Trabalho Prático 2

[Luis Antonio Duarte Sousa]

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

luisduarte.dev@gmail.com

1 Introdução

Neste trabalho prático, abordamos o problema de encontrar um caminho viável para o herói Linque escapar da floresta da neblina, representada por um grafo direcionado com n clareiras e m trilhas. A floresta possui trilhas unidirecionais que consomem energia proporcional à distância percorrida e portais mágicos que não consomem energia, mas possuem um uso limitado. Linque inicia sua jornada na clareira 0 e seu objetivo é alcançar a clareira n-1, utilizando no máximo s unidades de energia e atravessando no máximo k portais.

Para resolver este problema, implementamos duas abordagens clássicas para encontrar caminhos mínimos em grafos: o algoritmo de Dijkstra e o algoritmo A* (A-estrela). O algoritmo de Dijkstra é utilizado para grafos com pesos não negativos, enquanto o algoritmo A* é uma variação que incorpora uma heurística para melhorar a eficiência em encontrar o caminho mais curto.Este documento detalha a implementação das estruturas de dados necessárias, os algoritmos utilizados, bem como as estratégias adotadas para adaptar esses algoritmos às restrições específicas do problema. Além disso, são apresentadas análises de complexidade teóricas e experimentais, comparando a eficiência e a eficácia das implementações utilizando listas de adjacência e matrizes de adjacência. Por fim, discutimos os resultados obtidos, as dificuldades enfrentadas e as soluções adotadas para superá-las.

2 Implementação

2.1 Configurações do Sistema

- Sistema Operacional: Windows 11 pro
- Ambiente de execução: WSL2
- Linguagem: C++
- Compilador: GCC 13.2
- Processador: AMD Ryzen 5 735HS
- RAM: 24 GB

2.2 Definições

- const int infinity = 0x3f3f3f3f;

  Representa o infinito, ou seja, o máximo de distancia possivel entre vértices (63 em hexadecimal)

2.3 Utilitários Implementados

2.3.1 Funções Axiliares

- `swapp(T& a, T& b):`

  Troca elementos e objetivo de ser usado no heap.

- `pow(double n, int p):`

  Eleva n pela potência p.

- `euclid_dist(Pair<double,double> &u ...):`

  Calcula a distancia euclidiana entre dois pares de coordenadas x e y.

2.3.2 Estruturas de dados

- `Pair<T1,T2>:`

  A implementação do Pair se baseia em um template **(template <typename T1,typename T2 >)** que armazena dois tipos de dados juntos em um mesmo objeto e tem como objetivo permitir/facilitar a armazenagem das arestas e seu respectivo custo e possui como principais operações setters e getters para o primeiro e segundo elemento.

- `Min-Heap(Fila de prioridade):`

  Foi implementado um heap mínimo que é uma estrutura baseada em árvore binária completa, possibilitanto assim sua representação por meio de array onde para cada posição i de um nó

pai, seus filhos esquerdo e direito estarão localizados em **2*i + 1** e **2*i + 2** respectivamente . Além das propriedades de uma árvore completa um heap mínimo possui a característica que cada nó pai é menor que os filhos e consequentemente o nó raiz é o minimo da árvore, essa característica se mantem mesmo retirando ou colocando elementos devido aos métodos HeapfyUp e HeapfyDown.O heap foi adáptado para ordenar triplas **(Pair <Pair<int,double >, int >)** que representam nó alvo, custo da aresta até esse nó e numero de portais usados até respectivo nó. Além disso, possui como principais operações a inserção e remoção de minimo.

- `LinkedList<T>`:

  Lista simplesmente encadeada com intuito de armazenar as aréstas de vizinhas de cada vértice do grafo.Os principais métodos utilizados foram inserção de itens na ultima posição e recuperação do item de acordo com a posição desejada.

- `Adjacency_Matrix(n)`:

  Armazena as relações de vizinhança dos vértices usando uma matriz de dimensão nxn onde n é o numero de vértices do grafo e cada posição do **A[i,j]** é o custo da aresta que vai do vértice **i** para **j**.

- `Adjacency_List(n)`:

  Armazena as relações de vizinhança dos vértices usando um array de LinkedLists de dimensão n , onde n é o numero de vértices do grafo e cada posição **A[i]** é uma lista com pares **(Pair <int, double >)** que registram o vértice alvo da aresta e o custo dela e i é o vértice inicial.

2.4 Estratégia de Resolução

A resolução do problema é baseada em versões modificadas dos algoritmos e estrutura de dados, ajustada para lidar com uma restrição adicional de energia e um número limitado de portais. A seguir, descrevemos a estratégia utilizada:

Inicialmente, verificamos se o número de vértices (`adj_list.numberOfV`) é menor ou igual a 1, caso em que a função retorna `true` imediatamente, pois não há necessidade de realizar cálculos adicionais.

O algoritmo utiliza um `ModMinHeap` como fila de prioridade para gerenciar os vértices a serem explorados. Para cada vértice, mantemos um vetor `dist` que armazena a distância acumulada desde o vértice inicial e um vetor `visited` que marca se um vértice foi visitado.

Inicializamos as distâncias para todos os vértices com um valor infinito, exceto para o vértice inicial (0), cuja distância é definida como 0. O vértice inicial é enfileirado com uma distância de 0 e sem portais usados.

O algoritmo procede com a exploração dos vértices na fila de prioridade. Em cada iteração, retiramos o vértice com a menor distância acumulada. Se o vértice retirado for o vértice final (`adj_list.numberOfV - 1`), verificamos se a distância acumulada é menor ou igual à energia disponível. Se for o caso, retornamos `true`, indicando que é possível chegar ao vértice final dentro dos limites. Caso contrário, retornamos `false`.

Se o vértice já foi visitado, pulamos a iteração. Caso contrário, marcamos o vértice como visitado e exploramos seus vizinhos. Para cada vizinho, verificamos o peso da aresta. Se o peso for zero, verificamos se é possível usar mais portais (sem ultrapassar o limite de `portals`). Se for possível, atualizamos a distância para o vizinho e enfileiramos o vizinho com a distância atualizada (caso seja A* somamos a heurística) e o número de portais utilizados incrementado. Caso contrário, se o peso não for zero, atualizamos a distância e enfileiramos o vizinho com o número atual de portais.

Se a fila de prioridade ficar vazia e não tivermos encontrado um caminho válido, retornamos `false`, indicando que não é possível alcançar o vértice final dentro das restrições.

3 Instruções de Compilação e Execução

Este documento descreve como compilar e executar o projeto contido na pasta zipada. A estrutura do projeto é a seguinte:

- `include/` - Contém os arquivos de cabeçalho (`.h`).
- `src/` - Contém os arquivos de código-fonte (`.cpp`).
- `obj/` - Diretório para os arquivos objeto (`.o`) gerados durante a compilação.
- `bin/` - Diretório para o executável gerado.

Na raiz do projeto, há um arquivo `Makefile` que facilita a compilação, execução e limpeza do projeto. As funções principais do `Makefile` são:

- `make all` - Compila todos os arquivos `.cpp` loca-

lizados em `src/` e coloca os arquivos objeto (`.o`) em `obj/` e o executável em `bin/`.

- `make run` - Executa o arquivo executável que está em `bin/`.
- `make clean` - Remove todos os arquivos objeto (`.o`) de `obj/` e o executável de `bin/`.

Para compilar e executar o projeto, siga os seguintes passos:

1. Descompacte a pasta zipada em um diretório de sua escolha.
2. Abra um terminal e navegue até a raiz do diretório descompactado.
3. Execute o comando `make all` para compilar o projeto.

   ```
   make all
   ```

   Isso irá compilar os arquivos `.cpp` e gerar os arquivos objeto em `obj/` e o executável em `bin/`.
4. Para executar o programa, utilize o comando `make run`.

   ```
   make run
   ```

   Isso irá executar o arquivo executável que está em `bin/`.
5. Para limpar os arquivos objeto e o executável gerado, utilize o comando `make clean`.

   ```
   make clean
   ```

   Isso irá remover todos os arquivos em `obj/` e `bin/`.

Certifique-se de que você tenha o `make` instalado em seu sistema. Caso não tenha, você pode instalá-lo usando o gerenciador de pacotes de sua distribuição Linux ou baixar a versão apropriada para o seu sistema operacional.

Com essas instruções, você deve ser capaz de compilar e executar o projeto sem problemas. Em caso de dúvidas, consulte a documentação do `make` ou entre em contato com o autor do projeto.

4 Análise de Complexidade

O arquivo **main** do programa desenvolvido sempre recebe n vértices, m arestas e k portais pelo teclado, resultando em complexidades de entrada de $\theta(n)$, $\theta(m)$ e $\theta(k)$, respectivamente. Após várias operações $O(1)$, as funções de Dijkstra e A* são chamadas. Ambas possuem complexidades de tempo semelhantes, dependendo da escolha da representação do grafo. Usando uma lista de adjacência, a complexidade de tempo será $O((n+m)logn)$ e a complexidade de espaço será $O(n+m)$. Por outro lado, utilizando uma matriz de adjacência, a complexidade de tempo será $O(n^2)$ e a complexidade de espaço aumenta significativamente para $O(n^2)$, o que pode afetar negativamente a eficiência em termos de uso de memória e, potencialmente, o desempenho para grafos muito grandes e esparsos.

Para comprovar a complexidade de tempo do algoritmo usando Lista, observe que há um loop de busca por vizinhos na estrutura de adjacência com complexidade $O(n+m)$ (uma vez que depende de ambos os parâmetros nn vértices e mm arestas). Aninhado a isso, existe uma operação de retirada de elementos mínimos do heap com complexidade $O(logn)$. Portanto, a complexidade total é $O((n+m)logn)$.

Por outro lado, a diferença de complexidade de memória pode ser verificada pelo fato de que uma matriz de adjacência aloca memória para todos os seus n^2 elementos,enquanto a lista de adjacência aloca memória proporcional ao número m de arestas, evitando desperdício de memória. Isso faz com que a lista de adjacência seja mais eficiente em termos de uso de memória, especialmente para grafos esparsos onde $m<<n^2$.

Complexidade de tempo em Lista Adj.:

$$(O(n)+O(m)+O(k)+\alpha O(1)+O((n+m)logn)) \in O((n+m)logn)$$

Complexidade de tempo em Matriz Adj.:

$$(O(n)+O(m)+O(k)+\alpha O(1)+O(n^2) \in O(n^2)$$

Complexidade de espaço em Lista Adj.:

$$(O(n)+O(m)+O(k)+\alpha O(1)+O(n+m)) \in O(n+m)$$

Complexidade de espaço em Matriz Adj.:

$$(O(n)+O(m)+O(k)+\alpha O(1)+O(n^2)) \in O(n^2)$$

Observação:

A implementação utilizada foi um heap binário com complexidades de inserção e remoção de minimo, ambas, **O(logn)**. Entretanto, existe uma implementação do heap que utiliza a sequência de fibonacci e possui complexidade de inserção constante **O(1)**. E pode ser muito eficiente nos algoritmos implementados uma vez que inserções no heap são muito comuns.

5 Estratégias de Robustez

Os mecanismos de robustez incluem a verificação de limites de entrada, que assegura que as variáveis como o número de vértices, arestas e portais estejam dentro dos parâmetros esperados antes de prosseguir com as operações. A alocação segura de memória também foi uma preocupação central. Ao alocar memória para armazenar as coordenadas dos vértices, verificou-se se a alocação foi bem-sucedida, evitando falhas de segmentação que poderiam causar comportamentos imprevisíveis ou falhas. Além disso, utilizou-se manipulação de exceções em operações críticas, como a inserção e remoção de elementos do heap. Esse cuidado garante que o programa possa lidar com essas situações sem interrupções abruptas, permitindo uma resposta mais controlada a problemas inesperados. Antes de adicionar arestas e portais, foi feita uma validação dos dados de entrada para verificar se os vértices fornecidos eram válidos e estavam dentro do intervalo esperado. Esta validação ajuda a evitar acessos fora dos limites e possíveis erros de execução. Mensagens de erro claras e registros (logs) foram utilizados para ajudar na identificação e correção de problemas. As mensagens de erro fornecem feedback útil aos desenvolvedores e usuários sobre o que deu errado, facilitando a resolução de problemas. Além dos mecanismos de programação defensiva, testes extensivos foram realizados para garantir que o sistema se comporte conforme o esperado em diversos cenários, incluindo casos extremos e de erro. Os testes ajudam a identificar problemas potenciais antes que eles afetem os usuários finais. Essas estratégias de robustez asseguram que o sistema seja confiável e resiliente, proporcionando uma experiência de usuário mais estável e previsível. Implementando essas práticas, busca-se minimizar a ocorrência de falhas e garantir que, quando estas ocorrem, o sistema possa lidar com elas de maneira eficiente e eficaz.

6 Análise Experimental

Primeiramente, foram realizadas análises de complexidade experimentais. Os algoritmos implementados foram executados em várias instâncias de diferentes tamanhos para medir seu tempo de execução. A complexidade teórica foi comparada com os tempos observados para garantir a consistência entre teoria e prática. Observou-se que a complexidade teórica dos algoritmos foi confirmada pelos experimentos, validando as previsões de desempenho.

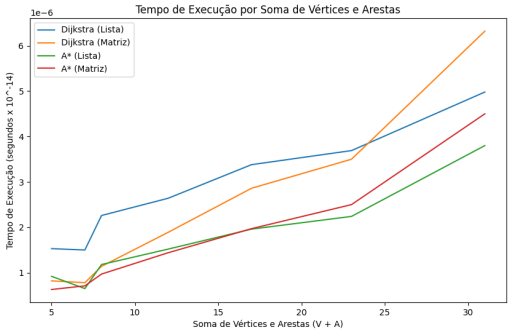

Figura 1: Análise de Tempo

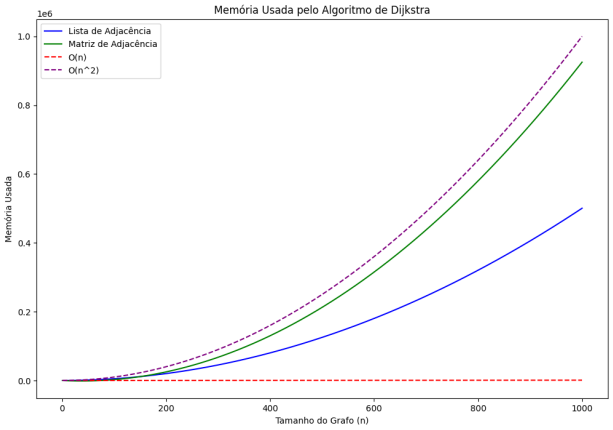

Figura 2: Análise de Espaço

Em relação à comparação das implementações de matriz e lista de adjacência, foram realizados testes para avaliar como cada estrutura impacta a execução dos algoritmos. A matriz de adjacência, devido ao seu uso de memória fixa, demonstrou melhor desempenho em grafos densos, onde o número de arestas é próximo ao quadrado do número de vértices porém pelo fato de fazer mais comparações tem um tempo prejudicado. Em contrapartida, a lista de adjacência mostrou-se mais eficiente em grafos esparsos, onde há um número significativamente menor de arestas.

Essa diferença ocorre porque a matriz de adjacência consome memória para todos os possíveis pares de vértices, independentemente de haver uma aresta entre eles, enquanto a lista de adjacência aloca memória apenas para as arestas existentes.

Além disso, analisou-se a qualidade das soluções obtidas pelos algoritmos. O Dijkstra que leva em consideração apenas o custo $f(x)$ do caminho, sempre encontrará o menor caminho (se existir), entretanto, o A* que leva em consideração a soma do custo $f(x)$ com uma heuristica $h(x)$ só encontrará o menor caminho caso a heuristica seja **admissivel**, isso é, matematicamente, se $h(n)$ é a função heurística para um nó n, e $h'(n)$ é o custo real do caminho mais curto do nó n até o objetivo, a heurística $h(n)$ é admissível se para todos os nós n na busca, $h(n) \leq h'(n)$.Dito isso, verificasse que a heurística selecionada para o exercicio não é admissível, uma vez que existem caminhos de distânia 0 que permitem que $h'(n)<h(n)$, logo seus resultados nem sempre serão iguais pois o A* não tera garantidamente solução ótima. Além disso, ao variar ou restringir alguns dos parâmetros de entrada, como a energia disponível ou o número de portais, a resposta pode mudar. Isso ocorre porque diferentes restrições podem influenciar as rotas possíveis, alterando a solução final. Por exemplo, ao aumentar o número de portais disponíveis, os algoritmos podem utilizar mais caminhos de custo zero, alterando o trajeto final escolhido.

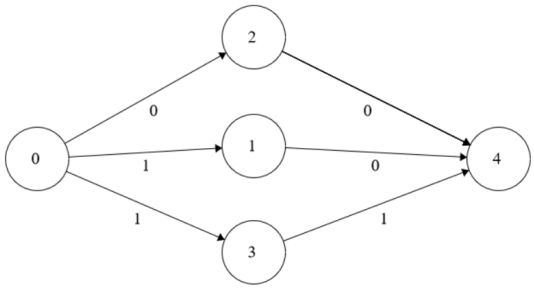

Figura 3: Exemplo

Por exemplo, no caso acima, em uma tentativa com 1 de energia e 1 portal usavel existe a solução ***0 - 1 - 4***, caso a energia seja reduzida a 0, já não existe mais solução, mas se os portais forem aumentados para dois a solução se torna ***0 - 2 - 4***. Além disso, se a energia for 2 e portais 0, existe a solução ***0 - 3 - 4***.

7 Conclusão

Neste trabalho, abordamos a problemática da busca pelo caminho mínimo em um grafo direcionado que representa uma floresta. A implementação de algoritmos clássicos de busca em grafos, como Dijkstra e A*, foi crucial para resolver o problema de encontrar uma rota viável para o herói Linque escapar da floresta, considerando as restrições de energia e o uso limitado de portais mágicos.

Os experimentos realizados demonstraram a eficácia das abordagens implementadas. Em particular, observou-se que a matriz de adjacência tende a ser mais eficiente em grafos densos devido à sua alocação de memória fixa, enquanto a lista de adjacência é mais adequada para grafos esparsos, devido ao uso de memória proporcional ao número de arestas. A análise experimental validou a consistência entre a complexidade teórica e os tempos de execução observados, reforçando a validade das implementações.

Foi identificado que, embora o algoritmo de Dijkstra sempre encontre o menor caminho, o algoritmo A* pode não garantir a solução ótima se a heurística utilizada não for admissível. A escolha da heurística é, portanto, um fator crucial para o desempenho e a precisão do A*. No contexto do problema específico deste trabalho, a heurística baseada na distância euclidiana, embora intuitiva, não garantiu sempre a solução ótima devido à presença de portais de custo zero.

Além das implementações e análises, estratégias de robustez foram empregadas para garantir a confiabilidade e a estabilidade do sistema. A verificação de limites de entrada, a alocação segura de memória, a manipulação de exceções e a validação de dados de entrada foram práticas essenciais para minimizar falhas e comportamentos inesperados.

Em resumo, este trabalho proporcionou uma compreensão aprofundada das técnicas de busca em grafos e suas aplicações práticas. As abordagens adotadas e as análises realizadas oferecem uma base sólida para o desenvolvimento de soluções eficientes para problemas semelhantes em grafos. As dificuldades enfrentadas e as soluções encontradas ao longo do projeto reforçam a importância de uma abordagem cuidadosa e meticulosa na implementação e otimização de algoritmos complexos.

Os resultados obtidos destacam a importância da escolha apropriada de estruturas de dados e heurísticas, dependendo das características específicas do problema. Futuras melhorias podem incluir a exploração de heurísticas mais sofisticadas e

a adaptação dos algoritmos para diferentes tipos de grafos e restrições adicionais, aumentando ainda mais a eficiência e a aplicabilidade das soluções desenvolvidas.

8 Referencias

Wagner Meira. (2024). Slides Virtuais de Estrutura de dados

https://www.youtube.com/watch?v=Kue4UUxstoUab$_channel = MaratonaUFMG$

https://pt.wikipedia.org/wiki/Algoritmo$_A*$

https://www.w3schools.com/