Trabalho Prático 2
Andando na Física

Gilson Urbano Ferreira Dias - 2015430959
guf@ufmg.br

1. Introdução

O problema proposto é um Labirinto, um labirinto geralmente é constituído por um conjunto de percursos intrincados criados com a intenção de desorientar quem os percorre. No caso em que iremos trabalhar temos alguns detalhes que devem trabalhados, o labirinto contém portas, chaves e buracos de minhoca (Portais), temos de entrada variáveis que informam o tamanho do mapa e quantidade de chaves que o jogador pode utilizar durante a execução, a ideia é modelar o problema utilizando grafos e procurar o menor caminho entre a entrada (V) e a saída (E).

3	V	d	D	.
2	#	#	.	.
1	#	#	00	c
0	E	.	.	C
	0	1	2	3

Figura 1 - Labirinto

Na Figura 1, temos o exemplo de uma entrada do programa, a partir da entrada podemos fazer uma modelagem utilizando grafos, um modelo possível e que será utilizada na solução do problema pode ser visualizada na Figura 2.

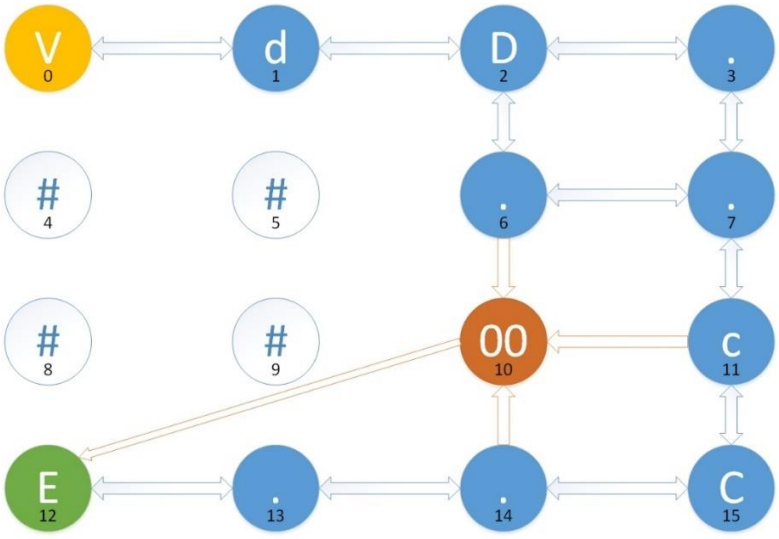


Figura 2 – Modelagem em Grafos

Cada posição do grid que representa a entrada será modelada por um vértice e as ligações entre os vértices serão modeladas por arestas sem peso e direcionadas, o caractere ‘#’ significa paredes ele não terá nenhuma ligação com os outros vértices, o ’.’ é um espaço que pode percorrido sem problemas, letras minúsculas ‘c’, ‘d’, ‘h’ e ‘s’ são chaves que abrem as portas representadas por ‘C’, ‘D’, ‘H’, e ‘S’ respectivamente, todas as arestas que ligam esses vértices são bi direcionadas, o caso especial de aresta que terá apenas uma direção ocorrerá no vértice que modela o buraco de minhoca, os vértices próximos levam ao buraco de minhoca mas o buraco de minhoca direciona para um vértice apenas, como se ele tele transportasse o jogador até o vértice direcionado.

2. Solução do Problema

A implementação foi criada a partir da definição do Tipo Abstrato de Dados (TAD), com cinco estruturas, a primeira de nome Control armazena as variáveis que iram controlar e definir a execução do programa N, M, T, indicando: Quantidades de Linhas, Caracteres e Chaves, respectivamente, na segunda estrutura Vertex temos a representação dos vértices, a estrutura Map irá reunir todos o vértices que representam o labirinto, Graph é onde armazenamos as ligações (arestas) dos vértices de acordo seus detalhes decididos na modelagem, para armazenar as ligações (arestas) será utilizando lista de adjacência, a última estrutura de nome Player armazena as chaves encontradas no mapa, respeitando o limite definido na entrada.

2.1 Funções e Procedimentos

As funções foram declaradas da seguinte forma no Tipo Abstrato de Dados:

void createMap(Map *maze, Control variables); : Essa função aloca memória dinamicamente, para que seja recebida a entrada que contêm o mapa do labirinto, durante a alocação as variáveis da estrutura Map são devidamente inicializadas.

void getMap(Map *maze, Control variables); : E a função que começa dar vida ao programa, ela irá receber da entrada padrão o mapa que representa o labirinto, e armazenar para uso futuro.

void insert(Map *maze, Graph *grafo, Control variables); : Função que a partir do mapa do labirinto verifica as possíveis ligações entre os vértices e os conecta com arestas caso haja alguma ligação.

void buscaLargura(Map *maze, Graph *grafo, Graph *aux, Control variables, int inicio, Graph *caminhos); : Procurando a solução do problema a primeira ideia foi utilizar a busca em largura para gerar um outro grafo de caminhos possíveis, podemos dar um exemplo de como essa função funciona a partir da entrada da Figura 1.

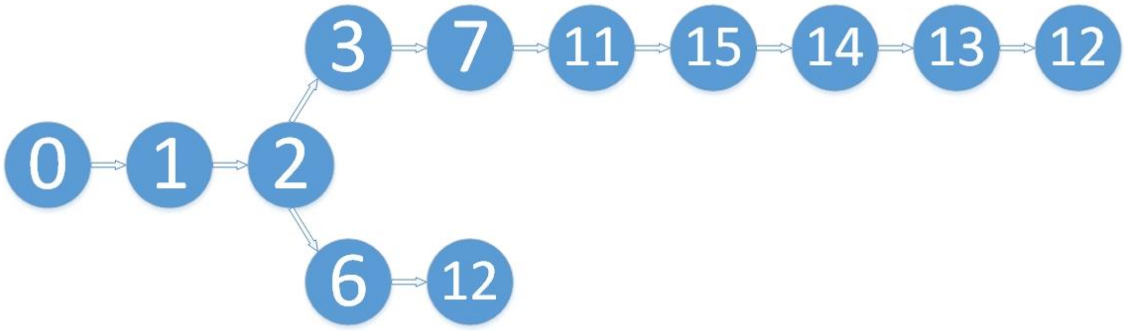


Figura 3 - Caminhos Possíveis

Na Figura 3 temos um grafo que mostra quais são os caminhos possíveis até o vértice final (12), é isso que essa função irá criar, possíveis caminhos do vértice inicial até o vértice final, o vértice do buraco de minhoca não precisa ser representado nesse caso ele funciona como um portal que irá direto ao vértice que ele direciona.

void DFS(Map *maze, Graph *caminhos, Graph *grafo, Graph *aux, Control variables, int inicio); : Utilizando a informação da buscaLargura(), iremos dividir os caminhos possíveis para facilitar uma futura verificação de cada caminho, a função DFS irá realizar uma busca em profundidade no grafo que a busca em largura gerou, aplicando a DFS() na Figura 3 que representa a saída da buscaLargura(), teremos o seguinte resultado apresentado na Figura 4.

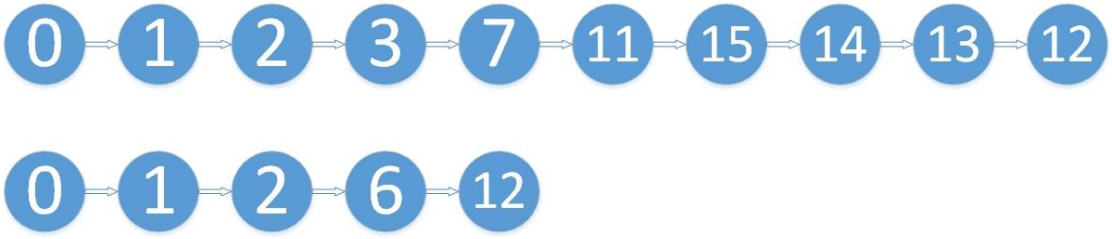


Figura 4 - Caminhos Possíveis Separados

O resultado são os possíveis caminhos separados o que irá facilitar a verificação se o caminho é valido ou não.

void VerificaCaminhos(Map *maze, Graph *aux, Graph *grafo, Player *one, Control variables); : Essa função finaliza a lógica que foi pensada para a resolução do problema, com os caminhos possíveis separados ela apenas irá verificar se é possível ou não realizar o percurso com a quantidade de chaves que podemos ter mãos, depois dessa verificação será imprimido o menor caminho, caso de nenhum caminho ser solução é colocado na saída o número -1, agora aplicando essa função na saída da DFS() (Figura 4), e podendo ter apenas uma chave em mãos teríamos o seguinte resultado, representado na Figura 5.

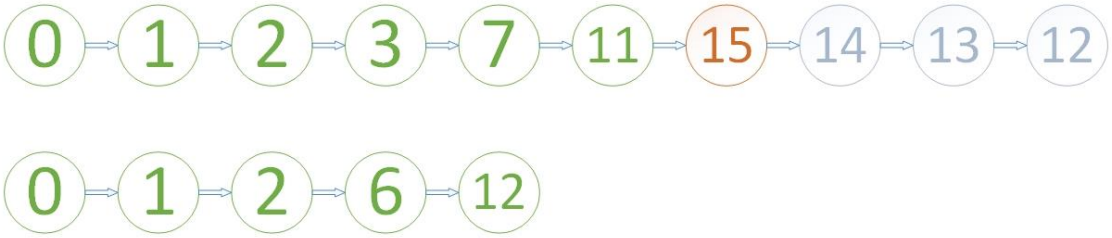


Figura 5 - Verificação dos Caminhos Possíveis Separados

A verificação testa o primeiro caminho, começando no vértice 1, pega a chave ‘d’ abre a porta ‘D’ no vértice 2, passa pelos vértices 3 e 7 sem nenhuma interrupção por serem espaços livres, no vértice 11 encontra outra chave mas não pode mais pegá-la então no vértice 15 ele encontra uma porta que não pode ser aberta pois não temos a chave, esse vértice também não é o vértice final, com isso se encerra a verificação desse caminho e o considera como inválido, no segundo caminho ele consegue chegar até o vértice final com a restrição de apenas uma chave com isso o programa retorna o menor caminho válido que nesse caso será 4.

2.2 Programa Principal

No programa principal são declaradas algumas variáveis importantes, que serão utilizadas ao longo de toda a execução, elas são one do tipo Player, maze do tipo Map, variables do tipo controle, grafos, caminhos e percursos do tipo Graph. No mais são realizadas apenas chamadas das funções que fazem com o programa funcione devidamente.

2.3 Organização do Código, Decisões de Implementação e Detalhes Técnicos

O código está dividido em três arquivos principais: TP2.c e TP2.h implementam o Tipo Abstrato de Dados enquanto o arquivo main.c implementa o programa principal.

O compilador utilizado foi o GCC 5.4.0 no Sistema operacional UBUNTU 16.04.

3. Análise de Complexidade

void createMap(Map *maze, Control variables); : Função simples tem dois loops aninhados que alocam e inicializam, com isso temos uma complexidade de tempo O(N*M), onde N*M é o número de vértices de acordos com a modelagem utilizada sendo assim temos uma complexidade O(|V|) para tempo, e O(|V) para espaço pois todos os vértices são armazenados em memória.

void getMap(Map *maze, Control variables); : Assim como na createMap() essa função tem complexidade O(N*M), onde N*M é o número de vértices sendo assim tempo uma complexidade O(|V|) para tempo e O(1) para espaço.

void insert(Map *maze, Graph *grafo, Control variables); : A função tem que percorrer todo o mapa que está representado por uma matriz então tempo uma complexidade de tempo O(N*M) onde N*M é o número de vértices sendo assim tempo uma complexidade O(|V|) para tempo e O(1) para espaço.

void buscaLargura(Map *maze, Graph *grafo, Graph *aux, Control variables, int inicio, Graph *caminhos); : Essa função irá visitar todos os vértices uma vez e passar por todas as arestas uma vez então a complexidade de tempo é O(|V| + |A|), e O(1) para espaço.

void DFS(Map *maze, Graph *caminhos, Graph *grafo, Graph *aux, Control variables, int inicio); : Assim como a busca em largura essa função tem complexidade O(|V| + |A|) pois todos os vértices e arestas serão visitados uma vez, e também O(1) para espaço.

void VerificaCaminhos(Map *maze, Graph *aux, Graph *grafo, Player *one, Control variables); : Por ter que verificar todos os caminhos gerados essa função terá complexidade O(caminhos*N*M), caminhos é uma constante e N*M é o número de vértices então a complexidade dessa função será O(|V|) para tempo e O(1) para espaço pois temos apenas comparações.

4. Avaliação Experimental

4.1 Testes

Vários testes foram realizados com o programa de forma a verificar o seu perfeito funcionamento. Os testes foram realizados em Intel Core i5, 6Gb de memória com um Intel Core i5-4210U rodando a 1.70GHz, e sistema operacional Ubuntu 16.04 LTS.

Os testes foram realizados dez vezes para cada entrada, foi feita uma média simples dos tempos de cada entrada, então pudemos plotar os gráficos mostrados a seguir:

No primeiro teste não contamos com nenhuma chave no labirinto, o mapa é como se fosse um deserto apenas uma entrada e saída e todas as outras posições estão livres, exemplo na Figura 6.

V	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	E

Figura 6 - N = 10 M = 10 T = 0

Um caminho possível com está ilustrado de verde na Figura 6, as entradas foram feitas a partir desse modelo, várias entradas foram criadas e testadas, a partir disso pudemos criar um gráfico de tempo, o resultado do primeiro teste pode ser visualizado abaixo.

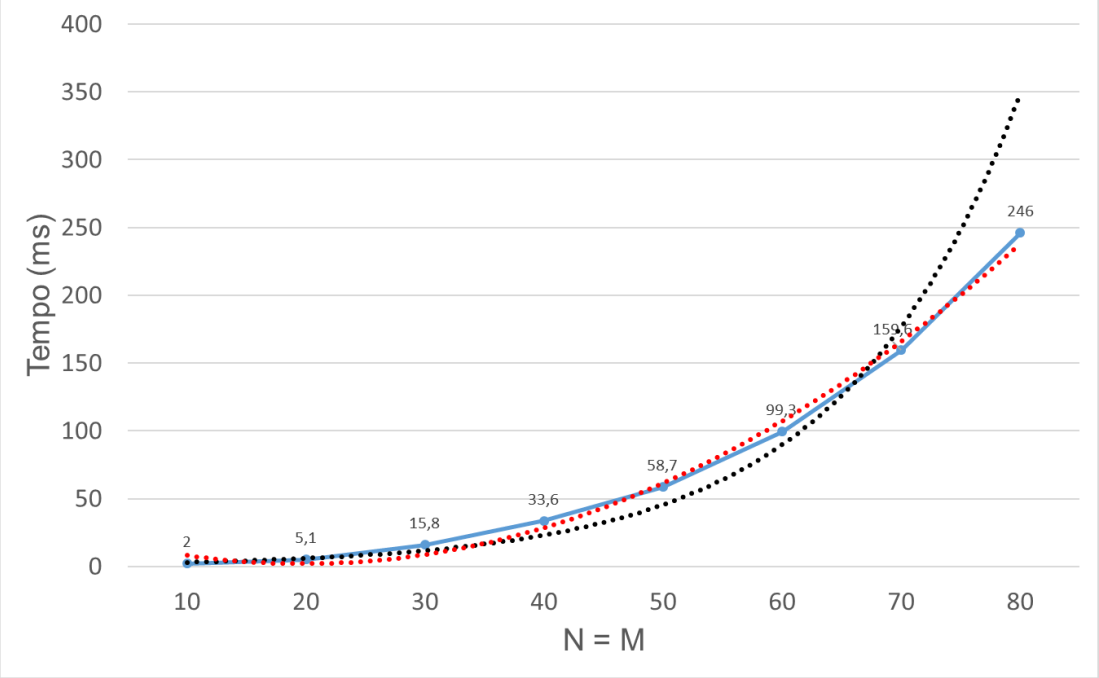


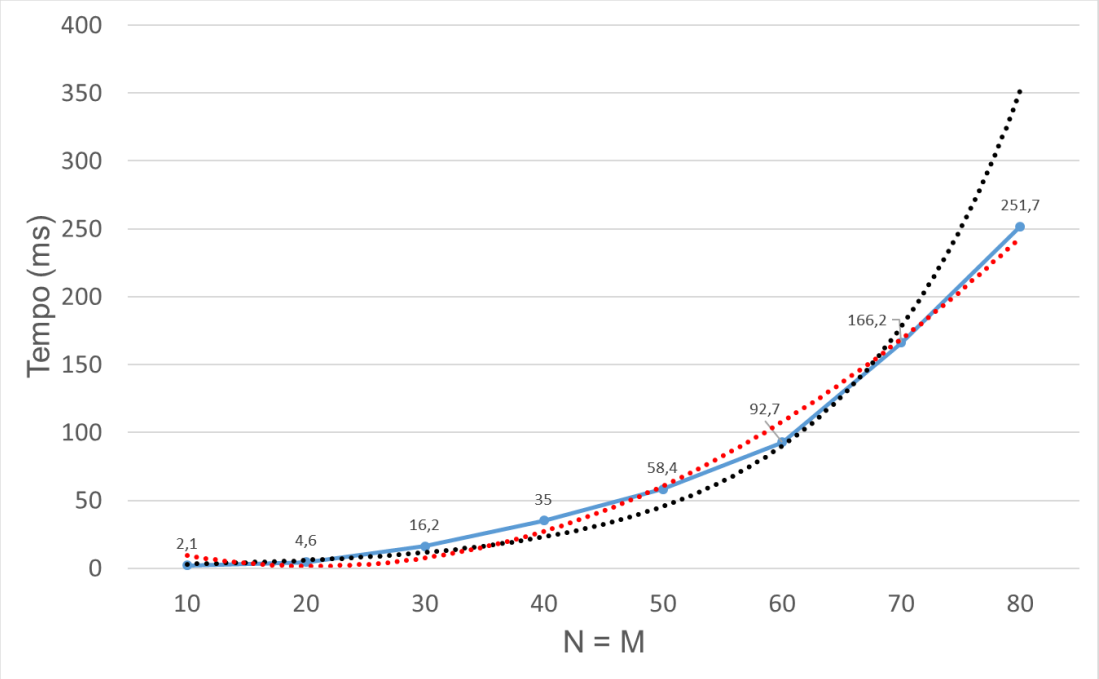
Figura 7 - N = M e T = 0

No gráfico de linha cor azul foram colocadas duas linhas de tendências, a linha preta tem uma tendência exponencial e a linha vermelha um polinômio de grau 2, o método empregado para solucionar o problema gerando todos os caminhos possíveis pode não se sair muito bem nesse caso, pois o problema é por definição não polinomial, entradas grandes podem gerar muitos caminhos a serem verificados, com isso caminhos pode mudar a complexidade do programa fazendo com que ele tenha complexidade quadrática, o gráfico comprova a dificuldade de resolver o problema para esse caso.
Um segundo teste foi feito a fim de verificar o quanto as chaves no caminho aumentam a complexidade do problema, as quatros chaves possíveis foram colocadas no percurso realizado pelo algoritmo e todas as posições são portas, o modelo pode ser visto na figura a seguir.

V	c	d	h	s	C	D	H	S	C
c	d	h	s	D	D	D	D	D	D
d	h	s	H	H	H	H	H	H	H
h	s	S	S	S	S	S	S	S	S
s	C	C	C	C	C	C	C	C	C
D	D	D	D	D	D	D	D	D	D
H	H	H	H	H	H	H	H	H	H
S	S	S	S	S	S	S	S	S	S
C	C	C	C	C	C	C	C	C	C
D	D	D	D	D	D	D	D	D	E

Figura 8 - N = 10 M = 10 T = 4

Mais uma vez um caminho possível está colorido de verde, o resultado dessas entras está no gráfico abaixo.



Os resultados obtidos são muitos próximos aos do teste 2, o diferencial foi um aumento de tempo, mas muito pequeno em relação ao primeiro teste, isso comprova o fato de que as chaves no labirinto aumentam o tempo de execução, como já era esperado, novamente duas linhas de tendência foram traçadas, a vermelha um polinômio de grau 2, e a preta uma exponencial, o gráfico fica entre as duas tendências.

5. Conclusão

A implementação ocorreu de forma tranquila, os algoritmos já haviam sidos estudados na disciplina de Algoritmos e Estruturas de Dados III, foi necessário algumas pesquisas e revisões sobre o assunto a fim de verificar alguns detalhes sobre as estruturas de dados a serem utilizadas, a modelagem em grafos afim de solucionar todos os problemas também foi um desafio, mas por fim o trabalho pode ser concluído, com o resultado ficando dentro do esperado.

O programa foi pensando não apenas com o objetivo de solucionar o problema, mas de encontrar uma solução com complexidade baixa, por isso a decisão de utilizar busca em largura e busca em profundidade para solucionar o problema e não o Djikstra por exemplo, pois os dois algoritmos de busca tem complexidade linear para número de arestas e vértices já o algoritmo de Djikstra é logarítmico.

Após a análise de complexidade e testes podemos concluir o quanto o número de caminhos possíveis entre a entrada e a saída interfere no programa, casos onde o número de caminhos possíveis é pequeno o algoritmo tem complexidade linear, isso ocorre em entradas bem definidas, mas nos testes

queríamos mostrar que se o número de caminhos possíveis é alto a complexidade é diferente, entradas com apenas um caminho possível como os labirintos clássicos o algoritmo se sai muito bem mesmo com muitas chaves e portas no percurso, agora entradas que definimos acima como desertos, com muitas possibilidades de caminhos possíveis a complexidade cresce muito e o termo caminhos da função VerificaCaminhos() deve ser considerado, com isso podemos dizer que a complexidade geral do programa é O(caminhos * |V|).

6. Referências

[1] Ziviani, N., Projeto de Algoritmos com Implementações em Pascal e C, 4ª Edição, Editora Pionera, 1999
[2] André Backes, Linguagem C: Completa e Descomplicada, 1ª Edição, Elsevier, 2012
[3] Kyle Loudon, Mastering Algorithms with C, 1ª Edição, Editora O'Reilly
[4] Dennis M. Ritchi, C Programming Language, 2ª Edição
[5] Robert Sedgewick, Algorithms, Algorithms in C, 1990, ADDISON-WESLEY
[6] Thomas H. Cormen ..., Algoritmos - Teoria e Prática, 2ª Edição, Elsevier, 2002
[7] Jon Kleinberg, Eva Tardos, Algorithm design, 1ª Edição, Pearson Education, 2006
[8] Jayme Luiz Szwarcfiter e Lilian Markenzon, Estruturas de Dados e Seus Algoritmos, 1994, LTC