UNIVERSIDADE FEDERAL DE VIÇOSA - CAMPUS FLORESTAL

Trabalho Prático 2 de Projeto e Análise de Algoritmos

Nome: Aline Cristina Santos Silva
Gustavo Luca Ribeiro Da Silva
Luana Tavares Anselmo

Matrícula: 5791,5787,5364

Florestal - MG
2024

UFV

Sumário

UFV

1. Introdução

O projeto desenvolvido é um programa em C que simula a navegação de um estudante em uma caverna cheia de desafios, com o objetivo de alcançar a saída enquanto maximiza seus Pontos de Vida (PV). A caverna é representada como uma matriz de células, onde o estudante inicia sua jornada em uma posição inicial e tenta chegar a um ponto final designado. Durante o percurso, ele pode encontrar células que aumentam ou diminuem seus PV, enfrentando barreiras e monstros que tornam o trajeto desafiador.

Este programa foi desenvolvido com o propósito de aplicar e consolidar conceitos fundamentais de **programação dinâmica**, estruturas de dados e algoritmos, enfatizando a resolução de problemas por meio de subproblemas otimizados. Além disso, o código utiliza técnicas de manipulação de arquivos para leitura da configuração da caverna e gravação do caminho percorrido, assim como estruturas eficientes para calcular o trajeto.

O projeto conta com as seguintes funcionalidades:

Carregar Caverna de Arquivo: Permite ao usuário carregar a configuração da caverna a partir de um arquivo. Este arquivo contém a matriz da caverna, os PV iniciais e a posição inicial e final do estudante.

Determinar o Caminho Ótimo: Utiliza um algoritmo baseado em programação dinâmica para encontrar o melhor caminho que leva o estudante do ponto inicial até o ponto final, garantindo que ele mantenha o máximo possível de PV. Caso não seja possível escapar com vida, o programa informa que o problema não tem solução.

Gerar Arquivo de Saída: Registra em um arquivo de texto o caminho percorrido pelo estudante, detalhando as coordenadas de cada movimento até o ponto final, caso a solução seja viável.

2. Compilação e Organização

O projeto está organizado em dois diretórios principais:

- **headers/**: contém todos os arquivos de cabeçalho (.h) utilizados no projeto. Esses arquivos definem as estruturas de dados e declarações de funções usadas em vários pontos do código. Exemplo: caverna.h, dp.h
- **sources/**: contém todos os arquivos de implementação (.c) do projeto, que possuem o código das funções definidas nos arquivos de cabeçalho. Exemplo: caverna.c, dp.c

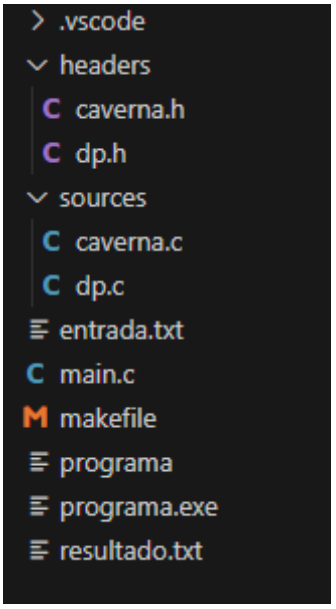

Figura 1: Organização das pastas.

O Makefile é usado para automatizar o processo de compilação, simplificando a criação do executável. Ele define como os arquivos-fonte devem ser compilados e onde o executável final será salvo. Abaixo está um exemplo de Makefile para o projeto:

```
makefile
1   all: main.c sources/caverna.c sources/dp.c
2       gcc -o programa main.c sources/caverna.c sources/dp.c -lm
```

Figura 2: MakeFile.

Como Compilar o Projeto?

- Pré-requisitos: Certifique-se de que o gcc está instalado em seu sistema.
- Compilação: Para compilar o projeto, navegue até o diretório raiz do projeto e execute o comando: **make**
- Em seguida: **./programa** entrada.txt (Ou o nome que deu para sua entrada).

3. Desenvolvimento

O projeto consiste em criar um programa que leia uma representação de uma caverna de um arquivo de entrada, identifique um caminho ótimo de um ponto inicial até um ponto final, e gere um arquivo de saída que contenha o caminho encontrado ou indique que não é possível encontrar um caminho.

3.1 Estrutura Caverna

A estrutura Caverna é responsável por modelar o ambiente no qual o problema será resolvido. Ao armazenar as dimensões, os pontos iniciais e a matriz da caverna, ela organiza as informações necessárias para a execução do algoritmo de busca, permitindo que o programa encontre o caminho mais eficiente do ponto inicial ao ponto final, respeitando as restrições impostas.

3.2 Função lerArquivo

A função lerArquivo é responsável por carregar as informações da caverna a partir de um arquivo de entrada, estruturando os dados na matriz da caverna e inicializando os parâmetros principais.

3.3 Função gerarArquivoResultado

A função gerarArquivoResultado transforma os resultados calculados pelo programa em um formato compreensível e padronizado, gravando-os no arquivo resultado.txt. Se o tamanho do caminho encontrado for zero, a função escreve "impossível" no arquivo, indicando que não há solução viável. Caso contrário, ela percorre a matriz de coordenadas do caminho e registra cada movimento do jogador em ordem, linha por linha.

```c
void gerarArquivoResultado(int caminho[][2], int tamanho) {
    FILE* resultado = fopen("resultado.txt", "w");

    if (tamanho == 0) {
        fprintf(resultado, "impossível\n");
    } else {
        for (int i = 0; i < tamanho; i++) {
            fprintf(resultado, "%d %d\n", caminho[i][0], caminho[i][1]);
        }
    }

    fclose(resultado);
}
```

Figura 3: Função gerarArquivoResultado.

3.4 Função encontrarMelhorCaminho

A função encontrarMelhorCaminho implementa a lógica principal do programa, utilizando Programação Dinâmica para calcular o caminho que maximiza os pontos de vida restantes ao sair da caverna. Ela inicializa uma matriz DP para rastrear os pontos acumulados em cada célula, propaga valores por meio de uma busca em largura e reconstrói o caminho do ponto final ao inicial usando os ponteiros armazenados. A função garante eficiência e clareza, mesmo em cenários complexos, sendo fundamental para alcançar a solução ótima do problema.

1. **Uso da Programação Dinâmica (DP)**: A função constrói uma matriz DP para armazenar os pontos de vida acumulados em cada célula da caverna.
2. **Busca em Largura (BFS)**: A fila é utilizada para explorar a matriz de forma eficiente, propagando os pontos de vida acumulados para as células vizinhas.
3. **Propagação Condicional**: Apenas células acessíveis (não marcadas como obstáculos) são atualizadas, e o valor é propagado se resultar em um maior número de pontos de vida do que o previamente registrado.
4. **Reconstrução do Caminho**: A função rastreia o melhor caminho do ponto final até o inicial usando os ponteiros armazenados na matriz DP, garantindo que o percurso ótimo seja extraído.

```c
int encontrarMelhorCaminho(Caverna* caverna, int caminho[][2]) {
    CelulaDP dp[MAX_LINHAS][MAX_COLUNAS];
    int inicio_x = -1, inicio_y = -1, fim_x = -1, fim_y = -1;

    // Inicializar a matriz DP e encontrar pontos de início ('I') e fim ('F')
    for (int i = 0; i < caverna->linhas; i++) {
        for (int j = 0; j < caverna->colunas; j++) {...
    }

    // Verificar se pontos inicial e final foram encontrados
    if (inicio_x == -1 || inicio_y == -1 || fim_x == -1 || fim_y == -1) {...

    // Inicializar a posição inicial na matriz DP
    dp[inicio_x][inicio_y].pontos_vida = caverna->pontos_iniciais;

    // Usar uma fila para processar as células
    typedef struct {...
    } Posicao;

    Posicao fila[MAX_CAMINHO];
    int frente = 0, tras = 0;

    fila[tras++] = (Posicao){inicio_x, inicio_y};

    // Movimentos possíveis (cima e esquerda apenas)
    int dx[] = {-1, 0};
    int dy[] = {0, -1};

    while (frente < tras) {...

    // Verificar se o ponto final foi alcançado
    if (dp[fim_x][fim_y].pontos_vida == INT_MIN) {...

    // Reconstruir o caminho do ponto final para o inicial
    int caminho_length = 0;
    int atual_x = fim_x, atual_y = fim_y;

    while (atual_x != -1 && atual_y != -1) {...

    // Inverter o caminho para começar do início
    for (int i = 0; i < caminho_length / 2; i++) {
        int temp_x = caminho[i][0];
        int temp_y = caminho[i][1];
        caminho[i][0] = caminho[caminho_length - 1 - i][0];
        caminho[i][1] = caminho[caminho_length - 1 - i][1];
        caminho[caminho_length - 1 - i][0] = temp_x;
        caminho[caminho_length - 1 - i][1] = temp_y;
    }

    printf("Melhor caminho encontrado com %d pontos de vida.\n", dp[fim_x][fim_y].pontos_vida);
    return caminho_length;
}
```

Figura 3: Função encontrarMelhorCaminho.

4. Resultados

entrada.txt

```
4 5 40
-20 0 -20 0 0
F -10 -10 -10 -10
-20 0 0 0 +20
0 0 -10 0 I
```

Figura 4: Arquivo de entrada utilizado.

```
./programa entrada.txt
```

Figura 5: Iniciando o programa.

```
PS C:\Users\guluc\OneDrive\Gustavo\UFV - Trabalhos\PAA\tp2> ./programa entrada.txt
Melhor caminho encontrado com 50 pontos de vida.
```

Figura 6: Retorno no terminal.

resultado.txt

```
3 4
2 4
2 3
2 2
2 1
1 1
1 0

```

Figura 7: Arquivo resultado gerado.

5. Conclusão

Esse programa conseguiu resolver o desafio de encontrar o melhor caminho na caverna, maximizando os pontos de vida do jogador, usando Programação Dinâmica. Apesar dos desafios, como configurar a matriz DP, lidar com restrições e reconstruir o caminho até o ponto final, tudo foi superado com uma lógica bem planejada e organização no código.

A estrutura modular e bem separada das funções ajudou a deixar o programa mais claro e fácil de entender, além de facilitar possíveis melhorias no futuro. A documentação explicou direitinho cada etapa, desde como o programa lê os dados de entrada até como ele gera a saída com o caminho encontrado.

No final, o trabalho foi uma ótima oportunidade para colocar em prática conceitos de algoritmos e mostrar como a programação pode ser aplicada para resolver problemas de forma eficiente e organizada.

UFV

6. Referências

Universidade de São Paulo, Instituto de Matemática e Estatística. Análise de Algoritmos - Programação Dinâmica. Disponível em: https://www.ime.usp.br/~pf/analise_de_algoritmos/aulas/dynamic-programming.html.

Universidade de São Paulo, Instituto de Matemática e Estatística. Panda - Resolução de Problemas Usando Python. Seção: Programação Dinâmica. Disponível em: https://panda.ime.usp.br/panda/static/pythonds_pt/04-Recursao/11-programacaoDinamica.html.

Luciano Digiampietri. **Aula 15a - Programação Dinâmica**. YouTube. Disponível em:https://www.youtube.com/watch?v=jBBpBeI5zQw.