



SERVIÇO PÚBLICO FEDERAL · MINISTÉRIO DA
EDUCAÇÃO UNIVERSIDADE FEDERAL DE VIÇOSA · UFV
CAMPUS FLORESTAL

Trabalho Prático 2
Algoritmos e Estrutura de Dados (AEDs)
Análise de complexidade de algoritmo

Alice Ladeira Gonçalves [5065]
Breno Junio de Oliveira Gomes [5085]
Jeiverson Christian Ventura Miranda dos Santos [5089]

Sumário

- 1. Introdução**
- 2. Organização**
- 3. Desenvolvimento**
 - 3.1 Função para gerar combinação**
 - 3.2 Função para gerar árvore**
 - 3.3 Função principal**
- 4. Resultados**
- 5. Conclusão**
- 6. Referências**

1. Introdução

O objetivo desse trabalho foi analisar a complexidade do algoritmo de resolução de um problema clássico da literatura, o Problema das Bolinhas de Natal (PBN). Esse problema é classificado como intratável, tendo em vista que não existe um algoritmo com complexidade polinomial para resolvê-lo. Ele é resolvido utilizando a técnica de força bruta, a única capaz de solucionar a questão e, a depender do valor de entrada, apresentar uma solução boa, mas não garantidamente a ideal. À medida que o valor de entrada aumenta, o tempo de execução tem o mesmo comportamento, o que se deve ao crescimento exponencial do número de comparações feitas.

2. Organização

A organização consiste uma pasta intitulada “tp2”, a qual possui os arquivos.c , os arquivos.h, a entrada.txt e uma outra pasta, “arquivos de teste”, que contém quatro arquivos.txt, com diferentes valores de entrada que serão usadas para medir o tempo de execução.

-gerarArvore.h e gerarArvore.c: essa função gera todas as árvores de natal possíveis até encontrar uma árvore válida de acordo com a matriz de adjacências. Ao encontrar uma árvore válida, ela é printada.

-gerarCombinacao.h e gerarCombinacao.c: cria um vetor que vai armazenar temporariamente um arranjo de números que definirá a árvore de natal.

-main.c: essa é a função responsável por gerar a matriz de adjacências de acordo com o arquivo de entrada e chamar as demais funções.

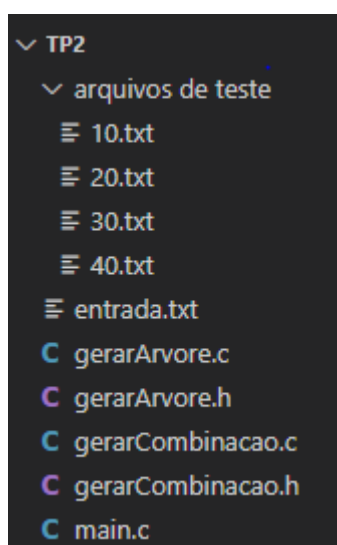


Figura 1 – organização dos arquivos

3. Desenvolvimento

O primeiro passo para o desenvolvimento do trabalho foi a leitura e interpretação da especificação, de forma a compreender o que deveria ser feito e encontrar a melhor forma de realizar esse trabalho. Tendo feito isso, notamos que o trabalho deveria ser realizado em partes, estando essas conectadas.

Essas partes estão distribuídas, no código, entre três funções: “gerarCombinacao”, “gerarArvore” e “main”. É importante destacar que as duas primeiras funções foram feitas com base em um algoritmo encontrado na internet [2], o qual passou por adaptações para atender nossas necessidades.

3.1 Função de gerar combinação

Essa função recebe cinco parâmetros do tipo inteiro, sendo esses, respectivamente: um vetor de espaços (espaços a serem ocupados por bolinhas de natal), a quantidade de espaços, quantidade de espaços ocupados pela cor azul, uma matriz quadrada (a matriz de adjacências) com as dimensões definidas pela quantidade de espaços e um ponteiro para a variável de controle que define quando o programa deve ser encerrado.

```
C gerarCombinacao.h > ...
1 void gerarCombinacao(int espacos[], int qE, int qA, int m[qE][qE], int *Pparar_programa);
2
```

Figura 2 – protótipo da função de gerar combinação (arquivo.h)

```
C gerarCombinacao.c
1 #include "gerarCombinacao.h"
2 #include "gerarArvore.h"
3
4 void gerarCombinacao(int espacos[], int qE, int qA, int m[qE][qE], int *Pparar_programa)
5 {
6     // vetor que guarda temporariamente a combinação gerada que define
7     // uma árvore com "qA" cores Azuis
8     int combinacaoAtual[qA];
9
10    // índices inicial e final para o vetor "espacos"
11    int inicio = 0;
12    int fim = qE-1;
13
14    // índice que será usado na outra função
15    int indice = 0;
16
17    int *PPparar_programa;
18    PPparar_programa = Pparar_programa;
19
20    gerarArvore(espacos, combinacaoAtual, inicio, fim, indice, qA, qE, m, PPparar_programa);
21 }
22
```

Figura 3 - função de gerar combinação (arquivo.c)

Ela é responsável por criar um vetor que armazenará temporariamente todas as combinações possíveis com a quantidade de bolinhas azuis e inicializar algumas

variáveis, as quais serão passadas como parâmetros para a função “gerarArvore” que está sendo chamada dentro dessa função.

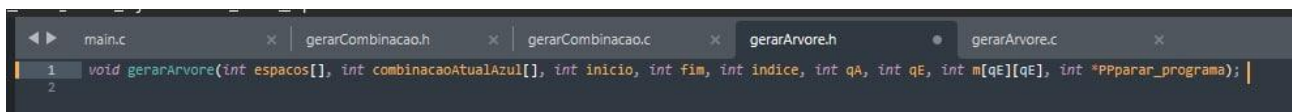
3.2 Função para gerar árvore

A função de gerar árvore recebe os parâmetros que são passados na função anterior. A primeira coisa que ele faz é gerar um arranjo de números que indicam em quais espaços ficarão as bolinhas azuis. Quando ele termina de gerar a combinação, entra na condição da linha 33.

A primeira condição só será verdadeira se o vetor “combinaçãoAtualAzul” já estiver formado, uma vez que isso seja verdade, então se a quantidade de azul for igual a zero, a árvore é construída apenas com bolinhas vermelhas. Quando a quantidade de azuis for maior que zero, com base na quantidade de azuis e suas respectivas posições, é feita uma combinação complementar com as cores vermelhas.

Exemplo: considerando uma árvore com 5 espaços, se Azuis = (0 1 3), a configuração das vermelhas será: Vermelhas = (2 4). Depois de gerar as combinações de vermelhos e azuis, a árvore é formada atribuindo “V” aos índices de vermelho e “A” para os índices de azuis, logo a árvore desse exemplo seria: (A A V A V).

Em seguida, usa a matriz de adjacências para verificar se a árvore é válida. Se for válida, o programa é parado, caso contrário, o próximo arranjo é feito e verificado, ou seja, o programa repete isso tudo até encontrar uma árvore válida ou acabarem as possibilidades de combinação.

The image shows a code editor with several tabs open: 'main.c', 'gerarCombinao.h', 'gerarCombinao.c', 'gerarArvore.h', and 'gerarArvore.c'. The 'gerarArvore.c' tab is active, displaying the following function prototype:

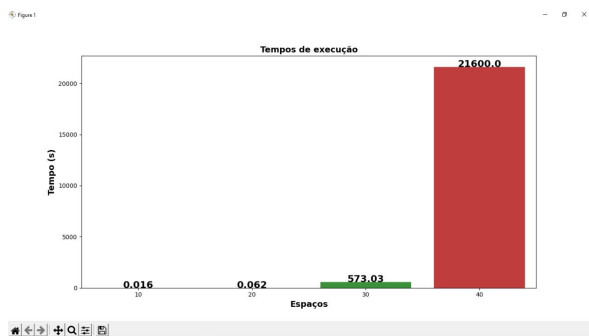
```
1 void gerarArvore(int espacos[], int combinacaoAtualAzul[], int inicio, int fim, int indice, int qA, int qE, int m[qE][qE], int *PPparar_programa); |
2
```

Figura 4 – protótipo da função de gerar árvore

3.3 Função principal

Essa função abre o arquivo de texto, constrói a matriz de adjacências a partir desse arquivo, chama as outras funções, as quais realizam o trabalho necessário e, por fim, imprime o tempo gasto para encontrar uma árvore válida ou o tempo gasto para fazer todas as verificações, quando não há árvores válidas.

4. Resultados



Os resultados alcançados foram organizados em forma de um gráfico que associa valores da entrada aos tempos de execução correspondentes. O último teste de tempo não tinha sido finalizado até a entrega do trabalho mas sua execução já tinha superado as 6 horas. A seguir estão as informações do computador usado para fazer os testes:

-Processador Intel® Core™ i5-4210U CPU @ 1.70GHz 2.40GHz

-RAM instalada 12,0 GB

-Tipo de Sistema: Sistema operacional de 64 bits, processador baseado em x64

5. Conclusão

Os objetivos esperados foram devidamente alcançados, uma vez que conseguimos observar o aumento exponencial do tempo de execução conforme a quantidade de espaços na árvore aumentava, de acordo com o arquivo de texto.

6. Referências

- [1] ZIVIANI, Nivio. Projeto de Algoritmos com Implementações em Pascal e C. 4a Edição. São Paulo: Pioneira, 1999.
- [2] Print all possible combinations of r elements in a given array of size n. Geeks for geeks, 19 de janeiro de 2022. disponível em: <https://www.geeksforgeeks.org/print-all-possible-combinations-of-r-elements-in-a-given-array-of-size-n/> Acesso em 29 de outubro de 2022.
- [3] SILVA, Thais Regina. Algoritmos e estruturas de dados I (CCF211): Comportamento Assintótico (Cap01 – Seção 1.3 – Ziviani). 24 de outubro de 2022. Apresentação do Power Point. Disponível em: https://ava.ufv.br/pluginfile.php/501370/mod_resource/content/1/Aula8-ComportamentoAssintotico-2022.pdf Acesso em 04 de novembro de 2022.

