Trabalho Prático 2 - Andando na Física

Murilo Vale Ferreira Menezes - 2013030996

Novembro de 2016

1 Introdução

Neste TP2, será tratado o problema de Vinícius, que se perdeu no Departamento de Física da UFMG e quer saber a melhor maneira de sair de lá. Para isso, é dado um mapa do departamento, consistindo de espaços livres, paredes, portas, chaves para estas portas e buracos de minhoca, que fazem Vinícius se teletransportar para outro ponto do mapa. Também são dadas aa posição da saída e a posição de início de Vinícius. As portas e chaves têm quatro tipos: c, d, h e s, de modo que as chaves c destrancam apenas as portas C, as chaves d destrancam as portas D e assim por diante.

São dadas como entradas, primeiramente, três inteiros: N e M, que são as dimensões do mapa, e C, o número máximo de chaves que Vinícius consegue carregar. Em seguida, são lidas N linhas e M colunas, representando o mapa. Posições livres são representadas por ".", paredes são representadas por "#", chaves são representadas por seu tipo, em letra minúscula (c, d, h ou s) e portas são representadas por seu tipo, em letra maiúscula (C, D, H ou S). Buracos de minhoca são representados pelas coordenadas do destino. A posição inicial do Vinícius é um caminho livre representado por "V"e a saída é um caminho livre representado por "E".

2 Descrição da Solução

O problema foi resolvido da seguinte maneira: O mapa da física, primeiramente recebido como entrada como um grid de caracteres NxM, é modelado como um grafo em lista de adjacências, de modo que cada posição do grid é um vértice, com arestas de peso 1 a todos os adjacentes no grid. No caso de buracos de minhoca, foi adicionada uma aresta de peso 0 para a posição de destino. Em cada vértice, existirão informações sobre se aquele vértice tem uma chave ou uma porta, e qual o seu tipo.

As chaves e portas foram modeladas com um sistema de máscara de bits: cada tipo de chave/porta representa uma máscara com determinado bit 1 e os

outros 0. Assim, com simples operações orientadas a bit de OR e AND, se pôde controlar o acesso às portas e a pegada de chaves.

Nos vértices também foram adicionadas informações sobre se aquele vértice é uma parede ou não (variável bloqueio), bem como se é um buraco de minhoca, e, se for, qual buraco é. Como são, no máximo, 5 buracos de minhoca, estes são enumerados na ordem de inserção. A identificação dos buracos funciona também com máscara de bits. Por fim, cada vértice tem cinco ponteiros para arestas, cada uma com o vértice de destino e seu peso. Um vértice nunca terá mais que cinco arestas, pois este pode ser adjacente aos quatro vértices adjacentes no grid e, no caso do buraco de minhoca, ter uma quinta aresta com peso zero para o destino do buraco.

Uma vez modelado o grafo representando o mapa, é executado neste grafo o algoritmo de caminho mais curto de Dijkstra, com algumas modificações dadas pelo contexto: os vértices adjacentes ao atual, quando explorados, são checados se são paredes ou portas sobre as quais não se possui chaves. Caso positivo, estas adjacências são ignoradas. Caso o vértice atual seja um buraco de minhoca ainda não utilizado, a única aresta a ser levada em consideração é a extra, com peso zero.

O programa foi separado em três partes fundamentais:

- A implementação da estrutura de dados Heap, que será usada como fila de prioridades para o Dijkstra.
- A implementação das funções necessárias para o funcionamento do programa no contexto dado, como funções utilizadas para modelar o grafo e conferir restrições.
- A função Main, que contém a leitura do grid e o fluxo de execução da modelagem do grafo, bem como o algoritmo implementado para encontrar o caminho mínimo.

A seguir, cada uma destas partes será explorada em separado.

2.1 Heap

Foi implementado um Heap Mínimo Binário para ser utilizado como fila de prioridades. Esta parte do código possui as seguintes funções:

- **parent**, que, dada uma posição no heap, retorna a posição do pai.
- **left**, que, dada uma posição no heap, retorna o filho à esquerda.
- **right**, que, dada uma posição no heap, retorna o filho à direita.

- **heapify**, que, dada uma posição no heap, a desloca até que esta satisfaça a propriedade de Heap.
- **buildHeap**, que recebe um array desordenado e, baseando-se na função heapify, o ordena para construir um heap.
- **tiraMin**, que coloca a primeira posição no fundo do Heap e decrementa a variável que determina seu tamanho.
- **encontraHeap**, que, dada uma posição de um vértice, encontra qual sua posição no Heap.

2.2 Utilitários

Esta parte do programa implementa funções necessárias para a boa execução do algoritmo proposto para o contexto, como funções de inicialização de vértices e arestas, de buracos de minhoca e funções para adicionar e conferir a existência de chaves. As funções são:

- **chaveNum**, que, ao receber o caracter de uma chave ou porta, retorna um inteiro para se fazer operações com a máscara de bits.
- **indice**, que, recebendo a posição no grid com coordenadas i e j, retorna o número do vértice no grafo que a representa.
- **novoVert**, que cria um vértice no grafo, com atributos de acordo com o caracter que o designa.
- **adicionaAresta**, que cria uma aresta comum (peso 1) para um dado vértice, dado o destino.
- **todasArestas**, baseando-se na função anterior, esta função cria arestas para todos os quatro vértices adjacentes no grid para determinado vértice.
- **adicionaBuraco**, que cria uma aresta de peso 0 vinda do vértice com um buraco de minhoca, para o vértice com coordenadas dadas pelo buraco.
- **encontraChave**, que adiciona a chave encontrada na coleção do Vinícius. Esta função é executada a cada vértice explorado, de modo que seu caráter orientado a bits simplesmente não modifica a máscara para posições sem chave, ou seja, com a variável chave = 0.
- **conferePorta**, que recebe duas máscaras de bits: uma da porta que se deseja conferir e outra das chaves que Vinícius possui. Retorna 1 caso a posição da porta não seja acessível e 0 caso seja.

2.3 Main

Por fim, a função Main contém o fluxo de execução do programa. Primeiramente, são recebidos os parâmetros e o mapa, e, concomitantemente, é criado o grafo que modela o mapa. Em seguida, é criado o Heap com todos os vértices do grafo e distância infinita do vértice de início (no código, o infinito é representado por 100000), exceto o vértice de início, que tem distância 0. É executado então o algoritmo de caminho mínimo de Dijkstra para que se encontre o menor caminho do Vinícius à saída, dadas todas as restrições.

Um dos primeiros pontos a se ressaltar é como é tratada a restrição de número de chaves. Na verdade, Vinícius pega todas as chaves que aparecem. Porém, o limite é colocado em prática com o número de portas diferentes que são atravessadas. Assim, limitar em 2 as chaves que podem ser carregadas é equivalente a limitar em 2 as portas diferentes que podem ser atravessadas.

Outro ponto interessante é como é colocada em prática a ideia de que a posição de um buraco de minhoca pode ser acessada novamente, como um caminho livre comum. Ao se utilizar um buraco de minhoca, diferentemente de outros vértices, o vértice atual não é extraído do Heap, e sim dado um peso infinito e colocado novamente ao fundo do Heap. Assim, o vértice ainda poderá ser aproveitado.

Este algoritmo de menor caminho pode ser interpretado como se existisse um grafo diferente para cada combinação de chaves e buracos de minhoca, uma vez que existem posições que seu acesso depende destas variáveis.

3 Análise de Complexidade

3.1 Complexidade Espacial

Analisando-se o programa, percebe-se que temos as seguintes estruturas como maiores consumidores de memória: o grid, o Heap, o vetor de distâncias e o grafo em si. Todas estas estruturas possuem complexidade espacial O(N*M), sendo N e M as dimensões do mapa e N*M a quantidade de vértices do grafo. Assim, a complexidade espacial total do programa é O(N*M).

3.2 Complexidade Temporal

Nesta subseção, será analisada a complexidade temporal de cada parte do programa, terminando com a função **main**.

- **Heap:** nesta parte do programa, temos um Heap binário. A operação de obter o mínimo é feita em O(1), bem como as funções parent, left e right. A função heapify tem complexidade assintótica O(log(N*M)), sendo N*M o tamanho do Heap, uma vez que o Heap é uma estrutura de dados

balanceada, de forma que sua altura é log(N*M). A função buildHeap executa em O(N*M*log(N*M)), já que esta consiste apenas na execução de heapify em metade dos vértices.

- **Utilitários:** nesta parte, todas as funções executam em tempo constante, ou seja, O(1), já que consistem em operações bem definidas e independentes da entrada. Por exemplo, a função novoVertice inicializa um vértice com um custo fixo, a função indice faz apenas uma operação para retornar o índice do vértice e as funções conferePorta e encontraChave fazem apenas uma operação orientada a bits.

- **main:** a função main pode ser dividida em três etapas, que serão tratadas separadamente a seguir: inicialização do grafo, inicialização para o Dijkstra e o Dijkstra em si.

  - Na inicialização, são recebidos os valores da dimensão e máximo de chaves. Em seguida, são recebidos todas as N*M posições do grid. Em cada posição, é executada a função novoVert, bem como todasArestas, e, caso seja um buraco de minhoca, adicionaBuraco. Todas as funções citadas executam em tempo constante, e, como são executadas em cada posição do grid, esta parte do Main tem complexidade temporal O(N*M).
  - A inicialização das estruturas necessárias para o Dijkstra, ou seja, o Heap e o vetor de distâncias, executa em O(N*M) também, uma vez que é feita a inicialização destas estruturas iterativamente, para cada vértice.
  - O algoritmo de caminho mínimo de Dijkstra, quando executado utilizando-se um heap binário, tem complexidade O(A*log(V)), sendo A o número de arestas e V o número de vértices do grafo. Neste caso, as arestas são limitadas superiormente em 5*V, assim, a complexidade pode ser escrita como O(V*log(V)). Como explicado anteriormente, devido às restrições de chaves e buracos, existem vários estados possíveis nos quais o grafo pode estar. Existem $2^{nchaves}$ estados dependentes do número de chaves e $2^{nburacos}$ estados dependentes do número de buracos. Assim, pode-se perceber que existem $2^{nchaves} * 2^{nburacos}$ grafos distintos. Desta forma, o número total de vértices não é somente N*M, e sim $N * M * 2^{nchaves} * 2^{nburacos}$. Desta forma, o algoritmo executa em $O(N * M * 2^{nchaves} * 2^{nburacos} * log(N * M * 2^{nchaves} * 2^{nburacos}))$.

Podemos perceber, então, que a complexidade temporal da função como um todo pode ser dada por $O(N * M * 2^{nchaves} * 2^{nburacos} * log(N * M * 2^{nchaves} * 2^{nburacos}))$. Vemos que este problema executa em tempo exponencial.

4 Avaliação Experimental

Para se analisar experimentalmente os resultados obtidos, foram utilizadas entradas de diferentes tamanhos.

4.1 Análise Espacial

Para a análise espacial, foi utilizada a função mallinfo(), da biblioteca malloc.h, que retorna qual a memória alocada em determinada parte do programa. Os dados foram coletados logo antes da fase de liberação da memória. Foram utilizadas entradas com número de vértices iguais a: 2, 8, 16, 30, 50 e 80. Os resultados obtidos foram plotados em um gráfico, tendo como resultado a Figura 1.

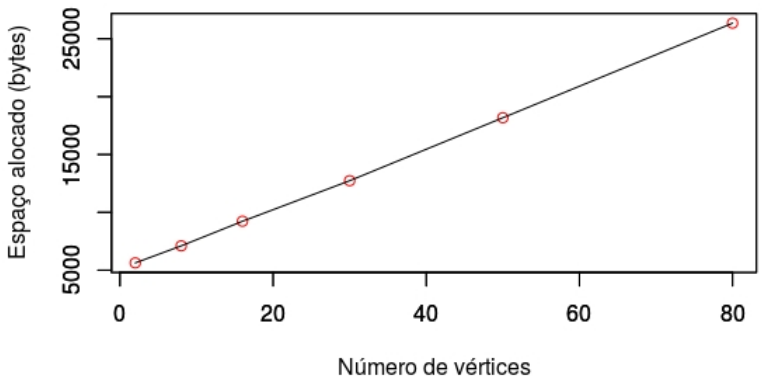

Figura 1: Gráfico gerado pela análise da complexidade espacial

Como podemos perceber pelo gráfico, o crescimento da memória utilizada é realmente linearmente dependente de N*M.

4.2 Análise Temporal

Para analisar o crescimento do tempo de execução com o tamanho das entradas, será marcado o número de ciclos de clock para a execução do programa com a função clock(), da biblioteca time.h. Primeiramente, será executado o programa variando-se apenas o número de vértices N*M. Cada teste é rodado 30 vezes e a média será levada em conta para a análise.

Foram feitos testes com 2, 8, 16, 30, 50 e 80 vértices.

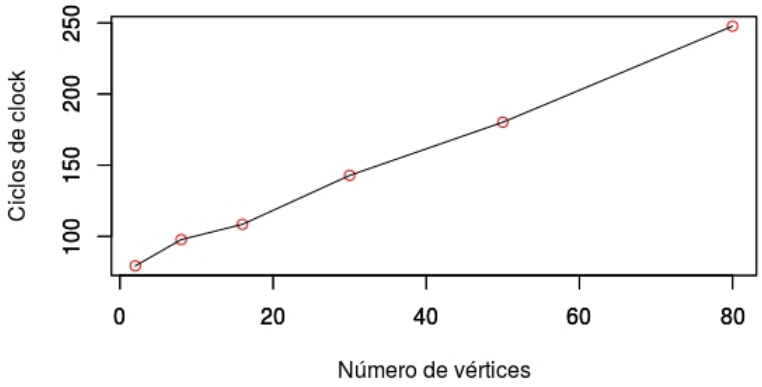

Figura 2: Gráfico gerado pela análise da complexidade temporal variando-se o número de vértices

Foi executado também o programa com número fixo de vértices, variando-se o número de chaves e o buracos de minhoca. Porém, os limites destes parâmetros são muito baixos (4 e 5, respectivamente). Desta forma, seu aumento não trouxe uma mudança sensível nos tempos de execução. Por exemplo, executando-se 30 vezes para cada número possível de buraco de minhoca, foi obtido o seguinte gráfico:

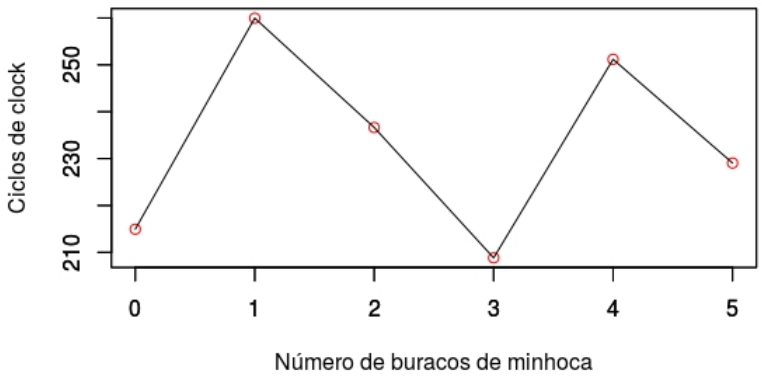

Figura 3: Gráfico gerado pela análise da complexidade temporal variando-se o número de vértices

Vemos que a variação em ciclos de clock foi muito baixa entre todos os pontos, além de não ser possível enxergar um padrão na curva.

5 Conclusão

O trabalho foi muito útil para exercitar os conhecimentos adquiridos referentes a grafos, bem como exercitar a solução de problemas complexos.

As análises experimentais de complexidade, de um modo geral, foram condizentes com o que se esperava. Porém, para os limites dados, o comportamento exponencial do algoritmo de caminho mínimo não fica muito claro, de forma que o número de vértices N*M exerce influência muito maior sobre o tempo de execução.