Universidade Federal de Juiz de Fora

Departamento de Ciência da Computação

Linguagem de Programação

Paradigma Lógico

Daniel Augusto Machado Baeta - 201965122C

Thiago do Vale Cabral - 201965220AC

Relatório do trabalho prático apresentado ao prof. Leonardo Vieira dos Santos Reis como requisito de avaliação da Disciplina DCC019 - Linguagem de Programação

Juiz de Fora

Maio de 2022

Relatório Técnico

Daniel Augusto Machado Baeta [1]
Thiago do Vale Cabral [1]

1 Introdução

O presente trabalho tem como temática central descrever a resolução de problemas de criptografia por meio da ótica de paradigma lógico. Para tal, propõe-se a estruturação de um algoritmo usando a linguagem de programação Prolog. Os resultados obtidos, assim como a estratégia de resolução utilizada, são alvos de discussão neste texto.

A realização desse experimento se deu como forma de consolidar e aplicar os conceitos teóricos vistos durante a disciplina DCC019 - Linguagem de Programação durante o primeiro semestre de 2022. O problema e a paradigma de programação apresentados como objeto de estudo neste trabalho são importantes para demonstrar aos discentes as diferentes estratégias, assim como seus respectivos prós e contras, de resolução de um mesmo problema.

Diante disso, na seção 2 descreve-se o contexto do problema; na seção 3 mostra-se a abordagem e estratégia utilizada, enquanto a seção 4 apresenta o experimento realizado juntamente com análise dos resultados obtidos. Por último, a seção 5 apresenta as conclusões do trabalho.

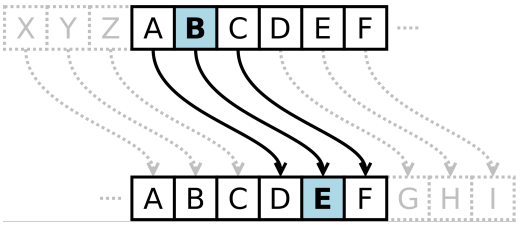
[1]Departamento de Ciência da Computação – Instituto de Ciências Exatas
Universidade Federal de Juiz de Fora (UFJF)
Rua José Kelmer, s/n, Campus Universitário, Bairro São Pedro
CEP: 36036-900 – Juiz de Fora/MG – Brasil
{daniel.baeta, thiago.cabral}@ice.ufjf.br

2 Descrição do problema

2.1 Cifra de César

A cifra de César, também conhecida como cifra de troca, é um dos métodos mais simples de criptografia. Esta é uma cifra de substituição que consiste na troca de cada caractere por outro correspondente. Este caractere resultante está abaixo do caractere original no alfabeto por um número fixo de vezes. Este número que corresponde à distância de uma letra a outra no alfabeto é a chave para realizar a cifra de César.

Figura 1 – Representação da cifra de César



É importante ressaltar que na cifra de César, o deslocamento dos caracteres é fixo. Além disso, conforme demonstrado na figura 4, o deslocamento volta para o inicio do alfabeto quando a última letra do mesmo é ultrapassada.

Na prática a cifra de César é incorporada em esquemas mais complexos de criptografia, como por exemplo a cifra de Vigenère (subseção 2.2). Isto se dá porque a aplicação pura de uma cifra monoalfabética é facilmente decifrável.

2.2 Cifra de Vigenère

Sendo outro método de criptografia conhecido, a cifra de Vigenère consiste na aplicação de uma série de cifras de César. Trata-se do método mais simples de substituição poli alfabética, onde a chave para criptografar a mensagem não é mais um valor fixo, mas sim outra outra chave. Assim, um texto criptografado torna-se mais difícil de ser decodificado.

Por exemplo, dado a mensagem:

maria saiu de casa para namorar

Escolhe-se a chave "socorro". Esta chave é estendida até atingir o comprimento máximo do texto:

socorrosocorrosocorrosocorrosoc

Assim, para cada letra da mensagem principal, é aplicada uma cifra de césar com a chave correspondente. Consequentemente temos como resultado a seguinte cifra:

Texto: *maria saiu de casa para namorar*
Chave: *socorrosocorrosocorrosocorrosoc*
Cifra: *epuxsrgtxxovwovpvprgpjpcbsdcjpu*

A cifra de Vigenère, apesar de ser a mais simples das substituições poli alfabéticas, é extremamente eficiente. Entretanto, é importante salientar que chaves muito pequenas torna o processo de identificação de padrões muito mais fácil. Logo, é altamente recomendado estruturar chaves mais longas para a códificação do texto.

2.3 Paradigma Lógico

Emergido na década de 1970, a abordagem do paradigma lógico tem como objetivo principal expressar programas de forma lógica simbólica e usar um processo de inferência lógica para produzir resultados. Por isto, é uma linguagem altamente recomendada para problemas na área de inteligência artificial, processamento de linguagem natural, criação de sistemas especialistas e até mesmo prova de teoremas. A linguagem mais popular de programação lógica é o Prolog.

De forma geral, linguagens de programação lógica abordam a produção de resultados através da união de predicados, onde o próprio programa utiliza da união de fatos para encontrar um objetivo. A base do programa é constituída por fatos, que são proposições que são incondicionalmente verdadeiras. Exemplo:

```
homem(Leonardo).
```

No exemplo acima, temos que "homem"é um predicado, "Leonardo"é um átomo e a união dos dois é definida como um fato. Além disso podemos declarar regras em programação lógica:

```
pais(X,Y) :- pai(X,Y).
pais(X,Y) :- mae(X,Y).
irmao(X,Y) :- pais(Z,X), pais(Z,Y), X Ȳ.
```

O predicado "pais" acima pode ser interpretado como "pai"ou "mae", e o predicado "irmao" pode ser interpretado como X e Y compartilhando o mesmo pai ou mãe. Assim, podemos construir relações maiores e mais complexas entre fatos e predicados.

Outro recurso muito interessante em um paradigma lógico é a reversibilidade das operações. Não há distinções entre entrada e saída. Assim, um predicado pode ser usado de diferentes formas dado o contexto da aplicação. Um exemplo de reversibilidade pode ser visto abaixo:

```
?- pai(Leonardo, Maria).
true

?- pai(X, Maria).
X = Leonardo

?- pai(Leonardo, X).
X = Maria
```

Assim, podemos verificar se a inferência está correta ao fornecer todos os argumentos como entrada. Quando omitimos algum argumento com a representação "X", o processo de unificação nos retorna um átomo que satisfaz a veracidade do predicado.

3 Abordagens para o problema

Frente ao desafio de construir um codificador e decodificador em Prolog, buscou-se estruturar um programa com os seguintes requisitos:

1. Possuir e expandir - de forma dinâmica e persistente - o banco de palavras;
2. Definir estaticamente um alfabeto válido para a mensagem;
3. Permitir encontrar uma cifra de César com a entrada de uma mensagem e uma chave numérica;
4. Permitir encontrar uma cifra de Vigenère com a entrada de uma mensagem e uma mensagem chave;
5. Permitir o procedimento reverso dos requisitos 3 e 4;
6. Decifrar uma mensagem criptografada pela cifra de César sem a chave;
7. Encontrar a chave sabendo a mensagem cifrada, tamanho da chave, uma palavra decifrada e sua posição;
8. Decifrar a mensagem sabendo a mensagem cifrada, tamanho da chave, uma palavra contida na mensagem;
9. Decifrar a mensagem sabendo a mensagem cifrada, tamanho da chave, litas de palavras contidas, ou não, na mensagem;

Ao que tange aos requisitos 1 e 2, foram criados os arquivos *words.pl* e *alphabet.pl*. Enquanto o alfabeto permanece fixo, o banco de arquivos foi declarado em conjunto com o predicado *add_word*, que permite adicionar uma palavra em tempo de execução com o predicado assertz e de forma persistente no arquivo com o predicado *write*.

Quanto à cifra de César referenciada no requisito 3, foi criado o predicado de 3-aridade cesar, que possui as seguintes entradas:

- *Message*: mensagem a ser cifrada;
- *Key*: chave numérica para aplicar a cifra;
- *Cipher*: mensagem cifrada resultante da codificação;

Para a cifra de Vigenère referenciada no requisito 4, foi também criado um predicado de 3-aridade vigenere. Similar a César (requisito 3), este predicado possui as entradas de forma semelhante, com exceção da chave *Key* que fora substituído por uma entrada *Message_Key* correspondente à mensagem chave responsável por determinar a sequência de cifras de César a ser aplicada, conforme falado na seção 2.2.

Ambas as cifras implementam um predicado que calcula o deslocamento de césar, chamado de *calculate_offset*. Este predicado segue a fórmula matemática abaixo que, de forma objetiva, atribui à saída o resto da divisão entre a soma da entrada e o deslocamento com o total de letras do alfabeto.

```
calculate_offset(Offset,Entry,Output) :-
  count_codes(Total_Codes),
  Output #= (Entry+Offset) mod Total_Codes.
```

Para garantir a reversibilidade de César e Vigenère no requisito 5, foi utilizado um predicado auxiliar *is_string_or_atom* para validar a condição de aplicação da cifra. O resultado deste predicado auxiliar determina a sequência de predicados subsequentes a serem seguidos. Esta reversibilidade forçada foi-se necessária e será justificada na seção 4 referente aos resultados.

Por fim, para os requisitos 6, 7, 8 e 9, foram criados predicados específicos e direcionados para abordar os respectivos problemas. Por se tratarem de situações muito bem delimitadas, os predicados foram modelados de forma irreversível, ou seja, não é possível fazer o caminho inverso. Esta decisão foi tomada pois o caminho inverso nessas situações simplesmente não faz sentido.

4 Resultados

Os predicados foram construídos e algumas dificuldades foram encontradas. Para as cifras de César e Vigenère funcionarem tanto para codificar quanto para decodificar, no mesmo predicado, foi necessário a implementação de um "if"para distinguir qual parâmetro havíamos recebido. O problema ocorreu devido a necessidade, na implementação, de transformar ambos parâmetros string (ou atom) recebidos em um vetor de códigos, então quando a consulta era feita com um dos parâmetros sendo uma variável o programa acusava erro. Outra dificuldade encontrada em relação a utilizar o mesmo predicado para codificação e decodificação foi devido à operação de mod realizada para calcular o deslocamento das letras. Tal operação garante que os valores de código não ultrapassarão o valor máximo, porém é gerado um backtracking devido a possibilidade de múltiplas repostas. Segue abaixo exemplo do funcionamento da codificação/decodificação de César e Vigenère:

Figura 2 – codificando/decodificando com cifra de César e Vigenère

```
?- cesar("camelo muito azul",4,X).
X = "geqipsdqymxsde yp".

?- cesar(X,4,"geqipsdqymxsde yp").
X = "camelo muito azul".

?- vigenere("camelo muito azul","ouro",X).
X = "rvat,fr.f h ovnf,".

?- vigenere(X,"ouro","rvat,fr.f h ovnf,").
X = "camelo muito azul".
```

Para decifrar a cifra de César por força bruta precisou-se gerar uma base de palavras, então o sucesso do método depende de quais palavras temos na base. Para facilitar a manipulação da base foram desenvolvidos predicados de inserção e remoção como podemos ver abaixo:

Figura 3 – Manipula banco e decifra César

```
?- word(barco).
false.

?- add_word(barco).
true.

?- word(barco).
true.

?- cesar("barco muito azul",4,X).
X = "fevgsdqymxsde yp".

?- decript_cesar("fevgsdqymxsde yp",X).
X = "barco muito azul".
```

Para decifrar uma mensagem cifrada por Vigenère precisamos descobrir o tamanho da palavra chave utilizada e para isso precisaríamos utilizar técnicas de índice de coincidência, que diz respeito a probabilidade de duas letras aleatórias no texto serem idênticas. Entretanto neste trabalho optamos pela implementação de três tipos diferentes no intuito de descobrir a palavra chave ou a mensagem cifrada. Podemos ver nas imagens abaixo o funcionamento de cada um deles:

Figura 4 – Predicados que decifram Vigenère

```
?- vigenere("um predicado que relaciona uma mensagem cifrada uma lista de possiveis palavras que ocorrem no texto e
 um tamanho de chave com a mensagem decifrada","bola",X).
X = "w.lqttpjepppbbcfbcqmcrupplvoplng:abityaexrscsmaw.manxaucopfba,tuxdfkdlqc,mwtpaasfqaqr,sttyap lugibpbtlvoobbop
ziqopfbrtbxtldq.lbb.qoupsfoopfexrscsm".

?- find_key("w.lqttpjepppbbcfbcqmcrupplvoplng:abityaexrscsmaw.manxaucopfba,tuxdfkdlqc,mwtpaasfqaqr,sttyap lugibpbt
lvoobbopziqopfbrtbxtldq.lbb.qoupsfoopfexrscsm",4,"predicado",4,X).
X = "bola".

?- decript_vigenere_one_word("w.lqttpjepppbbcfbcqmcrupplvoplng:abityaexrscsmaw.manxaucopfba,tuxdfkdlqc,mwtpaasfqaq
r,sttyap lugibpbtlvoobbopziqopfbrtbxtldq.lbb.qoupsfoopfexrscsm",4,"predicado",X).
X = "um predicado que relaciona uma mensagem cifrada uma lista de possiveis palavras que ocorrem no texto e um tama
nho de chave com a mensagem decifrada".

?- decript_vigenere_multiple_words("w.lqttpjepppbbcfbcqmcrupplvoplng:abityaexrscsmaw.manxaucopfba,tuxdfkdlqc,mwtpa
asfqaqr,sttyap lugibpbtlvoobbopziqopfbrtbxtldq.lbb.qoupsfoopfexrscsm",["predicado","carro"],4,X).
X = "um predicado que relaciona uma mensagem cifrada uma lista de possiveis palavras que ocorrem no texto e um tama
nho de chave com a mensagem decifrada".
```

5 Conclusão

De modo geral, ao observamos os resultados obtidos, foi possível afirmar que o paradigma lógico auxiliou bastante na resolução de um problema de codificação. Ser um paradigma que permite uma reversibilidade lógica de predicados foi algo extremamente vantajoso, pois reduziu drasticamente o *boilerplate* da resolução.

Entretanto, com a utilização de uma abordagem que manipula strings e átomos diretamente e o fato do cálculo do *offset* envolver a operação de resto, houveram casos onde o caminho reverso de um predicado teve-se de ser implementado manualmente e de forma separada. Ainda sim, o uso do paradigma lógico se manteve bastante vantajoso.

Observou-se também que algumas funcionalidades do trabalho prático eram muito imperativas para um paradigma lógico. Por causa disto, operações como escrita e remoção de palavras no banco de palavras se tornaram bastante confusas para o modelo de lógica de predicados.

Por fim, acredita-se que foi possível observar os *tradeoff's* de um paradigma com lógica de predicados, ressaltando que, principalmente para predicados onde a implementação bidirecional faz muito sentido, é uma excelente escolha de paradigma.