TP3: MyTeX

Giovani Amaral Barcelos

29 de junho de 2016

1 Introdução

Quando elaboramos um texto, deseja-se que sua estrutura seja harmoniosa, ou seja, as palavras estejam distribuídas nas linhas de maneira que facilite a leitura.

Dado um texto, precisamos de um método para organizá-lo de forma a buscar a uniformidade do comprimente das linhas, e o objetivo deste trabalho é implementar este método usando **Programação Dinâmica**, uma **heurística gulosa** e **Força bruta**

No nosso problema, o texto de entrada não possui quebras de linha e só podemos inserir uma quebra de linha nos espaços.

1.1 Função de custo

O ideal é que todas as linhas de um texto fiquem com o mesmo comprimento, mas em alguns casos isso só seria possível se pudéssemos separar as palavras.

Nossa tarefa é, então, maximizar a seguinte função de custo:

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

Figura 1: Função de custo a ser minimizada

Onde L é o comprimento máximo da linha, H é o número máximo de linhas, l é o conjunto de linhas em que o texto foi dividido, —l— é o número de linhas geradas, length(li) é o comprimento da linha i gerada (sem contar a quebra de linha), k é uma constante multiplicativa e X é o expoente, que serve para balancear o comprimento das linhas, ou seja, quando os espaços estão muito concentrados, o custo é maior.

Além disso, se o comprimento da linha for maior que L ou o número total de linhas for maior H, o custo será ∞.

2 Solução do problema

2.1 Heurística Gulosa

A heurística gulosa consiste em colocar palavras na linha até não caber mais. Se a adição da próxima palavra acarretar num estouro do tamanho da linha, apenas criamos uma nova linha e continuamos o processo, até que todas as palavras estejam inseridas no texto.

A heurística implementada pode ser aproximada pelo seguinte pseudo código:

Algorithm 1: Pseudo-código da heurística gulosa

```
1  saida = Array
2  for palavra in entrada do
3  |  if length(palavra) + length(linha) < comprimento_maximo then
4  |  |  linha+ = palavra
5  |  |  numeroPalavras − −
6  |  |  if numeroPalavras > 0 then
7  |  |  |  linha += ' '
8  |  |  end
9  |  end
10 |  else
11 |  |  saida+ = linha
12 |  |  linha = Array
13 |  end
14 end
15 print saida
```

Esta heurística é bem simples de ser entendida e implementada. O método guloso vê sempre a melhor solução para o momento em que ele está, ou seja, ele tenta sempre colocar palavras enquanto couberem, sem nenhuma análise do resto do texto, e é por isso que ela não é ótima. Em muitos casos, colocando uma palavra na linha pelo único motivo dela caber, acarreta em um mal aproveitamento de algumas linhas futuras.

Nesse exemplo, vemos onde a solução gulosa pode falhar:

Solução gulosa: **Custo: 4246**
A way of classifying
algorithms is by
their design
methodology or
paradigm.

Solução ótima: **Custo: 3418**
A way of classifying
algorithms is
by their design
methodology
or paradigm.

Algumas linhas, como a 2 e a 4, acabam ficando mais cheias, pois colocaram palavras pequenas que cabiam. Na solução ótima, as linhas tem um comprimento médio mais parecido. Com isso, podemos concluir que o guloso funciona melhor quando as palavras tem o tamanho parecido e tende a falhar quando o tamanho varia muito.

2.2 Força bruta

O segundo método utilizado para resolver este problema foi a clássica força bruta: testamos todas as possibilidades e utilizamos a de menor custo.

Isso nos garante um algoritmo ótimo, pois testamos todas os arranjos de texto e portanto sabemos qual(ou quais) gera o menor custo.

Algorithm 2: Pseudo-código força-bruta

Data: $parametrosCusto, i, numeroLinha$
Result: valorMinimo

```
1  // Poda:  se o número de linhas desta hipótese ultrapassa o máximo, não
       precisamos calcular o resto
2  if H < numeroLinhas then
3  |  return infinit
4  end
5  if i == quantidadePalavras then
6  |  return K * (H - numeroLinhas)^X
7  end
8  else
9  |  valor = min(for(j = i + 1 j ≤ quantidadePalavras
10 |  j++ ) valorAtual = badness(i, i) + BF(j, numeroLinhas + 1)
11 |  )
12 |  return valor
13 end
```

Onde nossa função badness retorna o curso da linha, ou seja, $k * (L - tamanhoLinha)^X$. Vale ressaltar que, cada o tamanho da linha seja maior que o permitido, ela retorna infinto.

Na linha 2, temos uma poda para nossa árvore de possibilidades: caso o número de linhas da possibilidade atual ultrapasse o máximo permitido, não precisamos calcular o resto, pois sabemos que não será válido, e portanto retornamos infinito.

Na linha 5 temos nossa condição de parada, ou seja, caso o arranjo de texto atual já tenha incluído todas as palavras, retornamos a primeira parte da função que custo, referente ao tamanho do texto em número de linhas.

Na linha 9, temos um for que vai da próxima palavra até a última e checa a possibilidade de montar uma linha com cada uma delas e faz a chamada recursiva para o resto das palavras seguintes. Pegamos assim o menor custo possível entre os testes do for e retornamos.

Desta forma, nossa força bruta calcula, de forma recursiva, a possibilidade de cada palavra começar uma linha ou não, testando todas as possibilidades(e podando as inválidas).

2.3 Programação dinâmica

A ideia da programação dinâmica é criar uma solução para subproblemas de forma que você consiga resolver o problema original.

Para tal, o **algoritmo de força bruta foi modificado** de modo a guardar os resultados dos subproblemas em dicionario, ou seja, não é necessário calcular as redundâncias, o que a torna muito mais eficiente do que a força bruta pura.

Esta técnica de guardar resultados, conhecida como memoization, onde temos uma matriz sendo as dimensões nossos parâmetros da função de DP. Nesse caso, as linhas são os I's e as colunas os NumeroLinha.

A utilização é simples:

No começo da função, possuímos o trecho:

```
1 if memo[i][numeroLinhas] ≠ −1 then
2 |  return memo[i][numeroLinhas]
```

E além disso, ao calcular um valor pela primeira vez, simplesmente salvamo-os na nossa matriz.

Fazendo isso, evitamos calcular as chamadas desta função que possuam os mesmos parâmetros, pois o resultado seria o mesmo. A checagem do -1 é para saber se já foi definido esse valor, que é setado na primeira vez que os cálculo do DP com esses valores é feito(o que seria na linha 10 do força bruta.).

3 Análise teórica do custo assintótico de tempo

Tomemos para esta análise **n** sendo o número de caracteres do texto de entrada e **P** o número de palavras.

3.1 Entrada de dados

Lemos os 4 parâmetros do arquivo de entrada e a string contendo o texto a ser justificado.

Chamamos então a função **copiaTexto** que percorre esta string e divide-a, colocando na nossa matriz de palavras. A complexidade desta função é **O(n)**.

Essa função é comum aos três métodos abaixo.

3.2 Greedy

Nosso algoritmo greedy é linear, pois passa uma vez em cada palavra da nossa matriz, tentando colocá-la na linha atual. Não é feita nenhuma checagem para o resto do texto, pois o guloso só vê a melhor solução para o momento. Para imprimir o resultado, nossa função passa uma vez em cada palavra. Portanto, a complexidade da função greedy é P + P = **O(2P)**

Como precisamos da função de entrada de dados, nosso programa greedy tem a complexidade **O(P+n)**

3.3 Força Bruta

O algoritmo de força bruta testa todas as soluções possíveis, ou seja, testa se cada palavra começa ou não uma linha. Sem fazer cálculo algum já sabemos que isso são 2^P tentativas no pior caso, pois para cada uma das T palavras temos 2 opções.

Olhando para o código, temos a seguinte equação de recorrência:

$$F(i) = \begin{cases} k * (H - numeroLinhas)^X, & \text{if } i = n \\ min(F(i+j) + badness(i,j) from j = i+1 to P), & \text{otherwise} \end{cases} \quad (1)$$

Expandido a recursão, podemos desenhar uma árvore que facilita a observação:

A primeira chamada, faz P chamadas recursivas, variando o índice j em DP(j) no intervalo [i+1, P]; Cada uma dessas chamadas faz (P - j) chamadas, e assim sucessivamente.

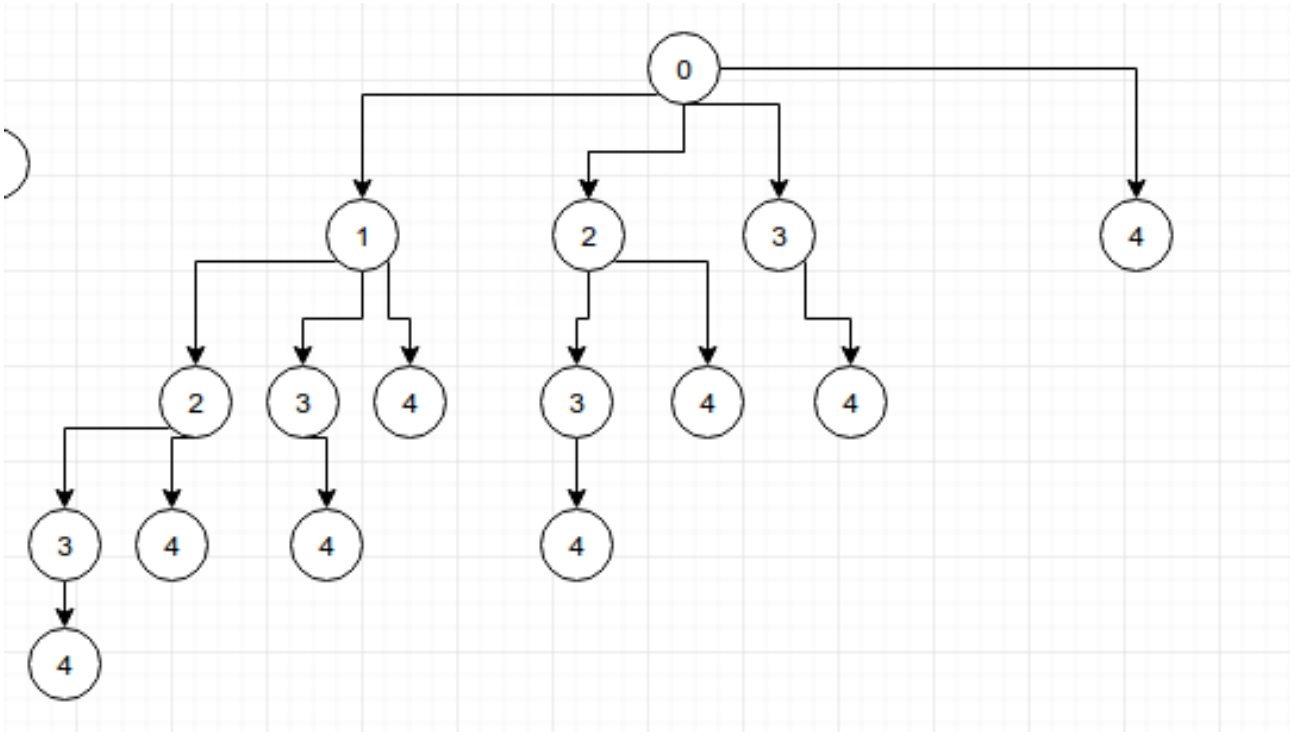

Figura 2: Árvore de recursão de força bruta

Para $n = 4$ temos 16 chamadas de DP, ou $2^N 4$

Logo, a complexidade desse programa, somando com o custo para manipulação dos dados de entrada, $\mathbf{O}(2^N + N)$

3.4 Dinâmica

Esta análise se assemelha bastante com a de força bruta, pois o código é pouco diferente. O que faz esta solução ser dinâmica é o uso de memoization, que faz com que chamadas iguais da função não sejam calculadas novamente. Isso é feito através de uma matriz P por P, onde as linhas representam o número de palavra e as colunas o número de linhas, pois ambos são parâmetros da função DP.

Para preencher nossa matriz memoization, são necessárias apenas P chamadas recursivas do DP, e todas as seguintes irão carregar os valores da nossa matriz e terão custo $\mathbf{O(1)}$.

Então nossa complexidade é P * custo, e o custo da chamada no pior caso é P, logo a complexidade do DP é P*P = P^2

A impressão, assim como na força bruta, é feita de forma linear, pois imprimimos palavra por palavrar. $\mathbf{O(P)}$.

A equação de recorrência desse método é

$$F(i) = \begin{cases} k * (H - numeroLinhas)^X, & \text{se } i = n \\ min(F(i+j) + badness(i,j) from j = i+1 to P), & \text{caso contrario} \end{cases} \quad (2)$$

Essa função de recorrência é nada mais do que a função de custo citada na introdução do problema. A primeira parte, que tem a ver com o número de linhas, é calculada no caso base, pois ao chegar na última palavra possuímos o número de linhas da solução, daí calcular $K * (H - numeroLinhas)^X$. O passo recursivo é a segunda parte, ou seja, o somatório da nossa função, que calcula $K * (L - tamanhoLinha)^X$. Cada chamada recursiva é uma linha, portanto um termo do somatório.

Levando em consideração que temos a chamada ao copia texto, a complexidade total deste programa é $O(P^2 + N + P)$.

4 Análise de experimentos

Para medir o tempo de execução do código, foi utilizada o comando time do Ubuntu. Cada teste foi rodado 5 vezes e foi feita uma média com os tempos. Para realizar os testes foi utilizada uma máquina com Linux Ubuntu 15, processador I3 e 6GB de RAM.

4.1 Comparando tempo de execução

Foram testados três textos nos 3 métodos.

Vimos que claramente o força bruta é o mais lento, e seu crescimento é muito acentuado. O guloso por ser linear, com aumentos pequenos apresenta pouca diferença no seu tempo de execução. O dinâmico também apresentou o crescimento esperado, maior que o linear.

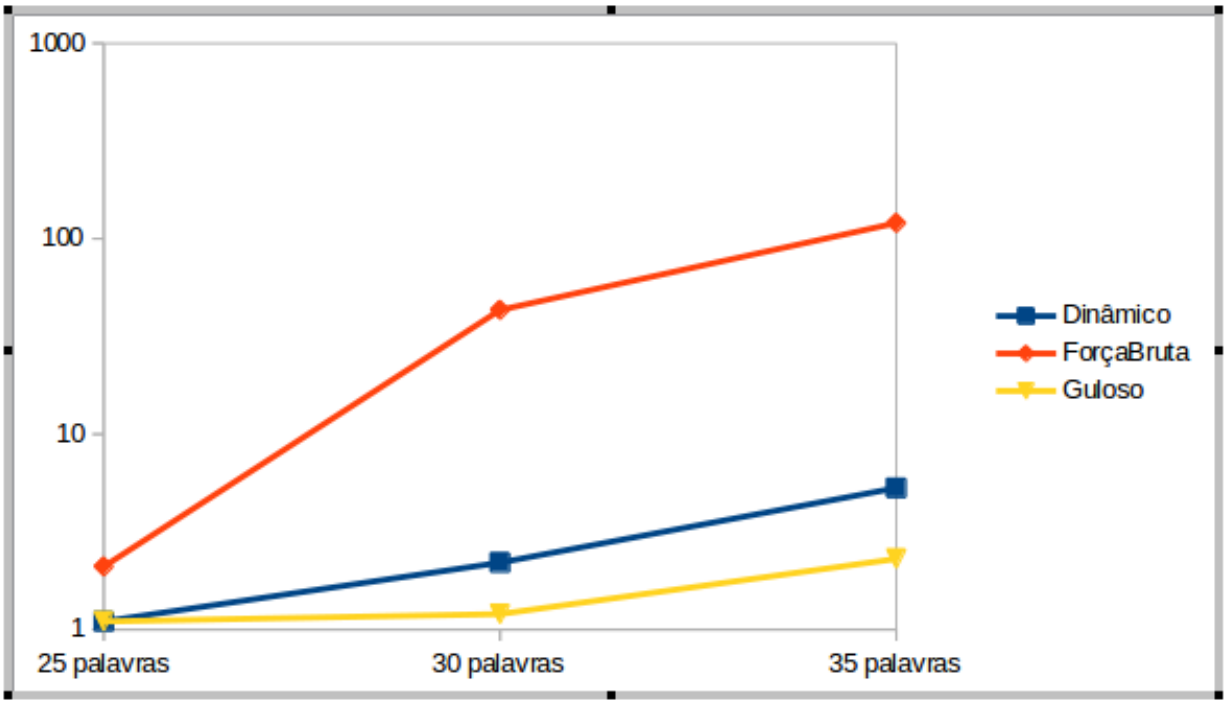

Figura 3: Tempo de execução para cada método

4.2 Comparando qualidade do resultado

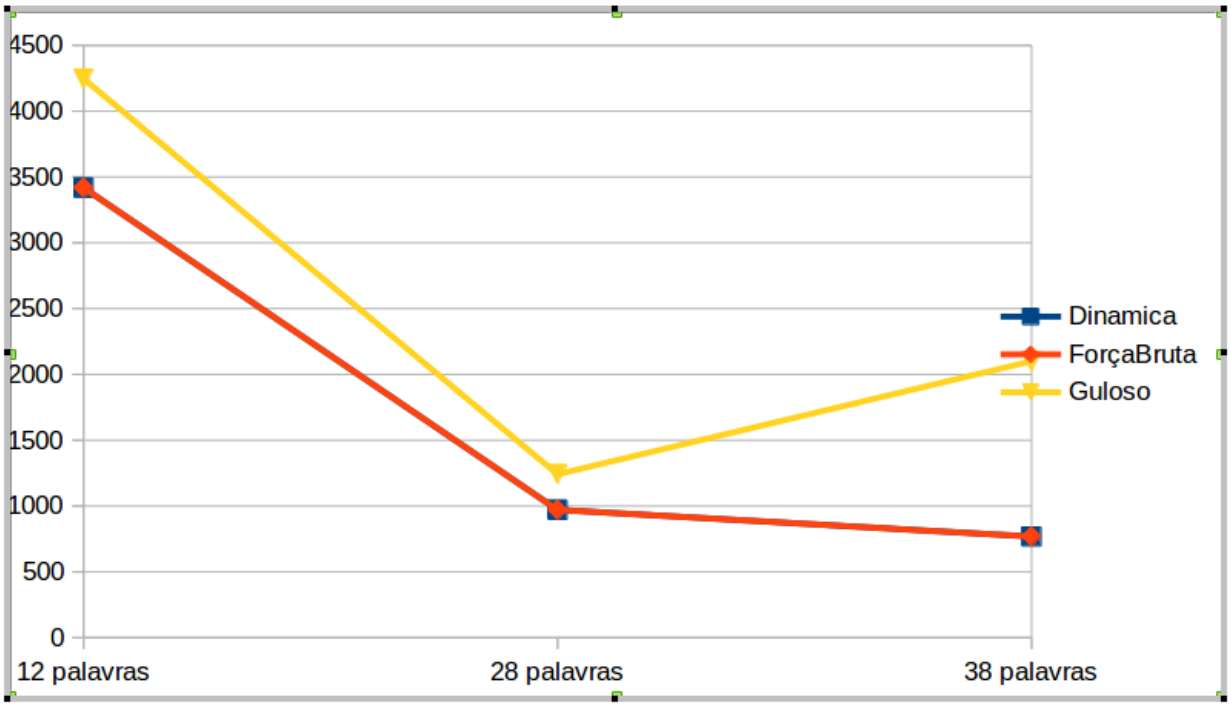

Figura 4: Tempo de execução para cada método

Como já era de se esperar, o custo do ForçaBruta e Dinâmico foram ótimos. Entretanto, o guloso foi mais custoso em todos os casos.

4.3 Variando L, H e parâmetros

Quanto mais aumentamos L, mais custo o diminui, pois podemos ter um número muito menor de linhas com um comprimento maior.

Aumentado H, o custo aumenta. Isso é explicado pela função de custo, que aumenta quanto maior for a diferença entre H e o número de linhas. Entretanto, se H diminui muito, pode ser impossível de justificar e quanto mais diminui, o custo tende ao infinito.

Aumentando o expoente, o custo aumenta absurdamente rápido. Isso pode ser explicado pois o expoente serve para definir quão ruim é uma não-harmonia, ou seja, um expoente maior é menos tolerante e gera um custo maior.

Aumentando o fator multiplicativo K, o custo aumenta linearmente, como era de se esperar.

5 Bibliografia

Referências

[Erik Demaine, 2011]