Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Projeto e Análise de Algoritmos
3º Trabalho Prático - Paradigmas de Programação

A Spy History

Alberto Hideki Ueda

Orientador: Berthier Ribeiro Neto

Belo Horizonte
14 de novembro de 2014

1 Descrição do problema

Dada um mensagem composta de *bits* '0' e '1' e possíveis erros de transmissão, representados pelo símbolo '-', determinar se existe ou não um comando de controle na mensagem. Um erro de transmissão pode ser tanto um '0' quanto um '1'. Um comando de controle ou é uma sequência "000"ou uma sequência "11111". A resposta deve ser uma das seguintes:

$$Resposta = \begin{cases} true, \text{ se a mensagem com certeza contém um comando de controle;} \\ \\ false, \text{ se a mensagem com certeza não contém um comando;} \\ \\ both, \text{ se a mensagem pode ou não conter um comando, dependendo do que os erros significarem.} \end{cases}$$

2 Modelagem

A modelagem do problema é simples. Dada uma sequência de caracteres do conjunto $\{0, 1, -\}$ com tamanho máximo de 100 caracteres, determinar se tal sequência contém a subsequência "000" ou "11111"; ou então se é possível substituir os caracteres '-' da sequência por *bits* '0' e '1' de forma que exista alguma destas subsequências.

Há o caso particular em que mesmo que uma mensagem possua um ou mais caracteres '-', todas as formas de substituição possíveis contém subsequências de controle. Por exemplo, na mensagem "1111-00" todas as possíveis mensagens originais ("1111100" e "1111000") contém um comando de controle. Neste e nos demais casos em que este cenário se aplica, naturalmente a resposta do algoritmo deve ser *true*.

Se há pelo menos uma possibilidade da mensagem não conter tais subsequências **E** pelo uma possibilidade dela conter, a resposta deve ser *both*. Caso não haja nenhuma possibilidade da cadeia de caracteres conter um comando de controle, a resposta deve ser $false$.

A seguir serão descritos os três algoritmos propostos para lidar com o problema, cada um seguindo um paradigma de programação diferente.

3 Algoritmo de Força Bruta

O algoritmo de força bruta proposto tem como principal objetivo testar todas as possibilidades de mensagem, dado um número finito de caracteres desconhecidos. Para diminuir a complexidade de tempo e memória para alguns casos específicos, foram acrescentadas ao algoritmo algumas podas (*prunes*) que evitam a necessidade de cálculos para cenários em que é trivial descobrir a solução. Porém, no pior caso, o algoritmo ainda testará cada solução, uma a uma, para chegar a resposta correta.

Para verificar cada possibilidade de mensagem foi implementado um algoritmo recursivo que, dado um caractere desconhecido, verifica todas as mensagens possíveis caso tal caractere seja '0' (substituindo tal caractere por '0') e também todas as mensagens possíveis caso este caractere seja '1' (instanciando uma nova mensagem, desta vez com o caractere igual a '1').

As duas respostas são então comparadas e combinadas, de forma que a resposta correta para a mensagem até aquele caractere passe a ser conhecida. Por exemplo, se descobrimos que tanto substituindo por '0' quanto por '1' a resposta é $true$, podemos afirmar com segurança que a resposta final também é $true$. Na seção "Algoritmo de Programação Dinâmica"há mais detalhes sobre esta combinação de repostas.

Para melhorar o desempenho em alguns casos específicos, foram implementadas as seguintes podas:

1. Se a mensagem original já possui um comando de controle ('000′ ou '11111′), devolve-se a resposta $true$;

2. Se a mensagem original não possui caracteres desconhecidos e não se enquadra no caso anterior, devolve-se $false$;

3. Se a mensagem original tem tamanho maior que 2, não possui '0' nem possui '1' e não se enquadra nos casos anteriores, devolve-se *both*. Esta poda melhora o desempenho para casos como 100 caracteres '_′;

4. Se para alguma das possibilidades (substituição por '0' ou por substituição por '1') a resposta é *both*, não é preciso realizar a outra substituição. A resposta combinada para aquele trecho de mensagem será *both*, pois existe pelo menos uma possibilidade da resposta ser $false$ e também uma de ser $true$.

Se nenhuma das podas se aplicar a mensagem, o algoritmo verifica todas as possibilidades de mensagens.

Pseudo-código 1: Algoritmo de Força Bruta

```
string forca_bruta(string mensagem)
{
    se a mensagem for prevista em uma poda {
        determina a resposta com base na poda;
        devolve a resposta;
    }
    senao {
        verifica_cada_mensagem_possivel(mensagem);
        devolve a resposta final obtida;
    }
}

string verifica_cada_mensagem_possivel(string mensagem);
{
    se existe caractere '-' na mensagem {
        verifica_cada_mensagem_possivel(mensagem_com_1);
        verifica_cada_mensagem_possivel(mensagem_com_0);
        combina as duas respostas e devolve uma unica;
    }
    senao {
        verifica se existe comando (mensagem);
        devolve resposta;
    }
}
```

3.1 Complexidade de Espaço

Considerando que no pior caso todas as mensagens possíveis são geradas e testadas, a complexidade de espaço é $O(2^n)$, onde n é o número de caracteres desconhecidos ('-'). Este caso ocorre quando nenhuma das podas é realizada e é necessário analisar a mensagem até o final para se chegar a resposta.

3.2 Complexidade de Tempo

No pior caso, todas as possibilidades de mensagem são testadas. Neste cenário, há 2^n verificações de mensagens, com n o número de '-'s. Dada

uma mensagem sem caracteres desconhecidos, o custo de tempo de uma verificação (busca por comando) é linear em relação ao tamanho da mensagem. Portanto, a complexidade de tempo é $O(2^n \times n)$.

3.3 Análise Experimental

4 Algoritmo Guloso

O algoritmo guloso decide qual *bit* deve substituir o caractere desconhecido e não volta atrás em sua decisão. Tal escolha é baseada na vizinhança deste caractere. Os principais critérios que o algoritmo utiliza para a tomada de decisão são os seguintes:

1. Se os últimos 2 caracteres (à esquerda) são '0', substitui o '-' por '0';
2. Se os últimos 4 caracteres (à esquerda) são '1', substitui o '-' por '1';
3. Se o primeiro caractere conhecido à esquerda e o primeiro conhecido à direita forem iguais, este será o caractere escolhido para substituição. Por exemplo: '0-0' $\rightarrow$ '000'.
4. Se não há um caractere conhecido na janela à esquerda mas há à direita, ou se não há à direita mas há à esquerda, utiliza o símbolo do lado que existe. Por exemplo: '-10' $\rightarrow$ '110';
5. Se o caractere desconhecido não se encaixar em nenhum dos casos anteriores, faz a substituição por '0'.

O algoritmo analisa cada caractere um a um, do início até o fim da mensagem, escolhendo os *bits* para todos os desconhecidos. Em cada passo verifica-se se um comando de controle foi encontrado. Se foi encontrado e em tal comando não houve a necessidade de substituição, devolve *true*. Se foi encontrado mas houve substituição, guarda a informação de que a resposta é pelo menos *both* e continua procurando por uma resposta *true*. Se não encontrar nem *both* nem *true* e a mensagem já foi analisada por completo, devolve $false$.

Pseudo-código 2: Algoritmo Guloso

```
1  string guloso(string mensagem)
2  {
3      para cada caractere do indice i da mensagem {
4          se o caractere i = '-' {
5              observa vizinhanca a esquerda e a direita;
6              decide qual bit deve ser colocado no lugar de '-';
7              verifica comando e guarda a resposta;
8          } senao {
9              verifica se existe um comando ate o caractere i;
10             se existir o comando sem '-', devolve TRUE;
11             se existir o comando com '-', guarda a resposta;
12         }
13         ++i;
14     }
15
16     se encontrou BOTH como resposta {
17         devolve BOTH;
18     } senao {
19         devolve FALSE;
20     }
21 }
```

4.1 Complexidade de Espaço

Em relação ao uso de memória, temos complexidade $O(n)$, sendo n o tamanho da mensagem original. Podemos visualizar o algoritmo como uma varredura simples na mensagem original, em que a cada passo são analisados trechos de mesmo tamanho, do início ao fim da mensagem. Após uma decisão tomada (um *bit* escolhido), não há uma reavaliação desta decisão. Desta forma, não há a necessidade de utilizar mais memória para o caractere antes desconhecido.

4.2 Complexidade de Tempo

Dando continuidade à análise do algoritmo como uma varredura, a cada passo há um número fixo v_{cons} de verificações que serão realizadas. O custo de tempo de cada verificação é linear em relação ao tamanho da mensagem que está sendo verificada. Este tamanho é $O(n)$, sendo n o tamanho da

mensagem original. Logo, a complexidade de tempo do algoritmo guloso é $O(n \times v_{cons} \times n) = O(n^2)$.

4.3 Análise Experimental

5 Algoritmo com Programação Dinâmica

O algoritmo de PD adotado também utiliza a ideia de combinar respostas de subproblemas (algoritmo de força bruta). Mas desta vez são utilizados também dois contadores, um para 0s e outro para 1s. O sub-problema é encontrar a resposta para um determinado trecho da mensagem, dados os estados atuais de ambos os contadores.

Cada resposta é armazenada em uma matriz de três dimensões ([índice do último caractere do trecho] X [estado atual do contador de 1s] X [estado atual do contador de 0s]).

Se for necessário resolver um sub-problema cuja resposta já esteja na matriz, basta consultar a resposta e utilizá-la. Caso a solução para este sub-problema ainda não tenha sido definida, o algoritmo realiza o cálculo de tal resposta com base nos resultados de outros sub-problemas.

Sejam:

- mensagem: a instância do problema;
- n: tamanho da mensagem;
- i : índice do caractere que está sendo analisado. A sub-mensagem [0..i-1] já foi analisada;
- c1 : contador de 1's;
- c0 : contador de 0's;
- opt : a resposta para o subproblema de acordo com os parâmetros. Um dos valores do conjunto $\{true, false, both\}$.

A função de recorrência utilizada é a seguinte:

$$opt(i, c1, c0) = \begin{cases} // \text{ condições de parada} \\ true, \text{ se c1 = 5 ou se c0 = 3;} \\ false, \text{ se i = n;} \\ \\ // \text{ recursão. Casos '1' ou '0'} \\ opt \text{ (i, c1+1, 0) , se mensagem[i] = '1';} \\ opt \text{ (i, 0, c0+1) , se mensagem[i] = '0';} \\ \\ // \text{ recursão. Caso '-'} \\ // \text{ compara as respostas das duas possibilidades} \\ // \text{ '-' = '0' e '-' = '1'} \\ both, \text{ se } opt \text{ (i, c1+1, 0) != } opt \text{ (i, 0, c0+1);} \\ \\ // \text{ se as respostas forem as mesmas, escolhe uma delas} \\ opt \text{ (i, 0, c0+1), caso contrário.} \end{cases}$$

Dessa forma, a combinação de respostas é feita da seguinte maneira: sejam r_0 a resposta correta considerando a substituição do caractere desconhecido por '0' e r_1 a resposta obtida com a substuição pelo caractere '1'.

- Se uma das respostas (r_0, r_1 ou ambas) for *both*, a resposta combinada é *both*, pois sabemos que existe tanto a possibilidade da mensagem conter um comando quanto a de não conter um comando;
- Se ambas as respostas (r_0 e r_1) forem $true$, a resposta combinada é $true$, pois em todas as possibilidades de mensagem existe comando;
- Se ambas as respostas (r_0 e r_1) forem $false$, a resposta combinada é $false$, pois em todas as possibilidades de mensagem não existe comando.

5.1 Complexidade de Espaço

Apesar de ser recursivo, o ordem de complexidade de espaço é limitada pelo tamanho da matriz de resultados. Todos os valores possíveis para resposta estão nesta matriz e uma vez que uma resposta é calculada, não há recálculos ou uso de memória adicional. Portanto, para o caso geral, a complexidade de espaço é $O(n)$, n = tamanho máximo da palavra, pois o tamanho

da matriz é $(n \times 3 \times 5)$ que é $O(n)$. No caso deste trabalho, n é fornecido e é igual a 100. Assim, a complexidade de espaço torna-se constante, $O(1)$.

5.2 Complexidade de Tempo

Ao contrário dos algoritmos de força bruta e guloso, neste algoritmo de programação dinâmica não são realizadas verificações em relação ao **texto das mensagens** e suas vizinhanças, mas apenas de **parâmetros** que definem os diversos cenários possíveis. Tais parâmetros são mapaeados em uma matriz e as respostas são calculadas e armazenadas conforme a demanda (abordagem *top-down*).

Toda solução de um subproblema ou é calculada em um tempo constante (condições de parada) ou são combinações de outras respostas da própria matriz. Uma vez que uma resposta é definida, não há recálculo, portanto cada elemento da matriz é calculado apenas uma vez. Cada combinação de respostas é realizada em um tempo c_{cons} constante. Além disso, apesar de ser uma matriz de três dimensões, duas delas são constantes e portanto o número de respostas é linear em relação ao tamanho máximo n da palavra. Desta forma, o custo máximo de tempo para os cálculos é $c_{cons} \times n$. Logo, o algoritmo possui complexidade de tempo $O(c_{cons} \times n) = O(n)$.

5.3 Análise Experimental

6 Conclusão

A descoberta de que o problema pode ser solucionado por um algoritmo de complexidade de tempo linear foi surpreendente. O algoritmo de programação dinâmica, apesar de utilizar mais espaço para sua execução, é muito mais veloz que o algoritmo guloso e naturalmente que o de força-bruta também. Graças à divisão do problema original em subproblemas que se repetem e que podem ter a resposta memoizada, o desepenho de tal algoritmo mostrou resultados bastante satisfatórios e por si só justifica o estudo e pesquisa empreendidos nesta área de conhecimento.

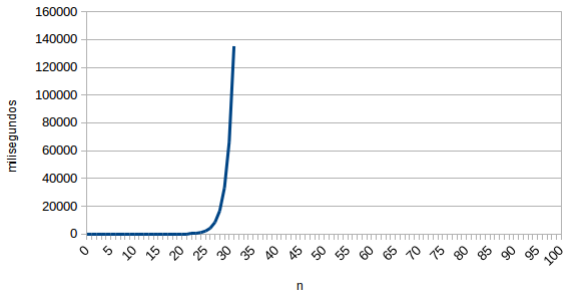

Figura 1

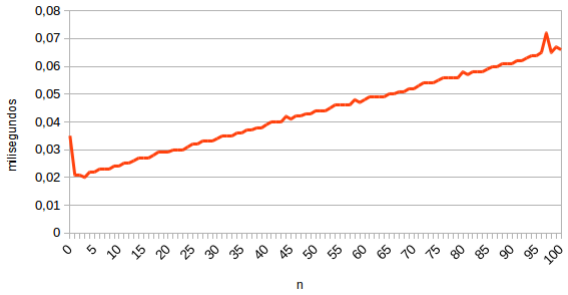

Figura 2

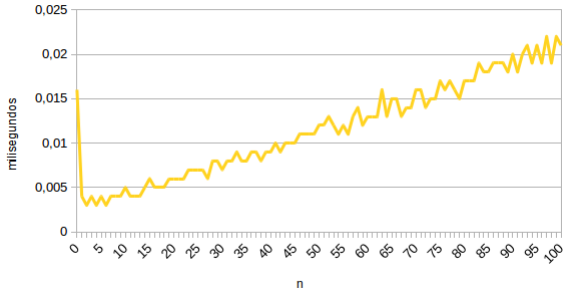

Figura 3

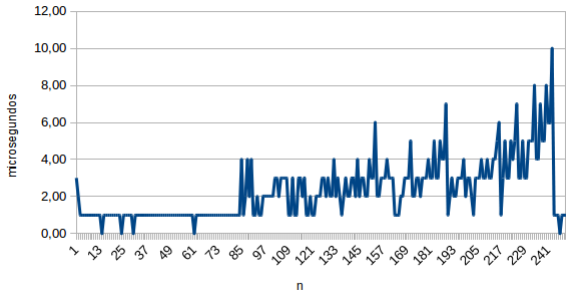

Figura 4

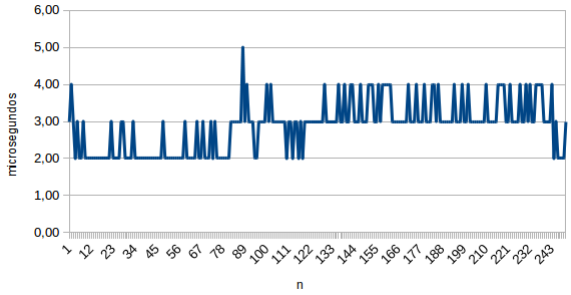

Figura 5

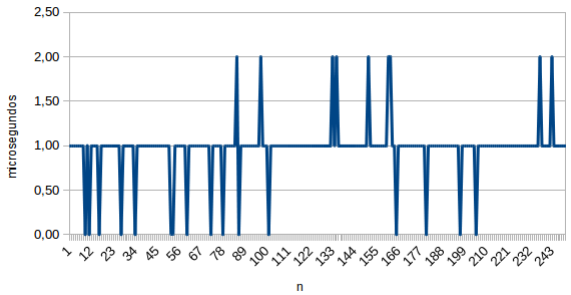

Figura 6

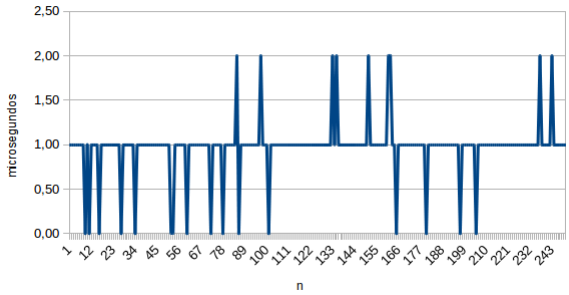

Figura 7

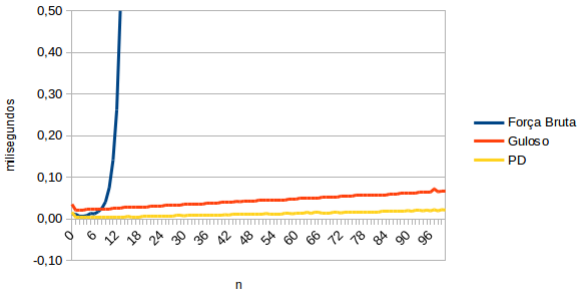

Figura 8