega$, the set of transmitters on that terminal is denoted by $\mathcal{A}_\omega$.
  - For each terminal $\omega \in \Omega$, the set of receiver on that terminal is denoted by $\mathcal{B}_\omega$. — there is a typo in the definition given in the paper, the last $\mathcal{B}_\omega$ should be changed to ω.
  - For each terminal $\omega \in \Omega$, the set of available feedback lines is denoted by Φ_ω. — there is a typo here too, $\Phi_\omega \subset \Omega$ and not Γ.
  - The channel W is discrete and memoryless.

  the given assumption can also be interpreted as follows

  - $\mathcal{A}_\omega \cap \mathcal{B}_\omega = \emptyset$: because otherwise, the transmitter and receiver would be placed on the same terminal which makes communication via channel unnecessary.

    $\cup_{\omega\in\Omega}\mathcal{A}_\omega = \cup_{\omega\in\Omega}\mathcal{B}_\omega = \Gamma$: we can assume that each transmitter/receiver is placed only on one terminal.
  - If $|\mathcal{X}_\omega| = |\mathcal{Y}_\omega| = 1$, then terminal can not transmit or receive any information.
  - If $|\mathcal{X}_\omega|$ then the terminal can not send information, hence no transmitter should be placed on it. Similarly for receiving.
  - I did not fully understand what is logic behind A_4 but I guess that is related to relay channels, since the relays do not send or decode data.
  - $\omega \in \Phi_\omega$ every terminal should know what it received.
  - If $\gamma \in \mathcal{A}_\omega \cap \mathcal{B}_{\omega'}$, then the transmitter of γ is on ω and its receiver is on ω'. Then, all the information available at ω' is feedbacked to ω, i.e. $\Phi_{\omega'} \subset \Phi_\omega$.
  - Passive decoders do not need to transmit anything.
- On section C:
  - Randomized feedback is not explained, I dont see what makes them different than the stochastic feedback defined later.
  - "concatentation of strategies" after equation (1.6) is ambiguous.
  - Derivation of equation (1.8) might be something like the following, but I am not sure as the definition are not formal.

$$
\begin{aligned}
\mu(\mathcal{F}_{m+n}) &= \max_{f^{n+m}\in\mathcal{F}_{n+m}} H(Y^{n+m}(f^{n+m})) \\
&\geq \max_{f^{n+m}\in\mathcal{F}_n\times\mathcal{F}_m} H(Y^{n+m}(f^{n+m})) \\
&= \max_{f^n\in\mathcal{F}_n} \max_{f^m\in\mathcal{F}_m} H(Y^n(f^n), Y^m(f^m)) \\
&= \max_{f^n\in\mathcal{F}_n} \max_{f^m\in\mathcal{F}_m} H(Y^n(f^n)) + H(Y^m(f^m)) \\
&= \mu(\mathcal{F}_n) + \mu(\mathcal{F}_m)
\end{aligned}
$$

    I used independence in the second and the third line.

– Given the above inequality, $\mu(\mathcal{F}_n) \geq n\mu(\mathcal{F}_1)$ and therefore

$$\mu((\mathcal{F}_n)_{n=1}^{\infty}) = \lim_{n\to\infty} \frac{1}{n\mu(\mathcal{F}_n)} \leq \lim_{n\to\infty} \frac{1}{n^2\mu(\mathcal{F}_1)} =?0$$

and since $\mu \geq 0$, then $\mu = 0$??!.

– I skimmed the remaining sections. I am not sure how the mystery number μ is related to the 3-step algorithm.

1.2 Randomized prime

I also worked on the randomized prime generation idea. Consider the following algorithm Instead of checking

Algorithm 1: Generating random primes

input : n, t
output: A uniform n-bit prime
for $i = 1 \to t$ **do**
  $p \leftarrow \{0,1\}^n$
  if *p is prime* **then**
    return p
  end
end
return $\perp$

that p is a prime, we can check if p passes the Miller-Rabin test or not, which is more efficient – running in $O(n^3)$ rather than in $O(2^n)$ for a simple primality test, or $\tilde{O}n^6$ for AKS primality test. By applying the Miller-Rabin test multiple times, the mathbbPability of error (a composite number passes the test) decreases rapidly. Moreover, by letting n to be large enough (an asymptotic formula can be derived from the Prime Number Theorem), we can be sure that π_K can be represented by n-bits, and therefore our analysis for the 3-step algorithms remains unchanged.

For the next week I am going to do a more thorough derivation of the above idea and implement it.

2 Week 2

Add the randomized prime generation and a testing python code to plot the error rate.

3 Week 3

3.1 The simulation code

3.1.1 `Channel` class

```
typedef function<uint64_t(uint64_t)> ChannelFunc;
typedef uint64_t chnl_output;
typedef uint64_t chnl_input;
class Channel
{
private:
    uint64_t x, y; /* size of the input and output alphabet
     X = {1, 2, ... , x} , Y = {1, 2, ... , y} */
    ChannelFunc* f; // the randomized function of channel

public:
    Channel(uint64_t x, uint64_t y, ChannelFunc* f);
    ~Channel();
    chnl_output transmit(chnl_input symb); // returns the result of
       tranmission of symb
};
```

The channel class models a discrete memoryless channel. The ChannelFunc is a function that takes an input character – the characters are modeled as indicies– and returns an output character. This function might be randomized as well. In fact the channel's transition matrix W should be implemented in ChannelFunc and be given as input to the constructor. It was wiser to let indicies start from 0 and I will refactor the code.

The transmit method simply calls the ChaanelFunc f on the given symbol.

3.1.2 Identification class

```
typedef function<vector<chnl_input>* (uint64_t)> ID_EncodingFunction; //
    encoder
typedef function<uint64_t (const vector<chnl_output> &)>
    ID_DecodingFunction; //decoder
typedef pair<ID_EncodingFunction* ,ID_DecodingFunction*> ID_Code;



class IdentificationCode
{
private:
     uint64_t N; // number of messages N = {1, 2, ... , N}
     uint64_t n; // block length


     ID_EncodingFunction* encoder;
     ID_DecodingFunction* decoder;
     double first_error ;
     double second_error;
     bool valid_construction;

public:
     IdentificationCode(uint64_t N, uint64_t n);
     ~IdentificationCode();
     void constructID_Code(const Channel & C, function<ID_Code* (const
        Channel &,uint64_t,uint64_t)> construction_method);
     uint64_t getN();
     uint64_t getn();
     double getFirstKindError();
     double getSecondKindError();
     vector<chnl_input>* encode(uint64_t message);
     uint64_t decode(const vector<chnl_output> &received);
};
```

This class models an identification code for a given channel C. The construction of the codes is done by constructID_Code which takes a construction_method to do the job. The encoder is a function that takes a message and encodes to a block of channel's inpute character. The decoder returns **the number of identified messages**. Under a uniform distribution on messages, the second error rate – false identification – is equal to the number of identified messages divided by the number of messages N. I have not implemented the getFirstKindError and getSecondKindError yet.

There is a similar transmission class which is supposed to model transmission codes. I added this class because of the constructions in [1] and [5] that use transmission codes. However, I have only implemented the 3-step algorithm so far.

3.1.3 Codes class

```
ID_Code* NoiselessBSC_ID(const Channel &C, uint64_t N, uint64_t n); // the
    3 step coding scheme
```

For now it only implements the 3-step algorithm.

3.1.4 Simulate class

```
double simulate(Channel & C,uint64_t N , uint64_t n ,  function<ID_Code* (
    const Channel &,uint64_t,uint64_t)> construction_method);
```

```
/* randomly generate a messages from N= {1,2, ... , N} and  transmit it
   over the given channel with the given code*/
```

It takes as input a `Channel C` and a identification code constructor `construction_method` then, it simulates the transmission of a message over channel. For now, it outputs the second error rate of the Identification code.

3.1.5 main

```
int main(int argv, char *argc[])
{
    Channel noiselessBSC = Channel(2, 2, new ChannelFunc([](chnl_input x)
                                                        { return x; }));
    file_address = (argv >= 2) ? argc[1] : "C:/Users/Emad␣Zinoghli/Desktop/
       Codes/IdentificationChannel/logs/log-default.txt";
    uint64_t N = (argv >= 3) ? atoi(argc[2]) : 10000;
    uint64_t n = (argv >= 4) ? atoi(argc[3]) : 0;
    uint64_t m = (argv >= 5) ? atoi(argc[4]) : 15;
    random_seed = (argv >= 6) ? argc[5] : "";
    double avg_error = 0;

    for (uint64_t i = 0; i < 10; i++)
    {
        avg_error += simulate(noiselessBSC, N, n, NoiselessBSC_ID);
    }
    cout << avg_error / m;
    *getOutputStream() << "End" << endl;
    getOutputStream()->close();
    return 0;
}
```

The `main` method takes some optional inputs

1. Address of a file for logging.
2. The parameter N, the number messages.
3. The parameter n, the block size – for 3-step algorithm is not important.
4. The parameter m, the number of simulations.
5. A random seed for initializing the random generator.

3.2 Random Prime Generation

The pseudocode of our prime generation algorithm is

Algorithm 2: pseudocode

input : positive integers m, s, k
output: A uniformly chosen odd prime less than or equal to m
for $i = 1 \to s$ **do**
  $n \leftarrow \{3, 5, \dots, m\}$
  if *Miller_Rabin*(n, k) *is PRIME* **then**
    return n
  end
end
return 23

Instead of returning $\perp$ when all of the randomly chosen numbers are COMPOSITE, it returns 23. Therefore, the probability of finding no prime number is

$$
\begin{aligned}
\mathbb{P}\{\perp\} &= \mathbb{P}\{M\_R(n_1, k) = \text{COMPOSITE}, \ldots, M\_R(n_s, k) = \text{COMPOSITE}\} \\
&= \prod_{i=1}^{s} \mathbb{P}\{M\_R(n_i, k) = \text{COMPOSITE}\} \\
&= \prod_{i=1}^{s} \mathbb{P}\{n_i \text{ is composite}\} \\
&= \prod_{i=1}^{s} (1 - \mathbb{P}\{n_i \text{ is prime}\}) \\
&\approx \prod_{i=1}^{s} \left(1 - \frac{1}{\ln m}\right) \\
&= \left(1 - \frac{1}{\ln m}\right)^{s}
\end{aligned}
$$

where we used the fact that $\frac{\pi(m)}{m} \approx \frac{1}{\ln m}$ asymptotically by prime number theorem [3]. From Theorem 31.39 of [4] and its following analysis, for moderate values of $s \approx 3$, the probability of error is negligible. That is, if the algorithm returns a number – not $\perp$– then it is most likely a prime. Then, the number of iteration to get a prime number is about

$$
\begin{aligned}
&\mathbb{P}\{\perp\} \leq \frac{1}{2} \\
\implies &s \lg \left(1 - \frac{1}{\ln m}\right) \leq -1 \\
\implies &s \geq \frac{-1}{\lg\left(1 - \frac{1}{\ln m}\right)} \\
\implies &s \geq \frac{-\ln 2}{\ln\left(1 - \frac{1}{\ln m}\right)}
\end{aligned}
$$

note that $\ln(1 - \frac{1}{x}) \approx -\frac{1}{x}$ for large enough x

$$
\implies s \geq \ln m \ln 2 = \ln^2 2 \lg m
$$

which means that s is in the order of number of bits of m.

References

[1] R. Ahlswede and G. Dueck. Identification via channels. *IEEE Transactions on Information Theory*, 35(1):15–29, 1989.

[2] R. Ahlswede and B. Verboven. On identification via multiway channels with feedback. *IEEE Transactions on Information Theory*, 37(6):1519–1526, 1991.

[3] Tom M. Apostol. *Introduction to Analytic Number Theory*. Springer New York, 1976.

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[5] S. Verdu and V.K. Wei. Explicit construction of optimal constant-weight codes for identification via channels. *IEEE Transactions on Information Theory*, 39(1):30–36, 1993.