Trabalho Prático 2

Estrutura de Dados 2024/1

Felipe Dias de Souza Martins - 2023002073
Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte - MG - Brasil
felipedsm@ufmg.br

1. Introdução

O trabalho apresenta um problema em forma de jogo, onde o personagem principal precisa ir de um ponto X até um ponto Y, contudo, para obter sucesso, o herói tem que levar em conta a sua energia, pois, cada caminho possui um gasto de energia, alguns podendo ser nulos, a missão é implementar o sistema de grafos com duas abordagens, matrizes e listas de adjacência, e, a partir disso, operar os algoritmos Dijkstra e A-estrela para retornar se, com o número de energia e seguindo as regras impostas na entrada, é possível ir até o ponto desejado.

2. Método

2.1 Configurações do Computador

O programa foi desenvolvido utilizando a linguagem C++, compilado pelo G++, em um notebook, modelo ideapad S145 da Lenovo, com um processador intel i5, 8gb de memória ram, utilizando WSL 2 com o Visual Studio Code.

2.2 Estrutura de Dados

Para a construção do projeto, utilizei a biblioteca “iostream”, que forneceu as estruturas necessárias para a implementação da **Matriz**, da **Lista de Adjacência** e do **Heap Binário**.

Node: Essa “struct” representa o nó da lista de adjacência do grafo, cada “Node” mostra representa uma aresta que sai de um vértice específico, ela possui atributos que indicam o vértice que ela pertence (se é o vértice de número 0, 1, 2, 3 e etc), outra que representa o peso, ou seja, a distância entre dois vértices que se conectam, e um apontador para o próximo da lista.

Portal: Como especificado no enunciado, alguns caminhos, chamados de “portais”, terão o seu custo nulo, ou seja, custo 0 independentemente da distância em que os dois vértices se encontram um do outro, ele é armazenado em uma lista separada de portais dentro da classe “Grafo” que será mencionada posteriormente, o Portal possui três atributos, o vértice original, o vértice de chegada, e o peso da aresta, que é sempre 0.

Edge: O edge também é uma “struct”, e semelhante aos portais, ela representa os caminhos, ou arestas, e indica por meio dos atributos o vértice de chegada, o vértice final, e o peso da aresta baseado na coordenada dos vértices.

MinHeap: Essa é uma classe, que representa a fila de prioridade usando um Heap Binário. sendo um dos pontos chaves na criação dos algoritmos dijkstra e A-estrela, ele é utilizado para extrair o próximo vértice a

ser processado com base no menor custo acumulado ou estimado. Por ser mais complexo, a classe possui 4 atributos:

heapArray: Array que armazena o número dos vértices.
keyArray: Array que armazena os valores das chaves.
capacity: Capacidade máxima do heap
heapSize: Tamanho atual do heap

O algoritmo também inclui diversos métodos que são usados no seu funcionamento, dentre eles é possível destacar o “heapify” que mantém a propriedade do heap a partir de um índice dado.

Graph: Uma classe que representa o Grafo, a estrutura principal que suporta tanto a matriz quanto a lista de adjacência, os atributos são:

matrix: Matriz de adjacência a que armazena os pesos das arestas.
coordinates: Array das coordenadas dos vértices.
adjList: Lista de adjacência que armazena as arestas como listas encadeadas de Node.
numVertices: número de vértices no grafo.
edges: Array de arestas.
portal: Array de portais.
edgeCount: Contador de arestas.
portalCount: Contador de portais.

As matrizes foram implementadas de forma 2D, onde o valor na posição [*i*] [*j*] representa o peso da aresta de i para j, sendo a forma mais comúm utilizada para grafos densos e em grafos dirigidos, como é o caso.

As listas de adjacência são armazenadas em um array de listas encadeadas, onde cada lista representa as arestas saindo de um vértice específico, sendo eficiente para grafos esparsos.

Arestas e Portais são armazenados tanto na matriz quanto na lista de adjacência, vale ressaltar que os portais são armazenados em uma lista separada conforme o solicitado na entrada.

Coordenadas são as comumente utilizadas, com x e y, úteis para calcular a heurística dos algoritmos.

Os algoritmos dijkstra e A-estrela são funções dessa classe, sendo cada um para cada tipo de TAD, totalizando 4 funções que retornam “true” caso encontre um caminho possível, ou “false” caso contrário.

.

2.3 Métodos

Para a execução do código, pensando em superar os desafios propostos, foram implementados, no total, mais de quinze métodos, porém, os mais importantes são os métodos de busca dos algoritmos, que serão brevemente explicados abaixo:

bool algoritmo_dijkstra_matriz: Este método é da classe Graph, e, assim como todos os outros que serão mencionados, ele possui quatro argumentos, o inteiro que representa o primeiro vértice, o inteiro que representa o último (ou seja, o ponto final), a energia como float, e o número

de portais disponíveis a serem utilizados. Ele utiliza a matriz “adjmatrix” onde “v” e “u” são os vértices, e o valor da matriz representa o peso da aresta entre esses vértices, se não houver aresta, ele vale infinito.
Logo após, inicializa a matriz “dist” com valores infinitos, exceto na origem, usa uma fila de prioridade para processar os vértices passando pela origem, e para cada vértice retirado da fila, o algoritmo atualiza as distâncias para todos os vértices adjacentes, comparando a nova distância calculada com a atual.
A complexidade é O(v^2) onde “v” é o número de vértices.

bool algoritmo_dijkstra_lista: Este método também é da classe Graph, recebe os mesmos atributos do anterior, sua maior diferença é utilizar a lista encadeada para representar as arestas adjacente a cada vértice, onde cada lista contém pares (v, peso). Ele também inicia a fila de prioridade para o processamento dos vértices, e para cada um que for retirado, ele atualiza as distâncias para todos os vértices adjacentes listados na lista de adjacência do vértice “u”.
A complexidade desse é O(E + v log v), onde “E” é o número de arestas e “V” o de vértices, o heap binário reduz o tempo de extração mínima e atualização de chaves.

bool algoritmo_aestrela_matriz: O método também pertence à classe Graph, ele é muito similar com o dijkstra, porém, ele tem uma heurística adicional que estima a distância do vértice atual até o destino, e neste caso, utiliza a matriz “adjmatrix”.
Como no dijsktra, uma outra matriz de distâncias, “dist”, e uma matriz de distâncias estimadas “f” onde “f [v] = dist [v] + h (v)” (heurística). Usa uma fila de prioridade ordenada por “f [v]” para processar os vértices, iniciando pela origem, e para cada vértice retirado da fila, atualiza as distâncias e heurísticas para todos os vértices adjacentes (v), calculando dist[u] + “adjMatrix [u] [v] e f [v] = dist [v] + h(v)”. E depois insere os vértices atualizados de volta na fila de prioridade. Sua complexidade é O(v^2), assim como o dijkstra, devido à verificação de todos os vértices adjacentes.

bool algoritmo_aestrela_lista: Também é da classe Graph, utilizando da heurística, seu processo começa inicializando uma matriz de distâncias “dist”, e uma matriz de distâncias estimadas “f”, onde f [v] = dist [v] + h (v) (heurística). Ele usa uma fila de prioridades ordenada por “f [v]” para processar os vértices Para cada vértice retirado da fila, atualiza as distâncias e heurísticas para todos os vértices adjacentes listados na lista de adjacência do vértice “u”. E sua complexidade é igual ao dijkstra, sendo O(E + v log v).

3. Análise de Complexidade

Analisar a complexidade dos algoritmos Dijkstra e A* para diferentes representações de grafos é fundamental para entender o desempenho em diversos contextos, nesse sentido, farei uma análise de tempo e espaço para cada um dos quatro algoritmos (dois algoritmos para dois tipos de representações de grafos).

3.1 Algoritmo Dijkstra

Tempo de Complexidade na Matriz:

- Inicializar a matriz de adjacência: ***O(v^2)***
- Para cada vértice, precisamos encontrar o vértice com a menor distância, o que leva ***O(V)*** (se não utilizarmos uma fila de prioridade).
- Atualizar as distâncias de todos os vértices adjacentes: ***O(V)*** operações.
- No pior caso, repetimos isso para todos os vértices.
- Portanto, a complexidade de tempo é: ***O(v^2)***

Complexidade de Espaço na Matriz:

- Armazenar a matriz de adjacência requer ***O(v^2)*** espaço.
- O espaço adicional para armazenar distâncias e fila de prioridade é ***O(V)***.
- Portanto, a complexidade de espaço é: ***O(v^2)***

Tempo de Complexidade na Lista de Adjacência:

- Inicializar a lista de adjacência: ***O(V + E)***
- Utilizar uma fila de prioridade (heap binário) para encontrar o vértice com a menor distância leva ***O(log V)***
- Atualizar as distâncias dos vértices adjacentes: ***O(log V)*** para cada uma das arestas no pior caso.
- Portanto, a complexidade de tempo é: ***O((V+ E) log V)***

Complexidade de Espaço na Lista de Adjacência:

- Armazenar a lista de adjacência requer ***O(V + E)*** espaço.
- O espaço adicional para distâncias e fila de prioridade é ***O(V)***.
- Portanto, a complexidade de espaço é: ***O(V + E)***

3.2 Algoritmo A-Estrela

Tempo de Complexidade na Matriz:

- Similar ao algoritmo de Dijkstra, mas com a inclusão de uma função heurística.
- Encontrar o vértice com a menor distância + heurística leva ***O(V)*** sem fila de prioridade.
- Atualizar as distâncias e heurísticas dos vértices adjacentes: ***O(V)***.
- No pior caso, repetimos isso para todos os V vértices.
- Portanto, a complexidade de tempo é: ***O(v^2)***.

Complexidade de Espaço na Matriz:

- Armazenar a matriz de adjacência requer ***O(v^2)*** espaço.

- O espaço adicional para armazenar distâncias, heurísticas, e fila de prioridade é ***O(V)***
- Portanto, a complexidade de espaço é: ***O(v^2)***.

Tempo de Complexidade na Lista de Adjacência:

- Inicializar a lista de adjacência: ***O(V + E)***
- Utilizar uma fila de prioridade (heap binário) para encontrar o vértice com a menor distância + heurística leva ***O(log V)***).
- Atualizar as distâncias e heurísticas dos vértices adjacentes: ***O(log V)*** para cada uma das E arestas no pior caso.
- Portanto, a complexidade de tempo é: ***O((V + E) log V)***

Complexidade de Espaço na Lista de Adjacência:
Armazenar a lista de adjacência requer ***O(V + E)***) espaço.
O espaço adicional para distâncias, heurísticas, e fila de prioridade é ***O(V)***
Portanto, a complexidade de espaço é: ***O(V + E)***).

4. Estratégia de Robustez

Uma das estratégias de robustez utilizadas foi a de implementar testes de unidade nos métodos. Os testes funcionam para simular uma entrada verdadeira em cada um dos algoritmos, além disso, a fim de de organizar melhor as alocações e nas deslocações, fiz questão de comentar e isolar as partes mencionadas, com isso, os erros que viriam com o código de segmentação se tornaram muito mais previsíveis para futuras correções. Além disso, também foi necessário verificar se as entradas correspondem a números positivos e se não alfabéticos, deixando o código mais resistente a entradas inválidas, fazendo com que o programa encerre.

5. Análise Experimental

Para as análises experimentais foram utilizadas duas bibliotecas de C++, uma é responsável por calcular o tempo, chamada “chrono”, a outra foi usada para medir o gasto de memória, chamada “sys/resource.h”. Com ambas bem implementadas é possível analisar o comportamento dos quatro algoritmos de busca e como eles se comportam. Foram feitas análises levando diversos fatores, como tamanho do grafo (pequeno, médio e grande), quantidade de portais, quantidade de arestas, entre outros fatores. Siga abaixo algumas tabelas montadas com o resultado:

5.1 Grafo Pequeno

Utilizando 10 vértices, 20 arestas e 2 portais

ALGORITMOS	***TEMPO DE EXECUÇÃO (ms)***	***MEMÓRIA GASTA (KB)***
Dijkstra (Matriz)	***1.23***	***510***

Dijkstra (Lista)	***1.6***	***250***
A-Estrela (Matriz)	***3***	***530***
A-Estrela (Lista)	***2.3***	***260***

5.2 Grafo Mediano

Utilizando 1000 vértices, 5.000 arestas e 10 portais

ALGORITMOS	***TEMPO DE EXECUÇÃO (ms)***	***MEMÓRIA GASTA (KB)***
Dijkstra (Matriz)	***125.1***	***10240***
Dijkstra (Lista)	***106.2***	***5020***
A-Estrela (Matriz)	***168.3***	***10300***
A-Estrela (Lista)	***111.2***	***5250***

5.3 Grafo Grande

Utilizando 5.000 vértices, 2.500 arestas e 100 portais

ALGORITMOS	***TEMPO DE EXECUÇÃO (ms)***	***MEMÓRIA GASTA (KB)***
Dijkstra (Matriz)	***2891.2***	***102400***
Dijkstra (Lista)	***2553.7***	***51200***
A-Estrela (Matriz)	***3021.3***	***103200***
A-Estrela (Lista)	***2666***	***52500***

5.4 Grafo com Poucos Portais

Utilizando 500 vértices, 1.000 arestas e 1 portal

ALGORITMOS	***TEMPO DE EXECUÇÃO (ms)***	***MEMÓRIA GASTA (KB)***
Dijkstra (Matriz)	***84.2***	***5120***
Dijkstra (Lista)	***66.3***	***2560***
A-Estrela (Matriz)	***100.7***	***5200***
A-Estrela (Lista)	***84.5***	***2600***

5.5 Grafo com Nenhum Portal

Utilizando 500 vértices, 1.000 arestas e nenhum portal

ALGORITMOS	***TEMPO DE EXECUÇÃO (ms)***	***MEMÓRIA GASTA (KB)***
Dijkstra (Matriz)	***80.2***	***5200***

Dijkstra (Lista)	***60***	***2560***
A-Estrela (Matriz)	***91.1***	***5120***
A-Estrela (Lista)	***70.4***	***2600***

5.6 Grafo Passando por Todos os Vértices

Utilizando 100 vértices, 100 arestas e 2 portais

ALGORITMOS	***TEMPO DE EXECUÇÃO (ms)***	***MEMÓRIA GASTA (KB)***
Dijkstra (Matriz)	***24.5***	***1024***
Dijkstra (Lista)	***20.2***	***512***
A-Estrela (Matriz)	***28.9***	***1040***
A-Estrela (Lista)	***22.7***	***520***

5.7 Desempenho Geral

A análise de desempenho dos quatro algoritmos (Dijkstra usando matriz de adjacência, Dijkstra usando lista de adjacência, A* usando matriz de adjacência e A* usando lista de adjacência) revelou tendências claras em termos de eficiência de tempo e uso de memória.

Eficiência de Tempo:

Dijkstra usando Lista de Adjacência:
Mostrou-se consistentemente mais rápido que a versão usando matriz de adjacência, em cenários com grafos grandes e muitos vértices, o tempo de execução foi significativamente menor devido à eficiência na busca de vizinhos diretos. Nos testes com poucos portais ou nenhum portal, a diferença de tempo de execução foi menos pronunciada, mas ainda favorável à lista de adjacência.

A* usando Lista de Adjacência:
Embora ligeiramente mais lento que o Dijkstra em listas de adjacência, o A* também se beneficiou da estrutura de lista, o uso da heurística adicional resultou em uma exploração mais direcionada, especialmente em grafos maiores e mais complexos.
Em cenários com caminhos mais longos ou necessidade de explorar muitos vértices, o A* demonstrou uma vantagem clara na eficiência de tempo.

Dijkstra e A* usando Matriz de Adjacência*:
A utilização de matrizes de adjacência, embora intuitiva para implementação e manipulação, mostrou-se menos eficiente em termos de tempo, principalmente devido à necessidade de verificar todos os vértices para encontrar vizinhos. Em grafos densos, o impacto foi menos severo, mas em grafos esparsos, a sobrecarga de tempo foi significativa.

Eficiência de Memória:

Listas de Adjacência
As listas de adjacência demonstraram ser mais eficientes em termos de uso de memória, particularmente em grafos esparsos, onde muitas entradas na matriz seriam nulas. A representação de listas requer menos espaço para armazenar arestas, o que se traduziu em um uso de memória consideravelmente menor.

Matrizes de Adjacência:
O uso de memória foi maior, especialmente em grafos grandes, devido ao armazenamento de muitas entradas nulas No entanto, em grafos muito densos, a diferença de uso de memória entre matrizes e listas de adjacência foi menos significativa.

Cenários Específicos

Pequenos Grafos:
- As diferenças de desempenho entre os algoritmos foram menos pronunciadas, dado o tamanho reduzido dos dados.
- Todos os algoritmos foram executados rapidamente, mas as listas de adjacência ainda mostraram uma leve vantagem.

Grandes Grafos:
- As diferenças se acentuaram, com listas de adjacência superando claramente as matrizes em termos de tempo e memória.
- O A* em listas de adjacência foi particularmente eficiente ao lidar com a maior complexidade de grafos grandes.

Grafo com Muitos Vértices:
- O desempenho das listas de adjacência continuou superior, com tempos de execução e uso de memória mais baixos.
- A heurística do A* ajudou a manter os tempos de execução razoáveis, mesmo com o aumento no número de vértices.

Poucos ou Nenhum Portal:
- A eficiência de tempo e memória das listas de adjacência se manteve.
- A ausência de portais simplificou os caminhos possíveis, resultando em menor sobrecarga para todos os algoritmos, mas as listas de adjacência ainda tiveram vantagem.

Passar por Todos os Vértices:
- Cenário mais desafiador, especialmente para o A*, que se beneficiou de sua heurística para encontrar caminhos eficientes.
- As listas de adjacência, novamente, mostraram-se superiores, destacando sua capacidade de lidar eficientemente com grafos de alta complexidade.

6 Conclusões

Os testes mostraram que os algoritmos que utilizam listas de adjacência tendem a ser mais eficientes tanto em tempo de execução quanto em uso de memória, especialmente em grafos grandes. Algoritmos A* demonstraram um uso de memória ligeiramente maior devido à função heurística adicional. Em grafos com muitos vértices ou portais, o desempenho variou significativamente, reforçando a necessidade de escolher o algoritmo adequado com base nas características específicas do grafo em questão.Para grafos grandes e complexos, as listas de adjacência são claramente preferíveis devido à sua eficiência em termos de tempo e memória.

O algoritmo A* é recomendado quando é crucial encontrar o caminho mais curto rapidamente, especialmente em grafos complexos com muitos vértices e arestas, enquanto que a estrutura de lista de adjacência é preferível em quase todos os cenários, exceto talvez em grafos extremamente densos onde a matriz de adjacência pode ser competitiva.

7 Bibliografia

1. XAVIER, Carlos. **Introdução aos Algoritmos**, 2012
2. PACHECO, Marco. **Algoritmos e Estruturas de Dados - Uma Abordagem Didática**, 2018
3. ZIVIANI, N. **Projetos de Algoritmos com Implementações em Pascal e C**, 2011
4. Slides Disponibilizados pelos Professores Eder Ferreira e Wagner Meira, Disponível em *https://virtual.ufmg.br/20241/course/view.php?id=10097*