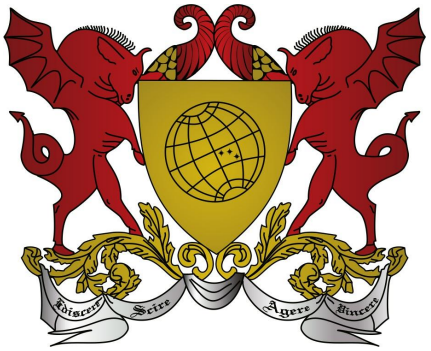


Universidade Federal de Viçosa

Trabalho Prático - Algoritmos e Estrutura de Dados: Avaliação do Desempenho de algoritmos em seu tempo de execução

Danilo Matos de Oliveira - 5073

Guilherme Broedel Zorzal - 5064

Álvaro Gomes da Silva Neto - 5095

FLORESTAL – MINAS GERAIS

2022

1. INTRODUÇÃO

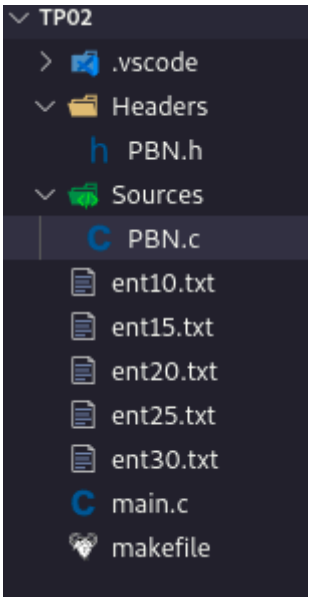
O objetivo do trabalho apresentado é a utilização e a avaliação do impacto do desempenho de algoritmos, por meio da medida de tempo de execução para diferentes valores de entrada.

Sendo assim, trabalhamos com um problema considerado intratável, pois sua solução só é possível com a avaliação de todas as possibilidades, a chamada força bruta. Portanto, neste trabalho, por intermédio de avaliação de tempo de execução e análise de complexidade de algoritmos, avaliamos o Problema das Bolinhas de Natal.

2. ORGANIZAÇÃO

Na Figura 1 é possível visualizar a organização do projeto, de forma que na pasta Sources/ está o escopo das funções do projeto, juntamente com a implementação do arquivo main, e na pasta Headers/ estão presentes os protótipos de funções utilizadas no projeto, juntamente com a constante N, que foi colocada como o valor de entrada no trabalho.

Figura 1 - Repositório do projeto.



Fonte: Elaborado pelo Autor

Para executar o projeto, criamos um makefile que realiza a criação do binário e também a execução do projeto.

O binário (ou vetor_binario) é o responsável por armazenar as possibilidades de arranjo das bolinhas. Como são utilizadas somente duas cores, as organizações possiveis das bolinhas são dadas por todas os binários presentes no intervalo das possibilidades, sendo que 1 representa uma cor e 0 representa outra cor. Num caso N = 3, por exemplo, as bolinhas poderiam ser organizadas de 000 até 111.

3.DESENVOLVIMENTO

3.1. HEADERS - PBN.h

Figura 2 - Implementação do PBN.h

```c
//PBN.h

#define N 31
//Prototipo das funcoes



//CRIA UMA MATRIZ PREENCHIDA COM -1 NA DIAGONAL PRINCIPAL E 0 NO RESTANTE
void inicia_matriz(int matriz[N][N]);

//LE O ARQUIVO DE TEXTO E MANDA OS VALORES LIDOS PARA A FUNCAO INCLUI_ARESTA
void constroi_matriz(int matriz[N][N], char *nome_arquivo);

//RECEBE UM VALOR LIDO NA MATRIZ E MODIFICA ESSA CONEXAO NA MATRIZ, SUBSTITUINDO O VALOR POR 1
void inclui_aresta(int linha, char *valor_lido, int matriz[N][N]);

//IMPRIME MATRIZ
void imprime_matriz(int matriz[N][N]);

//ATUALIZA O VETOR BINARIO QUE SERA USADO PARA FAZER AS COMPARAÇÕES
int atualiza_vetor(int *k, int *vetor);

//USANDO O VETOR BINARIO E A MATRIZ DE ADJACENCIA, VERIFICA SE AQUELA SEQUENCIA E VALIDA
int verifica_sequencia(int vetor_binario[N], int matriz_adjacente[N][N]);
```

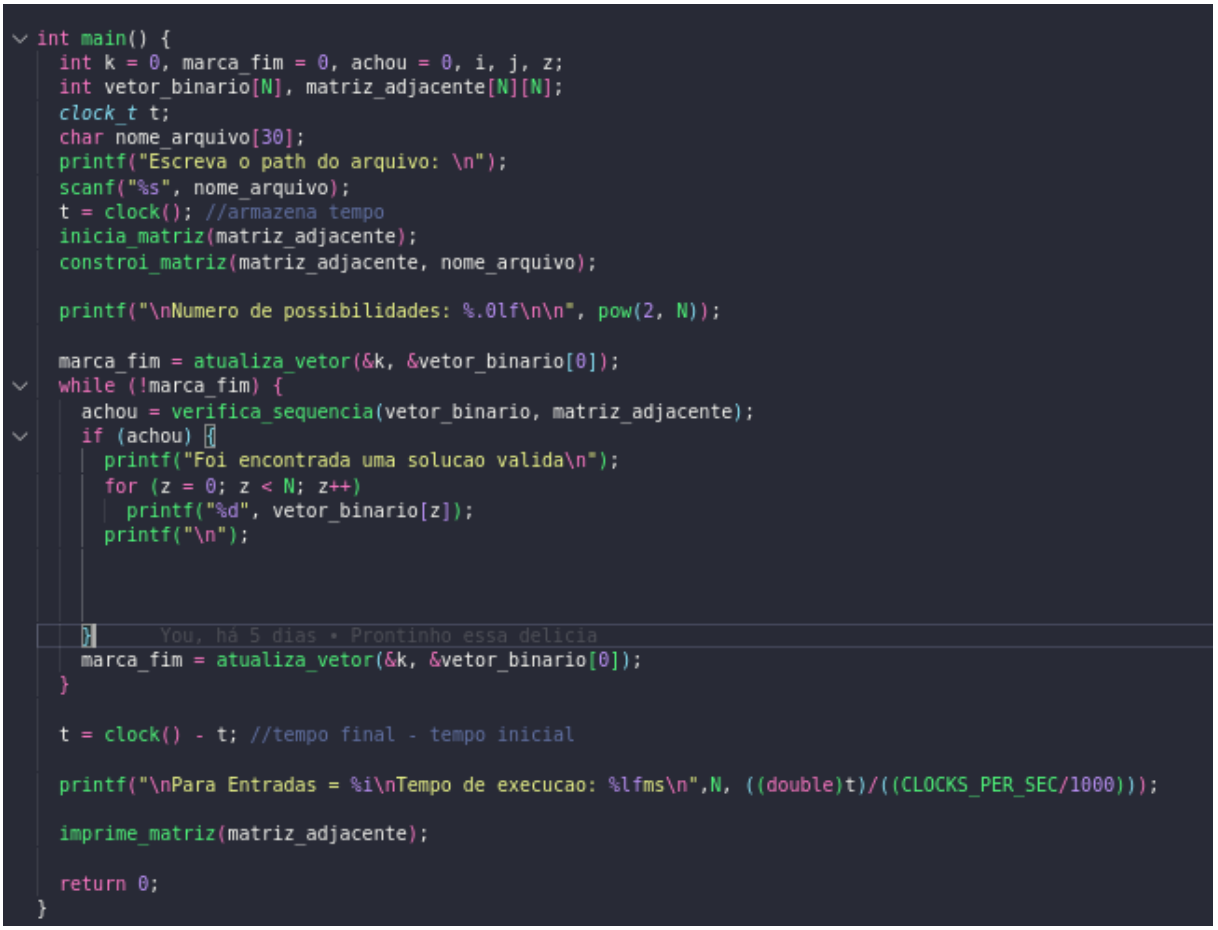
Fonte: Elaborado pelo Autor

Na figura acima, conseguimos ver o protótipo das funções que foram criadas para testar o Problema das Bolinhas de Natal. As funções criadas são utilizadas para a criação da matriz, e da leitura do arquivo de texto que será usado para preenchê-la.

.

3.2. Implementação do main.c

Figura 3 - Implementação do main.c



```c
int main() {
  int k = 0, marca_fim = 0, achou = 0, i, j, z;
  int vetor_binario[N], matriz_adjacente[N][N];
  clock_t t;
  char nome_arquivo[30];
  printf("Escreva o path do arquivo: \n");
  scanf("%s", nome_arquivo);
  t = clock(); //armazena tempo
  inicia_matriz(matriz_adjacente);
  constroi_matriz(matriz_adjacente, nome_arquivo);

  printf("\nNumero de possibilidades: %.0lf\n\n", pow(2, N));

  marca_fim = atualiza_vetor(&k, &vetor_binario[0]);
  while (!marca_fim) {
    achou = verifica_sequencia(vetor_binario, matriz_adjacente);
    if (achou) {
      printf("Foi encontrada uma solucao valida\n");
      for (z = 0; z < N; z++)
        printf("%d", vetor_binario[z]);
      printf("\n");



    }      You, há 5 dias • Prontinho essa delicia
    marca_fim = atualiza_vetor(&k, &vetor_binario[0]);
  }

  t = clock() - t; //tempo final - tempo inicial

  printf("\nPara Entradas = %i\nTempo de execucao: %lfms\n",N, ((double)t)/((CLOCKS_PER_SEC/1000)));

  imprime_matriz(matriz_adjacente);

  return 0;
}
```

Fonte: Elaborado pelo Autor

Na figura acima, conseguimos ver a implementação main.c que inicialmente lê do teclado o nome do arquivo de texto, depois ele inicializa a matriz e já começa a preenchê-la de acordo com o arquivo de texto. Após isso, é inicializado um vetor binário que será usado para fazer as comparações dentro da função verificação sequência, e cada solução válida é imprimida na tela. Ao final, a medida do tempo de execução é finalizada, e é exibida na tela junto com a matriz adjacente que foi montada.

4.RESULTADOS

4.1.Análise do Algoritmo

Figura 4 - Funções usadas para o Cálculo de Complexidade

```
int atualiza_vetor(int *k, int *vetor) {
  // contadores
  int i;
  int num = *(k);

  // avalia tamanho do binario
  if (num > (pow(2, N)-1)) { // ********** gasto = 1

    printf("\nTodas as possibilidades foram testadas\n");
    return 1;
  }

  // inicia vetor vazio(decimal = 0)
  for (i = 0; i < N; i++) { // ************** como nao ha ifs dentro do for, nao ha gasto
    *(vetor + i) = 0;

  }


  i = 0;
  while (num > 0) { // ************************gasto = como nao ha condicionais dentro do loop, gasto = 0
    // obtem o resto da divisão de num por 2
    vetor[N - i - 1] = num % 2;
    i++;
    num = num / 2;
  }



  // atualiza o valor k para a proxima repetiçao (para o vetor ser preenchido
  // corretamente)
  *(k) += 1;
  return 0;
}
//gasto total da funcao = 1
```

```
int verifica_sequencia(int vetor_binario[N], int matriz_adjacente[N][N]) {
  int i, j, z;
  for (i = 0; i < N; i++) { //repete n vezes
    for (j = 0; j < N; j++) { //repete n vezes

    /*
    Perceba o seguinte: o pior caso será quando a linha de execucao executa todos os 3 ifs.
    Primeiramente: independente da formula de complexidade dessa funcao, podemos garantir que
    todas as linhas terao um caso onde somente o primeiro if é executado.
    Em segundo lugar: a unica forma de (tirando a excecao mostrada no item anterior) a linha de
    execucao entrar nos dois ifs é se todo item da matriz_adjacente,
    com exceçao dos da diagonal principal, for 1. Logo, temos 3*n^2 que representa o pior caso.
    Subtraindo dos n casos onde i == j, temos 3n^2 - n.          Ou, agora * Uncommitted changes
    Ja o melhor caso é quando todos os itens da matriz sao 0. Logo, temos 3n - n = 2n.
    */

      if (i != j) {  //*********************************gasto 1
        if (matriz_adjacente[i][j] ==
          1) { // Procura dentro da matriz quais elementos são adjacentes
          //*************************************gasto 1
          if (vetor_binario[i] ==
              vetor_binario[j]) { // Compara de acordo com a matriz se as cores
                                  // são iguais em lados adjacentes

            return 0;
          }
        }
      }
    }
  }//gasto = melhor caso:  3n - n = 2n. Pior caso: 3n^2 - n
  return 1;
}
//gasto total da funcao = pior caso: 3n^2 - n. Melhor caso =  3n - n = 2n
```

Fonte: Elaborado pelo Autor

Conforme pode ser visualizado nas imagens acima, o programa executa estas funções que são também as mais custosas, em termo de tempo de execução, pois são feitas diversas comparações com o vetor binário criado, e as comparações feitas juntamente com a matriz de adjacência, a equação que descreve a complexidade do algoritmo.

Além disso, dada as devidas comparações, podemos traçar a complexidade que é exibida abaixo, sendo que, para elaborar essas funções, pensamos tanto no pior caso e também no melhor caso de entradas. Adicionalmente, gostaríamos de destacar que o detalhamento acerca das funções de complexidade foi feito dentro dos arquivos do próprio código.

Figura 5 - Função de complexidade do Pior Caso

$$f(n) = 2^n(3n^2 - n + 2) + n^2 + 3n(n-1) + 2$$

Fonte: Elaborado pelo Autor

Figura 6 - Função de Complexidade do Melhor Caso

$$f(n) = 2^n(2n + 2) + n^2 + 3n + 2$$

Fonte: Elaborado pelo Autor

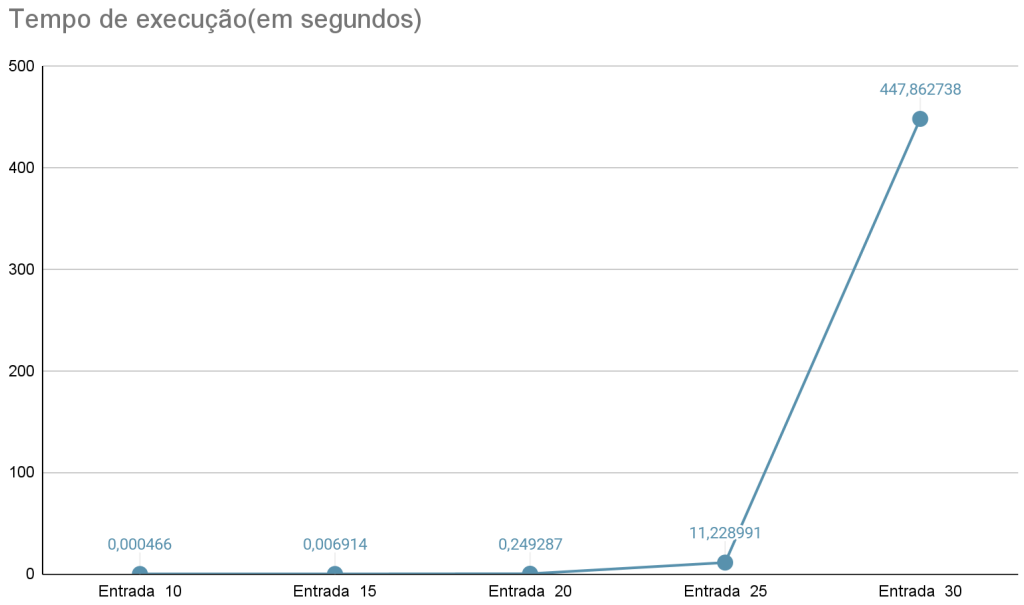
4.2.Tempo de execução

O programa em questão foi testado em uma máquina que possui as seguintes configurações:

OS: Debian 11
Kernel: x86_64 Linux 5.10.0-15-amd64
Interface: Gnome 3.38.5
CPU: AMD® Ryzen 5 3500u with radeon vega mobile gfx × 8
RAM: 20,480 MB

Figura 7 - Gráfico de tempo de execução

	Tempo de execução(em segundos)
Entrada 10	0.000466
Entrada 15	0.006914
Entrada 20	0.249287
Entrada 25	11.228991
Entrada 30	447.862738



Fonte: Elaborado pelo Autor

Conforme pode ser visualizado nas imagens acima, o programa foi testado até o valor de 30 entradas, e foi gerado um gráfico com o respectivo tempo de execução. A razão pela qual testamos os valores de 5 em 5, e nos limitamos a 30 valores de entrada, é devido ao fato de ser um problema intratável de forma que ele é resolvido usando a força bruta. Sendo assim, valores grandes de entrada tendem a ter um tempo descomunal de execução.

Além disso, é um problema $O(f(n)) = O(2^n)$, ou seja, é um problema que possui complexidade de algoritmo exponencial, conforme demonstrado no gráfico. Desta forma, o tempo de execução não é razoável para valores superiores a 30, então estas entradas não foram testadas.

5.CONCLUSÃO

Sendo assim, este projeto foi realizado utilizando conceitos como Complexidade de Algoritmos e Medida de Tempo de execução, nos permitindo ter uma grande capacidade prática nestes conteúdos.

Conforme a implementação foi realizada, percebeu-se um aumento significativo em nossa capacidade de analisar algoritmos e sua complexidade, pois, há casos em que não é só necessário que o código funcione, e sim que ele seja eficiente na tarefa que ele se propõe, e com o desenvolvimento deste trabalho prático, podemos utilizar esse conhecimentos na prática.

6.REFERÊNCIAS

ZIVIANI, Nivio et al. **Projeto de algoritmos: com implementações em Pascal e C**. Luton: Thomson, 2004.

CORMEN, Thomas. **Desmistificando algoritmos**. Elsevier Brasil, 2017.