Trabalho Prático 1 - Festival de Música

Luiz Felipe Gondim Araujo - 2023028188

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte - MG - Brasil

luizfelipegondimaraujo@ufmg.br

1 Introdução

Esta documentação apresenta a solução desenvolvida para atender à necessidade de garantir a mobilidade e segurança dos visitantes durante o festival de música. O sistema foi projetado para ajudar Ana e Bernardo, dois participantes, a se encontrarem sem se molhar, utilizando apenas a cobertura fornecida pelos abrigos distribuídos pelo festival. Com isso, a solução modela os abrigos como círculos e representa suas conexões através de grafos, permitindo determinar o número mínimo de abrigos a serem atravessados, o caminho ótimo de locomoção e os abrigos críticos cuja remoção poderia comprometer a acessibilidade do evento. Para isso, foram aplicadas estruturas de dados e algoritmos de grafos que garantem eficiência e robustez na análise dos deslocamentos e da estrutura do festival.

2 Implementação

O programa foi desenvolvido em linguagem C++, utilizando o compilador G++ da GNU Compiler Collection. Em relação ao ambiente de desenvolvimento, consistiu em uma máquina rodando o sistema operacional Linux Mint 21, equipada com 16 GB de memória RAM. A escolha do Linux Mint foi pensada visando a execução do código na avaliação efetuada pelos monitores. Além disso, ainda foram usados o valgrind e o gdb para capturar possíveis erros e vazamentos em relação ao uso de memória.

3 Modelagem

O problema foi modelado utilizando uma estrutura de grafo não direcionado, onde cada vértice representa um abrigo, e uma aresta entre dois vértices existe se suas áreas de cobertura se sobrepõem. Cada abrigo é caracterizado pelas suas coordenadas (x, y) e um raio r de cobertura. Através desses dados, a conexão entre dois abrigos foi determinada verificando se a distância euclidiana entre seus centros é menor ou igual à soma de seus raios.

Para representar o grafo, foi utilizada uma lista de adjacências (`connectionsList`), permitindo consultas rápidas às conexões. Além disso, filas foram utilizadas para a execução dos algoritmos de busca em largura (BFS) e profundidade (DFS). Também foi considerado se os pontos de partida (A_x, A_y) e de chegada (B_x, B_y) estavam localizados dentro da área de cobertura de algum abrigo, utilizando uma função de verificação de distância.

4 Solução

Para resolver os diferentes subproblemas do trabalho, foi desenvolvida a classe Graph que centraliza todas as soluções. Sabendo disso, temos os seguintes métodos:

4.1 Leitura da Entrada de Dados

O método `readInput` foi responsável por realizar a leitura e o processamento inicial dos dados de entrada. Ele interpretou as informações sobre as coordenadas de A e B, o número total de abrigos e os dados de cada abrigo (posição e raio de cobertura). A partir dessas informações, construiu-se uma estrutura de dados baseada em uma lista de adjacência, onde cada abrigo é representado como um vértice, e existe uma aresta entre dois abrigos se suas áreas de cobertura se sobrepõem. Essa representação do grafo permitiu modelar eficientemente as conexões entre abrigos, possibilitando a aplicação direta dos algoritmos de busca e análise de conectividade necessários para resolver o problema.

4.2 Parte 1: Caminho Mínimo Entre A e B

O método `minDistance` utilizou o algoritmo de busca em largura (BFS) para encontrar o menor número de abrigos necessários para ir de A até B. A ideia consiste em inicializar uma fila com todos os abrigos que contêm o ponto A, definindo para eles a distância inicial igual a zero. Em seguida, enquanto a fila não estiver vazia, o próximo abrigo é removido da fila e seus vizinhos são visitados. Para cada vizinho ainda não visitado ou que possa ter sua distância minimizada, atualizamos a distância como sendo a distância do abrigo atual mais um e o enfileiramos para continuar o processo. Ao final da execução, a resposta corresponde à menor distância registrada entre os abrigos que contêm o ponto B.

4.3 Parte 2: Maior Distância Entre Dois Abrigos

Através do método `maxDistance`, para calcular a maior distância entre quaisquer dois abrigos do grafo, foi realizado um BFS partindo de cada abrigo como ponto inicial. Para cada execução do BFS, percorremos todos os abrigos alcançáveis a partir daquele ponto inicial, atualizando e armazenando a maior distância atingida durante o percurso. Ao final, a maior das distâncias registradas entre todos os pares de abrigos é considerada como resposta.

4.4 Parte 3: Identificação de Pontos de Articulação

Por fim, o método `findArticulationPoints` implementou um algoritmo baseado em busca em profundidade (DFS) para identificar abrigos críticos, ou seja, pontos de articulação cuja remoção desconectaria o grafo. Durante a execução da DFS, para cada vértice visitado, são registrados o tempo de entrada (timeIn) e o menor tempo de descoberta alcançável (low). Para cada vizinho de um vértice, se o vizinho ainda não foi visitado, é feita uma chamada recursiva da DFS para ele, atualizando os valores de low de acordo com os resultados. Se o valor de low de um vizinho é maior ou igual ao tempo de entrada do vértice atual, e o vértice não é a raiz da DFS, ele é marcado como ponto de articulação. Caso o vértice atual seja a raiz da DFS, é considerado ponto de articulação apenas se tiver mais de um filho na árvore de recursão.

5 Instruções de Compilação e Execução

Para executar o software, efetue os seguintes passos:

- Acesse a raiz do projeto;
- Utilizando um terminal, execute o comando 'make' para compilar o código;
- Ainda na raiz do projeto, digite: ./tp1.out;

6 Análise de Complexidade

A análise de complexidade será feita, principalmente, por métodos da classe Graph, uma vez que estas permitem uma visão geral da complexidade que o sistema carrega.

6.1 readInput

A complexidade da função é dominada pela construção da lista de conexões. A leitura dos n abrigos e a identificação dos abrigos de partida e chegada ocorrem em $O(n)$. Já a verificação de conexões entre todos os pares de abrigos exige $O(n^2)$ chamadas à função `areConnected`, cada uma em tempo $O(1)$. Portanto, a complexidade total de `readInput` é $O(n^2)$. O espaço utilizado é $O(n^2)$ para armazenar a lista de adjacências (`connectionsList`), além de vetores auxiliares de tamanho $O(n)$ (`shelters`, `visited`, `timeIn`, `low` e `isArticulation`). Assim, a complexidade final de tempo e espaço é $O(n^2)$.

6.2 minDistance

A função `minDistance` executa uma busca em largura (BFS) sobre o grafo de n abrigos. Cada aresta é processada no máximo uma vez, resultando em complexidade temporal $O(n + m)$, onde m é o número de conexões. O uso de memória é dominado pelo vetor `shelterMinSteps` de tamanho $O(n)$ e pela fila `queue` que no pior caso armazena até $O(n)$ elementos. Assim, a complexidade de espaço é $O(n)$.

6.3 maxDistance

A complexidade da função `maxDistance` também é dominada pelo BFS iniciado a partir de cada um dos n abrigos. Cada BFS percorre no máximo $O(n + m)$ arestas, onde m é o número de conexões no grafo. Assim, a complexidade total é $O(n(n + m))$. No pior caso, em que o grafo é completo ($m = O(n^2)$), a complexidade é $O(n^3)$, porém é algo difícil de se acontecer pois seria necessário que todos os abrigos estivessem conectados entre si. O espaço utilizado é $O(n)$ para o vetor `visitedLocal` e a fila de `BFS`, além do espaço já ocupado pela lista de conexões.

6.4 findArticulationPoints e dfs

A complexidade da função `findArticulationPoints` é dominada pela execução do método `dfs` em cada componente conexo. Como o `dfs` percorre todas as arestas e vértices uma única vez, a complexidade total é $O(n + m)$, onde n é o número de abrigos e m é o número de conexões. O espaço utilizado é $O(n)$ para armazenar os abrigos críticos e os vetores auxiliares já inicializados anteriormente.

6.5 Análise geral

De maneira geral, a complexidade de tempo e espaço do sistema depende do número de abrigos n e do número de conexões m entre eles. Sendo assim, a maioria dos algoritmos utiliza buscas em largura ou profundidade, garantindo custos lineares ou quase lineares em relação ao tamanho do grafo, exceto na função de cálculo da maior distância, que no pior caso atinge complexidade cúbica. O espaço é majoritariamente dominado pela lista de conexões e vetores auxiliares, totalizando, no pior caso, uma ocupação de $O(n^2)$.

7 Considerações Finais

Este trabalho lidou com o problema relacionado à mobilidade no das pessoas no festival de música, mesmo sob condições adversas. A modelagem do problema através de grafos permitiu representar de maneira precisa as conexões entre os abrigos, enquanto a utilização de algoritmos de busca em largura (BFS) e profundidade (DFS) assegurou a eficiência na resolução dos subproblemas apresentados.

Ademais, com a solução adotada, foi possível verificar a eficiência desses algoritmos em diferentes cenários e avaliar a sua complexidade, o que contribuiu para um maior entendimento em relação ao funcionamento dos mesmos. Foi interessante notar que, apesar de BFS e DFS serem algoritmos performáticos na maioria dos casos, no problema em questão eles podem apresentar comportamento cúbico no pior caso, o que evidencia a importância de analisar cuidadosamente a estrutura do problema ao escolher a abordagem mais adequada.

Por fim, durante a implementação da solução, houveram importantes desafios a serem superados, por exemplo, como seria feito o cálculo de sobreposição dos abrigos e a representação de suas conexões, além da escolha dos algoritmos para resolver cada subproblema. Apesar de serem desafios, auxiliaram bastante para o aprendizado do tema.

Referências

[1] HACKEREARTH. Breadth First Search. Disponível em: `https://www.hackerearth.com/practice/algorithms/graphs/breadth-first-search/tutorial/`. Acesso em: 18 abr. 2025.

[2] USP. Depth-First Search (DFS). Disponível em: `https://www.ime.usp.br/~pf/algoritmos_para_grafos/aulas/dfs.html`. Acesso em: 18 abr. 2025.

[3] USP. Estruturas de Dados para Representação de Grafos. Disponível em: `https://www.ime.usp.br/~pf/algoritmos_para_grafos/aulas/graphdatastructs.html`. Acesso em: 18 abr. 2025.