Projeto B - Sequencial

Anderson Araujo de Oliveira, N USP: 11371311

1 Parte I - Discussão sobre o código

1.1 Leitura do Arquivo/Criação do Grafo

O código começa verificando se o arquivo foi aberto corretamente. Em seguida, ele lê o número de nós e arestas presentes no grafo, extraindo essas informações do arquivo.

```
    FILE *arquivo = fopen(argv[1], "r");

    if (arquivo == NULL) {//verificando se consegue abrir o arquivo
        fprintf(stderr, "Erro ao abrir o arquivo.\n");
        return 2;
    }

    int read1 =fscanf(arquivo, " %d", &N);
    int read2 = fscanf(arquivo, " %d", &M);

    if (read1!=1) {//verificando se os parametros de arestas e nos
        fprintf(stderr, "Erro na leitura da leitura da quantidade de n  .\n");
        return 3;
    }
    if (read2!=1) {
        fprintf(stderr, "Erro na leitura na leitura da quantidade de aresta.\n");
        return 4;
    }
```

1.1.1 Lista de adjacência

Nessa etapa o código lê as conexões entre os nós do grafo(as arestas). Para representar as arestas, ele utiliza uma lista de adjacência em vez de uma matriz de adjacência. Se fossemos utilizar uma matriz de adjacência, a complexidade seria $O(N^2)$, devido à verificação se um nó é vizinho, onde N é o número de nós. Com a lista de adjacência, a complexidade se torna $O(N)$, não tendo que fazer essa verificação.

```
    //Valor medio de vizinhos como tamanho das listas de adjacencia
    int k = (int)ceil(M/N);
    //criando o grafo usando lista de adjacencia
    Grafo *grafo = criarGrafo(N, k);
    //adicionando as arestas
    for (int i = 0; i < M; i++) {
        if (fscanf(arquivo, "%d %d", &u, &v) != 2) {
            fprintf(stderr, "Erro ao ler aresta.\n");
            return 1;
        }
        adicionarAresta(grafo, u - 1, v - 1); // Ajusta para  ndices  baseados em 0
    }
```

Função para criar o grafo.

```
// Estrutura para representar a lista de adjac ncia
typedef struct {
    int *tamanho_lista;
    int *indice;
    int **lista;
```

```
    int N;
} Grafo;

// Fun   o para criar um grafo com lista de adjac ncia
Grafo *criarGrafo(int N, int k) {
    //gerando o grafo
    Grafo* grafo = (Grafo *)malloc(sizeof(Grafo));
    grafo->N = N;
    //lista com tamanhos das listas
    grafo->tamanho_lista = (int *)malloc(N * sizeof(int));
    //salvando os indices
    grafo->indice = (int *)malloc(N * sizeof(int));
    //criando a lista de adjacencia

    grafo->lista = (int **)malloc(N * sizeof(int *));

    for (int i = 0; i < N; i++) {

        //criando vetor
        grafo->lista[i] = (int *)malloc(k * sizeof(int));

        //primeiro chute para a lista de adjcencia
        grafo->tamanho_lista[i] = k;

        //criando o primeiro ponteiro
        grafo->indice[i]=0;

        for (int j =0; j < k; j++){
            grafo->lista[i][j] = -1;
        }
    }
    return grafo;
}
```

Função para adicionar aresta.

```
// Fun   o para adicionar uma aresta ao grafo
void adicionarAresta(Grafo *grafo, int origem, int destino) {

    //adicionando aresta
    grafo->lista[origem][grafo->indice[origem]]=destino;

    //verificando o tamanho
    if(grafo->indice[origem]==grafo->tamanho_lista[origem]-1){

        //dobrando o tamanho da lista de adjacencia
        grafo->tamanho_lista[origem]*=2;
        grafo->lista[origem]= (int *)realloc(grafo->lista[origem],grafo->tamanho_lista[
   origem] * sizeof(int));

        for(int j = grafo->indice[origem]+1;j < grafo->tamanho_lista[origem];j++){
            grafo->lista[origem][j]=-1;
        }
    }
    //alterando o ponteiro
    ++grafo->indice[origem];
}
```

1.2 Calculo da eficiência

1.2.1 Calculo da eficiência

Esta etapa crucial no código, calcular a eficiência do grafo. Mas antes de calcular a eficiência a equação abaixo, precisamos determinar as distâncias entre todos os pares de nós. É denotada por d_{ij}, representa o menor número de arestas que devemos percorrer para ir do nó i ao nó j.

$$E = \frac{1}{N(N-1)} \sum_{i=0}^{N-1} \sum_{\substack{j=0 \\ j \neq i}}^{N-1} \frac{1}{d_{ij}} \tag{1}$$

O código utiliza o algoritmo de busca em largura (BFS) para calcular distância. A ideia é explorar o grafo camada por camada, partindo do nó de origem. Os nós vizinhos são armazenados em uma fila, garantindo a ordem de visitação. Durante a busca, as distâncias de cada nó em relação à origem são calculadas e armazenadas. O processo continua até que todos os nós acessíveis a partir da origem tenham sido visitados e suas distâncias calculadas.

```
// Fun   o para calcular a efici ncia do grafo direcionado
float calcularEficiencia(Grafo *grafo) {
    double eficiencia = 0;
    int *distancia = (int *)calloc(grafo->N , sizeof(int));

    for (int origem = 0; origem < grafo->N; origem++) {//percorrer todos os nos
        buscaEmLargura(grafo, origem, distancia);//algoritmo de busca largura
        for (int i = 0; i < grafo->N; i++) {
            if (i != origem && distancia[i] > 0) {
                eficiencia += 1.0 / distancia[i];
                distancia[i]=0;
            }
        }
    }
    free(distancia);
    eficiencia = eficiencia / (grafo->N * (grafo->N - 1));//calculo de efici ncia
    return eficiencia;
}
```

A abaixo a função de busca em largura no grafo. A escolha desse algoritmo se deve à sua capacidade de explorar os nós permitindo registrar as distâncias entre eles. Essa informação sobre as distâncias é crucial para a etapa de calcular a eficiência do grafo.

```
// Fun   o para realizar a busca em largura a partir de um v rtice de origem
void buscaEmLargura(Grafo *grafo, int origem, int *distancia) {
    int u,v;//criando variaveis para auxiliar
    int *visitado = (int *)calloc(grafo->N, sizeof(int));//criando vetor com os valores
   visitados

    Fila *q = criarFila(grafo->N);//criando fila com tamanho N

    visitado[origem] = 1;//colocar o primeiro no como visitado
    distancia[origem] = 0;//colocando distancia 0 no primeiro no
    enfileirar(q, origem);//adcionionando na fila

    while (!filaVazia(q)) {//parar while quando a fila tiver vazia

        u = desenfileirar(q);
        grafo->indice[u]=0;//come ar lista de vizinhos
        v = grafo -> lista[u][grafo->indice[u]];//inserir o primeiro vizinho
```

```
17
18          while (v != -1) {//continuar at  todos vizinhos sejam visitados
19
20              if (visitado[v]==0) {
21                  visitado[v] = 1;//colocar como vizitado
22                  distancia[v] = distancia[u] + 1;//calculo da distancia
23                  enfileirar(q, v);//adicionar vizinho
24              }
25
26              ++grafo->indice[u];// ir para o proximo vizinho
27              v = grafo->lista[u][grafo->indice[u]];
28          }
29      }
30      liberarFila(q);// liberar a fila e visitados
31      free(visitado);
32  }
```

Funções e estrutura da fila. A fila é utilizada para armazenar os nós vizinhos durante a busca em largura.

```
1
2   // Estrutura para representar uma fila com capacidade din mica
3   typedef struct {
4       int *itens;
5       int frente, tras;
6       int tamanho;
7       int capacidade;
8   } Fila;
9
10  // Fun  o para criar uma fila vazia com capacidade inicial
11  Fila *criarFila(int capacidade_inicial) {
12      Fila *q = (Fila *)malloc(sizeof(Fila));
13      q->itens = (int *)malloc(capacidade_inicial * sizeof(int));
14      q->frente = 0;
15      q->tras = -1;
16      q->tamanho = 0;
17      q->capacidade = capacidade_inicial;
18      return q;
19  }
20
21
22  int filaVazia(Fila *q) {
23      return q->tamanho == 0;// Fun  o para verificar se a fila est  vazia
24  }
25
26  // Fun  o para enfileirar um elemento na fila,
27  void enfileirar(Fila *q, int valor) {
28      if (q->tamanho == q->capacidade) {// aumentando a capacidade se necess rio
29          q->capacidade *= 2;
30          q->itens = (int *)realloc(q->itens, q->capacidade * sizeof(int));
31      }
32      q->tras = (q->tras + 1) % q->capacidade;
33      q->itens[q->tras] = valor;
34      q->tamanho++;
35  }
36
37  // Fun  o para desenfileirar um elemento da fila
38  int desenfileirar(Fila *q) {
39      if (filaVazia(q)) {//verificando se est  vazia a lista
40          fprintf(stderr, "Fila vazia!\n");
41          exit(1);
```

```
    }
    int item = q->itens[q->frente];//puxando a variavel
    q->frente = (q->frente + 1) % q->capacidade;
    q->tamanho--;
    return item;
}


void liberarFila(Fila *q) {
    free(q->itens);// Fun   o para liberar a mem ria alocada para a fila
    free(q);
}
```

1.2.2 Etapa final

Imprimir resultado e o tempo.

```
fclose(arquivo);
    //come ando leitura do tempo
    clock_gettime(CLOCK_MONOTONIC, &inicio);
    //calculo da efici ncia
    double eficiencia = calcularEficiencia(grafo);
    //terminando a leitura do tempo
    clock_gettime(CLOCK_MONOTONIC, &fim);

    tempo_decorrido = (fim.tv_sec - inicio.tv_sec) +
                      (fim.tv_nsec - inicio.tv_nsec) / 1e9;
    //imprindo resultado no terminal
    printf("%f %f\n",eficiencia,tempo_decorrido);

    liberarGrafo(grafo);

    return 0;
}
```

2 Parte II - Comparações

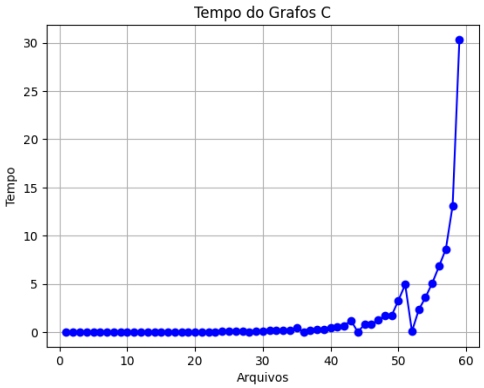
2.1 Tempo em função dos arquivos

O estudo compara três diferentes representações de grafos em código: matriz de adjacência e duas variações de lista de adjacência. As listas de adjacência se diferenciam na forma como armazenam os vizinhos: uma utiliza vetores e a outra, ponteiros para o próximo vizinho.

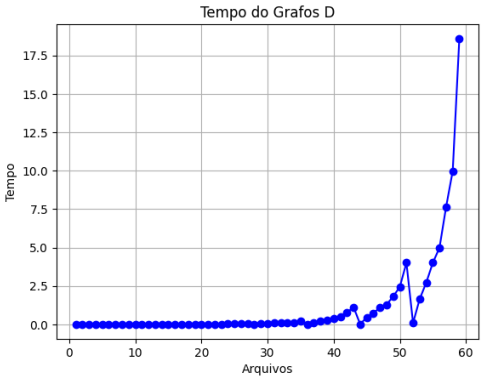
As imagens ilustram graficamente o desempenho de cada representação, exibindo o tempo de execução em função do arquivo de entrada (representado pelos grafos B, C e D) e o número do arquivo. Essa análise gráfica permite comparar diretamente a eficiência de cada método na representação do grafo. Os códigos foram executados em processador 13th Gen Intel(R) Core(TM) i5-13450HX, e o tempo foi considerado em segundo.

As figuras 1 e 2 ilustram o desempenho da representação do grafo utilizando lista de adjacência com vetores.

(a) Grafos B



(b) Grafos C



(c) Grafos D

Figura 1: Código utilizando lista de adjacência por vetor

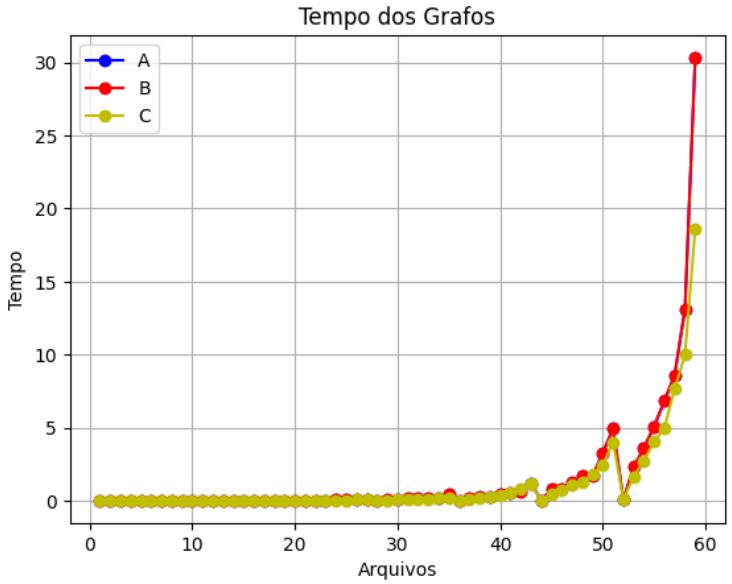
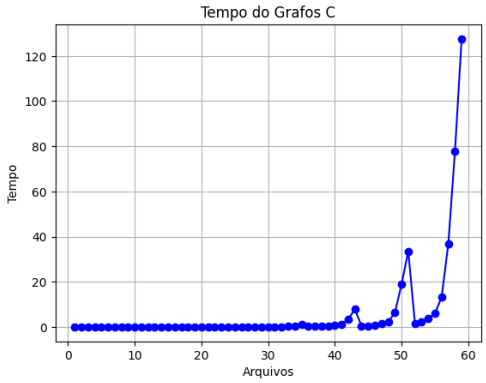


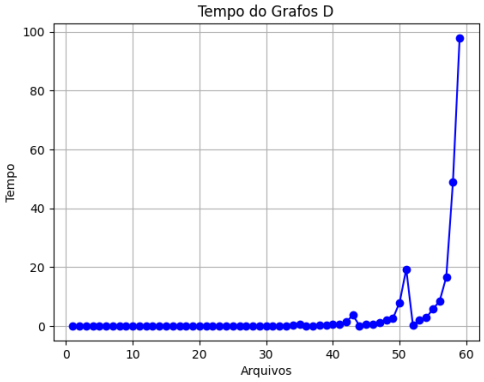
Figura 2: Código utilizando lista de adjacência por vetor

As figuras 3 e 4 ilustram o desempenho da representação do grafo utilizando lista de adjacência por ponteiro.

(a) Grafos B



(b) Grafos C



(c) Grafos D

Figura 3: Código utilizando lista de adjacência por ponteiro

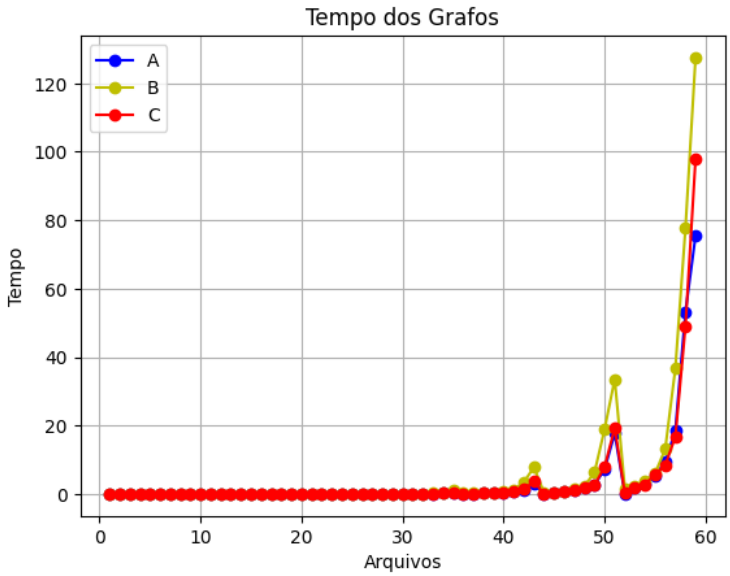
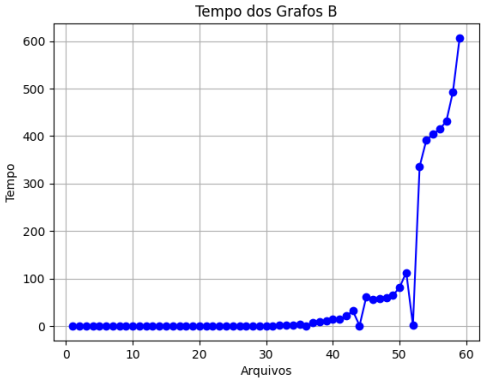
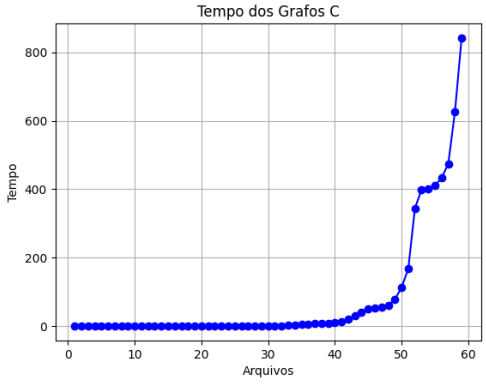


Figura 4: Código utilizando lista de adjacência por ponteiro

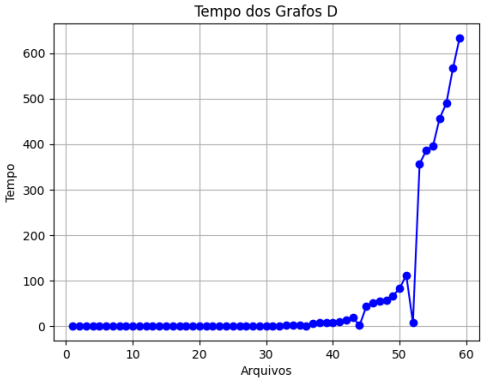
As figuras 5 e 6 ilustram o desempenho da representação do grafo utilizando Matriz de adjacência.



(a) Grafo B



(b) Grafo C



(c) Grafo D

Figura 5: Código utilizando Matriz de adjacência

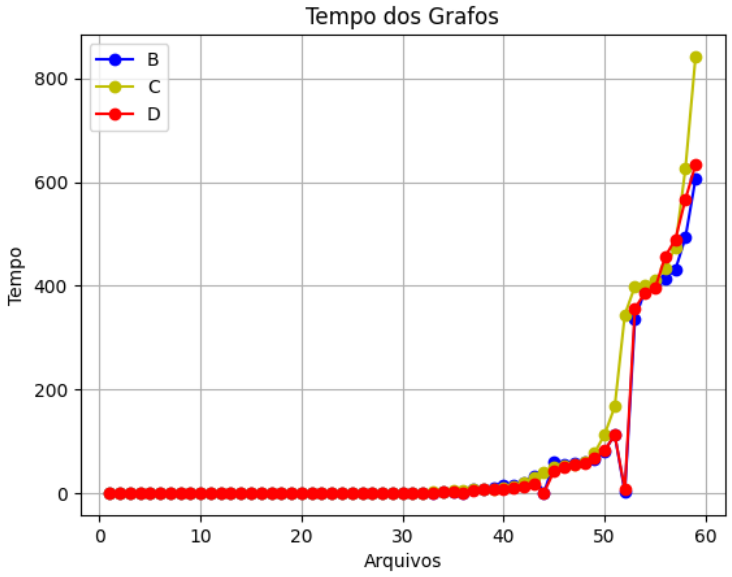


Figura 6: Código utilizando Matriz de adjacência

2.2 Complexidade dos algoritmos

Ao analisar a complexidade, consideramos que para cada um dos N nós, serão calculados as distâncias pelo BFS, ou seja, N vezes algo proporcional a quantidade de arestas M, a complexidade fica $O(NM)$.

No caso da matriz de adjacência terá uma etapa a mais para verificar se um Nó é vizinho, isso terá um custo computacional N, para a matriz de adjacência a complexidade fica $O(N^2M)$.

A fim de analisar o desempenho dos códigos que utilizam matriz de adjacência e lista de adjacência, serão conduzidos dois conjuntos de testes, no primeiro, o número de arestas será mantido fixo, o segundo, o número de nós será fixo.

A figura 7 mostra crescimento do algoritmo quando a quantidade nos(N) é constante.

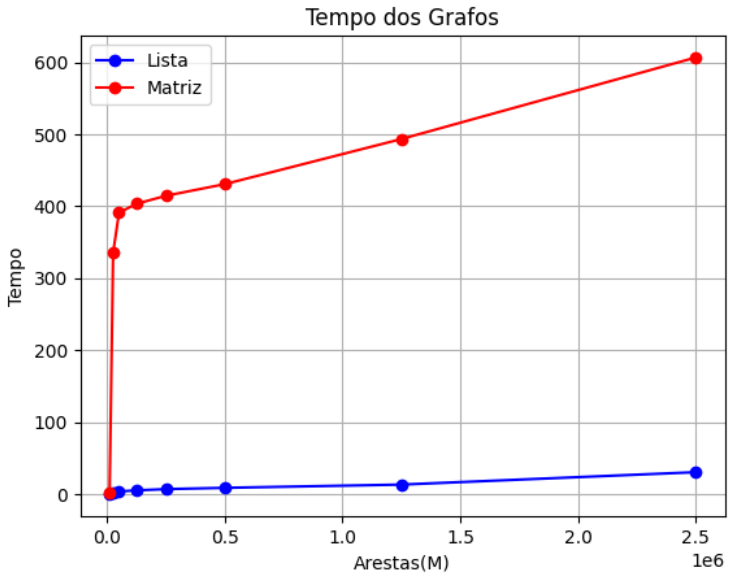


Figura 7: Tempo do código quando quantidade é fixado com N=10000

A figura 8 mostra crescimento do algoritmo quando a quantidade arestas(M) é constante.

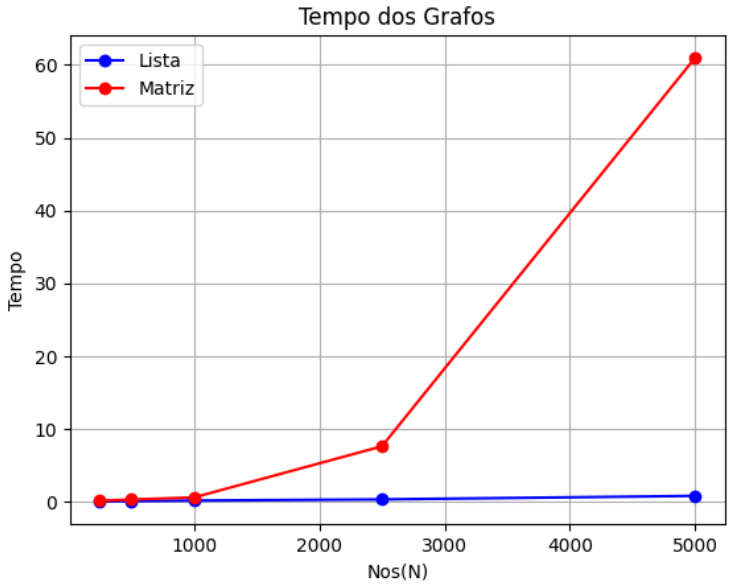


Figura 8: Tempo do código quando quantidade é fixado com M=12500

Os códigos tiveram um desempenho como esperado, mostrando que representar o grafo utilizando lista de adjacência é mais eficiente que matriz de adjacência, além de combinar algoritmo de busca em largura.