TP3: MyTex

Alef Henrique de Castro Monteiro

29 de junho de 2016

1 Introdução

O Trabalho Prático 3 tem por objetivo a implementação de um programa que realize a justificação de um texto, ou seja, torne o texto visualmente agradável. Para justificar um texto, o programa deve inserir quebras de linha no texto de forma que o comprimento das linhas deste seja uniforme. Além de possuir linhas de mesmo tamanho, um texto justificado não deve ter linhas vazias no final da página.

Para a realização da tarefa, o problema deve ser resolvido de 3 maneiras diferentes, cada uma delas utilizando um paradigma diferente. A primeira solução deve ser por meio de um algoritmo ótimo que utiliza força bruta. Outra solução para o problema deve apresentar um algoritmo que utilize uma heurística gulosa. Por fim, o programa deve solucionar o problema por meio de um algoritmo ótimo que utilize programação dinâmica.

2 Definição do Problema

Como foi dito anteriormente, o problema trazido pelo trabalho é, dado um texto contido em uma linha, justificar este texto. Para realizar a justificação, o programa deve ser capaz de minimizar o custo da função custo abaixo de 3 diferentes maneiras, cada uma utilizando um dos paradigmas.

$k(H - |l|)^x + \sum k(L - length(l_i))^x$

Nesta função, H representa o número máximo de linhas que a página suporta e L é a capacidade máxima de caracteres que cabe em cada linha. Além disso, l é o conjunto de linhas em que o texto foi dividido, $|l|$ é o número de linhas geradas, $length(l_i)$ é o comprimento da linha i gerada e, por fim, k e X são parâmetros.

Ao ser executado o programa, será dado um texto e será informado o paradigma que deve ser utilizado para justificar o texto. O objetivo do programa é, com a utilização do paradigma solicitado, obter e imprimir o custo mínimo para justificar o texto. Além de imprimir o custo mínimo, o programa deve também imprimir o texto justificado.

3 Solução do problema

Nesta seção do trabalho, serão apresentados os detalhes da implementação do programa utilizado no trabalho. A solução do problema é iniciada com a verificação dos parâmetros passados no momento da execução do programa, pelo terminal. Nesse momento, o programa verifica qual paradigma deve ser utilizado e recebe o arquivo de entrada. Ao ser executado, o programa verifica esse arquivo e, então, são obtidos os valores da capacidade L máxima de cada linha , do número máximo H de linhas que cabem em uma página e dos parâmetros x e k.

Além de obter esses valores que serão utilizados no cálculo do custo, o programa obtém o texto que deve ser justificado e o divide em palavras, onde cada palavra passa a ser uma posição de um vetor $WordsList$, do tipo $TypeWord$. Para armazenar as palavras do texto e seus respectivos tamanhos, o tipo de dados $TypeWord$ foi criado a partir de uma *'struct'*. Essa estrutura é composta pela palavra (string) e pelo valor de seu comprimento (int). Dessa forma, a função do programa passa a ser identificar a melhor forma de dividir o vetor de palavras

$WordsList$ em linhas, de modo que essa divisão gere linhas que minimizem a função custo. Além disso, a função $Line\_cost$(**Algoritmo 1**) tem acesso ao comprimento das palavras da linha e, consequentemente, pode calcular o comprimento total(linha 1) e o custo gerado pela linha(linha 2-6).

Algoritmo 1: Line_cost

Entrada: Linha $line(i, j)$, Comprimento máximo da linha L

1 $length \leftarrow line(i, j).length$
2 **if** $line(i, j).length > L$ **then**
3 | return ∞
4 **end**
5 **else**
6 | $cost \leftarrow k * (L - length)^x$
7 | return $cost$
8 **end**

A tarefa de calcular o custo mínimo para justificar o texto pode ser realizada por meio de 3 algoritmos diferentes: guloso, força bruta e de programação dinâmica. Cada um desses algoritmos serão detalhados a seguir.

3.1 Algoritmo Guloso

A solução do problema a partir do algoritmo guloso é bem simples. Como todo algoritmo guloso, esta estratégia realiza a escolha que parece ser a melhor no momento, ou seja, uma escolha ótima local, de forma que essas escolhas possam fazer com que a solução se aproxime da solução ótima do problema de otimização. No caso do problema de justificar o texto, a estratégia gulosa utilizada é descrita a seguir.

1) verificar, para cada palavra do vetor que representa o texto, se a palavra cabe ou não na linha que está sendo construída.

2) Se a palavra couber, acrescentá-la na linha que está sendo construída.

3) Se a palavra não couber, adicionar uma quebra de linha no texto e começar a escrever uma nova linha.

4) Repetir este processo até a última palavra do vetor.

Ao final deste processo, o algoritmo terá criado um texto justificado. Este processo pode ser detalhado a partir do pseudo-código a seguir.

Algoritmo 2: Heurística Gulososa(Greedy)

```
Entrada: Vetor de palavras WordsList, Vetor linebreaks_indexes, Comprimento máximo da linha L,
         Número máximo de linhas H, constante x, constante k, Número de palavras do texto
         num_words
num_lines ← 0
while j ← num_words do
    line_capacity ← 0
    while line_capacity + wordsList[j].lenght <= L e j < num_words do
        line_capacity ← line_capacity + wordsList[j].lenght
        j ← j + 1
    end
    linebreaks_indexes[num_lines] = j − 1
    num_lines ← num_lines + 1
    line_cost ← Line_cost()
    if line_cost = ∞ then
        return ∞
    end
    total_cost ← total_cost + line_cost
end
if num_lines > H then
    return ∞
end
total_cost ← total_cost + k ∗ (H − num_lines)^x
return total_cost
```

É importante verificar que o algoritmo guloso não é ótimo, é apenas uma heurística. Para mostrar que esta solução não é ótima, basta verificar o exemplo a seguir, onde o texto é "aaa bb cc ddddd", a capacidade L de cada linha é 6 e os outros parâmetros são $H = 3$, $x = 3$, $k = 1$.

Solução gulosa

aaa bb

cc

ddddd

Custo: $0 * (3-3)^3 + [1 * (6-6)^3 + 1 * (6-2)^3 + 1 * (6-5)^3] = 0 + 0^3 + 4^3 + 1^3 = 65$

Solução ótima

aaa

bb cc

ddddd

Custo: $0 * (3-3)^3 + [1 * (6-3)^3 + 1 * (6-5)^3 + 1 * (6-5)^3] = 0 + 3^3 + 1^3 + 1^3 = 29$

Como foi mostrado no exemplo acima, existem situações onde a escolha ótima local não implica na solução ótima global. Essa diferença de custo ocorre porque a solução gulosa ignora o fato de que, mesmo quando a próxima palavra cabe na linha que está sendo construída, colocar a palavra na linha de baixo pode tornar o custo geral menor, apesar de aumentar o custo da linha atual. Portanto, como o algoritmo, apesar de ser mais eficiente, nem sempre leva a uma solução ótima do problema, o algoritmo é apenas uma heurística.

3.2 Algoritmo de Programação Dinâmica

A solução que utiliza programação dinâmica utiliza a seguinte equação de recorrência:

$$\begin{cases} DP[i] = min line\_cost(i,j) + DP[j], para\ cada\ j\ no\ intervalo\ [i+1, n] \\ DP[n] = k * (H - num\_lines)^x \end{cases} \tag{1}$$

Nesta equação, i representa o índice da palavra que iniciará a linha e j representa o índice da última palavra da linha da linha no vetor de palavras $WordsList$. Pode-se verificar que, para cada i, é verificada todas as possibilidades de formar linhas com as palavras i até n. Para mostrar que a equação de recorrência resolve o problema, basta observar o problema como um problema de sufixos, onde $D[p]$ é o sufixo a ser calculado. Dessa forma, o caso em que $DP[j] = DP[n]$ resulta no caso base da recursão. Ou seja, o custo de se quebrar a linha $(n, n+1)$ para qualquer valor de n é apenas o custo da sobra de linhas ao fim da página, já que o custo da linha $(n, n+1)$ é inexistente. Assim, obtido o valor de $DP[n]$, é possível calcular $DP[i]$ para qualquer $i <= n$. O código correspondente ao algoritmo de programação dinâmica encontra-se abaixo.

Algoritmo 3: Dynamic_programming (Programação dinâmica)

```
Entrada: Índice i, Vetor linebreaks_indexes, Comprimento máximo da linha L, Número máximo de
         linhas H, constante x, constante k, Número de palavras do texto num_words
num_lines ← 0
minimum_cost ← ∞
//Condicoes de parada.
if num_lines > H then
    return ∞
end
if i = num_words then
    linebreaks_indexes[0] ← num_words
    return k * (H − num_lines)^x
end
//Utilizacao da tecnica de Memoization.
if memoization[i][num_lines] ≠ −1 then
    linebreaks_indexes ← memoization_linebreaks[i][num_lines]
    return memoization[i][num_lines]
end
for j ← i + 1 to num_words do
    Cria vetor break_index para salvar a sequencia de quebras de linha para cada (i, j)
    line_i_to_j_cost ← Line_cost(i, j)
    if line_i_to_j_cost ≠ ∞ then
        //Chamada recursiva.
        total_cost ←
        line_i_to_j_cost + Dynamic_programming(j, linebreaks_indexes, L, H, x, k, num_words)
    end
    else
        total_cost ← ∞
    end
    if minimum_cost > total_cost then
        minimum_cost ← total_cost
        linebreaks_indexes[1 :] ← break_index
        linebreaks_indexes[0] ← j − 1
    end
end
memoization[i][num_lines] ← minimum_cost
memoization_linebreaks[i][num_lines] ← linebreaks_indexes
return minimum_cost
```

3.3 Algoritmo Força Bruta

A solução por força bruta foi feita com a utilização de um algoritmo recursivo, com a utilização da equação recursiva a seguir, que teve seu funcionamento explicado na **Seção 3.2**.

$$\begin{cases} DP[i] = minline\_cost(i,j) + DP[j], para\ cada\ j\ no\ intervalo\ [i+1, n] \\ DP[n] = k * (H - num\_lines)^x \end{cases} \tag{2}$$

O funcionamento do algoritmo força bruta é semelhante ao do algoritmo de programação dinâmica. No entanto, este algoritmo não utiliza a técnica de memorizar o resultado das recursões já calculadas anteriormente (Memoization). Assim, este algoritmo verifica todas as possibilidades válidas, calculando todas as recursões possíveis. Esse processo pode ser verificado a partir do pseudo-código abaixo, que mostra o funcionamento do algoritmo de força bruta.

Algoritmo 4: Bruteforce (Força Bruta)

Entrada: Índice i, Vetor linebreaks_indexes, Comprimento máximo da linha L, Número máximo de linhas H, constante x, constante k, Número de palavras do texto num_words

$num\_lines \leftarrow 0$
$minimum\_cost \leftarrow \infty$
//Condicoes de parada.
if $num\_lines > H$ **then**
    return ∞
end
if $i = num\_words$ **then**
    $linebreaks\_indexes[0] \leftarrow num\_words$
    return $k * (H - num\_lines)^x$
end
for $j \leftarrow i + 1$ **to** num_words **do**
    Cria vetor $break\_index$ para salvar a sequencia de quebras de linha para cada (i, j)
    $line\_i\_to\_j\_cost \leftarrow Line\_cost(i, j)$
    if $line\_i\_to\_j\_cost \neq \infty$ **then**
        //Chamada recursiva.
        $total\_cost \leftarrow line\_i\_to\_j\_cost + Bruteforce(j, linebreaks\_indexes, L, H, x, k, num\_words)$
    end
    else
        $total\_cost \leftarrow \infty$
    end
    if $minimum\_cost > total\_cost$ **then**
        $minimum\_cost \leftarrow total\_cost$
        $linebreaks_indexes \leftarrow break\_index[1:]$
        $linebreaks\_indexes[0] \leftarrow j - 1$
    end
end
return $minimum\_cost$

4 Análise de Complexidade

O trabalho é composto do programa principal tp3.c e mais 2 tipos abstratos de dados (TADs): Text e Paradigms. Nesta seção, cada uma das principais funções será analisada separadamente e, por fim, será analisado o programa principal referente a cada um dos paradigmas. Para as análises a seguir, será utilizada a seguinte notação: T é o **tamanho do texto** fornecido na entrada e N é o **número de palavras do texto**.

4.1 Análise de complexidade das principais funções

- **void Num_words_text()** - A função verifica o número de palavras presentes no texto fornecido pela entrada. Além das operações $O(1)$, essa função contém um loop que depende do tamanho do texto, já que ela percorre cada caractere do texto e conta o número de espaços entre as palavras. Portanto, o loop é $O(T)$ e, portanto, a função é $O(T)$.
- **void Split_text()** - A função separa o texto em palavras e coloca cada uma das palavras em um vetor. A função contém apenas um 'while', que contém apenas operações $O(1)$. Como o 'while' depende do tamanho T do texto, sua complexidade é $O(T)$. Dessa maneira, a complexidade final da função é $O(T)$.
- **void Get_words()** - A função tem por finalidade obter os valores dos parâmetros e obter o texto, todos contidos no arquivo de entrada. Além das operações $O(1)$, a função faz uma chamada à função $Num\_words\_text - O(T)$ e uma chamada à função $Split\_text - O(T)$. Além disso, a função contém dois loops aninhados que tem a finalidade de alocar memória para cada palavra do vetor de palavras $wordsList$. Logo, esses loops dependem do número N de palavras. Dessa forma podemos dizer que a complexidade dos dois loops aninhados é $O(N)$. Portanto, a função tem complexidade $O(T + T + N) = O(N + T)$.
- **void Print_text()** - Esta função é a que imprime o texto justificado no arquivo de saída. Como ela imprime cada uma das palavras do texto, sua complexidade é $O(N)$.

4.2 Algoritmo Guloso

O algoritmo guloso é representado pela função *Greedy*, que foi detalhada no **Algoritmo 2**(Seção 3.1). Além das operações $O(1)$, esta função contém dois loops aninhados. O loop externo (linha 2-15) é executado enquanto houver palavras do texto que ainda não foram inseridas em nenhuma linha. O loop interno(linha 4-7), por sua vez, é executado para cada palavra, enquanto couberem palavras na linha que está sendo criada. Portanto, pode-se resumir o funcionamento dos loops aninhados da seguinte forma: para cada palavra do texto, o algoritmo verifica se a palavra cabe ou não na linha atual. Como existem N palavras no texto, a complexidade da função é $O(N)$.

Quando o programa principal utiliza o algoritmo guloso, no pior caso, ele chama uma vez as funções:

- $Get\_words - O(N + T)$
- $Greedy - O(N)$
- $Print\_text - O(N)$

Além disso, o programa realiza algumas outras operações $O(1)$ e contém um loop, que depende do número N de palavras do texto. Então, este loop tem complexidade $O(N)$. Dessa forma, quando utliza a heurística gulosa, o custo final do programa é $O(N + T + N + N + N) = O(N + T)$.

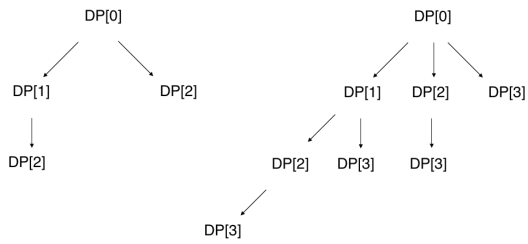
4.3 Algoritmo Força Bruta

O algoritmo de força bruta é representado pela função $Bruteforce$, que pode ser visualizada no **algoritmo 4**. Como a função é recursiva, a análise de complexidade desta função será feita a partir de sua equação de recorrência:

$$\begin{cases} DP[i] = minline\_cost(i, j) + DP[j], para\ cada\ j\ no\ intervalo\ [i + 1, n] \\ DP[n] = k * (H - num\_lines)^x \end{cases} \tag{3}$$

Para obter a complexidade do algoritmo de força bruta, é necessário verificar o número de chamadas recursivas realizadas pelo programa no pior caso e o custo de cada uma dessas chamadas. Assim, a complexidade final do algoritmo será $O(Custo\ da\ chamada * numero\ de\ chamadas)$.

Ao analisar o **algoritmo 4**, verifica-se que em cada chamada é realizado um loop (**linhas 11-26**). Para cada i, é necessário calcular o valor de $DP[j]$ para todo j no intervalo $[i + 1, n]$. Dessa forma, a recursão do algoritmo força bruta forma a seguinte árvore.



Ao analisar a árvore, percebe-se cada nó da árvore corresponde a uma chamada recursiva e sua altura corresponde a N. Assim, ao aumentar N de 2 para 3, o número de chamadas recursivas dobra. Se N fosse 4, cada nó da última árvore teria um filho $DP[4]$, ou seja, para calcular $DP[i]$ para qualquer valor seria necessário calcular $DP[4]$. Portanto, podemos concluir que a árvore contém 2^N nós. Assim, a complexidade do algoritmo é 2^N. Quando o programa principal utiliza o algoritmo de força bruta, no pior caso, ele chama uma vez as funções:

- $Get\_words - O(N + T)$
- $Bruteforce - O(2^N)$
- $Print\_text - O(N)$

Além disso, o programa realiza algumas outras operações $O(1)$ e contém um loop, que depende do número N de palavras do texto. Então, este loop tem complexidade $O(N)$. Dessa forma, o custo final do programa é $O(N + T + 2^N + N + N) = O(2^N + T)$.

4.4 Algoritmo de Programação Dinâmica

A análise de complexidade do algoritmo de programação dinâmica é semelhante à análise do algoritmo de força bruta, já que os dois algoritmos utilizam a mesma equação de recorrência. No entanto, o algoritmo de programação dinâmica utiliza **Memoization**, ou seja, os custos mínimos de cada chamada recursiva $DP[i]$ são armazenados em um vetor de tamanho N(uma posição do vetor para cada valor possível de i). Dessa forma, após ter calculado $DP[i]$ pela primeira vez, sempre que o programa necessitar calcular o valor de $DP[i]$ novamente, ele irá apenas resgatar o resultado da chamada no vetor. Como isso, o máximo de chamadas recursivas que serão realizadas pelo programa será N chamadas, para calcular $DP[i]$ para todo i no intervalo $[0, n-1]$. Portanto, a complexidade da função será $N * custo\ da\ chamada$. Como o custo de cada chamada é $O(N)$, que corresponde a verificação do mínimo(loop das linhas 16-31), a complexidade final do algoritmo é $O(N^2)$.

Quando o programa principal utiliza o algoritmo de programação dinâmica, no pior caso, ele chama uma vez as funções:

- $Get\_words - O(N + T)$
- $Dynamic\_programming - O(N^2)$
- $Print\_text - O(N)$

Além disso, o programa realiza algumas outras operações $O(1)$ e contém um loop, que depende do número N de palavras do texto. Então, este loop tem complexidade $O(N)$. Dessa forma, o custo final do programa é $O(N + T + N^2 + N + N) = O(N^2 + T)$.

4.5 Análise de Complexidade do Espaço

Durante o programa, é necessária a alocação de memória para o vetor de palavras e para o vetor que armazena as posições das quebras de linha. No caso dos dois vetores, a alocação depende do número N de palavras. Além disso, quando é utilizado o algoritmo de programação dinâmica, o programa precisa alocar memória para 2 vetores utilizados na Memoization. Como os dois vetores também dependem de N e um desses vetores é uma matriz, temos que a complexidade dessa alocação é $O(N^2)$. Portanto, a complexidade do espaço é $O(N)$ para a solução gulosa e para o algoritmo de força bruta. No caso da soluçao com o algoritmo de programação dinâmica, a complexidade é $O(N^2)$.

5 Análise de Experimentos

5.1 Metodologia

Para a realização do estudo experimental, foram realizados diversos testes. Os testes utilizados no experimento foram feitos com a finalidade de avaliar a diferença dos tempos de execução quando a solução para o problema utiliza paradigmas diferentes. Além disso, os experimentos também avaliam a influência dos parâmetros x e k no tempo de execução e no custo. Por fim, são feitos testes para verificar a interferência da capacidade máxima L da linha e da quantidade máxima H de linhas permitidas na página no custo e no tempo.

Inicialmente, foram realizados 9 testes com o objetivo de verificar a diferença dos tempos de execução entre os 3 algoritmos utilizados: programação dinâmica, guloso e força bruta. Para a realização destes testes, os valores L,H,x e k foram mantidos constantes: $L = 50$ $H = 30$ $x = 3$ $k = 1$. Os detalhes desses teste podem ser verificados na **tabela 1**. Nestes testes, todas as palavras tem o mesmo tamanho.

Tabela 1: Testes com diferentes paradigmas

Teste	Número de Palavras (N)	Número de caracteres do texto (T)
Teste 1	25	250
Teste 2	25	500
Teste 3	25	1000
Teste 4	50	250
Teste 5	50	500
Teste 6	50	1000
Teste 7	100	250
Teste 8	100	500
Teste 9	100	1000

Posteriormente, foram realizados mais 20 testes em 4 blocos. Cada bloco de testes variou um dos parâmetros L, H, x e k. O objetivos desses teste é avaliar a influência de cada um desses valores no custo e no tempo de execução. Todos esses testes foram realizados com o algoritmo de programação dinâmica estão detalhados nas tabelas a seguir.

Tabela 2: Testes com variação do valor de x e de k

Teste	Parâmetro x	Parâmetro k	Número máximo de linhas da página (H)	Comprimento máximo da linha (L)	Número de Palavras (N)	Número de caracteres do texto (T)
Teste 1	1	1	30	50	100	1000
Teste 2	2	1	30	50	100	1000
Teste 3	3	1	30	50	100	1000
Teste 4	4	1	30	50	100	1000
Teste 5	5	1	30	50	100	1000

Teste	Parâmetro x	Parâmetro k	Número máximo de linhas da página (H)	Comprimento máximo da linha (L)	Número de Palavras (N)	Número de caracteres do texto (T)
Teste 1	3	1	30	50	100	1000
Teste 2	3	1	60	50	100	1000
Teste 3	3	1	120	50	100	1000
Teste 4	3	1	240	50	100	1000
Teste 5	3	1	480	50	100	1000

Tabela 3: Testes com variação do valor de L e de H

Teste	Parâmetro x	Parâmetro k	Número máximo de linhas da página (H)	Comprimento máximo da linha (L)	Número de Palavras (N)	Número de caracteres do texto (T)
Teste 1	3	1	30	50	100	1000
Teste 2	3	1	30	100	100	1000
Teste 3	3	1	30	200	100	1000
Teste 4	3	1	30	400	100	1000
Teste 5	3	1	30	800	100	1000

Teste	Parâmetro x	Parâmetro k	Número máximo de linhas da página (H)	Comprimento máximo da linha (L)	Número de Palavras (N)	Número de caracteres do texto (T)
Teste 1	3	1	30	50	100	1000
Teste 2	3	2	30	50	100	1000
Teste 3	3	4	30	50	100	1000
Teste 4	3	8	30	50	100	1000
Teste 5	3	16	30	50	100	1000

A implementação dos experimentos foi feita com a utilização do compilador GCC, utilizando as flags $-O3$, $-g$ e $-Wall$. Os testes foram executados em uma máquina com 8GB de memória, processador Intel i5 2.7 GHz e sistema operacional OS X El Captain.

5.2 Resultado dos experimentos

Após a execução, foi verificado tempo de execução de cada um dos testes do experimento. Além do tempo de execução, na maioria dos teste foi analisado também o custo gerado pelos testes. Os resultados dos experimentos estão detalhados nas tabelas a seguir. Após cada uma das tabelas, está um gráfico da variável em função do tempo e em função dos custo, quando é o caso.

Tabela 4: Testes com diferentes paradigmas

Teste	Número de Palavras (N)	Número de caracteres do texto (T)	Tempo - Programação Dinâmica	Tempo - Força Bruta	Tempo - Heurística Gulosa	Custo Ótimo	Custo - Heurística Gulosa
Teste 1	25	250	0,004	13,321	0,003	15629	15629
Teste 2	25	500	0,003	0,216	0,003	47885	47885
Teste 3	25	1000	0,003	0,003	0,003	33069	33069
Teste 4	50	250	0,017	-	0,003	15629	15629
Teste 5	50	500	0,014	-	0,003	8009	8009
Teste 6	50	1000	0,007	-	0,003	33069	33069
Teste 7	100	250	0,096	-	0,003	13859	13859
Teste 8	100	500	0,084	-	0,003	8009	8009
Teste 9	100	1000	0,069	-	0,003	1019	1019

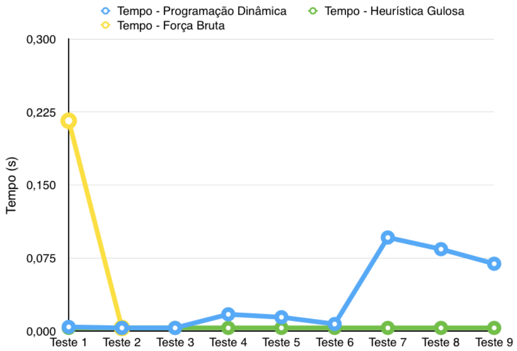
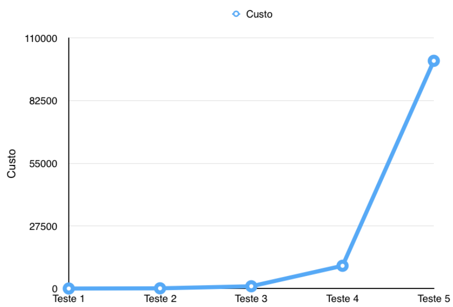


Tabela 5: Testes com variação do valor de x

Teste	Parâmetro x	Parâmetro k	Número máximo de linhas da página (H)	Comprimento máximo da linha (L)	Número de Palavras (N)	Número de caracteres do texto (T)	Custo	Tempo (s)
Teste 1	1	1	30	50	100	1000	29	0,066
Teste 2	2	1	30	50	100	1000	119	0,065
Teste 3	3	1	30	50	100	1000	1019	0,066
Teste 4	4	1	30	50	100	1000	10019	0,066
Teste 5	5	1	30	50	100	1000	100019	0,067



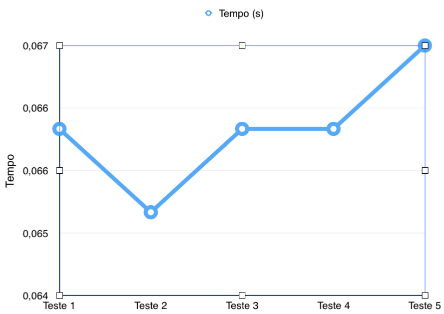
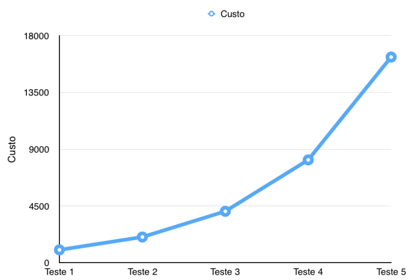


Tabela 6: Testes com variação do valor de k

Teste	Parâmetro x	Parâmetro k	Número máximo de linhas da página (H)	Comprimento máximo da linha (L)	Número de Palavras (N)	Número de caracteres do texto (T)	Custo	Tempo(s)
Teste 1	3	1	30	50	100	1000	1019	0,065
Teste 2	3	2	30	50	100	1000	2038	0,064
Teste 3	3	4	30	50	100	1000	4076	0,065
Teste 4	3	8	30	50	100	1000	8152	0,064
Teste 5	3	16	30	50	100	1000	16304	0,063



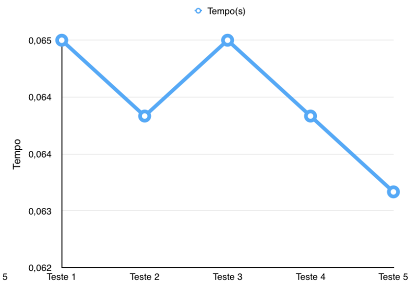
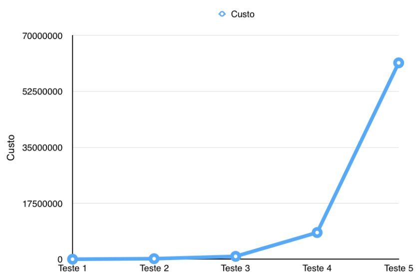


Tabela 7: Testes com variação do valor de H

Teste	Parâmetro x	Parâmetro k	Número máximo de linhas da página (H)	Comprimento máximo da linha (L)	Número de Palavras (N)	Número de caracteres do texto (T)	Custo	Tempo(s)
Teste 1	3	1	30	50	100	1000	1019	0,068
Teste 2	3	1	60	50	100	1000	164019	0,092
Teste 3	3	1	120	50	100	1000	890319	0,099
Teste 4	3	1	240	50	100	1000	8343949	0,104
Teste 5	3	1	480	50	100	1000	61457039	0,108



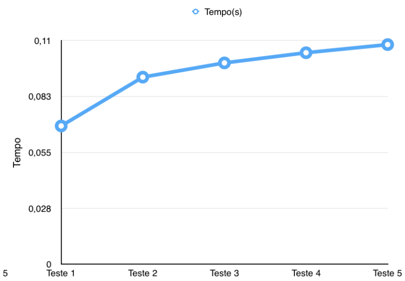
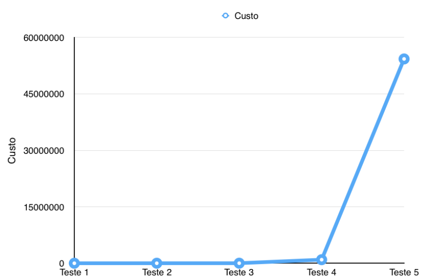
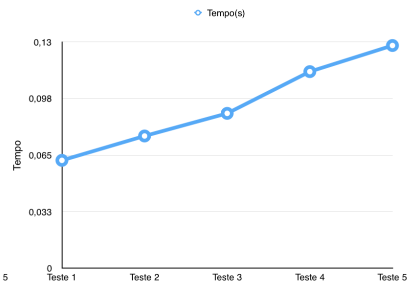


Tabela 8: Testes com variação do valor de L

Teste	Parâmetro x	Parâmetro k	Número máximo de linhas da página (H)	Comprimento máximo da linha (L)	Número de Palavras (N)	Número de caracteres do texto (T)	Custo	Tempo(s)
Teste 1	3	1	30	50	100	1000	1019	0,062
Teste 2	3	1	30	100	100	1000	8009	0,076
Teste 3	3	1	30	200	100	1000	15629	0,089
Teste 4	3	1	30	400	100	1000	947575	0,113
Teste 5	3	1	30	800	100	1000	54292853	0,128





5.3 Análise dos Resultados

Variação do tempo de execução entre os paradigmas A partir da **tabela 5** e do gráfico correspondente, é possível verificar que o tempo da heurística gulosa é sempre menor ou igual ao tempo dos demais paradigmas. Isso ocorre porque a ideia da heurística é justamente essa, ou seja, a vantagem da heurística é ser mais eficiente, já que ela não gera uma solução ótima. Além disso, pode-se verificar também que o algoritmo de força bruta torna-se ineficiente muito rápido, pois só conseguiu solucionar o problema em tempo hábil quando o número de palavras foi menor que 50. Isso se deve ao comportamento exponencial do tempo de execução do algoritmo de força bruta, que é $O(2^n)$ **(Seção 4.3)**.

Influência do parâmetro x Ao analisar a **tabela 6**, pode-se verificar que o parâmetro x praticamente não altera o tempo de execução do programa, apesar do gráfico não mostrar um crescimento linear. É importante notar que o gráfico não é linear Esse resultado já era esperado, pois a complexidade de tempo do programa não depende de x, já que este parâmetro é utilizado apenas na equação da função custo. Porém, a partir da tabela e do gráfico, verifica-se que este parâmetro aumenta exponencialmente o custo final da justificação do texto. Isso ocorre devido ao fato de x estar presente no expoente das equações de custo utilizadas pelo programa.

Influência do parâmetro k No que diz respeito ao parâmetro k, o comportamento do custo é diferente. Ao analisar a **tabela 7** e o gráfico, é possível verificar que o parâmetro k tem influência linear no custo. Essa influência pode ser observada na função custo, já que o k pode ser colocado em evidência: $k(H - |l|)^x + \sum k(L - length(l_i))^x = k[(H - |l|)^x + \sum(L - length(l_i))^x]$. Além disso, percebe-se, por meio da **tabela 6**, que o tempo permanece praticamente constante com a variação de k.

Influência do parâmetro L Na **tabela 9**, é possível verificar os testes em que o valor de L foi variado. Com o aumento de L, o tempo de execução dos testes também aumentou, assim como o custo. Portanto, pode-se afirmar que, quanto maior é o valor de L, maior será o custo e maior será o tempo de execução. O aumento no tempo, provavelmente, se deve ao fato de o algoritmo não realizar chamadas recursivas quando o custo da linha já for infinito. Assim, com o aumento da capacidade L de caracteres permitidos na linha, o número de linhas inválidas irá diminuir, fazendo com que o algoritmo realize mais chamadas recursivas. Já o impacto no custo, por sua vez, é esperado, já que o aumento de L aumenta o custo de cada uma das linhas.

Influência do parâmetro H Assim como o parâmetro L, o aumento de H também causa um aumento no tempo de execução e no custo final. No caso do H, o aumento do tempo também se deve ao aumento das chamadas recursivas, já que um maior valor de H reduz o número de configurações inválidas e obriga o programa a testar mais configurações. O aumento do custo também se deve ao fato de o aumento de H aumentar o valor da função custo.

Diferença da qualidade da solução entre o uso de programação dinâmica e o uso do algoritmo guloso A partir dos testes realizados com as soluções que utilizam tanto o algoritmo de programação dinâmica quanto o algoritmo guloso, é possível verificar que a solução gulosa tende a ser eficiente apenas quando o tamanho das palavras do texto tem aproximadamente o mesmo tamanho. Isso ocorre porque, quando as palavras são do mesmo tamanho, a solução gulosa é idêntica a solução ótima.

6 Conclusão

O Trabalho Prático 3 abordou a resolução de um problema de otimização por meio de 3 diferentes paradigmas. O problema foi solucionado com uma heurística gulosa, com um algoritmo de programação dinâmica e com um algoritmo de força bruta. Além de solucionar o problema, foi realizado um estudo experimental para verificar a variação do tempo de execução do programa quando cada um dos algoritmos são utilizados. Por meio do estudo experimental, pode-se confirmar a análise de complexidade do tempo para os algoritmos. No estudo, foi mostrado que o tempo do algoritmo de força bruta realmente tem um comportamento exponencial. Além disso, foi verificado que o tempo gasto pelo algoritmo guloso é sempre menor, ou seja, a heurística gulosa é de fato mais eficiente.

Este trabalho prático gerou algumas dificuldades, principalmente na implementação dos algoritmos recursivos. Encontrar a correta equação de recorrência é uma tarefa árdua, mas, depois de realizada, a implementação do programa torna-se simples. Por fim, a realização deste trabalho foi de extrema importância para praticar os conhecimentos adquiridos ao longo do curso e também para verificar, por meio de um estudo experimental, que o comportamento dos algoritmos correspondem ao comportamento esperado teoricamente.

Referências

[1] Erik Demaine, M. (2011). Dynamic programming ii: Text justification, blackjack. Disponível em

https://www.youtube.com/watch?v=ENyox7kNKeY.