TP3: MyTeX

Vitor Cláudio Chaves de Aguiar

29 de junho de 2016

1 Introdução

O objetivo desse trabalho prático é resolver o problema de justificação de um texto. Dado um texto inicial sem mais de um espaço em branco seguido e sem quebras de linha, estas deverão ser inseridas com o objetivo do texto ficar justificado no final. Porém a resolução do problema é dada por diferentes paradigmas de programação que podem ser definidos por visões ou abordagens diferentes na hora de criar um algoritmo. Os paradigmas implementados para esse trabalho foi os: força bruta, guloso, programação dinâmica.

A entrada de dados consiste em um arquivo contendo o número de caracteres máximo permitido por linha, o número de linhas máximo permitido por página, o expoente da função de custo, o fator da função de custo e o texto no qual será inserido quebras de linha para no final ser justificado. Na execução do programa será passado os parâmetros de qual algoritmo será executado e quais serão os arquivos de entrada e saída.

A função de custo dita anteriormente é utilizada para verificar o aproveitamento do espaço máximo determinado por linha dada uma configuração de um texto. A sua fórmula está descrita abaixo:

$$k(H - |l|)^X + \sum_{\forall l_i \in l} k(L - length(l_i))^X$$

Figura 1: Fórmula de custo

A solução apresentada consiste em verificar aonde serão inseridas as quebras de linhas nos três algoritmos diferentes utilizando a fórmula de custo aonde o valor de `length(Li)` e `l` serão constantemente alterados no cálculo do custo devido ao fato de serem o custo da linha e o número de linhas produzidas.

2 Solução do problema

Inicialmente, após o arquivo de entrada ser computado, é criado um vetor de offsets com a finalidade de guardar a posição de ínicio de cada palavra do vetor do texto da entrada. Feito isso, é possível saber qual é o custo (número de caracteres) de inserir cada palavra em uma solução. Para a criação desse vetor de offsets foi necessário extrair a informação de quantas palavras o texto possui e, para isso, se contou o número de espaços entre as palavras e somou + 1 unidade. É interessante resaltar nesse trabalho que ao usar o termo "infinito"o algoritmo está utilizando do valor máximo que um inteiro na linguagem C pode armazenar.

2.1 Força Bruta

Algorithm 1: Força Bruta

```
1  if linhas >maximo then
2  |  return infinito
3  end
4  if i = numeroPalavras then
5  |  return caso base
6  end
7  minJ ← infinito
8  valor ← infinito
9  for j ← i + 1 to numeroPalavras do
10 |  calcula o custo da linha
11 |  verifica se o custo é infinito
12 |  valor ← min(valor, custoDaLinha + ForcaBruta(j), j, minJ)
13 end
14 parent[i] ← minJ
15 return valor
```

Este paradigma consiste em testar todas as possíveis inserções de quebras de linha e verificar qual são as melhores. Para a execução desse algoritmo são realizadas diversas chamadas recursivas com a finalidade de chegar na solução ótima. Na linha 1 é possível observar uma verificação se em algum momento da execução o número de linhas formadas passou do limite estabelecido.

Na linha 9 é executado um loop que tem a finalidade de verificar os custos da posição inicial da linha que está até o final do texto. Considerando que o problema é dividido em subproblemas, esse início da linha vai variar em cada chamada recursiva.

Para se verificar o custo de determinado ponto inicial e final do texto é necessário o uso da função `badness` que calcula o número de caracteres nessa faixa, verifica se o valor obtido é maior que o máximo permitido por linha, se não for ele calcula a função de custo para aquela faixa. Afim de reduzir o custo do algoritmo, foi inserido uma validação se o valor retornado da função `badness` é infinito, pois se isso for verdade não há necessidade de continuar calculando as faixas para frente pois todas irão retornar infinito também.

Devido ao fato do objetivo ser escolher qual é o menor custo, uma função de mínimo é chamada a fim de determinar qual é o minimo entre o valor antigo calculado e o da próximo recursão que começará com um valor de `i` (posição inicial da linha) diferente.

Ao final da execução do algoritmo teremos qual é o custo mínimo da solução ótima de inserir quebras de linha em um texto e deixá-lo justificado considerando os limites de linhas e caracteres previamente estabelecidos. Entretanto é necessário também escrever no arquivo de saída o texto com a nova formatação. Diante disso, foi necessário a criação de um vetor `parent` que grava o índice `j` do valor mínimo calculado em cada recursão.

A utilização desse vetor `parent` para impressão funciona da seguinte forma:

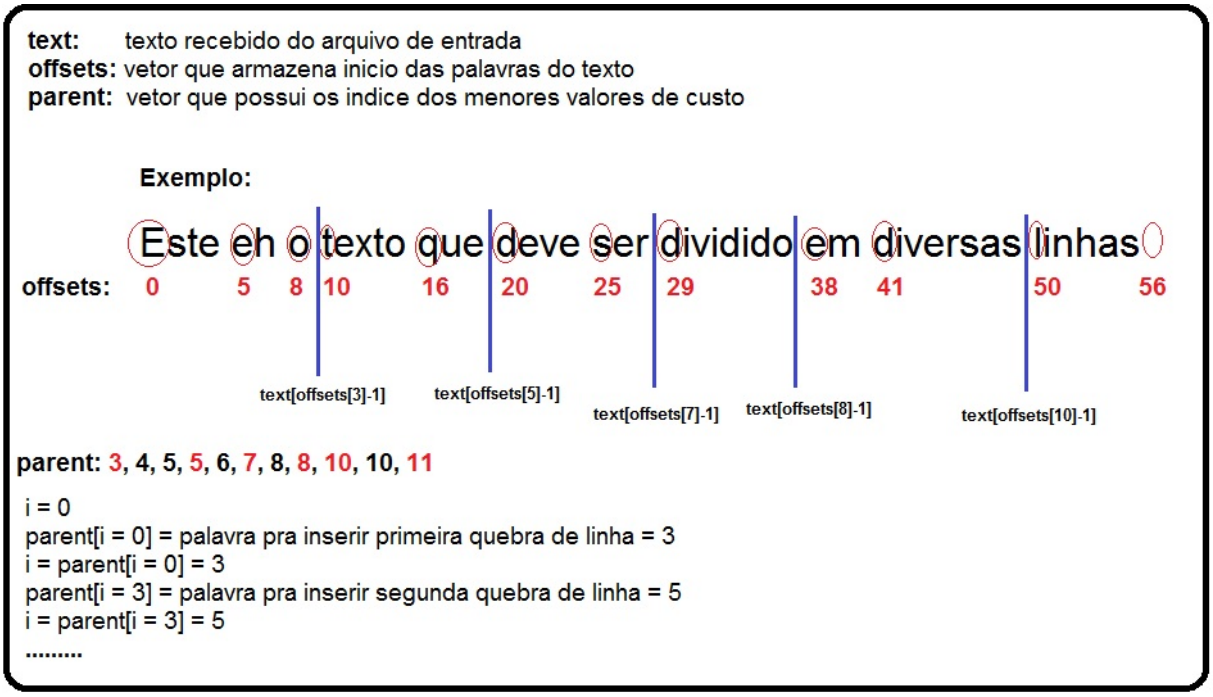

Figura 2: Utilização do vetor parent

É possível observar nessa imagem que os valores de `parent` que determinam o índice do vetor de offsets que vai conter quebra de linha são indexados à partir do conteúdo de seu anterior. Apenas os escreitos em vermelho indicam as quebras de linha. Devido ao offset informar o inicio de uma palavra, é necessário subtrair uma unidade para a quebra ser inserida no lugar do espaço.

É necessário ressaltar que as quebras são inseridas no vetor do texto original no lugar de espaços, desta forma fica mais simples a escrita sendo necessário apenas escrever o vetor modificado.

O último valor do `parent` se diz respeito ao final do arquivo, devido ao fato de não ser necessário inserir uma quebra de linha neste local, uma validação é realizada para evitar tal ação.

2.2 Programação Dinâmica

Algorithm 2: Programação Dinâmica

```
1  if linhas >maximo then
2  |  return infinito
3  end
4  if memo[i][numeroLinhas] != −1 then
5  |  return memo[i][numeroLinhas]
6  end
7  if i = numeroPalavras then
8  |  return caso base
9  end
10 minJ ← infinito
11 valor ← infinito
12 for j ← i + 1 to numeroPalavras do
13 |  valor ← min(valor, badness(i, j) + ForcaBruta(j), j, minJ)
14 end
15 parent[i] ← minJ
16 memo[i][numeroLinhas] ← valor
17 return valor
```

Este paradigma consiste em testar todas as possíveis inserções de quebras de linha e verificar qual são as melhores. Porém, diferentemente do algoritmo de força bruta, esse paradigma não calcula subproblemas repetidos. Todo valor que é calculado também é armazenado em uma matriz indexada pela variável `i` e o `número de linhas`, devido a isso, sempre que o algoritmo precisar de calcular um subproblema que já foi calculado ele só vai retirar da matriz os dados.

É possível observar a semelhança com o força bruta no pseudocódigo acima, porém a diferença está nas linhas 4 e 16. Na linha 4 é verificado se o valor já foi calculado para evitar de calcular de novo e na linha 16 o valor calculado naquela recursão é salvo nessa matriz. Desta maneira é possível notar o altíssimo ganho de performace desse algoritmo.

A matriz `memo` é criada e inicializada com valores igual a -1 para que, no ponto de vista da execução do algoritmo, possa saber se já foi calculado algum valor para aquele subproblema ou não. Um vetor `parent` também é criado igual na estratégia do força bruta e, desta forma, o custo e o texto com as quebras de linhas inseridas são escritos no arquivo.

2.3 Guloso

Algorithm 3: Guloso

```
1  inicio ← offsets[i]
2  k ← numeroPalavras
3  if k = 0 then
4  |  return caso base
5  end
6  valor ← 0
7  for k to 0 do
8  |  if podeInserirPalavra then
9  |  |  insere
10 |  |  k ← k − 1
11 |  |  i ← i + 1
12 |  end
13 |  else
14 |  |  texto[valor + inicio] ← quebraDeLinha
15 |  |  break
16 |  end
17 end
18 numeroLinhas ← numeroLinhas + 1
19 valor ← calculaCusto
20 return valor + Guloso(i, k, numeroLinhas)
```

Devido ao fato do paradigma guloso ser uma heurística, não é garantido uma solução ótima. A estratégia abordada acima adiciona as palavras na solução até que o máximo de caracteres permitidos não permita adicionar mais. Quando chegar nessa situação, é adicionado uma quebra de linha após a última palavra adicionada na solução, porém essa quebra é inserida no texto original igual nos outros dois algoritmos permitindo a escrita no arquivo de uma maneira simplificada.

O loop na linha 7 executa até não ter mais palavras ou até que não se possa adicionar mais palavras na solução devido ao limite de caracteres. A verificação é feita somando os caracteres da palavra e vendo se vai passar do limite, se não passar a palavra é inserida na linha se não é inserido uma quebra de linha.

Nesta solução não é necessário um vetor `parent` pois o local de inserção das quebras de linhas são conhecidos a cada chamada recursiva.

3 Análise de complexidade

3.1 Tempo

- A função `createArrayOffset` percorre os caracteres do texto uma vez para verificar o número de palavras e outra para verificar o início das palavras do texto inicializando o vetor de offsets. Portanto, a complexidade dessa função será O(2n) = **O(n)** sendo `n` o número de caracteres do texto.

- A função `createMemoMatrix` cria uma matriz p x h, sendo `p` o número de palavras do texto e `h` o npumero máximo de linhas determinado. Portanto a complexidade dessa função será **O(p*h)**.

- A função `badness` não possui nenhum loop, ela só executa operações condicionais, atribuição de valores e operações matemáticas, portanto a complexidade dessa função será **O(1)**.

- Na função `bruteForce` ocorre uma série de chamadas recursivas, diante disso é necessário avaliar sua complexidade a partir de sua equação de recorrência:

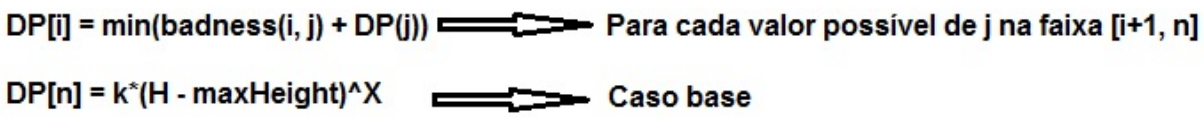

Figura 3: Recorrência

É possível observar com a imagem acima que `n-i` chamadas recursivas são executadas para cada possível linha (valor de i). Portando todas as possíveis combinações de palavras em cada linha é testado ocasionando em uma complexidade $O(2^n)$.

- A função `dynamicProgramming` utiliza da mesma equação de recorrência e, portanto, possui um algoritmo parecido. Portanto a estratégia usada nesse paradigma denominado `memoization`, faz com que todos os valores sejam calculados apenas uma vez e, com isso, quando o cálculo for executado novamente o custo de obter o resultado será **O(1)**.

Para guardar os valores de custos é criado uma matriz indexada pelo numero de palavras e o valor da linha. Para preencher essa matriz é necessário `p` chamadas recursivas sendo p o número de palavras **O(p)**. Diante disso, qualquer outra chamada recursiva vai acessar esse vetor e adquirir a informação do custo com **O(1)**.

Devido ao pior caso do cálculo do custo ser **O(p)**, a complexidade total dessa função será O(p*p) ou $O(p^2)$ para calcular o custo e preecher a matriz.

- A função `greedy` é mais simples de ser analisada. Mesmo ela sendo implementada recursivamente, as chamadas recursivas estão limitadas pelo número de palavras do texto. A estratégia abordada foi analisar cada linha separadamente colocando o máximo de palavras possível. Desta forma, a cada chamada recursiva o número de palavras reduz e, no pior caso, essa redução é feita de um em um, ou seja, uma palavra por linha ocasionando em `p` chamadas recursivas sendo `p` o número de palavras. Portanto a complexidade será **O(p)**.

Devido a essas análises é possível observar que a complexidade do programa total no pior caso será quando optar em resolver por força bruta, devido a isso, a complexidade total será de $O(2^n)$.

3.2 Espaço

No início do programa é criado um vetor com o texto possuindo uma complexidade **O(n)** sendo `n` o número de caracteres do texto. Logo após é alocado memória para o número de palavras e para o vetor de offsets possuindo uma complexidade de **O(p)** sendo p o número de palavras. No pior caso de complexidde de espaço a operação escolhida será programação dinâmica sendo necessário a criação de um vetor `parent` e um vetor `memo` com respectivas complexidades de **O(p)** e **O(p*h)**.

Como reflexo disso, a complexidade de espaço total do programa é definida por **O(n)**.

4 Análise Experimental

4.1 Metodologia

Para realizar a análise experimental cada instância foi testada 3 vezes em uma máquina Intel Core i5 2,67GHz de 4GB de memória RAM. Para medição do tempo de execução foi utilizado o comando `time` no linux e o cálculo final do tempo foi obtido pela média dos 3 testes.

4.2 Heurística Gulosa

Diante do fato que o algoritmo guloso é uma heurística, não é possível garantir uma solução ótima dele. Para provar e explicar porque a estratégia utilizada do algoritmo guloso desse trabalho não é ótima foi utilizado esse exemplo:

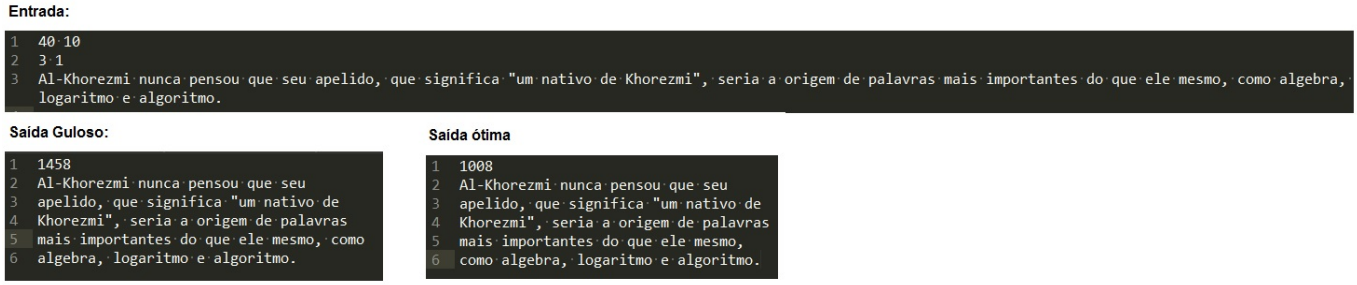

Figura 4: Exemplo Guloso

É possível observar desse exemplo extraído do toy que o algoritmo guloso teve um custo acima do algoritmo ótimo e isso pode ser explicado pois o guloso não se preocupa com o problema inteiro de uma vez. O algoritmo guloso escolhe qual é a melhor organização de palavras pra cada linha por vez. O que pode acontecer é que as vezes é necessário ter um custo pior em uma linha para ter um custo melhor na outra, porém esse custo na outra linha pode ser bom ao ponto de sobrepor a perda que teve na anterior.

Esse comportamento pode ser observado mais claramente na penúltima linha das saídas. No algoritmo guloso ele coloca na penúltima linha o máximo de palavras possíveis, como reflexo disso a última linha não aproveita ótimamente do espaço delimitado e fica com muito espaço em branco, perdendo o propósito de uma justificação de texto e aumentando o valor na função de custo. Esse comportamento na solução ótima é diferente, deixando mais proporcional a distribuição de palavras em cada linha não sobrando muitos espaços à direita.

4.3 Experimentos

Paradigma	Máximo de caracteres por linha	Máximo de linhas	Expoente da função	Fator da função	Custo	Tempo
Força Bruta	15	20	3	2	2856	0,27s
Guloso	15	20	3	2	5016	0,007s
Dinâmico	15	20	3	2	2856	0,007s
Força Bruta	17	20	3	2	4736	3,861s
Guloso	17	20	3	2	5684	0,015s
Dinâmico	17	20	3	2	4736	0,010s
Força Bruta	18	20	3	2	7170	7,725s
Guloso	18	20	3	2	9054	0,007s
Dinâmico	18	20	3	2	7170	0,008s

Figura 5: Variando máximo de caracteres por linha

A partir desse experimento é possível observar que o tempo aumenta devido ao número de palavras que podem ser acrescentadas com o aumento do número máximo de caracteres por linha. Com isso a faixa de valores de `j` pode aumentar acarretanto em novas chamadas recursivas. O custo tambem aumenta devido a operação envolver uma subtração do tamanho máximo com o número de caracteres de uma palavra.

Paradigma	Máximo de caracteres por linha	Máximo de linhas	Expoente da função	Fator da função	Custo	Tempo
Força Bruta	15	20	3	2	2856	0,27s
Guloso	15	20	3	2	5016	0,007s
Dinâmico	15	20	3	2	2856	0,007s
Força Bruta	15	25	3	2	3286	2,992s
Guloso	15	25	3	2	5446	0,006s
Dinâmico	15	25	3	2	3286	0,008s
Força Bruta	15	30	3	2	5516	7,698s
Guloso	15	30	3	2	7676	0,010s
Dinâmico	15	30	3	2	5516	0,008s

Figura 6: Variando número máximo de linhas

A partir desse experimento é possível observar que o tempo cresce a medida que o número máximo de linhas cresce. Isso se deve ao fato de ter que executar mais chamadas recursivas com um número de linhas maior. O custo também será maior pois no caso base a operação H-l (segundo a imagem na introdução do trabalho) será um número maior.

Paradigma	Máximo de caracteres por linha	Máximo de linhas	Expoente da função	Fator da função	Custo	Tempo
Força Bruta	15	20	1	2	120	0,356s
Guloso	15	20	1	2	120	0,010s
Dinâmico	15	20	1	2	120	0,008s
Força Bruta	15	20	5	2	117720	0,291s
Guloso	15	20	5	2	353880	0,007s
Dinâmico	15	20	5	2	117720	0,007s
Força Bruta	15	20	8	2	46574164	0,325s
Guloso	15	20	8	2	270888592	0,007s
Dinâmico	15	20	8	2	46574164	0,008s

Figura 7: Variando expoente da função

A partir desse experimento é possível observar o valor do custo aumentar drásticamente. Diante disso, é possível analisar que uma função de custo com expoente alto tem um índice de tolerância bem baixo quanto ao aproveitamento da linha, causando um valor de custo elevado.

Paradigma	Máximo de caracteres por linha	Máximo de linhas	Expoente da função	Fator da função	Custo	Tempo
Força Bruta	15	20	3	2	120	0,356s
Guloso	15	20	3	2	120	0,010s
Dinâmico	15	20	3	2	120	0,008s
Força Bruta	15	20	3	10	14280	0,417s
Guloso	15	20	3	10	25080	0,008s
Dinâmico	15	20	3	10	14280	0,010s
Força Bruta	15	20	3	50	71400	0,540s
Guloso	15	20	3	50	125400	0,006s
Dinâmico	15	20	3	50	71400	0,010s

Figura 8: Variando fator da função

A partir desse experimento é possível analisar o valor do custo que sobe linearmente a medida que o valor do fator da função aumenta. Isso é devido ao fato que na função de custo ele está sempre multiplicando outros fatores.

5 Conclusão

Neste trabalho foi possível trabalhar com três diferentes paradigmas de programação e observar o comportamento de cada um na solução de um mesmo problema. Cada um apresenta uma solução, sendo ótima ou não e apresenta um tempo de execução. Dependendo da situação, aplicar uma heurística de um algoritmo guloso pode ser vantajoso, mesmo que não entregue uma solução ótima, o tempo geralmente é bem melhor. O algoritmo dinâmico tem um tempo de execução muito bom e garante uma solução ótima, diferente do força bruta que tem um péssimo tempo de execução mesmo resultando no ótimo.