UNIVERSIDADE FEDERAL DE VIÇOSA
CAMPUS FLORESTAL

PABLO FERREIRA -3480
SAMUEL SENA - 3494

TRABALHO PRÁTICO 1

FLORESTAL
2019

UNIVERSIDADE FEDERAL DE VIÇOSA

CAMPUS FLORESTAL

PABLO FERREIRA -3480

SAMUEL SENA - 3494

TRABALHO PRÁTICO 1

Relatório prático apresentado a disciplina de projeto e análise de algoritmos, do curso Ciência da Computação da Universidade Federal de Viçosa — Campus Florestal.

Prof.: Daniel Mendes Barbosa

FLORESTAL

2019

Sumário

Introdução

O trabalho apresentado a seguir entrega um algoritmo executado em modo texto, com um menu interativo pelo teclado. O algoritmo tem como principal objetivo a análise de labirintos e através de tentativas exaustivas com o uso de *backtracking*, concluir se o labirinto tem ou não saída.

Inicialmente para se executar o programa, é necessário realizar a compilação do código fonte em C. Para isso, em algum terminal Linux execute o "*makefile"* da seguinte forma:

Para compilar:

$ make

E para executar:

$ make run

Há também a possibilidade de compilação manual, execute o seguinte código em algum terminal Linux devidamente navegado até a pasta contendo o arquivo "main.c":

$ gcc main.c -o EXEC Sources/menu.c Sources/labirinto.c Sources/dados.c Sources/gerador.c

E em seguida para executar:

$ ./EXEC

Sobre a utilização do modo analise que contabiliza o número total de chamadas recursivas que foram feitas e o nível máximo de recursão alcançado durante toda a solução, deverá abrir o arquivo main.c e trocar o #MODOANALISE para 1 caso queira utilizar e para 0 caso não queira.

Figura 01

```
9
10  #define MODOANALISE 1  //SETAR 1 PARA ATIVAR, 0 PARA DESATIVAR
11
```

Fonte: main.c

Desenvolvimento

Para leitura do arquivo, utilizamos a seguinte lógica: primeiro utilizamos a função *fscanf* para ler os 3 primeiros valores inteiros da função e adicionamos uma quebra de linha para continuarmos lendo o resto. Após isso, inicializamos a matriz labirinto com os valores passados por parâmetro. E então chegamos na parte mais difícil, onde utilizamos uma estrutura de repetição que tinha como condições de parada quando chegasse no final do arquivo e houvesse erro de leitura, encerra a repetição. E dentro dela, utilizamos uma variável chamada *valorAux* que recebe um caractere passado pela função *fgetc()* e após isso a variável *valor* converte para um valor inteiro utilizando calculo da tabela asc. E enquanto a primeira variável não recebe uma quebra de linha, incrementamos *j* e inserimos o valor na sua respectiva posição e mantemos o valor de *i*, porém quando isso não ocorre, incrementamos o *i* e zeramos o valor de *j*.

Figura 02

```
// lemos a primeira linha e armazenamos os valores das linhas colunas e  a quantidade de chaves que o estudante tem
fscanf(arq,"%d %d %d\n", &linhaArq, &colunaArq, &quantChaveArq);

labirinto = IniciarLabirinto(linhaArq, colunaArq, &itens, quantChaveArq); // criamos uma matriz com o tamanho passado

while(!feof(arq) && !ferror(arq)){// enquanto não for fim do arquivo e não for erro de leitura, continuamos lendo
  valorAux = fgetc(arq); //valorAux recebe um caracter
  valor = valorAux-48; //esse caracter é convertido para int e passado para valor
  if(valor == EOF){ //se esse valor for o final da leitura, encerramos o while
    break;
  }
  /*sempre que o valorAux for uma quebra de linha, aumentamos a linha e zeramos a coluna
  exemplo:
  012 <- quando chegar aqui, o i vai passar a valer 1 e a coluna que era 3 voltara para 0
  345
  678
  */
  if(valorAux == '\n'){
    i++;
    j = 0;
  }else{
    InserirPosicao(labirinto, i, j, valor);
    j++;
  }
}
```

Fonte: main.c

Para processarmos e exibirmos a resposta do labirinto, conferimos se o arquivo existe e caso não existe, exibimos uma mensagem de alerta e voltamos ao menu inicial. Caso o arquivo existe, procuramos em qual linha e coluna o estudante está, e então passamos como parâmetro o que for necessário para movimentar o estudante e após isso, exibimos o labirinto percorrido pelo estudante e os dados que contem quantas vezes ele andou.

Figura 03

```
case 2: // Processar e exibir resposta

  printf("%s\n", arquivo);
  if(strlen(arquivo) == 0){
    printf("Por favor carregue antes um arquivo de dados!\n");
    system("read -p 'Pressione Enter para continuar...' var");
    break;
  }
  else{
    linhaEstudante = LinhaEstudante(labirinto, linhaArq, colunaArq);
    colunaEstudante = ColunaEstudante(labirinto, linhaArq, colunaArq);

    if(MODOANALISE){
      numRecurcoes = -1; //inicializado como -1 para desconsiderar primeira chamada realizada pelo main
      Movimenta_Estudante_Analise(labirinto, &itens, linhaEstudante, colunaEstudante, linhaArq, colunaArq, &dados,&numRecurcoes);
      printf("\n\tMODO ANALISE!\n-->O numero total de chamadas recursivas foi de: %lld\n\n",numRecurcoes);
    }else{
      Movimenta_Estudante(labirinto, &itens, linhaEstudante, colunaEstudante, linhaArq, colunaArq, &dados);
    }
    ImprimirLabirinto(labirinto, linhaArq, colunaArq);
    ImprimirDados(dados);
    system("read -p 'Pressione Enter para continuar...' var");
    system("clear");
  }
  free(labirinto);
  strcpy(arquivo, "\0");
```

Fonte: main.c

As estruturas de dados utilizadas no código foram as seguintes:

- Uma estrutura do *TipoItem* que contém uma variável que armazena a quantidade de chaves que o estudante tem.

Figura 04

```
6   typedef struct{
7       int quantChave;
8   }TipoItem;
9
```

Fonte: labirinto.h

- Uma estrutura do *TipoDados* que contém três variáveis que armazenam a quantidade de movimentações do estudante, a condição de que ele consegue sair de determinada posição e a condição de ultima coluna.

Figura 05

```
4   typedef struct{
5       int quantMovimentacao;
6       int consegueSair;
7       int ultimaColuna;
8   }TipoDados;
9
```

Fonte: dados.h

Para simplificar as condições do backtracking, encapsulamos estas em várias funções que retornam 1 para verdadeiro e 0 para falso. Assim, montamos a função *Movimenta_Estudante* da seguinte forma:

- Conferimos se o estudante chegou no final do labirinto, caso tenha chegado, chamamos uma função para passar os dados finais, que são a ultima coluna, incremento de mais uma movimentação e a condição de ter saída verdadeira. Após isso, marcamos essa posição e exibimos essa movimentação e retornamos verdadeira. `j >= coluna || i >= linha || j < 0`
- Conferimos se o estudante ultrapassou os limites do labirinto, ou seja, em uma posição que não existe e então armazenamos falso na condição de conseguir sair do labirinto.
- Conferimos se o estudante está em uma posição válida olhando se a posição que ele tá não é uma parede e nem uma porta fechada (área acessível com chave), e então incrementamos a quantidade de movimentos do estudante e exibimos essa movimentação.
- Conferimos se a posição atual do estudante é uma porta fechada e se ele tem chaves suficientes para abrir essa porta, caso tenha, incrementamos mais uma movimentação, exibimos essa movimentação, marcamos como um local aberto e diminuímos a quantidade de chaves.

Figura 06

```
/*Função principal do programa, onde conferimos sempre se o estudante já chegou no final do labirinto e utilizamos o backtracking*/
int Movimenta_Estudante(int ** labirinto, TipoItem *itens, int i, int j, int linha, int coluna, TipoDados * dados){
    if(ChegouNoFim(labirinto, i, j)){ /*O estudante chegou no final do labirinto*/
        DadosFinais(dados, j);
        MarcarPosicao(labirinto, i, j);
        printf("Linha: %d Coluna: %d\n", i, j);
        return 1;
    }
    if(UltrapassouLimites(i, j, linha, coluna)){ //Posição fora do espaço do labirinto
        dados->conseguеSair = 0;
        return 0;
    }
    if(!EhParede(labirinto, i, j) && !EhPortaFechada(labirinto, i, j)){ //posição valida
        dados->quantMovimentacao++;
        printf("Linha: %d Coluna: %d\n", i, j);
    }
    if(EhPortaFechada(labirinto, i, j) && itens->quantChave > 0){ //chegou em uma posição que tem uma porta fechada, então abri a porta
      dados->quantMovimentacao++;
      printf("Linha: %d Coluna: %d\n", i, j);
      itens->quantChave--;
      AbrirPorta(labirinto, i, j);
    }
```

Fonte: labirinto.c

Após isso, conferimos se é uma posição válida, e caso seja, marcamos essa posição como já visitada e então iniciamos a movimentação para cima, para direita, para esquerda e para baixo.

Figura 07

```
if(!EhParede(labirinto, i, j) && !EstudantePassou(labirinto, i, j) && !EhPortaFechada(labirinto, i, j)){
    MarcarPosicao(labirinto, i, j);
    if(Movimenta_Estudante(labirinto, itens,i - 1, j, linha, coluna,dados)); // para cima
    else{
        if(Movimenta_Estudante(labirinto, itens, i, j + 1, linha, coluna,dados)); // para a direita
        else{
            if(Movimenta_Estudante(labirinto, itens, i, j - 1, linha, coluna,dados)); // para a esquerda
            else{
                if(Movimenta_Estudante(labirinto, itens, i + 1, j, linha, coluna,dados)); //para baixo
                else{
                    return 0;
                }
            }
        }
    }
}else{
  return 0;
```

Fonte: labirinto.c

Além disso, criamos três funções extras no programa que tem como objetivo a criação de arquivos de teste nos níveis fácil, médio e difícil. A criação das funções seguem como base a seguinte estratégia, criação de variáveis que recebem valores aleatórios que definimos como matriz de até 10x10 para nível fácil, matriz de 10x10 até 25x25 para nível médio e matriz de 25x25 até 50x50 para nível difícil. E para diferenciar os níveis, mudamos também a condição de preenchimento e quantidade de chaves recebidas de forma que fique balanceado em cada um dos níveis.

Figura 08

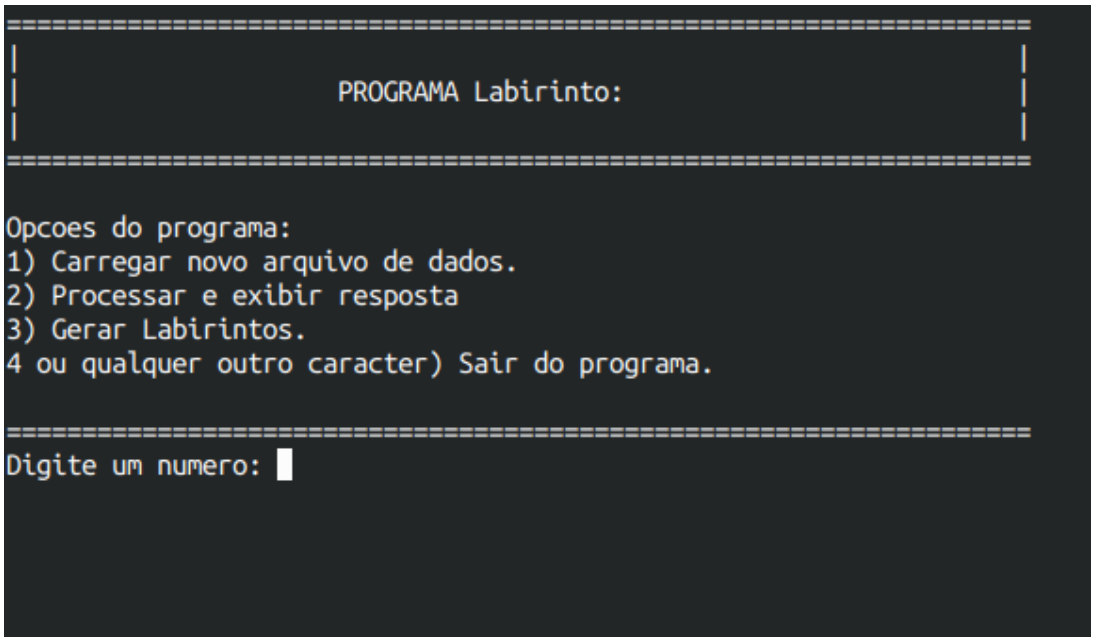
```
fprintf(arq, "%d %d %d \n", linha, coluna, quantChave);
for(int i = 0; i < linha; i++){
    for(int j = 0; j < coluna; j++){
        if( i == estudanteLinha && j == estudanteColuna){ // quando chegar na posição[i][j] correspondente
            valor = 0;
        }
        else if (i == rand() % linha && j == rand() % coluna){ // dificilmente terá portas ou paredes
            valor = 2 + rand() % 2; // somamaos 1 pra nunca cair no local onde o estudante já está
        }
        else{
            valor = 1;
        }
        fprintf(arq, "%d", valor);
    }
    fputc('\n', arq); // no final de cada linha adicionamos uma quebra de linha
}
fclose(arq);
```

Fonte: gerador.c – Função fácil

Resultados

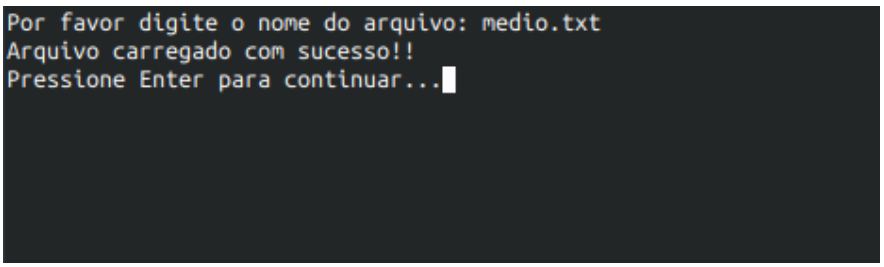
Abaixo, seguem alguns menus e resultados que gerados pelo programa, e também imagens dos arquivos de teste.

Figura 09



Fonte: Terminal Linux

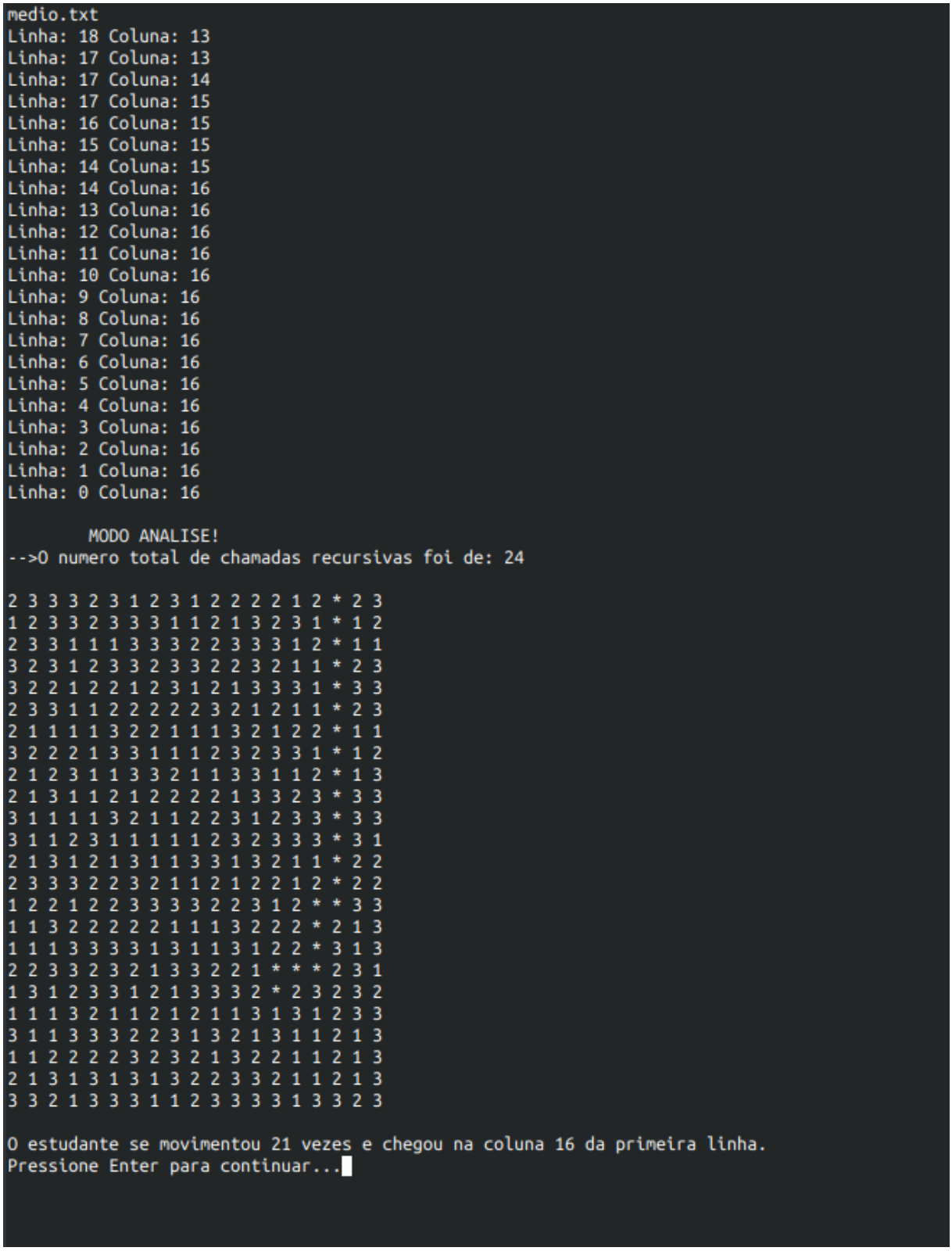
Figura 10



Fonte: Terminal Linux

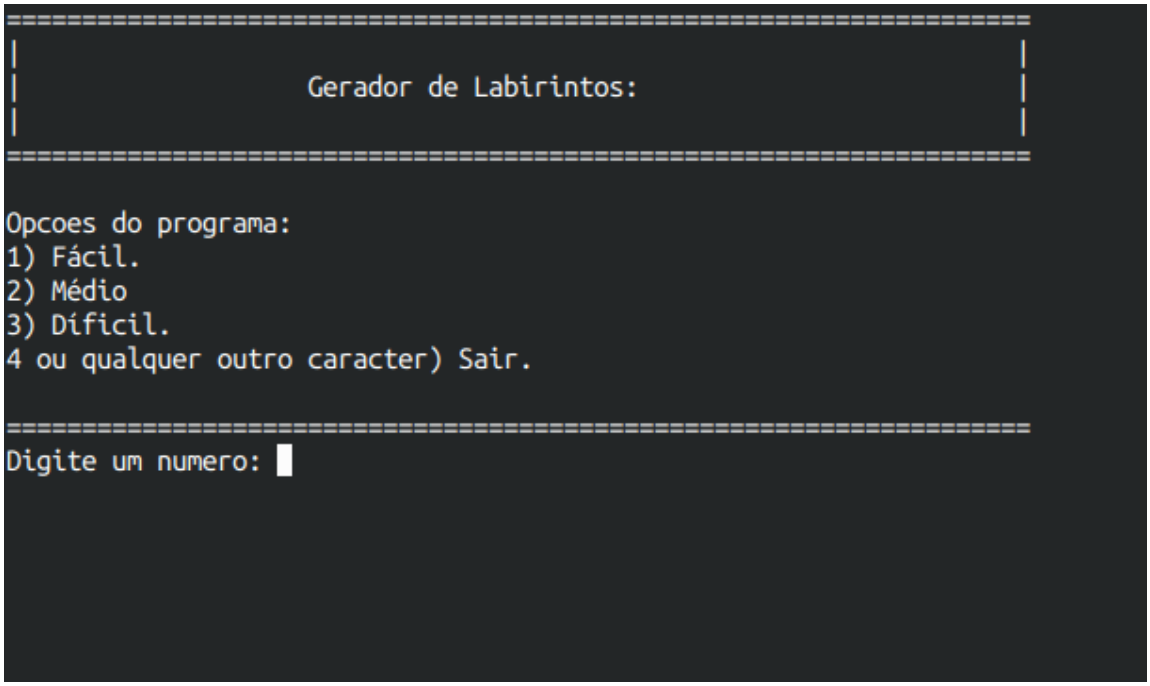
Resultado obtido através do medio.txt e todas as movimentações feitas pelo estudante e o número de chamadas recursivas executadas pelo programa.

Figura 11



Fonte: Terminal Linux

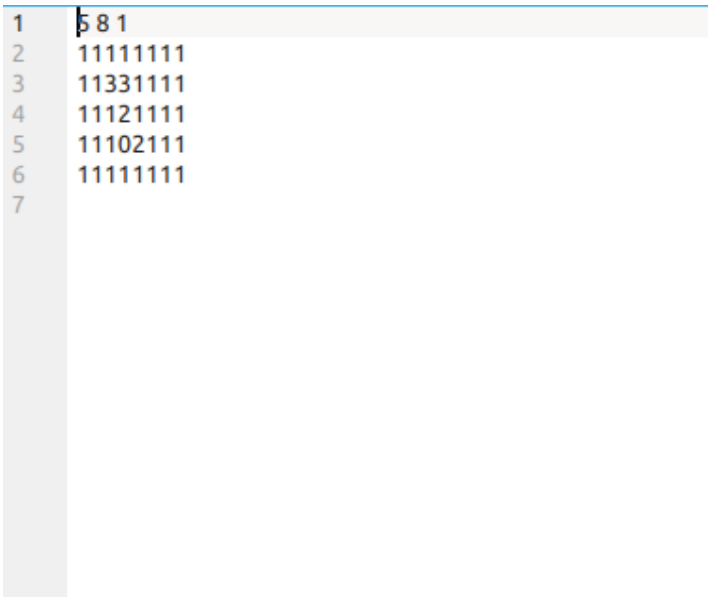
Figura 12



Fonte: Terminal Linux

Nivel Fácil.

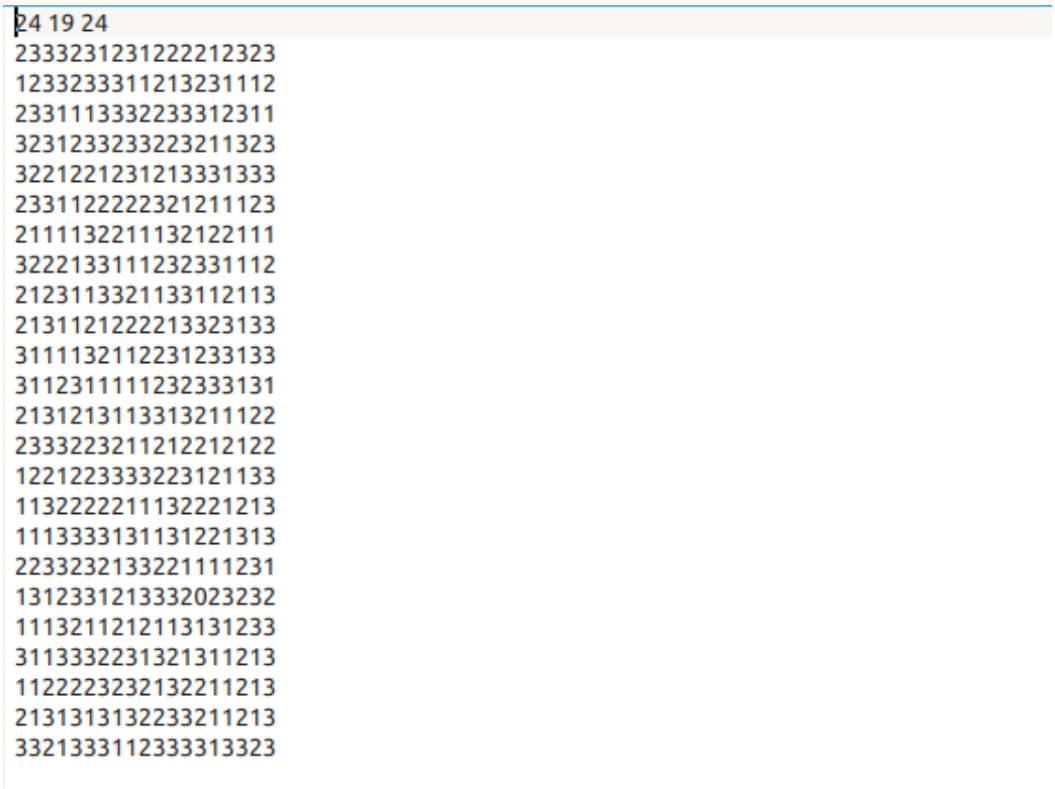
Figura 12



Fonte: Kate

Nivel Médio.

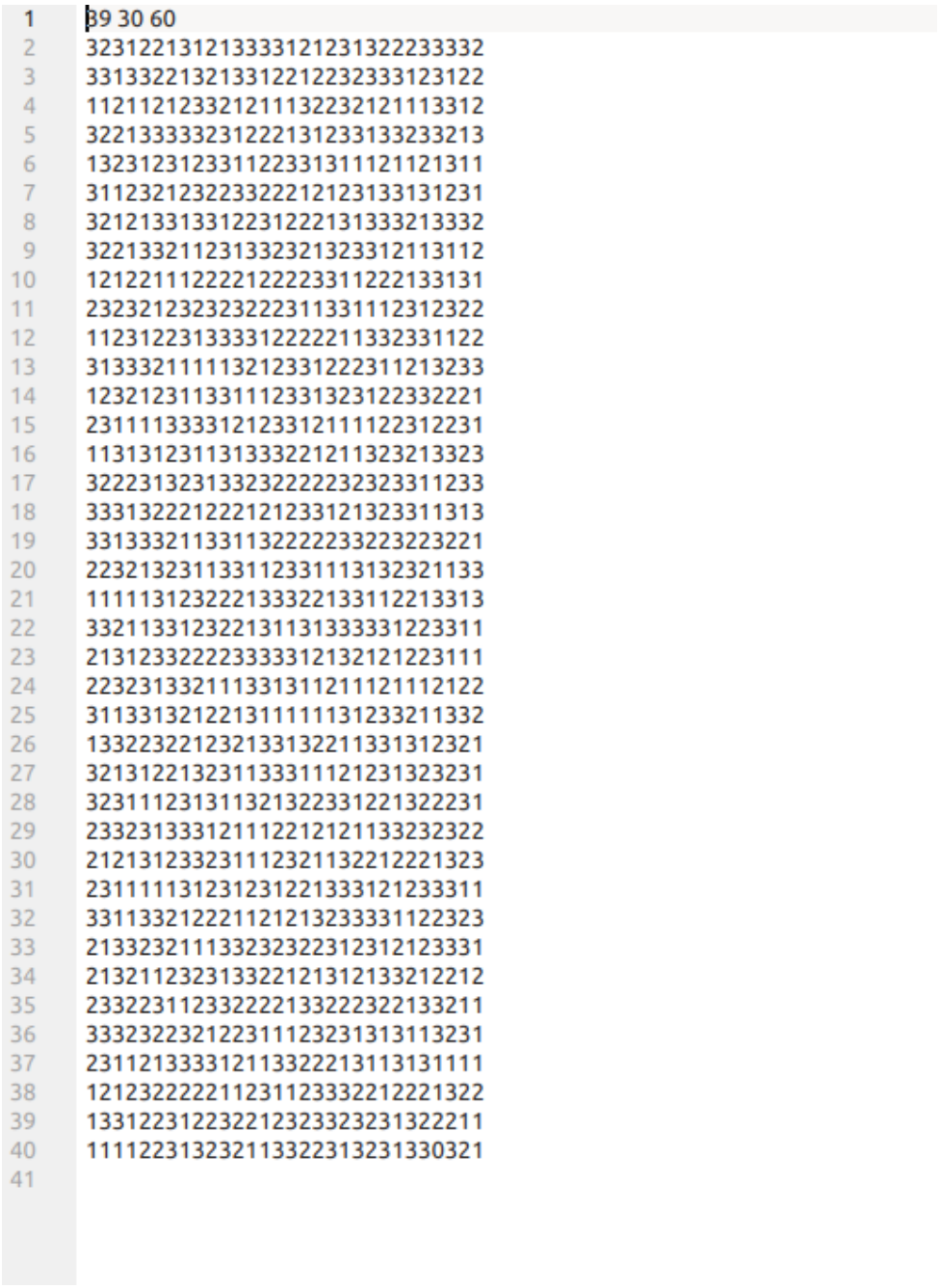
Figura 13



Fonte: Kate

Nivel Díficil.

Figura 13



Fonte: Kate

Dificuldades

O trabalho em si não era muito grande, porém era um pouco complexo, tivemos uma dificuldade inicial de fazermos a leitura de arquivo da forma que estava especificado nas diretrizes do trabalho, onde foi exigido que as células da matriz estivessem juntas no arquivo de leitura, porém após consultarmos alguns colegas, conseguimos encontrar a forma corretar de fazer essa leitura. Também, uma grande dificuldade foi na implementação do backtracking, onde foi necessário inúmeros testes e tempo para entender como exatamente a função deveria se comportar. Além disso, apenas erros de lógica e sintaxe da linguagem C.

Conclusão

Sem dúvidas, esse trabalho foi de grande importância para o nosso aprendizado, pois pudemos aplicar o que vemos em sala de aula em um problema real(ou quase real). Além disso, o trabalho foi de suma importância para que pudéssemos aprender a organizar o tempo melhor e dividirmos as tarefas de forma que ninguém ficasse sobrecarregado. Também foi possível os conceitos do backtracking. Então, após longos testes de execução e verificações no código fonte, o programa se encontra executando da maneira desejada. E por último, mas não menos importante, é notável a suma importância com relação a aprendizagem e aperfeiçoamento de conceitos em programação, não ficando limitado apenas a *sintax* da linguagem, mas abrangendo também ao pensamento lógico que possibilita a correta programação.

Agradecimentos ao professor Daniel Mendes pela oportunidade de realização do trabalho e aos colegas de turma por tirarem as nossas dúvidas.

Todo o desenvolvimento e distribuição do trabalho encontra-se hospedado na seguinte página do GitHub.

Referencias

MOREIRA, Jarlisson. Gerando números aleatórios em C: rand, srand e seed, 2013. Disponível em: https://www.cprogressivo.net/2013/03/Como-gerar-numeros-aleatorios-em-C-com-a-rand-srand-e- seed.html. Acesso em: 18 de Outubro de 2019.

Azeredo. Alocação Dinamica de Vetores e Matrizes, Sem data. Disponível em: http://mtm.ufsc.br/~azeredo/cursoC/aulas/ca70.html. Acesso em 18 de Outubro de 2019.

Wikipedia, ASCII, 2019. Disponíevel em: https://pt.wikipedia.org/wiki/ASCII. Acesso em 18 de Outubro de 2019