

Universidade Federal De Viçosa (UFV) - Campus Florestal
Disciplina: Projeto e Análise de Algoritmos
Professor(a): Daniel Mendes Barbosa (CCF330)

Trabalho prático 1

Backtracking on Fibonacci sequence

Autores:

Miguel Antônio Ribeiro e Silva - 4680

Mateus Henrique Vieira Figueiredo - 4707

Alan Gabriel Martins Silva - 4663

Sumário

1 - Introdução	3
2 - Metodologia	4
3 - Desenvolvimento	5
3.1 - fibonacci.c e fibonacci.h	5
3.2 - file.c e file.h	6
3.3 - matrix.c e matrix.h	9
3.4 - main.c	16
4 - Resultados	18
5 - Conclusão	21
6 - Referências	21

1 - Introdução

O trabalho prático tem como principal problema a elaboração de um programa que calcule a melhor rota, a fim de otimizar a colheita de batatas em uma fazenda fictícia.

Os engenheiros agrônomos, contratados por Fernando, dono da fazenda, decidiram que a rota mais produtiva seria a que seguisse uma sequência, ou seja, numera-se primeiro os campos de batata que só serão colhidos se estiverem em uma rota que siga essa sequência.

A sequência, que é composta por elementos da sucessão de Fibonacci, [1] é exemplificada a seguir:

1

1 1

1 1 2

1 1 2 3

1 1 2 3 5

É sabido que exista pelo menos uma ou nenhuma rota ótima.

Para resolver o problema foi necessário projetar um algoritmo com backtracking para encontrar o caminho ótimo.

2 - Metodologia

Após a formação do grupo, destacamos as principais tarefas a serem cumpridas para a realização do trabalho prático. Como:

- Implementação da entrada por arquivos de texto.
- Desenvolvimento de funções simples.
- Pesquisas e estudos sobre as características de um algoritmo com backtracking.
- Implementação da sequência de Fibonacci.
- Criação das funções necessárias para o funcionamento do backtracking.
- Interatividade com o usuário.
- Funções extras e testes.
- Makefile e documentação.

O código fonte foi desenvolvido em **C**[2] e versionado no **GitHub**[3], visando que seria a melhor forma de compartilhamento do mesmo entre os integrantes do grupo. Para uma melhor organização e visualização do projeto, este foi dividido em subpastas.

- /src** - implementação dos arquivos **.c** e **.h**.
- /testes** - arquivos **.txt** usados para testes.

Para compilar e executar o projeto, é necessário ter [gcc](#) e [make](#) instalado em sua máquina.

Comandos:

-make

Caso não funcione, tente:

**- gcc -o main src/fibonacci.c src/file.c src/matrix.c src/main.c -lm
./main**

3 - Desenvolvimento

Diversas funções foram criadas, organizadas na pasta /src dentre diversos arquivos fontes e cabeçalho, todas estão devidamente comentadas e referenciadas.

3.1 - fibonacci.c e fibonacci.h

Nesses arquivos, as funções (**figura 01**) necessárias para a elaboração da sequência, explicitada anteriormente, foram implementadas.

includes: <stdio.h> , <stddef.h> , <time.h> , <stdlib.h> , <string.h> , "matrix.h", "fibonacci.h", "file.h".

```
int realFibonacciSequence(int n);  
int triangular(int n)  
int getNthTermFromFibonacci(int n);
```

Figura 01 - fibonacci.h.

- **int realFibonacciSequence(int n):**
 - **função:** calcular o enésimo termo da sequência do problema, dada pela sucessão de Fibonacci.
 - **parâmetros:**
 - **n:** posição do termo na sequência.
 - **retorno:** (int) termo na enésima posição.
 - **detalhamento:** essa é a função principal para calcular o enésimo termo da sequência do problema.

- ***int triangular(int n):***
 - **função:** função auxiliar da `int realFibonacciSequence(int n)`
 - **parâmetros:**
 - ***n*** - variável da função.
 - **retorno:** (int) cálculo da função abaixo.
 - **detalhamento:** ao analisarmos a sequência do problema, encontrou-se a seguinte fórmula: $n * (n + 1) / 2$.

- ***int getNthTermFromFibonacci(int n):***
 - **função:** calcular o enésimo termo da sucessão de Fibonacci.
 - **parâmetros:**
 - ***n***: posição do termo na sucessão.
 - **retorno:** (int) termo na enésima posição.
 - **detalhamento:** funciona de maneira iterativa, usando a biblioteca `math.h`.

3.2 - file.c e file.h

Funções (**figura 02**) necessárias para a leitura e criação de arquivos de texto, usados para testes.

```
int **readFileIntoMatrix(char *filename, int *rows, int *cols);
char *generateRandomFile(int *rows, int *cols);
void flush_in();
```

Figura 02 - file.h.

- ***int **readFileIntoMatrix(char *filename, int *rows, int *cols):***
 - ***função:*** ler um arquivo de texto, formatado de acordo com a especificação e inserir seus dados em uma matriz de inteiros.
 - ***parâmetros:***
 - ***filename:*** nome do arquivo.
 - ***rows:*** ponteiro para um inteiro que armazenará o número de linhas da matriz.
 - ***cols:*** ponteiro para um inteiro que armazenará o número de colunas da matriz.
 - ***retorno:*** (int) matriz de inteiros.
 - ***detalhamento:*** a função lê a primeira linha do arquivo passado como parâmetro e inicializa uma matriz rows x cols com 0s, alocada dinamicamente (stdlib.h). Após isso, a preenche com os dados restantes do arquivo.

- ***char *generateRandomFile(int *rows, int *cols):***
 - ***função:*** gerar um arquivo de texto para testes, com dados aleatórios.
 - ***parâmetros:***
 - ***rows:*** ponteiro para um inteiro que armazenará o número de linhas da matriz.
 - ***cols:*** ponteiro para um inteiro que armazenará o número de colunas da matriz.
 - ***retorno:*** (char) caminho do arquivo.
 - ***detalhamento:*** a função primeiramente define o número de linhas e colunas usando **rand** e **srand** (time.h); por padrão, a matriz contém no máximo **8x25** inteiros. Após isso, é gerado inteiros aleatórios, sendo eles, elementos da sucessão de Fibonacci e os insere no arquivo. O algoritmo prioriza números menores, utilizando probabilidade (**figura 03**).

```

for (int i = 0; i < *rows; i++)
{
    for (int j = 0; j < *cols; j++)
    {
        int probability = rand() % 100 + 1;

        if (probability <= 30)
        {
            fprintf(file, "%d ", getNthTermFromFibonacci(rand() % 2 + 1));
        }

        else if (probability <= 50)
        {
            fprintf(file, "%d ", getNthTermFromFibonacci(rand() % 5 + 1));
        }

        else if (probability <= 70)
        {
            fprintf(file, "%d ", getNthTermFromFibonacci(rand() % 6 + 1));
        }

        else
        {
            fprintf(file, "%d ", getNthTermFromFibonacci(rand() % 8 + 1));
        }
    }
}

```

Figura 03 - Elementos menores da série, são priorizados.

```

6 19
2 1 3 1 2 1 5 1 1 1 1 1 21 1 1 3 1 1
1 1 1 3 3 1 1 1 1 1 1 1 1 1 1 1 1 2
1 1 2 1 1 1 1 8 21 1 1 2 8 3 5 1 1 13 2
1 1 1 21 1 1 1 5 1 1 3 13 1 2 2 3 8 5 1
1 5 1 1 5 1 1 13 8 1 1 2 5 1 1 1 1 1 1
1 1 2 2 3 1 21 1 1 5 1 1 5 1 1 1 1 1 2

```

Figura 04 - Exemplo de um arquivo gerado randomicamente.

- ***void flush_in():***

- ***função:*** limpeza do buffer de entrada do teclado.
- ***parâmetros:*** não possui.
- ***retorno:*** não possui.
- ***detalhamento:*** ver [4].

3.3 - matrix.c e matrix.h

As funções (figura 05) mais importantes do programa estão nestes arquivos.

includes: <stdio.h> , <stddef.h> , <stdlib.h> , <stdbool.h> , "matrix.h".

```
int **initializeMatrix(int rows, int cols);

bool isThereAPath(int **matrix, int **flagMatrix, int rows, int cols, int *totalRec, int *maxRec);

void printFlagMatrix(int **matrix, int rows, int cols);

void printMatrix(int **matrix, int rows, int cols);

void printPath(int **flagMatrix, int rows, int cols);
```

Figura 05 - matrix.h.

- ***int **initializeMatrix(int rows, int cols):***
 - **função:** inicializar uma matriz de inteiros com um determinado número de linhas e colunas.
 - **parâmetros:**
 - **rows:** número de linhas da matriz.
 - **cols:** número de colunas da matriz.
 - **retorno:** (int) matriz de inteiros
 - **detalhamento:** aloca uma matriz dinamicamente e a preenche com 0s.

- ***bool move(int currLine, int currCol, int rows, int cols, int **matrix, int **flag, int n, int *currRec):***

- ***função:*** encontrar um possível caminho ótimo para o problema.
- ***parâmetros:***
 - ***currLine:*** linha atual na matriz.
 - ***currCol:*** coluna atual na matriz.
 - ***rows:*** número de linhas da matriz.
 - ***cols:*** número de colunas da matriz
 - ***matrix*** ponteiro para a matriz.
 - ***flag:*** ponteiro para a matriz de flags.
 - ***n:*** posição do termo na sequência da sucessão de Fibonacci.
 - ***currRec:*** nível atual de recursão.
- ***retorno:*** (bool) se há um caminho para sequência ou não.
- ***detalhamento:*** é a função (**figura 06**) “cérebro” do programa, ela utiliza backtracking para calcular o possível melhor caminho para a colheita. Antes de tudo, é notório que ela sabe a posição onde deve começar (currLine e currCol) e ambas deverão ser passadas como parâmetro. Após isso, uma variável local **found** (**achou caminho**) é definida como false e o algoritmo inicia. Procura-se a princípio, um caminho para baixo, depois pra esquerda e direita e por último, para cima. Quando encontra um caminho possível, baseado na sequência definida, a posição é marcada como **visitada** em uma **matriz de flags** auxiliar e a função é recursivamente chamada, retornando **true**. Se partindo de um determinado ponto não é encontrado um caminho, a **posição atual é desmarcada na matriz de flags** e retorna **false**, voltando a recursão. Por fim, se a recursão atinge a última linha da matriz, é porque uma rota ótima foi encontrada, encerrando a recursão, se não, obviamente, não foi encontrado um caminho ótimo. Há também uma variável **currRec**, que armazena o nível atual da recursão.

```

bool move(int currLine, int currCol, int rows, int cols, int **matrix, int **flag, int n)
{
    if (currLine == rows - 1) { return true; }
    else
    {
        bool found = false;

        if (matrix[currLine + 1][currCol] == realFibonacciSequence(n + 1) &&
            flag[currLine + 1][currCol] == 0)
        {
            flag[currLine + 1][currCol] = n + 1;
            found = move(currLine + 1, currCol, rows, cols, matrix, flag, n + 1);
        }
        if (currCol > 0 && matrix[currLine][currCol - 1] == realFibonacciSequence(n + 1) &&
            flag[currLine][currCol - 1] == 0 && !found)
        {
            flag[currLine][currCol - 1] = n + 1;
            found = move(currLine, currCol - 1, rows, cols, matrix, flag, n + 1);
        }
        if (currCol < cols - 1 && matrix[currLine][currCol + 1] == realFibonacciSequence(n + 1) &&
            flag[currLine][currCol + 1] == 0 && !found)
        {
            flag[currLine][currCol + 1] = n + 1;
            found = move(currLine, currCol + 1, rows, cols, matrix, flag, n + 1);
        }
        if (currLine > 0 && matrix[currLine - 1][currCol] == realFibonacciSequence(n + 1) &&
            flag[currLine - 1][currCol] == 0 && !found)
        {
            flag[currLine - 1][currCol] = n + 1;
            found = move(currLine - 1, currCol, rows, cols, matrix, flag, n + 1);
        }

        if (!found)
        {
            flag[currLine][currCol] = 0;
            return false;
        }

        return true;
    }
}

```

Figura 06 - Função move.

- ***isThereAPath(int **matrix, int **flagMatrix, int rows, int cols, int *totalRec, int *maxRec):***
 - ***função:*** Descobre se há um início possível na primeira linha.
 - ***parâmetros:***
 - ***matrix:*** ponteiro para a matriz.
 - ***flagMatrix:*** ponteiro para a matriz de flags.
 - ***rows:*** quantidade de linhas da matriz.
 - ***cols:*** quantidade de colunas da matriz.
 - ***totalRec:*** número total de recursões.
 - ***maxRec:*** máximo número de recursões.
 - ***retorno:*** (bool) se existe um caminho de início ou não.
 - ***detalhamento:*** a função percorre a primeira linha da matriz a fim de encontrar uma posição que sirva para o início do cálculo do caminho ótimo. Caso a encontre, a função **move** é chamada. Também contabiliza o número máximo e o total de recursões, após o cálculo do possível caminho.

- ***void printFlagMatrix(int **matrix, int rows, int cols):***
 - ***função:*** imprimir a matriz de flags.
 - ***parâmetros:***
 - ***matrix:*** ponteiro para a matriz de flags:
 - ***rows:*** número de linhas da matriz.
 - ***cols:*** número de colunas da matriz.
 - ***retorno:*** não possui.
 - ***detalhamento:*** uma **forma extra** de exibir a rota ótima para o usuário. A matriz de flags detalha com precisão o caminho ótimo, e com o uso de **cores** é impresso no terminal (**figura 6**) a sequência de flags encontrados.

Flag matrix:

00	00	00	00	01	00	05	06	00	00	00
00	00	00	00	02	03	04	07	00	00	00
00	00	00	00	11	10	09	08	00	00	00
00	00	00	00	12	13	14	15	16	17	18
00	00	27	26	25	24	23	22	21	20	19
00	00	28	29	00	00	00	00	00	00	00
00	00	00	30	31	32	33	34	00	00	00
00	00	00	00	00	00	00	35	00	00	00
00	00	00	00	00	00	00	36	00	00	00
00	00	00	00	00	00	00	37	00	00	00

Figura 6 - Matriz de flags da matriz exemplo do problema.
Em verde, a rota ótima.

- ***void printMatrix(int **matrix, int rows, int cols):***
 - ***função:*** imprimir uma matriz no terminal.
 - ***parâmetros:***
 - ***matrix:*** ponteiro para a matriz.
 - ***rows:*** número de linhas da matriz.
 - ***cols:*** número de colunas da matriz.
 - ***retorno:*** não possui.
 - ***detalhamento:*** usada para imprimir uma matriz, criada a partir de um arquivo texto.

- ***void printPath(int **flagMatrix, int rows, int cols):***
 - ***função:*** imprimir as coordenadas da rota ótima.
 - ***parâmetros:***
 - ***flagMatrix:*** ponteiro para a matriz de flags.
 - ***rows:*** número de linhas da matriz.
 - ***cols:*** número de colunas da matriz.
 - ***retorno:*** não possui.
 - ***detalhamento:*** como especificado, essa função (**figura 7**) imprime (**figura 8**) no terminal exatamente as coordenadas do caminho ótimo, utilizando a matriz de flags. A princípio, a função percorre a primeira linha da matriz e encontra o ponto de partida, após isso, verifica se possui um flag para baixo, depois cima e esquerda e por fim, para cima. A verificação ocorre com base na sequência dos números inteiros naturais.

```

do
{
    if (x < rows - 1 && flagMatrix[x + 1][y] == path + 1)
    {
        printf("[%d %d]\n", x + 1 + 1, y + 1);
        x++;
        path++;
    }
    else if (y > 0 && flagMatrix[x][y - 1] == path + 1)
    {
        printf("[%d %d]\n", x + 1, y);
        y--;
        path++;
    }
    else if (y < cols - 1 && flagMatrix[x][y + 1] == path + 1)
    {
        printf("[%d %d]\n", x + 1, y + 1 + 1);
        y++;
        path++;
    }
    else if (x > 0 && flagMatrix[x - 1][y] == path + 1)
    {
        printf("[%d %d]\n", x, y + 1);
        x--;
        path++;
    }
} while (x != rows - 1);

```

Figura 7 - Cálculos necessários para a impressão no terminal.

Path has been found!

Path:

```
[1 5]  
[2 5]  
[2 6]  
[2 7]  
[1 7]  
[1 8]  
[2 8]  
[3 8]  
[3 7]  
[3 6]  
[3 5]  
[4 5]  
[4 6]  
[4 7]  
[4 8]  
[4 9]  
[4 10]  
[4 11]  
[5 11]  
[5 10]  
[5 9]  
[5 8]  
[5 7]  
[5 6]  
[5 5]  
[5 4]  
[5 3]  
[6 3]  
[6 4]  
[7 4]  
[7 5]  
[7 6]  
[7 7]  
[7 8]  
[8 8]  
[9 8]  
[10 8]
```

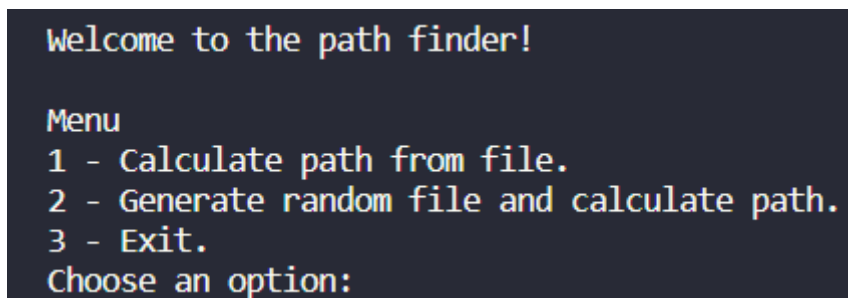
Figura 8 - Impressão das coordenadas da rota ótima da matriz exemplo do problema.

3.4 - main.c

Programa principal do projeto.

includes - `<stdio.h>` , `<stdlib.h>` , `<stdbool.h>` , `<limits.h>` , `"matrix.h"` , `"file.h"` , `"fibonacci.h"`.

Ao executar o programa, um menu (**figura 9**) será impresso na tela e o usuário deverá escolher uma das opções.



```
Welcome to the path finder!

Menu
1 - Calculate path from file.
2 - Generate random file and calculate path.
3 - Exit.
Choose an option:
```

Figura 9 - Menu do programa

1 - Calculate path from file.

Selecionando essa opção, o usuário deverá digitar o diretório do arquivo que deseja usar. Após a escolha, o arquivo será aberto, e as funções necessárias para o cálculo do caminho ótimo serão executadas. Ao fim, o resultado será impresso no terminal, juntamente com a análise da execução, caso o usuário prefira, selecionando a opção própria para o caso.

2 - Generate random file and calculate path.

Selecionando essa opção, um arquivo aleatório é gerado, salvo na pasta **/tests** como explicado anteriormente e executado em sequência. As funções para o cálculo do caminho ótimo serão chamadas. É impresso no terminal as propriedades do arquivo gerado, a matriz resultante e, por fim, o resultado, juntamente com a análise da execução, caso o usuário prefira, selecionando a opção própria para o caso.

3 - *Exit*.

Encerra o programa.

- ***void clearConsole():***

- ***função:*** limpa o terminal baseado no sistema operacional do usuário.
- ***parâmetros:*** não possui.
- ***retorno:*** não possui.
- ***detalhamento:*** se o usuário estiver usando o sistema operacional Windows, `system("cls")` é ativo, caso contrário, `system("clear")` é a opção.

4 - Resultados

O programa possui um modo análise, que é ativado por padrão.

Ao executá-lo, o número de recursões totais e o máximo nível recursivo alcançado é contabilizado e o usuário, por meio do menu, poderá decidir se deseja ver os resultados, ou não. Ao escolher a opção, é impresso no terminal as seguintes métricas:

-Total recursion

-Max recursion

A seguir, alguns testes foram feitos e registrados:

test1.txt
<ul style="list-style-type: none">• matriz: 4x5• rota ótima: Sim• número de recursões: 15• máximo nível de recursão: 13

Matriz pequena, com um número considerável de recursões.

test2.txt
<ul style="list-style-type: none">• matriz: 8x19• rota ótima: Sim• número de recursões: 73• máximo nível de recursão: 42

Percorre grande parte da matriz.

test3.txt
<ul style="list-style-type: none">• matriz: 10x11• rota ótima: Sim• número de recursões: 67• máximo nível de recursão: 68

Matriz exemplo do problema.

test4.txt

- **matriz:** 8x19
- **rota ótima:** Sim
- **número de recursões:** 52
- **máximo nível de recursão:** 52

Na primeira tentativa, achou um caminho ótimo, igualando os resultados.

teste5.txt

- **matriz:** 13x5
- **rota ótima:** Sim
- **número de recursões:** 82
- **máximo nível de recursão:** 82

Percorreu o número máximo de elementos na matriz.

random-1-23.txt

- **matriz:** 1x23
- **rota ótima:** Sim
- **número de recursões:** 1
- **máximo nível de recursão:** 1

Matriz contém apenas uma linha.

random-7-2.txt

- **matriz:** 7x2
- **rota ótima:** Não
- **número de recursões:** 0
- **máximo nível de recursão:** 0

Sequer achou um ponto de partida para o caminho.

random-8-24.txt

- **matriz:** 8x24
- **rota ótima:** Não
- **número de recursões:** 89
- **máximo nível de recursão:** 18

Percorreu bastante, porém não achou um caminho ideal.

random-6-3.txt
<ul style="list-style-type: none">• matriz: 6x3• rota ótima: Não• número de recursões: 1• máximo nível de recursão: 1

Quanto maior o número de linhas, comparado ao de colunas, mais difícil é achar uma rota ótima.

random-100-100.txt
<ul style="list-style-type: none">• matriz: 100x100• rota ótima: Não• número de recursões: 977• máximo nível de recursão: 145

Encontrar um caminho ótimo em uma matriz assim, dadas condições de probabilidade implementadas, é impossível.

Observações:

- Foram testados centenas de arquivos, alguns deles estão na pasta **/tests** e podem ser executados.
- Os números da matriz randomizada são pré definidos (ver função char ***generaterandomFile(int *rows, int *cols)**), pelo motivo de tentar otimizar sua criação, a fim de encontrar matrizes ótimas.
- Foi testado para N muito grandes, mas não achei necessário expor o resultado por motivos óbvios.

5 - Conclusão

Após o fim do trabalho prático podemos apontar certas dificuldades e facilidades encaradas pelo grupo durante a implementação desse projeto.

Notamos uma certa facilidade na hora de entender como o problema deveria ser resolvido e na implementação de funções básicas em relação a sequência de Fibonacci.

Tivemos dificuldades de implementar o backtracking, mas com algumas pesquisas e estudos [5][6][7] obtivemos sucesso.

Os resultados obtidos, foram úteis na compreensão de como um algoritmo recursivo funciona explicitando suas peculiaridades e possibilidades.

Por fim é visível o proveito e as lições aprendidas durante esse período de desenvolvimento do trabalho prático.

6 - Referências

- [1] [Sequência de Fibonacci - Toda Matéria](#)
- [2] [C \(programming language\) - Wikipedia](#)
- [3] <https://github.com/Mateus-Henr/Backtracking-On-Fibonacci-Sequence>
- [4] [Limpeza do buffer do teclado após scanf - Stack Overflow em Português](#)
- [5] Ziviani N. , Projeto de Algoritmos com implementações em Pascal e C
- [6] [Backtracking Algorithms - GeeksforGeeks](#)
- [7] <https://pt.wikipedia.org/wiki/Backtracking>
[Backtracking – Wikipédia, a enciclopédia livre](#)