Universidade Federal de Viçosa – Campus Florestal

Disciplina: CCF 211 - Algoritmos e Estrutura de Dados 1

Professora: Thais Regina de Moura Braga Silva

Alunos: Marcos Biscotto - 4236, Alan Araújo - 5096, Gabriel Marques - 5097

1. Introdução

Para esse Trabalho Prático, foi-nos solicitado avaliar o impacto causado pelo desempenho dos algoritmos em sua execução real. Tendo em vista a existência de problemas intratáveis de complexidade exponencial, criamos TAD's e utilizamos funções e comandos da Linguagem C para realizar combinações simultâneas de base 2 e partindo de um expoente 10 que iria dobrando a cada execução. Medimos o tempo de execução de cada caso separadamente até certo ponto e estipulamos a duração para valores do expoente os quais eram inviáveis deixar o programa executando. Através de um menu interativo, exibimos opções para o usuário.

2. Organização

Em relação à organização e arquitetura do repositório do projeto, consideramos deixá-los da forma mais organizada possível. Desta vez, colocamos a documentação do trabalho em uma pasta dentro do repositório do projeto e também deixamos alguns arquivos de entrada como exemplo dentro de uma pasta no repositório. Separamos os arquivos ".h" em uma pasta separada e deixamos na pasta geral os arquivos main.c e o makefile do projeto, como demonstrado na **Figura 1**.

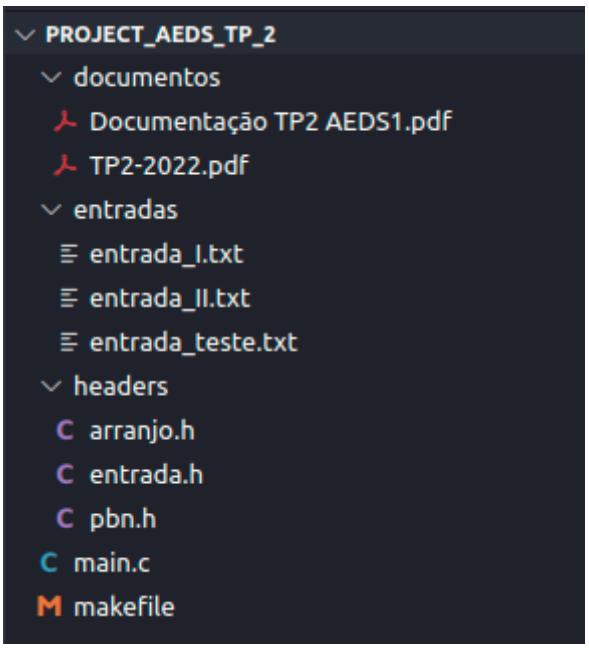

Figura 1 - Repositório do Projeto

Como as funções e variáveis do programa estão nomeadas de forma autoexplicativa e o código do mesmo está devidamente comentado, não achamos necessário colocar um descritivo ou um passo a passo em um arquivo “README”. O arquivo makefile possui os comandos de compilação e execução do programa, estruturado de maneira a compilar e rodar ao introduzir o comando “make” no terminal ou IDE.

3. Desenvolvimento

Abaixo relatamos o processo de desenvolvimento do tp, desde sua interpretação até o que foi solicitado como produto final, além de descrições sobre as tomadas de decisões e complicações a respeito do desenvolvimento do programa.

3.1) Interpretação:

Logo de início, a primeira complicação que tivemos para realizar esse trabalho prático, foi a dificuldade de nossa parte na interpretação da especificação do mesmo, o que nos custou certo tempo. Após tirarmos dúvidas com os monitores e a professora, compreendemos que o tp era dividido em duas partes: a primeira era para realizarmos as combinações de espaços na árvore, exibirmos o tempo que tal combinação demorou para ser executada e conferir a permutação com uma tabela de adjacência. A segunda parte é referente à leitura de um arquivo de entrada que seria os espaços e suas adjacências já feitas, e informariamos a matriz de adjacências como saída.

3.2) Headers:

Com base em nossa interpretação, além do arquivo main.c, utilizamos 3 arquivos “.h”, sendo eles responsáveis por gerar os arranjos de acordo com as cores e o tamanho, abertura e verificação do arquivo de entrada e um TAD que é o próprio Problema das Bolinhas de Natal em si. Não utilizamos outros arquivos “.c” pois deixamos o menu na própria main e as funções nos seus respectivos “.h”.

3.3) Tomada de Decisões:

Assim como esperado, a “resolução” desse problema se dá por força bruta, uma vez que quanto maior o número de espaços na árvore, mais execuções serão realizadas e maior será o tempo de execução. Apesar de na especificação do tp estar pedindo um código de Arranjo com repetição, procuramos pela internet mas não conseguimos encontrar um código em C de determinado algoritmo e mesmo tendo encontrado um código em PHP que fizesse tal procedimento, não tivemos a habilidade de converter PHP em C, então utilizamos um algoritmo de Permutação-R com repetição.

3.4) Algoritmo de Permutações-R com Repetição

Como mencionado anteriormente, foi utilizado um algoritmo que gera Permutações-R de um conjunto determinado. Um algoritmo que realiza essa tarefa é bastante útil para problemas que envolvem força bruta, como é no nosso caso. Para as Permutações-R com

repetição a quantidade de agrupamentos possíveis será de n^r. No sentido do nosso problema, n é a quantidade de cores de bolinhas de Natal e r o número de espaços na árvore de Natal.

O código utilizado para realização do trabalho é de autoria de Marcos Paulo Ferreira e pode ser encontrado no seguinte endereço: [Gerando Permutações-r com Repetição em C | Dæmonio Labs](#).

3.5) Explicação do algoritmo

O código começa fazendo a declaração de variáveis e a leitura do conjunto de entrada e do *r*. O conjunto pode ser uma *string* qualquer e o *r* deve ser um inteiro maior ou igual a 1. No nosso caso, a entrada *r* é obtida através da leitura do arquivo e pela função *get_size_arvore_natal*, que retorna o número o número de espaços disponíveis para a permutação. Já o conjunto a ser permutado, *string*, é prefixado, pois foi pedido o teste com apenas 2 cores distintas.

```
int *num;

char in_color[100] = "AB";
char string[100];

int n, r, i, j, k;
int m, o;

int num_arranjo; num_arranjo = 0;
int cont_true; cont_true = 0;
int cont_false; cont_false = 0;
int erro;

r = size;
```

Em seguida, elimina-se caracteres repetidos na entrada. Esse procedimento não faz parte do algoritmo, e só é feito por questões práticas. Devemos ter esse tipo de tratamento, pois a entrada deve conter elementos distintos.

```
n = strlen(in_color);
j = 0;
string[0] = 0;
for (i = 0; i < n; i++) {
    if (strchr(string, in_color[i]) == NULL) {
        string[j] = in_color[i];
        j++;
        string[j] = 0;
    }
}
strcpy(in_color, string);
n = strlen(in_color);
```

Na terceira etapa do algoritmo, há a alocação de *r + 1* posições para um vetor que irá representar nosso número. É mais prático trabalhar com um vetor em que cada posição é um algarismo do que trabalhar com um número inteiro de fato. A todo momento precisamos acessar uma posição específica de um único algarismo, e isso é mais fácil de ser feito com indexação de vetores. O *calloc* foi usado porque além de obter memória na heap, essa função também preenche com zeros esse novo espaço de memória. Como vimos, o vetor terá *r + 1* posições, sendo que essa última posição será usada para indicar quando todos os números já foram gerados.

```
num = (int *) calloc((r + 1), sizeof(int));
if (num == NULL) {
    perror("calloc");
    return -1;
}
```

O *while(num[r] == 0)* indexa a última posição do vetor que é usada como flag para o término do processo.

```
while (num[r] == 0) {
```

O loop *for* mais externo é o que incrementa o algarismo menos significativo. Esse algarismo é incrementado até o valor *n - 1*, onde *n* é a base do número (ex: na base 8, o valor limite é 8 - 1 = 7).

```
for (i = 0; i < n; i++) {
```

```
    num[0]++;
```

Há um mapeamento de cada posição de nosso vetor número em um elemento válido do conjunto de entrada. No fim do loop, o novo agrupamento estará na *string string*.

```
for (j = 0, k = r - 1; j < r; j++) {
    string[k] = in_color[num[j]];
    k--;
}
```

Agora, há uma parte que implementamos para a realização do trabalho. Essa parte, refere-se a verificação da matriz de adjacências. O código percorre a matriz e verifica os casos em que a adjacência ocorre. Em seguida, ele verifica a adjacência das cores do arranjo gerado. Por fim, ele apresenta se o arranjo é válido ou não, sua posição e o arranjo em si.

```
erro = 0;
for (m = 0; m < size; m++) {
    for (o = 0; o < size; o++) {
        if (matriz[m][o] == 1) {
            if (string[m] == string[o]) {
                erro = 1;
            }
        }
    }
}
string[r] = 0;
if (erro == 0) {
    printf("Arranjo [%d] válido: %s\n\n", num_arranjo, string);
    cont_true++;
}
if (erro == 1) {
    printf("Arranjo [%d] inválido: %s\n\n", num_arranjo, string);
    cont_false++;
}
num_arranjo++;
```

Na sequência é realizado o processo de mudança de casa e propagação de vai uns. Em outras palavras: todas as posições iguais a *n* devem ser setadas para 0, e ao mesmo tempo a posição à esquerda *(i + 1)* deve ser incrementada.

```
for (i = 0; i < r; i++) {
    if (num[i] == n) {
        num[i] = 0;
        num[i + 1]++;
    }
}
```

4. Resultados

Como resultado, temos um menu interativo funcional (**Imagem 2)** que disponibiliza algumas opções para o usuário, recebe um comando do mesmo e realiza uma operação baseada na escolha. Com a execução do código para diversos casos de entrada, plotamos

gráficos e estimamos tempos para operações que não convém serem realizadas na prática, uma vez que levam uma grande quantidade de tempo para serem finalizadas.

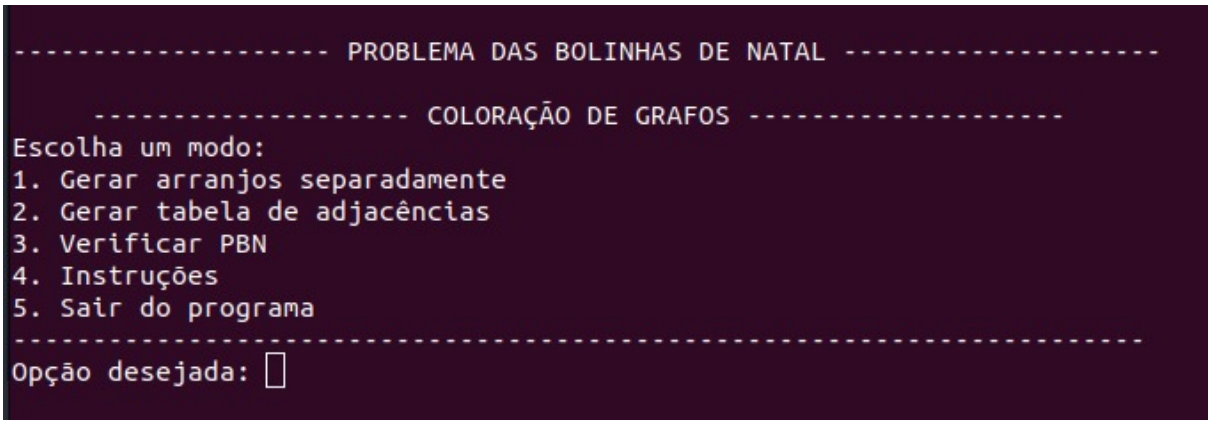

Figura 2 - Menu Interativo

O algoritmo foi executado em uma máquina com as seguintes especificações:

- Sistema operacional: Ubuntu 22.10
- Processador: Intel(R) Core(TM) i5-4690 CPU @ 3.50GHz com 4 núcleos e 4 threads
- Memória RAM: 12GB DDR3 1600MHz
- Armazenamento: SSD 240GB Kingston A400
- Placa de vídeo: NVIDIA GeForce GTX 960 SC 2GB
- Placa-mãe: B85M-E/BR (ASUSTeK COMPUTER INC.)

Nas **Figuras 3.1** e **3.2,** podemos observar o gráfico do Número de Espaços na Árvore por Número de Operações Realizadas, que segue modelo 2^n.

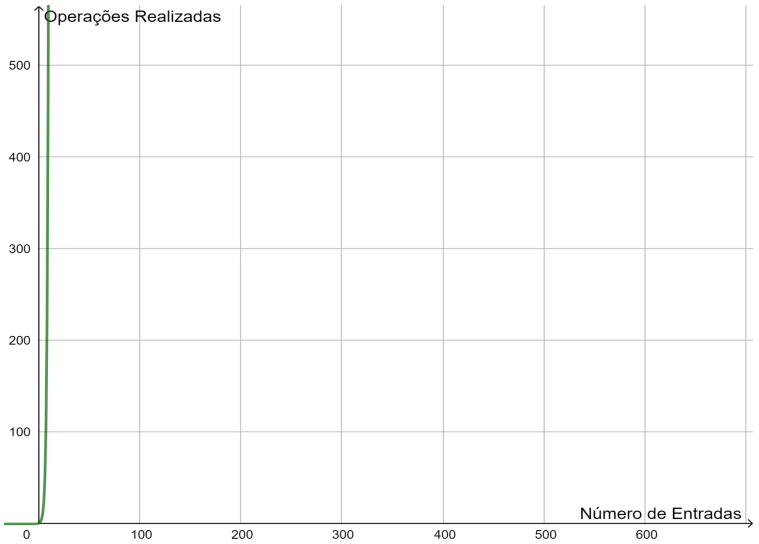

Figura 3.1 - Gráfico Entradas X

Operações com zoom menor*

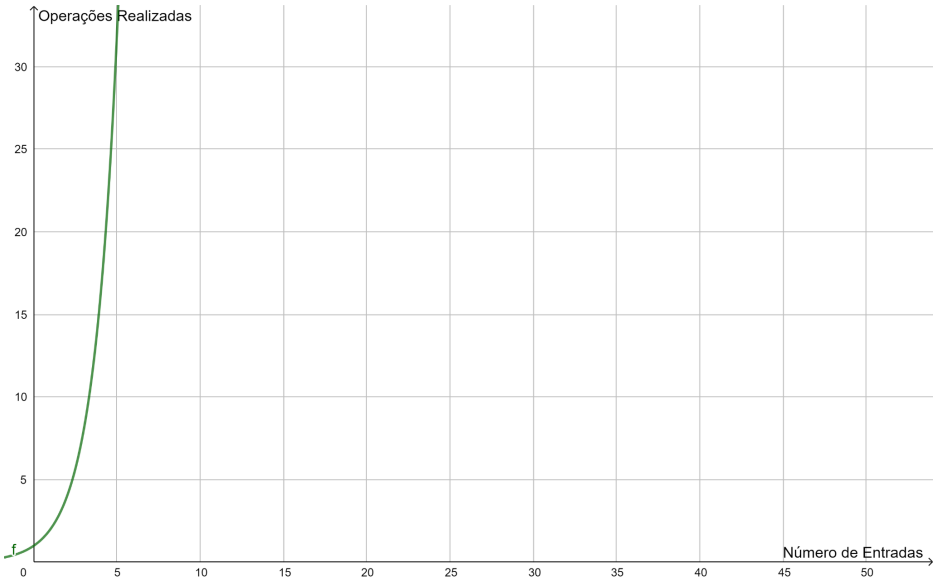

Figura 3.2 - Gráfico Entrada X Operações com zoom maior*

A **Figura 4** representa o gráfico Número de Entradas Disponíveis por Tempo de Execução em Minutos, que obedece a equação aproximada de $2^{n-23,97} + 0,02$. Para uma árvore com 40 espaços para as bolinhas, foi estimado um tempo de 73.324 minutos ou 50,919 dias.

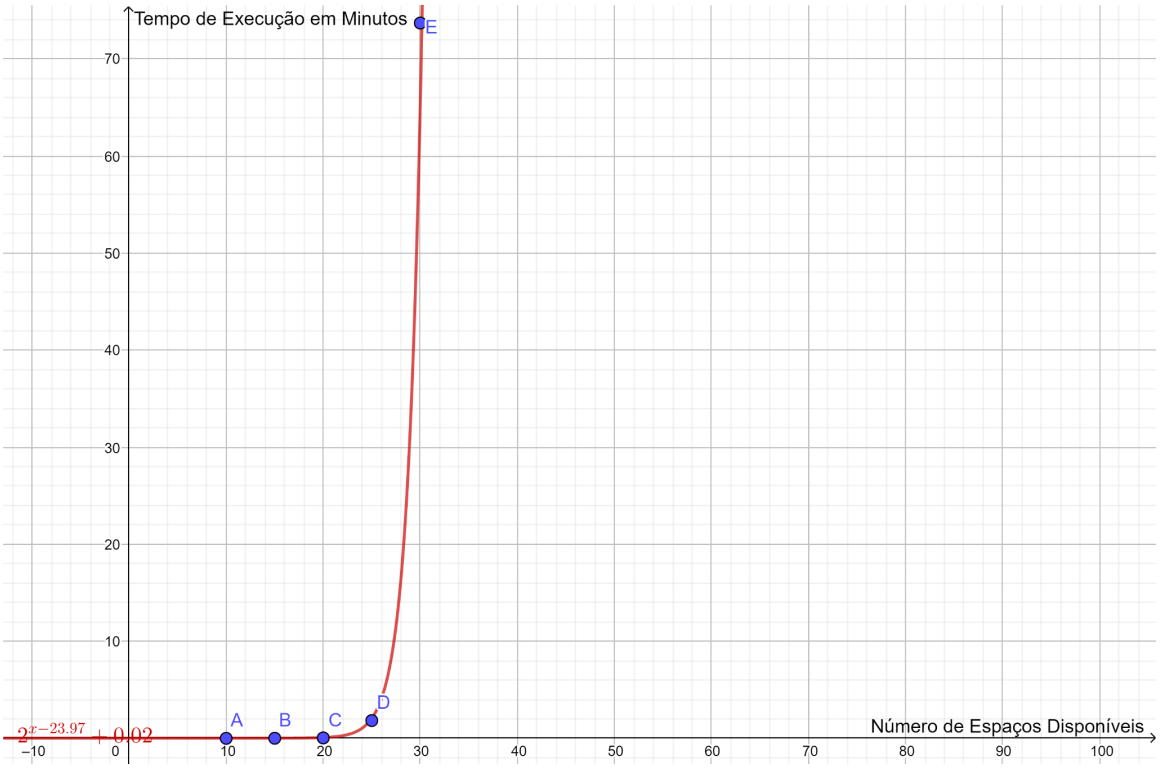

Figura 4 - Gráfico Espaços X Tempo em Minutos*

*Considerar apenas o quadrante positivo do gráfico

5. Conclusão

Por fim, com o término do trabalho, obtivemos um resultado que ao nosso ver atende às especificações anteriormente prescritas. Com a utilização de um código de terceiros e funções da linguagem C importadas de bibliotecas como time.h, conseguimos realizar as combinações necessárias e criar as matrizes para as comparações, juntamente com a captura do tempo de execução que é informado no término das combinações. Também lemos um arquivo e geramos sua respectiva matriz de adjacência. Como dito anteriormente, utilizamos força bruta para realizar os problemas resolvíveis, enquanto os demais foram calculados e obtidos resultados aproximados.

6. Referências

[1] Github. Disponível em: </https://github.com/> Último acesso em: 05 de novembro de 2022

[2] Gitlab. Disponível em: <https://gitlab.com/> Último acesso em 02 de novembro de 2022

[3] Stack Overflow. Disponível em <https://stackoverflow.com/> Último acesso em 05 de novembro de 2022

[4] Geeks for Geeks. Disponível em <https://www.geeksforgeeks.org/> Último acesso em 03 de outubro de 2022

[5] ZIVIANI, Nivio. **Projeto de Algoritmos com Implementações em Pascal e C.** 3ªEd. Cengage Learning, 23 junho 2010.

[6] Daemonio Labs. Disponível em <https://daemoniolabs.wordpress.com/2011/02/11/gerando-permutacoes-r-com-repeticao-em-c/> Último acesso em 03 de novembro de 2022.