Trabalho Prático 2: Andando na Física

Francis Carlos dos Santos

Matrícula: 2012022167

1 Introdução

Problemas de caminhamento mínimo em grafos representam uma classe muito importante de problemas em Ciência da Computação. Nesse trabalho é requerido o caminhamento mínimo em um labirinto, mais exatamente, Vinicius está perdido no departamento de fisica da UFMG. O labirinto contém portas, chaves buracos de minhoca e passagens bloqueadas. A entrada é dada em forma de uma matriz $n \times m$. As portas são representadas por letras maiúsculas e a chaves pelas letras minúsculas correspondentes. Os buracos de minhoca são representados por dois caracteres XY, os caminhos livres por um "." e caminhos bloqueados por um jogo da velha. A solução foi modelada utilizando a estrutura de grafos represetado por uma lista de adjacências e o menor caminho possive foi feito utilizando-se o algoritmo de busca em largura.

2 Solução do Problema

2.1 Entrada

Para facilitar e simplificar a solução do problema, a solução foi dividida em três partes principai, processamento da entrada, criação do grafo representado por uma lista de adjacencias e a busca pelo caminhamento mínimo. Cada posição da matriz de entrada foi considerada como sendo um vértice do grafo. Para armazenar a entrada foi criada a estrutura de dados *Vertice*, como pode ser isto no algoritmo abaixo, que armazena as informações necessárias para criação de cada vertice, como a posição *i,j* do vértice na matriz, informação se é um buraco de minhoca ou não e caso verdadeiro as coordenadas do destino.

Algoritmo 1: Estruturas de Dados utilizada na entrada

```
typedef struct{
   bool BuracoDeminhoca
   int x,y,i,j,v,conteudo

}Vertice;
```

A figura 1, mostra como a entrada (a) é recebiada, cada caractére é armazenado com seu valor na tabela ASCII, como mostrado em (b), e no mesmo processo os valores de cada vertice são estabelecidos, como mostrado em (c).

V	d	D	.
#	#	.	.
#	#	00	c
E	.	.	C

(a)

86	100	68	46
35	35	46	46
35	35	00	99
69	46	46	67

(b)

15	14	13	12
11	10	9	8
7	6	5	4
3	2	1	0

(c)

Figura 1: Exemplo de processamento da entrada (a),valores ASCII (b) respectivos vértices (c).

2.2 Criação do Grafo

Foi escolhida a representação do grafo por listas de adjacências. A nó do grafo armazena o vértice *v*, as coordenadas da posição do dado vértice na matriz (*i,j*) e um apontador para o próximo membro na lista. Cada vértice tem sua adjacência adicinada no grafo, excluindo os bloqueios que não são considerados adjacências. É interessante ressaltar que para para esse caso a estrutura da lista além dos adjacentes foi criado um campo *info*, para que fosse possível acessar cada vértice diretamente, isso facilitou muito a implementação da busca do melhor caminho. A estrutura utilizada pode ser vista no algoritmo abaixo. A

Algoritmo 2: Estruturas de Dados Grafo

```
typedef struct listaNo{
  int v,x,y
  struct listaNo *prox
 }no;
typedef struct listaNo{
  bool flag
  no *info,*head
 }adjList;
typedef struct gr{
  int v
  adjList *lista
 }Grafo;
```

figura 2 mostra a representação grafica da lista de adjacências criada para os três primeiros vértices da entrada mostrada anteriomente.

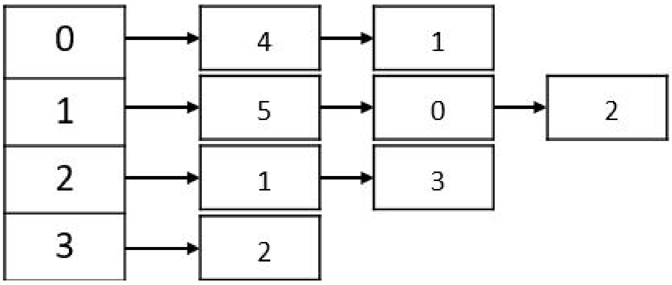

Figura 2: Lista de adjacência para os três primeiros vértices da entrada mostrada em 1

2.3 Busca do Caminho Mínimo

A busca foi implementada baseada em um algoritmo de busca em largura.As funções principais desse processo são descritas abaixo.

- **encontraMenorCaminho**: Essa função é bem similar a uma BFS simples, foi implementada utilizando fila de prioridades dinâmica. A principal diferença de uma busca em largura basica para essa função é que além da verificação se o vertice ja foi previamente visitado, também verificamos se existe um buraco de minhoca. Caso exista o vértice adjacente será aquele indicado pelo buraco de minhoca. Essa função retorna uma estrutura *Caminho*, contendo o vetor de vertices pais do menor caminho e o tamanho desse caminho.

- **verificaCaminho**: Em seguida essa função é chamada para analisar se o o caminho é valido ou não. Essa função realiza uma busca nos vertices Pai de cada vértice do caminho encontrado afim de descobri se existem portas e se existe a chave para essa porta e se essa chave é encontrada antes de passar pela porta. A cada iteração é verificado também se o numero de chaves que o Vinicius pode carregar é valido. O retorno dessa função é um numero inteiro que pode ser -1 caso o caminho seja valido ou qualquer outro inteiro que representará uma posição invalida.

Caso a posição seja invalida é verificado se a posição invalida é adjacente a posição inicial do Vinizin (nessa altura do Campeonato ja somos intímos). Caso essa condição seja considerada verdadera, a função é encerrada pois não existirá caminho até a saída e o Vinicius estará perdido para sempre. Caso a posição invalida não seja adjacente ao vinicius, o grafo voltará a sua condição inicial, com a diferença de que o vértice indicado pela posição inválida retornada será considerado não adjacente. Então o algoritmo irã buscar o caminho novamente.

3 Análise de Complexidade

Nesta seção, será apresentada a análise do custo teórico de tempo e de espaço para as principais funções do algoritmo.

3.1 Análise Teórica do Custo Assintótico de Tempo

Sejam n e m o numero de linhas e o numero de colunas na entrada rescpectivamente. A complexidade gera do algoritmo pode ser encontrada, fazendo uma analise das principais funções. As funçoes *free* e *malloc* serão consideradas como tendo um tempo de execeução $O(1)$. Consideremos também E o numero de arestas do do Grafo criado.

- **LeEntrada**: Essa função faz a leitura da matriz, para fazer tal coisa funciona com dois lopps aninhados.U loop de externo de tamanho n e outro interno de tamanho m. conseiderando que o numero de comparações não varia dentro da função e que o as atribuições ocorrem em $O(1)$. Podemos afirmar que o custo assintótico de tempo dessa função é $O(\mathtt{n} \times \mathtt{m})$.

- **criaListaAdjacencia**: Essa função recebe como entrada a *maze*, duas estruturas de repetição **for**, são o principal gargalo da função. O comprimentos dos loops são n e m, externo e interno respectivamente. Dentro dessa função o numero de compações dentro do loop é sempre constante, ou seja tem uma maxímo. Logo essa função assintóticamente tem uma complexidade de tempo de textitO$(\mathtt{n} \times \mathtt{m})$.

- **encontraMenorCaminho**: Como mensionado anteriormente essa função apenas difere de uma bsuca em largura basica, por uma comparação a mais e um vetor de distancias. Logo como os procedimentos citados ocorrem em tempo constante podemos dizer que a complexidade assintótica de tempo para a função de menor caminho é $O(|\mathtt{n} \times \mathtt{m}| + |\mathtt{e}|)$, que representa o caso em que todos os vértices e arestas são visitados. $O(|\mathtt{n} \times \mathtt{m}|)$ representa que todas as operações realizadas sobre os vértices ocorrem em tempo constante. A mesma definição vale para $O(|\mathtt{e}|)$.

- **verificaCaminho**: Essa função recebe como entrada o Grafo e a matriz e o caminho retornado pela função **encontraMenorCaminho**. Considerando k o tamanho do caminho retornado. temos um primeiro laço que executará k operações. O segundo laço é aninhado e depende intrissecamente do caminho, isso porque o for mais externo tem o executa um numero de operações igual ao número de portas encontrados no caminho e o loop mais interno executa é executado por um numero de vezes iguail ao numero de chaves encontrada no caminho. Considerando um pior caso onde todos os vertices do caminho seriam ou portas chaves e considerando que o número máximo de chaves é três. Considerando que varias portas iguais são enfileiradas no caminho em um numero de p portas. Temos que a complexidade do dado algoritmo é dada por *O(k) + O(p)*, como k representa todo o caminho sabemos que $O(\mathrm{k})$ domina assintóticamente $O(\mathrm{p})$.

Conclui-se assim que a complexidade assintótica de tempo do algoritmo é $O(|\mathtt{n} \times \mathtt{m}| + |\mathtt{e}|)$. O numero de arestas é dependente de condições internas da matriz, como o numero de bloqueios por exemplo.

3.2 Análise Teórica do Custo Assintótico de Espaço

Considerando as estruturas *Grafo* e *Vertice* as principais estruturas pois ocupam espaço consideravelmente maior que qualquer outras. Para a lista de adjacências temos que para um caso mais geral a complexidade de espaço da lista é de $O(|\mathfrak{n} \times \mathfrak{m}|+|e|)$. Para armazenar a matriz, contendo o labirinto, temos uma matriz de $\mathfrak{n} \times \mathfrak{m}$ logo assintóticamente teríamos $O(|\mathfrak{n} \times \mathfrak{m}|)$. Portanto no a complexidade teórica do custo de espaço para esse algoritmo seria $Omax(O(|\mathfrak{n} \times \mathfrak{m}|), O(|\mathfrak{n} \times \mathfrak{m}| + |e|))$. O que nos da $O(|\mathfrak{n} \times \mathfrak{m}| + |e|)$.

4 Análise Experimental

A análise experimental da implementação é mostrada pelas figuras (3) , (4),(5) e (6). Para realizar os experimentos foi utilizado um gerador de casos teste além dos casos teste. O tempo de execução foi obtido pelo terminal ao fim da execução do progama. O espaço utilizado em cada execução foi obtido por meio da utilização do Valgrind. Os testes foram realizados em uma máquina com processdor Core i7-4720HQ 2.6GHz e 8GB de memória ram.

Para a Figura 3, escolhemos variar o número máximo de chaves que o Vinicius poderia carregar de uma só vez. Como pode ser visto abaixo esse número não representa grande influencia ára o algoritmo. Pelo menos para entradas não muito grandes. Como o limite do trabalho é de oitenta e um vértices esse valor tende a não ter grande influência no desempenho do algoritmo. O gráfico da Figura 4 monstramos a variação do tempo em

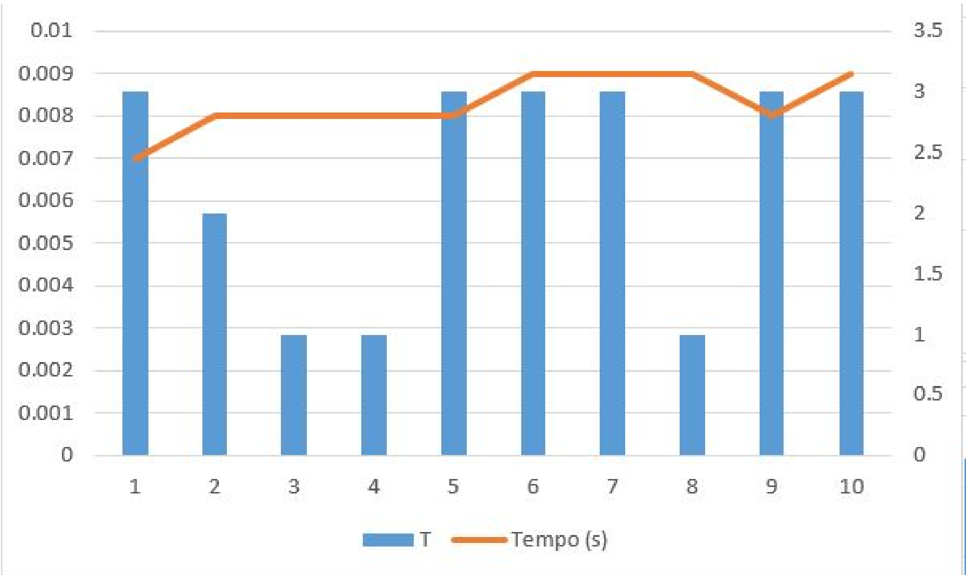

Figura 3: Grafico monstrando como a variação do tempo em função do numero máximo de chaves permitido.

relação ao número de vértices. Como esperado pelo cálculo teorico da complexidade do algoritmo o tempo se relaciona de maneira praticamente linear com o numero de vértices. Obviamente que se, o labirinto conter um grande numero de caminhos bloqueados é esperado que o tempo de execução caia pra esses algoritmos uma vez que o número de vértices será muito diferente do tamanho da matriz de entrada. A figuras 5 e 6 fornecem uma

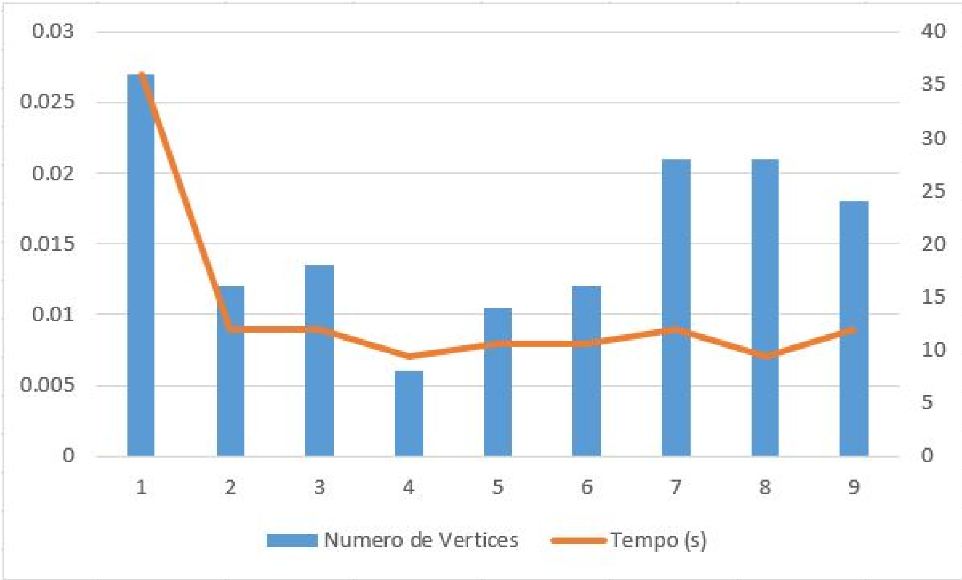

Figura 4: Gráfico mostrando variação da tempo em relação a número de vértices.

visão de como o espaço utilizado por esse algoritmo varia de forma linear com o numero de vértices da representação do labirinto. O primeiro foi obtido variando-se o numero de vértices e o segundo enquanto variava-se o numero de chaves máximo permitido. Como esperado pelo cálculo de teórico de complexidade assintótica de espaço, a relação entre memória utilizada e o numero de vértices é linear.

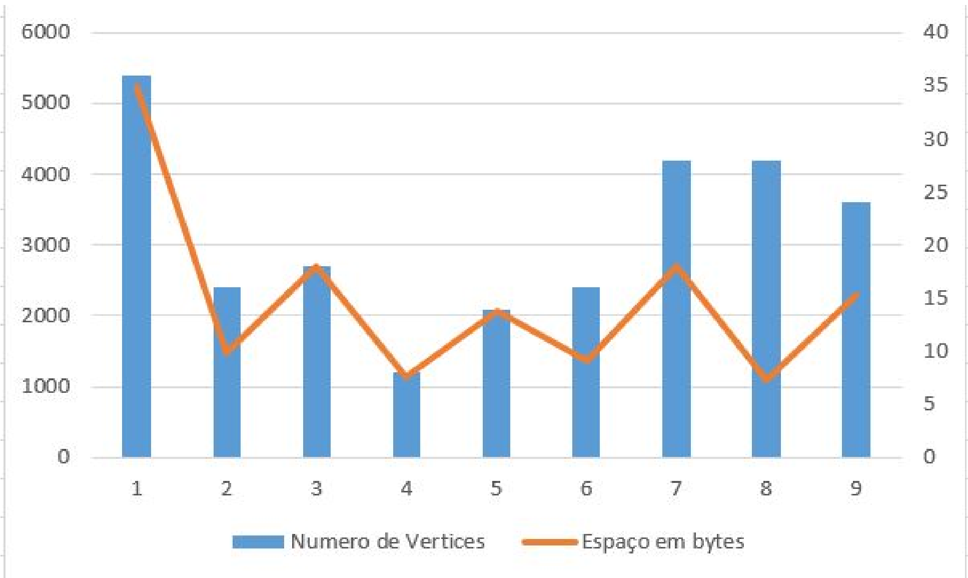

Figura 5: Gráfico mostrando variação da espaço alocado em bytes em relação ao numero de vértices

Ainda, durante a analise experimental foram feitos testes utilizando o Valgrind. Para os exemplos *TOYs* passados, apenas em dois o algoritmo não desalocava toda a memória alocada durante a execução do algoritmo. Em todos em que era possível o Vinicius conseguiu achar a saída do departamento de física.

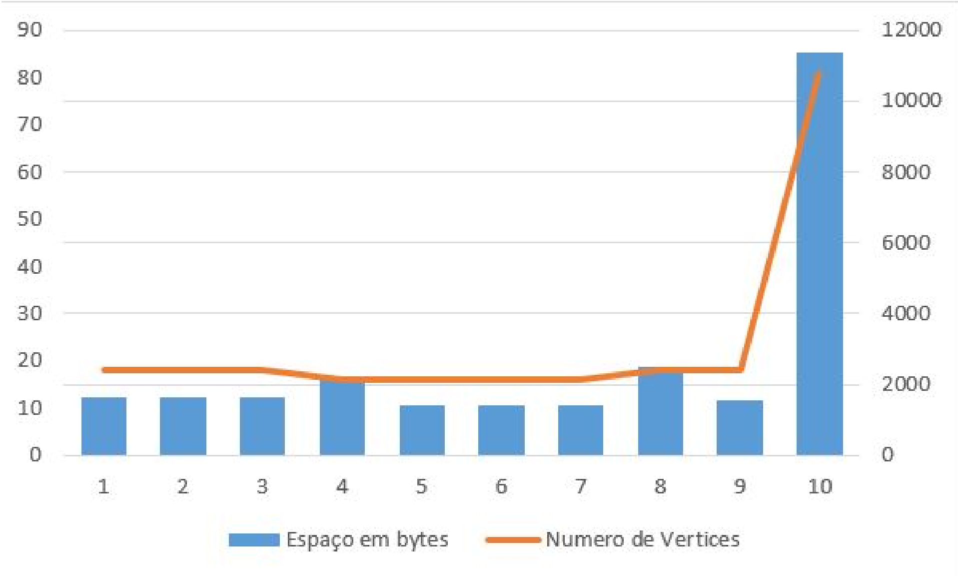

Figura 6: Gráfico mostrando variação de memoria usada em relação ao numero de vértices e chaves

5 Conclusões

Nesse trabalho foi implementada a solução para um problema de um labirinto. É um problema bem comum no estudo da teoria de grafos, porém com as adições de portas e chaves e buracos de minhoca, além do número de chaves como restrição o problema se tornou mais desafiador. As restrições mencionadas acima foram importantes porque descartavam aplicações de soluções banais para o problema, como a aplicação direta de algoritmos de melhor caminho. A implementação foi feita divida em partes, basicamente de iguais importacias. Primeiramente o a matriz de entrada,labirinto, foi lido e armazenado dentro de uma matriz para então ser utilizado na construção da lista de adjacências. O melhor caminho foi encotrado, ou não, utilizando um algoritmo baseado em busca em largura. O BFS pareceu interessante para solucionar esse problema pois os pesos eram iguais e ciclos não entrariam no escopo da solução, por não serem interessantes. Como pode ser observado a análise teórica de complexidade temporal e espacial se mostraram certos de acordo com os resultados mostrados na análise experimental.

6 Referências Bibliográgicas

Robert Sedgewick and Kevin Wayne. 2011. Algorithms (4th ed.). Addison-Wesley Professional.

Thomas T. Cormen, Charles E. Leiserson, and Ronald L. Rivest. 1990. Introduction to Algorithms. MIT Press, Cambridge, MA, USA.