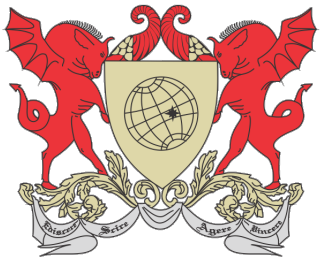

UNIVERSIDADE FEDERAL DE VIÇOSA
CAMPUS FLORESTAL

Trabalho Prático II
Projeto e Análise de Algoritmos

Luciano Belo - 3897
Mariana Souza - 3898
Guilherme Correa - 3509

Trabalho Prático apresentado à disciplina de Projeto e Análise de Algoritmos do curso de Ciência da Computação da Universidade Federal de Viçosa.

Florestal
Março de 2022

CCF 330 - Projeto e Análise de Algoritmos

TP 02 - Samus Aran

Luciano Belo - 3897
Mariana Souza - 3898
Guilherme Correa - 3509

09 de Março de 2022

Contents

1 Introdução **2**

2 Desenvolvimento **2**
2.1 Entrada 3
2.2 Tipos Abstratos de Dados 4
2.3 Programação Dinâmica 5

3 Funcionalidades Extras **6**
3.1 Tempo de Execução 7
3.2 Melhor caminho 7
3.3 Gerar arquivos de entrada 8

4 Execução **8**

5 Conclusão **10**

1 Introdução

O presente trabalho prático desenvolvido para a disciplina de Projeto e Análise de Algoritmos tem como objetivo ajudar a exploradora espacial que está com problemas no sistema de IA de sua nave, sendo assim foi desenvolvido um algoritmo usando programação dinâmica para que Samus consiga sair da caverna com segurança em um menor tempo possível, dado que umas áreas da caverna são bloquedas e algumas contém monstros que Samus precisa derrotar para sair.

Para um melhor desenvolvimento do grupo foi utilizado o GitHub para realizar o versionamento do código e a colaboração dos integrantes do grupo, a IDE Visual Studio Code para implementação do algoritmo em C e o Notion para uma melhor visualização as funcionalidades a serem implementadas.

2 Desenvolvimento

Dado a implemnetação do trabalho prático utilizando porgramação dinâmica foi implementado um algoritmo capaz de escolher um caminho mais rápido para Samus sair da caverna em um menor tempo e em segurança, sendo assim é apresentado o mapa abaixo onde se encontra os caminhos possíveis para que Samus saia, onde algumas áreas representadas em marrom estão bloqueadas por rochas e outras em amarelo são as partes que Samus pode passar e as que tem algum número indicam a presença de monstros que Samus precisa derrotar, na Figura 1 é apresentado o mapa.

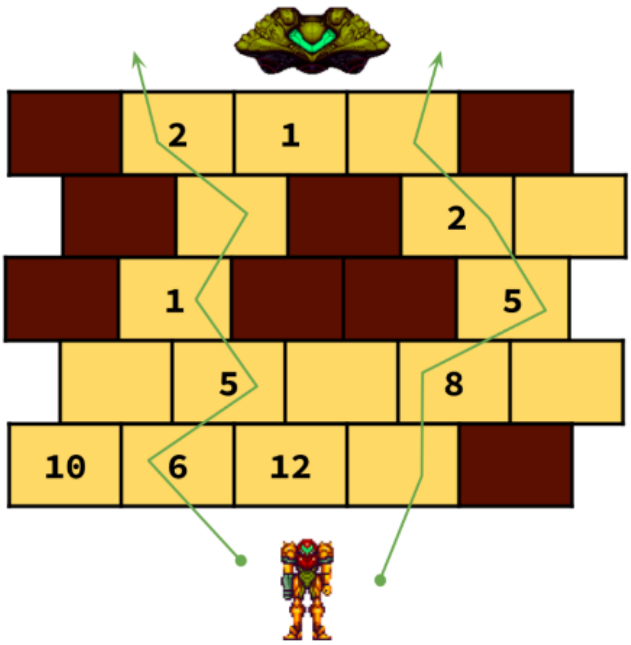

Figure 1: Mapa do caminho para Samus.

O algoritmo foi desenvolvido na linguagem C e está divido em pastas onde a pasta "src" é encontrado os arquivos .c da implemnetação e a pasta "headres" os arquivos .h, onde o arquivo para apresentar a interface com o usuário estão no "menu.c", contendo a interface para ler o arquivo de entrada desejado, a pasta "data" contém os arquivos de entrada com diferentes valores, também contém a pasta generate que arquivos com objetivo de gerar diferentes níveis de dificuldade para os mapas. Dessa forma, o trabalho segue a seguinte estrutura de pastas:

```
TP02-PAA
├── doc
├── data
├── generator
│   ├── generate
│   ├── generate.c
│   └── generate.h
├── headers
│   ├── cave.h
│   ├── menu.h
│   ├── position.h
│   ├── samus.h
│   └── subproblems.h
├── dist
│   └── main
└── src
    ├── main.c
    ├── cave.c
    ├── menu.c
    ├── position.c
    ├── samus.c
    └── subproblems.c
```

2.1 Entrada

Os arquivos de entrada utilizados possuem a seguinte configuração, onde o espaço da caverna é fornecido a partir do arquivo de texto de entrada com um formato padronizado, contendo na primeira linha a altura da caverna, largura de cada nível que tem na caverna, o tempo que Samus vai gastar para realizar cada movimentação e o tempo que a lava gasta para subir em cada nível da caverna. Nas linhas seguintes são informados o conteúdo que cada célula terá no mapa da caverna, onde cada célula contém três caracteres, dado que, se for um espaço vazio serão encontrados três zeros, se for uma célula que contém rochas será entrado três hashtags e se for um espaço com algum monstro no caminho terá o número entre 001 e 999.

Como apresentado na Figura 2 a caverna possui 3 níveis de altura e cada nível conta com 4 células de largura, onde Samus gasta 4 unidade de tempo para realizar cada movimento e a lava demora 7 unidades de tempo para subir cada nível. Dessa forma, foram criados nove diferentes arquivos de entrada para realizar os testes, onde todos estão disponíveis na pasta "data".

Figure 2: Arquivo de entrada.

2.2 Tipos Abstratos de Dados

Para implementação do algoritmo foram usados TADs nos arquivos "cave.h", "subproblems.h" e "samus.h". Dessa forma, o arquivo cave.h tem como objetivo gerar o espaço geográfico da caverna, então o TAD "Cave" apresentado no trecho de código abaixo tem os tamanhos para a altura e largura da caverna como também contém o tempo que a lava gasta para subir cada nível da caverna, sendo utilizado no "cave.c" para incializar a caverna com a determinada quantiadde de altura, largura e tempo da lava passados no arquivo de entrada, inicalizar a tabela de soluções com altura e largura da mesma, o possível caminho de soluções dado a largura.

```
typedef struct Cave
{
    int height, width, lavaTime;
    int **cave;
} Cave;
```

Dado o arquivo "subproblems.h" foram implementados os TADs "Subproblem" e "SubproblemsTable" como está sendo apresentado no trecho de código a seguir, onde no "Subproblem" é inserido os dados para a tabela que sera usada no TAD "SubproblemsTable" abaixo que contém as dimensões da tabela utilizada para realizar a programação dinâmica e encontrar o melhor caminho possível para Samus.

```
typedef struct Subproblem
{
    int line, column, prevLine, prevColumn, value;
} Subproblem;

typedef struct SubproblemsTable
{
    Subproblem **table;
    int width, height;
} SubproblemsTable;
```

No arquivo "samus.h" foi implementado o TAD "Samus" que contém o tempo que Samus realiza a cada movimento que é realizado, dessa forma foi criado o TAD para inicalizar a estrutura de dados Samus e imprimir o mesmo apresentando a quantidade total de tempo gasto a cada movimentação. Onde, dado que Samus está na linha mais baixa ela sempre deverá se mover para cima, cada movimento que ela realiza é gasto a unidade de tempo passada na primeira linha do arquivo de entrada e cada vez que encontra com um monstro é gasto um tempo para enfrentar o mesmo, então é o tempo gasto na movimentação mais o tempo para enfrentar o monstro representado pelo número que está na célula a partir do arquivo de entrada passado.

```
typedef struct Samus
{
    int time;
} Samus;
```

2.3 Programação Dinâmica

A programação dinâmica calcula a solução para todos os subproblemas, partindo dos subproblemas menores para os maiores, armazenando os resultados em uma tabela. Essa técnica é melhor empregada em uma grande classe de problemas conhecida como problemas de otimização, ou seja, temos um tipo de construção combinatória que satisfaz um conjunto de regras dadas e queremos extrair algum tipo de informação sobre esse conjunto, como por exemplo quantos conjuntos totais existem que satisfazem as regras dadas, qual o melhor/pior conjunto de acordo com uma função de custo também dada (essa função pode ser por exemplo o tamanho do conjunto ou a soma dos seus elementos), além de existir também casos em que só queremos saber se existe alguma construção dentro desse conjunto que satisfaça alguma outra propriedade especial.

A vantagem do método está no fato de que uma vez que um subproblema é resolvido, a resposta é armazenada em uma tabela e o subproblema nunca mais é recalculado. Quando o somatório dos tamanhos dos subproblemas é *O(n)* é provável que o algoritmo recursivo tenha complexidade polinominal. Entretanto, se a divisão de um problema resulta em n subproblemas de tamanho n - *1*, é provável que o algoritmo recursivo tenha complexidade exponencial.

O algoritmo desenvolvido conta com três funções principais, sendo elas *solver, move e lessInitialValue*. Assim como explicado anteriormente, a ideia é armazenar os resultados em uma tabela, sendo assim, a função *solver* encontra o melhor caminho a partir da primeira linha, utilizando uma abordagem **bottom-up**[1] já que a solução ótima começa a ser calculada a partir do subproblema mais trivial, no nosso caso, qual o menor caminho começando de onde será a sáida da caverna.

Para exemplificar melhor o funcionamento do algoritmo usaremos o mapa a seguir:

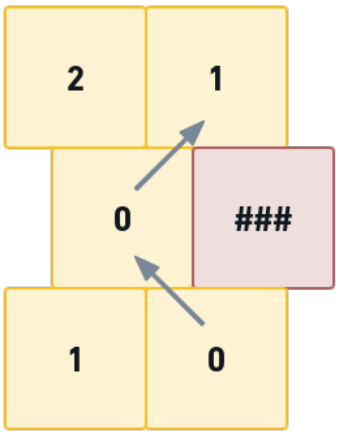

Figure 3: Caso de teste

A função *solver* primeiramente verifica se a posição atual é válida, em seguida caso seja, continuando a ideia de alinhamento das linhas da caverna, verifica-se se a linha atual é par ou ímpar, isso porque a próximo caminho a ser seguido seja para esquerda ou direita depende do alinhamento. Com os passos anteriores já definidos a ideia agora é definirmos qual será o próximo caminho a ser seguido. Caso a linha atual seja a primeira, os valores que serão preenchidos na tabela serão os respectivos no mapa, além disso não teremos linhas anteriores por ser a primeira

linha. O algoritmo realmente começa a fazer sentido a partir da segunda linha do mapa, isso porque é de agora em diante em que será definido qual o total gasto para percorrer determinado caminho, qual será o caminho a ser seguido, sendo esquerda ou direita e qual sua posição. Na figura acima, percebemos que na posição i = 1 e j = 0 (segundo linha e primeira coluna) o melhor caminho é o de valor 1, isso porque o valor total até o momento seria 0 (valor da posição atual) + 1 (valor do menor caminho), enquanto caso fosse para esquerda (de valor 2) o custo total seria 2. Chegando na última linha teremos o valor total para percorrer o menor caminho de cada célula, sendo assim, iremos saber qual o valor total mínimo para sair da caverna e também qual o caminho, já que cada célula (subproblema), armazena qual é o subtotal e as devidas posições a serem seguidas.

	0		1	
0	line	0	line	0
	column	0	column	-1
	prevLine	-1	prevLine	-1
	prevColumn	-1	prevColumn	-1
	value	2	value	1
1	line	1	line	1
	column	0	column	1
	prevLine	0	prevLine	-1
	prevColumn	1	prevColumn	-1
	value	1	value	-1
2	line	2	line	2
	column	0	column	1
	prevLine	1	prevLine	1
	prevColumn	0	prevColumn	0
	value	2	value	1

Figure 4: Tabela de subproblemas preenchida pós execução.

Na imagem acima temos a tabela de subproblemas preenchida após execução do algoritmo, sendo assim, o menor caminho começa na posição $i = 2$ e $j = 1$ (**line** e **column** respectivamente) apresentando valor total de 1 (**value**) e o próximo caminho a ser seguido será na posição $i = 1$ e $j = 0$ (**prevLine** e **prevColumn** respectivamente). Em seguida, a partir da posição $i = 1$ e $j = 0$ o próximo caminho será na posição $i = 0$ e 1 e a função finaliza a execução.

A função *lessInitialValue* retorna qual será o subproblema de menor custo para resolução e será utilizada na função *move*. Por fim, é na função *move* em que **Samus** percorre o menor caminho para sair da caverna, além disso é nesta função em que verificamos se **Samus** consegue vencer a missão, ou seja, se o tempo total para percorrer o caminho é igual ou inferior ao tempo que a lava gasta para preencher a caverna.

3 Funcionalidades Extras

As Funcionalidades extras que foram implementadas são as de plotar gráficos para analisar o desempenho do tempo do algoritmo para diferentes tamanhos de entrada, também é apresentada

a melhoria da interface mostrando de forma gráfica o melhor caminho e funções para geração de arquivos de entrada.

3.1 Tempo de Execução

Para calcular o tempo de execução gasto para diferentes tamanhos de entrada foi utilizado na implementação a biblioteca em C *time.h*, com as váriaveis *clock-t* e *CLOCK-PER-SEC* para realizar o cálculo do tempo de execução em segundos, dessa forma, foram utilizadas as entradas 5 de altura e 5 de largura, 10 , 50, 100, 200 e 250, onde pode-se observar no gráfico abaixo 5 um menor tempo para a menor entrada em relação as demais.

Figure 5: Gráfico do tempo de execução gasto.

3.2 Melhor caminho

No melhor caminho foi implementada a função *printPath* onde é exibida a tabela com os valores que são recibidos pelo arquivo de entrada onde as rochas estão representadas por hashtags, os monstros por pontos e a Samus e representado pelo emoticon que vai sendo apresentado a cada movimentação que o mesmo realiza dentro do mapa como é apresentado na imagem abaixo 7.

Figure 6: Execução do melhor caminho.

3.3 Gerar arquivos de entrada

Para gerar os arquivos de entrada foi criado a pasta *Generator* que possui o arquivo *generate.h*, onde é apresentado as função para criar um mapa vazio com as dimenções desejadas, além da função para gerar inteiros aleatórios a partir do nível inserido. Dessa forma foi implementado opções para cinco níveis diferentes, a partir do nível selecionado e gerado um número aleatório para os monstros do mapa. Os valores aleatórios para cada nível estão apresentados na tabela abaixo:

Nível	Intervalo de valores
1	0 a 50
2	51 a 150
3	151 a 300
4	301 a 550
5	551 a 999

Figure 7: Tabela de níveis.

4 Execução

O trabalho possui um arquivo makefile contendo um conjunto de diretivas usadas para automação de compilação, execução e remoção de arquivos binários e serão apresentados em seguida.

```
all:
	gcc src/main.c src/menu.c src/cave.c src/samus.c src/subproblems.c -o dist/main
run:
	dist/main
generate:
	gcc generator/generate.c -o generator/generate
generate_run:
	generator/generate
clean:
	rm dist/main
clean_exec:
	rm dist/main.exe
```

No terminal é apresentada a linha para inserir o arquivo de entrada necessária para iniciar o programa como é mostrado na Figura 8, dessa forma após a leitura do arquivo é apresentado o mapa de acordo com a entrada que foi inserida como é exibido na Figura 9, onde as rochas são representadas por -1, posteriormente temos o cálculo para o tempo gasto por Samus e o tempo da lava e a mensagem "Samus sai das cavernas em segurança!" caso o caminho consiga ser completado ou "Samus falhou na missão" se não conseguir completar o caminho, como também é exibido o caminho percorrido pelo mesmo, ao final é apresentado um menu com as opções para ver o melhor caminho graficamente ou mostrar o tempo gasto para a execução da entrada passada na leitura do arquivo.

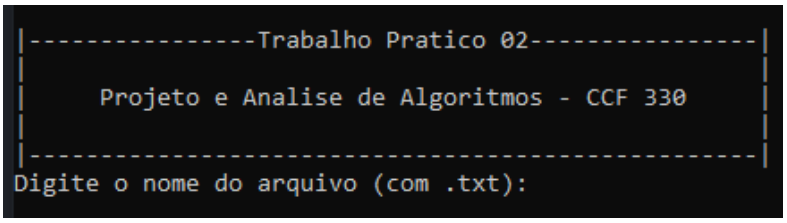

Figure 8: Menu para inserir o arquivo de entrada.

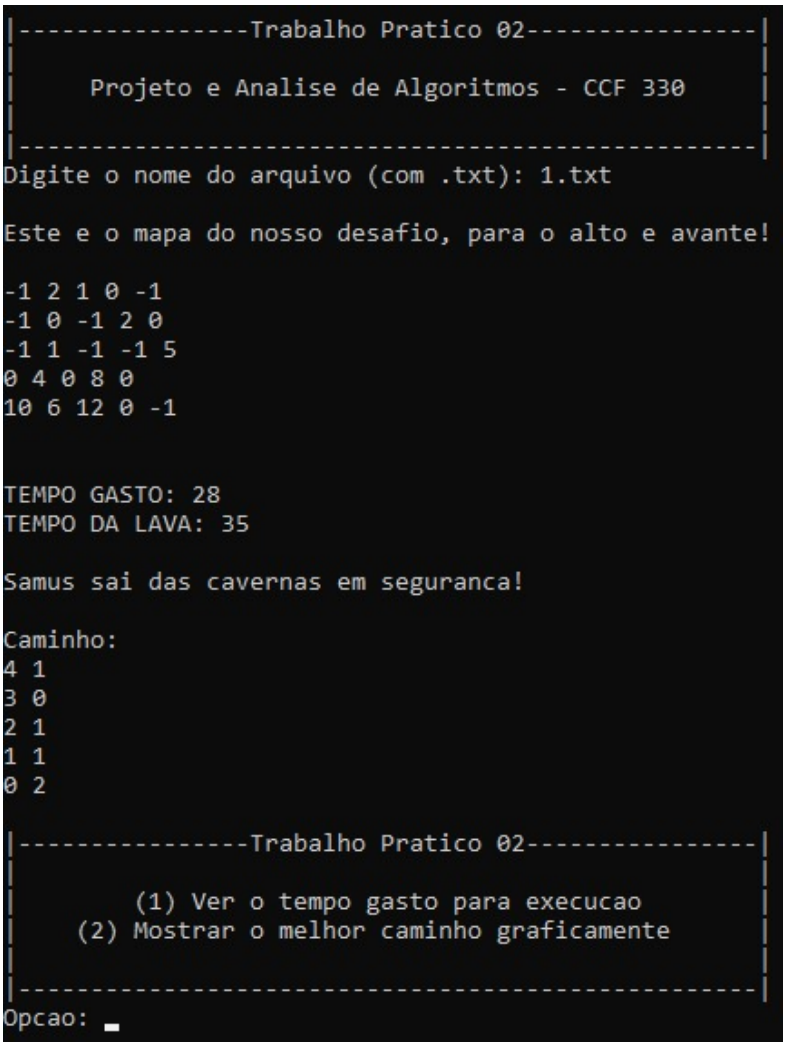

Figure 9: Execução.

5 Conclusão

Desta forma, podemos concluir que o trabalho em questão foi desenvolvido conforme o esperado, atingindo todas as especificações requeridas na descrição do mesmo, já que o intuito principal do trabalho foi atingido, utilizando programação dinâmica, a criação de um algoritmo capaz de escolher um caminho que permita que Samus saia das cavernas em segurança no menor tempo possível

Tivemos algumas dificuldades relacionadas a implementação do algoritmo que utiliza-se programação dinâmica para o caso proposto, porém posteriormente conseguimos compreender melhor a lógica e funcionamento do algoritmo, principalmente como utilizar subproblemas e assim construí-los para sanar o problema.

Posto isso, é válido dizer que apesar das dificuldades na implementação do código, o grupo foi capaz de superar e corrigir quaisquer erros no desenvolvimento dos algoritmos além de implementar tarefas extras. Haja vista que o trabalho foi executado conforme o planejado, sendo tratado tudo que foi pedido pelo professor e monitores. Por fim, verificou-se a assertiva para o objetivo do projeto em implementar um programa utilizando *programação dinâmica* capaz de escolher um caminho que permita que Samus saia das cavernas em segurança no menor tempo possível.

References

[1] Programacao dinamica. https://lamfo-unb.github.io/2019/05/30/Programacao-Dinamica/. (Accessed on 06/03/2022).

[1] https://www.geeksforgeeks.org/floyd-warshall-algorithm-dp-16/, Acesso em 07 de Março de 2022;

[2] https://github.com/riamaydewi/ShortestPath , Acesso em 05 de Março de 2022;

[3] https://www.geeksforgeeks.org/partition-a-set-into-two-subsets-such-that-the-difference-of-subset-sums-is-minimum/ , Acesso em 05 de Março de 2022;

[4] https://www.geeksforgeeks.org/find-the-longest-path-in-a-matrix-with-given-constraints/ , Acesso em 05 de Março de 2022;

[5] https://www.simplilearn.com/tutorials/data-structures-tutorial/floyd-warshall-algorithm-for-using-dynamic-programming , Acesso em 05 de Março de 2022;

[6] https://www.javatpoint.com/floyd-warshall-algorithm, Acesso em 05 de Março de 2022;