UNIVERSITÀ DI PISA

First hands-on: Universal Hash Family

Algorithm Design (2021/2022)

Gabriele Pappalardo
Email: g.pappalardo4@studenti.unipi.it
Department of Computer Science

February 2022

1 Introduction

1.1 Terminologies and Definitions

This subsection defines terminology and definitions in order to fully understand the following essay.

With the symbol U we denote the set of possible keys. The sets $\mathbb{Z}_p^*$ and $\mathbb{Z}_p$ are defined as $\mathbb{Z}_p^* = \{0, \dots, p-1\}, \mathbb{Z}_p = \{1, \dots, p-1\}$ where p is a *prime* number.

Definition 1.1 (Universal Hash Family)**.** *A hash family $\mathcal{H}$ is said to be **universal** if given two different keys k_1 and k_2, and given a random hash function from the family, the probability of collision is less than $\frac{1}{m}$.*

$$\forall k_1, k_2 \in U, k_1 \neq k_2. \boldsymbol{Pr}_{h \in \mathcal{H}}(\{h(k_1) = h(k_2)\}) \leq \frac{1}{m}$$

1.2 Problem definition

The first hands-on requests to prove that given the following family of functions:

$$\mathcal{H} := \{h_{ab}(x) = ((ax + b) \bmod p) \bmod m \mid a \in \mathbb{Z}_p^*, b \in \mathbb{Z}_p\}$$

is *universal* with $m > 1$, $p \in [m+1, 2m)$ *prime*. That is, for any $k_1 \neq k_2$, it holds that:

$$|\mathcal{B}| = |\{h \in \mathcal{H} \mid h(k_1) = h(k_2)\}| = \frac{|\mathcal{H}|}{m}$$

Proof. Two different keys k_1, k_2, collides if and only if the hash function image is equal. So, given a function $h_{ab} \in \mathcal{H}$, and $k_1 \neq k_2 \in \mathbb{Z}_p$:

$$h_{ab}(k_1) = h_{ab}(k_2) \iff ((ak_1 + b) \bmod p) \bmod m = ((ak_2 + b) \bmod p) \bmod m$$

Consider first r and s, defined as:

$$r = (ak_1 + b) \bmod p$$
$$s = (ak_2 + b) \bmod p$$

If the keys k_1 and k_2 are different, we can see that $r \neq s$. To show this, subtract r from s:

$$r - s \equiv (ak_1 + b) - (ak_2 + b) \pmod p \iff r - s \equiv a(k_1 - k_2) \pmod p$$

In order to be equal, the values r and s should be congruent to 0 in mod$p$. We know from our hypothesis that $k_1, k_2 \in \mathbb{Z}_p \wedge k_1 \neq k_2$, therefore $k_1 - k_2 \not\equiv 0 \pmod{p}$. Since by definition $a \in \mathbb{Z}_p$ we can conclude that $a \not\equiv 0 \pmod{p}$, thus $r - s \not\equiv 0 \pmod{p}$.

Knowing that value r is different from s, we can derive that a and b are uniquely determined. As a matter of fact, we know:

$$\begin{cases} r \equiv ak_1 + b & (\text{mod } p) \\ s \equiv ak_2 + b & (\text{mod } p) \end{cases} \iff \begin{cases} b \equiv r - ak_1 & (\text{mod } p) \\ s \equiv ak_2 + (r - ak_1) & (\text{mod } p) \end{cases}$$

$$\begin{cases} b \equiv r - ak_1 & (\text{mod } p) \\ s - r \equiv ak_2 - ak_1 & (\text{mod } p) \end{cases} \iff \begin{cases} b \equiv r - ak_1 & (\text{mod } p) \\ s - r \equiv a(k_2 - k_1) & (\text{mod } p) \end{cases}$$

$$\iff \begin{cases} b \equiv r - ak_1 & (\text{mod } p) \\ a \equiv (k_2 - k_1)^{-1}(r - s) & (\text{mod } p) \end{cases}$$

Thus, there is one-to-one correspondence between the pairs (a, b), with $a \neq 0$, and the pair (r, s), with $r \neq s$. We have $p(p-1)$ possibilities to select the pair (a, b) and (r, s).

Therefore, the probability that keys k_1 and k_2 collide is equal to the probability that $r \equiv s \pmod{m}$. For a fixed r, the number to choose s, with $s \neq r, s \equiv r \pmod{m}$, from the $(p-1)$ possibilities, is at most $\frac{(p-1)}{m}$. So, the number of bad hash functions $h \in \mathcal{H}$ is equal to $p\frac{(p-1)}{m} = \frac{|H|}{m}$. Finally, we have:

$$\mathbf{Pr}(\{h \in \mathcal{H} \mid h(k_1) = h(k_2)\}) = \frac{\#\text{ bad choices of } h}{\#\text{ all choices of } h \in \mathcal{H}} = \frac{|\mathcal{B}|}{|\mathcal{H}|} = \frac{\frac{|\mathcal{H}|}{m}}{|\mathcal{H}|} = \frac{1}{m}$$

$\mathcal{H}$ is an *universal* hash family.

□

UNIVERSITÀ DI PISA

Second hands-on: Depth of a node in a Random Search Tree

Algorithm Design (2021/2022)

Gabriele Pappalardo
Email: g.pappalardo4@studenti.unipi.it
Department of Computer Science

March 2022

1 Introduction

A **random binary search tree** for a set S can be defined as follows: if S is empty, then the null tree is a random search tree; otherwise, choose uniformly at random a key $k \in S$: the random search tree is obtained by picking k as *root*, and the random search trees on $L = \{x \in S : x < k\}$ and $R = \{x \in S : x > k\}$ become, respectively, the left and right subtrees of the root k.

Consider the *Randomized Quick Sort* (RQS) discussed in class and analyzed with indicator variables, and observe that the random selection of the pivots follows the above process, thus producing a random search tree of n nodes.

- Using a variation of the analysis with indicator variables X_{ij}, prove that the expected depth of a node (i.e. the random variable representing the distance of the node from the root) is nearly $2 \log n$.
- Prove that the expected size of its subtree is nearly $2 \log n$ too, observing that it is a simple variation of the previous analysis.
- Prove that the probability that the depth of a node exceeds $c2 \log n$ is small for any given constant $c > 2$.

2 Solution

The hands-on's solution relies on the analysis we did during the lectures with randomized version of Quick-Sort. Let us recall for a moment how the algorithm works.

```
def randomizedPartition(A: [int], p: int, r: int) → int:
    i = random.randint(p, r)
    swap(A[r], A[i])
    return partition(A, p, r)

def randomizedQuickSort(A: [int], p: int, r: int):
    if p < r:
        q = randomizedPartition(A, p, r)
        randomizedQuickSort(A, p, q - 1)
        randomizedQuickSort(A, q + 1, r)
```

Listing 1: 'Randomized QuickSort'

At the first sight of the code it seems that an RBST and the RQS do have nothing in common. Actually, the way how the RBST is build match 1-to-1 to the recursion tree created by the RQS.

When we select a random key $k \in S$ during the building of a RBST (the root of our tree), it corresponds to select a pivot in RQS. And the partition call allows us to create two subtrees (L, R) that will have elements less/greater than pivot, and then we call the tree build construction algorithm onto the subtrees.

2.1 Expected depth of a node

We define the elements of a random binary search tree with $\forall i \in [1, n].x_i \in S$. Let X_{ij} an indicator random variable defined as follows:

$$X_{ij} = \begin{cases} 1 & x_i \text{ is an ancestor of } x_j \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

The depth of a node can be expressed as the sum of all the indicator random variables. Let D_i be the depth of a node x_i in a random binary search tree, defined as $D_i = \sum_{j=1}^{n} X_{ij}$. We can express the expected depth as:

$$E[D_i] = E\left[\sum_{j=1}^{n} X_{ij}\right] = \sum_{j=1}^{n} E[X_{ij}] = \sum_{j=1}^{n} \Pr(\{X_{ij} = 1\}) \tag{2}$$

The probability of $X_{ij} = 1$ can be computed using the analysis we did during lectures with RQS. Since the RQS maps over the RBST we can say that $X_{ij} = 1$ when x_i is a pivot (select as root for a tree) or x_j is a pivot.

The expected depth of a node can be computed using this fact:

$$E[D_i] = \sum_{j=0}^{n} \Pr(\{X_{ij} = 1\}) =$$

$$\sum_{j=0}^{i} \frac{1}{i-j+1} + \sum_{j=i+1}^{n} \frac{1}{j-i+1} =$$

$$\sum_{k=1}^{i+1} \frac{1}{k} + \sum_{k=2}^{n-i+1} \frac{1}{k} \leq \log n + \log n = 2 \log n$$

2.2 Expected size of a subtree

The analysis for the expected size of a subtree is similar to the previous one. We change the meaning of our indicator random variable as:

$$X_{ij} = \begin{cases} 1 & x_i \text{ is an descendant of } x_j \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

2.3 Probability for the depth of a node

Knowing that the expected depth of a node $E[D_i] \leq 2 \log n$, we can prove that probability that the depth of a node exceeds $c2 \log n$ is small for any given constant $c \geq 2$.

$$\mathbf{Pr}(\{D_i \geq c2 \log n\}) \leq e^{-\frac{\lambda^2}{2\mu+\lambda}} \tag{4}$$

Knowing that $\mu = E[D_i] \leq 2 \log n$ we can derive λ:

$$\mu + \lambda = c2\log n$$
$$E[D_i] + \lambda = c2\log n$$
$$\lambda = c2\log n - E[D_i] \leq c2\log n - 2\log n = \log n(2c-2)$$

Therefore, the Chernoff Bound becomes:

$$\begin{aligned}
\mathbf{Pr}(\{D_i \geq c2\log n\}) &\leq e^{-\frac{(\log n(2c-2))^2}{2(2\log n)+\log n(2c-2)}} \\
&= e^{-\frac{(\log n)^2(2c-2)^2}{2(2\log n)+\log n(2c-2)}} \\
&= e^{-\frac{(\log n)^2(2c-2)^2}{\log n(2c+2)}} \\
&= e^{-\frac{\log n(2c-2)^2}{(2c+2)}} \\
&= n^{-\frac{(2c-2)^2}{(2c+2)}} \\
&= n^{-\frac{4c^2+4-8c}{2c+2}} \\
&= n^{-\frac{2c^2+2-4c}{c+1}} \\
&= \frac{1}{n^{\frac{2c^2+2-4c}{c+1}}}
\end{aligned}$$

At the end, the probability is small for any constant $c > 2$.

UNIVERSITÀ DI PISA

Third hands-on: Karp-Rabin fingerprinting on strings

Algorithm Design (2021/2022)

Gabriele Pappalardo
Email: g.pappalardo4@studenti.unipi.it
Department of Computer Science

March 2022

1 Introduction

Given a string $S \equiv S[0 \dots n-1]$, and two positions $0 \leq i < j \leq n-1$, the longest common extension `lce(i, j)` is the length of the maximal run of matching characters from those positions, namely: if $S[i] \neq S[j]$ then `lce(i, j) = 0`; otherwise, `lce(i, j)` $= \max\{l \geq 1 : S[i \dots i+l-1] = S[j \dots j+l-1]\}$. For example, if $S =$ "*abracadabra*", then `lce(1, 2) = 0`, `lce(0, 3) = 1`, and `lce(0, 7) = 4`. Given S in advance for preprocessing, build a data structure for S based on the **Karp-Rabin** fingerprinting, in $O(n \log n)$ time, so that it supports subsequent online queries of the following two types:

- `lce(i,j)`: it computes the longest common extension at positions i and j in $O(\log n)$ time.
- `equal(i,j,l)`: it checks if $S[i \dots i+l-1] = S[j \dots j+l-1]$ in constant time.

Analyze the cost and the error probability. The space occupied by the data structure can be $O(n \log n)$, but it is possible to use $O(n)$ space.

2 Solution

2.1 Baseline solution

Before giving the requested solution, we start from the baseline one. The problem can be solved in $O(n^2)$ space with $O(1)$ time to compute the `equal(i, j, l)` and $O(\log n)$ time for the `lce(i, j)`. The baseline solution makes use of a matrix $M \in \mathbb{Z}_p^{n \times n}$ where each cell contains the Karp-Rabin fingerprint F_{ij}.

$$\begin{bmatrix} F_{00} & F_{01} & F_{02} & F_{03} & \dots & F_{0n} \\ \emptyset & F_{11} & F_{12} & F_{13} & \dots & F_{1n} \\ \emptyset & \emptyset & F_{22} & F_{23} & \dots & F_{2n} \\ \emptyset & \emptyset & \emptyset & F_{33} & \dots & F_{3n} \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & F_{nn} \end{bmatrix} \tag{1}$$

Figure 1: Where F_{ij} is the Karp-Rabin of the substring $S[i \dots j]$

To compute the `equal(i, j, l)` we will use the function `equal(M, i, j, l)` shown in the pseudo-Python code list 1.

```
def equal(M, i, j, l):

    # n is the length of the string
    if (i + l - 1) ⩾ n or (j + l - 1) ⩾ n:
            return False

    fi = M[i][i + l - 1]
    fj = M[j][j + l - 1]

    return (fi == fj)
```

Listing 1: "Function to compute the equality."

To compute the `lce(i, j, l)` we will use the function `lce(M, i, j, l)` show in the pseudo-Python code list 2.

```
def lce(M, i, j, l):
    if l > 0:
        if equal(M, i, j, l):
            return l + lce(M, i + l, j + l, l)
        else:
            return lce(M, i, j, l / 2)
    else:
        return 0
```

Listing 2: "Function to compute the longest common extension."

These two algorithms respect the requested running time.

2.2 Improving space

To reduce the space used by the matrix M, we can use just one row, i.e. the first one. The Karp-Rabin fingerprint of a sub-string $S[i \dots j], i < j$ is computed using the equation 2.

$$F_{ij} = h(S) = \sum_{k=i}^{j} \text{ord}(S_k)\sigma^{k-i} \mod p \tag{2}$$

Where p is a prime number and $\text{ord} : \Sigma \to \mathbb{N}$ is a function converting a character belonging to the string alphabet to a natural number (e.g. the `String.fromCharCode( ... )` function in JavaScript). The hash function can be viewed as a polynomial equation in $\mathbb{Z}[x]_p$ field, as shown in Equation 3 (using the Horner's method).

$$\begin{aligned} F_{0n} = h(S) &= \text{ord}(S_0) + \text{ord}(S_1)\sigma + \text{ord}(S_2)\sigma^2 + \cdots + \text{ord}(S_{n-1})\sigma^{n-1} \mod p & (3)\\ &= \text{ord}(S_0) + \sigma(\text{ord}(S_1) + \sigma(\text{ord}(S_2) + \cdots + \sigma\text{ord}(S_{n-1})))) \mod p & (4) \end{aligned}$$

We can exploit the properties of this rolling hash function to compute the missing fingerprints that we removed from the matrix. In fact, given two indices, i, j we can find its fingerprint F_{ij}, using only the first row of the matrix, in a mathematical way as shown in Equation 5.

$$F_{ij} = ((F_{0j} - F_{0(i-1)})/\sigma^i) \mod p \tag{5}$$

We show a numerical example. Suppose we have a string S (where $|S| > 6$), we are going to find the fingerprint F_{36} using the statement above. We are assuming $\text{ord}(S_k) = S_k$ in order to ease computations.

$$\begin{aligned} F_{36} &= (F_{06} - F_{02})/\sigma^3 \mod p & (6)\\ F_{36} &= S_3 + S_4\sigma + S_5\sigma^2 + S_6\sigma^3 \mod p & (7) \end{aligned}$$

We know already, F_{06} and F_{02} which are computed as shown below.

$$F_{06} = S_0 + S_1\sigma + S_2\sigma^2 + S_3\sigma^3 + S_4\sigma^4 + S_5\sigma^5 + S_6\sigma^6 \mod p \tag{8}$$

$$F_{02} = S_0 + S_1\sigma + S_2\sigma^2 \mod p \tag{9}$$

$$F_{06} - F_{02} = S_3\sigma^3 + S_4\sigma^4 + S_5\sigma^5 + S_6\sigma^6 \mod p \tag{10}$$

$$F_{36} = (F_{06} - F_{02})/\sigma^3 \mod p = S_3 + S_4\sigma + S_5\sigma^2 + S_6\sigma^3 \mod p \tag{11}$$

2.3 Error analysis

The error probability for the equal function is $\frac{1}{n^c}$. For the LCE function we have an error probability of $\frac{1}{n^c}\log n$.

UNIVERSITÀ DI PISA

Fourth hands-on: Tweets

Algorithm Design (2021/2022)

Gabriele Pappalardo
Email: g.pappalardo4@studenti.unipi.it
Department of Computer Science

March 2022

1 Introduction

Given a stream of Tweets collected using Twitter, answer to these questions:

- Count the percentage of happy users in the different moments of the day (morning, afternoon, evening, night). Discuss what you find if compute also the percentage of unhappy users. Do the two percentages sum to 100%? Why?
- Spell the 30 favorite words of happy users.
- Find the number of distinct words used by happy users. How could we exclude words repeated only once?
- Decide if, in general, happy messages are longer or shorter than unhappy messages.

2 Solution

2.1 First Problem

The solution for the first point uses one linear counter for each moment of the day. Thus, we are going to have a total of four counters for the happy users, and other four for the unhappy ones. If we sum the obtained percentages we could obtain a 100% under certain conditions, that is, a user should post only a tweet. Otherwise, we could never obtain the 100% since the mood of a user could change during the day (see the user *@missbrandii*, he/she has three negative tweets and a positive one inside the `sample.csv` dataset.).

2.2 Second Problem

In order to spell the 30 favorite words of happy users we can use the *space saving algorithm*, seen during the lectures, and query the table at the end for the first 30 words.

2.3 Third Problem

To find the distinct words used by happy users, we thought to use a LogLog counter (combined with a Bloom Filter for the second question). The LogLog counter is suitable to count distinct elements. Moreover, to understand if a word is repeated only once we use a Bloom Filter in the following way:

1. Set the bit of the hashed value to 1;
2. If the bit was set to 1, then it means we already saw that word, and therefore we count it using the LogLog counter.

A pseudo Python code solution is shown Listing 1.

```
def count_distinct_words(s: Stream) → Int:

    counter = HyperLogLogCounter( ... )
    filter = BloomFilter( ... )

    word = s.next()

    while word is not None:
        if not filter.is_set(word):
            filter.set(word)
        else:
            counter.add(word)
        word = s.next()

    return counter.count()
```

Listing 1: "Pseudo-Code for the third point"

2.4 Fourth Problem

The fourth point can be solved using several approaches, we used the *median* as indicator. In the `mini.csv` and `sample.csv` datasets, the negatives messages are longer than the positives ones. Instead, using the *mean* as indicator, in the `mini.csv` dataset, positive messages are longer.

UNIVERSITÀ DI PISA

Fifth hands-on: Bloom Filters

Algorithm Design (2021/2022)

Gabriele Pappalardo
Email: g.pappalardo4@studenti.unipi.it
Department of Computer Science

March 2022

1 Introduction

The problem is composed in two parts:

1. Consider the Bloom Filters where a single random universal hash random function $h : U \to [m]$ is employed for a set $S \subseteq U$ of keys, where U is the universe of keys. Consider its binary array B of m bits. Suppose that $m \geq c|S|$, for some constant $c > 1$, and that both c and $|S|$ are unknown to us. Estimate the expected number of 1s in B under a uniform choice at random of $h \in \mathscr{H}$. Is this related to $|S|$? Can we use it to estimate $|S|$?

2. Consider B and its rank function: show how to use extra $O(m)$ bits to store a space-efficient data structure that returns, for any given i, the following answer in constant time: $\text{rank}(i) = \#1s \in B[1..i]$

   Hint: Easy to solve in extra $O(m \log m)$ bits. To get $O(m)$ bits, use prefix sums on B, and sample them. Use a lookup table for pieces of B between any two consecutive samples.

2 Solution

2.1 Bloom Filter

A Bloom Filter is a probabilistic data structure, invented in 1970 by Burton Bloom, that allows to check wheter an element x belongs to a set S without storing it. A filter works with $k > 0$ hash functions $h_i : U \to [m]$ belonging to a universal hash family.

2.2 Estimate expected number of bits set

The first point of the hands on asks us to estimate the expected number of bits set in a Bloom Filter using only one hash function. We start defining a new random indicator variable X_i such that:

$$X_i = \begin{cases} 1 & \text{if } B_i = 1 \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

Therefore, we can be build a new random variable $Y = \sum_{i=0}^{m-1} X_i$ to estimate the number of bits set to one. We define $n = |S|$, as the unknown value to find.

$$E[\![Y]\!] = E\left[\!\!\left[\sum_{i=0}^{m-1} X_i \right]\!\!\right] = \sum_{i=0}^{m-1} E[\![X_i]\!]$$

$$= \sum_{i=0}^{m-1} 1 - \left(1 - \frac{1}{m}\right)^n = (m-1)\left(1 - \left(1 - \frac{1}{m}\right)^n\right)$$

$$= m\left(1 - \left(1 - \frac{1}{m}\right)^n\right)$$

$$= m - m\left(1 - \frac{1}{m}\right)^n \simeq m\left(1 - e^{-\frac{n}{m}}\right)$$

Knowing that $\mu = E[\![Y]\!] \simeq m(1 - e^{-\frac{n}{m}})$, we can solve the equation to find the cardinality of the original set S, which is equal to n.

$$\begin{aligned}
&\mu \simeq m(1 - e^{-\frac{n}{m}}) \\
\iff &\frac{\mu}{m} = 1 - e^{-\frac{n}{m}} \\
\iff &\frac{\mu}{m} - 1 = -e^{-\frac{n}{m}} \\
\iff &1 - \frac{\mu}{m} = e^{-\frac{n}{m}} \\
\iff &\ln(1 - \frac{\mu}{m}) = -\frac{n}{m} \\
\iff &m \ln(1 - \frac{\mu}{m}) = -n \\
\iff &n = -m \ln(1 - \frac{\mu}{m})
\end{aligned}$$

Thus, the number of elements in S is $n = -m \ln(1 - \frac{\mu}{m})$.

2.3 Rank Computation

The second point of the hands-on asks us to compute the rank of the bit array B, using only $O(m)$ bits of space. To reach the requested space complexity, we start off by the baseline solution, the one gave by the hint.

2.3.1 Baseline Solution

Prefix sums allow us to answer the rank function in constant time, in fact, we can build a new array P, that contains in each position i the number of ones up to i.

$$P_i = \sum_{j=0}^{i} B_j \tag{2}$$

Since the maximum prefix sum can be $|B| = m$ (when all the bits are set to 1), we need $O(\log m)$ bits to store each sum, bringing us to use $O(m \log m)$ space to store the entire array of prefix sums.

2.3.2 Requested Solution

To achieve the requested space $O(m)$ we sample the prefix sums array P, and we create a new lookup table T, described below.

First, we have to determine how many samples of the prefix sums we do need. We are going to use $\frac{2m}{\log m}$ samples of prefix sums. We will save these samples in an array called P of size $|P| = \frac{2m}{\log m}$.

Second, how do we build T? When we are asking for the rank of a position which has not a corresponding sample, we have to use the nearest sample to the position, plus the number of ones between the sample and the position itself.

To do so, we define $L = \frac{|B|}{\frac{2m}{\log m}} = \frac{\log m}{2}$ between two samples. We can split the array B in L parts. Splitting the bit array leave us with $|B|/L$ portions, that we can use to index our lookup table T. How many strings of bits can be represented using L bits? That is, how many rows T will have? Exactly: $\#\text{rows} = 2^L$.

$$\#\text{rows} = 2^L = 2^{\frac{\log m}{2}} = (2^{\log m})^{\frac{1}{2}} = \sqrt{m} \tag{3}$$

We need $\sqrt{m}$ rows for T. How many columns will have the table T? $\#\text{columns} = L = \frac{\log m}{2}$. Therefore, our lookup table T will have a size of $\sqrt{m}\frac{\log m}{2}$. Assuming we have a bit array of size $m = 64$, then the lookup table will look like as shown below.

$$\begin{matrix} (0 & 0 & 0)_2 \\ (0 & 0 & 1)_2 \\ (0 & 1 & 0)_2 \\ (0 & 1 & 1)_2 \\ (1 & 0 & 0)_2 \\ (1 & 0 & 1)_2 \\ (1 & 1 & 0)_2 \\ (1 & 1 & 1)_2 \end{matrix} \quad T = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 1 \\ 0 & 1 & 2 \\ 1 & 1 & 1 \\ 1 & 1 & 2 \\ 1 & 2 & 2 \\ 1 & 2 & 3 \end{bmatrix} \tag{4}$$

Our table will contain the partial sums of each of the possible portions as shown in Equation 4.
At the end, to compute the rank function we are going to need both prefix sums samples and the lookup table T.

2.3.3 Space Complexity

We are keeping $\frac{2m}{\log m}$ samples of prefix sums, which takes up to $O(\log m)$ bits, therefore $O(m)$. Plus, for the lookup table T we are using $\sqrt{m}$ rows and $\frac{\log m}{2}$ columns, and for each cell we need $O(\log \log m)$ bits, thus $O(\sqrt{m} \log m (\log \log m))$ bits.

```
uint64_t rank(uint64_t B, uint64_t j) {
    // Assuming we have the sample prefix sums array P
    // and the lookup table T.

    uint64_t m = 64;
    uint64_t L = log2(m) / 2;

    // Row and Column indecies for the Lookup Table T
    uint64_t r = get_row(B, j);
    uint64_t c = j % L;

    return P[(j / L) - 1] + T[r][c];
}

uint64_t get_row(uint64_t B, uint64_t i) {
    // Shift and mask to get the row index
    return (B >> (0x40 - (0x3 * (i + 1)))) & 0x7;
}
```

Listing 1: 'Rank function implemented using a bit array of 64 elements.'

At the end we have $O(m) + O(\sqrt{m} \log m (\log \log m)) = O(m)$ total bits.

UNIVERSITÀ DI PISA

Sixth hands-on: Most frequent item in a stream

Algorithm Design (2021/2022)

Gabriele Pappalardo
Email: g.pappalardo4@studenti.unipi.it
Department of Computer Science

March 2022

1 Introduction

Suppose to have a stream of n items, so that one of them occurs $> n/2$ times in the stream. Also, the main memory is limited to keeping just $O(1)$ items and their counters (where the knowledge of the value of n is not actually required). Show how to find deterministically the most frequent item in this scenario.

Hint: the problem cannot be solved deterministically if the most frequent item occurs $\leq n/2$ times, so the fact that the frequency is $> n/2$ should be exploited.

2 Solution

A *streaming algorithm* is an algorithm that receives its input as *a stream* of data, and that proceeds by making only one pass through the data.

Assuming we have a stream of n items, it is possible to find the element with the highest frequency, knowing that it occurs $n/2$ times. The proposed solution makes use of *Boyer-Moore majority vote algorithm*, that is implemented in the Listing 1.

```
def boyer_moore_majority_vote(stream):

  m = None
  count = 0

  for i in range(len(stream)):
    if count == 0:
      m = stream[i]
      count = 1
    elif m == stream[i]:
      count += 1
    else:
      count -= 1

  return m
```

Listing 1: "Boyer-Moore majority vote algorithm"

The algorithm uses just two local variables: one for the element and the other for the counter. Thus, the algorithm *space complexity* is $O(1)$. The algorithm will always find the element with the higher frequency while it is repeated at least $n/2$ times. When analyzing the stream the following conditions are evaluated:

1. if the counter is set to zero then we mark the element, denoted by the variable m, equal to i-th stream value, as the most frequent element;

2. if the i-th stream element is equal to the most frequent one, then we increase the counter by one;

3. otherwise, if none of the above conditions have been met, then the i-th element is different from the most frequent one. Therefore, we decrease the counter.

If m is the truly majority element, the counter will be always greater or equal than 1, because the increments will be more than the decrements.

UNIVERSITÀ DI PISA

Seventh hands-on: Deterministic Data Stream

Algorithm Design (2021/2022)

Gabriele Pappalardo
Email: g.pappalardo4@studenti.unipi.it
Department of Computer Science

April 2022

1 Introduction

Consider a stream of n items, where items can appear more than once. The problem is to find the most frequently appearing item in the stream (where ties are broken arbitrarily if more than one item satisfies the latter). For any fixed integer $k \geq 1$, suppose that only k items and their counters can be stored, one item per memory word; namely, only $O(k)$ memory words of space are allowed.

Show that the problem cannot be solved deterministically under the following rules: any algorithm can only use these $O(k)$ memory words, and read the next item of the stream (one item at a time). You, the adversary, have access to all the stream, and the content of these $O(k)$ memory words: you cannot change these words and the past items, namely, the items already read, but you can change the future, namely, the next item to be read. Since any algorithm must be correct for any input, you can use any amount of streams and as many distinct items as you want.

Hints:

1. This is "classical" adversarial argument based on the fact that any deterministic algorithm A using $O(k)$ memory words gives the wrong answer for a suitable stream chosen by the adversary.
2. The stream to choose as an adversary is taken from a candidate set sufficiently large: given $O(k)$ memory works, let $f(k)$ denote the maximum number of possible situations that algorithm A can discriminate. Create a set of C candidate streams, where $|C| > f(k)$: in this way there are two streams $S1$ and $S2$ that A cannot distinguish, by the pigeon principle.

2 Solution

To begin with, let us define a new function called moreFreq such that, given an alphabet of characters Σ we have:

$$\text{moreFreq} : \Sigma^* \to \Sigma$$
$$\text{moreFreq}(S) = \text{most frequent item } c \text{ in the stream } S$$

Where Σ^* it the *Kleene* closure of the alphabet used in automata theory.

As suggested by the hints, our main goal is to find two streams S_1 and S_2 such that $\text{mostFreq}(S_1) \neq \text{mostFreq}(S_2)$, but for the algorithm A we have $A(S_1) = A(S_2)$. To do so, we take an universe of elements, denoted with U, and its subsets $\Sigma_i \subseteq U$ such that they have cardinality $|\Sigma_i| = \frac{|U|}{2} + 1$ and $\forall i, j, i \neq j$ they satisfy the following properties:

1. $|\Sigma_i \cap \Sigma_j| \geq 1$: have at least one element in common;
2. $|\Sigma_i \setminus \Sigma_j| \geq 1$: at least one character differ in both sets.

How many subsets satisfy the requesting criteria? We use the binomial coefficient to discover that are:

$$\binom{|U|}{\frac{|U|}{2} + 1} \simeq 2^{|U|} = \eta \tag{1}$$

Note that if the cardinality of U is huge then the number of sets will be exponentially large!

From these subsets we can build the set of candidate streams C such that $|C| > f(k)$. Given an alphabet Σ_i, the Equation 2 shows how to build a possible stream. Be aware: the streams contain each character coming from its alphabet repeated only once (this fact will be exploited later).

$$S_i = s_1^{(i)} s_2^{(i)} s_3^{(i)} \dots s_{\frac{|U|}{2}+1}^{(i)}, \forall k, j \in [|\Sigma_i|], k \neq j . s_k \neq s_j \tag{2}$$

But, how big is the set of candidate streams? We have η alphabets which from each one we build a string as described above, and we put it in C. Therefore, the set will have a cardinality of:

$$|C| = \eta = 2^{|U|}$$

Since we are requiring that $|C| > f(k)$ we can derive that:

$$|C| > f(k) \iff 2^{|U|} > f(k) \iff |U| > \log f(k)$$

Therefore, $\exists S_i \in \Sigma_i^*, S_j \in \Sigma_j^* (S_i, S_j \in C)$ such that $S_i \neq S_j$ that those streams are indistinguishable for the algorithm A, due to the pigeonhole principle (since $|C| > f(k)$).

Now, if we pick an element $x \in \Sigma_i \setminus \Sigma_j$ and $y \in \Sigma_j \setminus \Sigma_i$, such that $x \neq y$ [1], and we concatenate them to the streams S_i, S_j obtaining two new streams $S_i xy$ and $S_j xy$. Giving these two streams to A we have $A(S_i xy) = A(S_j xy)$ because S_i and S_j are indistinguishable for A. But:

$$\text{mostFreq}(S_i xy) \neq \text{mostFreq}(S_j xy)$$

Thus, the algorithm A fails.

[1] Since we are requesting the properties (1) and (2) we can take such x and y.

UNIVERSITÀ DI PISA

Eight hands-on: Count-min sketch: range queries

Algorithm Design (2021/2022)

Gabriele Pappalardo
Email: g.pappalardo4@studenti.unipi.it
Department of Computer Science

April 2022

1 Introduction

Consider the counters $F[i]$ for $1 \leq i \leq n$, where n is the number of items in the stream of any length. At any time, we know that $||F||$ is the total number of items (with repetitions) seen so far, where each $F[i]$ contains how many times the item i has been so far. We saw that CM-sketches provide a FPTAS $F'[i]$ such that $F[i] \leq F'[i] \leq F[i] + \varepsilon||F||$, where the latter inequality holds with probability at least $1 - \delta$.

Consider now a range query (a, b), where we want $F_{ab} = \sum_{a \leq i \leq b} F[i]$. Show how to adapt CM-sketch so that a FPTAS F'_{ab} is provided:

- Baseline is $\sum_{a \leq i \leq b} F'[i]$, but this has drawbacks as both time and error grows with $b - a + 1$.
- Consider how to maintain counters for just the sums when $b - a + 1$ is any power of 2 (less or equal to n):
  - Can we now answer quickly also when $b - a + 1$ is not a power of two?
  - Can we reduce the number of these power-of-2 intervals from $n \log n$ to $2n$?
  - Can we bound the error with a certain probability? *Suggestion*: it does not suffice to say that it is at most δ the probability of error of each individual counter; while each counter is still the actual wanted value plus the residual as before, it is better to consider the sum V of these wanted values and the sum X of these residuals, and apply Markov's inequality to V and X rather than on the individual counters.

2 Solution

2.1 Baseline Solution

The baseline solution is pretty straightforward to implement: we can use a `for` loop starting from the element a up to b and sum all the counters. Since each counter has some error, if we sum up all the counter in a large range, then also the final error will grow linearly as large as the range, that is $b - a + 1$.

```
def range(F: CountMinSketch, a: int, b: int): int:
    sum = 0
    for i in range(a, b + 1):
        sum += F[i]
    return sum
```

Listing 1: 'Range query for integer values'

2.2 Requested Solution

We can improve the baseline solution using ranges of powers of two. In fact, any range (a, b) can be expressed as disjoint union of length with the powers of two, e.g., if we have the range $(4, 10)$:

$$(4, 10) = (4) \cup (5, 8) \cup (9, 10)$$

We can state the following fact that will allow us to compute all the possible range queries.

Fact 1 (Dyadic Ranges)**.** *Any range in the interval from* 1 *to* n *is expressible as the disjoint union of* $2 \log n$ *intervals in the set of dyadic ranges.*

Therefore, given a universe U we can build a collection of *dyadic ranges*, and we will need at most $2n$ of them. Having these ranges, we can bind the error of the range query up to $\log n$, thus a logarithmic error.

$$F_{ab} \leq \tilde{F}_{ab} \leq F_{ab} + 2\varepsilon \log n ||F||$$

This can be achieved using $\log n$ Count-Min Sketches. We build a "logic" binary tree where each range is split in two parts. Starting from the root we represent the range $(1, n)$ and going down we will have $(1, n/2)$ and $(n/2 + 1)$. We repeat the splitting process until we are not able to do so anymore, that is, when we obtain n ranges of the form $(1, 1), (2, 2), (3, 3), \ldots, (n-1, n-1), (n, n)$. In each level of this "logic" binary tree we add a new Count-Min Sketch, having in total $\log n$ sketches.

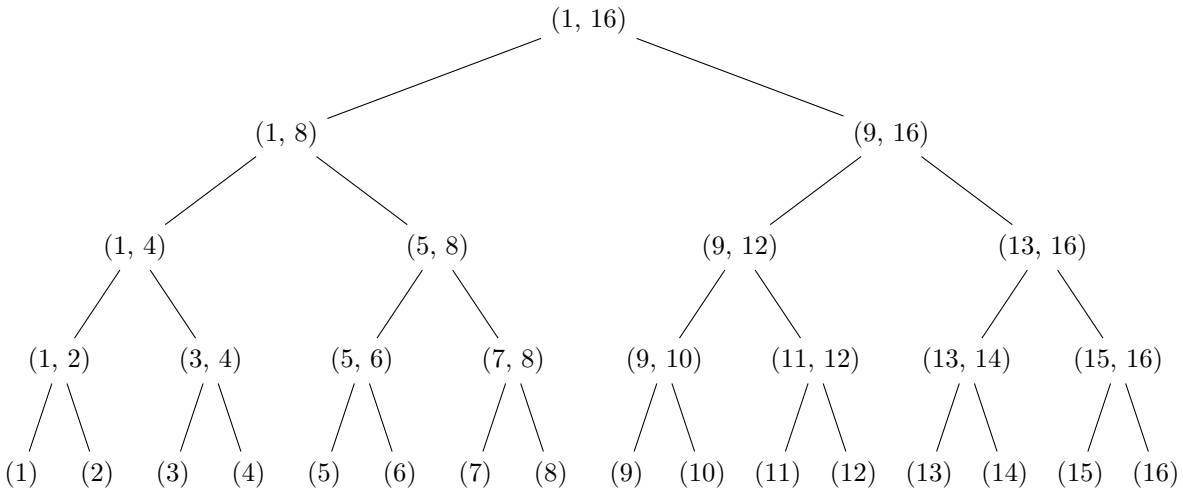

Figure 1: Generated ranges using $n = 16$.

When we have to update a counter, we traverse the tree updating the counters for each traversed range. For instance, if we have to update the counter for the element 7 in Figure 1, we are going to update the counters in each range containing the element, that are: $(1, 16), (1, 8), (5, 8), (7, 8), (7)$. The `update` operation takes $O(\log n)$ time, the pseudo-Python code is shown in Listing below.

```
def update(x: int) → None:
    # The first range to update is the root
    range = (1, n)
    level = 1
    # When the range has size one we stop
    while range ≠ (x, x):
        # Update the CMS in the current level
        update(CMSS[level], range)
        # Go a level below in the logical tree
        level += 1
        # Compute the new range to update
        if x in (range[0], int(range[1]/2)):
            range = (range[0], int(range[1] / 2))
        else:
            range = (int(range[0] / 2) + 1, range[1])
    return
```

The `query` operation relies on **Fact** 1, also this computation takes $O(\log n)$ time. We have to select each dyadic range needed to build the original range.

```
def query(r: (int, int)) → int:
    sum = 0
    # Assuming we have a function that computes
    # dyadic ranges.
    ranges = dyadicRanges(r)
    # Query each CMS in the respective level with
    # the computed ranges
    for (level, range) in ranges:
        sum += normal_query(CMSS[level], range)
    return 0
```

These algorithms work also with one Count-Min Sketch, the universe of the counters will be the dyadic ranges.

2.3 Bounding the failure probability

The error analysis will require some definitions to work with. First, we are going to define the set of dyadic ranges as:

$$D = \{(1,n),(1,n/2),(n/2+1,n),\ldots,(1,1),(2,2),\ldots,(n,n)\}, |D| = 2n$$

The analysis makes use of $\log n$ Count-Min Sketches as stated before, and it is restricted to the ranges contained in the set D. Furthermore, we introduce the function:

$$\mathrm{dy} : \mathscr{P}(\mathbb{N} \times \mathbb{N}) \to \mathscr{P}(D)$$

which computes the dyadic ranges of a given range, as described in the previous section. For any range r given to the function dy, thanks to the **Fact** 1, we have that:

$$\forall r \in \mathscr{P}(\mathbb{N} \times \mathbb{N}).|\mathrm{dy}(r)| \leq 2\log n$$

We express the counter F_i where the range $i = (a,b) \in D$. The approximate counter $\tilde{F}_i$ is obtained as follows in Equation 1.

$$\tilde{F}_i = F_i + X_i \tag{1}$$

The random variable X_i is a value representing the "garbage" (according to the definition given during the lectures). We would like to know the expected value of this random variable. To do so, we define a new indicator random variable, as seen in class:

$$I_{j,i,k} = \begin{cases} 1 & h_j(i) = h_j(k) \\ 0 & \text{othwerwise} \end{cases} \quad i,k \in D, j \in [d]$$

Therefore, when the j-th hash function has a collision with different ranges in D the variable will be set to 1. Fixed a $j \in [d]$ and given an $i \in D$ we can define $X_i^{(j)}$ as:

$$X_i^{(j)} = \sum_{k \in \mathrm{dy}(i)} I_{j,i,k} \cdot F_k$$

We can compute its expectation as follows:

$$
\begin{aligned}
E\Big[X_i^{(j)}\Big] &= E\Big[\sum_{k\in \mathrm{dy}(i)} I_{j,i,k}F_k\Big] \\
&= \sum_{k\in \mathrm{dy}(i)} E[I_{j,i,k}\cdot F_k] \\
&= \sum_{k\in \mathrm{dy}(i)} \Pr(\{I_{j,i,k}=1\})F_k \\
&= \sum_{k\in \mathrm{dy}(i)} \frac{\varepsilon}{e}\cdot F_k \\
&= \frac{\varepsilon}{e}\sum_{k\in \mathrm{dy}(i)} F_k \\
&\leq \frac{\varepsilon}{e} 2\log n||F||
\end{aligned}
$$

Knowing that $E\big[X_i^{(j)}\big] \leq 2\log n\frac{\varepsilon}{e}||F||$, we can bind the error probability in a single row using the Markov's inequality:

$$
\begin{gathered}
\Pr(\{\tilde{F}_i \geq F_i + \varepsilon 2\log n||F||\}) \leq \delta \iff \\
\Pr\Bigg(\Big\{F_i + X_i \geq F_i + \varepsilon 2\log n||F||\Big\}\Bigg) \leq \delta \iff \\
\Pr\Bigg(\Big\{X_i \geq 2\varepsilon\log n||F||\Big\}\Bigg) \leq \frac{E\big[X_i\big]}{\varepsilon 2\log n||F||} = \frac{2\log n\frac{\varepsilon}{e}||F||}{\varepsilon 2\log n||F||} = \frac{1}{e}
\end{gathered}
$$

Since we have d row, where each hash function is independent, we will have:

$$
\prod_{j\in[d]} \Pr\Bigg(\Big\{X_i^{(j)} \geq 2\varepsilon\log n||F||\Big\}\Bigg) \leq \left(\frac{1}{e}\right)^d = \delta
$$

UNIVERSITÀ DI PISA

Ninth hands-on: Game Theory I

Algorithm Design (2021/2022)

Gabriele Pappalardo
Email: g.pappalardo4@studenti.unipi.it
Department of Computer Science

April 2022

1 Introduction

1.1 Problem 1: Diabolik

After Diabolik's capture, Inspector Ginko is forced to free him because the judge, scared of Eva Kant's revenge, has acquitted him with an excuse. Outraged by the incident, the mayor of Clerville decides to introduce a new legislation to make judges personally liable for their mistakes. The new legislation allows the accused to sue the judge and have him punished in case of error. Consulted on the subject, Ginko is perplexed and decides to ask you to provide him with a formal demonstration of the correctness/incorrectness of this law.

1.2 Problem 2: Investment Agency

An investment agency wants to collect a certain amount of money for a project. Aimed at convincing all the members of a group of N people to contribute to the fund, it proposes the following contract: each member can freely decide either to contribute with 100 euros or not to contribute (retaining money on its own wallet). Independently on this choice, after one year, the fund will be rewarded with an interest of 50% and uniformly redistributed among all the N members of the group. Describe the game and find the Nash equilibrium.

2 Solutions

2.1 Diabolik

The main game's *players* are the *thief* (Diabolik), the *judge* and a third one, the *public opinion*. The thief has two strategies available, ha can sue or not sue the judge, having so $S_{\text{Diabolik}} = \{\text{Sue}, \text{Not Sue}\}$. In the other hand, the judge can *absolve* or *condemn* the defendant, therefore, his stategy set is the following one $S_{\text{Judge}} = \{\text{Absolve}, \text{Condemn}\}$.

The game can be modelled according to two cases:

1. Diabolik is guilty.
2. Diabolik is **not** guilty.

		Guilty Diabolik	
		Sue	Not Sue
Judge	Condemn	$(-1,-1)$	$(0,-1)$
	Absolve	$(-2,-1)$	$(0,0)$

Table 1: Payoff table when Diabolik is guilty.

2.2 1st Case: Diabolik is guilty

In the case where Diabolik is guilty we have the following payoff matrix (the Table 2.2).
In the Table 2.2 we have to consider the *public opinion* as well. If the judge acts correctly, condemning Diabolik when he is really guilty, then the public opinion will be happy about it. If the judge is corrupted and Diabolik is guilty, then the opinion will be upset with the judge's decision.

2.3 2nd Case: Diabolik is not guilty

In the case where Diabolik is not guilty we have the following payoff matrix:

		Guilty Diabolik	
		Sue	Not Sue
Judge	Condemn	$(-2,-1)$	$(0,-2)$
	Absolve	$(-1,-1)$	$(0,0)$

Table 2: Payoff table when Diabolik is not guilty.

In this case we can make the opposite considerations about the public opinion. If the judge condemns Diabolik, then the public opinion will not agree with judge's decision. Otherwise, if the judge makes the right choice the public opinion is happy.

2.4 Investment Agency

This game can be modelled as follows:

- we have N *players*, which are the contributors. We will label each player with P_i for $1 \leq i \leq N$.
- each player P_i has his/her own *set of strategies* S_i, where $S_i = \{\text{Contribute}, \text{Not Contribute}\}$.
- with the functions $u : S \to \mathbb{Z}$ and $c : S \to \mathbb{Z}$ to denote *payoff* and *cost* for each player, defined below:

$$u(s_i) = \frac{(k \cdot 100) \cdot 1.5}{N} \quad \text{where } k = \#\text{contributing players, including } i$$

$$c(s_i) = \begin{cases} 100 & \text{if player } P_i \text{ Contibutes} \\ 0 & \text{otherwise} \end{cases}$$

The total incomes for a player P_i, denoted with g_i, are computed using the following equation:

$$g_i = u(s_i) - c(s_i)$$

We can have two specific situations. In the foremost, if the incomes $g_i \geq 0$ then the player benefits from the investments, otherwise, if the incomes are $g_i \leq 0$ the contributor loses money. The important question to ask ourselves is the following one: when should I contribute for the project? To answer that, let us build an incoming matrix, containing all the possible situations for the game. We restrict ourselves in the case of $N = 2$ players (this could be easily generalized). With C we indicate a *contributing* player, instead, with NC a non-contributing one.
Let us analyze the table:

		Contributor 2	
		C	NC
Contributor 1	C	$(50, 50)$	$(-25, 75)$
	NC	$(75, -25)$	$(0, 0)$

Table 3: Incomes table for $N = 2$ players.

1. if the two players decide to contribute, then they will gain 50 euros, but they will have a gain of 150 euros.
2. if one of the two decides to contribute, then the non-contributing player will benefit of 75 euros for the next year, instead the contributing one has lost 25 euros.
3. if none of them contribute, then they will not gain or lose nothing.

Surprisingly enough, the only one *Nash Equilibrium* is obtained when none of the players contributes.

UNIVERSITÀ DI PISA

Tenth hands-on: Game Theory II

Algorithm Design (2021/2022)

Gabriele Pappalardo
Email: g.pappalardo4@studenti.unipi.it
Department of Computer Science

May 2022

1 Introduction

1.1 Problem 1: Crossing Streets

Two players drive up to the same intersection at the same time. If both attempt to cross, the result is a fatal traffic accident. The game can be modeled by a payoff matrix where crossing successfully has a payoff of 1, not crossing pays 0, while an accident costs 100.

- Build the payoff matrix
- Find the Nash equilibria
- Find a mixed strategy Nash equilibrium
- Compute the expected payoff for one player (the game is symmetric)

1.2 Problem 2: Mixed strategy for Back Stravinsky Game

Find the mixed strategy and expected payoff for the Back Stravinsky game.

1.3 Problem 3: Children to Kindergartens

The Municipality of your city wants to implement an algorithm for the assignment of children to kindergartens that, on the one hand, takes into account the desiderata of families and, on the other hand, reduces 'city traffic caused by taking children to school. Every school has a maximum capacity limit that cannot be exceeded under any circumstances. As a form of welfare the Municipality has established the following two rules:

1. in case of a child already attending a certain school, the sibling is granted the same school;
2. families with only one parent have priority for schools close to the workplace.

Model the situation as a stable matching problem and describe the payoff functions of the players. Question: what happens to twin siblings?

2 Solution

2.1 Solution for 'Crossing Streets'

The action set is build with two events $A = \{Cross, Stop\}$. With $D_i, i \in [1, 2]$ we define the i-th driver that is crossing the street. The payoff table is shown in Table 2.1.
As we can see from the table, we have two Nash equilibrium, which are the profiles: $(Cross, Stop)$ and $(Stop, Cross)$.

		Driver 2	
		Cross	Stop
Driver 1	Cross	$(-100, -100)$	$(1, 0)$
	Stop	$(0, 1)$	$(0, 0)$

Table 1: Payoff Table of Crossing Streets

To find the mixed strategies for this problem, we solve the following equations:

$$\begin{cases} \mu_{D_1}(Cross) = \mu_{D_1}(Stop) \\ \mu_{D_2}(Cross) = \mu_{D_2}(Stop) \end{cases} \iff \begin{cases} -100q + 1(1-q) = 0q + 0(1-q) \\ -100p + 1(1-p) = 0p + 0(1-p) \end{cases} \iff \begin{cases} q = \frac{1}{101} \\ p = \frac{1}{101} \end{cases}$$

Since the game is symmetric, the expected payoff is equal for both players:

$$\begin{aligned} E[\![\mu_{D_1}]\!] = E[\![\mu_{D_2}]\!] = & \\ &= pq(-100) + p(1-q)1 + (1-p)q0 + (1-p)(1-q)0 \\ &= -100pq + p - qp = -101pq + p = p(1 - 101q) = p(1 - 101p) = 0 \end{aligned}$$

2.2 Solution for 'Mixed Strategy for Back Stravinsky Game'

The payoff table for Bach-Stravinsky game is the following one:

		Wife	
		Bach	Stravinsky
Husband	Bach	$(2, 1)$	$(0, 0)$
	Stravinsky	$(0, 0)$	$(1, 2)$

Table 2: Payoff Table for Bach-Stravinsky

With μ_H and μ_W we refer to the payoff function for the husband and wife. The set of strategies that the couple can choose is the following one:

$$S_H = S_W = \{\text{Bach}, \text{Stravisnky}\}$$

To find a mixed strategy we need to convince the players to play a randomized strategy instead of a fixed one. We need to find a probability distribution $(p, 1-p)$ for the wife that makes the husband indifferent. This means that we have to equal $\mu_W(\text{Bach}) = \mu_W(\text{Stravisnky})$, where:

$$\begin{aligned} \mu_W(\text{Bach}) &= 1 \times p + 0 \times (1-p) \\ \mu_W(\text{Stravinsky}) &= 0 \times p + 2 \times (1-p) \end{aligned}$$

Computing the equation $\mu_W(\text{Bach}) = \mu_W(\text{Stravisnky})$ we obtain:

$$\begin{aligned} \mu_W(\text{Bach}) = \mu_W(\text{Stravisnky}) & \\ &\iff 1 \times p + 0 \times (1-p) = 0 \times p + 2 \times (1-p) \\ &\iff p = 2 \times (1-p) \\ &\iff p = \frac{2}{3} \end{aligned}$$

For the husband we have to act similar. Let us find a probability distribution $(q, 1-q)$ for the husband, such that makes the wife indifferent regard to the selected choice. So find q by solving the equation $\mu_H(\text{Bach}) = \mu_H(\text{Stravinsky})$:

$$\begin{aligned}\mu_H(\text{Bach}) = \mu_H(\text{Stravisnky}) & \\ &\iff 2 \times q + 0 \times (1-q) = 0 \times q + 1 \times (1-q) \\ &\iff 3q = 1 \\ &\iff q = \frac{1}{3}\end{aligned}$$

2.3 Solution for 'Children to Kindergartens'

Given two sets of equal size, the *stable matching problem* consists of matching each element of one set to one in the other. In this problem each element has some preferences.

The following solution gives the idea behind the real one. We can describe the stable matching problem creating two sets: the first one is the set containing the children will go at schools, and, since the number of schools of a city does not match the number of children, the second set is the number of "seats" available in each school. For example, given n schools indicated with S_i (the i-th school), each with its own capacity c_i, we define the slots in the following way:

$$s_{i_k} = k\text{-th slot available in the school } i \quad \text{for } 0 \leq k \leq c_i$$

According to the game's description, we can model the preferences, for each child, as the distance between the child's location (indicated with H_i for the house and W_i for the parent's workplace) and the school (defining a function $d(H_i, S_i)$).

First, families with only parent have the higher priority overall, so we try to minimize the distance between the parent's workplace and the school as:

$$\min_i \{d(W_i, S_i)\}$$

Second, if a child (with a sibling) is already displaced in a school, then his/her sibling will be placed in the same one, setting the distance d equal to 0. The twins will happen to be in the same school, since they can be seen as a "unique" child with weight 2.