Documentação - Trabalho Prático 2

Esthefanie Pessoa Lanza - esthefanielanza@gmail.com

Novembro, 2016

1 Introdução

O objetivo principal deste trabalho é aplicar os conceitos de modelagem de grafos visto na matéria de Algoritmos e Estruturas de Dados III e utilizar um dos algoritmos apresentados em aula para encontrar um caminho mínimo entre dois pontos.

O problema tem como objetivo ajudar Vinícius a sair dos laboratórios de física e encontrar o caminho de volta para o DCC. Em sua jornada, ele encontrará diversos obstáculos como becos sem saída, portas trancadas e buracos de minhoca. Entretanto, Vinícius possui algumas limitações, ele não pode carregar mais do que determinado número de chaves e, por ser muito apegado as suas coisas, não consegue trocar uma chave depois que pega ela. Em relação aos obstáculos, os buracos de minhoca desaparecem assim que são usados uma vez e cada porta só pode ser aberta por uma chave específica. Não são permitidos movimentos na diagonal. Os mapas utilizados para entrada seguem o exemplo da figura abaixo:

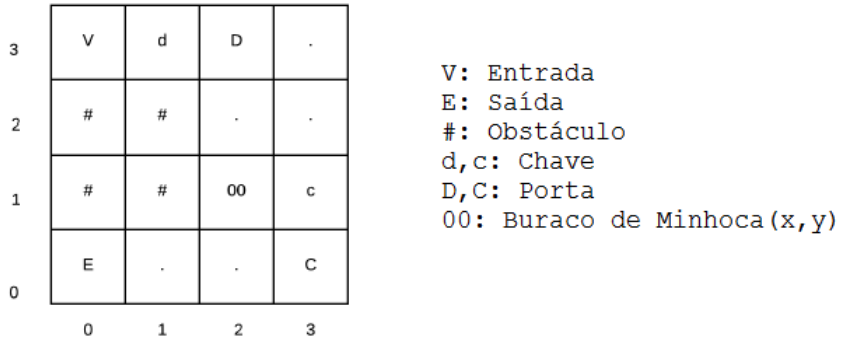


Figure 1: Exemplo mapa de entrada do problema

O algoritmo escolhido para calcular os caminhos mínimos foi o busca largura, já que o mapa do laboratório não possui pesos, e este será melhor discutido nas próximas seções.

2 Solução do Problema

2.1 Modelagem

Para a implementação do problema foi utilizado uma matriz n por m referente ao tamanho do labirinto. O grafo modelado, não ponderado e não direcionado, tem como vértice uma casa do mapa e como adjacentes as posições em volta do mesmo sendo estas no máximo quatro. Cada casa é interpretada como uma variável do tipo TVertice que possui os seguintes campos:

- Chave: o caractere que representa a posição;
- Distancia: a distancia mínima até aquele local;
- Cor: Se o vértice já foi visitado ou não;
- nChaves: O número de chaves necessárias para atingir aquele vértice;
- usouBuracoMinhoca: Um vetor que identifica se algum buraco de minhoca foi utilizado para atingir aquele vértice;

A partir da matriz, que é inicializada com um valor infinito para as distancias e zeros para os outros campos, é realizada a busca em largura a partir da posição inicial até todos os outros vértices alcançáveis. Se outra chave é encontrada, colocamos sua posição em outra fila para que a busca em largura seja realizada a partir desse ponto, considerando todas as novas posições alcançadas com a chave em mãos. Por outro lado, se um buraco de minhoca é encontrado finalizamos a busca em largura que está ocorrendo no momento e iniciamos uma nova a partir da saída de buraco de minhoca, isso ocorre pois ao adentrar em um buraco, vértices podem ser revisitados sendo necessário recolorir todos de branco para recalcular as novas distancias.

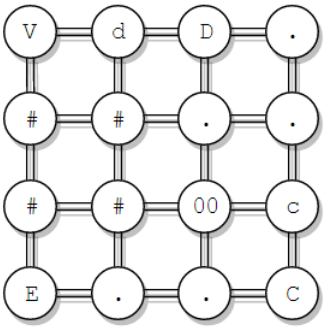


Figure 2: Grafo modelado para o mapa apresentado na figura 1

Em relação ao número de chaves limitado, a cada porta diferente atravessada é aumentado o número de chaves necessárias para passar por aquele caminho.

Esse número é associado a cada vértice e repassado aos seus adjacentes quando encontramos um caminho melhor, menos custoso, ao caminhar pelo mapa. Para os buracos de minhoca o algoritmo é similar, a cada novo caminho, se ele for melhor, marcamos que usamos determinado buraco de minhoca no grafo no campo usouBuracoMinhoca[b], sendo b um buraco qualquer.

2.2 Implementação

O fluxo do programa é iniciado com a leitura do tamanho e do mapa o qual rodaremos o algoritmo para encontrar o menor caminho. Nesse passo, todas os vértices serão preenchidos com um caractere referente a sua função no mapa, as distancias são inciadas como infinito, menos a posição inicial, e são guardados os pontos de inicio e fim. Ainda na fase de leitura, cada buraco de minhoca é armazenado no labirinto como apenas um número que vai de 0 até o número máximo permitido de buracos, usando este índice podemos acessar, em um vetor adicional, para qual posição do mapa ele levará.

Com o grafo já preparado, iniciamos a rotina para encontrar o menor caminho. Primeiramente criamos um vetor de chaves, para sinalizar se já temos uma chave em mãos. Além disso, são inicializadas uma matriz auxilar para calcular as distancias de uma chave ou buraco de minhoca até todas as outras posições, sem alterar as distancias minimas do grafo. Uma matriz similar será utilizada para armazenar se um buraco de minhoca foi utilizado ou não para acessar uma posição com determinado custo.

Enfim, a busca em largura é iniciada. Em um primeiro momento colocamos a posição inicial em uma fila e enquanto essa fila não estiver vazia, desenfileiramos um vértice, adicionamos seus adjacentes em uma fila, se forem válidos, o colorimos de cinza até que todo o mapa tenha sido percorrido e todas as distancias tenham sido calculadas. Para verificar os adjacentes apenas caminhamos para esquerda, direita, cima e baixo no grafo caso estas não ultrapassem os limites do mapa. Encontrando um vértice válido realizamos a rotina descrita no algoritmo 1.

Caso o vértice em questão tenha uma restrição de chaves maior do que a permitida para Vinícius não podemos acessar esse caminho e retornamos. Por outro lado se é um vértice não visitado e não é um obstáculos continuamos a execução. Se é encontrada uma porta diferente das já utilizadas por esse caminho, aumentamos a restrição de chaves do vértice em 1 e verificamos se já possuímos a chave. Caso a resposta seja positiva, anotamos no grafo se encontrarmos uma distancia inferior e em seguida enfileiramos o vértice.

Por outra lado, se encontras um buraco de minhoca finalizamos a busca em largura atual e iniciamos a próxima a partir do destino do buraco. Se destino de um buraco de minhoca for outro buraco de minhoca entramos em um loop até que esta condições seja disfeita, no ultimo buraco trocamos os valores de x e y para o destino final. Em seguida, recalculamos as distancias caso a encontrada seja menor do que a do vértice anterior mais uma unidade e enfileiramos o vértice.

Caso o vértice seja uma chave e esta ainda não foi adquirida enfileiramos sua posição para iniciar uma nova busca largura a partir deste ponto posteriromente.

Algorithm 1: Insere Adjacentes(Grafo, distanciasTemp, xGrafo, yGrafo, x, y, xAnt, yAnt, fila, filaChaves)

```
if nChavesMax < nChavesVertice then
    return
end
if Grafo[x][y].chave! = Grafo[x][y].cor == 0 then
    if Grafo[x][y].chave == Porta then
        encontrouPorta(Grafo, distanciasTemp)
    end
    while Grafo[x][y].chave == Buraco Minhoca do
        buracoAtual = converteBuraco(Grafo[x][y].chave)
        if Grafo[xAnt][yAnt].usouBuraco[buracoAtual == 0] then
            encontrouBuraco(Grafo, distanciasTemp)
        end
        else
            break
        end
    end
    recalculaDistancias(Grafo. distanciasTemp)
    if Grafo[x][y] == Chave then
        if pegouChave[Chave] != 1 then
            enfileira(filaChaves, x, y)
        end
    end
    enfileira(fila, x, y)
end
```

Este algoritmo será executado enquanto houverem vértices a serem visitados. Caso exista outra chave no labirinto a busca em largura será chamada recursivamente para cada chave. No entanto, as distancias presentes no grafo só serão substituídas caso encontremos um caminho menos custoso. Dessa forma, utilizaremos a matriz de distancias temporárias já mencionada anteriormente para calcular todos os caminhos possíveis a partir de uma chave. Assim, poderemos calcular distancias para caminhos que anteriormente não teriam sido visitados devido a ausência da chave.

Realizada a busca, imprimimos na tela um único número referente a distancia mínima entre a posição de Vinícius e o prédio do DCC. Caso não exista um caminho, imprimiremos -1 no stdout e o pobre Vinícius ficará perdido para sempre.

3 Análise de complexidade

Nesta seção, será apresentada a análise do custo teórico de tempo e de espaço.

3.1 Análise Teórica do Custo Assintótico de Tempo

Para a análise assintótica foram observadas as funções principais do código de maneira geral, sendo elas a responsável pela leitura do Grafo e a busca em largura.

- A função leituraGrafo é responsável por inicializar os valores dos vértices com os caracteres correspondentes e preencher os outros campos do struct TVertice para preparar o grafo para a busca em largura. Tudo isso é feito com um loop que percorre todas as posições do mapa, ou seja, sua complexidade é correspondente ao número de vértices $O(N \times M)$, sendo N e M as dimensões do grid.

- A função busca em largura também verifica todos os vértices. Para inicializar as distancias temporárias e o uso de buracos de minhoca é utilizado um loop com custo referente aos vértices $O(N \times M)$. Enfileirar tem custo $O(1)$, logo para percorrer toda a fila teremos custo $O(N \times M)$. Teremos também que visitar todos os adjacentes de cada vértice com custo correspondente ao número de arestas que pode atingir no máximo 4 vezes o número de vértices $O(4 \times (N \times M))$. O custo total da função seria o número de vértices mais o número de arestas, ou seja, $O((N \times M) + (4 \times (N \times M))$. Entretanto, para cada chave encontrada ou buraco de minhoca chamamos a função recursivamente. Dessa forma, o custo total da função é a complexidade citada anteriormente vezes o número de chaves, C, e buracos, B, alcançáveis presentes no labirinto $O((C + B) \times ((N \times M) + (4 \times (N \times M)))$. Caso não tenha nenhuma chave ou buraco, o algoritmo apresenta somente o custo de uma única busca em largura.

- A maioria das outras funções, tais quais rotinas para quando encontramos uma porta, buraco ou chave não possuem loops e dessa forma tem custo $O(1)$ e não serão explicadas detalhadamente.

- Para imprimir a solução na tela também temos custo $O(1)$, pois basta imprimir a distancia referente a posição final armazenada no grafo.

- O custo assintótico geral do algoritmo é correspondente ao da busca largura, pois esta é a que possuí maior complexidade dentre as funções presentes no código, ou seja, $O((C + B) \times ((N \times M) + (4 \times (N \times M)))$.

3.2 Análise Teórica do Custo Assintótico de Espaço

Para a análise assintótica foram observadas as funções principais do código de maneira geral, sendo elas a responsável pela leitura do Grafo e a busca em largura.

- A função leituraGrafo aloca espaço para uma matriz de vértices do tamanho do grid, ou seja, sua complexidade é $O(N \times M)$.

- A função encontraCaminhoMínimo, que faz a chamada da busca em largura, aloca espaço para três matrizes. Uma delas é utilizada para guardar as distancias mínimas temporárias de tamanho $N \times M$, outra para identificar se um buraco de minhoca foi utilizado ou não de mesmo tamanho e uma última para identificar se um caminho passou por uma porta ou não. A complexidade da função então se resumo a $O(3 \times N \times M)$, mas como 3 é uma constante podemos reduzir a complexidade para $O(N \times M)$.
- A complexidade espacial do programa em geral também é $O(N \times M)$.

4 Avaliação Experimental

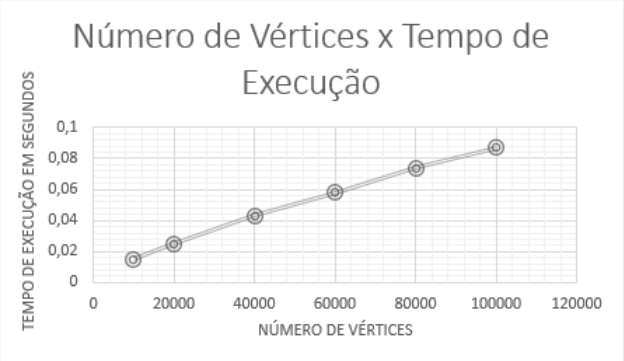
4.1 Metodologia

A implementação da solução proposta foi compilada utilizando gcc 4.8.4. Os experimentos foram executados em um sistema com 4 GB de memória e um processador Intel Core i3 2310K @ 2.1 GHz. Cada teste foi realizado em torno de 30 vezes e foi realizada uma média entre os valores obtidos.

4.2 Avaliação Experimental do Tempo de Execução

4.2.1 Análise do tempo de execução em relação ao número de vértices

Para a análise da variação do tempo em ao número de vértices. Foram criados mapas limpos, sem portas, chaves e buracos de minhoca, e variamos apenas o tamanho do grid para alcançar um determinado número de vértices.



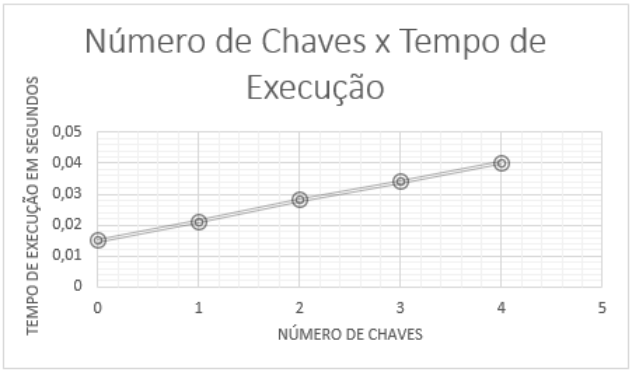
Número de Vértices	Tempo de Execução
10000	0,015
20000	0,025
40000	0,043
60000	0,058
80000	0,074
100000	0,087

Figure 3: Análise da variação do tempo em relação ao número de vértices

Foram usados tamanhos de grid maiores do que o limite especificado para o enunciado, pois os tempos gerados para mapas muito pequenos não apresentavam alterações consideráveis. Entretanto, o tempo de execução do algoritmo agiu como esperado e cresceu linearmente em relação ao número de vértices.

4.2.2 Análise do tempo de execução em relação ao número de chaves

Para esta análise foi criado um teste com um grid de tamanho maior para que as diferenças entre os tempos fosse mais significativas. O tamanho escolhido foi 100 e por 100 e adicionamos uma chave diferente das já presentes no mapa a cada nova execução.



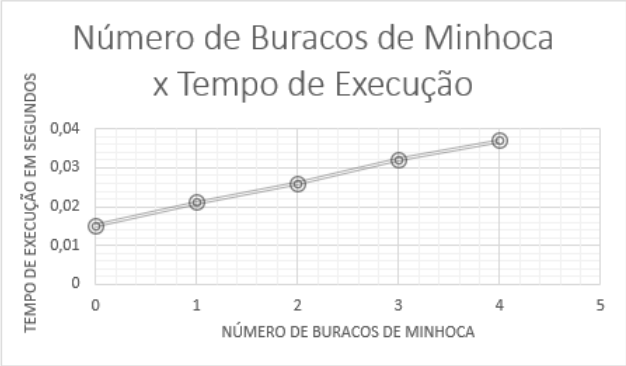
Número de Chaves	Tempo de Execução
0	0,015
1	0,021
2	0,028
3	0,034
4	0,040

Figure 4: Análise da variação do tempo em relação ao número de chaves no tabuleiro

Como o tamanho do labirinto foi mantido constante aumentar o número de chaves também faz com o que tempo de execução aumentasse linearmente. Isso ocorre pois a cada nova chave encontrada realizamos outra busca em largura a partir dela para encontrar novos caminhos que antes não eram acessíveis.

4.2.3 Análise do tempo de execução em relação ao número de buracos de minhoca

O teste foi similar ao das chaves, criamos um grid de tamanho 100 e por 100 e adicionamos buracos de minhoca até o número máximo permitido pelo enunciado.



Número de Buracos	Tempo de Execução
0	472
1	848
2	1224
3	1600
4	1976

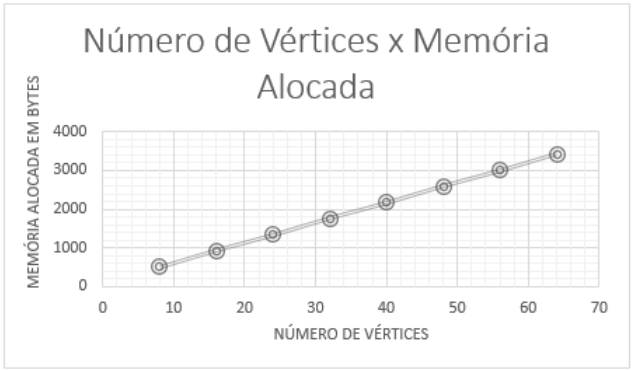
Figure 5: Análise da variação do tempo em relação ao número de buracos de minhoca

Assim como ao aumentar o número de chaves, podemos notar um aumento linear no tempo de execução ao aumentar o número de buracosdeminhoca. Isso acontece pois ao encontrar um buraco devemos iniciar uma nova busca em largura a partir do ponto de saída para que os vértices possam ser revistados.

4.3 Avaliação Experimental da Memória Utilizada

4.3.1 Análise da memória alocada em relação ao número de vértices

Nesse teste aumentamos o tamanho do grid até que ele atingisse o tamanho máximo permitido pelo enunciado do problema. Considerando o número de vértices sendo $N \times$M.



Número de Vértices	Memória Alocada
8	512
16	928
24	1344
32	1760
40	2176
48	2592
56	3008
64	3424

Figure 6: Análise da variação da memória alocada em relação ao número de vértices

Assim como o esperado, o gráfico cresceu linearmente ao aumentarmos o

tamanho do mapa.

5 Conclusão

Na implementação, o maior problema foi encontrar uma solução menos custosa para o uso de chaves no labirinto. Apesar do problema ter sido resolvido sem analisar todas as combinações possíveis, ainda era possível podar o grafo para este apresentar melhores tempos de execução ao lidar com mapas maiores do que os limites especificados.

Além disso, a implementação por matrizes idênticas ao mapa fez com que fosse alocado parte da memória para obstáculos que nunca farão parte da solução do problema, ou seja, parte da memória poderia ter sido economizada neste ponto.

De modo geral, a solução usada não foi muito complexa pois trata-se de modificações simples no algoritmo de busca em largura. Como as casas do mapa não tinham peso, precisamos implementar funções que tratassem apenas quando encontrássemos chaves, portas ou buracos de minhoca.