Projeto 1 - Segurança Computacional

Cláudio Roberto Oliveira Peres de Barros, 19/0097591

[1]Dep. Ciência da Computação – Universidade de Brasília (UnB)
CIC0201 - Segurança Computacional

190097591@aluno.unb.br

Resumo. *Este projeto visa a implementação de um programa que implemente o funcionamento de uma cifra de Visionère e funções que permitam descriptografar uma mensagem cifrada usando esta cifra com uma chave de tamanho desconhecido.*

1. Introdução

A cifra de Vigenère é uma cifra de substituição polialfabética, um avanço em relação às cifras de substituição monoalfabéticas simples, como a cifra de César. Ao contrário de cifras monoalfabéticas, a cifra de Vigenère utiliza uma chave para determinar o deslocamento das letras no processo de criptografia. O processo de criptografia envolve repetir a chave para igualar o comprimento da mensagem do texto original. Cada letra do texto original é, então, deslocada de acordo com a letra correspondente na chave repetida, gerando um padrão variável de deslocamento, aumento a complexidade do processo necessário para ”quebrar”textos cifrados com esse esquema.

2. Codificador

```
static std::map<char, int> letter_index= {
    {'a', 0}, {'b',1}, {'c',2}, {'d',3}, {'e',4}, {'f',5}, {'g', 6},
    {'h', 7}, {'i',8}, {'j',9}, {'k',10}, {'l',11}, {'m',12}, {'n', 13},
    {'o', 14}, {'p',15}, {'q',16}, {'r',17}, {'s',18}, {'t',19}, {'u', 20},
    {'v', 21}, {'w',22}, {'x',23}, {'y',24}, {'z',25}};

static std::string alphabet = "abcdefghijklmnopqrstuvwxyz";

std::string dec(std::string key,
                std::string s_in)
{

    std::string s_out(s_in.size(), 0); //ciphertext

    int k = 0; //key index
    int shift = 0;
    int base = 0;
    int k_size = key.size();
    int index = 0;

    for(int i = 0; i < s_in.size(); i++)
    {
        base = letter_index[s_in[i]];
```

```
        shift = letter_index[key[k % k_size]];

        index = (base < shift) ? ((base - shift) + 26) : (base - shift);

        s_out[i] = alphabet[index % 26];

        k++;
    }

    return s_out;
}
```

3. Decodificador

```
\label{sec:Decodificador}
std::string dec(std::string key,
                std::string s_in)
{

    std::string s_out(s_in.size(), 0); //ciphertext

    int k = 0; //key index
    int shift = 0;
    int base = 0;
    int k_size = key.size();
    int index = 0;

    for(int i = 0; i < s_in.size(); i++)
    {
        base = letter_index[s_in[i]];

        shift = letter_index[key[k % k_size]];

        index = (base < shift) ? ((base - shift) + 26) : (base - shift);

        s_out[i] = alphabet[index % 26];

        k++;
    }

    return s_out;
}
```

4. Criptoanálise

Uma das fraquesas da cifra de Vegenère está na repetição de sua chave, o que em certos casos permite a utilização de métodos que descubram o tamanho da chave. Uma vez que o tamanho da chave é conhecido, basta tratar o texto como uma sequência de n cifras de César intercaladas (sendo n o tamanho da chave). A partir daí, cada cifra obtida pode ser facilmente quebrada utilizando um ataque de análise de frequência.

Para obter o tamanho da chave, o método utilizado foi o de Kasiski [**?**] [**?**] juntamente com o teste de Friedman. O método de Kasiski consiste em observar o fato de que

grupos de duas ou mais letras podem ser encriptados usando a mesma sequência de letras, especialmente em mensagens mais longas. A distância dessa repetições é usada, juntamente com os números inteiros que compõe a fatoração do tamanho da repetição, para obter os tamanhos mais prováveis da chave. A partir daí, os tamanhos prováveis são utilizados para calcular o índice de coincidência (IC) do texto para cada tamanho provável, calculando o IC para cada cifra de César intercalada obtida a partir do tamanho da chave. Os tamanhos possíveis com ICs mais próximos à textos aleatórios do que ao IC da língua a ser descriptografada são descartados. Os tamanhos possíveis restantes são ordenados a partir da quantidade de vezes que esse número apareceu como fator na fatoração inteira das distâncias, e o número com maior frequência é retornado como possível tamanho de chave.

O índice de coincidência é obtido a partir da equação abaixo:

$$\mathrm{IC} = \frac{1}{N(N-1)} \sum_{i=1}^{n} F_i(F_i - 1)$$

Figura 1. Cálculo de índice de coincidência.

Abaixo, estão disponíveis os códigos que implementam o método de Kasiski e sua utilização para obter o tamanho da chave.

```
std::vector<std::pair<int,int>> kasiski(std::string s_in)
{
    std::vector<Trigram> trigrams;
    std::unordered_map<std::string, int> added; //{trigram, vector index}

    for(int i = 0; i <= (s_in.size()-3); i++)
    {
        std::string substr = s_in.substr(i, 3);

        auto it = added.find(substr);
        if(it == added.end())
        {
            trigrams.push_back(Trigram{substr, 1, {i}});
            added.insert({substr, trigrams.size()-1});
        }
        else
        {
            trigrams[it->second].frequency++;
            trigrams[it->second].indices.push_back(i);
        }
    }
    std::sort(trigrams.begin(), trigrams.end(), compTrigrams);

    //calculate distances and get factors:
    std::vector<std::pair<int, int>> factor_freq; //{factor, frequency}
    std::unordered_map<int, int> added_factors; //factor index

    // std::vector<std::vector<int>> factors;

    for(int i = 0; i < trigrams.size(); i++)
```

```
        {
            if(trigrams[i].frequency < 2)
            {
                continue;
            }

            for(int j = 0; j < trigrams[i].indices.size()-1; j++)
            {
                int a = trigrams[i].indices[j+1];
                int b = trigrams[i].indices[j];
                trigrams[i].distances.push_back(a-b);
                // factors.push_back(get_factors(a-b));

                std::vector<int> factors = get_factors(a-b);

                for(int f = 0; f < factors.size(); f++)
                {
                    auto iter = added_factors.find(factors[f]);
                    if(iter == added_factors.end())
                    {
                        factor_freq.push_back({factors[f], 1});
                        added_factors.insert({factors[f], factor_freq.size()-1});
                    }
                    else
                    {
                        factor_freq[iter->second].second++;
                    }
                }
            }
        }

        // return the factor (greater than or equal to 3) with the greatest frequency
        std::sort(factor_freq.begin(), factor_freq.end(), compFactors);

        return factor_freq;
    }


bool comp_length(std::tuple<int,int,double> &a,
                 std::tuple<int,int,double> &b)
{
    if(std::get<1>(a) == std::get<1>(b))
        return (std::get<2>(a) > std::get<2>(b));
    return (std::get<1>(a) > std::get<1>(b));
}

int guess_key_length(std::string s_in)
{
    std::vector<std::pair<int,int>> vec_factors = kasiski(s_in);

    std::vector<std::tuple<int, int, double>> vec_ics;

    for(auto f : vec_factors)
    {
        double acc = 0.0;
        std::vector<std::string> v = get_cosets(s_in, f.first);
```

```
        for(int i = 0; i < v.size(); i++)
        {
            acc += get_ic(v[i]);
        }

        double avg = acc/(double)v.size();

        //if avg is closer to random ic than to english ic, ignore value
        if(fabs(ic - avg) > fabs(ic_random - avg))
        {
            continue;
        }

        for(int i = 0; i < vec_factors.size(); i++)
        {
            if(vec_factors[i].first == f.first)
            {
                vec_ics.push_back({f.first, vec_factors[i].second, avg});
            }
        }

    }

    std::sort(vec_ics.begin(), vec_ics.end(), comp_length);

    if(vec_ics.size() == 0)
    {
        std::cout << "Unable to break ciphertext. Text is possibly too small.\n\n";
        exit(0);
    }

    return std::get<0>(vec_ics[0]);
}
```

Após descobrir o tamanho da chave, podemos utilizar análise de frequência em cada string de texto que corresponda a uma cifra de César codificada com uma letra específica da chave. Para cada uma dessas strings (n strings, sendo n o tamanho da chave), testamos as 26 possibilidades de deslocamento e selecionamos para a posição correspondente na chave a letra que corresponde ao deslocamento que, quando aplicado na string, obtem a frequência de letras do alfabeto mais próximas ao que é esperado na língua em que a mensagem foi escrita. Ao fim desse processo, obtemos a chave e podemos utilizá-la para descriptografar o texto.

O código que implementa esse processo é apresentado abaixo:

```
std::string recover_key(std::string s_in)
{
    int len = guess_key_length(s_in);

    std::cout << "---- The guessed key length is: " << len << "\n\n";

    std::vector<std::string> cosets = get_cosets(s_in, len);
    std::vector<std::vector<double>> delta_vec(len, std::vector<double>(26, 0.0));
```

```
    for(int i = 0; i < len; i++)
    {
        for(int j = 0; j < 26; j++)
        {
            delta_vec[i][j] = str_freq_analysis(dec(std::string(1, alphabet[j]), cose
        }
    }

    std::vector<int> min_idxs(len, 0);
    for(int i = 0; i < len; i++)
    {
        double min = 0.0;
        for(int j = 0; j < 26; j++ )
        {
            if(delta_vec[i][j] < min || j == 0)
            {
                min = delta_vec[i][j];
                min_idxs[i] = j;
            }
        }
    }

    std::string key;
    for(int i = 0; i < len; i++)
    {
        key.append(std::string(1, alphabet[min_idxs[i]]));
    }

    return key;
}
```

Referências

[Katz and Lindell 2015] Katz, J. and Lindell, Y. (2015). In *Introduction to modern cryptography*. CRC Press/Taylor amp; Francis.

[University] University, M. T. Kasiskiapos;s Method — pages.mtu.edu. `https://pages.mtu.edu/~shene/NSF-4/Tutorial/VIG/Vig-Kasiski.html`. [Accessed 03-10-2023].