UNIVERSIDADE DE SÃO PAULO
Instituto de Ciências Matemáticas e de Computação

Projeto 3
Programação Dinâmica 2

Venilton FalvoJr – 7902878

1. Introdução

Este relatório documenta o segundo trabalho desenvolvido para a disciplina de Projetos de Algoritmos (SCC5900), que consiste na implementação de um algoritmo de *Word Wrap* que minimize a soma dos quadrados dos espaços remanescentes de cada linha, utilizando o paradigma de Programação Dinâmica, calculando e justificando sua complexidade. O código-fonte submetido na plataforma Tidia-Ae também encontra-se disponível no GitHub[1].

O objetivo deste documento é explicar a estratégia de implementação utilizada e discutir brevemente seus resultados. Para isso, a Seção 2 apresenta a relação de recorrência de programação dinâmica utilizada para a implementação deste projeto. Na seção 3 detalhes técnicos e lógicos de tal implementação são apresentados. A Seção 4 sintetiza uma análise de performance e complexidade do algoritmo construído. Por fim, a Seção 5 apresentada a conclusão deste trabalho, bem como suas limitações e trabalhos futuros.

2. Função de Recorrência

Para este trabalho foi desenvolvido um algoritmo onde, dada uma sequência de palavras e um valor inteiro L, são geradas novas linhas como saída contendo as mesmas palavras, na mesma ordem, da sequência fornecida, separadas por um único espaço em branco ou por uma quebra de linha, de forma que nenhuma linha ultrapasse L caracteres (incluindo os espaços em branco), e que a soma dos quadrados dos espaços livres remanescentes de todas as linhas seja mínimo.

Para isso, uma relação de recorrência foi proposta para que tais objetivos fossem atingidos através de um algoritmo aderente ao conceito de programação dinâmica. A relação de recorrência a seguir busca minimizar a soma dos quadrados dos espaços livres remanescentes de todas as linhas. A função BADNESS retorna o custo de colocar as palavras do índice i ao j na mesma linha e é utilizada para auxiliar e facilitar o entendimento da expressão:

$$OPT(j) = \min_{0 < i \leq j} \{ OPT(i) + BADNESS(i, j) \}$$

$$BADNESS(i, j) = \begin{cases} \infty & \text{, caso as palavras de i a j ultrapassem L} \\ (L - LENGTH[i:j])^2 & \text{, caso contrário} \end{cases}$$

[1] Disponível em: https://github.com/falvojr-phd/scc5900/tree/master/Project3

3. Implementação

A solução implementada utiliza a estratégia interativa para construção da função de recorrência apresentada na seção anterior. Em termos técnicos, toda lógica foi desenvolvida na linguagem Java em sua versão 8. Tal implementação conta com algumas classes responsáveis por domínios específicos do problema em questão. A seguir, as principais dentre elas são explanadas de maneira sucinta.

3.1. br.usp.falvojr.wordwrap.Main

A classe *Main* é a classe principal do projeto. Nela o argumento de execução é processado com o objetivo de computar os dados nos padrões especificados para os arquivos de entrada. Tais argumentos são detalhados na tabela abaixo:

Tabela 1: Argumentos de Execução

Argumento	Descrição	Sintaxe
-d [path]	Define um diretório base para a obtenção dos arquivos de entrada.	$ java -jar Project3.jar **-d [path]**

A partir do diretório especificado, a classe *Main* processa cada arquivo válidos, recuperando seu respectivo valor de L e conjunto de palavras para a aplicação do algoritmo de *Word Wrap*, para isso a expressão regular **[\t\r\n]+** foi utilizada. Com o valor de L e o conjunto de palavras em mãos, a classe *DynamicProgramming* é acionada através de seu *Singleton*.

3.2. br.usp.falvojr.wordwrap.algorithm.DynamicProgramming

Classe mais importante no escopo deste projeto, nela o algoritmo da relação de recorrência foi desenvolvido. Nesse sentido, seus principais métodos serão sintetizados abaixo:

- ***init*:** método utilizado simplesmente para zerar o vetor de memoização. Ação necessária para o processamento de um novo conjunto de palavras;
- ***opt*:** método que representa a relação de recorrência deste trabalho. Ele recebe como parâmetros uma lista de palavras e o valor de L. Com isso, o algoritmo processa tais informações e às armazena em um vetor de memoização, garantindo acesso em ordem constante a seus subproblemas;

- ***printSolution*:** após o calculo do vetor de memoização é possível imprimir sua respectiva solução. Para isso, utiliza-se os dados relativos às quebras contidos no vetor para construção do resultado final.

3.3. br.usp.falvojr.wordwrap.WordWrapUtils

Classe utilitária que provê métodos genéricos e constantes.

4. Performance e Complexidade

Para avaliar a performance e complexidade foram consideradas as características da solução implementada, mensurando assim as complexidades de tempo e espaço implicadas.

Em termos práticos a solução utiliza um matriz para auxiliar nos cálculos relativos aos custos, gerando uma estrutura muito próxima triangular superior. Com isso foi possível minimizar a soma dos quadrados dos espaços remanescentes de cada linha em **O(n²)**, considerando a complexidade temporal.

Por outro lado, considerando o consumo de memória, a complexidade ficou proporcional a **O(n)**, já que a estratégia de memoização utilizou um vetor para armazenar as informações de quebra de linhas para este problema.

5. Conclusão e Trabalhos Futuros

Com base nos resultados obtidos foi possível entender que a complexidade de um algoritmo não se dá apenas por sua medida temporal. Analisar informações como consumo de memória podem ser cruciais para a caracterização de uma boa solução. Nesse sentido, a estratégia usando programação dinâmica se mostrou bastante eficaz para a resolução do problema.

Entretanto, como trabalhos futuros, seria interessante implementar uma solução com complexidade de tempo proporcional a O(n log n). Para isso, as estratégias de Dividir e Conquistar ou Busca Binária poderiam ser aplicadas. Adicionalmente, alguns autores relatam que o algoritmo SMAWK[2] poderia reduzir tal complexidade para O(n).

[2] Aggarwal, A.; Klawe, M. M.; Moran, S.; Shor, P. & Wilber, R. “Geometric applications of a matrix-searching algorithm”. Algorithmica, 1987, 2, 195-208.