Trabalho Prático da disciplina de Algoritmos 1

Leticia Ribeiro Miranda
Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brazil
leticia-ribeiro98@hotmail.com

1. Introdução

Este trabalho prático aborda o problema de movimentação em um festival durante uma tempestade: Ana e Bernardo precisam se encontrar utilizando apenas os abrigos disponíveis, representados por círculos que formam uma rede de conexões. O objetivo é modelar essa situação como um grafo e encontrar o menor número de abrigos a serem atravessados para um reencontro entre eles. Além disso, a organização gostaria de saber qual é o maior número de abrigos que uma pessoa qualquer conseguiria atravessar no festival e a identificação dos abrigos críticos para a conectividade do local, visando o planejamento de novas edições futuras.

Para resolver o problema, inicialmente foi feita uma análise dos algoritmos adequados ao projeto. Em seguida, foram utilizados os arquivos 'grafo.cpp' e 'main.cpp', que reúnem todas as funções necessárias para o funcionamento do programa. Por fim, realizou-se a análise de complexidade da solução implementada.

2. Modelagem

2.1. Ambiente de Desenvolvimento e Execução

O programa foi desenvolvido na linguagem C++, compilado pelo compilador GCC da GNU Compiler Collection. O Computador utilizado tem as seguintes especificações:

- Sistema operacional: Ubuntu 24.04.1 LTS
- Compilador: gcc (Ubuntu 13.2.0-23 ubuntu 4) 13.2.0
- Processador: AMD Ryzen 5 3500U with Radeon Vega Mobile Gfx 2.10 GHz
- RAM: 16Gb (utilizável: 13,9 GB)

2.2. Estrutura de Dados

O programa está organizado em dois arquivos-fonte (main.cpp e grafo.cpp) e um arquivo de cabeçalho (grafo.h). Todas as funções responsáveis pela lógica e pelas operações sobre o grafo estão implementadas em grafo.cpp e são chamadas a partir de main.cpp.

Como o problema envolve deslocamento entre regiões circulares, o primeiro passo foi buscar uma forma de transformar os dados espaciais dos abrigos em uma estrutura de grafo. Inicialmente, a construção foi feita utilizando uma verificação direta de interseção entre todos os pares de círculos, resultando em uma complexidade de tempo $O(N^2)$. Essa abordagem utilizava uma matriz de adjacência implícita por meio de uma lista de adjacência estática (com os vetores grafo[i][j] e grau[i]), onde dois laços aninhados testavam todas as combinações possíveis de abrigos para construir as conexões, o que se mostrou inviável para grandes entradas.

Para otimizar esse processo, foi adotada uma abordagem baseada em indexação espacial via grid. Nessa técnica, o plano é dividido em células quadradas de tamanho proporcional ao maior raio dos abrigos. Cada abrigo é então alocado na célula correspondente e a verificação de interseção é feita apenas com os abrigos presentes na mesma célula ou em células vizinhas, reduzindo o número de comparações necessárias. Com essa estrutura, a complexidade média da construção do grafo cai para O(N) ou O(N log N), dependendo da distribuição espacial dos abrigos, embora o pior caso continue sendo $O(N^2)$.

Por fim, para resolver as três partes do problema, foram utilizadas as buscas em largura (BFS) e em profundidade (DFS). A BFS foi aplicada nas duas primeiras etapas, pois, em grafos não ponderados, ela garante o menor caminho em termos de número de vértices visitados. Na primeira parte, isso permitiu encontrar a rota mais curta entre Ana e Bernardo e na segunda, ao ser aplicada a partir de cada abrigo, possibilitou determinar o maior entre todos os caminhos mínimos possíveis. Já na terceira parte, a DFS foi empregada por ser a abordagem clássica para identificar pontos de articulação, ou seja, abrigos cuja remoção comprometeria a conectividade do grafo.

3. Solução

No algoritmo, a construção do grafo está implementada na função “construirGrafoComGrid”, localizada no arquivo grafo.cpp e é chamada a partir do arquivo main.cpp, durante a fase de inicialização. Ela utiliza os vetores r, x e y para armazenar o raio e as coordenadas de cada abrigo, além de uma estrutura tridimensional “celula” para agrupá-los nas respectivas células do grid. Em seguida, a função percorre as células vizinhas de cada abrigo, verificando interseções apenas dentro desse subconjunto. A construção do grafo ocorre de forma implícita: para cada abrigo, a função percorre as células ao redor de sua posição no grid e, ao encontrar um outro abrigo que o intersecta, o adiciona à lista de adjacência e incrementa seu grau. Dessa forma, são criadas as conexões (arestas) entre os abrigos que se sobrepõem, formando um grafo não direcionado.

Com o grafo já construído, o próximo passo foi identificar em quais abrigos Ana e Bernardo estavam localizados. Para isso, foi utilizada a função “dentro”, implementada no arquivo grafo.cpp e chamada no main.cpp. Esse método verifica se as coordenadas de um ponto estão contidas dentro de um abrigo, considerando o raio e o centro do abrigo. O algoritmo percorre todos os abrigos e, ao encontrar aquele que contém Ana ou Bernardo, armazena seu índice nas variáveis “ana” e “bernardo”, respectivamente.

Diante do problema em que Ana e Bernardo desejam se encontrar atravessando o menor número possível de abrigos, aplicou-se uma busca em largura (BFS). Esse algoritmo foi implementado na função “bfsMinimo”, localizada no arquivo grafo.cpp e chamado a partir do main.cpp.

O algoritmo funciona utilizando uma fila para explorar os abrigos em ordem crescente de distância, iniciando pelo abrigo onde Ana se encontra. Em cada passo, o abrigo atual é removido da fila e todos os seus vizinhos são examinados. Se ainda não tiverem sido visitados, eles são marcados como visitados, têm sua distância atualizada, representando o número de abrigos atravessados até ali, e são inseridos na fila para futuras explorações. O processo continua até que o abrigo de Bernardo seja alcançado — nesse momento, a função retorna a menor distância encontrada. Caso não exista um trajeto possível entre ambos, o método retorna -1, indicando que o encontro é impossível.

O segundo problema busca determinar o maior número de abrigos que uma pessoa qualquer precisaria atravessar no festival. Para isso, utilizou-se o grafo já construído, mas com um novo objetivo: determinar a maior distância mínima entre dois abrigos que estejam conectados no grafo.

A resolução foi feita com a função “bfsMaximo”, localizada no arquivo grafo.cpp e chamada a partir do main.cpp. Ela aplica a busca em largura também, começando pelo abrigo de partida e explorando todos os vizinhos imediatamente conectados. Em seguida, ele continua explorando os vizinhos desses vizinhos e assim por diante, garantindo que todos os abrigos acessíveis a partir do ponto de partida sejam visitados de forma crescente em termos de distância. Durante o processo, é mantido um valor “maxDist”, que é atualizado sempre que uma nova distância maior é encontrada. Ao final da execução, “maxDist” representa o número máximo de abrigos que qualquer pessoa pode precisar atravessar durante o festival.

Por fim, para resolver o problema dos abrigos críticos — aqueles cuja remoção pode comprometer a conectividade entre diferentes áreas do festival — foi utilizado o conceito de pontos de articulação em grafos, que representam as vértices cuja remoção desconecta o grafo.

O método utilizado para identificar abrigos críticos é uma variação da busca em profundidade (DFS), baseada no algoritmo de Tarjan para detecção de pontos de articulação. Ele está implementado na função dfs, localizada no arquivo grafo.cpp e chamada em main.cpp. A partir de cada nó, a DFS calcula dois tempos fundamentais:

- tin[]: registra o tempo de chegada de cada vértice, ou seja, o momento em que ele foi visitado pela primeira vez na DFS;
- low[]: representa o menor tempo de descoberta (tin[]) que pode ser alcançado a partir de um determinado vértice, considerando tanto os filhos na DFS quanto qualquer aresta de retorno para um ancestral na árvore de busca.

Um vértice é identificado como ponto de articulação quando, ao visitar um de seus filhos “v”, verifica-se que “low[v] >= tin[u]”. Como “low[v]” representa o menor tempo alcançável de volta a um ancestral a partir de “v”, então se ele for maior ou igual ao “tin[u]”, isso quer dizer que “v” e seus descendentes estão "presos" abaixo de “u”. Além disso, trata-se separadamente do caso em que o vértice analisado é raiz da DFS, sendo ponto de articulação se possuir mais de um filho.

Após a identificação, é necessário exibir esses pontos em ordem crescente. Para isso, utilizou-se a função “ordenarCortes”, também presente em grafo.cpp, que aplica o algoritmo de ordenação bubble sort para organizar os índices antes da exibição.

4. Análise de Complexidade

Para fins de análise de complexidade, considera-se que N representa o número de vértices do grafo e M representa o número de arestas.

4.1. Parte 1

Complexidade de tempo: A BFS visita cada vértice uma vez e percorre todas as arestas do grafo. Portanto, a complexidade de tempo é O(N + M).

Complexidade de memória: Durante a execução, são utilizados uma fila para armazenar os vértices a serem processados, vetores auxiliares de distâncias e marcação de visitados que consomem O(N). Porém, o grafo é representado por listas de adjacência que consome O(N + M), logo, a complexidade de memória é O(N + M).

4.2. Parte 2

Complexidade de tempo: A BFS feita a partir de um vértice percorre O(N + M), como a função faz uma BFS para cada vértice, o custo total é O(N*(N + M).

Complexidade de memória: Temos o vetor dist e a fila que ocupam tamanho O(N), porém a estrutura do grafo ocupa O(N + M), logo, a complexidade de memória é O(N + M).

4.3. Parte 3

Complexidade de tempo: Cada vértice é visitado uma única vez e cada aresta é considerada no máximo duas vezes, resultando em uma complexidade base de O(N + M). No entanto, como a verificação para saber se um vértice já foi registrado como ponto de articulação é feita dentro da DFS, no pior caso, como em grafos onde quase todos os vértices são pontos de corte, essa verificação pode elevar a complexidade para $O(N^2 + M)$.

Para a ordenação dos vértices de corte, foi utilizado o algoritmo Bubble Sort, cuja complexidade é $O(K^2)$, onde K representa o número de vértices de corte encontrados.

Assim, considerando as etapas descritas, a complexidade de tempo total da parte 3 do algoritmo é dominada por $O(N^2 + M)$, que representa o maior custo entre as operações envolvidas.

Complexidade de memória: Todos os vetores utilizados no algoritmo ocupam O(N) de espaço, enquanto a representação do grafo consome O(N + M), considerando a estrutura de listas de adjacência. Portanto, a complexidade total de memória é O(N + M).

5. Considerações Finais

A realização deste trabalho prático proporcionou o estudo e a aplicação do algoritmo de buscas em largura (BFS) e profundidade (DFS) em situações reais, fortalecendo o entendimento de suas utilidades. Uma das facilidades deste trabalho foi compreender como o algoritmo funciona, já que havia um material didático disponibilizado no moodle da disciplina.

Por outro lado, as maiores dificuldades surgiram na construção tanto do grafo quanto dos algoritmos. Para a modelagem do grafo, foi necessário buscar materiais externos que explicassem como funcionavam os grids espaciais. Já na implementação dos algoritmos BFS e DFS, o desafio foi precisar escrever o código do zero, mesmo que guiada pela lógica dos métodos.

No geral, esta atividade permitiu a prática de modelagem com grafos e reforçou a importância da análise de complexidade na construção de soluções escaláveis.

6. Referências

ALMEIDA, Jussara M. (2025). Slides virtuais da disciplina de estruturas de dados. Disponibilizado via Moodle. Departamento de Ciência da Computação, Universidade Federal de Minas Gerais, Belo Horizonte.

DAVIS, Clodoveu. Slides virtuais sobre métodos de acesso, indexação espacial e tuning. Disponível em: https://homepages.dcc.ufmg.br/~clodoveu/files/Classes/BDG-09o%20Metodos%20de%20acesso,%20indexacao%20espacial%20e%20tuning.pdf. Acesso em: abr. 2025.

WIKIPÉDIA. Busca em largura. Disponível em: https://pt.wikipedia.org/wiki/Busca_em_largura. Acesso em: abr. 2025.

WIKIPÉDIA. Busca em profundidade. Disponível em: https://pt.wikipedia.org/wiki/Busca_em_profundidade. Acesso em: abr. 2025.

WIKIPÉDIA. Grid (spatial index). Disponível em: https://en.wikipedia.org/wiki/Grid_(spatial_index). Acesso em: abr. 2025.

GEEKSFORGEEKS. Articulation Points (or Cut Vertices) in a Graph. Disponível em: https://www.geeksforgeeks.org/articulation-points-or-cut-vertices-in-a-graph/. Acesso em: abr. 2025

NYSTROM, Robert. Spatial Partition — Game Programming Patterns. Disponível em: https://gameprogrammingpatterns.com/spatial-partition.html. Acesso em: abr. 2025.