An spy history
Trabalho Prático 3

Luís Eduardo Oliveira Lizardo

[1]Projeto e Análise de Algoritmos
Departamento de Ciência da Computação
Universidade Federal de Minas Gerais (UFMG)

lizardo@dcc.ufmg.br

1. Introdução

Neste trabalho fazemos a interceptação de comunicações por meio de um ataque "Man-in-the-middle". A interceptação permite descobrir o conteúdo de mensagens trocadas pelo centro de operações de uma agência de petróleo da Coreia do Norte. As informações a serem decodificadas aparecem entre dois comandos de controle no fluxo de bits: as sequências "000" e "11111".

No entanto, devido a diferentes erros de transmissão, nem sempre é possível identificar alguns bits no fluxo da mensagem. Por isso, o objetivo deste trabalho é construir um programa que verifique se a mensagem tem ou não os comandos de controle, ou se é impossível distinguir se a mensagem tem ou não os comandos devido aos erros.

O programa recebe como entrada várias mensagens em um arquivo e para cada uma delas ele imprime: *true* – caso a mensagem tenha os comandos de controle; *false* – caso não tenha; ou *both* – quando não é possível definir se a mensagem tem ou não os comandos de controle. Uma mensagem é representada por uma *string* que pode conter os caracteres '0', '1' ou '-'. Os caracteres '0', '1' são os bits corretamente identificados na transmissão, e o caractere '-' representa os bits que não podem ser identificados devido aos erros. Os resultados são impressos num arquivo de saída "output.txt" no mesmo diretório da execução do programa.

Neste trabalho, o programa foi implementado utilizando três paradigmas diferentes de programação: *força bruta (BF)*, *algoritmo guloso (GA)* e *programação dinâmica (DP)*. BF e DP apresentam soluções exatas e o GA apresenta uma solução aproximada. As três versões do programa serão analisadas neste relatório.

O restante deste relatório está organizado da seguinte forma: a Seção 2 descreve o problema abordado e explica a entrada e saída do programa. A Seção 3 explica as três versões do programa implementadas e apresenta a análise de complexidade de cada uma. A Seção 4 mostra os resultados obtidos nas análises experimentais dos programas.

2. Descrição do Problema

Dada uma *string* de caracteres '0', '1' ou '-' que representam bits, o problema neste trabalho consiste em identificar se ela possui "000" ou "11111" como *substring*. O caractere que representa erros ('-') pode assumir tanto o valor de '0' ou '1'. Dessa forma, uma *string* "10-0", pode ser identificada como "1000" ou "1010".

Quando a *string* contém "000" ou "11111", a resposta do programa é *true*. Quando ela não contém nenhuma das duas *substrings*, a resposta é *false*. No exemplo acima, a

string “10-0”, pode ser identificada como “1000”, cuja resposta é *true*, ou “1010”, cuja resposta é *false*. Neste caso, a resposta do programa é *both*, pois a existência da *substring* está condicionada ao valor assumido por ‘-’.

Como exemplo, temos as seguintes *strings*:

“0100011” – *true* – porque contém “000”;
“011111-” – *true* – porque contém “11111”;
“010-111” – *false* – não contém “000” ou “11111” e nem é possível formá-los com o caractere ‘-’;
“00-1111” – *true* – porque independentemente do valor assumido por ‘-’, a *string* sempre terá “000” ou “11111”;
“00-011” – *both* – poque está condicionado ao valor de ‘-’. Se ‘-’ for ‘0’ é *true*, pois será formada a *string* “000011”. Caso ‘-’ seja ‘1’, a resposta seria *false*, pois a *string* “001011” não possui “000” ou “11111”.

2.1. Entrada e saída

A entrada do programa é feita através de um arquivo texto que contém, na primeira linha, um inteiro que representa o número de mensagens no arquivo, e nas demais linhas cada uma das mensagens.

A saída do programa é feita em um arquivo texto, “output.txt” salvo no mesmo diretório da execução do programa. Este arquivo contém uma resposta por linha para cada mensagem do arquivo de entrada, conforme os exemplos:

input.txt:	**output.txt:**
4	false
1011010111	both
101101-111	false
1–10101–	true
00-1111	

3. Soluções propostas

Nesta seção são explicados as três versões implementadas do programa. Cada uma utiliza um paradigma de programação diferente.

3.1. Estratégia de força bruta (BF)

Para esta solução foi implementado um algoritmo recursivo que gera todas as possíveis *strings* candidatas a solução. O algoritmo recebe como parâmetros a mensagem avaliada, um índice i, que aponta o caractere atual da mensagem na iteração e dois contadores *c0* e *c1*, que contam, respectivamente, o número de ‘0s’ e ‘1s’ consecutivos.

A cada iteração o algoritmo verifica o caractere da mensagem apontado por i. Se ele é ‘0’, *c0* é incrementado e *c1* é zerado. Se ele é ‘1’, *c1* é incrementado e *c0* é zerado. Caso o caractere seja ‘-’, o algoritmo se chama recursivamente duas vezes. Uma considerando ‘-’ como ‘0’, passando como parâmetro *c0+1* e *c1 = 0*, e a outra considerando ‘-’ como ‘1’, passando *c1+1* e zerando *c0 = 0*.

O algoritmo verifica em cada iteração se *c0* é igual a 3, ou se *c1* é igual 5. Caso seja verdade, o algoritmo retorna *true*, pois ele encontrou as *substrings* de comandos de controle. Quando o algoritmo chega ao final da mensagem sem encontrar os comandos de controle, ele retorna *false*. *Both* é apenas retornado quando há pelo menos um '-' e as duas chamadas recursivas retornam valores diferentes, por exemplo, *false* e *true*. O Algoritmo 1 mostra, de forma simplificada, a implementação.

Algoritmo 1: Algoritmo simplificado de força bruta.

```
Entrada: Mensagem msg, índice i e contadores c0 e c1
Saída: True, False ou Both
while true do
    if c0 == 3 ou c1 == 5 then
        retorna True;
    if i == msg.length() then
        retorna False;
    switch msg[i] do
        case '0'
            i++; c0++; c1 = 0;
        case '1'
            i++; c1++; c0 = 0;
        case '-'
            r0 = ForçaBruta(msg, i + 1, c0 + 1, 0);
            r1 = ForçaBruta(msg, i + 1, 0, c1 + 1);
            if r0 != r1 then
                retorna Both;
            else
                retorna r0;
```

A Equação 1 mostra a equação de recorrência do algoritmo de força bruta. A resolução dessa equação mostra que este algoritmo é exponencial no pior caso, com complexidade de tempo igual a $O(2^n)$, onde n é o número de caracteres da mensagem.

$$T(n) = 2T(n-1) + O(1) \tag{1}$$

3.2. Estratégia gulosa (GA)

O algoritmo guloso implementado é iterativo e escolhe um valor para cada caractere do tipo '-' de acordo com as informações que ele possui na iteração. Neste algoritmo, diferentemente do BF, é utilizado apenas um contador de repetição de caracteres c, e também é armazenado o tipo do caractere que está sendo contado em t. A cada iteração, se o caractere atual é igual a t, c é incrementado. Caso contrário, c recebe 1 como valor, e t é atualizado para o caractere atual. O Algoritmo 2 mostra, de forma simplificada, a implementação.

Dessa forma, quando t é '0' e c é 3 ou quando t é '1' e c é 5, o algoritmo retorna *true*. Se o GA chegar ao final da mensagem sem que as condições acima sejam verdadeiras, ele retorna *false*.

Quando o caractere da mensagem lido é '-', o GA verifica qual é a melhor opção para esse caractere, se é considerá-lo como '0' ou como '1'. Essa decisão é feita com base nos valores de t e c seguindo a seguinte ordem:

1. Se t é do tipo '0' e c é 2, assume o caractere como sendo '0', pois assim ele formará "000".
2. Se t é do tipo '1' e c é 4, assume o caractere como sendo '1', pois assim ele formará "11111".
3. Se t é do tipo '0', $msg[i+1]$ é do tipo '1' e c é menor que 2, assume o caractere como sendo '1'.
4. Se t é do tipo '1', $msg[i+1]$ é do tipo '0' e c é menor que 4, assume o caractere como sendo '0'.
5. Assume '-' como sendo igual ao tipo de t.

Os itens 1 e 2 verificam se é possível formar "000"ou "11111"com o que já foi lido anteriormente. Os itens 3 e 4 avaliam que não seria possível formar "000"ou "11111"com o que já foi lido, pois o próximo caractere na mensagem $msg[i+1]$ é diferente dos anteriores. Dessa forma, '-' é assumido como sendo diferente dos anteriores para tentar casar com uma possível *substring* que virá depois. Caso nenhuma das condições 1, 2, 3 e 4 sejam possíveis, na condição 5 '-' é assumido como sendo do tipo t e incrementa o contador c.

Sempre que algum caractere do tipo '-' é encontrado uma *flag* b é setada. Ao final do processamento da mensagem, se a b é *true* o programa retorna *both*, caso contrário ele retorna *true*.

O GA não é um algoritmo exato e apresenta soluções aproximadas que serão avaliadas na seção 4. A complexidade de tempo do GA é $\theta(n)$, onde n é tamanho da mensagem em número de caracteres.

3.3. Programação dinâmica (DP)

Nesta seção é apresentada a solução utilizando o paradigma de programação dinâmica do problema. As seções 3.3.1 e 3.3.2 explicam as propriedades de *subestrutura ótima* e da *sobreposição de subproblemas* do problema. A seção 3.3.3 apresenta a equação de recorrência do problema e a seção 3.3.4 explica o algoritmo.

3.3.1. Subestrutura ótima

O problema deste trabalho apresenta subestrutura ótima. A solução para uma mensagem M de tamanho n, tal que $M = \langle m_1, m_2, m_3, \ldots, m_n \rangle$, é obtida a partir da solução de uma mensagem M' de tamanho $n-1$, onde $M' = \langle m_2, m_3, \ldots, m_n \rangle$ de tamanho $n-1$, adicionando m_1 a solução.

3.3.2. Sobreposição de subproblemas

A Figura 1 mostra a árvore de recorrência do algoritmo para a mensagem "0-1-0". Podemos observar na figura que os subproblemas em vermelho são sobrepostos, assim como os dois em azul.

Algoritmo 2: Algoritmo simplificado guloso.

Entrada: Mensagem msg
Saída: *True*, *False* ou *Both*
r = *False*;
t = ' ';
for *i = 0 to msg.length()* **do**
  if $msg[i]$ == t **then**
    c++
  else
    if $msg[i]$!= '-' **then**
      c = 1;
      t = m;
      b = false;
    else
      if *condições 1, 2, 3 ou 4 são satisfeitas* **then**
        c = 1;
        t = $msg[i+1]$;
      else
        c++;
      b = true;
  if *(t == '0' c == 3) or (t == '1' c == 5)* **then**
    if b == *false* **then**
      retorna *True*;
    else
      r = *Both*;
retorna r;

3.3.3. Equação de recorrência

A equação de recorrência do DP é dada pelas Equações 2 e 3.

$$F(i, c0, c1) = \begin{cases} True, & se\ c0 = 3\ or\ c1 = 5\ , \\ F(i+1, c0+1, 0), & se\ msg[i] = 0\ , \\ F(i+1, 0, c1+1), & se\ msg[i] = 1\ , \\ G(i+1, c0, c1) & se\ msg[i] = -\ , \\ False, & se\ i = msg.length()\ , \end{cases} \tag{2}$$

$$G(i, c0, c1) = \begin{cases} True, & se\ F(i, c0+1, 0) = F(i, 0, c1+1) = True\ , \\ False, & se\ F(i, c0+1, 0) = F(i, 0, c1+1) = False\ , \\ Both, & caso\ contrario. \end{cases} \tag{3}$$

3.3.4. Algoritmo

Foram implementadas duas versões do algoritmo utilizando o paradigma de programação dinâmica: uma versão *top-down* e a outra *bottom-up* apresentada no algoritmo 3.

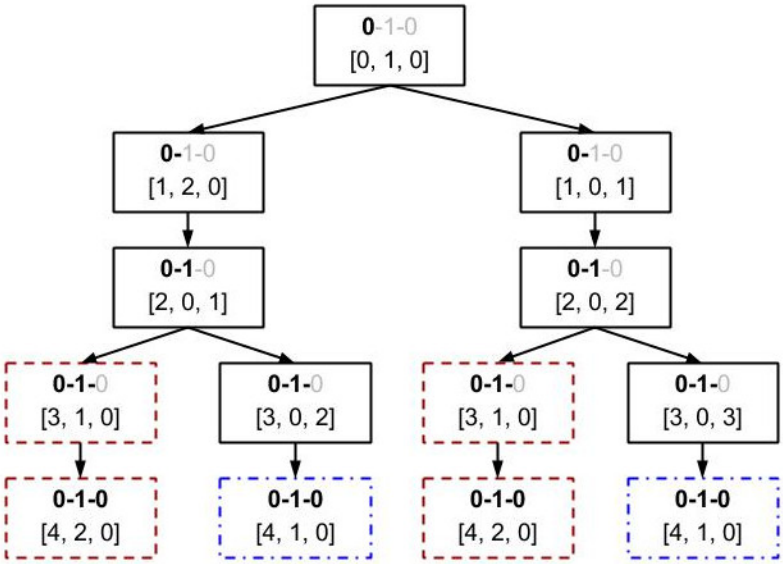

Figura 1. Árvore de recorrência da solução DP para a mensagem "0-1-0". A segunda linha representa [*i, c0, c1*], onde *i* é a iteração atual e *c0* e *c1* são os contadores de '0s' e '1s' respectivamente. Os subproblemas em vermelho se sobrepõem, assim como os dois em azul.

A versão *top-down* é muito semelhante ao algoritmo de força bruta, porém ela utiliza uma tabela tridimensional $t[0..msg.lenth, 0..3, 0..5]$ para memorizar os resultados dos subproblemas computados. Assim, antes de se chamar recursivamente, o algoritmo verifica na tabela se o valor já foi computado, evitando o recálculo.

Já a versão *bottom-up* parte do princípio da otimalidade dos subproblemas e inicia a computação a partir do último caractere da mensagem. Ele vai adicionando novos caracteres em sequência decrescente até que a mensagem inteira seja computada. Para cada caractere ele computa o resultado parcial considerando cada um dos possíveis valores $c0$ e $c1$ podem assumir.

No algoritmo *top-down* apenas os resultados dos subproblemas que fazem parte da solução são registrados na tabela t. Já na versão *bottom-up*, a tabela inteira é preenchida, mesmo com resultados de subproblemas que não fazem parte da solução ótima.

O algoritmo *bottom-up* preenche completamente uma tabela de dimensões $(n \times 4 \times 6)$, por isso seu tempo execução é proporcional a $24n$, onde n é o tamanho da mensagem. Assim, podemos concluir que o algoritmo *bottom-up* tem complexidade de tempo igual a $\theta(n)$. Já o *top-down* é no pior caso $O(n)$.

4. Avaliação Experimental

Nesta seção são apresentados e analisados os resultados de cada paradigma dos problemas. Todos os testes foram realizados no sistema operacional Ubuntu 14.04, rodando num notebook com processador 3rd Generation Intel® Core™ i7-3630QM (2.40GHz 6MB Cache), 8GB RAM, HHD 1 TB S-ATA (5,400 rpm). Para cada teste, foram feitas 10 execuções e retirado a média.

Algoritmo 3: Algoritmo simplificado bottom-up

```
Entrada: Mensagem msg
Saída: True, False ou Both
Seja n = msg.length;
Seja t[0 .. n, 0 .. 3, 0 .. 5] uma tabela de memorização;
for c0=0 to 3 do
    for c1=0 to 5 do
        if c0 == 3 or c1 == 5 then
            table[n][c0][c1] = TRUE;
        else
            table[n][c0][c1] = FALSE;
for i=n-1 to 0 do
    for c0=0 to 3 do
        for c1=0 to 5 do
            if c0 == 3 or c1 == 5 then
                table[i, c0, c1] = TRUE;
            else
                if msg[i] == '0' then
                    table[i, c0, c1] = table[i + 1, c0 + 1, 0];
                else
                    if msg[i] == '1' then
                        table[i, c0, c1] = table[i + 1, 0, c1 + 1];
                    else
                        r0 = table[i + 1, c0 + 1, 0];
                        r1 = table[i + 1, 0, c1 + 1];
                        if r0 != r1 then
                            table[i, c0, c1] = BOTH;
                        else
                            table[i, c0, c1] = r0;
retorna t[0, 0, 0];
```

4.1. Qualidade do algoritmo guloso

A Tabela 1 apresenta o percentual de acerto do algoritmo guloso para uma entrada com mais de 14 milhões de mensagens.

Tabela 1. Percentual de acerto do algoritmo guloso para uma entrada com 14.348.907 de mensagens.

True	False	Both	Total
66,96%	100,00%	96,07%	**86,45%**

Conforme podemos observar na tabela, o algoritmo guloso apresentou 86,4% de acerto. Sendo que sua melhor classificação foi sobre mensagens cuja saída é *false*, com 100% de acerto. O algoritmo de programação dinâmica, que é ótimo, foi utilizado para classificar corretamente as mensagens para a comparação.

4.2. Tempo de execução dos algoritmos

Nesta seção os algoritmos são avaliados empiricamente em relação ao tempo de execução.

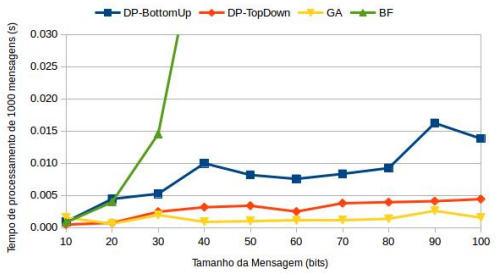

Figura 2. Tempo de processamento dos algoritmos por diferentes tamanhos de mensagens. Cada resultado corresponde ao processamento de mil mensagens.

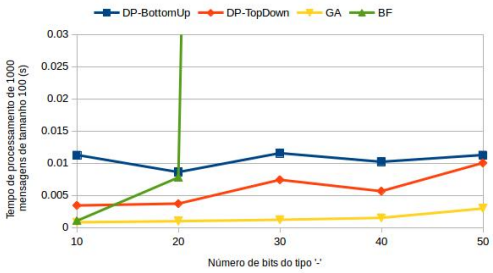

Figura 3. Tempo de processamento dos algoritmos por diferentes tamanhos de mensagens. Cada resultado corresponde ao processamento de mil mensagens.

4.2.1. Tamanho das mensagens

O gráfico da Figura 2 apresenta o tempo de execução dos algoritmos no processamento de mil mensagens para cada tamanho. As mensagens foram geradas aleatoriamente com uma probabilidade igual para cada tipo de bit por posição.

Conforme podemos observar no gráfico, os algoritmos *GA* e *DP-TopDown* e *DP-BottomUp* apresentaram tempo de execuções linearmente proporcionais ao tamanho das mensagens. Já o algoritmo de força bruta teve um crescimento exponencial chegando a gastar 1.383 segundos para processar mil mensagens de 90 bits.

Outra observação interessante é que o *DP-TopDown* apresentou tempos melhores que a versão *Bottom-Up*. Isso pode ser explicado pelo fato da tabela de memorização ser esparsa, já que o *Top-Down* não a preenche inteiramente, diferentemente do *Bottom-Up*.

4.2.2. Número de bits do tipo '-'

O gráfico da Figura 3 apresenta o tempo de execução dos algoritmos no processamento de mil mensagens com número variado de bits do tipo '-'.

Conforme é apresentado no gráfico, o algoritmo BF apresentou um tempo de execução exponencialmente proporcional ao número de bits do tipo '-'. Isso ocorre por causa das duas chamadas recursivas que o algoritmo faz para cada '-' encontrado. Já os demais algoritmos, apresentaram tempo de execução praticamente constante.

5. Conclusão

Neste trabalho foi resolvido um problema computacional utilizando três paradigmas diferentes de programação: *força bruta*, *algoritmo guloso* e *programação dinâmica*. Sendo

que para a estratégia de programação dinâmica foram implementadas duas versões: uma *top-down* e a outra *bottom-up*.

Com a exceção do algoritmo de força bruta que é exponencial, os demais possuem complexidade de tempo linear em relação ao tamanho da mensagem, conforme analisado teoricamente e demonstrado nos testes. O algoritmo com estratégia gulosa não é ótimo, mas apresentou resultados muito bons, com 86% de acerto em relação ao ótimo. Os testes também mostraram que somente o algoritmo de força bruta é sensível ao número de bits do tipo '-' na mensagem.